Cálculo de Programas
Trabalho Prático
MiEI+LCC — 2020/21

Departamento de Informática
Universidade do Minho

Junho de 2021

Grupo nr.	79
a85829	Meriem Khammassi
a89482	Simão Paulo da Gama Castel-Branco e Brito
a89578	Patrícia Gonçalves Pereira

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`[1] 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-processador que faz "pretty printing" de código Haskell em LaTeX 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
```

[1] 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 depêndencias 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,

$$
\begin{aligned}
&\mathbf{data}\ \mathit{ExpAr}\ a = X\\
&\quad \mid N\ a\\
&\quad \mid \mathit{Bin}\ \mathit{BinOp}\ (\mathit{ExpAr}\ a)\ (\mathit{ExpAr}\ a)\\
&\quad \mid \mathit{Un}\ \mathit{UnOp}\ (\mathit{ExpAr}\ a)\\
&\quad \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathbf{data}\ \mathit{BinOp} = \mathit{Sum}\\
&\quad \mid \mathit{Product}\\
&\quad \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show})\\
&\mathbf{data}\ \mathit{UnOp} = \mathit{Negate}\\
&\quad \mid E\\
&\quad \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{aligned}
$$

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,

$$\mathit{Bin}\ \mathit{Sum}\ X\ (N\ 10)$$

designa $x + 10$ na notação matemática habitual.

1. A definição das funções $\mathit{inExpAr}$ e $\mathit{baseExpAr}$ para este tipo é a seguinte:

$$
\begin{aligned}
&\mathit{inExpAr} = [\underline{X}, \mathit{num\_ops}]\ \mathbf{where}\\
&\quad \mathit{num\_ops} = [N, \mathit{ops}]\\
&\quad \mathit{ops} = [\mathit{bin}, \widehat{\mathit{Un}}]\\
&\quad \mathit{bin}\ (\mathit{op}, (a, b)) = \mathit{Bin}\ \mathit{op}\ a\ b\\
&\mathit{baseExpAr}\ f\ g\ h\ j\ k\ l\ z = f + (g + (h \times (j \times k) + l \times z))
\end{aligned}
$$

Defina as funções $\mathit{outExpAr}$ e $\mathit{recExpAr}$, e teste as propriedades que se seguem.

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

$$
\begin{aligned}
&\mathit{prop\_in\_out\_idExpAr} :: (\mathit{Eq}\ a) \Rightarrow \mathit{ExpAr}\ a \to \mathit{Bool}\\
&\mathit{prop\_in\_out\_idExpAr} = \mathit{inExpAr} \cdot \mathit{outExpAr} \equiv \mathit{id}\\
&\mathit{prop\_out\_in\_idExpAr} :: (\mathit{Eq}\ a) \Rightarrow \mathit{OutExpAr}\ a \to \mathit{Bool}\\
&\mathit{prop\_out\_in\_idExpAr} = \mathit{outExpAr} \cdot \mathit{inExpAr} \equiv \mathit{id}
\end{aligned}
$$

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 13 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.*

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

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

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

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

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

que se encontra na página 13 expressa como um hilomorfismo[2] e teste as propriedades:

Propriedade [QuickCheck] 4 *A função $optimize\_eval$ respeita a semântica da função $eval$.*

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

4. Para calcular a derivada de uma expressão, é necessário aplicar transformações à expressão original que respeitem as regras das derivadas:[3]

- Regra da soma:

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

[2]Qual é a vantagem de implementar a função $optimize\_eval$ utilizando um hilomorfismo em vez de utilizar um catamorfismo com um gene "inteligente"?

[3]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.*

$$
\begin{array}{l}
prop\_const\_rule :: (Real\ a, Floating\ a) \Rightarrow a \rightarrow Bool \\
prop\_const\_rule\ a = sd\ (N\ a) \equiv N\ 0 \\
prop\_var\_rule :: Bool \\
prop\_var\_rule = sd\ X \equiv N\ 1 \\
prop\_sum\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow ExpAr\ a \rightarrow Bool \\
prop\_sum\_rule\ exp1\ exp2 = sd\ (Bin\ Sum\ exp1\ exp2) \equiv sum\_rule\ \mathbf{where} \\
\quad sum\_rule = Bin\ Sum\ (sd\ exp1)\ (sd\ exp2) \\
prop\_product\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow ExpAr\ a \rightarrow Bool \\
prop\_product\_rule\ exp1\ exp2 = sd\ (Bin\ Product\ exp1\ exp2) \equiv prod\_rule\ \mathbf{where} \\
\quad prod\_rule = Bin\ Sum\ (Bin\ Product\ exp1\ (sd\ exp2))\ (Bin\ Product\ (sd\ exp1)\ exp2) \\
prop\_e\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow Bool \\
prop\_e\_rule\ exp = sd\ (Un\ E\ exp) \equiv Bin\ Product\ (Un\ E\ exp)\ (sd\ exp) \\
prop\_negate\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow Bool \\
prop\_negate\_rule\ exp = sd\ (Un\ Negate\ exp) \equiv Un\ Negate\ (sd\ exp)
\end{array}
$$

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 cálculando 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.*

   $$
   \begin{array}{l}
   prop\_congruent :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\
   prop\_congruent\ a\ exp = ad\ a\ exp \stackrel{?}{=} eval\_exp\ a\ (sd\ exp)
   \end{array}
   $$

Problema 2

Nesta disciplina estudou-se como fazer programação dinâmica por cálculo, recorrendo à lei de recursividade mútua.[4]

Para o caso de funções sobre os números naturais ($\mathbb{N}_0$, com functor $\mathsf{F}\ X = 1 + X$) 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

$$
\begin{array}{l}
fib\ 0 = 1 \\
fib\ (n+1) = f\ n
\end{array}
$$

[4] Lei (3.94) em [2], página 98.

$$f\ 0 = 1$$
$$f\ (n+1) = \mathit{fib}\ n + f\ n$$

Obter-se-á de imediato

$$\begin{array}{l} \mathit{fib}' = \pi_1 \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where} \\ \quad \mathit{loop}\ (\mathit{fib}, f) = (f, \mathit{fib} + f) \\ \quad \mathit{init} = (1, 1) \end{array}$$

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.[5]
- Para os resultados vão-se buscar as expressões respectivas, retirando a variável n.
- Em init coleccionam-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[6], de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

$$\begin{array}{l} f\ 0 = c \\ f\ (n+1) = f\ n + k\ n \end{array}$$
$$\begin{array}{l} k\ 0 = a + b \\ k\ (n+1) = k\ n + 2\ a \end{array}$$

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

$$\begin{array}{l} f'\ a\ b\ c = \pi_1 \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where} \\ \quad \mathit{loop}\ (f, k) = (f + k, k + 2 * a) \\ \quad \mathit{init} = (c, a + b) \end{array}$$

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

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

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

$$\mathit{cat} = \cdots \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where}\ \cdots$$

que implemente esta função.

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

$$\mathit{prop\_cat} = (\geqslant 0) \Rightarrow (\mathit{catdef} \equiv \mathit{cat})$$

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 $\{P_0, ..., P_N\}$ 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.

[5] Podem obviamente usar-se outros símbolos, mas numa primeira leitura dá jeito usarem-se tais nomes.
[6] Secção 3.17 de [2] e tópico Recursividade mútua nos vídeos das aulas teóricas.

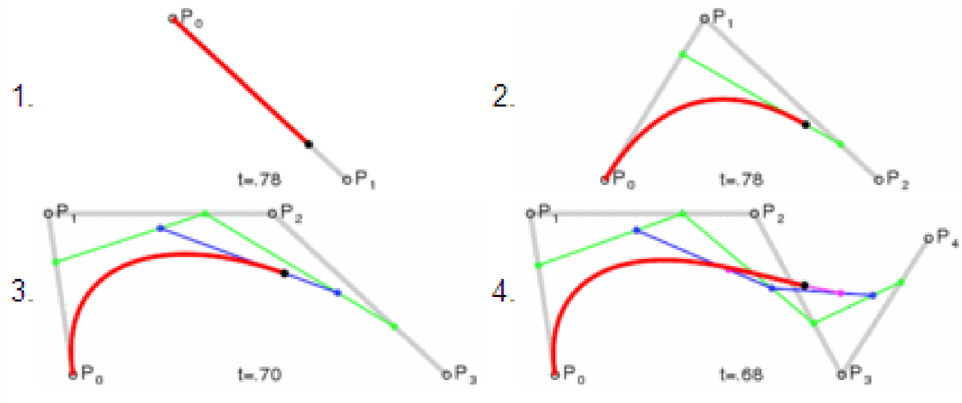

Figura 1: Exemplos de curvas de Bézier retirados da Wikipedia.

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:

$$
\begin{array}{l}
\mathit{linear1d} :: \mathbb{Q} \to \mathbb{Q} \to \mathit{OverTime}\ \mathbb{Q} \\
\mathit{linear1d}\ a\ b = \mathit{formula}\ a\ b\ \mathbf{where} \\
\quad \mathit{formula} :: \mathbb{Q} \to \mathbb{Q} \to \mathit{Float} \to \mathbb{Q} \\
\quad \mathit{formula}\ x\ y\ t = ((1.0 :: \mathbb{Q}) - (\mathit{to}_{\mathbb{Q}}\ t)) * x + (\mathit{to}_{\mathbb{Q}}\ t) * y
\end{array}
$$

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.

$$\mathbf{type}\ \mathit{NPoint} = [\mathbb{Q}]$$

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

$$
\begin{array}{l}
\mathit{p2d} = [1.2, 3.4] \\
\mathit{p3d} = [0.2, 10.3, 2.4]
\end{array}
$$

O tipo de dados $\mathit{OverTime}\ a$ representa um termo do tipo a num dado instante (dado por um Float).

$$\mathbf{type}\ \mathit{OverTime}\ a = \mathit{Float} \to a$$

O anexo C tem definida a função

$$\mathit{calcLine} :: \mathit{NPoint} \to (\mathit{NPoint} \to \mathit{OverTime}\ \mathit{NPoint})$$

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

$$\mathit{deCasteljau} :: [\mathit{NPoint}] \to \mathit{OverTime}\ \mathit{NPoint}$$

que implementa o algoritmo respectivo.

1. Implemente $\mathit{calcLine}$ como um catamorfismo de listas, testando a sua definição com a propriedade:

   Propriedade [QuickCheck] 8 *Definição alternativa.*

   $$
   \begin{array}{l}
   \mathit{prop\_calcLine\_def} :: \mathit{NPoint} \to \mathit{NPoint} \to \mathit{Float} \to \mathit{Bool} \\
   \mathit{prop\_calcLine\_def}\ p\ q\ d = \mathit{calcLine}\ p\ q\ d \equiv \mathit{zipWithM}\ \mathit{linear1d}\ p\ q\ d
   \end{array}
   $$

2. Implemente a função $\mathit{deCasteljau}$ como um hilomorfismo, testando agora a propriedade:

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

$$
\begin{array}{l}
prop\_bezier\_sym :: [[\mathbb{Q}]] \rightarrow Gen\ Bool \\
prop\_bezier\_sym\ l = all\ (<\Delta) \cdot calc\_difs \cdot bezs\ \langle\$\rangle\ elements\ ps\ \mathbf{where} \\
\quad calc\_difs = (\lambda(x,y) \rightarrow zipWith\ (\lambda w\ v \rightarrow \mathbf{if}\ w \geqslant v\ \mathbf{then}\ w - v\ \mathbf{else}\ v - w)\ x\ y) \\
\quad bezs\ t = (deCasteljau\ l\ t, deCasteljau\ (reverse\ l)\ (from_{\mathbb{Q}}\ (1 - (to_{\mathbb{Q}}\ t)))) \\
\quad \Delta = 1\mathrm{e}{-2}
\end{array}
$$

3. Corra a função $runBezier$ e aprecie o seu trabalho[7] clicando na janela que é aberta (que contém, a verde, um ponto inicila) 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,

$$avg\ x = \frac{1}{k}\sum_{i=1}^{k} x_i \tag{2}$$

onde $k = length\ x$. 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 facil de ver que

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

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.

1. Recorra à lei de recursividade mútua para derivar a função $avg\_aux = (\![[b, q]]\!)$ tal que $avg\_aux = \langle avg, length\rangle$ em listas não vazias.
2. 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:*

$$
\begin{array}{l}
prop\_avg :: [Double] \rightarrow Property \\
prop\_avg = nonempty \Rightarrow diff \leqslant \underline{0.000001}\ \mathbf{where} \\
\quad diff\ l = avg\ l - (avgLTree \cdot genLTree)\ l \\
\quad genLTree = [\![lsplit]\!] \\
\quad nonempty = (>[\,])
\end{array}
$$

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.

[7] 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:[8]

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

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

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

B Programação dinâmica por recursividade múltipla

Neste anexo dão-se os detalhes da resolução do Exercício 3.30 dos apontamentos da disciplina[9], 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:

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

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:

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

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

$$
\begin{array}{l}
e'\ x = prj \cdot \mathsf{for}\ loop\ init\ \mathbf{where} \\
\quad init = (1, x, 2) \\
\quad loop\ (e, h, s) = (h + e, x \mathbin{/} s * h, 1 + s) \\
\quad prj\ (e, h, s) = e
\end{array}
$$

[8] Exemplos tirados de [2].
[9] Cf. [2], página 102.

C Código fornecido

Problema 1

$$
\begin{aligned}
&expd :: Floating\ a \Rightarrow a \rightarrow a\\
&expd = Prelude.exp\\
&\mathbf{type}\ OutExpAr\ a = () + (a + ((BinOp, (ExpAr\ a, ExpAr\ a)) + (UnOp, ExpAr\ a)))
\end{aligned}
$$

Problema 2

Definição da série de Catalan usando factoriais (6):

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

Oráculo para inspecção dos primeiros 26 números de Catalan[10]:

$$
\begin{aligned}
&oracle = [\\
&\quad 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845,\\
&\quad 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020,\\
&\quad 91482563640, 343059613650, 1289904147324, 4861946401452\\
&\quad]
\end{aligned}
$$

Problema 3

Algoritmo:

$$
\begin{aligned}
&deCasteljau :: [NPoint] \rightarrow OverTime\ NPoint\\
&deCasteljau\ [\,] = nil\\
&deCasteljau\ [p] = \underline{p}\\
&deCasteljau\ l = \lambda pt \rightarrow (calcLine\ (p\ pt)\ (q\ pt))\ pt\ \mathbf{where}\\
&\quad p = deCasteljau\ (init\ l)\\
&\quad q = deCasteljau\ (tail\ l)
\end{aligned}
$$

Função auxiliar:

$$
\begin{aligned}
&calcLine :: NPoint \rightarrow (NPoint \rightarrow OverTime\ NPoint)\\
&calcLine\ [\,] = \underline{nil}\\
&calcLine\ (p : x) = \overline{g}\ p\ (calcLine\ x)\ \mathbf{where}\\
&\quad g :: (\mathbb{Q}, NPoint \rightarrow OverTime\ NPoint) \rightarrow (NPoint \rightarrow OverTime\ NPoint)\\
&\quad g\ (d, f)\ l = \mathbf{case}\ l\ \mathbf{of}\\
&\qquad [\,] \rightarrow nil\\
&\qquad (x : xs) \rightarrow \lambda z \rightarrow concat\ \$\ (sequenceA\ [singl \cdot linear1d\ d\ x, f\ xs])\ z
\end{aligned}
$$

2D:

$$
\begin{aligned}
&bezier2d :: [NPoint] \rightarrow OverTime\ (Float, Float)\\
&bezier2d\ [\,] = \underline{(0, 0)}\\
&bezier2d\ l = \lambda z \rightarrow (from_{\mathbb{Q}} \times from_{\mathbb{Q}}) \cdot (\lambda[x, y] \rightarrow (x, y))\ \$\ ((deCasteljau\ l)\ z)
\end{aligned}
$$

Modelo:

$$
\begin{aligned}
&\mathbf{data}\ World = World\ \{points :: [NPoint]\\
&\quad , time :: Float\\
&\quad \}\\
&initW :: World\\
&initW = World\ [\,]\ 0
\end{aligned}
$$

[10]Fonte: Wikipedia.

$$
\begin{array}{l}
tick :: Float \to World \to World\\
tick\ dt\ world = world\ \{ time = (time\ world) + dt \}
\end{array}
$$

$$
\begin{array}{l}
actions :: Event \to World \to World\\
actions\ (EventKey\ (MouseButton\ LeftButton)\ Down\ \_\ p)\ world =\\
\quad world\ \{ points = (points\ world) +\!\!+ [(\lambda(x,y) \to \mathsf{map}\ \ to_{\mathbb{Q}}\ [x,y])\ p] \}\\
actions\ (EventKey\ (SpecialKey\ KeyDelete)\ Down\ \_\ \_)\ world =\\
\quad world\ \{ points = cond\ (\equiv [\,])\ id\ init\ (points\ world) \}\\
actions\ \_\ world = world
\end{array}
$$

$$
\begin{array}{l}
scaleTime :: World \to Float\\
scaleTime\ w = (1 + cos\ (time\ w))\ /\ 2
\end{array}
$$

$$
\begin{array}{l}
bezier2dAtTime :: World \to (Float, Float)\\
bezier2dAtTime\ w = (bezier2dAt\ w)\ (scaleTime\ w)
\end{array}
$$

$$
\begin{array}{l}
bezier2dAt :: World \to OverTime\ (Float, Float)\\
bezier2dAt\ w = bezier2d\ (points\ w)
\end{array}
$$

$$
\begin{array}{l}
thicCirc :: Picture\\
thicCirc = ThickCircle\ 4\ 10
\end{array}
$$

$$
\begin{array}{l}
ps :: [Float]\\
ps = \mathsf{map}\ \ from_{\mathbb{Q}}\ ps'\ \mathbf{where}\\
\quad ps' :: [\mathbb{Q}]\\
\quad ps' = [0, 0.01 \mathinner{..} 1] \quad \text{-- interval}
\end{array}
$$

Gloss:

$$
\begin{array}{l}
picture :: World \to Picture\\
picture\ world = Pictures\\
\quad [animateBezier\ (scaleTime\ world)\ (points\ world)\\
\quad , Color\ white \cdot Line \cdot \mathsf{map}\ \ (bezier2dAt\ world)\ \$\ ps\\
\quad , Color\ blue \cdot Pictures\ \$\ [Translate\ (from_{\mathbb{Q}}\ x)\ (from_{\mathbb{Q}}\ y)\ thicCirc \mid [x,y] \leftarrow points\ world]\\
\quad , Color\ green\ \$\ Translate\ cx\ cy\ thicCirc\\
\quad]\ \mathbf{where}\\
\quad (cx, cy) = bezier2dAtTime\ world
\end{array}
$$

Animação:

$$
\begin{array}{l}
animateBezier :: Float \to [NPoint] \to Picture\\
animateBezier\ \_\ [\,] = Blank\\
animateBezier\ \_\ [\_] = Blank\\
animateBezier\ t\ l = Pictures\\
\quad [animateBezier\ t\ (init\ l)\\
\quad , animateBezier\ t\ (tail\ l)\\
\quad , Color\ red \cdot Line\ \$\ [a, b]\\
\quad , Color\ orange\ \$\ Translate\ ax\ ay\ thicCirc\\
\quad , Color\ orange\ \$\ Translate\ bx\ by\ thicCirc\\
\quad]\ \mathbf{where}\\
\quad a@(ax, ay) = bezier2d\ (init\ l)\ t\\
\quad b@(bx, by) = bezier2d\ (tail\ l)\ t
\end{array}
$$

Propriedades e *main*:

$$
\begin{array}{l}
runBezier :: \mathsf{IO}\ ()\\
runBezier = play\ (InWindow\ \texttt{"Bézier"}\ (600, 600)\ (0, 0))\\
\quad black\ 50\ initW\ picture\ actions\ tick
\end{array}
$$

$$
\begin{array}{l}
runBezierSym :: \mathsf{IO}\ ()\\
runBezierSym = quickCheckWith\ (stdArgs\ \{ maxSize = 20, maxSuccess = 200 \})\ prop\_bezier\_sym
\end{array}
$$

Compilação e execução dentro do interpretador:[11]

$$main = runBezier$$

$$run = \mathbf{do}\ \{ system\ \texttt{"ghc cp2021t"}; system\ \texttt{"./cp2021t"} \}$$

[11] 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:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Arbitrary}\ \mathit{UnOp}\ \textbf{where} \\
\quad \mathit{arbitrary} = \mathit{elements}\ [\mathit{Negate}, E] \\
\textbf{instance}\ \mathit{Arbitrary}\ \mathit{BinOp}\ \textbf{where} \\
\quad \mathit{arbitrary} = \mathit{elements}\ [\mathit{Sum}, \mathit{Product}] \\
\textbf{instance}\ (\mathit{Arbitrary}\ a) \Rightarrow \mathit{Arbitrary}\ (\mathit{ExpAr}\ a)\ \textbf{where} \\
\quad \mathit{arbitrary} = \textbf{do} \\
\quad\quad \mathit{binop} \leftarrow \mathit{arbitrary} \\
\quad\quad \mathit{unop} \leftarrow \mathit{arbitrary} \\
\quad\quad \mathit{exp1} \leftarrow \mathit{arbitrary} \\
\quad\quad \mathit{exp2} \leftarrow \mathit{arbitrary} \\
\quad\quad a \quad\ \leftarrow \mathit{arbitrary} \\
\quad\quad \mathit{frequency} \cdot \mathsf{map}\ (\mathit{id} \times \mathit{pure})\ \$\ [(20, X), (15, N\ a), (35, \mathit{Bin}\ \mathit{binop}\ \mathit{exp1}\ \mathit{exp2}), (30, \mathit{Un}\ \mathit{unop}\ \mathit{exp1})] \\
\textbf{infixr}\ 5 \stackrel{?}{=} \\
(\stackrel{?}{=}) :: \mathit{Real}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Bool} \\
(\stackrel{?}{=})\ x\ y = (\mathit{to}_{\mathbb{Q}}\ x) \equiv (\mathit{to}_{\mathbb{Q}}\ y)
\end{array}
$$

Outras funções auxiliares

Lógicas:

$$
\begin{array}{l}
\textbf{infixr}\ 0 \Rightarrow \\
(\Rightarrow) :: (\mathit{Testable}\ \mathit{prop}) \Rightarrow (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{prop}) \rightarrow a \rightarrow \mathit{Property} \\
p \Rightarrow f = \lambda a \rightarrow p\ a \Rightarrow f\ a \\
\textbf{infixr}\ 0 \Leftrightarrow \\
(\Leftrightarrow) :: (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{Bool}) \rightarrow a \rightarrow \mathit{Property} \\
p \Leftrightarrow f = \lambda a \rightarrow (p\ a \Rightarrow \mathit{property}\ (f\ a))\ .\&\&.\ (f\ a \Rightarrow \mathit{property}\ (p\ a)) \\
\textbf{infixr}\ 4 \equiv \\
(\equiv) :: \mathit{Eq}\ b \Rightarrow (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow \mathit{Bool}) \\
f \equiv g = \lambda a \rightarrow f\ a \equiv g\ a \\
\textbf{infixr}\ 4 \leqslant \\
(\leqslant) :: \mathit{Ord}\ b \Rightarrow (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow \mathit{Bool}) \\
f \leqslant g = \lambda a \rightarrow f\ a \leqslant g\ a \\
\textbf{infixr}\ 4 \land \\
(\land) :: (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{Bool}) \\
f \land g = \lambda a \rightarrow ((f\ a) \land (g\ a))
\end{array}
$$

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, disgramas 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:

$$
\begin{aligned}
&cataExpAr\ g = g \cdot recExpAr\ (cataExpAr\ g) \cdot outExpAr \\
&anaExpAr\ g = inExpAr \cdot recExpAr\ (anaExpAr\ g) \cdot g \\
&hyloExpAr\ h\ g = cataExpAr\ h \cdot anaExpAr\ g \\
&eval\_exp :: Floating\ a \Rightarrow a \to (ExpAr\ a) \to a \\
&eval\_exp\ a = cataExpAr\ (g\_eval\_exp\ a) \\
&optmize\_eval :: (Floating\ a, Eq\ a) \Rightarrow a \to (ExpAr\ a) \to a \\
&optmize\_eval\ a = hyloExpAr\ (gopt\ a)\ clean \\
&sd :: Floating\ a \Rightarrow ExpAr\ a \to ExpAr\ a \\
&sd = \pi_2 \cdot cataExpAr\ sd\_gen \\
&ad :: Floating\ a \Rightarrow a \to ExpAr\ a \to a \\
&ad\ v = \pi_2 \cdot cataExpAr\ (ad\_gen\ v)
\end{aligned}
$$

outExpAr

$$
\begin{aligned}
& outExpAr\ .\ inExpAr = id \\
\equiv \quad & \{\ \text{inExpAr}\ \} \\
& outExpAr\ .\ [\underline{X},\ num\_ops] = id \\
\equiv \quad & \{\ \text{Fusão-+ (20)}\ \} \\
& [outExpAr\ .\ \underline{X},\ outExpAr\ .\ num\_ops] = id \\
\equiv \quad & \{\ \text{Universal-+ (17)}\ \} \\
& \left\{ \begin{array}{l} id \cdot i_1 = outExpAr \cdot \underline{X} \\ id \cdot i_2 = outExpAr \cdot num\_ops \end{array} \right. \\
\equiv \quad & \{\ \text{Natural-id (1), Igualdade Extensional (69), Def-comp (70)}\ \} \\
& \left\{ \begin{array}{l} outExpAr\ X = i_1\ () \\ outExpAr\ (N\ a) = i_2\ (i_1\ a) \end{array} \right. \\
& \left\{ \begin{array}{l} outExpAr\ (Bin\ op\ a\ b) = i_2\ (i_2\ (i_1\ (op, (a, b)))) \\ outExpAr\ (Un\ op\ a) = i_2\ (i_2\ (i_2\ (op, a))) \end{array} \right.
\end{aligned}
$$

recExpAr

$$
\begin{tikzcd}
ExpAr\ a \arrow[rr, bend left, "outExpAr"] \arrow[d, "f"'] \arrow[r, phantom, "\cong"] & & 1 + (a + (BinOp \times (ExpAr\ a \times ExpAr\ a) + (UnOp \times ExpAr\ a))) \arrow[ll, bend left, "inExpAr"] \arrow[d, "id+(id+(id \times (f \times f)+(id \times f)))"] \\
C & & 1 + (a + ((BinOp \times (C \times C)) + (UnOp \times C))) \arrow[ll]
\end{tikzcd}
$$

$$
\begin{aligned}
& recExpr\ f = id + (id + (id \times (f \times f) + id \times f)) \\
\equiv \quad & \{ \text{Aplicando a definição dada de baseExpAr} \} \\
& recExpr\ f = baseExpAr\ id\ id\ id\ f\ f\ id\ f
\end{aligned}
$$

g_eval_exp

De forma a obter o valo final de uma expressão Aritmética, podemos recorrer ao uso do catamorfismo de *ExprAr*.

$$
\begin{aligned}
& eval\_exp\ a = cataExpAr\ (\ g\_eval\_exp\ a\ ) \\
\equiv \quad & \{ \text{Aplicando a definição dada de cataExpAr} \} \\
& eval\_exp\ a = (\ g\_eval\_exp\ a\ )\ .\ recExpAr\ (\ cataExprAr\ (\ g\_eval\_exp\ a\ )\ )\ .\ outExprAr
\end{aligned}
$$

Definindo o diagrama de *eval_exp*, podemos obter a expressão pretendida *g_eval_exp*.

$$
\begin{tikzcd}
ExpAr\ a \arrow[rr, bend left, "outExpAr"] \arrow[d, "(\!| g\_eval\_exp\ \ a |\!)"'] \arrow[r, phantom, "\cong"] & & 1 + (a + ((BinOp \times (ExpAr\ a \times ExpAr\ a)) + (UnOp \times ExpAr\ a))) \arrow[d, "recExpAr\ (\!| g\_eval\_exp\ \ a |\!)"] \\
a & & 1 + (a + ((BinOp \times (a \times a)) + (UnOp \times a))) \arrow[ll, "g\_eval\_exp\ \ a"]
\end{tikzcd}
$$

Neste caso, percorremos todas as expressões, de modo a determinar o valor de cada uma, aplicando posteriormente a cada par de valores a operaçõa determinada pelo *BinOp*, emparelhando inicialmente, com cada par de expressões. Ou seja, o gene de *eval_exp* pode ser representado pelo seguinte diagrama:

$$
\begin{tikzcd}
1 + (a + (BinOp \times (a \times a) + (UnOp \times a))) \arrow[d, "(calculaX\ a) + (id + binOpAux + unOpAux)"'] \\
a + (a + a + a) \arrow[d, "{[\ id, [\ id, [\ id, id\ ]\ ]\ ]}"'] \\
a
\end{tikzcd}
$$

Onde *binOpAux* é a função que dada uma operação *Sum* ou *Product* e dois *float* realiza a operação (+) ou (*). A função *unOpAux* é a função que dada um *Negate* ou um *E* e um *Floating a* devolve o simétrico ou o mesmo número. O *()* é substituiso pelo escalar *a* aplicando a função *calculaX a*. Assim sendo:

$$
\begin{aligned}
& g\_eval\_exp\ a = [\ id, [\ id, [id, id]\ ]\ ] \cdot ((calculaX\ a)\ + (id + binOpAux + unOpAux)) && (4)\\
\equiv \quad & \quad \{ \text{ Absorção-+ (22) ; Natural-id (1) } \} \\
& g\_eval\_exp\ a = [calculaX\ a, [id, [binOpAux, unOpAux]\ ]\ ] && (5)
\end{aligned}
$$

clean e gopt

Com o objetivo de otimizar o cálculo do valor de uma expressão Aritmética, tirando proveito dos elementos absorventes das operações, recorremos ao uso do hilomorfismo de *ExprAr*.

$$
\begin{aligned}
& optmize\_eval\ a = hyloExpAr\ (\ gopt\ a\ )\ clean \\
\equiv \quad & \quad \{ \text{ Aplicando a definição dada de hyloExpAr } \} \\
& optmize\_eval\ a = cataExpAr(gopt\ a)\ .\ anaExpAr(clean) \\
\equiv \quad & \quad \{ \text{ Cancelamento-cata (44); Cancelamento-ana(53) } \} \\
& optmize\_eval\ a = (gopt\ a).F(\!|gopt\ a|\!).inExpr.outExpr.F[\!(clean)\!].clean \\
\equiv \quad & \quad \{ \text{ in.out = id; Aplicando as definições em Haskell já determinadas } \} \\
& optmize\_eval\ a = (gopt\ a).recExpAr(cataExpAr(gopt\ a)).recExpAr(anaExpArclean).clean
\end{aligned}
$$

De seguida definimos os diagramas de *cataExpAr(gopt a)* e de *anaExpAr(clean)* de forma a obter os genes *clean* e *gopt a*:

$$
\begin{tikzcd}
ExpAr\ a \arrow[d, "anaExpAr\ (clean)"'] \arrow[r, bend left, "clean"] & 1 + (a + ((BinOp \times (ExpAr\ a \times ExpAr\ a)) + (UnOp \times ExpAr\ a))) \arrow[d, "recExpAr\ (anaExpAr\ (clean))"] \\
ExpAr\ a & 1 + (a + ((BinOp \times (ExpAr\ a \times ExpAr\ a)) + (UnOp \times ExpAr\ a))) \arrow[l, "inExpAr"]
\end{tikzcd}
$$

$$
\begin{tikzcd}
ExpAr\ a \arrow[d, "(\!|gopt\ a|\!)"'] \arrow[r, phantom, "\cong"] \arrow[r, bend left, "outExpAr"] & 1 + (a + ((BinOp \times (ExpAr\ a \times ExpAr\ a)) + (UnOp \times ExpAr\ a))) \arrow[d, "recExpAr\ (\!|gopt\ a|\!)"] \\
a & 1 + (a + ((BinOp \times (a \times a)) + (UnOp \times a))) \arrow[l, "gopt\ a"]
\end{tikzcd}
$$

Deste modo chegamos às expressões dos genes *clean* e *gopt a*, presentes na area *Soluções* deste Problema. Estes têm o propósito de eliminar as somas ou os produtos onde se possa aplicar as propriedades absorventes destas, de forma a otimizar o cálculo das expressões aritméticas.

sd_gen e ad_gen

$$
\begin{aligned}
& sd = \pi 2.\ cataExpAr\ sd\_gen \\
\equiv & \qquad \{ \text{Aplicando a definição dada de cataExpAr} \} \\
& sd = \pi 2.\ sd\_gen\ .\ recExpAr\ (\ cataExprAr\ sd\_gen\ )\ .\ outExprAr
\end{aligned}
$$

sd_gen é o gene da função *sd* que calcula a derivada de uma função. O gene tem como resultado (ExpAr a,ExpAr a), sendo o primeiro elemento do par a expressão que pretendemos derivar e o segundo a sua derivada.

$$
\begin{aligned}
& ad = \pi 2.\ cataExpAr\ (\ ad\_gen\ v\ ) \\
\equiv & \qquad \{ \text{Aplicando a definição dada de cataExpAr} \} \\
& ad = \pi 2.\ (\ ad\_gen\ v\ ).\ recExpAr\ (\ cataExprAr\ (\ ad\_gen\ v\ ))\ .\ outExprAr
\end{aligned}
$$

ad_gen é o gene da função *ad* que calcula a derivada num ponto de uma função. O gene tem como resultado (a,a), sendo o primeiro elemento do par a solução da funçao no ponto e o segundo a derivada no mesmo. As soluções dos dois encontram-se abaixo, na área soluções.

Solução:

$$
\begin{array}{l}
outExpAr\ X = i_1\ ()\\
outExpAr\ (N\ a) = i_2\ (i_1\ a)\\
outExpAr\ (Bin\ op\ a\ b) = i_2\ (i_2\ (i_1\ (op, (a, b))))\\
outExpAr\ (Un\ op\ a) = i_2\ (i_2\ (i_2\ (op, a)))\\
\quad\text{--}\\
recExpAr\ f = baseExpAr\ id\ id\ id\ f\ f\ id\ f\\
\quad\text{--}\\
g\_eval\_exp\ a = [calculaX\ a, [id, [binOpAux, unOpAux]]]\\
calculaX\ a\ () = a\\
binOpAux\ (Sum, (a, b)) = (+)\ a\ b\\
binOpAux\ (Product, (a, b)) = (*)\ a\ b\\
unOpAux\ (Negate, a) = -a\\
unOpAux\ (E, a) = (expd\ a)\\
\quad\text{--}\\
clean\ (Bin\ Product\ (N\ 0)\ a) = i_2\ (i_1\ 0)\\
clean\ (Bin\ Product\ a\ (N\ 0)) = i_2\ (i_1\ 0)\\
clean\ (Bin\ Sum\ (N\ 0)\ a) = clean\ a\\
clean\ (Bin\ Sum\ a\ (N\ 0)) = clean\ a\\
clean\ (Bin\ Product\ (N\ 1)\ a) = clean\ a\\
clean\ (Bin\ Product\ a\ (N\ 1)) = clean\ a\\
clean\ (Un\ E\ (N\ 0)) = i_2\ (i_1\ 1)\\
clean\ a = outExpAr\ a\\
\quad\text{--}\\
gopt\ a = [calculaX\ a, [id, [goptAux, unOpAux]]]\\
goptAux\ (Sum, (0, a)) = a\\
goptAux\ (Sum, (a, 0)) = a\\
goptAux\ (Product, (1, a)) = a\\
goptAux\ (Product, (a, 1)) = a\\
goptAux\ a = binOpAux\ a\\
\quad\text{--}\\
sd\_gen :: Floating\ a \Rightarrow\\
\quad () + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + (UnOp, (ExpAr\ a, ExpAr\ a)))) \to (ExpAr\ a\\
sd\_gen = [sdAuxX, [sdAuxN, [sdAuxB, sdAuxU]]]\\
sdAuxX\ () = (X, N\ 1)\\
sdAuxN\ a = (N\ a, N\ 0)\\
sdAuxB\ (Sum, ((a, b), (c, d))) = (Bin\ Sum\ a\ c, Bin\ Sum\ b\ d)\\
sdAuxB\ (Product, ((a, b), (c, d))) = (Bin\ Product\ a\ c, Bin\ Sum\ (Bin\ Product\ a\ d)\ (Bin\ Product\ b\ c))\\
sdAuxU\ (Negate, (a, b)) = (Un\ Negate\ a, Un\ Negate\ b)\\
sdAuxU\ (E, (a, b)) = (Un\ E\ a, Bin\ Product\ (Un\ E\ a)\ b)
\end{array}
$$

$$
\begin{array}{l}
ad\_gen\ v = [adAuxX\ v, [adAuxN, [adAuxB, adAuxU]]]\\
adAuxX\ v\ () = (v, 1)\\
adAuxN\ a = (a, 0)\\
adAuxB\ (Sum, ((a, b), (c, d))) = ((+)\ a\ c, (+)\ b\ d)\\
adAuxB\ (Product, ((a, b), (c, d))) = ((*)\ a\ c, (+)\ ((*)\ a\ d)\ ((*)\ b\ c))\\
adAuxU\ (Negate, (a, b)) = (-a, -b)\\
adAuxU\ (E, (a, b)) = (expd\ a, (*)\ (expd\ a)\ b)
\end{array}
$$

Problema 2

$$Seja \qquad C_n = \frac{(2n)!}{(n+1)!(n!)}, \quad consideremos\ f\ =\ C_n \tag{6}$$

pretendemos aplicar a *regra da Algibreira* para esta função. Para isto descobrimos o valor da função no pontos zero e n+1:

$$\begin{array}{l} f\ 0 = 1 \\ f\ (n+1) = \frac{(2\ (n+1))!}{((n+1)+1)!(n+1)!} = \frac{(2\ n+2)!}{(n+2)!(n+1)!} = \frac{(2\ n+2)\ (2\ n+1)\ (2\ n)!}{(n+2)\ (n+1)\ (n)!(n+1)!} = \frac{2\ (n+1)\ (2\ n+1)\ (2\ n)!}{(n+2)\ (n+1)\ (n!)\ (n+1)!} = \\ \quad = \frac{2\ (2\ n+1)\ (2\ n)!}{(n+2)\ (n!)\ (n+1)!} = \frac{(2\ n)!}{(n+1)!(n)!} \cdot \frac{4\ n+2}{n+2} = f\ n \cdot \frac{k\ n}{s\ n} \end{array}$$

Tendo este resultado fomos descobrir os valores das função k e s nos pontos 0 e n+1

$$\begin{array}{l} k\ n = 4\ n + 2 \\ k\ 0 = 1 \\ k\ (n+1) = (4\ (n+1) + 2) = 4\ n + 6 = k\ n + 4 \\ \quad \text{--} \\ s\ n = n + 2 \\ s\ 0 = 2 \\ s\ (n+1) = n + 1 + 2 = s\ n + 1 \end{array}$$

A solução encontrada, tem como sesultado o seguinte *loop*

$$\begin{array}{l} cat = prj \cdot \mathsf{for}\ loop\ inic\ \mathbf{where} \\ \quad loop\ (f, k, s) = (f * k \div s, k + 4, s + 1) \\ \quad inic = (1, 2, 2) \\ \quad prj\ (f, k, s) = f \end{array}$$

Problema 3

A interpolação dada entre dois pontos pode é a soma da multiplicação de todas as coordenadas do primeiro ponto por (1-t) com a do segundo por t.

$$\begin{tikzcd} NPoint \arrow[r, "outList"] \arrow[d, "calcLine"'] & 1 + (\mathbb{Q} \times NPoint) \arrow[d, "recList\ (|h|)"] \\ ((OverTime\ NPoint) \uparrow NPoint) & 1 + (Q \times (OverTime\ NPoint \uparrow NPoint)) \arrow[l, "h"] \end{tikzcd}$$

Analisando o tipo da função calcLine podemos concluir que o nosso objetivo final deve ser retornar uma função que dado um *Npoint* resulta um *OverTime NPoint*.

Solução:

$$\begin{array}{l} calcLine :: NPoint \rightarrow (NPoint \rightarrow OverTime\ NPoint) \\ calcLine = cataList\ h\ \mathbf{where} \\ \quad h = [n, linear] \\ \quad n :: a \rightarrow (NPoint \rightarrow OverTime\ NPoint) \\ \quad n\ a\ b = nil \\ \quad linear :: (\mathbb{Q}, NPoint \rightarrow OverTime\ NPoint) \rightarrow (NPoint \rightarrow OverTime\ NPoint) \\ \quad linear\ (d, f)\ l = \mathbf{case}\ l\ \mathbf{of} \\ \quad\quad [\,] \rightarrow nil \\ \quad\quad (x : xs) \rightarrow \lambda z \rightarrow concat\ \$\ (sequenceA\ [singl \cdot linear1d\ d\ x, f\ xs])\ z \end{array}$$

Com a solução *calcLine* podemos obeter a função *deCastelejau* que implementa o algoritmo respetivo, ou seja faz a interpolação para um conjunto de pontos.

$$
\begin{array}{l}
deCasteljau :: [NPoint] \rightarrow OverTime\ NPoint \\
deCasteljau\ xs = \lambda pt \rightarrow hyloAlgForm\ (alg\ xs\ pt)\ (coalg\ xs\ pt)\ pt\ \mathbf{where} \\
\quad alg\ [\,]\ pt = nil\ pt \\
\quad alg\ [x]\ pt = \underline{x}\ pt \\
\quad alg\ xs\ pt = deCasteljau\ (init\ xs)\ pt \\
\quad coalg\ [\,]\ pt = nil\ pt \\
\quad coalg\ [x]\ pt = nil\ pt \\
\quad coalg\ xs\ pt = deCasteljau\ (tail\ xs)\ pt \\
hyloAlgForm = calcLine
\end{array}
$$

Problema 4

$$
\begin{array}{ll}
 & avg\_aux = \llparenthesis [b, q] \rrparenthesis \\
\equiv & \qquad \{ \text{ Aplicando a definição dada de avg\_aux } \} \\
 & \langle avg, length \rangle\ = \llparenthesis [b, q] \rrparenthesis \\
\equiv & \qquad \{ \text{ Univelsal-cata } \} \\
 & \langle avg, length \rangle\ = [b\ ,\ q]\ .\ recList\ \llparenthesis [b, q] \rrparenthesis\ .\ outList
\end{array}
$$

Para resolução desta função, de maneira a ignorar listas fazias criamos o código seguinte:

$$
\begin{array}{l}
outSingle\ (a : [\,]) = i_1\ a \\
outSingle\ (x : xs) = i_2\ (x, xs) \\
cataSingle\ g = g \cdot recList\ (cataSingle\ g) \cdot outSingle
\end{array}
$$

Com estas funções conseguimos definir o catamorfismo *avg_Aux*:

$$
\begin{tikzcd}
A* \arrow[r, "outSingle"] \arrow[d, "\langle avg, length \rangle"'] & A + (A \times A*) \arrow[d, "recList\ \llparenthesis [b,q] \rrparenthesis"] \\
(A \times B) & A + (A \times B) \arrow[l, "{[b,q]}"]
\end{tikzcd}
$$

$$
\begin{array}{l}
avg = \pi_1 \cdot avg\_aux \\
avg\_aux = cataSingle\ [b, q]\ \mathbf{where} \\
\quad b = \langle id, \underline{1} \rangle \\
\quad q = \langle avgAux, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \rangle \\
\quad avgAux\ (x, (a, l)) = (x + (a * l))\ /\ (l + 1)
\end{array}
$$

Solução para árvores de tipo LTree:

$$
\begin{array}{l}
avgLTree = \pi_1 \cdot \llparenthesis [b, q] \rrparenthesis\ \mathbf{where} \\
\quad b = \langle id, \underline{1} \rangle \\
\quad q = \langle agvLTreeAux, lTreeSize \rangle \\
\quad agvLTreeAux\ ((x1, x2), (y1, y2)) = ((x1 * x2) + (y1 * y2))\ /\ (x2 + y2) \\
\quad lTreeSize\ ((x1, x2), (y1, y2)) = x2 + y2
\end{array}
$$

Problema 5

Encontra-se em baixo o código F# desenvolvido:

```
// (1) Datatype definition -----------------------------------------------------

type BTree<'a> = Empty | Node of a * of BTree<'a> * BTree<'a>

let inBtree x = either (const Empty) Node x

let outBTree x =
     match x with
     | Empty  -> left ()
     | Node (a,t1,t2) -> Rigth (a,t1,t2)

// (2) Ana + cata + hylo -------------------------------------------------------

let baseBTree g f = g -|- (g >< (f >< f))

let recBTree f = baseBTree id f

let rec cataBTree g = g << (recBTree (cataBTree g)) << outBTree

let rec anaBTree g = inBTree << (recBTree (anaBTree g) ) << g

let hyloBTree h g = cataBTree h << anaBTree g

// (3) Map ---------------------------------------------------------------------

//instance Functor BTree
//         where fmap f = cataBTree ( inBTree . baseBTree f id )
let fmap f = cataBTree ( inBTree << baseBTree f id )

// (4) Examples ----------------------------------------------------------------

// (4.1) Inversion (mirror) ----------------------------------------------------

let invBTree x = cataBTree (inLTree << (id -|- id >< swap)) x

(* Recall the pointwise version:
invBTree () = Empty
invBTree (Node (a,(b,c))) = Node (a, (invBTree c,invBTree b))
*)

// (4.2) Counting --------------------------------------------------------------

let countBTree x = cataBTree (either (const 0) (succ . (uncurry (+)) . p2)) x

// (4.3) Serialization ---------------------------------------------------------
let inordt x = cataBTree inord x

let inordt x  = either nil join x
    where join (x,(l,r)) = l @ [x] @ r

let preordt x = cataBTree preord

let preord = (either nil f)
      where  f(x,(l,r))= x: l @ r

postordt = cataBTree (either nil f)
           where  f(x,(l,r))=l @ r @ [x]

// (4.4) Quicksort -------------------------------------------------------------
```

```
let qSort = hyloBTree inord qsep

let qsep []    = Left ()
let qsep (h:t) = Right (h,(s,l))
      where (s,l) = part (<h) t

let part p []                = ([],[])
let part p (h:t) | p h       = let (s,l) = part p t in (h:s,l)
                 | otherwise = let (s,l) = part p t in (s,h:l)

// (4.5) Traces ------------------------------------------------------------

let traces x = cataBTree (either (const [[]]) tunion) x

let tunion (a,(l,r)) = union (map (a:) l) (map (a:) r)

// (4.6) Towers of Hanoi --------------------------------------------------

let hanoi x = hyloBTree present strategy x

let present x = inord x

strategy (d,0) = Left ()
strategy (d,n+1) = Right ((n,d),((not d,n),(not d,n)))

-- (5) Depth and balancing (using mutual recursion) --------------------------

balBTree = p1.baldepth

depthBTree = p2.baldepth

baldepth = cataBTree g where
     g = either (const(True,1)) (h.(id><f))
     h(a,((b1,b2),(d1,d2))) = (b1 && b2 && abs(d1-d2)<=1,1+max d1 d2)
     f((b1,d1),(b2,d2)) = ((b1,b2),(d1,d2))

// (6) Going polytipic ---------------------------------------------------

let tnat f = either (const mempty) (theta << (f >< theta))
      where theta = uncurry mappend


let monBTree f = cataBTree (tnat f)

// alternative to (4.2) serialization ---------------------------------------

let preordt' = monBTree singl

// alternative to (4.1) counting ---------------------------------------------

countBTree' = monBTree (const (Sum 1))

// (7) Zipper ----------------------------------------------------------------

let plug [] t = t
let plug ((Dr False a l):z) t = Node (a,(plug z t,l))
let plug ((Dr True  a r):z) t = Node (a,(r,plug z,t))
```

Índice

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.