Cálculo de Programas
Trabalho Prático
MiEI+LCC — 2018/19

Departamento de Informática
Universidade do Minho

Junho de 2019

Grupo nr.	105
a83610	Rui Nuno Borges Cruz Oliveira
a84475	Ana Rita Miranda Rosendo
a85731	Gonçalo José Azevedo Esteves

1 Preâmbulo

A disciplina de 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, restringe-se a aplicação deste método à programação funcional em Haskell. 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, 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 `cp1819t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp1819t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp1819t.zip` e executando

```
$ lhs2TeX cp1819t.lhs > cp1819t.tex
$ pdflatex cp1819t
```

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
```

Por outro lado, o mesmo ficheiro `cp1819t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
$ ghci cp1819t.lhs
```

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

Abra o ficheiro `cp1819t.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}
```

vai ser seleccionado pelo GHCi para ser executado.

3 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de três 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 cp1819t.aux
$ makeindex cp1819t.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
```

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

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

Problema 1

Um compilador é um programa que traduz uma linguagem dita de *alto nível* numa linguagem (dita de *baixo nível*) que seja executável por uma máquina. Por exemplo, o GCC compila C/C++ em código objecto que corre numa variedade de arquitecturas.

Compiladores são normalmente programas complexos. Constam essencialmente de duas partes: o *analisador sintático* que lê o texto de entrada (o programa *fonte* a compilar) e cria uma sua representação interna, estruturada em árvore; e o *gerador de código* que converte essa representação interna em código executável. Note-se que tal representação intermédia pode ser usada para outros fins, por exemplo, para gerar uma listagem de qualidade (*pretty print*) do programa fonte.

O projecto de compiladores é um assunto complexo que será assunto de outras disciplinas. Neste trabalho pretende-se apenas fazer uma introdução ao assunto, mostrando como tais programas se podem construir funcionalmente à custa de cata/ana/hilo-morfismos da linguagem em causa.

Para cumprirmos o nosso objectivo, a linguagem desta questão terá que ser, naturalmente, muito simples: escolheu-se a das expressões aritméticas com inteiros, *eg.* $1+2$, $3*(4+5)$ etc. Como representação interna adopta-se o seguinte tipo polinomial, igualmente simples:

$$\mathbf{data}\ Expr = Num\ Int \mid Bop\ Expr\ Op\ Expr$$
$$\mathbf{data}\ Op = Op\ String$$

1. Escreva as definições dos {cata, ana e hilo}-morfismos deste tipo de dados segundo o método ensinado nesta disciplina (recorde módulos como *eg.* BTree etc).

2. Como aplicação do módulo desenvolvido no ponto 1, defina como {cata, ana ou hilo}-morfismo a função seguinte:

   - $calcula :: Expr \to Int$ que calcula o valor de uma expressão;

     Propriedade QuickCheck 1 *O valor zero é um elemento neutro da adição.*

     $$\begin{array}{l} prop\_neutro1 :: Expr \to Bool \\ prop\_neutro1 = calcula \cdot addZero \equiv calcula \ \mathbf{where} \\ \quad addZero\ e = Bop\ (Num\ 0)\ (Op\ \texttt{"+"})\ e \\ prop\_neutro2 :: Expr \to Bool \\ prop\_neutro2 = calcula \cdot addZero \equiv calcula \ \mathbf{where} \\ \quad addZero\ e = Bop\ e\ (Op\ \texttt{"+"})\ (Num\ 0) \end{array}$$

     Propriedade QuickCheck 2 *As operações de soma e multiplicação são comutativas.*

     $$\begin{array}{l} prop\_comuta = calcula \cdot mirror \equiv calcula \ \mathbf{where} \\ \quad mirror = cataExpr\ [Num, g2] \\ \quad g2 = \widehat{\widehat{Bop}} \cdot (swap \times id) \cdot assocl \cdot (id \times swap) \end{array}$$

3. Defina como {cata, ana ou hilo}-morfismos as funções

   - $compile :: String \to Codigo$ - trata-se do compilador propriamente dito. Deverá ser gerado código posfixo para uma máquina elementar de stack. O tipo *Codigo* pode ser definido à escolha. Dão-se a seguir exemplos de comportamentos aceitáveis para esta função:

     ```
     Tp4> compile "2+4"
     ["PUSH 2", "PUSH 4", "ADD"]
     Tp4> compile "3*(2+4)"
     ["PUSH 3", "PUSH 2", "PUSH 4", "ADD", "MUL"]
     Tp4> compile "(3*2)+4"
     ["PUSH 3", "PUSH 2", "MUL", "PUSH 4", "ADD"]
     Tp4>
     ```

   - $show' :: Expr \to String$ - gera a representação textual de uma *Expr* pode encarar-se como o *pretty printer* associado ao nosso compilador

     Propriedade QuickCheck 3 *Em anexo, é fornecido o código da função readExp, que é "inversa"da função show', tal como a propriedade seguinte descreve:*

     $$\begin{array}{l} prop\_inv :: Expr \to Bool \\ prop\_inv = \pi_1 \cdot head \cdot readExp \cdot show' \equiv id \end{array}$$

Valorização Em anexo é apresentado código Haskell que permite declarar *Expr* como instância da classe *Read*. Neste contexto, *read* pode ser vista como o analisador sintático do nosso minúsculo compilador de expressões aritméticas.

Analise o código apresentado, corra-o e escreva no seu relatório uma explicação **breve** do seu funcionamento, que deverá saber defender aquando da apresentação oral do relatório.

Exprima ainda o analisador sintático *readExp* como um anamorfismo.

Problema 2

Pretende-se neste problema definir uma linguagem gráfica "brinquedo" a duas dimensões (2D) capaz de especificar e desenhar agregações de caixas que contêm informação textual. Vamos designar essa linguagem por *L2D* e vamos defini-la como um tipo em Haskell:

$$\mathbf{type}\ L2D = X\ Caixa\ Tipo$$

onde X é a estrutura de dados

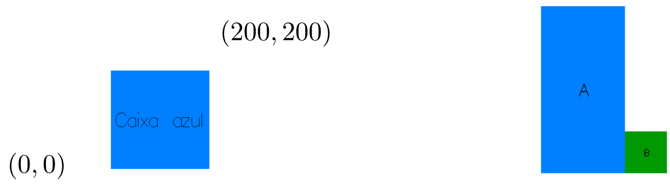

Figura 1: Caixa simples e caixa composta.

$$\mathbf{data}\ X\ a\ b = Unid\ a \mid Comp\ b\ (X\ a\ b)\ (X\ a\ b)\ \mathbf{deriving}\ Show$$

e onde:

$$\mathbf{type}\ Caixa = ((Int, Int), (Texto, G.Color))$$
$$\mathbf{type}\ Texto = String$$

Assim, cada caixa de texto é especificada pela sua largura, altura, o seu texto e a sua côr.[2] Por exemplo,

$((200, 200), (\texttt{"Caixa azul"}, col\_blue))$

designa a caixa da esquerda da figura 1.

O que a linguagem *L2D* faz é agregar tais caixas tipográficas umas com as outras segundo padrões especificados por vários "tipos", a saber,

$$\mathbf{data}\ Tipo = V \mid Vd \mid Ve \mid H \mid Ht \mid Hb$$

com o seguinte significado:

- V - agregação vertical alinhada ao centro
- Vd - agregação vertical justificada à direita
- Ve - agregação vertical justificada à esquerda
- H - agregação horizontal alinhada ao centro
- Hb - agregação horizontal alinhada pela base
- Ht - agregação horizontal alinhada pelo topo

Como *L2D* instancia o parâmetro b de X com *Tipo*, é fácil de ver que cada "frase" da linguagem *L2D* é representada por uma árvore binária em que cada nó indica qual o tipo de agregação a aplicar às suas duas sub-árvores. Por exemplo, a frase

$$\begin{aligned} ex2 = Comp\ Hb\ &(Unid\ ((100, 200), (\texttt{"A"}, col\_blue)))\\ &(Unid\ ((50, 50), (\texttt{"B"}, col\_green))) \end{aligned}$$

deverá corresponder à imagem da direita da figura 1. E poder-se-á ir tão longe quando a linguagem o permita. Por exemplo, pense na estrutura da frase que representa o *layout* da figura 2.

É importante notar que cada "caixa" não dispõe informação relativa ao seu posicionamento final na figura. De facto, é a posição relativa que deve ocupar face às restantes caixas que irá determinar a sua posição final. Este é um dos objectivos deste trabalho: *calcular o posicionamento absoluto de cada uma das caixas por forma a respeitar as restrições impostas pelas diversas agregações*. Para isso vamos considerar um tipo de dados que comporta a informação de todas as caixas devidamente posicionadas (i.e. com a informação adicional da origem onde a caixa deve ser colocada).

[2] Pode relacionar *Caixa* com as caixas de texto usadas nos jornais ou com *frames* da linguagem HTML usada na Internet.

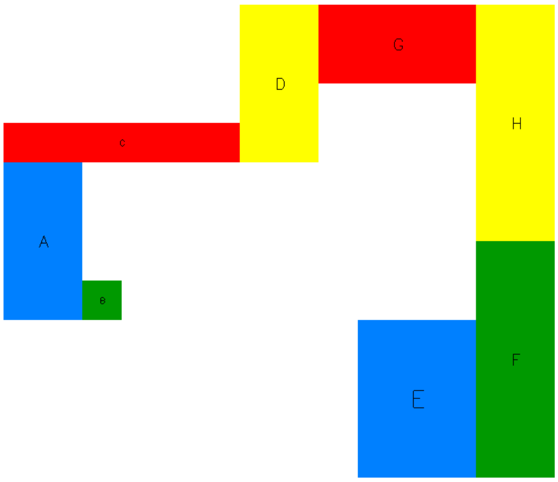

Figura 2: *Layout* feito de várias caixas coloridas.

$$\mathbf{type}\ Fig = [(Origem, Caixa)]$$
$$\mathbf{type}\ Origem = (Float, Float)$$

A informação mais relevante deste tipo é a referente à lista de "caixas posicionadas" (tipo $(Origem, Caixa)$). Regista-se aí a origem da caixa que, com a informação da sua altura e comprimento, permite definir todos os seus pontos (consideramos as caixas sempre paralelas aos eixos).

1. Forneça a definição da função $calc\_origems$, que calcula as coordenadas iniciais das caixas no plano:

$$calc\_origems :: (L2D, Origem) \rightarrow X\ (Caixa, Origem)\ ()$$

2. Forneça agora a definição da função $agrup\_caixas$, que agrupa todas as caixas e respectivas origens numa só lista:

$$agrup\_caixas :: X\ (Caixa, Origem)\ () \rightarrow Fig$$

Um segundo problema neste projecto é *descobrir como visualizar a informação gráfica calculada por* $desenho$. A nossa estratégia para superar o problema baseia-se na biblioteca Gloss, que permite a geração de gráficos 2D. Para tal disponibiliza-se a função

$$crCaixa :: Origem \rightarrow Float \rightarrow Float \rightarrow String \rightarrow G.Color \rightarrow G.Picture$$

que cria um rectângulo com base numa coordenada, um valor para a largura, um valor para a altura, um texto que irá servir de etiqueta, e a cor pretendida. Disponibiliza-se também a função

$$display :: G.Picture \rightarrow \mathsf{IO}\ ()$$

que dado um valor do tipo $G.picture$ abre uma janela com esse valor desenhado. O objectivo final deste exercício é implementar então uma função

$$mostra\_caixas :: (L2D, Origem) \rightarrow \mathsf{IO}\ ()$$

que dada uma frase da linguagem $L2D$ e coordenadas iniciais apresenta o respectivo desenho no ecrã. **Sugestão**: Use a função $G.pictures$ disponibilizada na biblioteca Gloss.

Problema 3

Nesta disciplina estudou-se como fazer programação dinâmica por cálculo, recorrendo à lei de recursividade mútua.[3]

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

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

Obter-se-á de imediato

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

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.[4]
- 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 no segundo grau a $x^2 + bx + c$ em $\mathbb{N}_0$. Seguindo o método estudado nas aulas[5], 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:

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

Qual é o assunto desta questão, então? Considerem fórmula que dá a série de Taylor da função coseno:

$$\mathit{cos}\ x = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i)!} x^{2i}$$

Pretende-se o ciclo-for que implementa a função $\mathit{cos}'\ x\ n$ que dá o valor dessa série tomando i até n inclusivé:

$$\mathit{cos}'\ x = \cdots \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \textbf{where}\ \cdots$$

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

Propriedade QuickCheck 4 *Testes de que $\mathit{cos}'\ x$ calcula bem o coseno de π e o coseno de $\pi\ /\ 2$:*

$$\begin{array}{l} \mathit{prop\_cos1}\ n = n \geqslant 10 \Rightarrow \mathit{abs}\ (\mathit{cos}\ \pi - \mathit{cos}'\ \pi\ n) < 0.001 \\ \mathit{prop\_cos2}\ n = n \geqslant 10 \Rightarrow \mathit{abs}\ (\mathit{cos}\ (\pi\ /\ 2) - \mathit{cos}'\ (\pi\ /\ 2)\ n) < 0.001 \end{array}$$

[3] Lei (3.94) em [2], página 98.
[4] Podem obviamente usar-se outros símbolos, mas numa primeiraleitura dá jeito usarem-se tais nomes.
[5] Secção 3.17 de [2].

Valorização Transliterar cos' para a linguagem C; compilar e testar o código. Conseguia, por intuição apenas, chegar a esta função?

Problema 4

Pretende-se nesta questão desenvolver uma biblioteca de funções para manipular *sistemas de ficheiros* genéricos. Um sistema de ficheiros será visto como uma associação de *nomes* a ficheiros ou *directorias*. Estas últimas serão vistas como sub-sistemas de ficheiros e assim recursivamente. Assumindo que a é o tipo dos identificadores dos ficheiros e directorias, e que b é o tipo do conteúdo dos ficheiros, podemos definir um tipo indutivo de dados para representar sistemas de ficheiros da seguinte forma:

$$\mathbf{data}\ FS\ a\ b = FS\ [(a, Node\ a\ b)]\ \mathbf{deriving}\ (Eq, Show)$$
$$\mathbf{data}\ Node\ a\ b = File\ b \mid Dir\ (FS\ a\ b)\ \mathbf{deriving}\ (Eq, Show)$$

Um caminho (*path*) neste sistema de ficheiros pode ser representado pelo seguinte tipo de dados:

$$\mathbf{type}\ Path\ a = [a]$$

Assumindo estes tipos de dados, o seguinte termo

```
FS [("f1", File "Ola"),
  ("d1", Dir (FS [("f2", File "Ole"),
    ("f3", File "Ole")
    ]))
  ]
```

representará um sistema de ficheiros em cuja raíz temos um ficheiro chamado *f1* com conteúdo `"Ola"` e uma directoria chamada `"d1"` constituída por dois ficheiros, um chamado `"f2"` e outro chamado `"f3"`, ambos com conteúdo `"Ole"`. Neste caso, tanto o tipo dos identificadores como o tipo do conteúdo dos ficheiros é $String$. No caso geral, o conteúdo de um ficheiro é arbitrário: pode ser um binário, um texto, uma colecção de dados, etc.

A definição das usuais funções $inFS$ e $recFS$ para este tipo é a seguinte:

$$inFS = FS \cdot \mathsf{map}\ (id \times inNode)$$
$$inNode = [File, Dir]$$
$$recFS\ f = baseFS\ id\ id\ f$$

Suponha que se pretende definir como um *catamorfismo* a função que conta o número de ficheiros existentes num sistema de ficheiros. Uma possível definição para esta função seria:

$$conta :: FS\ a\ b \rightarrow Int$$
$$conta = cataFS\ (sum \cdot \mathsf{map}\ ([\underline{1}, id] \cdot \pi_2))$$

O que é para fazer:

1. Definir as funções $outFS$, $baseFS$, $cataFS$, $anaFS$ e $hyloFS$.
2. Apresentar, no relatório, o diagrama de $cataFS$.
3. Definir as seguintes funções para manipulação de sistemas de ficheiros usando, obrigatoriamente, catamorfismos, anamorfismos ou hilomorfismos:
   (a) Verificação da integridade do sistema de ficheiros (i.e. verificar que não existem identificadores repetidos dentro da mesma directoria).
   $check :: FS\ a\ b \rightarrow Bool$

   Propriedade QuickCheck 5 *A integridade de um sistema de ficheiros não depende da ordem em que os últimos são listados na sua directoria:*

   $$prop\_check :: FS\ String\ String \rightarrow Bool$$
   $$prop\_check = check \cdot (cataFS\ (inFS \cdot reverse)) \equiv check$$

(b) Recolha do conteúdo de todos os ficheiros num arquivo indexado pelo *path*.
$tar :: FS\ a\ b \rightarrow [(Path\ a, b)]$

Propriedade QuickCheck 6 *O número de ficheiros no sistema deve ser igual ao número de ficheiros listados pela função* tar.

$$\begin{aligned}&prop\_tar :: FS\ String\ String \rightarrow Bool\\&prop\_tar = \mathsf{length} \cdot tar \equiv conta\end{aligned}$$

(c) Transformação de um arquivo com o conteúdo dos ficheiros indexado pelo *path* num sistema de ficheiros.
$untar :: [(Path\ a, b)] \rightarrow FS\ a\ b$
Sugestão: Use a função $joinDupDirs$ para juntar directorias que estejam na mesma pasta e que possuam o mesmo identificador.

Propriedade QuickCheck 7 *A composição* $tar \cdot untar$ *preserva o número de ficheiros no sistema.*

$$\begin{aligned}&prop\_untar :: [(Path\ String, String)] \rightarrow Property\\&prop\_untar = validPaths \Rightarrow ((\mathsf{length} \cdot tar \cdot untar) \equiv \mathsf{length})\\&validPaths :: [(Path\ String, String)] \rightarrow Bool\\&validPaths = (\equiv 0) \cdot \mathsf{length} \cdot (filter\ (\lambda(a, \_) \rightarrow \mathsf{length}\ a \equiv 0))\end{aligned}$$

(d) Localização de todos os *paths* onde existe um determinado ficheiro.
$find :: a \rightarrow FS\ a\ b \rightarrow [Path\ a]$

Propriedade QuickCheck 8 *A composição* $tar \cdot untar$ *preserva todos os ficheiros no sistema.*

$$\begin{aligned}&prop\_find :: String \rightarrow FS\ String\ String \rightarrow Bool\\&prop\_find = curry\ \$\\&\quad \mathsf{length} \cdot \widehat{find} \equiv \mathsf{length} \cdot \widehat{find} \cdot (id \times (untar \cdot tar))\end{aligned}$$

(e) Criação de um novo ficheiro num determinado *path*.
$new :: Path\ a \rightarrow b \rightarrow FS\ a\ b \rightarrow FS\ a\ b$

Propriedade QuickCheck 9 *A adição de um ficheiro não existente no sistema não origina ficheiros duplicados.*

$$\begin{aligned}&prop\_new :: ((Path\ String, String), FS\ String\ String) \rightarrow Property\\&prop\_new = ((validPath \wedge notDup) \wedge (check \cdot \pi_2)) \Rightarrow\\&\quad (checkFiles \cdot \widehat{\widehat{new}})\ \mathbf{where}\\&\quad validPath = (\not\equiv 0) \cdot \mathsf{length} \cdot \pi_1 \cdot \pi_1\\&\quad notDup = \neg \cdot \widehat{elem} \cdot (\pi_1 \times ((\mathsf{fmap}\ \pi_1) \cdot tar))\end{aligned}$$

Questão: *Supondo-se que no código acima se substitui a propriedade* $checkFiles$ *pela propriedade mais fraca* $check$*, será que a propriedade* $prop\_new$ *ainda é válida? Justifique a sua resposta.*

Propriedade QuickCheck 10 *A listagem de ficheiros logo após uma adição nunca poderá ser menor que a listagem de ficheiros antes dessa mesma adição.*

$$\begin{aligned}&prop\_new2 :: ((Path\ String, String), FS\ String\ String) \rightarrow Property\\&prop\_new2 = validPath \Rightarrow ((\mathsf{length} \cdot tar \cdot \pi_2) \leqslant (\mathsf{length} \cdot tar \cdot \widehat{\widehat{new}}))\ \mathbf{where}\\&\quad validPath = (\not\equiv 0) \cdot \mathsf{length} \cdot \pi_1 \cdot \pi_1\end{aligned}$$

(f) Duplicação de um ficheiro.
$cp :: Path\ a \rightarrow Path\ a \rightarrow FS\ a\ b \rightarrow FS\ a\ b$

Propriedade QuickCheck 11 *A listagem de ficheiros com um dado nome não diminui após uma duplicação.*

$$\begin{aligned}&prop\_cp :: ((Path\ String, Path\ String), FS\ String\ String) \rightarrow Bool\\&prop\_cp = \mathsf{length} \cdot tar \cdot \pi_2 \leqslant \mathsf{length} \cdot tar \cdot \widehat{\widehat{cp}}\end{aligned}$$

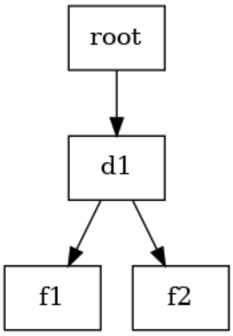

Figura 3: Exemplo de um sistema de ficheiros visualizado em Graphviz.

(g) Eliminação de um ficheiro.
$rm :: Path\ a \rightarrow FS\ a\ b \rightarrow FS\ a\ b$

Sugestão: Construir um anamorfismo $nav :: (Path\ a, FS\ a\ b) \rightarrow FS\ a\ b$ que navegue por um sistema de ficheiros tendo como base o $path$ dado como argumento.

Propriedade QuickCheck 12 *Remover duas vezes o mesmo ficheiro tem o mesmo efeito que o remover apenas uma vez.*

$$prop\_rm :: (Path\ String, FS\ String\ String) \rightarrow Bool$$
$$prop\_rm = \widehat{rm} \cdot \langle \pi_1, \widehat{rm} \rangle \equiv \widehat{rm}$$

Propriedade QuickCheck 13 *Adicionar um ficheiro e de seguida remover o mesmo não origina novos ficheiros no sistema.*

$$prop\_rm2 :: ((Path\ String, String), FS\ String\ String) \rightarrow Property$$
$$prop\_rm2 = validPath \Rightarrow ((\mathsf{length}\ \cdot tar \cdot \widehat{rm} \cdot \langle \pi_1 \cdot \pi_1, \widehat{new} \rangle)$$
$$\leqslant (\mathsf{length}\ \cdot tar \cdot \pi_2))\ \mathbf{where}$$
$$validPath = (\not\equiv 0) \cdot \mathsf{length}\ \cdot \pi_1 \cdot \pi_1$$

Valorização Definir uma função para visualizar em Graphviz a estrutura de um sistema de ficheiros. A Figura 3, por exemplo, apresenta a estrutura de um sistema com precisamente dois ficheiros dentro de uma directoria chamada `"d1"`.

Para realizar este exercício será necessário apenas escrever o anamorfismo

$$cFS2Exp :: (a, FS\ a\ b) \rightarrow (Exp\ ()\ a)$$

que converte a estrutura de um sistema de ficheiros numa árvore de expressões descrita em Exp.hs. A função $dotFS$ depois tratará de passar a estrutura do sistema de ficheiros para o visualizador.

Anexos

A Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Estudar o texto fonte deste trabalho para obter o efeito:[6]

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

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

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

B Programação dinâmica por recursividade múltipla

Neste anexo dão-se os detalhes da resolução do Exercício 3.30 dos apontamentos da disciplina[7], 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:

$$
exp\ x \quad = \quad \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{1}
$$

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:

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

Segundo a *regra de algibeira* descrita na página 3 deste enunciado, ter-se-á, de imediato:

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

[6]Exemplos tirados de [2].
[7]Cf. [2], página 102.

C Código fornecido

Problema 1

Tipos:

```
data Expr = Num Int
  | Bop Expr Op Expr deriving (Eq, Show)
data Op = Op String deriving (Eq, Show)
type Codigo = [String]
```

Functor de base:

$$baseExpr\ f\ g = id + (f \times (g \times g))$$

Instâncias:

```
instance Read Expr where
  readsPrec _ = readExp
```

Read para Exp's:

```
readOp :: String → [(Op, String)]
readOp input = do
  (x, y) ← lex input
  return ((Op x), y)
readNum :: ReadS Expr
readNum = (map (λ(x, y) → ((Num x), y))) · reads
readBinOp :: ReadS Expr
readBinOp = (map (λ((x, (y, z)), t) → ((Bop x y z), t))) ·
  ((readNum `ou` (pcurvos readExp))
    `depois` (readOp `depois` readExp))
readExp :: ReadS Expr
readExp = readBinOp `ou` (
  readNum `ou` (
  pcurvos readExp))
```

Combinadores:

```
depois :: (ReadS a) → (ReadS b) → ReadS (a, b)
depois _ _ [] = []
depois r1 r2 input = [((x, y), i2) | (x, i1) ← r1 input,
  (y, i2) ← r2 i1]
readSeq :: (ReadS a) → ReadS [a]
readSeq r input
  = case (r input) of
    [] → [([], input)]
    l → concat (map continua l)
      where continua (a, i) = map (c a) (readSeq r i)
        c x (xs, i) = ((x : xs), i)
ou :: (ReadS a) → (ReadS a) → ReadS a
ou r1 r2 input = (r1 input) ++ (r2 input)
senao :: (ReadS a) → (ReadS a) → ReadS a
senao r1 r2 input = case (r1 input) of
  [] → r2 input
  l    → l
readConst :: String → ReadS String
readConst c = (filter ((≡ c) · π1)) · lex
pcurvos = parentesis '(' ')'
```

```haskell
prectos = parentesis '[' ']'
chavetas = parentesis '{' '}'

parentesis :: Char -> Char -> (ReadS a) -> ReadS a
parentesis _ _ _ [] = []
parentesis ap pa r input
  = do
    ((_, (x, _)), c) <- ((readConst [ap]) `depois` (
      r `depois` (
      readConst [pa]))) input
    return (x, c)
```

Problema 2

Tipos:

```haskell
type Fig = [(Origem, Caixa)]
type Origem = (Float, Float)
```

“Helpers”:

```haskell
col_blue = G.azure
col_green = darkgreen

darkgreen = G.dark (G.dark G.green)
```

Exemplos:

```haskell
ex1Caixas = G.display (G.InWindow "Problema 4" (400, 400) (40, 40)) G.white $
  crCaixa (0, 0) 200 200 "Caixa azul" col_blue
ex2Caixas = G.display (G.InWindow "Problema 4" (400, 400) (40, 40)) G.white $
  caixasAndOrigin2Pict ((Comp Hb bbox gbox), (0.0, 0.0)) where
  bbox = Unid ((100, 200), ("A", col_blue))
  gbox = Unid ((50, 50), ("B", col_green))
ex3Caixas = G.display (G.InWindow "Problema 4" (400, 400) (40, 40)) G.white mtest where
  mtest = caixasAndOrigin2Pict $ (Comp Hb (Comp Ve bot top) (Comp Ve gbox2 ybox2), (0.0, 0.0))
  bbox1 = Unid ((100, 200), ("A", col_blue))
  bbox2 = Unid ((150, 200), ("E", col_blue))
  gbox1 = Unid ((50, 50), ("B", col_green))
  gbox2 = Unid ((100, 300), ("F", col_green))
  rbox1 = Unid ((300, 50), ("C", G.red))
  rbox2 = Unid ((200, 100), ("G", G.red))
  wbox1 = Unid ((450, 200), ("", G.white))
  ybox1 = Unid ((100, 200), ("D", G.yellow))
  ybox2 = Unid ((100, 300), ("H", G.yellow))
  bot = Comp Hb wbox1 bbox2
  top = (Comp Ve (Comp Hb bbox1 gbox1) (Comp Hb rbox1 (Comp H ybox1 rbox2)))
```

A seguinte função cria uma caixa a partir dos seguintes parâmetros: origem, largura, altura, etiqueta e côr de preenchimento.

```haskell
crCaixa :: Origem -> Float -> Float -> String -> G.Color -> G.Picture
crCaixa (x, y) w h l c = G.Translate (x + (w / 2)) (y + (h / 2)) $ G.pictures [caixa, etiqueta] where
  caixa = G.color c (G.rectangleSolid w h)
  etiqueta = G.translate calc_trans_x calc_trans_y $
    G.Scale calc_scale calc_scale $ G.color G.black $ G.Text l
  calc_trans_x = (-((fromIntegral (length l)) * calc_scale) / 2) * base_shift_x
  calc_trans_y = (-calc_scale / 2) * base_shift_y
  calc_scale = bscale * (min h w)
  bscale = 1 / 700
```

$$
\begin{aligned}
&base\_shift\_y = 100\\
&base\_shift\_x = 64
\end{aligned}
$$

Função para visualizar resultados gráficos:

$$display = G.display\ (G.InWindow\ \texttt{"Problema 4"}\ (400,400)\ (40,40))\ G.white$$

Problema 4

Funções para gestão de sistemas de ficheiros:

$$
\begin{aligned}
&concatFS = inFS \cdot \widehat{(+\!\!+)} \cdot (outFS \times outFS)\\
&mkdir\ (x,y) = FS\ [(x, Dir\ y)]\\
&mkfile\ (x,y) = FS\ [(x, File\ y)]\\
&joinDupDirs :: (Eq\ a) \Rightarrow (FS\ a\ b) \rightarrow (FS\ a\ b)\\
&joinDupDirs = anaFS\ (prepOut \cdot (id \times proc) \cdot prepIn)\ \mathbf{where}\\
&\quad prepIn = (id \times (\mathsf{map}\ \ (id \times outFS))) \cdot sls \cdot (\mathsf{map}\ \ distr) \cdot outFS\\
&\quad prepOut = (\mathsf{map}\ \ undistr) \cdot \widehat{(+\!\!+)} \cdot ((\mathsf{map}\ \ i_1) \times (\mathsf{map}\ \ i_2)) \cdot (id \times (\mathsf{map}\ \ (id \times inFS)))\\
&\quad proc = concat \cdot (\mathsf{map}\ \ joinDup) \cdot groupByName\\
&\quad sls = \langle lefts, rights \rangle\\
&joinDup :: [(a,[b])] \rightarrow [(a,[b])]\\
&joinDup = cataList\ [nil, g]\ \mathbf{where}\ g = return \cdot \langle \pi_1 \cdot \pi_1, concat \cdot (\mathsf{map}\ \ \pi_2) \cdot \widehat{(:)} \rangle\\
&createFSfromFile :: (Path\ a, b) \rightarrow (FS\ a\ b)\\
&createFSfromFile\ ([a], b) = mkfile\ (a, b)\\
&createFSfromFile\ (a : as, b) = mkdir\ (a, createFSfromFile\ (as, b))
\end{aligned}
$$

Funções auxiliares:

$$
\begin{aligned}
&checkFiles :: (Eq\ a) \Rightarrow FS\ a\ b \rightarrow Bool\\
&checkFiles = cataFS\ (\widehat{(\wedge)} \cdot \langle f, g \rangle)\ \mathbf{where}\\
&\quad f = nr \cdot (\mathsf{fmap}\ \pi_1) \cdot lefts \cdot (\mathsf{fmap}\ distr)\\
&\quad g = and \cdot rights \cdot (\mathsf{fmap}\ \pi_2)\\
&groupByName :: (Eq\ a) \Rightarrow [(a,[b])] \rightarrow [[(a,[b])]]\\
&groupByName = (groupBy\ (curry\ p))\ \mathbf{where}\\
&\quad p = \widehat{(\equiv)} \cdot (\pi_1 \times \pi_1)\\
&filterPath :: (Eq\ a) \Rightarrow Path\ a \rightarrow [(Path\ a, b)] \rightarrow [(Path\ a, b)]\\
&filterPath = filter \cdot (\lambda p \rightarrow \lambda(a,b) \rightarrow p \equiv a)
\end{aligned}
$$

Dados para testes:

- Sistema de ficheiros vazio:

$$efs = FS\ [\,]$$

- Nível 0

$$
\begin{aligned}
&f1 = FS\ [(\texttt{"f1"}, File\ \texttt{"hello world"})]\\
&f2 = FS\ [(\texttt{"f2"}, File\ \texttt{"more content"})]\\
&f00 = concatFS\ (f1, f2)\\
&f01 = concatFS\ (f1, mkdir\ (\texttt{"d1"}, efs))\\
&f02 = mkdir\ (\texttt{"d1"}, efs)
\end{aligned}
$$

- Nível 1

$$
\begin{aligned}
&f10 = mkdir\ (\texttt{"d1"}, f00)\\
&f11 = concatFS\ (mkdir\ (\texttt{"d1"}, f00), mkdir\ (\texttt{"d2"}, f00))\\
&f12 = concatFS\ (mkdir\ (\texttt{"d1"}, f00), mkdir\ (\texttt{"d2"}, f01))\\
&f13 = concatFS\ (mkdir\ (\texttt{"d1"}, f00), mkdir\ (\texttt{"d2"}, efs))
\end{aligned}
$$

- Nível 2

$$
\begin{aligned}
&f20 = mkdir\ (\texttt{"d1"}, f10)\\
&f21 = mkdir\ (\texttt{"d1"}, f11)\\
&f22 = mkdir\ (\texttt{"d1"}, f12)\\
&f23 = mkdir\ (\texttt{"d1"}, f13)\\
&f24 = concatFS\ (mkdir\ (\texttt{"d1"}, f10), mkdir\ (\texttt{"d2"}, f12))
\end{aligned}
$$

- Sistemas de ficheiros inválidos:

$$
\begin{aligned}
&ifs0 = concatFS\ (f1, f1)\\
&ifs1 = concatFS\ (f1, mkdir\ (\texttt{"f1"}, efs))\\
&ifs2 = mkdir\ (\texttt{"d1"}, ifs0)\\
&ifs3 = mkdir\ (\texttt{"d1"}, ifs1)\\
&ifs4 = concatFS\ (mkdir\ (\texttt{"d1"}, ifs1), mkdir\ (\texttt{"d2"}, f12))\\
&ifs5 = concatFS\ (mkdir\ (\texttt{"d1"}, f1), mkdir\ (\texttt{"d1"}, f2))\\
&ifs6 = mkdir\ (\texttt{"d1"}, ifs5)\\
&ifs7 = concatFS\ (mkdir\ (\texttt{"d1"}, f02), mkdir\ (\texttt{"d1"}, f02))
\end{aligned}
$$

Visualização em Graphviz:

$$
\begin{aligned}
&dotFS :: FS\ String\ b \rightarrow \mathsf{IO}\ ExitCode\\
&dotFS = dotpict \cdot bmap\ \underline{\texttt{""}}\ id \cdot (cFS2Exp\ \texttt{"root"})
\end{aligned}
$$

Outras funções auxiliares

Lógicas:

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

Compilação e execução dentro do interpretador:[8]

$$
run = \mathbf{do}\ \{system\ \texttt{"ghc cp1819t"}; system\ \texttt{"./cp1819t"}\}
$$

D Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções aos 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 e/ou outras funções auxiliares que sejam necessárias.

[8]Pode ser útil em testes envolvendo Gloss. Nesse caso, o teste em causa deve fazer parte de uma função $main$.

Problema 1

inExpr

$$
\begin{aligned}
& inExpr = [Num,\ Bop] \\
\equiv \quad & \{ \text{ Universal-+ (17) } \} \\
& \begin{cases} inExpr \cdot i_1 = Num \\ inExpr \cdot i_2 = Bop \end{cases} \\
\equiv \quad & \{ \text{ Igualdade Extensional (69), Def-comp (70) } \} \\
& \begin{cases} inExpr\ (i_1\ n) = Num\ n \\ inExpr\ (i_2\ (o, (e_1, e_1))) = Bop\ e_1\ o\ e_2 \end{cases}
\end{aligned}
$$

outExpr

$$
\begin{aligned}
& outExpr\ .\ [Num,\ Bop] = id \\
\equiv \quad & \{ \text{ Fusão-+ (20) } \} \\
& [outExpr\ .\ Num,\ outExpr\ .\ Bop] = id \\
\equiv \quad & \{ \text{ Universal-+ (17) } \} \\
& \begin{cases} id \cdot i_1 = outExpr \cdot Num \\ id \cdot i_2 = outExpr \cdot Bop \end{cases} \\
\equiv \quad & \{ \text{ Natural-id (1), Igualdade Extensional (69), Def-comp (70) } \} \\
& \begin{cases} outExpr\ (Num\ n) = i_1\ n \\ outExpr\ (Bop\ e_1\ o\ e_2) = i_2\ (o, (e_1, e_2)) \end{cases}
\end{aligned}
$$

recExpr

$$
\begin{tikzcd}
Expr \arrow[rr, bend left, "outExpr"] \arrow[r, phantom, "\cong"] \arrow[dd, "f"'] & {} & Int + (Op \times (Expr \times Expr)) \arrow[ll, bend left, "inExpr"] \arrow[dd, "id+(id\times(f\times f))"] \\
& & \\
C & & Int + (Op \times (C \times C)) \arrow[ll]
\end{tikzcd}
$$

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

cataExpr

$$
\begin{aligned}
\equiv & \quad \{ \text{Cancelamento-cata (44)} \} \\
& (\!| g |\!) \,.\, in = g \,.\, F (\!| g |\!) \\
\equiv & \quad \{ \text{in.out = id} \} \\
& (\!| g |\!) = g \,.\, F (\!| g |\!) \,.\, out \\
\equiv & \quad \{ \text{Aplicando as definições em Haskell já determinadas} \} \\
& cataExpr\ g = g \,.\, recExpr(cataExpr\ g) \,.\, outExpr
\end{aligned}
$$

anaExpr

$$
\begin{aligned}
\equiv & \quad \{ \text{Cancelamento-ana (53)} \} \\
& out \,.\, [\!(g)\!] = F [\!(g)\!] \,.\, g \\
\equiv & \quad \{ \text{in.out = id} \} \\
& [\!(g)\!] = in \,.\, F [\!(g)\!] \,.\, g \\
\equiv & \quad \{ \text{Aplicando as definições em Haskell já determinadas} \} \\
& anaExpr\ g = inExpr \,.\, recExpr(anaExpr\ g) \,.\, g
\end{aligned}
$$

hyloExpr

$$
\begin{aligned}
\equiv & \quad \{ \text{Definição de hilomorfismo} \} \\
& hyloExpr\ g\ h = (\!| g |\!) . [\!(h)\!] \\
\equiv & \quad \{ \text{Cancelamento-cata (44); Cancelamento-ana(53)} \} \\
& hyloExpr\ g\ h = g.F(\!| g |\!).inExpr.outExpr.F[\!(h)\!].h \\
\equiv & \quad \{ \text{in.out = id; Aplicando as definições em Haskell já determinadas} \} \\
& hyloExpr\ g\ h = g.recExpr(cataExprg).recExpr(anaExprh).h
\end{aligned}
$$

calcula

Por forma a obtermos o valor final de uma expressão, podemos recorrer ao uso do catamorfismo de *Expr*, definindo o diagrama de *calcula* da seguinte maneira.

$$
\begin{tikzcd}
Expr \arrow[r, phantom, "\cong"] \arrow[rr, bend left, "outExpr"] \arrow[d, "calcula"'] & & Int + (Op \times (Expr \times Expr)) \arrow[ll, bend left, "inExpr"] \arrow[d, "id+(id\times(calcula\times calcula))"] \\
Int & & Int + (Op \times (Int \times Int)) \arrow[ll, "g"]
\end{tikzcd}
$$

Neste caso, percorremos todas as expressões, de modo a determinar o valor de cada uma, aplicando posteriormente a cada par de valores (obtido a partir de cada par de expressões) a operação determinada pelo *Op*, emparelhado, inicialmente, com cada par de expressões (agora um par de *Int*). Ou seja, o gene g pode ser representado pelo seguinte diagrama:

$$\begin{tikzcd}
Int + (Op \times (Int \times Int)) \arrow[d, "id+((operAux.outOp)\times id)"'] \\
Int + (F \times (Int \times Int)) \arrow[d, "id+cal"'] \\
Int + Int \arrow[d, "{[id,id]}"'] \\
Int
\end{tikzcd}$$

Onde *operAux* é a função que, dada uma *String* correspondende ao símbolo de uma operação aritmética, devolve essa mesma operação; e *cal* é a função que recebe um par com uma operação aritmética e um par de *Int* e devolve o resultado da aplicação dessa operação aos elementos do par de inteiros. Assim sendo, podemos afirmar que o gene g se define como:

$$\begin{aligned}
& g = [id,\ id].(id\ +\ cal).(id\ +\ ((operAux.outOp)\ \times id)) \\
\equiv & \qquad \{ \text{ Absorção-+ (22) } \} \\
& g = [id.id.id,\ id.cal.((operAux.outOp) \times id)] \\
\equiv & \qquad \{ \text{ Natural-id (1); Def-x (10) } \} \\
& g = [id,\ cal. < operAux.outOp.\pi 1,\ \pi 2 >]
\end{aligned}$$

Por fim, podemos concluir que a função *calcula* poderá ser definida como:

$$\boxed{calcula = cataExpr\ [id, cal \cdot \langle operAux \cdot outOp \cdot \pi_1, \pi_2 \rangle]}$$

show'

Tal como para *calcula*, podemos definir a função *show'* como um catamorfismo de *Expr*. Assim sendo, chegamos ao seguinte diagrama representativo de *show'*.

$$\begin{tikzcd}
Expr \arrow[r, bend left, "outExpr"] \arrow[d, "show'"'] \arrow[r, phantom, "\cong"] & Int + (Op \times (Expr \times Expr)) \arrow[l, bend left, "inExpr"] \arrow[d, "id+(id\times(show'\times show'))"] \\
String & Int + (Op \times (String \times String)) \arrow[l, "g"]
\end{tikzcd}$$

Após isto, é necessário definir o gene deste catamorfismo. Para esta função, a ideia será determinar o formato em *String* de cada uma das *Expr*, concatenando-as depois com o respetivo operador aritmético associado, guardado em cada *Op*. No entanto, temos de manter o formato ”original”da expressão, ou seja, a concatenação terá de respeitar a seguinte ordem: a *String* da esquerda (do par), seguida do operador aritmético, seguida da *String* da direita. Deste modo, chegamos a seguinte representação por diagrama do gene *g*.

$$\begin{array}{c} Int + (Op \times (String \times String)) \\ \downarrow {\scriptstyle id+(assocl.(outOp \times id))} \\ Int + ((String \times String) \times String) \\ \downarrow {\scriptstyle id+(conc.swap \times id)} \\ Int + (String \times String) \\ \downarrow {\scriptstyle id+pcurv.conc} \\ Int + String \\ \downarrow {\scriptstyle [showMod, id]} \\ String \end{array}$$

Neste diagrama, a função *pcurv* coloca uma *String* dentro de parêntesis curvos, enquanto que a função *showMod* aplica a função *show* a um inteiro, transformando-o na *String* correspondente, e caso o inteiro seja negativo, ainda aplica a função *pcurv* ao valor em *String* desse inteiro. Assim sendo, o gene *g* poderá ser definido da seguinte forma:

$$\begin{array}{ll} & g = [showMod,\ id].(id\ +\ pcurv.conc).(id\ +\ (conc.swap\ \times id)).(id\ +\ (assocl.(outOp\ \times id))) \\ \equiv & \qquad \{ \text{ Absorção-+ (22), Natural-id (1) } \} \\ & g = [showMod,\ pcurv.conc.(conc.swap \times id).assocl.(outOp\ \times id)] \\ \equiv & \qquad \{ \text{ Def-x (10) } \} \\ & g = [showMod,\ pcurv.conc. < conc.swap.\pi 1,\ \pi 2 > .assocl. < outOp.\pi 1,\ \pi 2 >] \end{array}$$

Finalizando, podemos definir a função *show′* como:

$$\boxed{show' = cataExpr\ [showMod, pcurv \cdot \mathsf{conc} \cdot \langle \mathsf{conc} \cdot swap \cdot \pi_1, \pi_2 \rangle \cdot assocl \cdot \langle outOp \cdot \pi_1, \pi_2 \rangle]}$$

compile

Por forma a ser mais fácil definir esta função, podemos recorrer à função dada *readExp*, que transforma uma *String* numa lista de pares *Expr* e *String*. Sabendo que o primeiro par da lista criada por *readExp* possui a *Expr* da *String* recebida como input, podemos obtê-la, tal como o diagrama demonstra.

$$\begin{array}{c} String \\ \downarrow {\scriptstyle readExp} \\ (Expr \times String)* \\ \downarrow {\scriptstyle \pi 1.head} \\ Expr \end{array}$$

Agora, podemos, novamente, recorrer ao catamorfismo de *Expr*, por forma a obter a definição de *compilaAux*, que será a função que, para uma dada expressão, a transforma em código-máquina.

$$\begin{tikzcd} Expr \arrow[rr, bend left, "outExpr"] \arrow[d, "compileAux"'] \arrow[r, phantom, "\cong"] & & Int + (Op \times (Expr \times Expr)) \arrow[ll, bend left, "inExpr"] \arrow[d, "id+(id\times(compileAux\times compileAux))"] \\ Codigo & & Int + (Op \times (Codigo \times Codigo)) \arrow[ll, "g"] \end{tikzcd}$$

Posto isto, teremos de determinar o gene para este catamorfismo. Neste caso, a intenção será concatenar o par de *Codigo* obtido de cada par de *Expr*, e a este concatenar, no fim, a operação aritmética a aplicar, "traduzida"para linguagem máquina. Isto deve-se ao facto de que, em linguagem máquina, primeiro são introduzidos os dois valores aos quais se vai aplicar uma operação, e só depois a respetiva operação. Assim sendo, chegamos ao diagrama representativo do gene deste catamorfismo.

$$\begin{tikzcd} Int + (Op \times (Codigo \times Codigo)) \arrow[d, "id+(swap.(singl.get.outOp\times conc))"'] \\ Int + (Codigo \times Codigo) \arrow[d, "{[singl.push,conc]}"'] \\ Codigo \end{tikzcd}$$

Convém realçar que a função *get* transforma uma *String* que seja um símbolo de uma operação aritmética, na *String* representativa dessa operação em linguagem máquina; e que a função *push* concatena a expressão *PUSH* à representação, em *String*, do valor do *Int* que recebe (representação essa fornecida pela função *show*). Como tal, podemos definir o gene *g* como:

$$g = [singl.push,\ conc].(id\ +\ (swap.(singl.get.outOp \times conc)))$$

$$\equiv \qquad \{ \text{ Absorção-+ (22), Natural-id (1) } \}$$

$$g = [singl.push,\ conc.swap.(singl.get.outOp \times conc)]$$

$$\equiv \qquad \{ \text{ Def-x (10); swap = (split (p2) (p1)); Fusão-x (9) } \}$$

$$g = [singl.push,\ conc. < \pi 2. < singl.get.outOp.\pi 1,\ conc.\pi 2 >,\ \pi 1. < singl.get.outOp.\pi 1,\ conc.\pi 2 >>]$$

$$\equiv \qquad \{ \text{ Cancelamento-x (7) } \}$$

$$g = [singl.push,\ conc. < conc.\pi 2,\ singl.get.outOp.\pi 1 >]$$

Deste modo, podemos definir a função *compileAux* como:

$$\boxed{compileAux = cataExpr\ [singl \cdot push, \mathsf{conc} \cdot \langle \mathsf{conc} \cdot \pi_2, singl \cdot get \cdot outOp \cdot \pi_1 \rangle]}$$

Concluindo, assim, a seguinte definição de *compile*:

$$\boxed{compile = compileAux \cdot \pi_1 \cdot head \cdot readExp}$$

Solução

$$
\begin{array}{l}
\mathit{inExpr} :: \mathit{Int} + (\mathit{Op}, (\mathit{Expr}, \mathit{Expr})) \rightarrow \mathit{Expr} \\
\mathit{inExpr}\ (i_1\ n) = \mathit{Num}\ n \\
\mathit{inExpr}\ (i_2\ (o, (e_1, e_2))) = \mathit{Bop}\ e_1\ o\ e_2 \\
\mathit{outExpr} :: \mathit{Expr} \rightarrow \mathit{Int} + (\mathit{Op}, (\mathit{Expr}, \mathit{Expr})) \\
\mathit{outExpr}\ (\mathit{Num}\ n) = i_1\ n \\
\mathit{outExpr}\ (\mathit{Bop}\ e_1\ o\ e_2) = i_2\ (o, (e_1, e_2)) \\
\mathit{inOp} :: \mathit{String} \rightarrow \mathit{Op} \\
\mathit{inOp} = \mathit{Op} \\
\mathit{outOp} :: \mathit{Op} \rightarrow \mathit{String} \\
\mathit{outOp}\ (\mathit{Op}\ a) = a \\
\mathit{recExpr}\ f = \mathit{baseExpr}\ \mathit{id}\ f \\
\mathit{cataExpr}\ g = g \cdot \mathit{recExpr}\ (\mathit{cataExpr}\ g) \cdot \mathit{outExpr} \\
\mathit{anaExpr}\ g = \mathit{inExpr} \cdot \mathit{recExpr}\ (\mathit{anaExpr}\ g) \cdot g \\
\mathit{hyloExpr}\ h\ g = h \cdot \mathit{recExpr}\ (\mathit{cataExpr}\ h) \cdot \mathit{recExpr}\ (\mathit{anaExpr}\ g) \cdot g \\
\mathit{calcula} :: \mathit{Expr} \rightarrow \mathit{Int} \\
\mathit{calcula} = \mathit{cataExpr}\ [\mathit{id}, \mathit{cal} \cdot \langle \mathit{operAux} \cdot \mathit{outOp} \cdot \pi_1, \pi_2 \rangle] \\
\quad \textbf{where}\ \mathit{cal}\ (o, (x, y)) = o\ x\ y \\
\mathit{operAux}\ \texttt{"+"} = (+) \\
\mathit{operAux}\ \texttt{"-"} = (-) \\
\mathit{operAux}\ \texttt{"*"} = (*) \\
\mathit{operAux}\ \texttt{"/"} = \cdot \div \cdot \\
\mathit{show}' = \mathit{cataExpr}\ [\mathit{showMod}, \mathit{pcurv} \cdot \mathsf{conc} \cdot \langle \mathsf{conc} \cdot \mathit{swap} \cdot \pi_1, \pi_2 \rangle \cdot \mathit{assocl} \cdot \langle \mathit{outOp} \cdot \pi_1, \pi_2 \rangle] \\
\quad \textbf{where}\ \mathit{pcurv} = \mathsf{conc} \cdot \langle \underline{\texttt{"("}}, \mathsf{conc} \cdot \langle \mathit{id}, \underline{\texttt{")"}} \rangle \rangle \\
\qquad \mathit{showMod} = \mathit{cond}\ (<0)\ (\mathit{pcurv} \cdot \mathit{show})\ (\mathit{show}) \\
\mathit{compile} :: \mathit{String} \rightarrow \mathit{Codigo} \\
\mathit{compile} = \mathit{compileAux} \cdot \pi_1 \cdot \mathit{head} \cdot \mathit{readExp} \\
\mathit{compileAux} = \mathit{cataExpr}\ [\mathit{singl} \cdot \mathit{push}, \mathsf{conc} \cdot \langle \mathsf{conc} \cdot \pi_2, \mathit{singl} \cdot \mathit{get} \cdot \mathit{outOp} \cdot \pi_1 \rangle] \\
\quad \textbf{where}\ \mathit{push} = \mathsf{conc} \cdot \langle \underline{\texttt{"PUSH "}}, \mathit{show} \rangle \\
\mathit{get}\ \texttt{"+"} = \texttt{"ADD"} \\
\mathit{get}\ \texttt{"-"} = \texttt{"SUB"} \\
\mathit{get}\ \texttt{"*"} = \texttt{"MUL"} \\
\mathit{get}\ \texttt{"/"} = \texttt{"DIV"}
\end{array}
$$

Valorização

Para começar, a função *readExp* consegue ”compreender”que as operações compreendidas entre parêntesis são prioritárias e, como tal, estas deverão estar compreendidas dentro de uma mesma *Expr*.

Esta função interpreta a *String* inserida, dividindo a em partes: separa os números, os operadores aritméticos e as expressões compreendidas entre parêntesis. Posto isto, começa a construir a expressão final por partes, criando primeiramente uma *Expr* com o primeiro número/expressão inserido, o primeiro operador inserido (fora de parêntesis) e, por fim, o segundo número/expressão inserido/a; o restante que não foi lido, é guardado numa *String*, que emparelha com a *Expr* criada. Para criar o segundo par, a função pega no par anteriormente criado, coloca a *Expr* lá presente como primeira *Expr* da nova *Expr* (no conjunto *Bop Expr Op Expr*); depois, pega na *String* emparelhada e vai buscar o próximo operador (que será o primeiro caractér da *String*) e o próximo número/expressão, que será a segunda *Expr* dentro da nova *Expr* que está a ser criada. Mais uma vez, guarda o resto da *String* não lida numa *String*, emparelhada com a *Expr* recém-criada.

Isto ocorre sucessivamente, até que a *String* fica vazia, o que significa que a *Expr* emparelhada com esta *String* vazia, será a *Expr* final. Deste modo, ficamos com uma lista de pares *(Expr, String)*, estando presente na cabeça da lista a *Expr* correspondente à *String* introduzida.

Problema 2

inL2D

$$
\begin{aligned}
& inL2D = [Unid,\ Comp] \\
\equiv \quad & \{ \text{ Universal-+ (17) } \} \\
& \begin{cases} inL2D \cdot i_1 = Unid \\ inL2D \cdot i_2 = Comp \end{cases} \\
\equiv \quad & \{ \text{ Igualdade Extensional (69), Def-comp (70) } \} \\
& \begin{cases} inL2D\ (i_1\ a) = Unid\ a \\ inL2D\ (i_2\ (b, (a, c))) = Comp\ b\ a\ c \end{cases}
\end{aligned}
$$

outL2D

$$
\begin{aligned}
& outL2D\ .\ [Unid,\ Comp] = id \\
\equiv \quad & \{ \text{ Fusão-+ (20) } \} \\
& [outL2D\ .\ Unid,\ outL2D\ .\ Comp] = id \\
\equiv \quad & \{ \text{ Universal-+ (17) } \} \\
& \begin{cases} id \cdot i_1 = outL2D \cdot Unid \\ id \cdot i_2 = outL2D \cdot Comp \end{cases} \\
\equiv \quad & \{ \text{ Natural-id (1), Igualdade Extensional (69), Def-comp (70) } \} \\
& \begin{cases} outL2D\ (Unid\ a) = i_1\ a \\ outL2D\ (Comp\ b\ a\ c) = i_2\ (b, (a, c)) \end{cases}
\end{aligned}
$$

baseL2D e recL2D

$$
\begin{tikzcd}
X \arrow[r, bend left, "outL2D"] \arrow[r, phantom, "\cong"] \arrow[d, "g"'] & A + (B \times (X \times X)) \arrow[l, bend left, "inL2D"] \arrow[d, "h+(f\times(g\times g))"] \\
C & A' + (B' \times (C \times C)) \arrow[l]
\end{tikzcd}
$$

$$
baseL2D\ h\ f\ g = h + (f \times (g \times g))
$$

$$
\begin{aligned}
& recL2D\ f = id + (id \times (f \times f)) \\
\equiv \quad & \{ \text{ Aplicando a definição dada de baseL2D } \} \\
& recL2D\ f = baseL2D\ id\ id\ f
\end{aligned}
$$

cataL2D

$$
\begin{aligned}
\equiv & \quad \{ \text{ Cancelamento-cata (44) } \} \\
& (\!|g|\!) \; . \; in = g \; . \; F (\!|g|\!) \\
\equiv & \quad \{ \text{ in.out = id } \} \\
& (\!|g|\!) = g \; . \; F (\!|g|\!) \; . \; out \\
\equiv & \quad \{ \text{ Aplicando as definições em Haskell já determinadas } \} \\
& cataL2D \; g = g \; . \; recL2D (cataL2D \; g) \; . \; outL2D
\end{aligned}
$$

anaL2D

$$
\begin{aligned}
\equiv & \quad \{ \text{ Cancelamento-ana (53) } \} \\
& out \; . \; [\!(g)\!] = F [\!(g)\!] \; . \; g \\
\equiv & \quad \{ \text{ in.out = id } \} \\
& [\!(g)\!] = in \; . \; F [\!(g)\!] \; . \; g \\
\equiv & \quad \{ \text{ Aplicando as definições em Haskell já determinadas } \} \\
& anaL2D \; g = inL2D \; . \; recL2D (anaL2D \; g) \; . \; g
\end{aligned}
$$

hyloL2D

$$
\begin{aligned}
\equiv & \quad \{ \text{ Definição de hilomorfismo } \} \\
& hyloL2D \; g \; h = (\!|g|\!) . [\!(h)\!] \\
\equiv & \quad \{ \text{ Cancelamento-cata (44); Cancelamento-ana(53) } \} \\
& hyloL2D \; g \; h = g . F (\!|g|\!) . inL2D . outL2D . F [\!(h)\!] . h \\
\equiv & \quad \{ \text{ in.out = id; Aplicando as definições em Haskell já determinadas } \} \\
& hyloL2D \; g \; h = g . recL2D (cataL2D g) . recL2D (anaL2D h) . h
\end{aligned}
$$

dimen

Logo à partida, sabemos que quando duas caixas são agregadas na vertical, independetemente de como as agregamos (*V*, *Ve* ou *Vd*), a altura total do conjunto será a soma das alturas das duas caixas, enquanto que a sua largura será igual á da caixa mais larga. O mesmo se aplica, de forma inversa, a quando da agregação de caixas na horizontal (*H*, *Ht* ou *Hb*) - a sua largura será a soma das larguras de ambas as caixas, enquanto que a altura será igual à da caixa mais alta. Posto isto, e recorrendo ao uso de um catamorfismo, podemos definir o seguinte diagrama:

$$
\begin{tikzcd}
L2D \arrow[rr, bend left, "outL2D"] \arrow[d, "dimen"'] \arrow[r, phantom, "\cong"] & & Caixa + (Tipo \times (L2D \times L2D)) \arrow[ll, bend left, "inL2D"] \arrow[d, "id + (id \times (dimen \times dimen))"] \\
(Float \times Float) & & Caixa + (Tipo \times ((Float \times Float) \times (Float \times Float))) \arrow[ll, "g"]
\end{tikzcd}
$$

Deste modo, fica assim a ser necessário determinar o gene *g*. No caso de apenas termos uma caixa, as dimensões desta já estão declaradas, em formato *Int*. Como tal, é apenas necessário aplicar-lhes apenas a função *fromIntegral*, de modo a ficar compatível com o tipo de saída da função *dimen* (*(Float, Float)*). No caso de termos um par de *L2D* e o *Tipo* de agregação que as une, após aplicarmos a função *recL2D (dimen)* vamos ficar com dois pares de *Float*, um para cada *L2D*. Uma vez que as caixas da direita são sempre agregadas as da esquerda tendo em conta o tipo associado ao par *L2D*, vamos calcular a dimensão total de um par de *L2D* usando as dimensões de ambas as caixas e o tipo associado. Assim sendo, podemos definir o gene *g* com o seguinte diagrama:

$$
\begin{tikzcd}
Caixa + (Tipo \times ((Float \times Float) \times (Float \times Float))) \arrow[d, "{[(fromIntegral \times fromIntegral).\pi 1, aux]}"'] \\
(Float \times Float)
\end{tikzcd}
$$

Agora, é necessário definir a função *aux*, que tratará de calcular as dimensões de um par de caixas, usando as dimensões de cada uma e sabendo a forma como estão unidas. Assim sendo, o mais fácil será defini-la usando um condicional, da seguinte forma:

$$\boxed{aux = (cond\ (detTipo \cdot \pi_1)\ ((\widehat{max} \times \widehat{(+)}) \cdot \pi_2)\ ((\widehat{(+)} \times \widehat{max}) \cdot \pi_2)) \cdot get}$$

Neste caso, a função *detTipo* determina o tipo de união das caixas - se for uma união vertical (*V, Ve, Vd*) retorna *True*, se for uma união horizontal (*H, Ht, Hb*) retorna *False*; enquanto que a função *get* reorganiza os pares com as dimensões, juntando as larguras no primeiro par e as alturas no segundo. Ou seja, para um input de entrada igual a *((x,y),(a,b))* ficamos com *((x,a),(y,b))*. Deste modo, quando estamos perante o "caso verdadeiro", ou seja, quando verificamos que a união é vertical, determinamos qual a largura máxima, de entre as duas caixas, e calculamos a soma das alturas. Quando estamos perante uma união horizontal, fazemos o oposto. No fim de tudo isto, podemos chegar à seguinte definição de *dimen*:

$$\boxed{dimen = cataL2D\ [(fromIntegral \times fromIntegral) \cdot \pi_1, aux]}$$

collectLeafs

De forma indutiva, é possível chegar a uma definição para *collectLeafs*, tendo por base o catamorfismo de *L2D*. Como tal, poderemos estabelecer o seguinte diagrama:

$$\begin{tikzcd}
X \arrow[rr, bend left, "outL2D"] \arrow[d, "collectLeafs"'] & \cong & A + (B \times (X \times X)) \arrow[ll, bend left, "inL2D"] \arrow[d, "id+(id\times(collectLeafs\times collectLeafs))"] \\
A* & & A + (B \times (A* \times A*)) \arrow[ll, "g"]
\end{tikzcd}$$

É agora necessário, no entanto, definir o gene g do catamorfismo. Ora, como o nosso objetivo será simplesmente determinar uma lista de A's, para qualquer A, precisamos, então, de determinar a lista de A's de cada um dos X do par, e concatenar as listas. Assim sendo, poderemos definir o diagrama que se segue para caracterizar o gene do catamorfismo.

$$\begin{tikzcd}
A + (B \times (A* \times A*)) \arrow[d, "{[singl, conc.\pi2]}"'] \\
A*
\end{tikzcd}$$

Deste modo, podemos concluir a seguinte definição de *collectLeafs*:

$$\boxed{collectLeafs = cataL2D\ [singl, \mathsf{conc} \cdot \pi_2]}$$

agrupcaixas

Olhando para a tipagem da função (recebe um *(X (Caixa, Origem) ())* e devolve uma *Fig*, que é uma lista de *(Origem, Caixa)*), podemos concluir que a função devolve todas as "folhas", às quais é aplicada um swap. Como tal, podemos chegar à seguinte definição da função *agrupcaixas*:

$$\boxed{agrup\_caixas = (\mathsf{map}\ \ swap) \cdot collectLeafs}$$

calcOrigins

Uma vez que a tipagem da função nos indica que ela parte de um *X A B* e chega a outro *X A B*, podemos assumir que esta função poderá ser definida como um anamorfismo de *L2D*, apesar de possuirem *A* e *B* diferentes. Como tal, chegamos ao seguinte diagrama

$$\begin{tikzcd}
X \arrow[r, "h"] \arrow[d, "calcOrigins"'] & (Caixa \times Origem) + (1 \times (X \times X)) \arrow[d, "id+(id\times(calcOrigins\times calcOrigins))"] \\
X' & (Caixa \times Origem) + (1 \times (X' \times X')) \arrow[l, "inL2D"]
\end{tikzcd}$$

Onde X é o tipo *(L2D, Origem)* e X' o tipo *(X (Caixa, Origem) ())*. Deste modo, necessitamos agora de definir o gene h do anamorfismo. Uma vez que a nossa função recebe todas as caixas guardadas na estrutura *L2D* e a origem inicial, o nosso intuito será atribuir a origem à primeira caixa (que será a que estará o mais à esquerda possível) e depois calcular a origem da caixa da direita, tendo por base as dimensões que a caixa da esquerda ocupa, e o tipo de enquandramento que queremos fazer. Como tal, podemos desenvolver o gene *h* da seguinte forma:

$$\begin{array}{c} L2D \times Origem \\ \downarrow{\scriptstyle outL2D \times id} \\ (Caixa + (Tipo \times (L2D \times L2D))) \times Origem \\ \downarrow{\scriptstyle distl} \\ (Caixa \times Origem) + ((Tipo \times (L2D \times L2D)) \times Origem) \\ \downarrow{\scriptstyle calcOriginsAux} \\ (Caixa \times Origem) + (1 \times (X' \times X')) \end{array}$$

Posto isto, é agora necessário definir a função *calcOriginsAux*. Este poderá ser definida através do seguinte diagrama:

$$\begin{array}{c} (Caixa \times Origem) + ((Tipo \times (L2D \times L2D)) \times Origem) \\ \downarrow{\scriptstyle id+aux1} \\ (Caixa \times Origem) + (Tipo \times ((L2D \times Origem) \times (L2D \times Origem))) \\ \downarrow{\scriptstyle id+aux2} \\ (Caixa \times Origem) + (Tipo \times ((L2D \times Origem) \times (L2D \times Origem))) \\ \downarrow{\scriptstyle id+(! \times id)} \\ (Caixa \times Origem) + (1 \times ((L2D \times Origem) \times (L2D \times Origem))) \end{array}$$

Neste caso, a função *aux1* irá "distribuir"a origem inicial pelos dois *L2D* existentes, criando dois pares *L2D x Origem*, enquanto que a função *aux2* apenas irá alterar a origem do segundo par, atualizando-a tendo em conta a dimensão da *L2D* do primeiro par (determinada através do uso da função *dimen*), o *Tipo* que caracteriza a união dos dois *L2D* e a origem inicial. Estes atributos serão utilizados pela função *calc* para determinar a nova origem. Deste modo, podemos definir a função *calcOriginsAux* como:

$$\begin{aligned} & calcOriginsAux = (id\ +\ (!\ \times id)).(id\ +\ aux2).(id\ +\ aux1) \\ \equiv \quad & \{ \text{ Def-+ (21), Absorção-+ (22), Natural-id (1) } \} \\ & calcOriginsAux = [i1,\ i2.(!\ \times id).aux2.aux1)] \\ \equiv \quad & \{ \text{ Def-x (10) } \} \\ & calcOriginsAux = [i1,\ i2. <!.\pi1,\ id.\pi2 > .aux2.aux1)] \end{aligned}$$

Onde *aux1* e *aux2* são definidas como:

$$\boxed{\begin{array}{c} aux1 = \langle \pi_1 \cdot \pi_1, \langle \langle \pi_1 \cdot \pi_2 \cdot \pi_1, \pi_2 \rangle, \langle \pi_2 \cdot \pi_2 \cdot \pi_1, \pi_2 \rangle \rangle \rangle \\ aux2 = \langle \pi_1, \langle \pi_1 \cdot \pi_2, \langle \pi_1 \cdot \pi_2 \cdot \pi_2, calcOriginsAux2 \rangle \rangle \rangle \\ calcOriginsAux2 = aplicaCalc \cdot \langle \pi_1, \langle \pi_2 \cdot \pi_2, dimen \cdot \pi_1 \cdot \pi_1 \rangle \cdot \pi_2 \rangle \\ aplicaCalc\ (t, (o, c)) = calc\ t\ o\ c \end{array}}$$

Assim sendo, podemos definir a função *calcOrigins* como:

$$\boxed{calcOrigins = anaL2D\ (calcOriginsAux \cdot distl \cdot \langle outL2D \cdot \pi_1, \pi_2 \rangle)}$$

caixasAndOrigin2Pict

Uma vez que temos definida a função *crCaixa*, caso consigamos "juntar"cada caixa com a sua respetiva origem, podemos aplicar esta função para criar a *G.Picture* correspondente. Ora, uma vez que temos as funções *calcOrigins* e *agrupcaixas* definidas, conseguimos percorrer toda a árvore de caixas, e juntar cada caixa com a sua origem respetiva, criando uma lista de pares *(Origem, Caixa)*. No entanto, a função *agrupcaixas* aplica um *swap* que neste caso será desnecessário; mais ainda, as funções são baseadas num anamorfismo e num catamorfismo, respetivamente, o que nos leva a pensar que as poderemos juntar num hilomorfismo. Como o tipo de entrada de *calcOrigins* é o mesmo que o de *caixasAndOrigin2Pict* (*(L2D,Origem)*), o tipo de saida de *calcOrigins* é o mesmo que o tipo de entrada de *agrupcaixas* (*X (Caixa, Origem) ()*), e o tipo de saída de *agrupcaixas*, se não considerarmos o *swap*, é *[(Caixa, Origem)]*, podemos criar o seguinte hilomorfismo.

$$\boxed{hilo = hyloL2D\ ([singl, \mathsf{conc} \cdot \pi_2])\ (calcOriginsAux \cdot distl \cdot \langle outL2D \cdot \pi_1, \pi_2 \rangle)}$$

Posto isto, precisamos agora de aplicar a função *crCaixa* a cada par, por forma a criar uma lista de *G.Picture*. Podemos fazê-lo, utilizando a seguinte função, que recebe um par *(Caixa, Origem)*:

$$\boxed{criarPic\ (((x,y),(t,c)),o) = crCaixa\ o\ (fromIntegral\ x)\ (fromIntegral\ y)\ t\ c}$$

Neste caso, o par *(x,y)* representa as dimensões da caixa, *t* representa o texto escrito nela, *c* a cor usada para preencher a caixa, e *o* a origem. Assim sendo, aplicando esta função a todos os elementos da lista de pares, criamos uma lista de *G.Picture*. Havendo uma função *G.pictures* que transforma uma lista de figuras numa só figura, podemos definir *caixasAndOrigin2Pict* como:

$$\boxed{caixasAndOrigin2Pict = (G.pictures) \cdot (\mathsf{map}\ \ criarPic) \cdot hilo}$$

mostracaixas

Aproveitando a função *caixasAndOrigin2Pict*, que transforma um *(L2D, Origem)* numa *G.Picture*, e a função *display*, que transforma uma figura em *IO ()*, podemos definir a função *mostracaixas* como:

$$\boxed{mostra\_caixas = display \cdot caixasAndOrigin2Pict}$$

Solução

$$\begin{array}{l}
inL2D :: a + (b, (X\ a\ b, X\ a\ b)) \to X\ a\ b \\
inL2D\ (i_1\ a) = Unid\ a \\
inL2D\ (i_2\ (b,(a,c))) = Comp\ b\ a\ c \\
outL2D :: X\ a\ b \to a + (b, (X\ a\ b, X\ a\ b)) \\
outL2D\ (Unid\ a) = i_1\ a \\
outL2D\ (Comp\ b\ a\ c) = i_2\ (b,(a,c))
\end{array}$$

$$
\begin{array}{l}
baseL2D\ h\ f\ g = h + f \times (g \times g)\\
recL2D\ f = baseL2D\ id\ id\ f\\
cataL2D\ g = g \cdot recL2D\ (cataL2D\ g) \cdot outL2D\\
anaL2D\ g = inL2D \cdot recL2D\ (anaL2D\ g) \cdot g\\
hyloL2D\ c\ d = c \cdot recL2D\ (cataL2D\ c) \cdot recL2D\ (anaL2D\ d) \cdot d\\
collectLeafs = cataL2D\ [singl, \mathsf{conc} \cdot \pi_2]\\
dimen :: X\ Caixa\ Tipo \to (Float, Float)\\
dimen = cataL2D\ [(fromIntegral \times fromIntegral) \cdot \pi_1, aux]\\
\quad \textbf{where}\ aux = (cond\ (detTipo \cdot \pi_1)\ ((\widehat{max} \times \widehat{(+)}) \cdot \pi_2)\ ((\widehat{(+)} \times \widehat{max}) \cdot \pi_2)) \cdot get\\
\qquad get = id \times \langle\langle \pi_1 \cdot \pi_1, \pi_1 \cdot \pi_2\rangle, \langle \pi_2 \cdot \pi_1, \pi_2 \cdot \pi_2\rangle\rangle\\
\qquad detTipo\ V = True\\
\qquad detTipo\ Ve = True\\
\qquad detTipo\ Vd = True\\
\qquad detTipo\ H = False\\
\qquad detTipo\ Hb = False\\
\qquad detTipo\ Ht = False\\
agrup\_caixas :: X\ (Caixa, Origem)\ () \to Fig\\
agrup\_caixas = (\mathsf{map}\ \ swap) \cdot collectLeafs\\
calcOrigins :: ((X\ \ Caixa\ \ Tipo), Origem) \to X\ (Caixa, Origem)\ ()\\
calcOrigins = anaL2D\ (calcOriginsAux \cdot distl \cdot \langle outL2D \cdot \pi_1, \pi_2\rangle)\\
calcOriginsAux = [i_1, i_2 \cdot \langle bang \cdot \pi_1, \pi_2\rangle \cdot aux2 \cdot aux1]\\
\quad \textbf{where}\ aux1 = \langle \pi_1 \cdot \pi_1, \langle\langle \pi_1 \cdot \pi_2 \cdot \pi_1, \pi_2\rangle, \langle \pi_2 \cdot \pi_2 \cdot \pi_1, \pi_2\rangle\rangle\rangle\\
\qquad aux2 = \langle \pi_1, \langle \pi_1 \cdot \pi_2, \langle \pi_1 \cdot \pi_2 \cdot \pi_2, calcOriginsAux2\rangle\rangle\rangle\\
calcOriginsAux2 = aplicaCalc \cdot \langle \pi_1, \langle \pi_2 \cdot \pi_2, dimen \cdot \pi_1 \cdot \pi_1\rangle \cdot \pi_2\rangle\\
\quad \textbf{where}\ aplicaCalc\ (t, (o, c)) = calc\ t\ o\ c\\
calc :: Tipo \to Origem \to (Float, Float) \to Origem\\
calc\ V\ (a, b)\ (x, y) = (a + (x\ /\ 2), b + y)\\
calc\ Vd\ (a, b)\ (x, y) = (a + x, b + y)\\
calc\ Ve\ (a, b)\ (x, y) = (a, b + y)\\
calc\ H\ (a, b)\ (x, y) = (a + x, b + (y\ /\ 2))\\
calc\ Ht\ (a, b)\ (x, y) = (a + x, b + y)\\
calc\ Hb\ (a, b)\ (x, y) = (a + x, b)\\
caixasAndOrigin2Pict = (G.pictures) \cdot (\mathsf{map}\ \ criarPic) \cdot (hyloL2D\ (agrup)\ (origens))\\
\quad \textbf{where}\ agrup = [singl, \mathsf{conc} \cdot \pi_2]\\
\qquad origens = (calcOriginsAux \cdot distl \cdot \langle outL2D \cdot \pi_1, \pi_2\rangle)\\
\qquad criarPic\ (((x, y), (t, c)), o) = crCaixa\ o\ (fromIntegral\ x)\ (fromIntegral\ y)\ t\ c\\
mostra\_caixas :: (L2D, Origem) \to \mathsf{IO}\ ()\\
mostra\_caixas = display \cdot caixasAndOrigin2Pict
\end{array}
$$

Problema 3

Uma vez que o cosseno de um ângulo pode ser dado pelo somatório

$$cos\ x = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i)!} * x^{2i}$$

podemos fazer um cálculo deste para um dado n, ao invês de infinito, por forma a permitir a sua determinação aproximada.

$$cosFst\ x\ n = \sum_{i=0}^{n} \frac{(-1)^i}{(2i)!} * x^{2i}$$

Deste modo, podemos concluir que:

$$cosFst\ x\ 0 = 1$$

$$cosFst\ x\ (n+1) = (cosFst\ x\ n) + \frac{(-1)^{n+1}}{(2(n+1))!} * x^{2(n+1)}$$

Se definirmos agora uma nova função *cosSnd* como sendo

$$cosSnd\ x\ n = \frac{(-1)^{n+1}}{(2(n+1))!} * x^{2(n+1)}$$

teremos *cosFst* e *cosSnd* em recursividade mútua, tal que:

$$cosFst\ x\ 0 = 1$$

$$cosFst\ x\ (n+1) = (cosFst\ x\ n) + (cosSnd\ x\ n)$$

$$cosSnd\ x\ 0 = \frac{-1}{2} * x^2$$

$$cosSnd\ x\ (n+1) = \frac{(-1)^{n+1+1}}{(2(n+1+1))!} * x^{2(n+1+1)}$$

Desenvolvendo a expressão de *cosSnd* chegamos à seguinte definição:

$$cosSnd\ x\ 0 = \frac{-1}{2} * x^2$$

$$cosSnd\ x\ (n+1) = (cosSnd\ x\ n) * \frac{(-1) * x^2}{4n^2 + 14n + 12}$$

Considerando agora que existe uma função *cosTrd*, expressa da seguinte maneira

$$cosTrd\ n = 4n^2 + 14n + 12$$

podemos chegar a:

$$cosTrd\ 0 = 12$$

$$cosTrd\ (n+1) = (cosTrd\ n) + 8n + 18$$

Se agora criarmos uma última função, *cosFth*, que é definida como se segue

$$cosFth\ n = 8n + 18$$

chegamos à conclusão de que

$$cosFth\ 0 = 18$$

$$cosFth\ (n+1) = (cosFth\ n) + 8$$

Posto tudo isto, criamos 4 funções mutuamente recursivas, desenvolvendo um caso de recursividade múltipla, podendo combiná-las da seguinte forma:

$$cosFst\ x\ 0 = 1$$

$$cosFst\ x\ (n+1) = (cosFst\ x\ n) + (cosSnd\ x\ n)$$

$$cosSnd\ x\ 0 = \frac{-1}{2} * x^2$$

$$cosSnd\ x\ (n+1) = (cosSnd\ x\ n) * \frac{(-1) * x^2}{(cosTrd\ n)}$$

$$cosTrd\ 0 = 12$$

$$cosTrd\ (n+1) = (cosTrd\ n) + (cosFth\ n)$$

$$cosFth\ 0 = 18$$

$$cosFth\ (n+1) = (cosFth\ n) + 8$$

Solução

Deste modo, e usando a *regra de algibeira* que nos foi apresentada, podemos definir a função *cos'* como o seguinte ciclo *for*:

$$\begin{array}{l} cos'\ x = prj \cdot \mathsf{for}\ loop\ init\ \mathbf{where} \\ \quad loop\ (cFs, cS, cT, cFt) = (cFs + cS, cS * ((-1) * (x \uparrow 2)\ /\ (cT)), cT + cFt, cFt + 8) \\ \quad init = (1, (-0.5) * (x \uparrow 2), 12, 18) \\ \quad prj\ (cFs, cS, cT, cFt) = cFs \end{array}$$

Valorização

Tendo em vista a valorização proposta para este problema, e baseando-nos no ciclo *for* obtido anteriormente, em linguagem Haskell, definimos a seguinte função *mycos*, em linguagem C, em que o argumento *x* é o valor do ângulo cujo cosseno pretendemos obter, e *n* é o número de iterações que o ciclo *for* deve correr (quanto maior o número de iterações, maior a precisão do valor do cosseno obtido).

```
double mycos (double x, int n){
  double fst = 1, snd = (-0.5)*(x*x), trd = 12, fth = 18;
  for(int i = 0; i < n; i++){
    fst += snd;
    snd *= ((-1)*(x*x)/(trd));
    trd += fth;
    fth += 8;
  }
  return fst;
}
```

Através da execução do código e análise dos resultados obtidos, concluímos que a partir das 10 iterações, ou seja, para valores de *n* superiores ou iguais a 10, começamos a obter valores de cosseno próximos do esperado.

Concluímos, também, que dificilmente chegaríamos a esta resolução apenas por intuição, sendo necessária a determinação das expressões auxiliares que levam à construção da função.

Problema 4

outNode

$$
\begin{aligned}
& outNode\ .\ [File,\ Dir] = id \\
\equiv \quad & \quad \{\ \text{Fusão-+ (20)}\ \} \\
& [outNode\ .\ File,\ outNode\ .\ Dir] = id \\
\equiv \quad & \quad \{\ \text{Universal-+ (17)}\ \} \\
& \left\{ \begin{array}{l} id \cdot i_1 = outNode \cdot File \\ id \cdot i_2 = outNode \cdot Dir \end{array} \right. \\
\equiv \quad & \quad \{\ \text{Natural-id (1), Igualdade Extensional (69), Def-comp (70)}\ \} \\
& \left\{ \begin{array}{l} outNode\ (File\ b) = i_1\ b \\ outNode\ (Dir\ f) = i_2\ (outFS\ f) \end{array} \right.
\end{aligned}
$$

outFS

$$
\begin{aligned}
& outNode\ .\ FS\ .\ map\ (id\ \times inNode) = id \\
\equiv \quad & \quad \{\ \} \\
& outNode\ .\ FS = map\ (id\ \times outNode) \\
\equiv \quad & \quad \{\ \text{Igualdade Extensional (69), Def-comp (70)}\ \} \\
& outNode\ (FS\ l) = map\ (id\ \times outNode)\ l
\end{aligned}
$$

baseFS e recFS

$$
\begin{tikzcd}
FS \arrow[r, phantom, "\cong"] \arrow[r, bend left, "outFS"] \arrow[d, "h"'] & (A \times (B + FS))* \arrow[l, bend left, "inFS"] \arrow[d, "\mathsf{map}\ \ (f \times (g+h))"] \\
C & (A' \times (B' + C))* \arrow[l]
\end{tikzcd}
$$

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

$$
\begin{aligned}
& recFS\ f = \mathsf{map}\ \ (id \times (id + h)) \\
\equiv \quad & \quad \{\ \text{Aplicando a definição dada de baseFS}\ \} \\
& recFS\ f = baseFS\ id\ id\ f
\end{aligned}
$$

cataFS

$$\begin{tikzcd}
FS \arrow[rr, bend left, "outFS"] \arrow[rr, phantom, "\cong"] \arrow[d, "h=\llparenthesis g \rrparenthesis"'] & & (A \times (B + FS))* \arrow[ll, bend left, "inFS"] \arrow[d, "\mathsf{map}\ \ (id \times (id + h))"] \\
C & & (A \times (B + C))* \arrow[ll, "g"]
\end{tikzcd}$$

$\equiv$ { Cancelamento-cata (44) }

$\llparenthesis g \rrparenthesis \ . \ in = g \ . \ F \llparenthesis g \rrparenthesis$

$\equiv$ { in.out = id }

$\llparenthesis g \rrparenthesis = g \ . \ F \llparenthesis g \rrparenthesis \ . \ out$

$\equiv$ { Aplicando as definições em Haskell já determinadas }

$cataFS \ g = g \ . \ recFS(cataFS \ g) \ . \ outFS$

anaFS

$\equiv$ { Cancelamento-ana (53) }

$out \ . \ \llbracket g \rrbracket = F \llbracket g \rrbracket \ . \ g$

$\equiv$ { in.out = id }

$\llbracket g \rrbracket = in \ . \ F \llbracket g \rrbracket \ . \ g$

$\equiv$ { Aplicando as definições em Haskell já determinadas }

$anaFS \ g = inFS \ . \ recFS(anaFS \ g) \ . \ g$

hyloFS

$\equiv$ { Definição de hilomorfismo }

$hyloFS \ g \ h = \llparenthesis g \rrparenthesis . \llbracket h \rrbracket$

$\equiv$ { Cancelamento-cata (44); Cancelamento-ana(53) }

$hyloFS \ g \ h = g.F \llparenthesis g \rrparenthesis .inFS.outFS.F \llbracket h \rrbracket .h$

$\equiv$ { in.out = id; Aplicando as definições em Haskell já determinadas }

$hyloFS \ g \ h = g.recFS(cataFSg).recFS(anaFSh).h$

untar

Uma vez que o tipo de entrada desta função se trata de uma lista, é plausível considerar que possamos resolver o problema com o recurso a um catamorfismo de listas. Como tal, podemos chegar ao seguinte diagrama representativo de *untar*:

$$
\begin{tikzcd}
(Path \times B)* \arrow[rr, bend left, "outList"] \arrow[d, "untar"'] & \cong & 1 + ((Path \times B) + (Path \times B)*) \arrow[ll, bend left, "inList"] \arrow[d, "id+(id \times untar)"] \\
FS & & 1 + ((Path \times B) \times FS) \arrow[ll, "g"]
\end{tikzcd}
$$

Deste modo, fica agora a necessidade de definir o gene g. A utilidade desta função passa pela transformação da lista de pares *(Path a, b)* num único *FS*. Como tal, através da interpretação do diagrama do catamorfismo de listas, chegamos à conclusão de que a cauda da lista será transformada num *FS*, ficando a restar apenas a necessidade de transformar a cabeça da lista num *FS* tambem, e de alguma forma juntar as duas num único *FS*. Ora, recorrendo as funções auxiliares disponibilizadas, podemos chegar ao seguinte diagrama do gene g.

$$
\begin{tikzcd}
1 + ((Path \times B) \times FS) \arrow[d, "id+((createFSfromFile) \times id)"'] \\
1 + (FS \times FS) \arrow[d, "id+(joinDupDirs.concatFS)"'] \\
1 + FS \arrow[d, "{[inFS \cdot nil, id]}"'] \\
FS
\end{tikzcd}
$$

A função *createFSfromFile* transforma um par *(Path a, b)* no *FS* correspondente, enquanto que a função *concatFS* concatena dois *FS* num só. Recorre-se, por fim, à função *joinDupDirs*, por forma a juntar diretorias com o mesmo nome, presentes na mesma pasta, numa só diretoria, mantendo todos os ficheiros. Deste modo, podemos simplificar o gene da seguinte forma:

$$
\begin{aligned}
& g = [inFS.nil,\ id].(id\ +\ (joinDupDirs.concatFS)).(id\ +\ ((createFSfromFile)\ \times id)) \\
\equiv \quad & \qquad \{ \text{ Absorção-+ (22); Natural-id (1) } \} \\
& g = [inFS.nil,\ joinDupDirs.concatFS.(createFSfromFile \times id)]
\end{aligned}
$$

Por fim, podemos concluir que a função *untar* poderá ser definida como:

$$
\boxed{untar = cataList\ [inFS \cdot nil, joinDupDirs \cdot concatFS \cdot (createFSfromFile \times id)]}
$$

Solução

Triologia "ana-cata-hilo":

$$outFS\ (FS\ l) = \mathsf{map}\ \ (id \times outNode)\ l$$
$$outNode\ (File\ b) = i_1\ b$$
$$outNode\ (Dir\ f) = i_2\ f$$
$$baseFS\ f\ g\ h = \mathsf{map}\ \ (f \times (g + h))$$
$$cataFS :: ([(a, b + c)] \rightarrow c) \rightarrow FS\ \ a\ \ b \rightarrow c$$
$$cataFS\ \ g = g \cdot recFS\ \ (cataFS\ \ g) \cdot outFS$$
$$anaFS :: (c \rightarrow [(a, b + c)]) \rightarrow c \rightarrow FS\ \ a\ \ b$$
$$anaFS\ \ g = inFS \cdot recFS\ \ (anaFS\ \ g) \cdot g$$
$$hyloFS\ \ g\ \ h = g \cdot recFS\ \ (cataFS\ \ g) \cdot recFS\ \ (anaFS\ \ h) \cdot h$$

Outras funções pedidas:

$$check :: (Eq\ \ a) \Rightarrow FS\ \ a\ \ b \rightarrow Bool$$
$$check = \bot$$
$$tar :: FS\ \ a\ \ b \rightarrow [(Path\ \ a, b)]$$
$$tar = \bot$$
$$untar :: (Eq\ \ a) \Rightarrow [(Path\ \ a, b)] \rightarrow FS\ \ a\ \ b$$
$$untar = cataList\ [inFS \cdot nil, joinDupDirs \cdot concatFS \cdot (createFSfromFile \times id)]$$
$$find :: (Eq\ \ a) \Rightarrow a \rightarrow FS\ \ a\ \ b \rightarrow [Path\ \ a]$$
$$find = \bot$$
$$new :: (Eq\ \ a) \Rightarrow Path\ \ a \rightarrow b \rightarrow FS\ \ a\ \ b \rightarrow FS\ \ a\ \ b$$
$$new\ \ a\ \ b\ f = \bot$$
$$cp :: (Eq\ \ a) \Rightarrow Path\ \ a \rightarrow Path\ \ a \rightarrow FS\ \ a\ \ b \rightarrow FS\ \ a\ \ b$$
$$cp = \bot$$
$$rm :: (Eq\ \ a) \Rightarrow (Path\ \ a) \rightarrow (FS\ \ a\ \ b) \rightarrow FS\ \ a\ \ b$$
$$rm = \bot$$
$$auxJoin :: ([(a, b + c)], d) \rightarrow [(a, b + (d, c))]$$
$$auxJoin = \bot$$
$$cFS2Exp :: a \rightarrow FS\ \ a\ \ b \rightarrow (Exp\ ()\ \ a)$$
$$cFS2Exp = \bot$$

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.