FAQs

Q1 - *Estou com dúvidas no exercício na última questão do teste de 2010/11: não tenho ideia de como pegar!*

R: Não é de admirar: este ano não demos esta matéria, que corresponde à secção 4.10 do capítulo sobe mónades. Mas não é difícil e, para quem estiver interessado, há também estes slides: EasyMonads.

Q2 - *Estou sem conseguir resolver a questão 8 do teste do ano passado: definir um anamorfismo de naturais como um catamorfismo de listas. Tentei usar a lei universal-ana(44) mas fiquei bloqueado a meio.*

R: Sim, universal-ana (naturais) é bom começo (expandindo out = [nil,cons]º):

f = [((id+p2). [nil,cons]º)]

== { universal-ana (naturais); álgebra in dos naturais é [0,succ] }

f = in. (id + f) . (id+p2). [nil,cons]º

== { passando isomorfismo [nil,cons]º para o outro lado, "trocando o sinal" }

f . [nil,cons] = in. (id + f) . (id+p2)

== { ... }

Aqui começa a preparação das coisas para termos f como catamorfismo de listas: fica como exercício.

A dificuldade desta questão é que começa com um anamorfismo de naturais (F f = id + f) e termina com um catamorfismo de listas (F f = id + id x f). A mudança de F dá-se a partir do ponto em que se parou acima. Como sempre, as propriedades naturais não se devem ignorar (neste caso a do p2).

Q3 - *Na questão 7 do exame de recurso do ano passado não percebo a primeira parte: instanciar com a função fac a função lms.*

R: O que se pretende é resolver a equação

fac = lms p g h i j

em ordem às variáveis p, g, h, i, j - o que é a mesma coisa que encontrar essas funções na equação

fac = p -> g, h.<i,fac.j>

Partimos da definição de fac sem variáveis (onde 1 representa a função constante 1, (const 1)):

fac.in = [1,mul.<succ,fac>]

== { passagem de in para o outro lado (isomorfismo) ; out dos naturais }

fac = [1,mul.<succ,fac>].((=0) -> i1.!,i2.pred)

== { lei do condicional que facilmente se identifica }

fac = (=0) -> [1,mul.<succ,fac>].i1.!, [1,mul.<succ,fac>].i2.pred

== { cancelamento-+ ; função constante 1 }

fac = (=0) -> 1, mul.<succ,fac>.pred

== { Fusão-x ; succ.pred = id, para argumentos > 0 }

fac = (=0) -> 1, mul.<id,fac.pred>

Comparando, obtém-se:

p= (=0)

g= 1 (f. constante 1)

h=mul

i=id

j=pred

e assim

fac = lms (=0) (const 1) mul id pred

Sugestão: corram esta versão no interpretador de Haskell.

Q4 - *Tenho dificuldades ao provar a igualdade* ***f.g=id*** *na questão 6b da Ficha 2.*

R: Ver resolução do miniteste, onde se faz essa prova para o caso dual, com injecções e *eithers* em lugar de projecções e *splits*: o argumento é em tudo semelhante.

Q5 - *Tenho dificuldades a resolver a questão 6 da Ficha 3.*

R: O diagrama representa a equação

[x,y] = [k,g].(id+id >< f)

que se quer resolver em ordem a x e y. Por absorção (etc), o lado direito fica [k,g.(id >< f)] ; pela igualdade de *eithers* tem-se, de imediato,

x = k
y = g.(id >< f)

Quanto aos pontos de interrogação, comecemos pelo tipo genérico da seta vertical, que é a soma de uma identidade com um produto:

A + (B >< C) <-- **id + (id >< f)** -- A + (B >< D) (repetimos A e B por estarem ligados a identidades). Como nada sabemos sobre k e g, ter-se-á

E <--- **[k,g]** --- A + (B >< C)

Logo, os pontos de interrogação são os tipos genéricos (paramétricos) A + (B >< D) (em cima); E (em baixo, à esquerda) e A + (B >< C) (à direita).

Q6 - *Na questão 2a da Ficha 5, o isomorfismo pretendido parte de uma soma para um produto, logo posso usar tanto um split como um either. Qual devo escolher?*

R: Pela lei da troca, as suas soluções são idênticas. (Se se pretender depois escrever a função em Haskell, a conversão é mais simples se se optar pelo combinador *either*). Detalha-se a seguir essa versão: designemos o isomorfismo por **iso**, que vamos decompor em **iso = [f,g]**, com tipos, **f : A >< B -> A >< (B+1)** e **g : A -> A >< (B+1)**. Tanto **f** como **g** vão para produtos e como tal serão *splits*. Começando por **g = <g1,g2>**, **g1** = **id** e **g2** tem tipo **A -> B+1**. Como nada sabemos sobre **B**, **g2** terá que dar o seu resultado em **1**, injectando-o à direita: **g2 = i2.!**. Um processo semelhante decomporá **f** em **id >< i1**. Juntando tudo:

iso = [(id >< i1),<id,i2.!>]

Q7 - *No exercício 7 (parte 2 do teste de 2010/2011) tentei chegar da função escrita em Haskell à versão point-free mas não consegui.*

R: Escolheu a versão mais complicada, já que no sentido inverso é mais simples: faça absorção-cata no lado direito para obter

hwmny p = ([g])

(aqui o objectivo é calcular o gene **g** do catamorfismo). Depois, dessa equação, deriva **hwmny p** usando a lei universal-cata e introduzindo variáveis no fim. Importante: como o tipo em causa é **LTree**, **in = [Leaf, Fork]** e o functor de base a usar na lei de absorção-cata é **B(f,g) = f + g >< g** - ver *baseLTree* no ficheiro LTree.hs.

Q8 - *Não percebo bem o que se pretende na questão 4 da Ficha nº 5.*

R: A questão resume-se a tentar calcular o tipo polimórfico do combinador **comb f g**, para qualquer **f** e **g** à entrada. Temos que **comb f g = <f,[g,f]>**. Sejam então dadas **f : A->B** e **g : C -> D**. Ter-se-á de imediato: **[g,f] : (C + A) -> B**, já que **B=D** na construção do *either*.

Agora derive-se o tipo do *split* **<f,[g,f]>**. Para isso é preciso que **f** e **[g,f]** tenham o *mesmo* tipo de entrada, isto é, terá que verificar-se:

A = C + A

Ora esta equação define **A** como um tipo recursivo: mas, se substituirmos **A** por **C+A** e assim sucessivamente, teremos **A = C + (C + (C + ...))**, isto é, **A** é um tipo infinito: é esta, assim, a explicação para a mensagem de erro que o interpretador de Haskell dá: *"cannot construct the infinite type: a = Either c a"*.

Q9 - *Na questão 5(a) da Ficha nº 5 eu escrevi* ***Maybe A*** *isomorfo a* ***A + 1*** *e na 5(b) escrevi* ***1+1*** *isomorfo a* ***Bool****, mas não sei prosseguir com nenhum dos exercícios.*

R: Para acabar os exercícios tem que se lembrar da estrutura dos tipos **Maybe** e **Bool** em Haskell: **data Maybe a = Just a | Nothing** e **data Bool = False | True**. Repare nos tipos: **Just :: a -> Maybe a** e **const Nothing :: () -> Maybe a**, **const True :: () -> Bool** e **const False :: () -> Bool**. Usando estas funções constantes é fácil definir os isomorfismos pedidos sob a forma de *eithers*.

Q10 - *Na questão 4 da Ficha nº6 eu substitui as três ocorrências de* ***exp*** *pela sua definição mas não sei prosseguir...*

R: Sabendo-se que **exp f = curry (f.ap)**, é conveniente olharmos para as leis da exponenciação primeiro antes de fazermos substituições: para a lei fusão-exp (31) ser útil, basta começar por expandir apenas **exp f**:

(exp f) . (exp g) = exp(f.g)
== { definição de *exp f*}
(curry (f.ap)) . (exp g) = exp(f.g)
== { fusão-exp }
curry(f.ap.((exp g)><id)) = exp(f.g)
== { definição de *exp g*}
curry(f.ap.((curry (g.ap))><id)) = exp(f.g)
== { cancelamento-exp }
curry(f.(g.ap)) = exp(f.g)
== { composição é associativa ; definição de exp }
exp(f.g) = exp(f.g)

Q11 - *Tentei resolver o exercicio 6 do exame de 2012 mas não chego ao resultado que era pretendido. De* ***tri f = (| in . B(id,T f) |)*** *inferi, por cancelamento-cata,* ***(tri f) . in = in . B(id,T f) . F(tri f)****. Sabendo que* ***B(id,T f) = id + id >< T f*** *e* ***F (tri f) = id + id >< tri f*** *(listas) chego a* ***[tri f . nil, tri f . cons] = [nil, cons . (T f x tri f)]*** *de onde não consigo chegar ao código Haskell dado.*

R: O engano está no cálculo da composição

B(id,T f) . F (tri f)

que deverá dar

id + id >< (T f . (tri f))

e não

id + (T f) >< (tri f)

Resolvido este engano, deverá ser imdediato.

Q12 - *No exercicio 5 da ficha 12, em que* ***join = (| [id, Fork] |)****, pede-se para demonstrar as leis da multiplicação e unidade do monáde* ***LTree****, sendo u ->* ***Leaf*** *e mu ->* ***join****. Usando universal-cata e decompondo, chega-se a* ***join . Leaf = id****, que prova a lei* ***mu.u = id****. Mas não consigo chegar a* ***mu.mu = mu . (T mu)****.*

R: Deverá usar absorção-cata à direita seguida de fusão-cata à esquerda (tudo em **LTree**):

join.join = join. (LTree join)

== { definição de join }

join.join = (| [id, Fork] |). (LTree join)

== { absorção-cata: T f = LTree f ; B(f,g) = f + g >< g }

join.join = (| [id, Fork] . B(join,id) |)

== { B(f,id) = f + id ; definição de **join** }

join. (| [id, Fork] |) = (| [join, Fork] |)

<= { fusão cata }

join.[id, Fork] = [join, Fork] . (id + join >< join)

== { leis dos coprodutos }

[join,join.Fork] = [join, Fork . (join >< join)]

== { Eq-+ }

join = join e join.Fork = Fork . (join >< join)

== { cancelamento-cata sobre (| [id, Fork] |) }

true

Q13 - *Na página 78 do capítulo 3, onde se faz a demostração da composição de (|g|) . [(h)], diz-se que é uma função que vai de B para C. No entanto, o diagrama leva-me a crer que se trata de uma função de C para B. Está-me a escapar alguma coisa?*

R: Não está a escapar nada: é de facto uma gralha, que foi agora corrigida no original (LaTeX).

Q14 - *Não estou a conseguir resolver o exercício 8 da ficha 4, nomeadamente na prova de **map f . nil = nil**. Quais são as leis a aplicar para chegar à solução?*

R: Terá que ser a equação (7), já que nada sabemos (na pergunta) sobre **map**. E, de facto, a equação (8) corresponde ao lado esquerdo de (7) com **k=id** e **f=id**. Se fizermos a substituição, teremos o seguinte lado direito:

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

que é evidentemente verdadeira. Para verificarmos a (9), basta em (7) fazermos a substituição de **k** por **map f** no lado direito e seleccionarmos o caso **nil**:

$$(map\ f)\cdot[nil,cons]=[nil,cons]\cdot(id+f\times(map\ f))$$

Por fusão-+ no lado esquerdo e absorção-+ no direito obtemos, após eq-+, duas equações, das quais uma delas é (9). Finalmente, (10) corresponde à outra equação que resta, após introduzirmos variáveis.

Q15 - *No exercício 5 da ficha 9, após a remoção das variáveis, introdução de in, cancelamento-+ e absorção-+, não estou a conseguir obter o resultado **f1.in = ... (id + id x <f1, f2>)** e **g1.in = ... (id + id x <f1,f2>)** para depois aplicar a lei de Fokkinga.*

R: Após remoção de variáveis, as equações dadas convertem em

$$\begin{array}{c} f1.nil=nil \\ f1.cons=cons.(id \times f2) \end{array}$$

e

$$\begin{array}{c} f2.nil=nil \\ f2.cons=p2.(id \times f1) \end{array}$$

Onde está **f1** ou **f2** teremos que ter um termo que envolva $\mathbf{\langle f1,f2\rangle}$, o que não é difícil usando as leis de cancelamento-x:

$$\begin{array}{c} f1.nil=nil \\ f1.cons=cons.(id \times p2.\langle f1,f2\rangle) \end{array}$$

e

$$\begin{array}{c} f2.nil=nil \\ f2.cons=p2.(id \times p1.\langle f1,f2\rangle) \end{array}$$

Juntando cada par de equações num só, fazendo **in=[nil,cons]**, ter-se-á:

$$f1.in=[nil,cons.(id \times p2.\langle f1,f2\rangle)]$$

e

$$f2.in=[nil,p2.(id \times p1.\langle f1,f2\rangle)]$$

Falta agora factorizar a chamada recursiva $\mathbf{id + id \times \langle f1,f2\rangle}$ em ambas as equações:

$$f1.in=[nil,cons.(id \times p2)].(id + id \times \langle f1,f2\rangle)]$$

e

$$f2.in=[nil,p2.(id \times p1).(id + id \times \langle f1,f2\rangle)]$$

Agora já pode ser aplicada a lei da recursividade múltipla (acabar).

Q16 - *No exercício 2 da ficha 10, no anamorfismo **suffixes** consegui desenhar o diagrama mas não estou a conseguir descobrir o gene da função **g** para proceder ao cálculo da versão em Haskell.*

R: Uma vez mais, pegue-se na definição do gene

$$\begin{array}{c} g\ []=i1\ [] \\ g\ (h : t\ ) = i2\ (h : t\ ,\ t\ ) \end{array}$$

e retirem-se as variáveis:

$$\begin{array}{c} g.nil=i1.nil \\ g.cons=i2.\langle cons,p2\rangle \end{array}$$

já que $\mathbf{\langle cons,p2\rangle(h,t)} = \mathbf{(cons(h,t),p2(h,t))} = \mathbf{(h:t,t)}$. Daqui procede-se da mesma maneira, juntando essas equações por Eq-+:

$$g.in=[i1.nil,i2.\langle cons,p2\rangle]$$

== { isomorfismos in e out }

$$g=[i1.nil,i2.\langle cons,p2\rangle].out$$

Q16 - *No exercício 6 do teste deste ano eu faço **f = g**, universal-cata, cancelamento-cata, absorção-+, **const k . f = const k** e obtenho o resultado **[const k, const k] = [const k, f]**. O que estou a fazer de errado?*

R: Nada, só falta acabar: por cancelamento-+ obtém **const k = const k** (trivialmente verdadeiro) e **const k=f**. Isto é: a igualdade **f=g** é equivalente a **f=const k**. Por transitividade da equivalência obtém também **g=const k**, logo as duas funções são iguais à função constante **k**.

Q17 - *No exercício 4 do mesmo teste podemos partir de qualquer propriedade da exponenciação (cancelamento por exemplo)? Ou partimos da igualdade **curry f a b = f (a,b)**?*

R: A direcção do raciocínio fica à sua escolha - veja detalhes acima, na nota [What is the meaning of curry / uncurry?]

Q18 - *Tenho visto em testes de anos anteriores mais para o final uma questão do tipo: "Defina a função .... (...) como resultado da "monadificação" da função: (...) Eu tentei preparar-me para teste resolvendo todas as fichas que demos nas aulas mas não encontrei nenhuma pergunta na ficha de monades com semelhança a esta. Como tal sempre que chego aquela pergunta fico sem perceber como "pegar". Saírá algo parecido com isto no teste? Será que poderia dizer como se resolve?*

R: Essa matéria não foi dada este ano, logo não sairá nenhuma pergunta desse género.

Q19 - *O functor **F p1** é igual a **id + (p1><p1)** e o **F p2** = **id + (p2 >< p2)** ?*

R: É isso se estiver a trabalhar com LTrees, pois **B(f,g) = f + g><g** para essa estrutura e **F f = B(id,f)**. Para outras estruturas terá que ver seu functor de base **B(f,g)** e calcular o respectivo **F f**.

Q20 - **LTree p1 = (| in . (p1 + id) |)*, da pergunta 9 do teste resolvido do ano passado, foi deduzido ou trata-se de matéria teórica que devemos saber?*

R: LTree p1 = (| in . (p1 + id) |) é um caso particular de functor de tipo, deduzido a partir da sua definição no formulário **T f = (| in.B(f,id) |)**. Como se viu na FAQ anterior, **B(f,id) = f + id><id** para este tipo de árvores (T = LTree). Como **id><id=id**, o cálculo é imediato.

Q21 - *No mini-teste deste ano, na questão 6, como é que mostro que **max . const(0,0) = const 0**?*

R: Não esquecer as propriedades naturais da função constante, entre elas esta: **f.(const k) = const (f k)**. Assim, **max . const(0,0)** = **const(max(0,0))** = **const 0**.

Q22 - *Nas justificações dos passos numa prova analítica, quando introduzimos variáveis, a justificação tanto pode ser "igualdade extensional", bem como "introdução de variáveis"? Ou há uma justificação que seja mais correcta que a outra?*

R: As duas estão bem: a introdução de variáveis é equivalente à igualdade extensional.

Q23 - *Ao tentar resolver a questão 8 do este de recurso do ano passado, não consigo entender bem o que é pedido.*

R: É dado um anamorfismo **g = [(i2. <id,id>)]** e pretende-se mostrar, usando as leis dos anas, que

map f. g = g . f

Ora o anamorfismo é de listas, o que quer dizer que sabemos que: **B(f,g)=id+f><g**, **F f = B(id,f) = id + id><f** e **map f = T f**. Estamos então em condições de usar a lei de absorção-ana,

map f . [(i2. <id,id>)] = B(f,id) ....

e acabar de resolver o exercício.

Q24 - *Quando queremos identificar um isomorfismo recorrendo a diagramas, depois de construir os tipos das duas testemunhas, para provar o isomorfismo basta escrever, por exemplo,* ***(A x A)+(B x A) ~= (A+B) x A*** *no fim e fica provado?*

R: Não fica. O diagrama apenas revela os tipos (polimórficos) das duas funções. Para o isomorfismo ficar provado precisamos de mostrar que as duas funções, compostas por qualquer ordem, dão a identidade.

Q25 - *No recurso do ano passado, na questão 5, quando chegamos ao passo* ***pair p = for (<succ.p1,k p>) (<1,0>)*** *como é que introduzimos aqui as variáveis* **(x,y)***? É para introduzir dentro de cada argumento do for?*

R: Não! O **for** já está definido ao nível da variável. O que se pretende no exercício é obter **loop** e mostrar que, após introdução de variáveis (apenas em **loop**), essa função é a dada no exame.