Métodos de Programação 1

Notas Teórico-Práticas

Alcino Cunha

14 de Novembro de 2006

Conteúdo

Capítulo 1

Introdução aos *folds*

1.1 Motivação

Considere as seguintes funções definidas recursivamente sobre listas. Dada uma lista, a primeira calcula o seu somatório e a segunda inverte-a.

$$
\begin{aligned}
&sum :: [Int] \to Int \\
&sum\ [\,] \quad = 0 \\
&sum\ (h : t) = h + sum\ t \\
&reverse :: [a] \to [a] \\
&reverse\ [\,] \quad = [\,] \\
&reverse\ (h : t) = reverse\ t +\!\!+ [h]
\end{aligned}
$$

Facilmente se verifica que ambas partilham um *padrão de recursividade* muito típico: quando a lista não é vazia, começam por invocar recursivamente a função sobre a cauda da lista, sendo este resultado posteriormente combinado com a cabeça da lista por forma a produzir o resultado final. Esta semelhança é ainda mais evidente se isolarmos a função que processa o resultado recursivo e a cabeça da lista usando a notação lambda do Haskell.

$$
\begin{aligned}
&sum :: [Int] \to Int \\
&sum\ [\,] \quad = 0 \\
&sum\ (h : t) = (\lambda x\ y \to x + y)\ h\ (sum\ t) \\
&reverse :: [a] \to [a] \\
&reverse\ [\,] \quad = [\,] \\
&reverse\ (h : t) = (\lambda x\ y \to y +\!\!+ [x])\ h\ (reverse\ t)
\end{aligned}
$$

1.2 O meu primeiro *fold*

Agora é evidente que a única diferença entre estas funções, para além do nome, é a função que processa o resultado recursivo e o valor devolvido quando a lista está

vazia. Podemos então escrever uma função de ordem superior que recebe estes dois parâmetros e implementa este padrão de recusividade. Esta função chama-se *foldr* e já existe predefinida em Haskell.

$$
\begin{aligned}
&\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{foldr}\ f\ z\ [\,] \quad = z \\
&\mathit{foldr}\ f\ z\ (h : t) = f\ h\ (\mathit{foldr}\ f\ z\ t)
\end{aligned}
$$

Usando o *foldr* podemos agora redefinir as funções anteriores sem mencionar explicitamente a invocação recursiva.

$$
\begin{aligned}
&\mathit{sum} :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sum}\ l = \mathit{foldr}\ (\lambda x\ y \to x + y)\ 0\ l \\
&\mathit{reverse} :: [a] \to [a] \\
&\mathit{reverse}\ l = \mathit{foldr}\ (\lambda x\ y \to y +\!\!+ [x])\ [\,]\ l
\end{aligned}
$$

Dadas duas funções f e g, se $\forall x. f\,x = g\,x$ então $f = g$. Esta lei chama-se *igualdade extensional* e permite simplifcar as definições acima, pois as equações que as definem são obviamente válidas para qualquer l. Por exemplo, a função *sum* pode definir-se simplesmente da seguinte forma.

$$
\begin{aligned}
&\mathit{sum} :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sum} = \mathit{foldr}\ (\lambda x\ y \to x + y)\ 0
\end{aligned}
$$

Se passarmos o operador binário de soma para a notação prefixa é possível também retirar as variáveis do primeiro argumento deste *foldr*.

$$
\begin{aligned}
&\mathit{sum} :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sum} = \mathit{foldr}\ (+)\ 0
\end{aligned}
$$

Exercício 1 *Escreva a seguinte função usando o foldr. Neste caso a função que processa o resultado recursivo deverá ignorar a cabeça da lista.*

$$
\begin{aligned}
&\mathit{length} :: [a] \to \mathit{Int} \\
&\mathit{length}\ [\,] \quad = 0 \\
&\mathit{length}\ (h : t) = 1 + \mathit{length}\ t
\end{aligned}
$$

Exercício 2 *Escreva a seguinte função que concatena duas listas unsando o foldr. Note que a recursividade é feita sobre o primeiro argumento, enquanto que o segundo se comporta como uma constante. Será possível neste caso escrever uma definição sem argumentos, tal como foi feito acima para a função sum?*

$$
\begin{aligned}
&\mathit{cat} :: [a] \to [a] \to [a] \\
&\mathit{cat}\ [\,] \quad l = l \\
&\mathit{cat}\ (h : t)\ l = h : \mathit{cat}\ t\ l
\end{aligned}
$$

Como se pode facilmente verificar, dada uma lista qualquer, a função *foldr f z* substitui os contrutores (:) e [ ] pelos argumentos f e z, respectivamente.

$$foldr\ f\ z\ [x1, x2, ..., xn] = foldr\ f\ z\ (x1 : x2 : ... : xn : [\,]) = f\ x1\ (f\ x2\ ...\ (f\ xn\ z))$$

No caso de os argumentos serem outra vez os construtores das listas, o *foldr* resultante comporta-se como a função identidade.

$$foldr\ (:)\ [\,] = \mathsf{id}$$

Esta lei, denominada *reflexão*, é o primeiro exemplo de uma lei de cálculo que nos permite raciocinar e optimizar programas funcionais. Neste caso, a elegância da sua formulação foi conseguida à custa da utilização do padrão de recursividade *foldr* e da notação *point-free* (ou seja, sem variáveis). Estes dois princípios irão acompanhar-nos ao longo do semestre: sempre que quisermos calcular com uma dada função recursiva, será necessário convertê-la primeira para uma função equivalente sem recursividade explícita (usando padrões de recusividade) e sem variáveis.

1.3 *Folds* sobre árvores

Considere agora o seguinte tipo de dados para codificar árvores binárias e duas funções para determinar, respectivamente, a altura e a travessia inorder de uma árvore.

```
data Tree a = Empty | Node a (Tree a) (Tree a)

height :: Tree a → Int
height Empty        = 0
height (Node _ l r) = 1 + max (height l) (height r)

inorder :: Tree a → [a]
inorder Empty        = []
inorder (Node x l r) = (inorder l) ++ [x] ++ (inorder r)
```

Mais uma vez se verifica que ambas as funções partilham um padrão de recursividade semelhante ao do *foldr*: quando a árvore não é vazia, a função é invocada recursivamente sobre ambas as sub-árvores, sendo os dois resultados posteriormente combinados com o conteúdo do nodo por forma a produzir o resultado final. Podemos então escrever uma função de ordem superior que codifica este padrão de recursividade.

```
foldt :: (a → b → b → b) → b → Tree a → b
foldt f z Empty        = z
foldt f z (Node x l r) = f x (foldt f z l) (foldt f z r)
```

Exercício 3 *Qual será a lei de reflexão para o foldt?*

Usando o *foldt* podemos então reescrever a função *height* sem usar recursividade explícita.

```
height :: Tree a → Int
height = foldt (λ_ l r → 1 + max l r) 0
```

Exercício 4 *Codifique a função inorder usando o foldt.*

Exercício 5 *Considere o seguinte tipo de dados para codificar números naturais.*

data *Nat* = *Zero* | *Succ Nat*

Implemente a função foldn que codifica um padrão de recursividade típico para este tipo semelhante ao foldr para as listas. Escreva a sua lei de reflexão.

Exercício 6 *Implemente as seguintes funções usando o foldn.*

$$
\begin{aligned}
&\mathit{toint} :: \mathit{Nat} \to \mathit{Int}\\
&\mathit{toint}\ \mathit{Zero} = 0\\
&\mathit{toint}\ (\mathit{Succ}\ n) = 1 + \mathit{toint}\ n\\
&\mathit{soma} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}\\
&\mathit{soma}\ n\ \mathit{Zero} = n\\
&\mathit{soma}\ n\ (\mathit{Succ}\ m) = \mathit{Succ}\ (\mathit{soma}\ n\ m)
\end{aligned}
$$

1.4 Infelizmente nem tudo são *folds*

Nem todas as funções recursivas podem ser implementadas usando *folds*. No caso das listas, para que uma função possa ser definida usando o *foldr* é necessário que se verifiquem as seguintes condições:

1. O caso de paragem deve ser na lista vazia.
2. A invocação recursiva tem que ser feita sobre a cauda da lista.
3. O resultado final só pode depender do resultado recursivo e da cabeça da lista.

Exercício 7 *Será possível implementar a seguinte função com um foldr? Justifique a sua resposta.*

$$
\begin{aligned}
&\mathit{qsort} :: \mathit{Ord}\ a \Rightarrow [a] \to [a]\\
&\mathit{qsort}\ [\,] \quad = [\,]\\
&\mathit{qsort}\ (h : t) = (\mathit{qsort}\ \mathit{menores}) +\!\!+ [h] +\!\!+ (\mathit{qsort}\ \mathit{maiores})\\
&\quad \textbf{where}\quad \mathit{menores} = \mathit{filter}\ (<h)\ t\\
&\qquad\qquad\ \ \mathit{maiores} = \mathit{filter}\ (\geqslant h)\ t
\end{aligned}
$$

Exercício 8 *Será possível implementar a seguinte função com um foldn? Justifique a sua resposta.*

$$
\begin{aligned}
&\mathit{downto} :: \mathit{Nat} \to [\mathit{Nat}]\\
&\mathit{downto}\ \mathit{Zero} \quad = [\,]\\
&\mathit{downto}\ (\mathit{Succ}\ n) = \mathit{Succ}\ n : \mathit{downto}\ n
\end{aligned}
$$

Capítulo 2

Introdução aos *monads*

2.1 Motivação

Considere o seguinte tipo de dados para representar expressoões aritméticas com constantes, variáveis, produto e divisão inteira.

```
data Exp = Const Int | Var String | Prod Exp Exp | Div Exp Exp
```

Por exemplo, a expressão $3 \times (4 \div x)$ pode ser representada como

```
Prod (Const 3) (Div (Const 4) (Var "x"))
```

Para este tipo é possível declarar uma instância da classe *Show* da forma usual.

```
instance Show Exp where
  show (Const x) = show x
  show (Var x)   = x
  show (Prod l r) = "(" ++ (show l) ++ "*" ++ (show r) ++ ")"
  show (Div l r)  = "(" ++ (show l) ++ "/" ++ (show r) ++ ")"
```

Para determinar o valor de uma expressão é necessário saber qual o valor das variáveis. Para tal vamos usar um dicionário definido como uma lista de pares com o nome da variável e o respectivo valor.

```
type Dict = [(String, Int)]
```

Dado um dicionário e uma expressão podemos agora determinar o seu valor usando a seguinte função.

```
eval :: Dict → Exp → Int
eval d (Const x) = x
eval d (Var x)   = fromJust (lookup x d)
eval d (Prod l r) = (eval d l) * (eval d r)
eval d (Div l r)  = (eval d l) `div` (eval d r)
```

lookup :: (*Eq a*) ⇒ *a* → [(*a*, *b*)] → *Maybe b* é uma função parcial que, dada uma lista de pares chave e valor, procura uma dada chave, devolvendo o respectivo valor caso exista. Note a utilização do tipo *Maybe* para representar a parcialidade. *fromJust* :: *Maybe a* → *a* é uma função declarada na biblioteca *Data.Maybe* que extrai o valor de um *Maybe* caso esteja contido num *Just*. Usando esta função podemos determinar o valor da expressão acima para o caso em que *x* é 2.

```
> let exp = Prod (Const 3) (Div (Const 4) (Var "x"))
> eval [("x", 2)] exp
6
```

Infelizmente, esta função é parcial e pode terminar anormalmente com um erro se uma das variáveis não estiver no dicionário ou se o divisor for 0.

```
> eval [("y", 2)] exp
*** Exception : Maybe.fromJust : Nothing
> eval [("x", 0)] exp
*** Exception : divide by zero
```

Este problema pode ser evitado se o tipo do resultado for *Maybe Int*, devolvendo *Nothing* caso tenha ocorrido um erro. Esta nova versão pode ser codificada da seguinte forma.

```
evalm :: Dict → Exp → Maybe Int
evalm d (Const x)  = Just x
evalm d (Var x)    = lookup x d
evalm d (Prod l r) = case evalm d l
                     of Nothing → Nothing
                        Just x  → case evalm d r
                                  of Nothing → Nothing
                                     Just y  → Just (x * y)
evalm d (Div l r)  = case evalm d l
                     of Nothing → Nothing
                        Just x  → case evalm d r
                                  of Nothing → Nothing
                                     Just y  → if y ≡ 0
                                               then Nothing
                                               else Just (x `div` y)
```

Voltando ao interpretador teríamos agora

```
> evalm [("x", 2)] exp
Just 6
> evalm [("y", 2)] exp
Nothing
```

Outra possível solução para o problema consiste em usar listas de inteiros como resultado, passando a nossa função a ser não-determinista. Neste caso podemos até

devolver vários resultados se no dicionário existirem vários valores para uma variável. A situação de erro passa a ser assinalada pela lista vazia. Neste caso uma possível definição seria

$$
\begin{aligned}
&\mathit{evall} :: \mathit{Dict} \to \mathit{Exp} \to [\mathit{Int}]\\
&\mathit{evall}\ d\ (\mathit{Const}\ x) = [x]\\
&\mathit{evall}\ d\ (\mathit{Var}\ x) \quad = \mathit{procura}\ x\ d\\
&\mathit{evall}\ d\ (\mathit{Prod}\ l\ r) = [x * y \mid x \leftarrow \mathit{evall}\ d\ l, y \leftarrow \mathit{evall}\ d\ r]\\
&\mathit{evall}\ d\ (\mathit{Div}\ l\ r) \quad = [x\ `\mathit{div}`\ y \mid x \leftarrow \mathit{evall}\ d\ l, y \leftarrow \mathit{evall}\ d\ r, y \not\equiv 0]
\end{aligned}
$$

onde *procura* é uma generalização de *lookup* que devolve todos os valores de uma dada chave.

$$
\begin{aligned}
&\mathit{procura} :: \mathit{Eq}\ a \Rightarrow a \to [(a,b)] \to [b]\\
&\mathit{procura}\ x\ d = [v \mid (y,v) \leftarrow d, x \equiv y]
\end{aligned}
$$

Mais uma vez, no interpretador podemos testar a nova função.

```
> evall [("x", 2)] exp
[6]
> evall [("y", 2)] exp
[]
> evall [("x", 2), ("x", 1)] exp
[6, 12]
> evall [("x", 0), ("x", 1)] exp
[12]
```

Mesmo que para uma dada valoração a função dê um erro, as restantes valorações são na mesma consideradas no resultado final. No caso do *Maybe* apenas a primeira valoração é considerada, mesmo que dê origem a um erro.

Exercício 9 *Considere agora que para além de assinalar a ocorrência de um erro pretende também saber que erro ocorreu. Para tal, será usado o seguinte tipo de dados muito semelhante ao Maybe, mas onde o construtor que assinala o erro passa a conter uma string com a descrição do erro ocorrido.*

$$\textbf{data}\ \mathit{Erro}\ a = \mathit{Erro}\ \mathit{String} \mid \mathit{OK}\ a$$

Defina uma função evale $:: \mathit{Dict} \to \mathit{Exp} \to \mathit{Erro}\ \mathit{Int}$ *que determina o valor de uma expressão e que, caso ocorra um erro, propague a respectiva mensagem de erro para o resultado final.*

Infelizmente, pelo facto de o tipo do resultado da nossa função ter ficado mais sofisticado a complexidade das definições aumentou bastante. No caso do *evalm* é necessário testar se ocorreu um erro nos resultados recursivos antes de determinar o valor final, e no caso do *evall* temos produzir resultados finais para todas as possíveis combinações de resultados recursivos. Para além da complexidade adicional, também passa a ser dificil reconhecer a essência da função *eval* nas novas funções: a maior parte do esforço é agora dirigido para a gestão da parcialidade ou do não-determinismo.

2.2 Nem todas as funções são igualmente aplicadas

É possível evitar estes problemas se pensarmos nas novas funções de forma ligeiramente diferente. Vamos assumir que continuam a ser funções que retornam simplesmente inteiros, mas que estes inteiros são entregues dentro de uma caixa surpresa cuja forma varia. No caso do *Maybe* essa caixa pode conter o resultado esperado, mas também pode não conter nada. No caso das listas a caixa pode até conter vários resultados. No caso da nossa função original a caixa não tem surpresa nenhuma e contém sempre um resultado.

Se os valores são agora entregues em caixas, a noção fundamental de aplicar uma função a um valor deveria mudar de acordo com a forma da caixa. Se a caixa não tem surpresas podemos sempre aplicar a função ao valor lá contido. Se a forma da caixa for *Maybe* só podemos aplicar a função se existir um valor na caixa de entrada. Finalmente, no caso de a forma ser [ ] temos que aplicar a função a todos os valores lá contidos. Em Haskell não se usa um operador explícito para a aplicação de funções, sendo esse papel cumprido pelo espaço existente entre uma função e o seu argumento. No entanto, podemos declarar um operador infixo para aplicar explicitamente uma função a um argumento.

```
infix 5 ⊡↦
(⊡↦) :: a → (a → b) → b
x ⊡↦ f = f x
```

O ponto dentro da caixa serve para assinalar que ela contém sempre um valor. Usando este operador em conjunto com a notação lambda podemos redefinir a função *eval* da seguinte forma.

```
eval :: Dict → Exp → Int
eval d (Const x)  = x
eval d (Var x)    = fromJust (lookup x d)
eval d (Prod l r) = eval d l ⊡↦ (λx → eval d r ⊡↦ (λy → x ∗ y))
eval d (Div l r)  = eval d l ⊡↦ (λx → eval d r ⊡↦ (λy → x `div` y))
```

No caso da os valores estarem contidos em caixas da forma *Maybe* a aplicação é um pouco mais complexa, pois podemos não ter valor.

```
infix 5 □↦
(□↦) :: Maybe a → (a → Maybe b) → Maybe b
Nothing □↦ _ = Nothing
Just x   □↦ f = f x
```

Usando este operador podemos redefinir a função *evalm* da seguinte forma.

```
evalm :: Dict → Exp → Maybe Int
evalm d (Const x)  = Just x
evalm d (Var x)    = lookup x d
evalm d (Prod l r) = evalm d l □↦ (λx → evalm d r □↦ (λy → Just (x ∗ y)))
```

```
evalm d (Div l r)  = evalm d l □→ (λx → evalm d r □→
                       (λy → if y ≡ 0 then Nothing else Just (x `div` y)))
```

Finalmente, no caso da caixa ser uma lista temos as seguintes definições.

```
infix 5 □⇒
(□⇒) :: [a] → (a → [b]) → [b]
l □⇒ f = [y | x ← l, y ← f x]
evall :: Dict → Exp → [Int]
evall d (Const x) = [x]
evall d (Var x)   = procura x d
evall d (Prod l r) = evall d l □⇒ (λx → evall d r □⇒ (λy → [x * y]))
evall d (Div l r)  = evall d l □⇒ (λx → evall d r □⇒
                       (λy → if y ≡ 0 then [] else [x `div` y]))
```

Repare que para além de terem ficado mais simples, as novas definições passaram a ser muito semelhantes entre si e muito semelhantes à definição original. Para além da função de aplicação, a única diferença entre elas passou a ser a forma como se constroi uma caixa com um valor (no caso do *Maybe* usando o construtor *Just* e no caso das listas construindo uma lista com apenas um elemento), a forma como se constroi uma caixa vazia, e função de procura no dicionário que obviamente é a única grande diferença entre elas.

Exercício 10 *Defina um operador de aplicação para o tipo de dados Erro e reescreva a função evale usando esse operador.*

2.3 Finalmente os *monads*

Em Haskell a classe *Monad* agrupa todas os formatos de caixas para as quais a noção de aplicação de funções faz sentido. Como já vimos, o formato das caixas não é um tipo, mas um construtor de tipos como, por exemplo, *Maybe*, [ ] ou *Erro*. Sendo assim esta classe agrupa construtores de tipo e não tipos normais como, por exemplo, a classe *Show*. Os métodos da classe *Monda* são o ≻= (denominado *bind*) que codifica a aplicação de uma função a uma *computação* (denominação usada para um valor dentro de uma caixa), e o *return* que indica como se pode construir uma computação a partir de um valor normal.

```
class Monad m where
  (≻=)   :: m a → (a → m b) → m b
  return :: a → m a
```

Data esta classe podemos definir instâncias para os *monads* que já conhecemos de forma trivial.

```
instance Monad Maybe where
  (≻=) = (□→)
```

```
  return = Just
instance Monad [ ] where
  (>>=) = (□⇒)
  return = (:[ ])
```

Exercício 11 *Escreva uma instância da class Monad para o construtor de tipos Erro.*

Exercício 12 *O* monad *mais simples de todos é o* monad *identidade, ou seja aquele em as computações contém sempre um valor. Este* monad *pode definido usando o seguinte tipo.*

```
newtype Id a = Id a
```

Escreva uma instância da classe Monad para o construtor de tipos Id e reescreva a função eval usando como resultado Id Int.

Agora $\gg\!=$ e *return* passam a ser funções *overloaded*, comportando-se de forma diferente de acordo com o tipo do *monad*. Por exemplo, a função *evalm* pode ser definida da seguinte forma.

```
evalm :: Dict → Exp → Maybe Int
evalm d (Const x) = return x
evalm d (Var x)   = lookup x d
evalm d (Prod l r) = evalm d l >>= (λx → evalm d r >>= (λy → return (x * y)))
evalm d (Div l r)  = evalm d l >>= (λx → evalm d r >>=
                       (λy → if y ≡ 0 then Nothing else return (x `div` y)))
```

Uma das grandes vantagens de declarar uma instância da classe *Monad* é que passamos a poder usar também uma notação especial para escrever funções monádicas. Esta notação baseia-se na equivalência

$$d \gg\!= \lambda x \to e \equiv \textbf{do}\ \{x \leftarrow d; e\}$$

e torna explícita a noção de extraír um valor de uma computação antes de ser usado. Neste caso d é uma computação e x um dos valores que está dentro dessa computação e que poderá ser usado em e. Usando a denominada notação-do a função *evalm* pode ser reescrita de forma muito mais clara como

```
evalm :: Dict → Exp → Maybe Int
evalm d (Const x) = return x
evalm d (Var x)   = lookup x d
evalm d (Prod l r) = do x ← evalm d l
                        y ← evalm d r
                        return (x * y)
evalm d (Div l r)  = do x ← evalm d l
                        y ← evalm d r
                        if (y ≡ 0) then Nothing
                                   else return (x `div` y)
```

Da mesma forma que é possível calcular com funções normais, também podemos calcular com funções monádicas. Ao definir uma instância da class *Monad* é necessário garantir que as funções *return* e $\gg=$ satisfazem as seguintes leis.

$$(return\ x) \gg= f \equiv f\ x$$
$$m \gg= return \equiv m$$
$$(m \gg= f) \gg= g \equiv m \gg= (\lambda x \rightarrow f\ x \gg= g)$$

As duas primeiras determinam que *return* se comporte como a identidade quando usada em conjunto com $\gg=$. A última é uma espécie de associatividade para o $\gg=$. Mais tarde iremos estudar uma notação *point-free* para *monads*. Com essa notação estas leis serão muito mais claras e simples de usar.

Exercício 13 *Escreva uma função sequence* :: *Monad m* $\Rightarrow$ *[m a]* $\rightarrow$ *m [a] que dada uma lista de computações monádicas, executa todas essas computações e devolve uma computação com a lista dos resultados.*

2.4 *Monads* com zero e soma

Alguns *monads* satisafazem leis adicionais devido ao facto de as suas computações possuirem mais estrutura. Por exemplo, muitos *monads* possuem uma noção de zero (caixa vazia), usada para assinalar computações falhadas, e uma noção de soma de computações (misturar os conteúdos de duas caixas). Estes mondas estão agrupados na classe *MonadPlus*, que tem a seguinte definição.

```
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

Um exemplo clássico destes *monads* é o *Maybe*, onde a soma dá prioridade à computação da esquerda e o zero é o construtor *Nothing*.

```
instance MonadPlus Maybe where
  mzero = Nothing
  Just x    `mplus` _ = Just x
  Nothing `mplus` y = y
```

Exercício 14 *Escreva uma instância da classe MondaPlus para o* monad *das listas.*

Exercício 15 *Escreva uma generalização da função lookup que retorne computações de um qualquer MonadPlus. O tipo pretendido para esta função é*

$$lookupM :: (MonadPlus\ m, Eq\ a) \Rightarrow a \rightarrow [(a, b)] \rightarrow m\ b$$

Nos *monads* desta classe a computação pode sempre falhar. É muito útil definir uma função *guard* que testa uma dada pré-condição antes de prosseguir com o cálculo. Se a condição não for verificada é imediatamente devolvida a computação vazia.

```
guard :: MonadPlus m ⇒ Bool → m ()
guard False = mzero
guard True  = return ()
```

Esta função está pré-definida na biblioteca *Control.Monad*. Usando o *guard* em conjunto com a função *lookupM* é possível definir uma versão do *eval* genérica que funciona para qualquer *MonadPlus*.

```
eval :: MonadPlus m ⇒ Dict → Exp → m Int
eval d (Const x)  = return x
eval d (Var x)    = lookupM x d
eval d (Prod l r) = do x ← eval d l
                       y ← eval d r
                       return (x ∗ y)
eval d (Div l r)  = do x ← eval d l
                       y ← eval d r
                       guard (y ≢ 0)
                       return (x `div` y)
```

A seguinte interação no interpretador seria agora possível. Note que é necessário indicar qual o tipo pretendido para o resultado, por forma a que a instância apropriada de *MonadPlus* seja seleccionada pelo interpretador.

```
> let exp = Prod (Const 3) (Div (Const 4) (Var "x"))
> eval [("x", 2), ("x", 1)] exp :: Maybe Int
Just 6
> eval [("x", 2), ("x", 1)] exp :: [Int]
[6, 12]
```

2.5 A caixa mágica do *IO*

Em Haskell todas as operções de *input/output* tem que ser realizadas usando o *monad IO*. O *monad IO* tem várias características muito próprias que o distinguem de todos os outros monads. Em primeiro lugar, a definição deste tipo é opaca, ou seja, os seus construtores não são conhecidos.

```
data IO a
```

Este facto tem várias consequências, nomeadamente, não é possível definir directamente valores deste tipo nem usar *pattern-matching* para aceder ao seu conteúdo. Sendo assim, a implementação da instância da classe *Monad* para este tipo nunca poderá ser feita pelo utilizador, estando pré-definida nas bibliotecas do Haskell.

Outra característica importante é que um valor do tipo *IO a* é uma computação que pode realizar operações de *input/output* antes de produzir um valor do tipo *a*. Por exemplo, considere a seguinte computação.

$$getChar :: IO\ Char$$

Esta computação é usada para ler um caracter do teclado. Como já vimos, podemos vê-la como uma caixa que contém o caracter. No entanto, quando abrimos esta caixa o caracter pode não estar imediatamente disponível: só depois de utilizador pressionar uma tecla é que o respectivo caracter aparece lá dentro. De certa forma é como uma "caixa mágica", cujo conteúdo depende de acções que acontecem fora da mesma. Estas acções são normalmente designadas efeitos secundários, ou, na terminologia anglo-saxónica, *side effects*.

Uma computação do tipo $IO\ a$ contém um resultado do tipo a. O cálculo deste resultado pode implicar alguns efeitos secundários, mas existirá sempre, a não ser que a execução termine anormalmente. Sendo assim, o *monad IO* não pode pertencer à classe *MonadPlus*, pois as suas caixas nunca podem estar vazias. Muitas operações de *input/output* apenas são importantes pelos efeitos secundários que provocam, sendo o seu resultado irrelevante. Como é sempre necessário que as computações de *IO* contenham um resultado, normalmente opta-se nestes casos por devolver um resultado do tipo (). Por exemplo, considere a seguinte função para imprimir um caracter.

$$putChar :: Char \rightarrow IO\ ()$$

Embora o valor contido na computação resultante seja irrelevante, dado que até já sabemos o seu valor, para ficar disponível é necessário que antes ocorram alguns efeitos secundários, nomeadamente a desejada impressão do caracter no monitor.

Se não conhecemos os construtores do tipo *IO* como é podemos manipular os valores nele contidos? A única hipótese de o fazer é recorrendo à função $\gg=$ da classe *Monad*: dado uma computação do tipo $IO\ a$ podemos sequenciá-la com uma função do tipo $a \rightarrow IO\ b$ que processa o valor nela contido e produz uma computação do tipo $IO\ b$. A implicação mais importante deste facto é a obrigação de, uma vez criada uma computação de *IO*, processar todos os resultados consequentes usando este *monad*. Embora muitos de nós não gostem desta implicação, ela é crucial para preservar a semântica dos nossos programas em Haskell. Vamos por momentos supôr que seria possível extrair directamente o resultado contido numa computação de *IO*. De facto, até existe uma função nas bibliotecas do Haskell para o fazer:

$$unsafePerformIO :: IO\ a \rightarrow a$$

Supondo que existe também uma computação $getInt :: IO\ Int$ que lê um inteiro do teclado, podemos definir o seguinte valor:

$$\textbf{let}\ x = unsafePerformIO\ (getInt)\ \textbf{in}\ x + x$$

Devido à transparência referencial, uma das leis fundamentais da programação funcional, é possível trocar qualquer expressão por uma expressão equivalente dentro de um programa. Sendo assim, a expressão acima deveria ser equivalente à seguinte:

$$unsafePerformIO\ (getInt) + unsafePerformIO\ (getInt)$$

No entanto, se executarmos estas expressões no nosso interpretador o seu resultado poderá ser bastante diferente. Se não forem efectuadas optimização pelo interpretador,

na primeira será lido um inteiro e calculado o seu dobro, enquanto que na segunda serão lidos e somados dois inteiros diferentes. Esta discrepância é inaceitável: para além de tornar imprevisíveis os resultados de qualquer programa com operações de *input/output*, torna impossível raciocinar sobre estes programas usando leis de cálculo. Neste caso, se o efeito pretendido é ler um inteiro e calcular o seu dobro devemos usar a seguinte expressão:

$$getInt \gg\!= \lambda x \rightarrow return\ (x + x)$$

Note que o seu resultado já não pode ser do tipo *Int*, pois o segundo argumento do $\gg\!=$ tem que ser uma função cujo resultado está contido numa computação de *IO*. Daí a utilização da função *return* para colocar o valor $x + x$ dentro de uma computação de tipo *IO Int*. No caso de pretendermos somar dois inteiros diferentes devemos usar a seguinte expressão:

$$getInt \gg\!= \lambda x \rightarrow getInt \gg\!= \lambda y \rightarrow return\ (x + y)$$

Estas duas expressões são agora incomparáveis e possuem comportamentos bem distintos. A utilização do $\gg\!=$ força-nos a ser precisos quanto à quantidade e sequência das operações de *input/output* a realizar.

Usando o *monad IO* e um conjunto bastante limitado de operações de *input/output* primitivas é possível definir um reportório de funções muito úteis. Por exemplo uma função que imprime uma string pode ser definida recursivamente da seguinte forma.

$$\begin{aligned} &putStr :: String \rightarrow IO\ () \\ &putStr\ [\,] \quad\ = return\ () \\ &putStr\ (h : t) = putChar\ h \gg\!= (\lambda\_ \rightarrow putStr\ t) \end{aligned}$$

Neste caso o operador $\gg\!=$ está a ser usado apenas para sequenciar as operações de *input/output*: o resultado da primeira operação é ignorado pela segunda função. Este padrão ocorre frequentemente quando se encadeiam computações de tipo *IO* (). Para facilitar a implementação destes casos, existe pré-definido em Haskell um operador que sequencia duas computações monádicas, ignorando o resultado da primeira.

$$\begin{aligned} &(\gg) :: Monad\ m \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ b \\ &f \gg g = f \gg\!= (\lambda\_ \rightarrow g) \end{aligned}$$

Usando este operador podemos redefinir a função acima da seguinte forma.

$$\begin{aligned} &putStr :: String \rightarrow IO\ () \\ &putStr\ [\,] \quad\ = return\ () \\ &putStr\ (h : t) = putChar\ h \gg putStr\ t \end{aligned}$$

Exercício 16 *Defina uma computação getLine*::*IO String que lê uma string do teclado. Para tal, deverá ler sucessivamente caracteres do teclado, usando o getChar, até que um* `'\n'` *seja pressionado.*

Exercício 17 *Usando a função read e o getLine, definido no exercício anterior, defina uma computação getInt* :: *IO Int que lê um inteiro do teclado.*

Exercício 18 *Relembre o tipo Exp declarado na secção 2.1.*

```
data Exp = Const Int | Var String | Prod Exp Exp | Div Exp Exp
```

Como já vimos, é possível definir uma função que avalia uma expressão, desde que exista um dicionário que determina o valor das suas variáveis. Implemente uma função evalio :: Dict → Exp → IO Int que avalia uma expressão, perguntando ao utilizador o valor de todas as variáveis que não existam no dicionário. Uma possível utilização desta função seria

```
> let exp = Prod (Const 3) (Div (Const 4) (Var "x"))
> evalio [ ] exp >>= λn → putStr (show n)
x? 2
6
```

2.6 Algumas caixas são cofres

Se trocarmos a ordem dos argumentos na função que avalia o valor de uma expressão, podemos obter a seguinte função que dada uma expressão devolve uma função de dicionários para inteiros.

```
eval :: Exp → (Dict → Int)
eval (Const x)  = λ_ → x
eval (Var x)    = λd → fromJust (lookup x d)
eval (Prod l r) = λd → (eval l d) * (eval r d)
eval (Div l r)  = λd → (eval l d) `div` (eval r d)
```

Como a função está *curried* podemos passar-lhe apenas o primeiro argumento: dada uma expressão obtemos uma função que pode ser usada para determinar o seu valor com vários dicionários diferentes. A seguinte interação no interpretador clarifica este facto.

```
> let exp = Prod (Const 3) (Div (Const 4) (Var "x"))
> let f = eval exp
> f [("x", 2)]
6
> f [("x", 1)]
12
```

Um dos problemas com a definição acima é a necessidade de explicitamente propagar o parametro constante *d* para todas as invocações recursivas. É possível evitar este problema se definirmos um novo tipo de *monad*: a nova "caixa" irá neste caso conter uma função que apenas retorna o resultado quando lhe passarmos como argumento um dicionário. De facto, esta caixa comporta-se mais como um "cofre" que contém o resultado, mas cuja abertura implica uma determinada "chave", que neste caso deverá

ser um dicionário. Este monad designa-se normalmente por *Reader* e pode ser generalizado para qualquer tipo de chaves. Na seguinte declaração c representa o tipo das chaves e a o tipo dos resultados.

```
newtype Reader c a = Reader{runReader :: c → a}
```

Note a utilização do mesmo identificador para o construtor de tipos e para o contrutor de valores. Como seria de esperar, um valor do tipo *Reader c a* não é mais do que uma função de c para a. Esta declaração introduz simultaneamente a função *runReader* que pode ser usada para extrair essa função.

```
runReader :: Reader c a → (c → a)
```

Obviamente, como neste caso conhecemos a declaração do tipo respectivo também podemos usar *pattern matching* para extrair a função contida numa computação do tipo *Reader*.

Fixado um tipo c para as chaves, é relativamente simples definir a instância da classe *Monad* para o construtor *Reader c*.

```
instance Monad (Reader c) where
  return x          = Reader (λ_ → x)
  Reader f >>= g = Reader (λy → runReader (g (f y)) y)
```

No caso do *return* temos que construir um cofre que, independentemente da chave que é usada para o abrir, contém sempre o resultado especificado. Na implementação do $>\!\!>\!\!=$ começa-se por extrair o resultado da primeira computação. Para tal, usa-se como chave do cofre o argumento y que irá ser recebido mais tarde. O resultado $f\ y$ é depois usado como argumento da função g por forma a obter uma segunda computação $g\ (f\ y)$. Para abrir este segundo cofre começamos por determinar a função respectiva usando o *runReader*, sendo posteriormente usada a mesma chave y para o abrir. Note que a utilização repetida da mesma chave y é que motivou a introdução deste *monad* em primeiro lugar.

Embora na maior parte da definição do *eval* seja irrelevante qual o dicionário a usar, na segunda cláusula precisamos efectivamente de o conhecer para poder determinar o valor de uma variável. Para usar o monad *Reader* é necessário definir uma computação especial que contém o parâmetro propagado. Essa computação define-se trivialmente usando a função identidade.

```
ask :: Reader c c
ask = Reader id
```

Continuando com a nossa analogia, o cofre *ask* contém dentro uma cópia da chave usada para o abrir. Podemos agora redefinir a função *eval* usando o *monad Reader*.

```
evalr :: Exp → Reader Dict Int
evalr (Const x) = return x
evalr (Var v)   = do d ← ask
                     return (fromJust (lookup v d))
```

```
evalr (Prod l r) = do x ← evalr l
                      y ← evalr r
                      return (x ∗ y)
evalr (Div l r)  = do x ← evalr l
                      y ← evalr r
                      return (x `div` y)
```

Note a utilização do *ask* na segunda cláusula. Se quisermos determinar o valor de uma expressão para um dado dicionário podemos extrair a função contida na computação resultante usando o *runReader*. Esta função pode depois ser usada de igual forma à calculada pela função *eval* no início desta secção.

```
> let f = runReader (evalr exp)
> f [("x", 2)]
6
```

Exercício 19 *Considere o seguinte tipo para representar árvores binárias.*

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Usando o monad *Reader defina uma função decora :: b → Tree a → Tree b que, dado um valor do tipo b e uma árvore do tipo Tree a, substitui o conteúdo de todos os seus nodos pelo primeiro argumento. Comece por definir uma função auxiliar de tipo Tree a → Reader b (Tree b).*

2.7 Computações com estado

Vamos agora considerar uma pequena linguagem de programação para manipular expressões. Nesta linguagem só existem duas instruções:

- **let** $x = n$, onde x é uma variável e n é um inteiro que lhe é atribuído.
- **eval** e, que avalia a expressão aritmética e.

As instruções podem ser sequenciadas com ";" para compor programas mais sofisticados, como por exemplo:

$$\textbf{let}\ x = 2; \textbf{eval}\ (3 * (4\ /\ x))$$

Estes programas podem ser representados pelo seguinte tipo de dados.

```
data Prog = Eval Exp | Let String Int | Seq Prog Prog
```

Por exemplo, o programa anterior poderia ser representado pelo seguinte valor.

```
prog = Seq (Let "x" 2)
           (Eval (Prod (Const 3) (Div (Const 4) (Var "x"))))
```

O resultado de avaliar um programa é uma lista com todos os resultados nele calculados. Para definir uma função que avalia um programa é necessário ir criando um dicionário conforme se avança no programa. Isto implica usar uma função auxiliar que recebe o dicionário calculado até ao momento e produz não só o resultado da avaliação mas também um novo dicionário.

```
evalp :: Prog → [Int]
evalp p = fst (aux p [ ])
  where aux :: Prog → Dict → ([Int], Dict)
    aux (Eval e)  d = ([eval d e], d)
    aux (Let v x) d = ([ ], (v, x) : d)
    aux (Seq l r) d = let (x, e) = aux l d
                          (y, f) = aux r e
                      in  (x ++ y, f)
```

Note que a avaliação da instrução de sequenciação implica a propagação do dicionário calculado no sub-programa esquerdo para a avaliação do sub-programa direito. Esta propagação é necessária pois no sub-programa esquerdo podem existir instruções **let** que definem valores usados no sub-programa direito. O dicionário funciona como uma variável global ou um estado que é necessário propagar ao longo da execução da função *aux*.

Muitas funções tem uma definição facilitada se o estado for propagado de forma implícita, ao contrário da definição acima, onde o programador tem que o fazer explicitamente. Para o conseguir vamos, mais uma vez, recorrer ao conceito de *monad*: neste caso a nossa "caixa" irá conter uma função que dado um estado de tipo *s* devolve não só um resultado de tipo *a*, mas também um novo valor do estado que deverá ser propagado às computações seguintes.

```
newtype State s a = State{runState :: s → (a, s)}
```

Esta declaração introduz simultaneamente a função *runState* que pode ser usada para extrair a função respectiva.

```
runState :: State s a → (s → (a, s))
```

Fixado o tipo do estado *s* é possível definir a instância da classe *Monad* para o construtor de tipos *State s* da seguinte forma.

```
instance Monad (State s) where
  return x = State (λs → (x, s))
  State f >>= g = State (λs → let (x, t) = f s
                              in  runState (g x) t)
```

A função *return* devolve uma computação que, dado um estado qualquer *s* devolve o valor pretendido *x* e o mesmo estado *s*. Na implementação do >>= começa-se por determinar o resultado da primeira computação *State f*. Para tal, é necessário passar como parâmetro à função *f* o estado inicial *s*. O resultado *x* da primeira computação é depois usado para obter a segunda computação *g x* (note que a função >>= tem o

tipo $State\ s\ a \rightarrow (a \rightarrow State\ s\ b) \rightarrow State\ s\ b$). A função *runState* é depois usada para estrair desta computação uma função de tipo $s \rightarrow (b, s)$. Para calcular o resultado final temos finalmente que invocar esta função com o estado t que resultou da primeira computação.

Para utilizar efectivamente o *monad State* não é suficiente a instância respectiva da classe *Monad*. Para além de propagar o estado (através da função $\gg=$) é ocasionalmente necessário saber qual o estado actual e alterar o valor do mesmo. Tal como no caso do *monad Reader* vamos definir algumas computações especiais para este efeito.

```
get :: State s s
get = State (λs → (s, s))
put :: s → State s ()
put s = State (λ_ → ((), s))
```

A computação *get* devolve como resultado o próprio estado de entrada. Note que para aceder ao mesmo é necessário usar a função $\gg=$. Dado um estado qualquer s, a função *put* substitui o estado actual por s, devolvendo um resultado irrelevante de tipo ().

Podemos agora redefinir a nossa função de avaliação da seguinte forma.

```
evalp :: Prog → [Int]
evalp p = fst (runState (aux p) [])
  where aux :: Prog → State Dict [Int]
    aux (Eval e)  = do d ← get
                       return [eval d e]
    aux (Let v x) = do d ← get
                       put ((v, x) : d)
                       return []
    aux (Seq l r) = do x ← aux l
                       y ← aux r
                       return (x ++ y)
```

Note a utilização do *get*, na primeira cláusula da função *aux*, para aceder ao dicionário calculado até ao momento. Na segunda cláusula é acrescentada uma nova valoração ao dicionário, usando-se primeiro o *get* para obter o dicionário actual, seguido de um *put* para definir o novo dicionário. Como o *put* devolve um valor de tipo (), enquanto que a função *aux* devolve um valor de tipo $[Int]$, foi necessário inserir um *return* [ ] para indicar que neste caso não existem resultados.

Exercício 20 *Considere o seguinte tipo para representar árvores binárias.*

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Usando o monad *State defina uma função decora* :: *Tree a* → *Tree Int que substitui o conteúdo de cada nodo da árvore pela sua posição numa travessia* inorder.

2.8 Matrioscas monádicas

Na secção 2.3 implementamos uma função de avaliação para o tipo *Exp* usando o *monad Maybe*.

$$evalm :: Dict \rightarrow Exp \rightarrow Maybe\ Int$$

Esta implementação tinha por objectivo tratar a parcialidade da função de avaliação. Na secção 2.6 implementamos outra versão da mesma função usando o *monad Reader*.

$$evalr :: Exp \rightarrow Reader\ Dict\ Int$$

Neste caso pretendia-se evitar a propagação explícita do dicionário.

A questão que se impõe é: como fazer para combinar estes dois *monads* e obter uma função de avaliação com parcialidade e propagação implícita do dicionário? Uma possível solução consiste em definir um novo *monad* que os combina, com a respectiva instância da classe *Monad*. Um possível tipo de dados para este *monad* poderia ser

$$\textbf{data}\ ReaderMaybe\ c\ a = ReaderMaybe\{runReaderMaybe :: c \rightarrow Maybe\ a\}$$

No entanto, se mais tarde quisermos combinar o *monad Reader* com o *monad IO* é necessário declarar um novo tipo de dados e uma nova instância da classe *Monad*. Obviamente esta solução não escala facilmente para combinações mais sofisticadas, envolvendo três ou mais *monads*.

A solução mais adequada consiste na utilização do conceito de *monad transformer*. Em vez de definir *monads* diferentes para todas as combinações possíveis de *Reader* com outros *monads*, vamos definir um tipo *ReaderT c* que dado um *monad* qualquer *m* o "transforma" num *monad* mais sofisticado que combina os efeitos de *Reader c* com *m*.

$$\textbf{newtype}\ ReaderT\ c\ m\ a = ReaderT\{runReaderT :: c \rightarrow m\ a\}$$

Continuando a analogia da secção 2.6, *ReaderT c m a* é um "cofre" que quando aberto com uma chave de tipo *c* contém lá dentro um resultado *a* dentro de uma "caixa" *m*: dentro de um *monad* temos agora outro *monad*. Compondo vários *transformers* podemos construir verdadeiras "matrioscas monádicas".

Dado um *monad m* podemos então definir uma instância da classe *Monad* para *Reader c m* da seguinte forma:

$$\begin{aligned}&\textbf{instance}\ Monad\ m \Rightarrow Monad\ (ReaderT\ r\ m)\ \textbf{where}\\&\quad return\ x = ReaderT\ (\lambda\_ \rightarrow return\ x)\\&\quad ReaderT\ f \gg\!= g = ReaderT\ (\lambda y \rightarrow f\ y \gg\!= \lambda z \rightarrow runReaderT\ (g\ z)\ y)\end{aligned}$$

Note que ao definir a função *return* é usada a função *return* do *monad m*: primeiro o valor *x* é colocado dentro da "caixa" *m* e só depois é construída a "caixa" *ReaderT c m*. Não se deve confundir a invocação do *return* com uma invocação recursiva: apenas estamos a definir o *return* para o *ReaderT c m* à custa do *return* de *m*, e embora estas funções tenham o mesmo nome (é uma função *overloaded*) são completamente distintas. A mesma situação se passa em relação ao $\gg\!=$: a utilização do mesmo no lado direito da definição serve para retirar o valor $z :: a$ de dentro da computação $f\ y :: m\ a$. É também conveniente definir uma computação especial para aceder ao parâmetro constante de tipo *c*, que é propagado implicitamente por este *monad*.

$$\begin{aligned}&ask :: Monad\ m \Rightarrow ReaderT\ c\ m\ c\\&ask = ReaderT\ return\end{aligned}$$

Mais uma vez o *return* usado é o do *monad m*. Note que estamos a usar o mesmo nome já usado para a computação equivalente do *monad Reader*. Obviamente, tal como está declarado as duas não poderiam coexistir. No entanto, recorrendo a algumas habilidades com classes, foi possível declarar um nodo idêntico para ambas na biblioteca *Control.Monad.Reader*. Desta forma garante-se que quando uma determinada implementação evolui de um *monad* para o *transformer* respectivo tudo continue a funcionar como antes. O mesmo se passa em relação às computações *get* e *put* do *monad State*.

Quando usamos *monad transformers* é conveniente ter um *return* menos poderoso: muitas vezes temos já um resultado *a* dentro computação do tipo *m a*, que apenas desejamos inserir dentro de uma computação *ReaderT c m a*. Nestes casos o *return* não serve porque coloca directamente um *a* dentro duma computação *ReaderT c m a*. A função que deverá ser usada nestes casos é o *lift*. Por forma a permitir que seja uma função *overloaded* que funcione com qualquer *monad transformer*, vamos definir uma instância que agrupa todos os *transformers* e para os quais esta função faz sentido existir.

```
class MonadTrans t where
  lift :: Monad m ⇒ m a → t m a
```

A instância para o *transformer ReaderT c* é trivial.

```
instance MonadTrans (ReaderT c) where
  lift x = ReaderT (λ_ → x)
```

Podemos finalmente definir uma função de avaliação que combina as funcionalidades do *monad Reader* e do *monad Maybe*.

```
evalrm :: Exp → ReaderT Dict Maybe Int
evalrm (Const x)  = return x
evalrm (Var v)    = do d ← ask
                       lift (lookup v d)
evalrm (Prod l r) = do x ← evalrm l
                       y ← evalrm r
                       return (x ∗ y)
evalrm (Div l r)  = do x ← evalrm l
                       y ← evalrm r
                       if y ≡ 0
                         then lift Nothing
                         else return (x `div` y)
```

Note a utilização do *lift* sempre que se pretende executar alguma computação do tipo *Maybe*. As seguintes interações são agora possíveis no interpretador.

```
> runReaderT (evalrm exp1) [("x", 2)]
Just 6
> runReaderT (evalrm exp1) [("x", 0)]
```

Nothing
> runReaderT (*evalrm exp1*) [ ]
Nothing

No primeiro caso a avaliação foi possível, sendo o resultado construído com *Just*. Sempre que um erro ocorre, quer por ocorrer uma divisão por 0, quer por uma variável não estar definida, é devolvolvido o valor *Nothing*.

Exercício 21 *Implemente uma função evalrio*::*Exp* → *ReaderT Dict IO Int que utilize os* monads *Reader e IO na avalição de uma expressão. Pretende-se que, sempre que uma variável não esteja definida, o seu valor seja lido do teclado.*

Exercício 22 *Relembre o* monad *Erro que permite representar computações parciais, mas onde uma computação de erro bem acompanhada de uma descrição sob a forma de uma String.*

data *Erro a* = *Erro String* | *OK a*

Defina um transformer *para este* monad *usando o seguinte tipo de dados.*

data *ErroT m a* = *ErroT*{*runErroT* :: *m* (*Erro a*)}

Devem ser implementadas instâncias pas as classes Monad e MonadTrans.

Exercício 23 *Implemente uma função supereval*::*Exp* → *ReaderT Dict* (*ErroT IO*) *Int que combine os* monads *Reader, Erro e IO na avalição de uma expressão. Pretende-se que, sempre que uma variável não esteja definida, o seu valor seja lido do teclado, e que sejam produzidas mensagens de erro sempre que o valor lido não seja um inteiro ou uma divisão por zero ocorra.*

Exercício 24 *Reescreva a função anterior usando o* transformer *ErrorT da biblioteca Control.Monad.Error em vez do ErroT.*

Exercício 25 *Defina as instâncias das classes Monad e MonadTrans para o* transformer *associado ao* monad *State. Utilize o seguinte tipo de dados.*

data *StateT s m a* = *StateT*{*runStateT* :: *s* → *m* (*a*, *s*)}

Defina também as computações get e put para este transformer.

Exercício 26 *Implemente uma versão da função evalp para avaliação de programas, usando uma combinação de State e Maybe para permitir a propagação de erros sempre que uma variável não estiver definida. Poderá usar a função evalm na avaliação das expressões.*

Exercício 27 *Considere uma máquina de stack muito simples para calcular expressões aritméticas. Os comandos suportados por esta máquina são os seguintes:*

push :: *Int* → *Comando* ()
pop :: *Comando Int*

add :: *Comando* ()
mult :: *Comando* ()

Como esperado, push insere um elemento no topo da stack, pop retira e devolve o elemento que está no topo da stack, e add e mult retiram dois elementos do topo da stack substituindo-os, respectivamente, pela sua soma e produto. Note que o comando pop nem sempre pode ser executado. Pretende-se implementar esta máquina usando o monad *de estado, sendo o tipo dos comandos definido da seguinte forma:*

type *Stack* = [*Int*]
type *Comando a* = *StateT Stack Maybe a*

Implemente os comandos acima referidos por forma a obter o seguinte comportamento.

> *evalStateT* (*push* 2 >> *push* 4 >> *mult* >> *push* 3 >> *add* >> *pop*) [ ]
Just 9
> *evalStateT* (*push* 2 >> *push* 4 >> *mult* >> *add* >> *pop*) [ ]
Nothing

No primeiro caso é calculada correctamente a expressão $3 + (4 * 2)$. *No segundo caso tal não é possível pois quando é executado o comando add só existe um argumento na stack.*

Capítulo 3

Cálculo *point-free* não recursivo

O estilo de programação *point-free*, ao contrário do estilo *pointwise*, caracteriza-se pela ausência de variáveis na definição de funções. No estilo *pointwise* uma função é definida especificando directamente o seu comportamento num determinado ponto do domínio, usualmente denotado por uma variável ou um padrão envolvendo variáveis. No estilo *point-free* uma função é definida por combinação de outras funções mais simples usando um conjunto limitado the combinadores. A escolha destes combinadores é ditada pelo poder das leis de cálculo que lhes estão associadas. Este facto implica que, normalmente, seja mais fácil demonstrar propriedades sobre programas escritos neste estilo.

3.1 Composição e identidade

O combinador mais fundamental do estilo *point-free* é a composição de funções:

$$\begin{array}{l}(\circ) :: (b \to c) \to (a \to b) \to (a \to c) \\ (f \circ g)\ x = f\ (g\ x)\end{array} \qquad \textsc{Def}\text{-}\circ$$

Este combinador é uma função de ordem superior que, dadas duas funções de tipo adequado, retorna uma nova função correspondente à sua composição. Igualmente importante é a função primitiva identidade:

$$\begin{array}{l}\mathsf{id} :: a \to a \\ \mathsf{id}\ x = x\end{array} \qquad \textsc{Def}\text{-}\mathsf{id}$$

A composição e identidade satisfazem as seguintes leis:

$$(f \circ g) \circ h = f \circ (g \circ h) \qquad \text{Assoc-}\circ$$

$$f \circ \mathsf{id} = f = \mathsf{id} \circ f \qquad \textsc{Nat}\text{-}\mathsf{id}$$

Estas duas leis são o primeiro exemplo de leis de cálculo equacional *point-free*, e podem ser usadas para demonstrar propriedades de programas definidos neste estilo. Considere, por exemplo, a seguinte proposição, onde $\mathsf{succ} :: \mathit{Int} \to \mathit{Int}$ é a função que determina o sucessor de um número.

$$\mathsf{succ} \circ ((\mathsf{id} \circ \mathsf{succ}) \circ (\mathsf{succ} \circ \mathsf{id})) = (\mathsf{succ} \circ \mathsf{succ}) \circ \mathsf{succ}$$

Para demonstrar esta igualdade podemos usar as leis acima apresentadas: começando num dos lados da equação deve ser possível obter o outro lado, aplicando sucessivamente uma das leis de cálculo para substituir termos por termos iguais. Neste caso podemos efectuar a seguinte prova:

$$\begin{array}{ll} & \mathsf{succ} \circ ((\mathsf{id} \circ \mathsf{succ}) \circ (\mathsf{succ} \circ \mathsf{id})) \\ = & \quad \{ \textsc{Nat}\text{-}\mathsf{id} \} \\ & \mathsf{succ} \circ (\mathsf{succ} \circ (\mathsf{succ} \circ \mathsf{id})) \\ = & \quad \{ \textsc{Assoc}\text{-}\circ \} \\ & (\mathsf{succ} \circ \mathsf{succ}) \circ (\mathsf{succ} \circ \mathsf{id}) \\ = & \quad \{ \textsc{Nat}\text{-}\mathsf{id} \} \\ & (\mathsf{succ} \circ \mathsf{succ}) \circ \mathsf{succ} \end{array}$$

Note como cada passo da prova é justificado pela correspondente lei de cálculo usada para progredir.

Uma das consequências da lei Assoc-$\circ$ é que podemos omitir os parêntesis numa sequência de composições, sem que haja qualquer ambiguidade sobre a semântica da função definida. Este facto permite simplificar ainda mais as provas no estilo *point-free* pois deixa de ser necessário usar explicitamente a lei Assoc-$\circ$. A prova anterior poderia ser simplificada da seguinte forma.

$$\begin{array}{ll} & \mathsf{succ} \circ \mathsf{id} \circ \mathsf{succ} \circ \mathsf{succ} \circ \mathsf{id} \\ = & \quad \{ \textsc{Nat}\text{-}\mathsf{id} \} \\ & \mathsf{succ} \circ \mathsf{succ} \circ \mathsf{succ} \circ \mathsf{id} \\ = & \quad \{ \textsc{Nat}\text{-}\mathsf{id} \} \\ & \mathsf{succ} \circ \mathsf{succ} \circ \mathsf{succ} \end{array}$$

Naturalmente, a mesma prova poderia ser feita no estilo *point-wise* usando as definições da composição e da identidade. Para tal teremos que usar um dos axiomas mais importantes da matemática, denominado extensionalidade ou igualdade extensional, que nos diz que duas funções são iguais se tiverem o mesmo domínio e produzirem o mesmo resultado para todos os valores desse domínio.

$$f = g \Leftrightarrow \forall\, x \,.\, f\ x = g\ x \qquad \textsc{Ext}\text{-}=$$

Equipados com este axioma podemos demonstrar a propriedade acima enunciada da seguinte forma.

$$\begin{array}{ll} & \mathsf{succ} \circ ((\mathsf{id} \circ \mathsf{succ}) \circ (\mathsf{succ} \circ \mathsf{id})) = (\mathsf{succ} \circ \mathsf{succ}) \circ \mathsf{succ} \\ \Leftrightarrow & \quad \{ \textsc{Ext}\text{-}= \} \\ & (\mathsf{succ} \circ ((\mathsf{id} \circ \mathsf{succ}) \circ (\mathsf{succ} \circ \mathsf{id}))) \ x = ((\mathsf{succ} \circ \mathsf{succ}) \circ \mathsf{succ})\ x \\ \Leftrightarrow & \quad \{ \textsc{Def}\text{-}\circ \} \\ & \mathsf{succ}\ (((\mathsf{id} \circ \mathsf{succ}) \circ (\mathsf{succ} \circ \mathsf{id}))\ x) = ((\mathsf{succ} \circ \mathsf{succ}) \circ \mathsf{succ})\ x \\ \Leftrightarrow & \quad \{ \textsc{Def}\text{-}\circ \} \\ & \mathsf{succ}\ ((\mathsf{id} \circ \mathsf{succ})\ ((\mathsf{succ} \circ \mathsf{id})\ x)) = ((\mathsf{succ} \circ \mathsf{succ}) \circ \mathsf{succ})\ x \\ \Leftrightarrow & \quad \{ \textsc{Def}\text{-}\circ \} \end{array}$$

$$
\begin{array}{ll}
& \mathsf{succ}\ ((\mathsf{id} \circ \mathsf{succ})\ (\mathsf{succ}\ (\mathsf{id}\ x))) = ((\mathsf{succ} \circ \mathsf{succ}) \circ \mathsf{succ})\ x \\
\Leftrightarrow & \quad \{\ \textsc{Def}\text{-}\mathsf{id}\ \} \\
& \mathsf{succ}\ ((\mathsf{id} \circ \mathsf{succ})\ (\mathsf{succ}\ x)) = ((\mathsf{succ} \circ \mathsf{succ}) \circ \mathsf{succ})\ x \\
\Leftrightarrow & \quad \{\ \textsc{Def}\text{-}\circ\ \} \\
& \mathsf{succ}\ (\mathsf{id}\ (\mathsf{succ}\ (\mathsf{succ}\ x))) = ((\mathsf{succ} \circ \mathsf{succ}) \circ \mathsf{succ})\ x \\
\Leftrightarrow & \quad \{\ \textsc{Def}\text{-}\mathsf{id}\ \} \\
& \mathsf{succ}\ (\mathsf{succ}\ (\mathsf{succ}\ x)) = ((\mathsf{succ} \circ \mathsf{succ}) \circ \mathsf{succ})\ x \\
\Leftrightarrow & \quad \{\ \textsc{Def}\text{-}\circ\ \} \\
& \mathsf{succ}\ (\mathsf{succ}\ (\mathsf{succ}\ x)) = (\mathsf{succ} \circ \mathsf{succ})\ (\mathsf{succ}\ x) \\
\Leftrightarrow & \quad \{\ \textsc{Def}\text{-}\circ\ \} \\
& \mathsf{succ}\ (\mathsf{succ}\ (\mathsf{succ}\ x)) = \mathsf{succ}\ (\mathsf{succ}\ (\mathsf{succ}\ x))
\end{array}
$$

Para simplificar foi omitido o quantificador universal. Mesmo assim, a prova ficou muito mais extensa e difícil de acompanhar porque, devido à necessidade de utilizar o axioma da extensionalidade, foi necessário avançar na prova usando equivalências em vez de simples igualdades.

Exercício 28 *Demonstre as leis* Assoc-$\circ$ *e* Nat-id *usando as respectivas definições,* Def-$\circ$ *e* Def-id*, e o axioma da extensionalidade* Ext-=.

Embora o cálculo *point-free* seja muito elegante, a maior parte dos programas estão definidos no estilo *pointwise*. Assim, sempre que se pretender demonstrar alguma propriedade sobre estes programas, é necessário obter primeiro as respectivas definições *point-free*. Para tal vamos usar as definições dos combinadores no sentido inverso, e a igualdade extensional para deixar cair as variáveis sempre uma equação $\forall\ x\ .\ f\ x = g\ x$ é encontrada. Como exemplo, considere a seguinte função.

$$
\begin{array}{l}
\mathsf{sss} :: \mathit{Int} \to \mathit{Int} \\
\mathsf{sss}\ x = \mathsf{succ}\ (\mathsf{succ}\ (\mathsf{succ}\ x))
\end{array}
$$

Note que nesta definição (tal como em qualquer definição em Haskell envolvendo variáveis) a variável x está implicitamente quantificada. É possível obter uma definição *point-free* para sss aplicando sucessivamente a lei Def-$\circ$ seguida de Ext-=.

$$
\begin{array}{ll}
& \mathsf{sss}\ x = \mathsf{succ}\ (\mathsf{succ}\ (\mathsf{succ}\ x)) \\
\Leftrightarrow & \quad \{\ \textsc{Def}\text{-}\circ\ \} \\
& \mathsf{sss}\ x = \mathsf{succ}\ ((\mathsf{succ} \circ \mathsf{succ})\ x) \\
\Leftrightarrow & \quad \{\ \textsc{Def}\text{-}\circ\ \} \\
& \mathsf{sss}\ x = (\mathsf{succ} \circ \mathsf{succ} \circ \mathsf{succ})\ x \\
\Leftrightarrow & \quad \{\ \textsc{Ext}\text{-}=\ \} \\
& \mathsf{sss} = \mathsf{succ} \circ \mathsf{succ} \circ \mathsf{succ}
\end{array}
$$

3.2 Produtos

Para combinar funções que partilham o mesmo domínio vamos recorrer ao conceito de produto. Informalmente, o produto de dois tipos a e b é um tipo de dados $a \times b$ que

contém exactamente um valor de tipo a e um valor de tipo b. Considere os seguintes tipos Haskell como possivéis implementações do tipo $Char \times Int$.

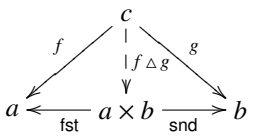
```
type A = Char
type B = (Char, Int)
data C = C Char Int
data D = D Char Int Int
type E = (Int, Char)
```

De acordo com a definição anterior é óbvio que o tipo A não corresponde ao tipo desejado pois apenas contém um carácter. O tipo D também não corresponde ao tipo desejado pois pode armazenar dois inteiros. Todos os restantes são possíveis implementações em Haskell para o tipo $Char \times Int$, dado que todos eles contêm exactamente um carácter e um inteiro.

Será possível dar uma definção formal para o produto, que corresponda à noção informal acima apresentada? Uma possibilidade seria dar uma definição extensional, indicando todos os possíveis tipos que implementam um determinado produto. Como se pode inferir dos exemplos apresentados, esta estratégia não é muito adequada pois existem muito tipos (de facto, um número infinito deles) que podem implementar correctamente um dado produto. Felizmente, é possível dar uma definição intensional e formal de produto recorrendo ao conceito de unicidade. Um tipo $a \times b$ é o produto entre a e b se e só se:

1. Existirem as funções $\mathsf{fst} :: a \times b \to a$ e $\mathsf{snd} :: a \times b \to b$ que nos permitem aceder ao seu conteúdo.

2. Dadas duas funções quaisqer $f :: c \to a$ e $g :: c \to b$ existir uma única forma de as combinar por forma a obter uma função $f \vartriangle g :: c \to a \times b$, de tal forma que o seguinte diagrama comute.



O combinador $\vartriangle$ denomina-se *split*, e a sua unicidade no diagrama acima é realçada usando uma seta tracejada. Diz-se que um diagrama comuta se todas as possíveis funções entre dois tipos forem idênticas. Neste caso em particular, isso implica que $\mathsf{fst} \circ (f \vartriangle g) = f$ e $\mathsf{snd} \circ (f \vartriangle g) = g$. No caso dos tipos antes apresentados, a primeira propriedade é suficiente para excluir o tipo A como possível implementação de $Char \times Int$. No entanto, para excluir o tipo D a segunda propriedade é fundamental, dado que é possível definir as projecções fst e snd para este tipo. Uma possível implementação seria:

```
fst :: D → Char
fst (D x y z) = x
```

$$\begin{aligned}&\mathsf{snd} :: D \to \mathit{Int}\\&\mathsf{snd}\ (D\ x\ y\ z) = y\end{aligned}$$

No entanto, fixadas estas definições, existe mais do que uma implementação possível para o *split* que faz o diagrama comutar: na seguinte definição o valor 0 pode ser substituído por qualquer outro inteiro pois essa informação é ignorada pela função snd.

$$\begin{aligned}&(\triangle) :: (c \to \mathit{Char}) \to (c \to \mathit{Int}) \to (c \to D)\\&(f \triangle g)\ x = D\ (f\ x)\ (g\ x)\ 0\end{aligned}$$

Em alternativa ao diagrama apresentado acima, podemos caracterizar o produto usando a seguinte propriedade dita universal:

$$h = f \triangle g \Leftrightarrow \mathsf{fst} \circ h = f \wedge \mathsf{snd} \circ h = g \qquad \textsc{Univ-}\times$$

Esta propriedade dá-nos uma definição implicita das funções fst e snd e do combinador $\triangle$, que é independente da implementação concreta escolhida para o produto. No entanto, para efeitos de conversão entre os estilos *pointwise* e *point-free* é conveniente fixar uma implementação standard para o produto $a \times b$. Em Haskell vamos usar o tipo (a, b) para esse efeito. Fixada esta implementação, podemos definir explicitamente as funções anteriores da seguinte forma.

$$\begin{aligned}&\mathsf{fst} :: (a, b) \to a\\&\mathsf{fst}\ (x, y) = x\end{aligned} \qquad \textsc{Def-fst}$$

$$\begin{aligned}&\mathsf{snd} :: (a, b) \to a\\&\mathsf{snd}\ (x, y) = y\end{aligned} \qquad \textsc{Def-snd}$$

$$\begin{aligned}&(\triangle) :: (c \to a) \to (c \to b) \to (c \to (a, b))\\&(f \triangle g)\ x = (f\ x, g\ x)\end{aligned} \qquad \textsc{Def-}\triangle$$

Para poder fazer provas equacionais envolvendo produtos é conveniente caracterizar este tipo usando apenas leis equacionais. É possível demonstrar que a lei universal pode ser caracterizada pelas seguintes três leis, denomindas, respectivamente, cancelamento, reflexão e fusão.

$$\mathsf{fst} \circ (f \triangle g) = f \wedge \mathsf{snd} \circ (f \triangle g) = g \qquad \textsc{Cancel-}\times$$

$$\mathsf{fst} \triangle \mathsf{snd} = \mathsf{id} \qquad \textsc{Reflex-}\times$$

$$(f \triangle g) \circ h = f \circ h \triangle g \circ h \qquad \textsc{Fusion-}\times$$

Note que o combinador $\circ$ tem uma prioridade mais alta que todos os outros combinadores: daí a omissão dos parêntesis no lado direito da lei Fusion-×. É possível demonstrar estas leis facilmente recorrendo à lei universal dos produtos. Por exemplo, assumindo que lei Cancel-× foi previamente provada, a lei Fusion-× pode ser demonstrada da seguinte forma.

$$\begin{aligned}&(f \triangle g) \circ h = f \circ h \triangle g \circ h\\\Leftrightarrow\quad &\{\ \textsc{Univ-}\times\ \}\end{aligned}$$

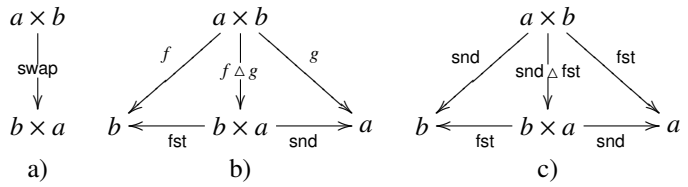


Figura 3.1: Derivação do swap usando diagramas.

$$
\begin{array}{ll}
& \mathsf{fst} \circ (f \vartriangle g) \circ h = f \circ h \wedge \mathsf{snd} \circ (f \vartriangle g) \circ h = g \circ h \\
\Leftrightarrow & \quad \{ \text{CANCEL-}\times \} \\
& f \circ h = f \circ h \wedge \mathsf{snd} \circ (f \vartriangle g) \circ h = g \circ h \\
\Leftrightarrow & \quad \{ \text{CANCEL-}\times \} \\
& f \circ h = f \circ h \wedge g \circ h = g \circ h
\end{array}
$$

Para economizar espaço, é normal que num único passo de prova se aplique a mesma lei em mais do que um termo. Por exemplo, na prova anterior os dois últimos passos poderiam ter sido condensados num único. Dado que esta prova usou a lei universal fica demonstrada a veraciadade de FUSION-× qualquer que seja a implementação concreta de produto escolhida. Obviamente, a mesma prova poderia ter sido feita no estilo *pointwise* (recorrendo às definições) para a implementação concreta (a, b).

Exercício 29 *Demonstre as leis* CANCEL-× *e* REFLEX-× *usando a lei universal dos produtos.*

Exercício 30 *Demonstre a lei* FUSION-× *no estilo* point-wise *recorrendo à lei* EXT-= *e às definições* DEF-∘ *e* DEF-△.

Como exemplo da utilização dos produtos vamos demonstrar o isomorfismo $a \times b \cong b \times a$. Como sabem, para demonstrar um isomorfismo $a \cong b$ é necessário encontrar duas funções $f :: a \to b$ e $g :: b \to a$ tal que $f \circ g = \mathsf{id} \wedge g \circ f = \mathsf{id}$. Neste caso concreto, as duas funções são idênticas, bastando encontrar uma função $\mathsf{swap} :: a \times b \to b \times a$ e demonstrar que $\mathsf{swap} \circ \mathsf{swap} = \mathsf{id}$. Para podermos usar o cálculo equacional *point-free* temos primeiro que encontrar uma definição para swap neste estilo. Esta definição pode ser encontrada de duas formas: usando os diagramas para a inferir ou definindo primeiro uma versão *point-wise*, derivando posteriormente a versão *point-free* por cálculo.

Na figura 3.1 é apresentada a derivação da definição *point-free* do swap recorrendo aos diagramas. Começamos por desenhar o diagrama a) que apenas indica o tipo da função pretendida. Como esta função calcula um produto, em princípio pode ser definida através de um *split*, desde que seja possível determinar as funções $f :: a \times b \to b$ e $g :: a \times b \to a$ indicadas no diagrama b). Neste caso é trivial identificar estas funções, sendo obtido o diagrama c) onde fica clara qual a definição pretendida para o swap.

$$\mathsf{swap} = \mathsf{snd} \vartriangle \mathsf{fst} \qquad \text{DEF-swap}$$

Dada esta definição *point-free* é trivial demonstrar o isomorfismo pretendido.

$$
\begin{array}{ll}
 & \mathsf{swap} \circ \mathsf{swap} \\
= & \quad \{ \textsc{Def}\text{-}\mathsf{swap} \} \\
 & (\mathsf{snd} \vartriangle \mathsf{fst}) \circ (\mathsf{snd} \vartriangle \mathsf{fst}) \\
= & \quad \{ \textsc{Fusion}\text{-}\times \} \\
 & \mathsf{snd} \circ (\mathsf{snd} \vartriangle \mathsf{fst}) \vartriangle \mathsf{fst} \circ (\mathsf{snd} \vartriangle \mathsf{fst}) \\
= & \quad \{ \textsc{Cancel}\text{-}\times \} \\
 & \mathsf{fst} \vartriangle \mathsf{snd} \\
= & \quad \{ \textsc{Reflex}\text{-}\times \} \\
 & \mathsf{id}
\end{array}
$$

O segundo método para obter uma definição *point-free* consiste em escrever a respectiva versão *pointwise*, sendo a definição pretendido derivada por cálculo. Para fazer esta derivação é conveniente usar a seguinte lei, que nos permite eliminar um par de variáveis do argumento de uma função.

$$f\ a = b \Leftrightarrow f\ a\ [(x, y)\ /\ z] = b\ [x\ /\ \mathsf{fst}\ z, y\ /\ \mathsf{snd}\ z] \qquad \textsc{Elim}\text{-}\times$$

Em relação a notação usada, note que o termo $a\ [b\ /\ c]$ resulta do termo a substituindo todas as ocorrências do termo b por c. Sendo assim, esta lei diz-nos que qualquer par de variáveis (x, y) pode ser substituido por uma única variável z, desde que no lado direito da definição se substituam todas as ocorrências de x por $\mathsf{fst}\ z$ e de y por $\mathsf{snd}\ z$.

Considere a seguinte definição *pointwise* do swap.

$$
\begin{array}{l}
\mathsf{swap} :: (a, b) \to (b, a) \\
\mathsf{swap}\ (x, y) = (y, x)
\end{array}
$$

Usando lei Elim-× é relativamente simples derivar a respectiva definição *point-free*.

$$
\begin{array}{ll}
 & \mathsf{swap}\ (x, y) = (y, x) \\
\Leftrightarrow & \quad \{ \textsc{Elim}\text{-}\times \} \\
 & \mathsf{swap}\ z = (\mathsf{snd}\ z, \mathsf{fst}\ z) \\
\Leftrightarrow & \quad \{ \textsc{Def}\text{-}\vartriangle \} \\
 & \mathsf{swap}\ z = (\mathsf{snd} \vartriangle \mathsf{fst})\ z \\
\Leftrightarrow & \quad \{ \textsc{Ext}\text{-}= \} \\
 & \mathsf{swap} = \mathsf{snd} \vartriangle \mathsf{fst}
\end{array}
$$

Depois de eliminar o par de variáveis no argumento, note a utilização da definição do *split* no sentido inverso para introduzir o respectivo combinador.

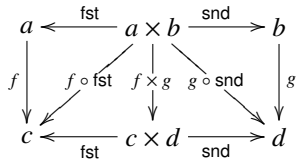
Exercício 31 *Demonstre o isomorfismo* $(a \times b) \times c \cong a \times (b \times c)$*. Para tal deverá começar por definir as funções* $\mathsf{assocr} :: (a \times b) \times c \to a \times (b \times c)$ *e* $\mathsf{assocl} :: a \times (b \times c) \to (a \times b) \times c$ *no estilo* point-free*. Depois terá que demonstrar as propriedades* $\mathsf{assocr} \circ \mathsf{assocl} = \mathsf{id}$ *e* $\mathsf{assocl} \circ \mathsf{assocr} = \mathsf{id}$*. Calcule a definição* point-free *do* assocr *usando diagramas e a do* assocl *a partir da respectiva definição* pointwise*:*

$$
\begin{array}{l}
\mathsf{assocl} :: (a, (b, c)) \to ((a, b), c) \\
\mathsf{assocl}\ (x, (y, z)) = ((x, y), z)
\end{array}
$$

Quando se pretende definir uma função que aceita um par e produz outro par pode ser conveniente usar o seguinte combinador, que dadas duas funções f e g aplica-as, respectivamente, ao primeiro e segundo componentes do par.

$$f \times g = f \circ \mathsf{fst} \vartriangle g \circ \mathsf{snd} \qquad \textsc{Def}\text{-}\times$$

Esta definição resulta do seguinte diagrama.



Este combinador satisfaz as seguintes propriedades, que podem ser facilmente demonstradas recorrendo à sua definição.

$$(f \times g) \circ (h \vartriangle i) = f \circ h \vartriangle g \circ i \qquad \textsc{Absor}\text{-}\times$$
$$(f \times g) \circ (h \times i) = f \circ h \times g \circ i \qquad \textsc{Functor}\text{-}\times$$
$$\mathsf{id} \times \mathsf{id} = \mathsf{id} \qquad \textsc{Functor-Id}\text{-}\times$$

Exercício 32 *Demonstre as propriedades* Absor-×, Functor-× *e* Functor-Id-× *usando cálculo equacional.*

Usando o combinador × é possível definir a função assocr de forma mais sucinta.

$$\mathsf{assocr} = (\mathsf{fst} \circ \mathsf{fst}) \vartriangle (\mathsf{snd} \times \mathsf{id}) \qquad \textsc{Def}\text{-}\mathsf{assocr}$$

Usando esta definição, a propriedade $(\mathsf{id} \times \mathsf{snd}) \circ \mathsf{assocr} = \mathsf{fst} \times \mathsf{id}$ poderia ser demonstrada da seguinte forma.

$$\begin{array}{ll}
 & (\mathsf{id} \times \mathsf{snd}) \circ \mathsf{assocr} \\
= & \quad \{\ \textsc{Def}\text{-}\mathsf{assocr}\ \} \\
 & (\mathsf{id} \times \mathsf{snd}) \circ ((\mathsf{fst} \circ \mathsf{fst}) \vartriangle (\mathsf{snd} \times \mathsf{id})) \\
= & \quad \{\ \textsc{Absor}\text{-}\times\ \} \\
 & \mathsf{id} \circ \mathsf{fst} \circ \mathsf{fst} \vartriangle \mathsf{snd} \circ (\mathsf{snd} \times \mathsf{id}) \\
= & \quad \{\ \textsc{Nat}\text{-}\mathsf{id}\ \} \\
 & \mathsf{fst} \circ \mathsf{fst} \vartriangle \mathsf{snd} \circ (\mathsf{snd} \times \mathsf{id}) \\
= & \quad \{\ \textsc{Def}\text{-}\times\ \} \\
 & \mathsf{fst} \circ \mathsf{fst} \vartriangle \mathsf{snd} \circ (\mathsf{snd} \circ \mathsf{fst} \vartriangle \mathsf{id} \circ \mathsf{snd}) \\
= & \quad \{\ \textsc{Cancel}\text{-}\times\ \} \\
 & \mathsf{fst} \circ \mathsf{fst} \vartriangle \mathsf{id} \circ \mathsf{snd} \\
= & \quad \{\ \textsc{Def}\text{-}\times\ \} \\
 & \mathsf{fst} \times \mathsf{id}
\end{array}$$

Exercício 33 *Demonstre que* swap *é uma transformação natural, ou seja, que verifica a seguinte propriedade para quaisquer f e g.*

$$\mathsf{swap} \circ (f \times g) = (g \times f) \circ \mathsf{swap}$$

Exercício 34 *Demonstre que* assocr *é uma transformação natural, ou seja, que verifica a seguinte propriedade para quaisquer f, g e h.*

$$\mathsf{assocr} \circ ((f \times g) \times h) = (f \times (g \times h)) \circ \mathsf{assocr}$$

3.3 Somas

Para combinar funções que partilham o mesmo contra-domínio vamos recorrer ao conceito de soma. Informalmente, a soma de dois tipos a e b é um tipo $a + b$ que contém apenas um valor de tipo a ou um valor de tipo b. Mais uma vez, podemos ter vários candidatos para implementar o tipo *Char* + *Int* em Haskell:

```
type A = Char
type B = Either Char Int
type C = Either Int Char
type D = Either Char (Either Int Bool)
data E = Inl Char | Inr Int
```

Relembre que o tipo *Either* se encontra pré-definido como

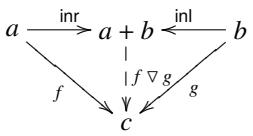
```
data Either a b = Left a | Right b
```

De acordo com a definição informal antes apresentada, é possível excluir os tipos A e D das possíveis implementações para *Char* + *Int*: o primeiro não pode conter um inteiro, enquanto que o segundo pode também conter apenas um booleano.

Tal como para os produtos, é possível dar uma definição formal para a soma recorrendo ao conceito de unicidade. Um tipo $a + b$ é a soma entre a e b se e só se:

1. Existirem as funções $\mathsf{inl} :: a \to a + b$ e $\mathsf{inr} :: b \to a + b$ que nos permitem criar uma soma $a + b$ usando um valor de tipo a ou um valor de tipo b.

2. Dadas duas funções quaisqer $f :: a \to c$ e $g :: b \to c$ existir uma única forma de as combinar por forma a obter uma função $f \triangledown g :: a + b \to c$, de tal forma que o seguinte diagrama comute.



O combinador $\triangledown$ denomina-se *either*. A unicidade exclui o tipo D acima, porque é possível definir várias versões do *either* com comportamentos distintos para o caso em que o valor é um booleano. Em alternativa ao diagrama podemos caracterizar a soma pela respectiva lei universal:

$$h = f \triangledown g \Leftrightarrow h \circ \mathsf{inl} = f \wedge h \circ \mathsf{inr} = g \qquad \text{Univ-+}$$

Embora esta lei defina o *either* independentemente da implementação concreta para as somas, é conveniente fixar uma implementação de referência para efeitos de conversção entre *pointwise* e *point-free*. Neste caso será usado o tipo *Either*, sendo os combinadores definidos da seguinte forma.

$$\begin{array}{l}\mathsf{inl} :: a \to \mathit{Either}\ a\ b \\ \mathsf{inl}\ x = \mathit{Left}\ x\end{array} \qquad \textsc{Def-inl}$$

$$\begin{array}{l}\mathsf{inr} :: b \to \mathit{Either}\ a\ b \\ \mathsf{inr}\ x = \mathit{Right}\ x\end{array} \qquad \textsc{Def-inr}$$

$$\begin{array}{l}(\triangledown) :: (a \to c) \to (b \to c) \to (\mathit{Either}\ a\ b \to c) \\ (f \triangledown g)\ x = \mathbf{case}\ x\ \mathbf{of}\ \{\mathit{Left}\ y \to f\ y; \mathit{Right}\ z \to g\ z\}\end{array} \qquad \textsc{Def-}\triangledown$$

Em provas equacionais podemos usar as seguintes leis em vez da universal.

$$(f \triangledown g) \circ \mathsf{inl} = f \wedge (f \triangledown g) \circ \mathsf{inr} = g \qquad \textsc{Cancel-+}$$

$$\mathsf{inl} \triangledown \mathsf{inr} = \mathsf{id} \qquad \textsc{Reflex-+}$$

$$f \circ (g \triangledown h) = f \circ g \triangledown f \circ h \qquad \textsc{Fusion-+}$$

Exercício 35 *Demonstre as leis* Cancel-+*,* Reflex-+ *e* Fusion-+ *usando a lei universal das somas.*

Na derivação de código *point-free* a partir the código *pointwise* que involva somas é mais fácil usar a lei universal do que a definição de *either* dada acima. Para usar esta definição seria necessário que as funções estivessem definidas usando apenas uma equação com *cases*. No entanto, na maior parte dos casos os programadores preferem usar várias equações e *pattern-matching*, como, por exemplo, na seguinte definição da função *coswap*, que testemunha o isomorfismo $a + b \cong b + a$.

$$\begin{array}{l}\mathsf{coswap} :: \mathit{Either}\ a\ b \to \mathit{Either}\ b\ a \\ \mathsf{coswap}\ (\mathit{Left}\ x) \quad = \mathit{Right}\ x \\ \mathsf{coswap}\ (\mathit{Right}\ x) = \mathit{Left}\ x\end{array}$$

Dadas estas equações é possível derivar a respectiva versão *point-free* da seguinte forma.

$$\begin{array}{ll} & \mathsf{coswap}\ (\mathit{Left}\ x) = \mathit{Right}\ x \wedge \mathsf{coswap}\ (\mathit{Right}\ x) = \mathit{Left}\ x \\ \Leftrightarrow & \quad \{\ \textsc{Def-inl}, \textsc{Def-inr}\ \} \\ & \mathsf{coswap}\ (\mathsf{inl}\ x) = \mathsf{inr}\ x \wedge \mathsf{coswap}\ (\mathsf{inr}\ x) = \mathsf{inl}\ x \\ \Leftrightarrow & \quad \{\ \textsc{Def-}\circ\ \} \\ & (\mathsf{coswap} \circ \mathsf{inl})\ x = \mathsf{inr}\ x \wedge (\mathsf{coswap} \circ \mathsf{inr})\ x = \mathsf{inl}\ x \\ \Leftrightarrow & \quad \{\ \textsc{Ext-=}\ \} \\ & \mathsf{coswap} \circ \mathsf{inl} = \mathsf{inr} \wedge \mathsf{coswap} \circ \mathsf{inr} = \mathsf{inl} \\ \Leftrightarrow & \quad \{\ \textsc{Univ-+}\ \} \\ & \mathsf{coswap} = \mathsf{inr} \triangledown \mathsf{inl}\end{array}$$

Note a utilização da lei Univ-+ no último passo, para passar duas equações da forma $h \circ \mathsf{inl} = f \wedge h \circ \mathsf{inr} = g$ para uma única equação $h = f \triangledown g$. Como a função coswap

é dual do swap a prova do isomorfismo $a + b \cong b + a$ segue exactamente os passos da prova do isomorfismo $a \times b \cong b \times a$.

$$
\begin{array}{ll}
& \mathsf{coswap} \circ \mathsf{coswap} \\
= & \quad \{ \textsc{Def}\text{-}\mathsf{coswap} \} \\
& (\mathsf{inr} \triangledown \mathsf{inl}) \circ (\mathsf{inr} \triangledown \mathsf{inl}) \\
= & \quad \{ \textsc{Fusion}\text{-}+ \} \\
& (\mathsf{inr} \triangledown \mathsf{inl}) \circ \mathsf{inr} \triangledown (\mathsf{inr} \triangledown \mathsf{inl}) \circ \mathsf{inl} \\
= & \quad \{ \textsc{Cancel}\text{-}+ \} \\
& \mathsf{inl} \triangledown \mathsf{inr} \\
= & \quad \{ \textsc{Reflex}\text{-}+ \} \\
& \mathsf{id}
\end{array}
$$

Se a função coswap estiver definida usando um *case*, a derivação da versão *point-free* pode usar directamente a definição Def-$\triangledown$.

$$
\begin{array}{ll}
& \mathsf{coswap}\ x = \mathbf{case}\ x\ \mathbf{of}\ \{Left\ y \to Right\ y; Right\ z \to Left\ z\} \\
\Leftrightarrow & \quad \{ \textsc{Def}\text{-}\triangledown \} \\
& \mathsf{coswap}\ x = (Right \triangledown Left)\ x \\
\Leftrightarrow & \quad \{ \textsc{Def}\text{-}\mathsf{inl}, \textsc{Def}\text{-}\mathsf{inr} \} \\
& \mathsf{coswap}\ x = (\mathsf{inr} \triangledown \mathsf{inl})\ x \\
\Leftrightarrow & \quad \{ \textsc{Ext}\text{-}= \} \\
& \mathsf{coswap} = \mathsf{inr} \triangledown \mathsf{inl}
\end{array}
$$

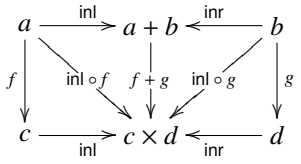
Exercício 36 *Demonstre o isomorfismo* $(a+b)+c \cong a+(b+c)$. *Para tal deverá começar por definir as funções* coassocr $:: (a + b) + c \to a + (b + c)$ *e* coassocl $:: a + (b + c) \to (a + b) + c$ *no estilo* point-free. *Calcule a definição* point-free *do* coassocr *usando diagramas e a do* coassocl *a partir da respectiva definição* pointwise*:*

$$
\begin{array}{l}
\mathsf{coassocl} :: Either\ a\ (Either\ b\ c) \to Either\ (Either\ a\ b)\ c \\
\mathsf{coassocl}\ (Left\ x) \qquad\quad = Left\ (Left\ x) \\
\mathsf{coassocl}\ (Right\ (Left\ x)) \quad = Left\ (Right\ x) \\
\mathsf{coassocl}\ (Right\ (Right\ x)) = Right\ x
\end{array}
$$

Quando se pretende definir uma função que aceita uma soma e produz uma soma pode ser conveniente usar o seguinte combinador.

$$f + g = \mathsf{inl} \circ f \triangledown \mathsf{inr} \circ g \qquad \textsc{Def}\text{-}+$$

Esta definição resulta do seguinte diagrama.



Este combinador satisfaz as seguintes propriedades, que podem ser facilmente demonstradas recorrendo à sua definição.

$$(f \triangledown g) \circ (h + i) = f \circ h \triangledown g \circ i \qquad \text{Absor-+}$$
$$(f + g) \circ (h + i) = f \circ h + g \circ i \qquad \text{Functor-+}$$
$$\mathsf{id} + \mathsf{id} = \mathsf{id} \qquad \text{Functor-Id-+}$$

Exercício 37 *Demonstre as propriedades* Absor-×, Functor-× *e* Functor-Id-× *usando cálculo equacional.*

Exercício 38 *Demonstre que* coswap *é uma transformação natural, ou seja, que verifica a seguinte propriedade para quaisquer f e g.*

$$\mathsf{coswap} \circ (f + g) = (g + f) \circ \mathsf{coswap}$$

Para finalizar esta secção vamos apresentar mais um exemplo de derivação de uma definição *point-free*. Considere o isomorfismo $a \times (b + c) \cong (a \times b) + (a \times c)$. A função que testemunha o isomorfismo da direita para a esquerda pode ser definida em Haskell da seguinte forma.

```
undistr :: Either (a, b) (a, c) → (a, Either b c)
undistr (Left (x, y))  = (x, Left y)
undistr (Right (x, y)) = (x, Right y)
```

Usando cálculo podemos converter esta função para *point-free* da seguinte forma.

$$\begin{array}{ll}
& \mathsf{undistr}\ (\mathit{Left}\ (x, y)) = (x, \mathit{Left}\ y) \wedge \mathsf{undistr}\ (\mathit{Right}\ (x, y)) = (x, \mathit{Right}\ y) \\
\Leftrightarrow & \quad \{\ \text{Def-inl, Def-inr}\ \} \\
& \mathsf{undistr}\ (\mathsf{inl}\ (x, y)) = (x, \mathsf{inl}\ y) \wedge \mathsf{undistr}\ (\mathsf{inr}\ (x, y)) = (x, \mathsf{inr}\ y) \\
\Leftrightarrow & \quad \{\ \text{Def-}\circ\ \} \\
& (\mathsf{undistr} \circ \mathsf{inl})\ (x, y) = (x, \mathsf{inl}\ y) \wedge (\mathsf{undistr} \circ \mathsf{inr})\ (x, y) = (x, \mathsf{inr}\ y) \\
\Leftrightarrow & \quad \{\ \text{Elim-}\times\ \} \\
& (\mathsf{undistr} \circ \mathsf{inl})\ z = (\mathsf{fst}\ z, \mathsf{inl}\ (\mathsf{snd}\ z)) \wedge (\mathsf{undistr} \circ \mathsf{inr})\ z = (\mathsf{fst}\ z, \mathsf{inr}\ (\mathsf{snd}\ z)) \\
\Leftrightarrow & \quad \{\ \text{Def-}\circ\ \} \\
& (\mathsf{undistr} \circ \mathsf{inl})\ z = (\mathsf{fst}\ z, (\mathsf{inl} \circ \mathsf{snd})\ z) \wedge (\mathsf{undistr} \circ \mathsf{inr})\ z = (\mathsf{fst}\ z, (\mathsf{inr} \circ \mathsf{snd})\ z) \\
\Leftrightarrow & \quad \{\ \text{Def-}\triangle\ \} \\
& (\mathsf{undistr} \circ \mathsf{inl})\ z = (\mathsf{fst} \triangle \mathsf{inl} \circ \mathsf{snd})\ z \wedge (\mathsf{undistr} \circ \mathsf{inr})\ z = (\mathsf{fst} \triangle \mathsf{inr} \circ \mathsf{snd})\ z \\
\Leftrightarrow & \quad \{\ \text{Ext-=}\ \} \\
& \mathsf{undistr} \circ \mathsf{inl} = \mathsf{fst} \triangle \mathsf{inl} \circ \mathsf{snd} \wedge \mathsf{undistr} \circ \mathsf{inr} = \mathsf{fst} \triangle \mathsf{inr} \circ \mathsf{snd} \\
\Leftrightarrow & \quad \{\ \text{Def-}\times\ \} \\
& \mathsf{undistr} \circ \mathsf{inl} = \mathsf{id} \times \mathsf{inl} \wedge \mathsf{undistr} \circ \mathsf{inr} = \mathsf{id} \times \mathsf{inr} \\
\Leftrightarrow & \quad \{\ \text{Univ-+}\ \} \\
& \mathsf{undistr} = (\mathsf{id} \times \mathsf{inl}) \triangledown (\mathsf{id} \times \mathsf{inr})
\end{array}$$

Naturalmente, também é possível determinar a definição do undistr usando diagramas. Na figura 3.2 é apresentada uma possível derivação. Começamos por desenhar o diagrama a) que nos indica o tipo da função pretendida. Como esta função tem como

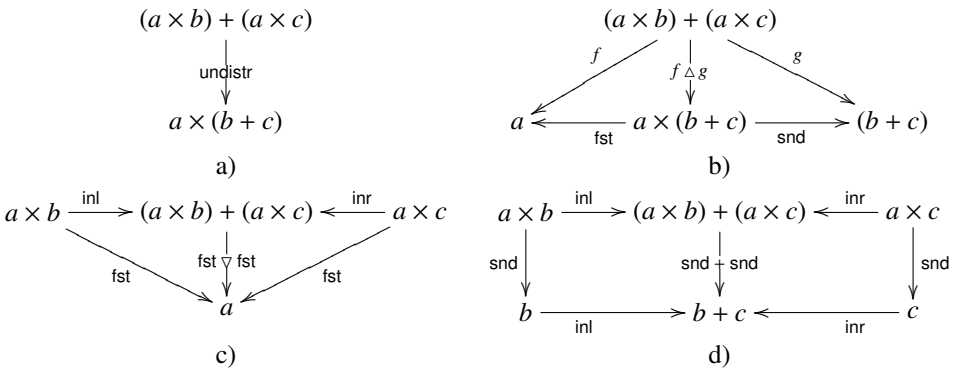


Figura 3.2: Derivação do undistr usando diagramas.

domínio uma soma e como contra-domínio um produto, em princípio será possível defini-la usando um *split* ou um *either*. Neste caso optámos pela primeira hipótese, que dá origem ao diagrama b). Para conseguir a definição do undistr temos agora que obter definições para as funções f e g. Dado que o domínio da primeira é uma soma podemos obter a sua definição recorrendo ao diagrama c). Para a segunda, como tanto o domínio como o contra-domínio são somas, é mais directo tentar uma definição usando o combinador de soma de funções, sendo obtido o diagrama c). Após este exercício chegamos à seguinte definição para o undistr.

$$\mathsf{undistr} = (\mathsf{fst} \triangledown \mathsf{fst}) \vartriangle (\mathsf{snd} + \mathsf{snd}) \qquad \textsc{Def}\text{-undistr}$$

Como optámos pela definição usando um *split* obtivemos uma definição diferente da calculada a partir da definição *pointwise*. No entanto, recorrendo à seguinte lei da troca é possível demonstrar que são idênticas.

$$(f \vartriangle g) \triangledown (h \vartriangle i) = (f \triangledown h) \vartriangle (g \triangledown i) \qquad \textsc{Troca}$$

A demonstração é muito simples:

$$\begin{array}{ll} & (\mathsf{fst} \triangledown \mathsf{fst}) \vartriangle (\mathsf{snd} + \mathsf{snd}) \\ = & \quad \{ \textsc{Def}\text{-}+ \} \\ & (\mathsf{fst} \triangledown \mathsf{fst}) \vartriangle (\mathsf{inl} \circ \mathsf{snd} \triangledown \mathsf{inr} \circ \mathsf{snd}) \\ = & \quad \{ \textsc{Troca} \} \\ & (\mathsf{fst} \vartriangle \mathsf{inl} \circ \mathsf{snd}) \triangledown (\mathsf{fst} \vartriangle \mathsf{inr} \circ \mathsf{snd}) \\ = & \quad \{ \textsc{Def}\text{-}\times \} \\ & (\mathsf{id} \times \mathsf{inl}) \triangledown (\mathsf{id} \times \mathsf{inr}) \end{array}$$

Exercício 39 *Demonstre a lei da troca.*

Exercício 40 *Demonstre que* undistr *é uma transformação natural, ou seja, que verifica a seguinte propriedade para quaisquer f, g e h.*

$$\mathsf{undistr} \circ (f \times g + f \times h) = (f \times (g + h)) \circ \mathsf{undistr}$$

3.4 Constantes e Condicionais

No lote dos tipos não recursivos primitivos é fundamental incluir um tipo que apenas contém um habitante. Em Haskell vamos usar o seguinte tipo de dados pré-definido para o representar.

data () = ()

Tal como no caso das somas e dos produtos, este tipo pode ser caracterizado intensionalmente recorrendo ao conceito de unicidade: o tipo 1 é caracteriza-se por existir apenas uma função $\mathsf{bang} : a \to 1$ para qualquer tipo a. Esta propriedade pode ser capturada na seguinte lei universal.

$$f = \mathsf{bang} \qquad \textsc{Univ}\text{-}1$$

À primeira vista esta lei parece absurda. No entanto, se tivermos em consideração os tipos das funções envolvidas torna-se mais clara. Numa igualdade os tipos das funções comparadas tem que ser idênticos. Dado que $\mathsf{bang} :: a \to 1$, isto implica que o tipo de f seja também $a \to 1$. Sendo assim, o que nos diz a lei Univ-1 é que qualquer função de tipo $a \to 1$ tem necessariamente que ser igual a bang, pois apenas existe um resultado possível. Em Haskell, o combinador bang pode ser implementado da seguinte forma:

$$\begin{array}{l} \mathsf{bang} :: a \to () \\ \mathsf{bang}\ x = () \end{array} \qquad \textsc{Def}\text{-}\mathsf{bang}$$

Como neste tipo não existem projecções ou injecções, como no caso dos produtos e das somas, não vai existir lei de cancelamento: apenas temos reflexão e fusão.

$$\mathsf{bang} = \mathsf{id} \qquad \textsc{Reflex}\text{-}1$$

$$\mathsf{bang} \circ f = \mathsf{bang} \qquad \textsc{Fusion}\text{-}1$$

Mais uma vez, é necessário ter atenção aos tipos quando se usa a lei de reflexão: neste caso apenas se aplica a funções de tipo $1 \to 1$.

Como exemplo de utilização destas leis, vamos demonstrar o isomorfismo $a \cong 1 \times a$. Neste caso é relativamente simples determinar as funções que o testemunham: da esquerda para a direita temos $\mathsf{bang} \vartriangle \mathsf{id}$ e da direita para a esquerda snd. Provar $\mathsf{snd} \circ (\mathsf{bang} \vartriangle \mathsf{id}) = \mathsf{id}$ é imediato a partir da lei Cancel-×. A outra prova faz-se da seguinte forma.

$$\begin{array}{ll} & (\mathsf{bang} \vartriangle \mathsf{id}) \circ \mathsf{snd} \\ = & \quad \{\ \textsc{Fusion}\text{-}\times, \textsc{Nat}\text{-}\mathsf{id}\ \} \\ & \mathsf{bang} \circ \mathsf{snd} \vartriangle \mathsf{snd} \\ = & \quad \{\ \textsc{Fusion}\text{-}1\ \} \\ & \mathsf{bang} \vartriangle \mathsf{snd} \\ = & \quad \{\ \textsc{Fusion}\text{-}1\ \} \\ & \mathsf{bang} \circ \mathsf{fst} \vartriangle \mathsf{snd} \\ = & \quad \{\ \textsc{Reflex}\text{-}1, \textsc{Nat}\text{-}\mathsf{id}\ \} \\ & \mathsf{fst} \vartriangle \mathsf{snd} \end{array}$$

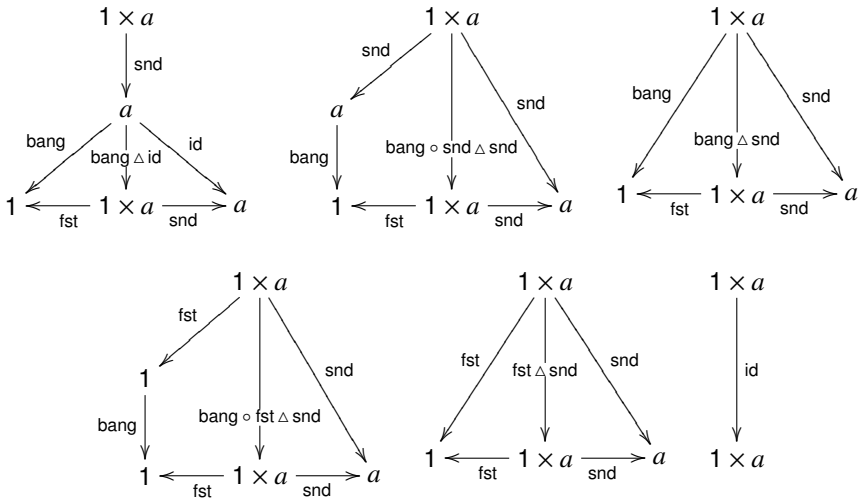


Figura 3.3: Demonstração do isomorfismo $a \cong 1 \times a$.

$$
\begin{array}{ll}
= & \{ \text{REFLEX-}\times \} \\
& \mathsf{id}
\end{array}
$$

Para que se perceba a aplicação das leis FUSION-1 e REFLEX-1 é necessário saber os tipos das funções envolvidas. Uma forma conveniente de o fazer consiste na utilização dos diagramas. Na figura 3.3 são apresentados sucessivamente todos os diagramas intermédios correspondentes a esta prova, ficando claro o contexto em que cada lei se aplica. Note que o terceiro e quarto passos poderiam ser condensados apenas num usando a lei universal: a função $\mathsf{bang} :: 1 \times a \to 1$ é igual a qualquer outra função deste tipo, nomeadamente a função $\mathsf{fst} :: 1 \times a \to 1$

Exercício 41 *Demonstre a seguinte igualdade:*

$$(\mathsf{id} \vartriangle \mathsf{inl} \circ \mathsf{bang}) \triangledown (\mathsf{id} \times \mathsf{inr}) \circ \mathsf{swap} = (\mathsf{id} \triangledown \mathsf{snd}) \vartriangle (\mathsf{bang} + \mathsf{fst})$$

Qual o isomorfismo que esta função estabelece? Justifique através de um diagrama ilustrativo.

Para introduzir constantes arbitrárias no cálculo *point-free* vamos usar o seguinte combinador:

$$
\begin{array}{l}
(\underline{\cdot}) :: a \to () \to a \\
\underline{x}\ () = x
\end{array}
\qquad \text{DEF-const}
$$

Dada uma constante $x :: a$, $\underline{x}$ é uma função de tipo $1 \to a$ que quando aplicada ao único argumento possível devolve a própria constante. Para obter uma função $a \to b$ que aceita qualquer a e devolve sempre a constante $x :: b$ basta definir $\underline{x} \circ \mathsf{bang}$.

O tipo $2 = 1 + 1$ é útil para manipular booleanos no estilo *point-free*. O isomorfismo $Bool \cong 2$ pode ser testemunhado pelas seguintes funções.

$$
\begin{aligned}
&\mathsf{out}_B :: \mathit{Bool} \to \mathit{Either}\ ()\ () \\
&\mathsf{out}_B\ \mathit{True}\ \ = \mathit{Left}\ () \\
&\mathsf{out}_B\ \mathit{False} = \mathit{Right}\ () \\
&\mathsf{in}_B :: \mathit{Either}\ ()\ () \to \mathit{Bool} \\
&\mathsf{in}_B\ (\mathit{Left}\ ())\ \ \ = \mathit{True} \\
&\mathsf{in}_B\ (\mathit{Right}\ ()) = \mathit{False}
\end{aligned}
$$

Esta técnica de usar um isomorfismo entre um tipo de dados e uma representação usando produtos, somas e o tipo unitário (uma representação dita *polinomial*), vai ser muitas vezes usada para permitir trabalhar com esse tipo usando combinadores *point-free*. A função que converte de um dado tipo a para a representação polinomial chama-se out_a e a função inversa in_a. Para facilitar a leitura serão usadas abreviaturas para os tipos nas anotações. Como estabelecem um isomorfismo temos a seguinte lei:

$$\mathsf{in}_a \circ \mathsf{out}_a = \mathsf{id} = \mathsf{out}_a \circ \mathsf{in}_a \qquad \text{Iso-in-out}$$

Recorrendo a este isomorfismo podemos escrever funções sobre booleanos usando os combinadores *point-free* já apresentados. Por exemplo a função de negação pode ser implementada da seguinte forma:

$$
\begin{aligned}
&\mathsf{not} :: \mathit{Bool} \to \mathit{Bool} \\
&\mathsf{not} = \mathsf{in}_B \circ \mathsf{coswap} \circ \mathsf{out}_B
\end{aligned}
$$

Usando a noção de constante é possível obter uma definição *point-free* para in_B:

$$
\begin{aligned}
&\ \ \mathsf{in}_B\ (\mathit{Left}\ ()) = \mathit{True} \wedge \mathsf{in}_B\ (\mathit{Right}\ ()) = \mathit{False} \\
\Leftrightarrow\ &\quad \{\ \textsc{Def}\text{-}\mathsf{inl},\ \textsc{Def}\text{-}\mathsf{inr}\ \} \\
&\ \ \mathsf{in}_B\ (\mathsf{inl}\ ()) = \mathit{True} \wedge \mathsf{in}_B\ (\mathsf{inr}\ ()) = \mathit{False} \\
\Leftrightarrow\ &\quad \{\ \textsc{Def}\text{-}\mathsf{const}\ \} \\
&\ \ \mathsf{in}_B\ (\mathsf{inl}\ ()) = \underline{\mathit{True}}\ () \wedge \mathsf{in}_B\ (\mathsf{inr}\ ()) = \underline{\mathit{False}}\ () \\
\Leftrightarrow\ &\quad \{\ \textsc{Def}\text{-}\circ\ \} \\
&\ \ (\mathsf{in}_B \circ \mathsf{inl})\ () = \underline{\mathit{True}}\ () \wedge (\mathsf{in}_B \circ \mathsf{inr})\ () = \underline{\mathit{False}}\ () \\
\Leftrightarrow\ &\quad \{\ \textsc{Ext}\text{-}=\ \} \\
&\ \ \mathsf{in}_B \circ \mathsf{inl} = \underline{\mathit{True}} \wedge \mathsf{in}_B \circ \mathsf{inr} = \underline{\mathit{False}} \\
\Leftrightarrow\ &\quad \{\ \textsc{Univ}\text{-}+\ \} \\
&\ \ \mathsf{in}_B = \underline{\mathit{True}} \triangledown \underline{\mathit{False}}
\end{aligned}
$$

Exercício 42 *Relembre a definição do tipo Maybe:*

data *Maybe a = Nothing | Just a*

Escreva as funções $\mathsf{out}_M :: \mathit{Maybe}\ a \to 1 + a$ *e* $\mathsf{in}_M :: 1 + a \to \mathit{Maybe}\ a$ *que estabelecem o isomorfismo entre este tipo e a representação polinomial* $1 + a$*. Derive uma versão* point-free *para a função* in_M*.*

Outro isomorfismo muito útil é $2 \times a \cong a + a$. A função que testemunha o ismorfismo da direita para a esquerda pode ser definida em Haskell da seguinte forma:

$$\begin{aligned}&\mathsf{distwo} :: \mathit{Either}\ a\ a \to (\mathit{Either}\ ()\ (), a)\\ &\mathsf{distwo}\ (\mathit{Left}\ x) \quad = (\mathit{Left}\ (), x)\\ &\mathsf{distwo}\ (\mathit{Right}\ x) = (\mathit{Right}\ (), x)\end{aligned}$$

A partir desta definição *pointwise* podemos derivar a respectiva definição *point-free*.

$$\begin{aligned}&\quad \mathsf{distwo}\ (\mathit{Left}\ x) = (\mathit{Left}\ (), x) \wedge \mathsf{distwo}\ (\mathit{Right}\ x) = (\mathit{Right}\ (), x)\\ \Leftrightarrow &\quad \{\ \textsc{Def}\text{-inl}, \textsc{Def}\text{-inr}\ \}\\ &\quad \mathsf{distwo}\ (\mathsf{inl}\ x) = (\mathsf{inl}\ (), x) \wedge \mathsf{distwo}\ (\mathsf{inr}\ x) = (\mathsf{inr}\ (), x)\\ \Leftrightarrow &\quad \{\ \textsc{Def}\text{-bang}\ \}\\ &\quad \mathsf{distwo}\ (\mathsf{inl}\ x) = (\mathsf{inl}\ (\mathsf{bang}\ x), x) \wedge \mathsf{distwo}\ (\mathsf{inr}\ x) = (\mathsf{inr}\ (\mathsf{bang}\ x), x)\\ \Leftrightarrow &\quad \{\ \textsc{Def}\text{-}\circ\ \}\\ &\quad (\mathsf{distwo} \circ \mathsf{inl})\ x = ((\mathsf{inl} \circ \mathsf{bang})\ x, x) \wedge (\mathsf{distwo} \circ \mathsf{inr})\ x = ((\mathsf{inr} \circ \mathsf{bang})\ x, x)\\ \Leftrightarrow &\quad \{\ \textsc{Def}\text{-}\triangle\ \}\\ &\quad (\mathsf{distwo} \circ \mathsf{inl})\ x = (\mathsf{inl} \circ \mathsf{bang} \triangle \mathsf{id})\ x \wedge (\mathsf{distwo} \circ \mathsf{inr})\ x = (\mathsf{inr} \circ \mathsf{bang} \triangle \mathsf{id})\ x\\ \Leftrightarrow &\quad \{\ \textsc{Ext}\text{-}=\ \}\\ &\quad \mathsf{distwo} \circ \mathsf{inl} = \mathsf{inl} \circ \mathsf{bang} \triangle \mathsf{id} \wedge \mathsf{distwo} \circ \mathsf{inr} = \mathsf{inr} \circ \mathsf{bang} \triangle \mathsf{id}\\ \Leftrightarrow &\quad \{\ \textsc{Univ}\text{-}+\ \}\\ &\quad \mathsf{distwo} = (\mathsf{inl} \circ \mathsf{bang} \triangle \mathsf{id}) \triangledown (\mathsf{inr} \circ \mathsf{bang} \triangle \mathsf{id})\\ \Leftrightarrow &\quad \{\ \textsc{Troca}\ \}\\ &\quad \mathsf{distwo} = (\mathsf{inl} \circ \mathsf{bang} \triangledown \mathsf{inr} \circ \mathsf{bang}) \triangle (\mathsf{id} \triangledown \mathsf{id})\\ \Leftrightarrow &\quad \{\ \textsc{Def}\text{-}+\ \}\\ &\quad \mathsf{distwo} = (\mathsf{bang} + \mathsf{bang}) \triangle (\mathsf{id} \triangledown \mathsf{id})\end{aligned}$$

Dado que não foram apresentados os combinadores *point-free* para lidar com exponenciais (funções) não é possível definir o inverso directamente. No entanto, assumindo a existência da função $\mathsf{distl} :: (a + b) \times c \to (a \times c) + (b \times c)$ podemos definir essa função da seguinte forma.

$$\begin{aligned}&\mathsf{undistwo} :: 2 \times a \to a + a\\ &\mathsf{undistwo} = (\mathsf{snd} + \mathsf{snd}) \circ \mathsf{distl}\end{aligned}$$

Este isomorfismo permite-nos definir um combinador de guarda, que dado um predicado $a \to \mathit{Bool}$ cria uma função $a \to a + a$, que dado um valor de tipo a injecta esse valor no lado esquerdo da soma caso o predicado se verifique, ou no lado direito em caso contrário. Este combinador é muito útil pois permite testar um predicado sem perder o valor testado. A sua definição é muito simples se recorrermos à definição de undistwo apresentada acima, e ao diagrama da figura 3.4.

$$\begin{aligned}&(\cdot?) :: (a \to \mathit{Bool}) \to (a \to a + a)\\ &p? = (\mathsf{snd} + \mathsf{snd}) \circ \mathsf{distl} \circ (\mathsf{out}_B \circ p \triangle \mathsf{id})\end{aligned} \qquad \textsc{Def}\text{-guard}$$

Este combinador satisfaz a seguinte lei de fusão:

$$p? \circ f = (f + f) \circ (p \circ f)? \qquad \textsc{Fusion}\text{-guard}$$

Para demonstrar esta lei é necessário recorrer à propriedade natural da função distl.

$$\mathsf{distl} \circ ((f + g) \times h) = ((f \times h) + (g \times h)) \circ \mathsf{distl} \qquad \textsc{Nat}\text{-distl}$$

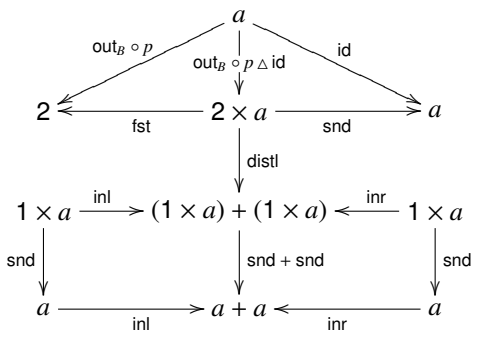


Figura 3.4: Derivação da definição de guarda.

Dada esta propriedade, a lei Fusion-guard prova-se da seguinte forma:

$$
\begin{aligned}
& p? \circ f \\
= \quad & \{ \text{Def-guard} \} \\
& (\mathsf{snd} + \mathsf{snd}) \circ \mathsf{distl} \circ (\mathsf{out}_B \circ p \vartriangle \mathsf{id}) \circ f \\
= \quad & \{ \text{Fusion-}\times, \text{Nat-id} \} \\
& (\mathsf{snd} + \mathsf{snd}) \circ \mathsf{distl} \circ (\mathsf{out}_B \circ p \circ f \vartriangle f) \\
= \quad & \{ \text{Nat-id, Absor-}\times \} \\
& (\mathsf{snd} + \mathsf{snd}) \circ \mathsf{distl} \circ (\mathsf{id} \times f) \circ (\mathsf{out}_B \circ p \circ f \vartriangle \mathsf{id}) \\
= \quad & \{ \text{Functor-Id-+} \} \\
& (\mathsf{snd} + \mathsf{snd}) \circ \mathsf{distl} \circ ((\mathsf{id} + \mathsf{id}) \times f) \circ (\mathsf{out}_B \circ p \circ f \vartriangle \mathsf{id}) \\
= \quad & \{ \text{Nat-distl} \} \\
& (\mathsf{snd} + \mathsf{snd}) \circ ((\mathsf{id} \times f) + (\mathsf{id} \times f)) \circ \mathsf{distl} \circ (\mathsf{out}_B \circ p \circ f \vartriangle \mathsf{id}) \\
= \quad & \{ \text{Functor-+} \} \\
& (\mathsf{snd} \circ (\mathsf{id} \times f) + \mathsf{snd} \circ (\mathsf{id} \times f)) \circ \mathsf{distl} \circ (\mathsf{out}_B \circ p \circ f \vartriangle \mathsf{id}) \\
= \quad & \{ \text{Def-}\times, \text{Cancel-}\times \} \\
& (f \circ \mathsf{snd} + f \circ \mathsf{snd}) \circ \mathsf{distl} \circ (\mathsf{out}_B \circ p \circ f \vartriangle \mathsf{id}) \\
= \quad & \{ \text{Functor-+} \} \\
& (f + f) \circ (\mathsf{snd} + \mathsf{snd}) \circ \mathsf{distl} \circ (\mathsf{out}_B \circ p \circ f \vartriangle \mathsf{id}) \\
= \quad & \{ \text{Def-guard} \} \\
& (f + f) \circ (p \circ f)?
\end{aligned}
$$

Recorrendo à guarda é trivial definir um condicional *point-free*:

$$
\begin{aligned}
& (\cdot \to \cdot, \cdot) :: (a \to \mathit{Bool}) \to (a \to b) \to (a \to b) \to (a \to b) \\
& p \to f, g = (f \triangledown g) \circ p?
\end{aligned}
\qquad \text{Def-cond}
$$

Para efeitos de derivação de código *point-free* é conveniente dar também uma definição *pointwise* deste combinador:

$$
\begin{aligned}
& (\cdot \to \cdot, \cdot) :: (a \to \mathit{Bool}) \to (a \to b) \to (a \to b) \to (a \to b) \\
& p \to f, g\ x = \mathbf{if}\ p\ x\ \mathbf{then}\ f\ x\ \mathbf{else}\ g\ x
\end{aligned}
\qquad \text{Def-Pw-cond}
$$

O condicional satisfaz duas leis de fusão:

$$f \circ (p \to g, h) = p \to (f \circ g), (f \circ h) \qquad \text{Fusion-L-cond}$$
$$(p \to f, g) \circ h = (p \circ h) \to (f \circ h), (g \circ h) \qquad \text{Fusion-R-cond}$$

Exercício 43 *Demonstre as leis* Fusion-L-cond *e* Fusion-R-cond.

Bibliografia

[1] *GHC Documentation*. `http://www.haskell.org/ghc/documentation.html`.

[2] J.C. Bacelar, L.S. Barbosa, J.B. Barros, A. Cunha, M.J. Frade, J.N. Oliveira F.L. Neves, and J. S. Pinto. *Métodos de Programação I - Exercícios*. Departamento de Informática, Universidade do Minho, Dezembro 2004.

[3] Richard Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 2nd edition, 1998.

[4] Jeff Newbern. *All About Monads*. `http://www.nomaware.com/monads/html`.

[5] José Nuno Oliveira. *Program Design by Calculation*. Departamento de Informática, Universidade do Minho, 2005. Draft.

Apêndice A

Leis de cálculo

$$f = g \Leftrightarrow \forall\, x \,.\, f\ x = g\ x \qquad \text{Ext-=}$$
$$f \circ h = g \circ h \Leftarrow f = g \qquad \text{Leibniz}$$

$$(f \circ g)\ x = f\ (g\ x) \qquad \text{Def-}\circ$$
$$(f \circ g) \circ h = f \circ (g \circ h) \qquad \text{Assoc-}\circ$$
$$\mathsf{id}\ x = x \qquad \text{Def-id}$$
$$f \circ \mathsf{id} = f = \mathsf{id} \circ f \qquad \text{Nat-id}$$

$$h = f \vartriangle g \Leftrightarrow \mathsf{fst} \circ h = f \wedge \mathsf{snd} \circ h = g \qquad \text{Univ-}\times$$
$$\mathsf{fst}\ (x, y) = x \qquad \text{Def-fst}$$
$$\mathsf{snd}\ (x, y) = y \qquad \text{Def-snd}$$
$$(f \vartriangle g)\ x = (f\ x, g\ x) \qquad \text{Def-}\vartriangle$$
$$f\ a = b \Leftrightarrow f\ a\ [(x, y)\ /\ z] = b\ [x\ /\ \mathsf{fst}\ z, y\ /\ \mathsf{snd}\ z] \qquad \text{Elim-}\times$$
$$\mathsf{fst} \circ (f \vartriangle g) = f \wedge \mathsf{snd} \circ (f \vartriangle g) = g \qquad \text{Cancel-}\times$$
$$\mathsf{fst} \vartriangle \mathsf{snd} = \mathsf{id} \qquad \text{Reflex-}\times$$
$$(f \vartriangle g) \circ h = f \circ h \vartriangle g \circ h \qquad \text{Fusion-}\times$$
$$f \times g = f \circ \mathsf{fst} \vartriangle g \circ \mathsf{snd} \qquad \text{Def-}\times$$
$$(f \times g) \circ (h \vartriangle i) = f \circ h \vartriangle g \circ i \qquad \text{Absor-}\times$$
$$(f \times g) \circ (h \times i) = f \circ h \times g \circ i \qquad \text{Functor-}\times$$
$$\mathsf{id} \times \mathsf{id} = \mathsf{id} \qquad \text{Functor-Id-}\times$$

$$h = f \triangledown g \Leftrightarrow h \circ \mathsf{inl} = f \wedge h \circ \mathsf{inr} = g \qquad \textsc{Univ}\text{-}+$$

$$\mathsf{inl}\ x = \mathit{Left}\ x \qquad \textsc{Def}\text{-}\mathsf{inl}$$

$$\mathsf{inr}\ x = \mathit{Right}\ x \qquad \textsc{Def}\text{-}\mathsf{inr}$$

$$(f \triangledown g)\ x = \mathbf{case}\ x\ \mathbf{of}\ \{\mathit{Left}\ y \to f\ y; \mathit{Right}\ z \to g\ z\} \qquad \textsc{Def}\text{-}\triangledown$$

$$(f \triangledown g) \circ \mathsf{inl} = f \wedge (f \triangledown g) \circ \mathsf{inr} = g \qquad \textsc{Cancel}\text{-}+$$

$$\mathsf{inl} \triangledown \mathsf{inr} = \mathsf{id} \qquad \textsc{Reflex}\text{-}+$$

$$f \circ (g \triangledown h) = f \circ g \triangledown f \circ h \qquad \textsc{Fusion}\text{-}+$$

$$f + g = \mathsf{inl} \circ f \triangledown \mathsf{inr} \circ g \qquad \textsc{Def}\text{-}+$$

$$(f \triangledown g) \circ (h + i) = f \circ h \triangledown g \circ i \qquad \textsc{Absor}\text{-}+$$

$$(f + g) \circ (h + i) = f \circ h + g \circ i \qquad \textsc{Functor}\text{-}+$$

$$\mathsf{id} + \mathsf{id} = \mathsf{id} \qquad \textsc{Functor-Id}\text{-}+$$

$$(f \triangle g) \triangledown (h \triangle i) = (f \triangledown h) \triangle (g \triangledown i) \qquad \textsc{Troca}$$

$$f = \mathsf{bang} \qquad \textsc{Univ}\text{-}1$$

$$\mathsf{bang}\ x = () \qquad \textsc{Def}\text{-}\mathsf{bang}$$

$$\mathsf{bang} = \mathsf{id} \qquad \textsc{Reflex}\text{-}1$$

$$\mathsf{bang} \circ f = \mathsf{bang} \qquad \textsc{Fusion}\text{-}1$$

$$\underline{x}\ () = x \qquad \textsc{Def}\text{-const}$$

$$\mathsf{in}_a \circ \mathsf{out}_a = \mathsf{id} = \mathsf{out}_a \circ \mathsf{in}_a \qquad \textsc{Iso}\text{-in-out}$$

$$p? = (\mathsf{snd} + \mathsf{snd}) \circ \mathsf{distl} \circ (\mathsf{out}_B \circ p \triangle \mathsf{id}) \qquad \textsc{Def}\text{-guard}$$

$$p? \circ f = (f + f) \circ (p \circ f)? \qquad \textsc{Fusion}\text{-guard}$$

$$p \to f, g = (f \triangledown g) \circ p? \qquad \textsc{Def}\text{-cond}$$

$$p \to f, g\ x = \mathbf{if}\ p\ x\ \mathbf{then}\ f\ x\ \mathbf{else}\ g\ x \qquad \textsc{Def-Pw}\text{-cond}$$

$$f \circ (p \to g, h) = p \to (f \circ g), (f \circ h) \qquad \textsc{Fusion-L}\text{-cond}$$

$$(p \to f, g) \circ h = (p \circ h) \to (f \circ h), (g \circ h) \qquad \textsc{Fusion-R}\text{-cond}$$