Aula 3: os monads *IO* e *Reader*.

Alcino Cunha

2 de Outubro de 2006

1 A caixa mágica do *IO*

Em Haskell todas as operções de *input/output* tem que ser realizadas usando o monad *IO*. Tal como vimos na aula passada, monads são "caixas" que contém resultados. É possível construir uma caixa com um resultado usando a função *return* e aplicar uma função monádica a uma caixa usando a função $\gg=$. Estas funções estão declaradas na classe *Monad*, cuja assinatura já conhecemos:

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

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.

```
putStr :: String → IO ()
putStr [ ]     = return ()
putStr (h : t) = putChar h >>= (λ_ → putStr t)
```

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.

```
(>>) :: Monad m ⇒ m a → m b → m b
f >> g = f >>= (λ_ → g)
```

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

```
putStr :: String → IO ()
putStr [ ]     = return ()
putStr (h : t) = putChar h >> putStr t
```

Exercício 1 *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 2 *Usando a função read e o getLine, definido no exercício anterior, defina uma computação getInt* :: *IO Int que lê um inteiro do teclado.*

Exercício 3 *Relembre o tipo Exp declarado na aula anterior.*

```
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.

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

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 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.

$$runReader :: Reader\ c\ a \rightarrow (c \rightarrow a)$$

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 4 *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 :: $b \rightarrow Tree\ a \rightarrow Tree\ b$ *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* $Tree\ a \rightarrow Reader\ b\ (Tree\ b)$.