Aula 2: introdução aos *monads*.

Alcino Cunha

21 de Setembro de 2006

1 Introdução

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 :: $(Eq\ a) \Rightarrow a \rightarrow [(a, b)] \rightarrow Maybe\ b$ 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

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

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 1 *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 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 2 *Defina um operador de aplicação para o tipo de dados Erro e reescreva a função evale usando esse operador.*

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

Exercício 4 *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

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

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.

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

Exercício 6 *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]
```