Resumo Paradigmas de programação - Professor Emílio
Aluno - Diogo Miyake - RA 21048813

Semana 2 Cálculo Lambda e Conceitos Básicos

Cálculo Lambda : A grosso modo é uma anotação simples para aplicações e funções, também pode ser chamado “a menor linguagem de programação universal” do mundo. surgiu na década de 1930, foi criado por Alonzo Church, alguns dos problemas associados ao cálculo lambda são:
Decisão: Exemplo: testar se um número é primo,
Função: Exemplo: inverter uma string.
Busca: Exemplo: buscar um clique em um grafo.
Otimização: Exemplo: quanto devo colocar em cada possível investimento para maximizar os lucros.
Alguns Exemplos:
A expressão Lambda se estende para a direita
λ f. x y ≡ λ f.(x y)
A aplicação é associativa à esquerda
x y z ≡ (x y) z
Múltiplos lambdas podem ser omitidos
λ f g. x ≡ λ f . λ g. x
β-redução: é uma abstração que implica na substituição das ocorrências das variáveis correspondentes ao argumento
Exemplo
(λ x. + x 1) 4 → + 4 1
Funções também podem ser passadas como argumento:
(λ f. f 6) (λ x. + x 1) → (λ x. + x 1) 6 → + 6 1 → 7
Conceitos Básicos:
Surgiu em 1990 com o objetivo de ser a primeira linguagem puramente funcional.
Por muito tempo considerada uma linguagem acadêmica.
Atualmente é utilizada em diversas empresas (totalmente ou em parte de projetos).
Sobre os recursos da linguagem foram abordados:

- Imutabilidade;
- Funções recursivas (sem laços);
- Funções de Alta Ordem (podem receber funções como parâmetro);
- Tipos polimórficos;
- Avaliação preguiçosa;
- Precedência: Funções > Operadores;
- Blocos definidos por indentação;
- Definição: v :: T;
- Verificar tipo: ":t expressão/variável";
- Listas contém elementos do mesmo tipo;
- Tuplas são finitas e contém tipos diferentes;
- Classe de Tipo;
- Instância de Tipo;
- Assinaturas de funções;
- Operações em Listas: !!, tail, head, take, drop, length, sum, product, zip;
- Compreensão de Listas;

Resumo Paradigmas de programação - Professor Emílio
Aluno - Diogo Miyake - RA 21048813

Semana 3 - QuickCheck e Funções de alta Ordem

QuickCheck
Biblioteca para testes a partir de propriedades definidas.

Funções de alta ordem: funções que podem ser passadas como argumento ou serem retornadas de uma função.
Um exemplo mostrado é a função soma, onde ela pode ter sua assinatura reescrita retornando uma função, abaixo o exemplo mostrado:
De isso

```
soma :: Int -> Int -> Int
soma x y = x+y
```

Para isso:

```
soma :: Int -> (Int -> Int)
soma x y = x+y
```

Algumas funções mostradas são MAP, FILTER, FOLDR, FOLDL

```
map (+1) [1,2,3]
[2,3,4]
```

```
filter even [1..10]
[2,4,6,8,10]
```

```
all even [2,4,6,8]
True
```

```
any odd [2,4,6,8]
False
```

```
 takeWhile even [2,4,6,7,8]
[2,4,6]
```

```
dropWhile even [2,4,6,7,8]
[7,8]
```

Fold: dobra a lista aplicando a função f em cada elemento da lista e um resultado parcial.

```
foldr (+) 0 [1,2,3]    —>>> se torna --->>>>  1 + (2 + (3 + 0))
```

Beleza, agora vamos dar uma olhada na função $, também chamada de aplicação de função. Antes de qualquer coisa, vamos ver como isto é definido:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Resumo Paradigmas de programação - Professor Emílio
Aluno - Diogo Miyake - RA 21048813

Considere a expressão **sum (map sqrt [1..130])**. Como **$** tem baixa precedência nós podemos reescrever esta expressão como **sum $ map sqrt [1..130],**

Folding
dobra a lista aplicando a função f em cada elemento da lista e um resultado parcial. Quando usar o foldr e foldl?
Uma regra do dedão para trabalharmos por enquanto é:
- Se a lista passada como argumento é infinita, use foldr
- Se o operador utilizado pode gerar curto-circuito,
use foldr
- Se a lista é finita e o operador não irá gerar curto-circuito, use foldl - Se faz sentido trabalhar com a lista invertida, use foldl

- Bônus! E temos ainda uma outra função chamada foldl' que aprenderemos mais para frente.
Composição de funções significa aplicar uma função a um argumento e, depois, aplicar outra função ao resultado da primeira função.
Em haskell é utilizado a assinatura (.)
Ela pode ser associativa, e ter um elemento neutro. Na matemática a composição de função $f \circ g$ define uma nova função z tal que $z(x)=f(g(x))$.

Resumo Paradigmas de programação - Professor Emílio
Aluno - Diogo Miyake - RA 21048813

Semana 4 - ADT - Tipos de Dados Algébricos

Type permite criar um alias para tipos

```
type Produto = (Integer, String, Double)
type Cliente = (Integer, String, Double)

pago  (_, _, p) = p
preco          = pago

troco :: Produto -> Cliente -> Double
troco produto cliente = pago cliente - preco produto
```

ADTs: Uma forma de criar novos tipos de dados, onde podemos criar novos valores para tipos ou absorver tipos de dados primitivos, foi abordado 4 tipos: Soma, Produto, Recursivo e Exponencial.

Tipo soma, exemplo:

```
data Bool = True | False
```

- data: declara que é um novo tipo
- Bool: nome do tipo
- True | False: poder assumir ou True ou False

Tipo produto

```
data Ponto = MkPonto Double Double
```

- data: declara que é um novo tipo
- Ponto: nome do tipo
- MkPonto: construtor (ou envelope) - declaração implícita de uma função usada para criar um valor do tipo Ponto
- Double Double: tipos que ele encapsula

```
dist :: Ponto -> Ponto -> Double
dist (MkPonto x y) (MkPonto x' y') = sqrt $ (x-x')^2 + (y-y')^2

dist (MkPonto 1 2) (MkPonto 1 1)
```

Uma outra forma é usar os tipos soma e produto juntos

```
data Forma = Circulo Ponto Double
           | Retangulo Ponto Double Double

-- um quadrado é um retângulo com os dois lados iguais
quadrado :: Ponto -> Double -> Forma
quadrado p n = Retangulo p n n
```

NewType:

Resumo Paradigmas de programação - Professor Emílio
Aluno - Diogo Miyake - RA 21048813

Permite criar um novo tipo com apenas um construtor:

```
newtype Nat = N Int
```

- A diferença entre newtype e type é que o primeiro define um novo tipo enquanto o segundo é um sinônimo.

- A diferença entre newtype e data é que o primeiro define um novo tipo até ser compilado, depois ele é substituído como um sinônimo. Isso ajuda a garantir a checagem de tipo em tempo de compilação.

Tipos recursivos:

Um exemplo é uma árvore binária:

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

- Ou seja, ou é um nó folha contendo um valor do tipo a, ou é um nó contendo uma árvore à esquerda, um valor do tipo a no meio e uma árvore à direita.

Tipo Exponencial:

Tipo que define uma relação de um tipo a para um tipo b,

Algebra de tipos

Elementos algébricos que proporcionam possibilidade de operações algébricas

entre tipos, como o Zero = Void e Um = () - unit.

Either: União disjunta do conjunto, usado como tipo soma.

Pair: Usado como construção genérica para um tipo produto.

Zipper:

Permite otimizar algumas operações que poderiam ser custosas usando álgebras de tipos

Como o nome diz, permite se mover sobre os elementos de uma lista como um zipper, e isso pode ser estendido por outros tipos de algoritmos como a árvore binária mostrada na aula.

Classe de tipos:

Definem tipos de grupos de tipos que contêm funções especificadas..
Exemplo:

Resumo Paradigmas de programação - Professor Emílio
Aluno - Diogo Miyake - RA 21048813

Para criar uma nova classe de tipos utilizamos a palavra reservada class:

```
class Eq a where

  (==), (/=) :: a -> a -> Bool

  x /= y = not (x == y)
```

Essa declaração diz: *para um tipo a pertencer a classe Eq deve ter uma implementação das funções (==) e (/=)*.

Instâncias de classe:

```
instance Eq Bool where
  False == False = True
  True  == True  = True
  _     == _     = False
```

Apenas tipos definidos por data e newtype podem ser instâncias de alguma classe.

Outros exemplos são: Read, Show, Ord, Bool, Eq. Num, Integral, Fractional, Floating, Enum…

Resumo Paradigmas de programação - Professor Emílio
Aluno - Diogo Miyake - RA 21048813

Semana 5 - Monoid e Foldable & Functors

Semigroup

Um tipo 'a' é um Semigroup se fornecer uma função associativa (<>) que permite combinar quaisquer dois valores do tipo 'a' em um, isto é:
(x <> y) <> z = x <> (y <> z)

(<>) :: a -> a -> a [mínima]
sconcat :: NonEmpty a -> a
stimes :: Integral b => b -> a -> a

Monoid

Um monóide é uma operação associativa binária com uma identidade. Esta definição diz muito - se você está acostumado a separar as definições matemáticas. Um monóide é uma função que recebe dois argumentos e segue duas leis: associatividade e identidade. Associatividade significa que os argumentos podem ser reagrupados (ou reenquadrados ou reassociados) em ordens diferentes e dar o mesmo resultado, como adicional. Identidade significa que existe algum valor tal que, quando o passamos como entrada para nossa função, a operação se torna discutível e o outro valor é retornado, como quando adicionamos zero ou multiplicamos por um.

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Foldable
A classe do tipo Foldable fornece uma generalização da dobradura de lista (foldr e amigos) e operações derivadas dela para estruturas de dados arbitrárias. Além de ser extremamente útil, o Foldable é um ótimo exemplo de como os monoides podem ajudar a formular boas abstrações.

```
class Foldable t where
  fold    :: Monoid a => t a -> a
  foldMap :: Monoid b => (a -> b) -> t a -> b
  foldr   :: (a -> b -> b) -> b -> t a -> b
  foldl   :: (a -> b -> a) -> a -> t b -> a
```

Exemplo pratico do tipo soma:

```
newtype Sum a = Sum a
  deriving (Eq, Ord, Show, Read)

newtype Product a = Product a
```

Resumo Paradigmas de programação - Professor Emílio
Aluno - Diogo Miyake - RA 21048813

```haskell
  deriving (Eq, Ord, Show, Read)

getSum :: Sum a -> a
getSum (Sum x) = x

getProd :: Product a -> a
getProd (Product x) = x
```

Tipo de Dado Algébrico Fold
Permite a simplificação da aplicação de várias funções a estruturas dobráveis. O tipo Fold é paramétrico recebendo o tipo de dado que a estrutura foldable recebida carrega, e também o tipo resultante da aplicação do fold.

Functors

Functor em Haskell é um tipo de representação funcional de diferentes tipos que podem ser mapeados. É um conceito de alto nível de implementação de polimorfismo. De acordo com os desenvolvedores do Haskell, todos os Tipos, como Lista, Mapa, Árvore, etc., são a instância do Haskell Functor.
A classe functor pode ser definida como:

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Semana 6 - Applicative Functor e Traversable

Aplicative
Um functor com uma aplicação, fornecendo operações para transformar uma função pura em um determinado contexto computacional e uma sequencia de computações combinando os seus resultados

Permite aplicar funções com mais argumentos Fmap 2, fmap3, 4 … assim consegue -se mais generalização. Algo parecido com uma calculadora polonesa : pure f (+) <*> Just 3 <*> Just 4 <*> Just 5 que ele faria um : (3+4+5) retornando um 12.

pode ser definida como:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

para o Maybe:

```
instance Applicative Maybe where
  pure           = Just
  Nothing  <*> _    = Nothing
  (Just g) <*> mx  = fmap g mx
```

Traversable
Permite que tipos possam ser mapeados.Essa classe é útil quando, por exemplo, temos uma função que mapeia um tipo a para Maybe b e temos uma lista de a. A classe de tipo Traversable se relaciona ao mapa da mesma maneira que Foldable se relaciona à dobra.

Um exemplo abaixo:

```
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f =>
                        (a -> f b) -> t a -> f (t b)
```

Outro exemplo:

```
instance Traversable [] where
  traverse g []   = pure []
  traverse g (x:xs) = pure (:) <*> g x <*> traverse g xs
```

Folds
Folding é um conceito que se estende em utilidade e importância além das listas, mas as listas são muitas vezes como eles são introduzidos. Folds como um conceito geral são chamados catamorfismos. Catamorfismos são um meio de desconstruir dados. Se a espinha dorsal de uma lista é a estrutura de uma lista, então um fold é o que pode reduzir essa estrutura.

Um exemplo de fold junto com applicative:

```
instance Applicative (Fold i) where
```

```
 pure o = Fold (\_ -> ()) (\_ -> o)

 Fold toMonoidF summarizeF <*> Fold toMonoidX summarizeX = Fold toMonoid summarize
   where
         toMonoid i = (toMonoidF i, toMonoidX i)

         summarize (mF, mX) = summarizeF mF (summarizeX mX)
```

Semana 7 - Monads

Permite uma generalização de sequenciamento de computação, quando precisamos de funções que geram efeitos colaterais;

Sequência de computações com efeito em que uma computação depende da anterior; - Pode ser visto como um Monoid das categorias dos Functors;

Assim como functors tem o tipo de classe Functor e applicative functors tem o tipo de classe Applicative, monads tem seu próprio tipo de classe: Monad! Wow, quem teria adivinhado? Assim é como esse tipo de classe se parece:

```
class Monad m where

    return :: a -> m a

      (>>=) :: m a -> (a -> m b) -> m b

      (>>) :: m a -> m b -> m b

    x >> y = x >>= \_ -> y

    fail :: String -> m a

    fail msg = error msg
```

Em que o return encapsula qualquer 'a' em um monad, que antes precisa ser um applicative.

Semelhante ao pure do Applicative.
Exeemplo de functor
instance Functor (Either a) where

Resumo Paradigmas de programação - Professor Emílio
Aluno - Diogo Miyake - RA 21048813

```
 fmap f (Left x) = Left x
 fmap f (Right x) = Right (f x)
```

Exemplo do maybe
instance Monad Maybe where

```
   return x = Just x

   Nothing >>= f = Nothing

   Just x >>= f  = f x

   fail _ = Nothing
```

Monads e seus efeitos:

1. Funções impuras necessárias:
2. Parcialidade (Valor bottom ⊥)
3. Não-determinismo (diferentes saídas dependendo de condições internas e externas)
4. Efeitos colaterais: read only, write only, read/write
5. Exceções (Either)
6. Continuações
7. Entrada e saida interativa

Funçoes de alta ordem para monads

As funções de alta ordem possuem versões para Monads na biblioteca específica Control.Monad
Temos funções como map (mapM) e filter (filterM), onde a principal diferença é:
filter seleciona elementos de uma lista
filterM seleciona possíveis eventos de uma sequência de computações

Syntatic Sugar:
Essa mesma expressão pode ser escrita com a notação chamada **do-notation** :

```
do x1 <- m1
   x2 <- m2
   ...
   xn <- mn
   f x1 x2 ... xn
```

Esse tipo de operação forma uma nova classe de tipos denominada **Monads** :

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

   return = pure
```

Resumo Paradigmas de programação - Professor Emílio
Aluno - Diogo Miyake - RA 21048813