Mônadas: Da Teoria de Categorias à Programação Funcional

Universidade Federal de Viçosa
Centro de Ciências Exatas e Tecnológicas
Departamento de Matemática

Manoela Werneck Auad (Bolsista IM) *manoela.auad@ufv.br - DPI*

Rogério Carvalho Picanço (Orientador) *rogerio@ufv.br*

Instituto do Milênio
Avanço Global e Integrado da Matemática Brasileira

Palavras chave: mônadas, programação funcional, haskell
Área temática: Ciência da Computação

1. Introdução

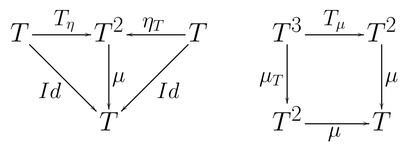
Embora as mônadas tenham surgido na teoria de categorias, um ambiente bastante abstrato, uma consulta rápida na internet da palavra "Monads" levará a um grande número de trabalhos e referências à computação. Mônadas são um ótimo exemplo da evolução de um conceito criado numa das áreas mais puras da Matemática que se tornaram bastante úteis na resolução de problemas práticos, como na programação funcional.

2. Mônadas em Categorias

Dada uma categoria $\mathcal{C}$, uma mônada sobre $\mathcal{C}$ consiste em:

- um *endofuntor* $T : \mathcal{C} \longrightarrow \mathcal{C}$, ou seja, uma aplicação entre objetos de $\mathcal{C}$, que preserva quaisquer morfismos entre eles,
- uma transformação natural $\eta : Id \longrightarrow T$ chamada *unidade*,
- uma transformação natural $\mu : T^2 \longrightarrow T$ chamada *produto*.

satisfazendo os seguintes diagramas comutativos:



A unidade da mônada pode ser vista como o elemento neutro de um monoide, e o produto pode ser visto como associativo. Essas associações podem ficar mais claras observando os diagramas acima.

3. Programação funcional

O paradigma funcional descreve uma computação como uma expressão a ser avaliada. Diferente das linguagens imperativas, que se baseiam na mudança de estado do programa, uma linguagem funcional trabalha com estruturas que não são mutáveis, e o resultado do programa é resultante da aplicação de uma função.

A categoria Hask

Para ver como podemos estender as estruturas matemáticas de uma categoria dentro do ambiente da programação funcional, é importante definir a categoria *Hask*. Os objetos de *Hask* são os tipos primários da linguagem (Int, Double, Bool...). Os morfismos são funções entre os tipos. Um morfismo $f : a \longrightarrow b$ é uma função `f :: a -> b`, que recebe um valor do tipo `a` e retona um valor do tipo `b`.
A composição é avaliação de uma função dentro de outra, representada pelo operador `(.)` definido como:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

E a identidade dos morfismos é a função

```
id :: a -> a
id x = x
```

4. Tipos paramétricos

As linguagens funcionais têm construtores de tipo, ou seja, construtores que recebem um tipo como parâmetro, e assim criam um novo tipo, chamado *tipo paramétrico*. Por exemplo, o tipo **maybe** envolve um valor opcional. Um valor do tipo `Maybe a` pode ser um valor do tipo `a`, representando por `Just a`, ou um "valor vazio", representado por `Nothing`. Esse tipo constitui uma computação que pode dar errado, que pode resultar em nada. Uma função de divisão inteira, por exemplo, deve retornar um `Maybe Int`, que pode ser o valor inteiro resultante da divisão, ou nada, se tentarmos dividir por 0.
As funções que definem um programa funcional operam como na matemática: cada elemento de entrada está relacionado a um único elemento de saída.
No entanto, ao contrário das funções puras, os tipos paramétricos definem computações que podem gerar efeitos no estado do programa. O tipo `Maybe` pode entregar um valor, ou pode não entregar nada. Logo, se temos uma função `gtz :: Int -> Bool`, que verifica se um número inteiro é maior que zero, não podemos aplicá-la ao tipo `Maybe Int`, porque aqui o inteiro está encapsulado em um contexto, o de um valor opcional.
O que precisamos fazer aqui é mapear a função `gtz` em uma função `f` que saiba lidar com o contexto em volta. Para isso, temos os `Functors`.

5. Functors

Se temos uma função `g :: a -> b` e um tipo paramétrico `f`, *Functors* nos permitem aplicar `g` em `f a` para obter `f b`. Eles são definidos da seguinte forma:

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

`fmap` é uma função que, recebe uma função pura e um valor em um contexto, aplica a função neste valor, e retorna o resultado envolto no contexto. Todo tipo paramétrico que defina como `fmap` deve ser aplicada, é um Functor. Assim, podemos fazer:

```
> fmap (*2) (Just 5)
Just 10
> fmap (*2) Nothing
Nothing
```

E isto funciona porque o tipo `Maybe` é um Functor, definido como:

```
instance Functor Maybe where
    fmap func (Just val) = Just (func val)
    fmap func Nothing = Nothing
```

Se temos um valor `Just`, `fmap` aplica a função neste valor. Mas se começamos com `Nothing`, a função não é aplicada e terminamos com `Nothing`. Dessa forma, podemos executar ações que surtem efeito de forma sequencial. Porém, essas ações devem ser independentes entre si. Se tentarmos encadear computações cujos resultados geram outras computações, nossa sequência será interrompida.

6. Monads

As *Monads* resolvem muitos problemas que vêm com a programação funcional.
Elas são definidas pelo operador `>>=`, denominado **bind**, e uma função `return`, que recebe um valor e coloca ele em uma mônada, envolvendo-o em um contexto mínimo.

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

O operador `>>=` nos permite executar ações em sequência, envolvidas em contextos separados, de forma que uma ação afeta a próxima. Se conseguimos definir `>>=` para um tipo paramétrico, ele é uma Monad.

Maybe Monad

Suponha que `half` seja função que só opera em números pares.

```
half::Int -> Maybe Int
half x = if even x
         then Just (x `div` 2)
         else Nothing
```

Funciona assim:

```
> half 2
Just 1
> half 3
Nothing
> half (half 20)
error
```

Não conseguimos encadear essas operações, pois `half` não sabe o que fazer com valores em um contexto, como `Just 10`. Mas, se usarmos Monads:

```
> Just 20 >>= half >>= half
Just 5
> Just 20 >>= half >>= half >>= half
Nothing
```

Isso significa que se temos um valor `Maybe`, ou qualquer outro tipo paramétrico, sendo sequenciado (usando `bind`) com uma função `f`, o que acontece por traz é uma análise do valor `Maybe` para aplicarmos `f` corretamente. Assim, nossas funções podem ficar bem mais simples, ainda que saibam lidar com os efeitos de mudança de estado.
`Maybe` é uma Monad definida por:

```
instance Monad Maybe where
 return = Just
 (Just x) >>= f = f x
 Nothing >>= f = Nothing
```

Se houver valor, ele é extraído do contexto e aplica-se `f`. Mas se houver uma falha (retorna `Nothing`), o programa não chama a função, e as próximas computações não ocorrerão de forma errada.

7. Conclusão

Este trabalho mostra a relação entre conceitos que, nascidos em ambientes abstratos, como a teoria de categorias que surgiu no âmbito da topologia algébrica, assumem vida própria e podem se aplicar em ambientes mais pragmáticos, como a programação funcional. Existem outras diversas aplicações da teoria de categorias na computação, e muitas aplicações de mônadas em várias áreas da matemática pura e aplicada.

8. Referências

Dentre outras, as principais referências deste trabalho são:

MacLane, Saunders. *Categories for the Working Mathematician*, 1st ed. (1970).

A. Kuhnle Master thesis. *Modeling Uncertain Data using Monads and an Application to the Sequence Alignment Problem.* Karlsruhe Institute of Technology (2013).

S. Grahn, *Monads in Haskell and Category Theory.* Examensarbete 15hp, september 2019, Uppsala Universitet.