Abstracciones

4/11/2024

¿Por qué abstraer?

Al usar abstracciones:

- simplificamos la escritura de los programas,
- los programas son más legibles,
- facilitamos el reuso (pensando a la abstracción como una generalización de un concepto).

Buenas abstracciones

Consideramos buenas abstracciones aquellas que:

- Tienen una definición **precisa**.
- Tienen un **fundamento teórico** con resultados y/o propiedades sobre la abstracción.
- Son **generales**, son útiles en diversas situaciones.

Ejemplos de abstracciones

Veremos 3 abstracciones en Haskell que provienen de la **teoría de categorías**:

1. Functores: generalizan la noción de mapeo de una función.
2. Functores aplicativos: generalizan la aplicación de una función.
3. Mónadas: generalizan la noción de programación con efectos.

Generalizando map

La idea de aplicar una función sobre los elementos de una estructura de datos no es específica de listas:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

data Tree a = L a | N (Tree a) (Tree a)

mapT :: (a ->b) -> Tree a -> Tree b
mapT f (L x)    = L (f x)
mapT f (N l r) = N (mapT f l) (mapT f r)
```

Generalizando map

- No siempre podremos definir una función que se comporte como **map**. Por ejemplo:

```haskell
data Func a = Func (a -> a)

mapFunc :: (a -> b) -> Func a -> Func b
mapFunc g (Func h) = Func id
```

- La definición anterior tipa pero no mapea la función que recibe sobre los elementos de la estructura.

Functores en Haskell

En Haskell se define la siguiente clase para definir la función que realiza el mapeo de una función (fmap):

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

Llamamos functor a los constructores de tipos que poseen una función fmap y además satisfacen las ecuaciones:

1. **(Functor.1)** `fmap id = id`
2. **(Functor.2)** `fmap f . fmap g = fmap (f.g)`

Ejemplos de Functores

```haskell
instance Functor [] where
   fmap = map

data Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)

data Tree a = L a | N (Tree a) (Tree a)

instance Functor Tree where
  fmap f (L x)   = L (f x)
  fmap f (N l r) = N (fmap f l) (fmap f r)
```

Ejemplos de Functores

Probamos **Functor.1** : $\forall$ `xs :: [a] . fmap id xs = xs`, por inducción sobre `xs`

$$
\begin{array}{ll}
 & \texttt{fmap id []} \\
= & \{\text{fmap.1}\} \\
 & \texttt{[]}
\end{array}
$$

$$
\begin{array}{ll}
 & \texttt{fmap id (x:xs)} \\
= & \{\text{fmap.2}\} \\
 & \texttt{id x : fmap id xs} \\
= & \{\text{id.1, HI}\} \\
 & \texttt{x : xs}
\end{array}
$$

Ejercicio: Probar **Functor.2**

- Supongamos que queremos generalizar `fmap` para que mapee funciones de cualquier número de argumento. Por ejemplo,

```
fmap2 ::(a -> b -> c)-> f a-> f b-> f c

fmap3 ::(a -> b -> c -> d)-> f a ->f b ↩
   ->f c -> f d
...
```

- Usando la currificación de funciones, las funciones `fmapn` podrán definirse usando dos funciones bases con tipos:

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

Por ejemplo:

```
fmap2 g x y = pure g <*> x <*> y

fmap3 g x y z = pure g <*> x <*> y <*> z
```

A este estilo de definición de función se lo llama aplicativo.

- Los functores aplicativos son functores que permiten aplicar funciones dentro del functor.

- En Haskell se implementan mediante una clase de tipos.

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

- Veremos en el siguiente ejemplo, como las operaciones `pure` y `<*>` son útiles para combinar funciones que pueden fallar.

```
instance Applicative Maybe where
   pure = Just
   (Just f ) <*> (Just x) =  Just (f x)
   _         <*>  _       =  Nothing
```

Por ejemplo,

```
> pure (+2) <*> Just 4
Just 6
> pure (+3) <*> Just 7
Just 10
> pure (+) <*> Just 3 <*> Just 5
Just 8
> pure (+) <*> Nothing <*> Just 2
Nothing
```

```haskell
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

Por ejemplo,

```
ghci> [(+1), (^2)] <*> [1,2,3]
[2,3,4,1,4,9]
ghci> [(+), (^)] <*> [3,5] <*> [2,3]
[5,6,7,8,9,27,25,125]
```

Functores Aplicativos

Leyes

- El concepto de functor aplicativo también viene de la teoría de categorías.

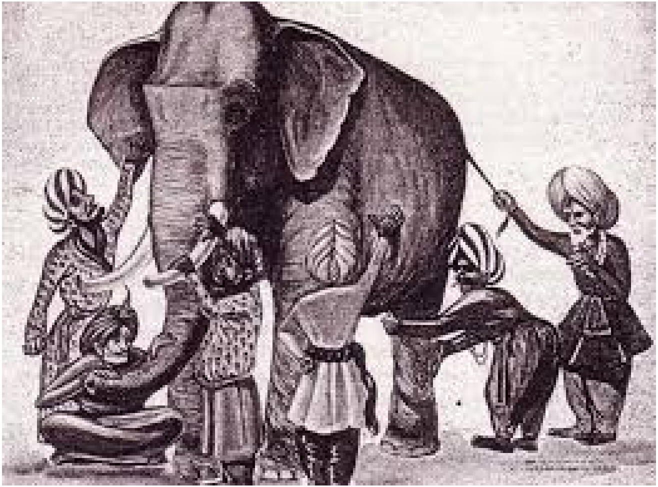
- En el caso de los functores aplicativos el "buen comportamiento" queda especificado por las siguientes leyes:

```
pure id <*> x = x
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure (\g -> g y) <*> u
```

- McBride y Paterson llevaron este concepto a la programación funcional y mostraron que los functores aplicativos generalizan a las mónadas. Toda mónada es un functor aplicativo.

  Mónadas $\Rightarrow$ Functores aplicativos $\Rightarrow$ Functores

Mónadas



¡No hay una única manera de enseñar mónadas! Visitar
https://wiki.haskell.org/Monad_tutorials_timeline

Mónadas

- El concepto de Mónada viene de la Teoría de Categorías.
- Fueron utilizadas por Moggi para dar una semántica operacional a los lenguajes de programación [1991].
- P. Wadler y S. Peyton Jones continuaron su trabajo aplicando las mónadas para dar estructura a los programas funcionales en Haskell. Obteniendo un estilo de programación para escribir programas que no sean funciones matemáticas puras.

Función pura: El resultado depende únicamente de los argumentos de la función y no genera efectos secundarios.

Función impura: El resultado puede depender de entradas externas y puede modificar el estado externo.
Ejemplo: Función que toma como entrada una cadena desde el teclado.

Sea `f` un functor aplicativo, si tenemos un valor de tipo `f a` y queremos aplicarle una función de tipo `a -> f b` al valor contenido en este contexto necesitamos una función con el siguiente tipo:

```
f a -> (a -> f b) -> f b
```

Mónadas

- En Haskell las mónadas se implementan mediante la clase:

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

- El programador debe chequear que se satisfacen las siguientes leyes:
  - (monad.1):

    ```
    return a >>= k =  k a
    ```

  - (monad.2):

    ```
    m >>= return  =  m
    ```

  - (monad.3):

    ```
    m >>=(\x -> k x >>= h) = (m >>= k)>>= h
    ```

Una función no determinista cuyo resultado es de tipo a no es una función pura, pero puede transformase en una función pura cambiando el tipo de retorno a por [a].

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
```

Se puede probar fácilmente que se cumplen las 3 leyes monádicas. Por ejemplo,

```
ghci> [1,2,3] >>= \x -> [x,-x]
[1,-1,2,-2,3,-3]
```

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

Se puede probar fácilmente que se cumplen las 3 leyes monádicas. Por ejemplo,

```haskell
ghci> Just 5 >>= \x -> return (x*10)
Just 50
ghci> Nothing >>= \x -> return (x*10)
Nothing
```

Ejercicios

1. Probar que todo mónada es un functor, es decir, proveer una instancia

   ```
   instance Monad m => Functor m where
      fmap ...
   ```

   y probar que se cumplen las leyes de functores.

2. Probar que toda mónada es un functor aplicativo.

Resumen

- Vimos 3 abstracciones más que vienen de teoría de categorías: functores, functores aplicativos y mónadas.
- En Haskell estas abstracciones se definen mediante clases y el programador debe chequear que se satisfacen las leyes de buen comportamiento.
- Vimos que los functores aplicativos y las mónadas pueden usarse para combinar computaciones no deterministas y computaciones que pueden fallar.
- Con los functores aplicativos se utiliza un estilo de programación aplicativo, mientras que con las mónadas (con la notación do) un estilo secuencial (o imperativo).

Bibliografía

G. Hutton.
Programming in haskell (2nd ed).
2007.