Category Theory for Programmers

Homework (Session 4)

Bruno Vandekerkhove

Contents

Alternative definition of a natural transformation

Saunders MacLane defines natural transformations in terms of what's basically a commutative diagram, the naturality square (here with two functors F and G, $C \to D$) :

$$\begin{tikzcd}
F(c_1) \arrow[r, "F(f)"] \arrow[d, "\alpha_x"'] & F(c_2) \arrow[d, "\alpha_y"] \\
G(c_1) \arrow[r, "G(f)"'] & G(c_2)
\end{tikzcd}$$

The rest of this whole paragraph is loosely equivalent with *nLab's* definition of a natural transformation in terms of the cartesian closed monoidal structure [1]. A cartesian monoidal category C is closed when there's an *internal hom* functor $[X, -] : C \to C$ such that :

$$Hom(a \times b, c) \cong Hom(a, [b, c] = b^c)$$

Where the rightward map is called currying. The exponential notation is used because the category is cartesian.

Since `Cat` is a cartesian monoidal category the question is if there is an *internal hom* functor $[C, -] :$ `Cat` $\to$ `Cat` which would make it closed such that for $A, C, D \in$ `Cat` :

$$Funct(A \times C, D) \cong Funct(A, [C, D])$$

This is true when $[C, D]$ represents the category of functors from C to D. The morphisms in that category are natural transformations, and to see what they are we consider a functor $I \to E$ with E any category and I the *interval* category (a category with just two objects and one morphism between them, denoted as $\{a \to b\}$). This functor corresponds to a choice of morphism in E. So we can find out what a morphism is in $[C, D]$ by taking a close look at the functors $I \to [C, D]$. The following holds :

$$Funct(I, [C, D]) \cong Funct(I \times C, D)$$

This means we can consider functors $I \times C \to D$ instead. The morphisms of the category $I \times C$ are pairs of morphisms in I and C subject to composition laws such that we end up with a commuting diagram :

$$\begin{tikzcd}
(c_1, a) \arrow[r, "{(f, id)}"] \arrow[d, "{(id, a \to b)}"'] & (c_2, a) \arrow[d, "{(id, a \to b)}"] \\
(c_1, b) \arrow[r, "{(f, id)}"'] & (c_2, b)
\end{tikzcd}$$

We can therefor conclude that a natural transformation between functors $C \to D$ is given by the image of this square in D. For a natural transformation $\alpha : F \to G$ one can trace back the *hom*-isomorphism to end up with a commuting diagram which corresponds to the naturality square depicted at the start of the paragraph (also depicted in Bartosz's book).

The bifunctor `Pair a b`

Here's an implementation of the `Pair` bifunctor :

```haskell
import Data.Bifunctor

data Pair a b = Pair a b

instance Bifunctor Pair where
    bimap g h (Pair x y) = Pair (g x) (h y)
    first g (Pair x y) = Pair (g x) y
    second h (Pair x y) = Pair x (h y)
```

`Pair a b` can be proven to be a bifunctor with equational reasoning (it's enough to show that it is functorial in each argument separately) :

```haskell
bimap id id (Pair x y) = Pair (id x) (id y)
        = Pair x y
        = id (Pair x y)
bimap (f . g) id (Pair x y) = Pair (f . g x) (id y)
        = Pair (f . g x) y
        = bimap g . (Pair (f x) y)
        = bimap g . bimap f (Pair x y)
-- bimap id (f . g) (Pair x y) boils down to the same thing
```

This is not a surprise given that `Pair` is isomorphic with the `(,)` bifunctor. We now show that the above implementation is equivalent to using the default ones :

```haskell
bimap g h (Pair x y) = first g . second h = first g . (Pair x (h y)) = Pair (g x) (h y)
first g = bimap g id = Pair (g x) (id y) = Pair (g x) y
second h = bimap id h = Pair (id x) (h y) = Pair x (h y)
```

Isomorphism of `Maybe` and `Maybe'`

It can be shown that `Maybe'` and `Maybe` are isomorphic by constructing two mappings and using equality reasoning to show that they're each other's inverse :

```
import Data.Functor.Identity
import Data.Functor.Const

type Maybe' a = Either (Const () a) (Identity a)

toMaybe' :: Maybe a -> Maybe' a
toMaybe' Nothing = Left (Const ())
toMaybe' (Just a) = Right (Identity a)

toMaybe :: Maybe' a -> Maybe a
toMaybe (Left (Const ())) = Nothing
toMaybe (Right (Identity a)) = Just a

toMaybe' . toMaybe (Left (Const ())) = toMaybe' Nothing = Left (Const ())
toMaybe' . toMaybe (Right (Identity a)) = toMaybe' (Just a) = Right (Identity a)

toMaybe . toMaybe' Nothing = toMaybe (Left (Const ())) = Nothing
toMaybe . toMaybe' (Just a) = toMaybe (Right (Identity a)) = Just a
```

The `PreList` bifunctor

The `PreList` bifunctor can be implemented as follows :

```
data PreList a b = Nil | Cons a b

instance BiFunctor (PreList a b) where
        bimap g h Nil = Nil
        bimap g h (Cons a b) = Cons (g a) (h b)
```

Aside from proving that it is functorial in both arguments (which is analogous to what we did for `Pair`) it can be noted that it is equivalent to `Either (Const () a) (a,b)` which is a bifunctor because `Either` and `(,)` are bifunctors and `Const ()` is a functor. The reasoning is similar to what happens with `Maybe = Either (Const () a) (Identity a)` except that in that case you really end up with a functor because the type parameter is the same for both parts of the sum.

`K2`, `Fst` and `Snd`

Implementation of `bimap` follows :

```
instance BiFunctor (K2 c a b) where
        bimap _ _ = id
instance BiFunctor (Fst a b) where
        bimap g _ (Fst a) = Fst (g a)
instance BiFunctor (Snd a b) where
        bimap _ h (Snd b) = Fst (h b)
```

In the referenced study `K2` is described as the bifunctorial 'cousin' of `K1` which is a constant. They're part of a repertoire of components called *'polynomial (bi)functors'* which are used by the author to deal with data types in a generic way.

Natural transformation of `Maybe` to `List`

Here's a natural transformation from `Maybe` to `List` and a proof that it satisfies the naturality condition :

```haskell
alpha :: Maybe a -> [a]
alpha Nothing = []
alpha (Just x) = [x]

fmap f (alpha Nothing) = fmap f [] = []
alpha (fmap f Nothing) = alpha Nothing = []

fmap f (alpha (Just x)) = fmap f [x] = [f x]
alpha (fmap f (Just x)) = alpha (Just (f x)) = [f x]
```

Many more are possible, which is illustrated in the next challenge. It can be noted that Haskell's parametric polymorphism made the proof unnecessary, which was hinted at in the previous lesson where the 'magic trick' showed that polymorphic functions do nothing to the elements themselves so the order of application isn't important.

Natural transformation from `Reader` to `List` or `Maybe`

There's an infinite amount of natural transformations from `Reader ()` to `List` because you can transform to the empty list or to a series of applications of g to () :

```haskell
-- :: Reader () a -> [a]
alpha0 (Reader g) = []
alpha1 (Reader g) = [g ()]
alpha2 (Reader g) = [g (), g ()]
...
```

This doesn't hold in the case of `Reader Bool` and `Maybe` where you can only transform in three ways (returning `Nothing`, `Just (g True)` and `Just (g False)` respectively).

References

[1] nLab. natural transformation in nlab, April 2019. ncatlab.org [Last revised on April 3, 2019 at 03:15:34.].