

Now that we have the definition of a natural transformation fleshed out, let's take a look at least a couple of examples before we go off talking about how we can glue them together in various ways.

I've been meaning to get some stuff into this blog about functional programming, so let's borrow our first real example from Haskell. I think most of our readers probably know already, but for anyone who isn't aware, Haskell is a nice functional programming language with lots of cool features, and I highly recommend checking it out if you're a mathematician.

So, in Haskell, the various datatypes, together with the Haskell-definable functions between them form a category. Composition of functions  $f :: B \rightarrow C$  and  $g :: A \rightarrow B$  is given by the composition function defined in the Prelude:

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

And of course, the identity maps are also in the Prelude:

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

So we'll refer to this category of Haskell types and functions as **Hask**. In addition to being a category, we have a convenient way within Haskell to talk about functors  $\mathbf{Hask} \rightarrow \mathbf{Hask}$ .

Our functor needs to have a part which is a function on types, and parametric types allow us to define some. For instance, for each type **a**, we have the type **[a]** of lists of values of type **a**, and we can define other datastructures which are parametrised on a type, such as binary trees with values of type **a** at each of the branches:

```
data Tree a where
  Leaf    :: Tree a
  Branch  :: a -> Tree a -> Tree a -> Tree a
```

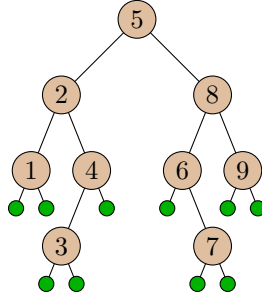
This defines **Leaf** as a value of type **Tree a** for any type **a**, and **Branch** as a function which takes a value of type **a** and two trees, each of type **Tree a**, and formally constructs a value of type **Tree a**.

I am using GADT syntax here, which is a syntax extension supported by GHC. You can turn it on with the flag `-XGADTs`, or by writing

```
{-# LANGUAGE GADTs #-}
```

at the top of your source file.

For example, we might represent the tree of integers:



using the value:

```
t :: Tree Integer
t = Branch 5 (Branch 2 (Branch 1 Leaf
                        Leaf)
              (Branch 4 (Branch 3 Leaf
                        Leaf))
              (Branch 8 (Branch 6 Leaf
                        (Branch 7 Leaf
                          Leaf))
                        (Branch 9 Leaf
                          Leaf)))
```

Of course, being a function on types is not enough. If the type constructor `[]` for lists or `Tree` for trees is to be a functor, we need a way to send functions  $(a \rightarrow b)$  to functions  $[a] \rightarrow [b]$  or  $\text{Tree } a \rightarrow \text{Tree } b$ .

In order to provide a uniform interface to such mappings, the Haskell Prelude also defines the following class of type constructors:

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

Of course, these are really just endofunctors, but for purposes of programming, those are the ones we're usually most concerned with.

To define `[]` and `Tree` as functors then, we write instances:

```
instance Functor [] where
  fmap f [] = []
  fmap f (x:xs) = f x : fmap f xs
```

For those unfamiliar with Haskell, the first equation says that applying `fmap` to the empty list gives the empty list. The second says that applying it to a nonempty list whose first element is `x` and where the remainder of the list is called `xs` (think plural), is the list whose first element is `f x` and where the rest of the list is `fmap f xs`. The end result is that it just applies the

function `f` to each of the values in the list, leaving the list structure otherwise unchanged.

Very similarly, we can define a functor instance for trees:

```
instance Functor Tree where
  fmap f Leaf = Leaf
  fmap f (Branch x left right) = Branch (f x) (fmap f left)
                                   (fmap f right)
```

This again just applies the function `f` to each of the values on the branches of the tree, leaving the structure alone.

Now, these are both functors with common domain and codomain, so perhaps we can describe a natural transformation between them?

A natural transformation  $\eta : Tree \rightarrow []$  would give, for each type `a`, a function whose type would be `Tree a -> [a]`. So, we would expect that perhaps some polymorphic function will serve as a natural transformation.

However, this gets better: Since a polymorphic function of that type is disallowed by the type system to actually inspect values of type `a` as it does its work, it can only shuffle them around in some way which is independent of their values, and the only way it can get values of type `a` to fill the list is by plucking values from the tree it receives.

So, we would expect that if we applied a function to the values on the branches of the tree beforehand, and then the polymorphic function from trees to lists, the result would be the same as if we applied the polymorphic function first to get a list, and then applied the function to the values of the resulting list. The naturality square commutes!

To write the naturality equation using Haskell syntax, if

```
eta :: Tree a -> [a]
```

is our natural transformation, then

```
fmap f . eta = eta . fmap f
```

(This equation cannot be used directly as part of a program, but will be a law that our code satisfies.)

To rephrase all that, *every* polymorphic function of that type is a natural transformation, and Haskell's type system will actually ensure that we don't write code which violates this.

So, what's a good example? How about something like an inorder traversal?

```
inorder :: Tree a -> [a]
inorder Leaf = []
inorder (Branch x l r) = inorder l ++ [x] ++ inorder r
```

Here, the operator `(++)` is concatenation of lists.

Aside: This is not the best algorithm to choose from a complexity standpoint because concatenation of lists takes time on the order of the length of the first list, and we can actually get a much faster implementation by factoring this map through the type of functions

```
[a] -> [a]
```

by using functions which add appropriate elements to the beginning of a list in lieu of actual lists. Concatenation of lists becomes composition of functions then, which is constant time. We can then apply our function to the empty list once it is constructed.

```
inorder :: Tree a -> [a]
inorder t = inorder' t []
  where
    inorder' :: Tree a -> ([a] -> [a])
    inorder' Leaf = id
    inorder' (Branch x l r) = inorder' l . (x:) . inorder' r
    -- NB. (x:) is the function which adds x
    --                to the beginning of a list.
```

So to reiterate, from this very discrete programming standpoint, we have that one sort of natural transformations are these polymorphic functions which take one sort of data structure to another, leaving the elements of the structure untouched, and merely permuting, duplicating, or deleting them somehow in a way which is independent of their individual values.

One can imagine from this standpoint many sorts of natural transformations on sorts of combinatorial structures, expressed as functors  $\mathbf{Set} \rightarrow \mathbf{Set}$  (or variations thereof: the category of finite sets whose morphisms are only the bijections is a good choice for those who are interested primarily in enumeration). I will leave the details of this approach to enumerative combinatorics for another time, but the keyword if you're interested is "combinatorial species".