Properties of natural transformations

With code examples in Scala

Sergei Winitzki

Academy by the Bay

2020-05-30

Refactoring code by permuting the order of operations

- Expected properties of refactored code:

First extract user information, then convert stream to list; or first convert to list, then extract user information:

`db.getRows.toList.map(getUserInfo)` gives the same result as

`db.getRows.map(getUserInfo).toList`

First extract user information, then exclude invalid rows; or first exclude invalid rows, then extract user information:

`db.getRows.map(getUserInfo).filter(isValid)` gives the same result as

`db.getRows.filter(getUserInfo andThen isValid).map(getUserInfo)`

- These refactorings are guaranteed to be correct...
  - ... because `_.toList` is a natural transformation `Stream[A] => List[A]`
  - and `_.filter` is also a natural transformation in disguise
- Natural transformations satisfy the "naturality properties"

Refactored code: equations

Introduce short syntax to write those properties as equations:

`def toList[A]: Stream[A] => List[A]`	$\text{toList}^A : \text{Str}^A \to \text{List}^A$
`val f: A => B`	$f^{:A\to B}$
`_.map(f)` with type `List[A] => List[B]`	$f^{\uparrow\text{List}}$ or $f^{\uparrow}$
`_.toList.map(f)`	$\text{toList} ⨾ f^{\uparrow\text{List}}$
`f andThen g`	$f ⨾ g = g \circ f$
`_.map(f).map(g) == _.map(f andThen g)`	$f^{\uparrow\text{List}} ⨾ g^{\uparrow\text{List}} = (f ⨾ g)^{\uparrow\text{List}}$

The "short syntax" is equivalent to Scala code

Refactored code: equations

Rewrite the previous examples as equations and type diagrams:

`def toList[A]: Stream[A] => List[A]` written as $\text{toList}^A : \text{Str}^A \to \text{List}^A$

$$\begin{tikzcd} \text{Str}^A \arrow[r, "\text{toList}^A"] \arrow[d, "f^{\uparrow\text{Str}}"] & \text{List}^A \arrow[d, "f^{\uparrow\text{List}}"] \\ \text{Str}^B \arrow[r, "\text{toList}^B"] & \text{List}^B \end{tikzcd}$$

```
_.toList.map(f) == _.map(f).toList
```

$$\text{toList}^A \fatsemi f^{\uparrow\text{List}} = f^{\uparrow\text{Str}} \fatsemi \text{toList}^B$$

$$\text{toList} ; f^{\uparrow\text{List}} = f^{\uparrow\text{Str}} ; \text{toList}$$

`def filt[A]: (A => Boolean) => Stream[A] => Stream[A]`

$$\begin{tikzcd} \text{Str}^A \arrow[r, "\text{filt}^A(f \fatsemi p)"] \arrow[d, "f^{\uparrow\text{Str}}"] & \text{Str}^A \arrow[d, "f^{\uparrow\text{Str}}"] \\ \text{Str}^B \arrow[r, "\text{filt}^B(p)"] & \text{Str}^B \end{tikzcd}$$

$$\text{filt}^A : (A \to \mathbb{2}) \to \text{Str}^A \to \text{Str}^A$$

$$f^{\uparrow\text{Str}} \fatsemi \text{filt}^B(p) = \text{filt}^A(f \fatsemi p) \fatsemi f^{\uparrow\text{Str}}$$

$f^{\uparrow}$ $f^{\uparrow}$

$\mathbb{1}$ - Unit

$\mathbb{2}$ - Boolean

```
_.map(f).filter(p) == _.filter(x => p(f(x))).map(f)
```

- A transformation before `map` equals a transformation after `map`
- This is called a **naturality law**
- We expect it to hold if the code works the same way for all types
  - The naturality law is a mathematical expression of the programmer's intuition about code "working the same way for all types"

Naturality laws: equations

Naturality law for a function t is an equation involving an arbitrary function f that permutes the order of application of t and of a lifted f

$$\begin{tikzcd} \mathrm{List}^A \arrow[r, "\mathrm{headOpt}^A"] \arrow[d, "f^{\uparrow\mathrm{List}}"] & \mathrm{Opt}^A \arrow[d, "f^{\uparrow\mathrm{Opt}}"'] \\ \mathrm{List}^B \arrow[r, "\mathrm{headOpt}^B"] & \mathrm{Opt}^B \end{tikzcd}$$

```
_.map(f).headOption == _.headOption.map(f)
```

$$f^{\uparrow\mathrm{List}} \fatsemi \mathrm{headOpt} = \mathrm{headOpt} \fatsemi f^{\uparrow\mathrm{Opt}}$$

- Lifting f before t equals to lifting f after t
- Intuition: t rearranges data in a collection, not looking at values

Further examples:

- Reversing a list; $\mathrm{reverse}^A : \mathrm{List}^A \to \mathrm{List}^A$

```
_.map(f).reverse == _.reverse.map(f)
```

$$(f^{:A\to B})^{\uparrow\mathrm{List}} \fatsemi \mathrm{reverse}^B = \mathrm{reverse}^A \fatsemi (f^{:A\to B})^{\uparrow\mathrm{List}}$$

- The `pure` method, `pure[A]: A => L[A]`. Notation: $\mathrm{pu}_L : A \to L^A$

```
pure(x).map(f) == pure(f(x))
```

$$\mathrm{pu}^A \fatsemi (f^{:A\to B})^{\uparrow L} = f^{:A\to B} \fatsemi \mathrm{pu}^B$$

Natural transformations and their laws

def $t[A] : F[A] \Rightarrow G[A]$

A **natural transformation** is a function t with type signature $F^A \to G^A$ that satisfies the naturality law $f^{\uparrow F} ⨾ t = t ⨾ f^{\uparrow G}$. Notation $t : F \rightsquigarrow G$

Mnemonic rule: if $t : F \rightsquigarrow G$ then the lifting to F is on the left, the lifting to G is on the right

- Many standard methods have the form of a natural transformation
  - Examples: `headOption`, `lastOption`, `reverse`, `swap`, `map`, `flatMap`, `pure`
- If there are several type parameters, use one at a time:

def $\text{flm}[A,B] : (A \Rightarrow M[B]) \Rightarrow M[A] \Rightarrow M[B]$

  - For `flatMap`, denote $\text{flm} : \left(A \to M^B\right) \to M^A \to M^B$, fix A
    - ★ $\text{flm} : F^B \to G^B$ where $F^B \triangleq A \to M^B$ and $G^B \triangleq M^A \to M^B$
  - The naturality law $f^{\uparrow F} ⨾ \text{flm} = \text{flm} ⨾ f^{\uparrow G}$ then gives the equation

$$\text{flm}\left(p^{:A\to M^B} ⨾ f^{\uparrow M}\right) = \text{flm}\left(p^{:A\to M^B}\right) ⨾ f^{\uparrow M}$$

for any $f : B \to C$

if we write out the code for $f^{\uparrow F}$ and $f^{\uparrow G}$:

$$f^{\uparrow F} = p^{:A\to M^B} \to p ⨾ f^{\uparrow M} \quad , \qquad f^{\uparrow G} = q^{:M^A\to M^B} \to q ⨾ f^{\uparrow M}$$

$f^{\uparrow F} : F^B \to F^C$

More practical uses of natural transformations I

Recognize natural transformations in code and refactor

```
def ensureName(name: Option[String], id: Long): Option[(String, Long)] =
      name.map((_, id))
```

- Recognize that the code works the same way for all types
- Introduce type parameters `A` and `B` instead of `String` and `Long`
- The refactored code is a natural transformation:

```
def toOptionPair[A, B](x: Option[A], b: B): Option[(A, B)] =
      x.map((_, b))
```

Optrm [A]

$\mathbb{1} + A$

The type signature is of the form `F[A] => G[A]` if we define `type F[A] = (Option[A], B])` and `type G[A] = Option[(A, B)]` and consider `B` as a fixed type

Alternatively, consider `A` as a fixed type and obtain a natural transformation `K[B] => L[B]` with suitable definitions of `K[B]` and `L[B]`

- The naturality law can be verified directly
  - But it also follows from the product and co-product constructions for natural transformations (to be shown below)

More practical uses of natural transformations II

Building up natural transformations from parts

```
def toOptionList[A, B]: List[(Option[A], B)] => List[Option[(A, B)]] =
    _.map { case (x, b) => x.map((_, id)) }
```

b

- If we have a functor F and a natural transformation $G^A \to H^A$, we can implement a natural transformation $F^{G^A} \to F^{H^A}$ def t[A]: F[G[A]] => F[H[A]]
- In this example, the notation is $F = \text{List}$, $G^A = (\mathbb{1} + A) \times B$, and $H^A = \mathbb{1} + A \times B$
  - The type notation such as $(\mathbb{1} + A) \times B$ helps recognize type equivalences by using the rules of ordinary polynomial algebra:

$$(\mathbb{1} + A) \times B \cong \mathbb{1} \times B + A \times B \cong B + A \times B$$

- Another example: `List[(Try[A], B)] => List[Try[(A, B)]]` with the same code as `toOptionList[A, B]`
- Denote `Try[A]` by $E + A$ where E denotes the type of the exception

E+A ... Either[E,A]

$$\text{List}^{(E+A)\times B} \to \text{List}^{E+A\times B}$$

- To prove the general property, write out the naturality law

$t : G^A \to H^A$ $\qquad$ F – a functor

$u : F^{G^A} \to F^{H^A}$ $\qquad$ $u \triangleq t^{\uparrow F}$ $\qquad$ $(F \circ G)^A = F^{G^A}$

Naturality law: $\forall f : A \to B$:

$F[G[A]]$

$f^{\uparrow(F\circ H)} = (f^{\uparrow H})^{\uparrow F}$

$_.map(_.map(f))$

? — $f^{\uparrow(F\circ G)} ; u = u ; f^{\uparrow(F\circ H)}$

$$\underbrace{(f^{\uparrow G})^{\uparrow F} ; t^{\uparrow F}}_{(\ldots)^{\uparrow F} ; (\ldots)^{\uparrow F}} \overset{?}{=} t^{\uparrow F} ; (f^{\uparrow H})^{\uparrow F}$$

$$(f^{\uparrow G} ; t)^{\uparrow F} \overset{?}{=} (t ; f^{\uparrow H})^{\uparrow F}$$

✓

Naturality law of t:

$f^{\uparrow G} ; t = t ; f^{\uparrow H}$

$(\downarrow)^{\uparrow F} = (\downarrow)^{\uparrow F}$

More practical uses of natural transformations III

Using a constant functor ("phantom type parameter")

```
def length[A]: List[A] => Int = { _.length }
```

- The type signature is of the form `F[A] => G[A]` or $F^A \to G^A$ if we define $F = \text{List}$ and $G^A = \text{Int}$, so that G^A is a constant functor
- The naturality law gives $f^{\uparrow F} ; \text{length} = \text{length} ; f^{\uparrow G}$, but $\cancel{F}^{\uparrow G} = \text{id}$, so $f^{\uparrow F} ; \text{length} = \text{length}$ for any $f^{:A \to B}$
- We can choose $f(x) = c$ with any constant c
  - The length of a list does not depend on the values stored in the list

Reasoning with naturality I: Simplifying the `pure` method

The naturality law of `pure` for a functor L:

$$\begin{tikzcd} A \arrow[r, "\mathrm{pu}_L"] \arrow[d, "f"] & L^A \arrow[d, "f^{\uparrow L}"] \\ B \arrow[r, "\mathrm{pu}_L"] & L^B \end{tikzcd}$$

```
pure(a).map(f) == pure(f(a))
```

$$\mathrm{pu}_L \,⨾\, f^{\uparrow L} = f \,⨾\, \mathrm{pu}_L$$

$\mathrm{pu}_L : \underbrace{F^A}_{A} \to G^A$?

$F^A \cong A \quad f^{\uparrow F} = f$

Fix a value $b^{:B}$ and set $A = \mathbb{1}$ and $f \triangleq 1 \to b$ in the naturality law:

$$\begin{tikzcd} \mathbb{1} \arrow[r, "\mathrm{pu}_L"] \arrow[d, "1\to b"] & L^{\mathbb{1}} \arrow[d, "(1\to b)^{\uparrow L}"] \\ B \arrow[r, "\mathrm{pu}_L"] & L^B \end{tikzcd}$$

```
pure(()).map(_ => b) == pure(b)
```

$$\mathrm{pu}_L^{\mathbb{1}} \,⨾\, (1 \to b)^{\uparrow L} = (1 \to b) \,⨾\, \mathrm{pu}_L$$

$\mathrm{pu}_L(b) = \mathrm{pu}_L(1) \triangleright (1 \to b)$

() _ 1

$x \triangleright f = f(x)$

x.length ... x ▷ length

$\uparrow L$

We have expressed `pure(b)` via a constant value `pure(())` of type `L[Unit]`

The resulting function `pure` will automatically satisfy the naturality law!

The naturality law of `pure` makes it *equivalent* to a "wrapped unit" value

This simplifies the definition of a `Pointed` typeclass:

```
abstract class Pointed[L[_]: Functor] { def wu: L[Unit] }
```

val

Examples: for `Option`, `wu = Some(())`. For `List`, `wu = List(())`

Reasoning with naturality II: `flatMap` and `flatten`

Use the curried type signature for `flatMap` for a monad M:

$\text{ftn} : M^{M^A} \to M^A$

def flatten[A] : M[M[A]] => M[A]

```
def flatMap[A, B]: (A => M[B])=> M[A] => M[B]
```

$$\text{flm}^{A,B} : (A \to M^B) \to M^A \to M^B$$

The naturality law with respect to the type parameter `A`:

$$\begin{tikzcd} M^A \arrow[d, "f^{\uparrow M}"'] \arrow[dr, "\text{flm}\,(f \,⨾\, g)"] & \\ M^B \arrow[r, "\text{flm}\,(g)"] & M^C \end{tikzcd}$$

```
_.flatMap(f andThen g) == _.map(f).flatMap(g)
```

$$\text{flm}\,(\underbrace{f^{:A\to B} \,⨾\, g^{:B\to M^C}}_{:A \to M^C}) = f^{\uparrow M} \,⨾\, \text{flm}\,(g) \quad .$$

Express `flatMap` through `flatten`:

```
_.flatMap(f) == _.map(f).flatten
```

g g

$$\text{flm}\,(g) = g^{\uparrow M} \,⨾\, \text{ftn}$$

Express `flatten` through `flatMap`:

$$\text{ftn} = \text{flm}\,(\text{id}^{:M^A \to M^A})$$

The function `flatten` is equivalent to `flatMap` with naturality law

$$flm(g) = g^{\uparrow M} ; ftn \qquad ftn = flm(id)$$

$$flm(f;g) = f^{\uparrow M} ; flm(g)$$

1) Take flm satisfying the law

Define $ftn = flm(id)$

Define $flm'(g) = g^{\uparrow M} ; ftn = g^{\uparrow M} ; flm(id)$

Then $flm' = flm$

$flm'(g) = g^{\uparrow M} ; flm(id) = flm(g ; id) = flm(g)$

for any $g : A \to M^B$ ✓

2) Take ftn, define $flm(g) = g^{\uparrow M} ; ftn$

Then define $ftn' = flm(id) = id^{\uparrow M} ; ftn = id ; ftn = ftn$

Then show $ftn' = ftn$? ✓

3) If $flm(g) = g^{\uparrow M} ; ftn$ then flm satisfies the naturality law:

$$flm(f;g) = (f;g)^{\uparrow M} ; ftn$$

$$f^{\uparrow M} ; flm(g) = f^{\uparrow M} ; g^{\uparrow M} ; ftn = (f;g)^{\uparrow M} ; ftn$$ ✓

Reasoning with naturality III: The covariant Yoneda identity

We have shown that the set of all natural transformations $A \to L^A$ is equivalent to the set of all values $L^{\mathbb{1}}$ L[Unit]

This property can be generalized to any type Z instead of the unit type ($\mathbb{1}$):

The set of all natural transformations $(Z \to A) \to L^A$ is equivalent to the set of all values L^Z, where Z is a fixed type

To indicate that Z is fixed by A is varying within the natural transformation, use a type signature with the universal quantifier:

$$(\forall A.\, A \to L^A) \cong L^{\mathbb{1}}$$

$\forall A.$

$$(\forall A.\, (Z \to A) \to L^A) \cong L^Z \qquad \text{– the covariant Yoneda identity}$$

To prove:

1. Implement the isomorphism, $p : (\forall A.\, (Z \to A) \to L^A) \to L^Z$ and $q : L^Z \to \forall A.\, (Z \to A) \to L^A$
2. Show that $p \fatsemi q = \text{id}$ and $q \fatsemi p = \text{id}$

1) $p : \underbrace{(\forall A.\ (Z \to A) \to L^A)}_{y} \to L^Z$

def $y[A]$ ...

$p \triangleq y \to y^Z(id)$

def $p = \{y \Rightarrow y[Z](identity[Z])\}$

$q : L^Z \to (\forall A.\ (Z \to A) \to L^A)$

$q \triangleq \ell^{:L^Z} \to \forall A.\ (f^{:Z\to A} \to \ell \triangleright f^{\uparrow L})$

def $q = \ell \Rightarrow f \Rightarrow \ell.map(f)$

2) $\underline{p ; q = id}$, $\underline{q ; p = id}$.

Take any $y : \forall A.\ (Z \to A) \to L^A$, $y' \triangleq q(p(y))$; show $y' = y$.

$y' = (\ell \to \forall A.\ (f \to \ell \triangleright f^{\uparrow L}))(y(id)) = \forall A.\ f \to y(id) \triangleright f^{\uparrow L}$

$= \forall A.\ f \to \underbrace{id \triangleright y \triangleright f^{\uparrow L}}_{(identity).\ y.map(f)}$

Naturality law: $y ; f^{\uparrow L} = f^{\uparrow (Z \to \bullet)} ; y$

$y' = \forall A.\ f \to id \triangleright f^{\uparrow (Z\to\bullet)} \triangleright y = \forall A\ f \to (id ; f) \triangleright y = \forall A.\ \underbrace{f \to f \triangleright y}_{\underbrace{f \to y(f)}_{y}}$

$y' = y$. ✓

$p(q(\ell^{:L^Z})) \overset{?}{=} \ell$. ✓

$p(q(\ell)) =$

$p(\ (f^{:Z\to A} \to \ell \triangleright f^{\uparrow L}))$

$= (f \to \ell \triangleright f^{\uparrow L})(id)$

$= \ell \triangleright id^{\uparrow L} = \ell \triangleright id = \ell$.

3) Define $y = q(\ell)$ then y is a natural transformation.

$q(\ell) ; f^{\uparrow L} = f^{\uparrow (Z \to \bullet)} ; q(\ell)$

$q(\ell) = f \to \ell \triangleright f^{\uparrow L}$

$x \triangleright q(\ell) \triangleright f^{\uparrow L} = \ell \triangleright x^{\uparrow L} \triangleright f^{\uparrow L}$

$x \triangleright f^{\uparrow (Z \to \bullet)} ; q(\ell) = (x ; f) \triangleright q(\ell) = \ell \triangleright (x ; f)^{\uparrow L}$. ✓

$\ell \triangleright x^{\uparrow L} ; f^{\uparrow L} = \ell \triangleright (x ; f)^{\uparrow L}$.

$\ell \triangleright x^{\uparrow L} \triangleright f^{\uparrow L} = \ell \triangleright (x^{\uparrow L} ; f^{\uparrow L})$

Reasoning with naturality IV: other derivations

$f^{\uparrow K} ; t = t ; f^{\uparrow K} \ldots$

Naturality laws are often used in derivations of various typeclass laws
Within the 11 existing chapters of my upcoming free book, "*The Science of Functional Programming*" (https://github.com/winitzki/sofp), naturality laws are used at least 31 times in about 100 derivations

- Examples of such derivations:
  - Composition of two co-pointed functors is again co-pointed
    - ★ A functor F is co-pointed if there exists a natural transformation $\text{ex} : \forall A.\, F^A \to A$

$(M[A], N[A])$

  - The product of two monads is again a monad
  - The product of two monad transformers is again a monad transformer

The most useful derivation technique is rewriting equations

Example: properties of horizontal and vertical composition

Bartosz Milewski's book "Category theory for programmers", Chapter 10, defines the horizontal and the vertical composition of natural transformations

The vertical composition of $\alpha : F^A \to G^A$ and $\beta : G^A \to H^A$ is the ordinary function composition $(\alpha \fatsemi \beta) : F^A \to H^A$

The horizontal composition of $\alpha : F^A \to G^A$ and $\alpha' : F'^A \to G'^A$ is $(\alpha \star \alpha') : F^{F'^A} \to G^{G'^A}$

$F[F1[A]] \Rightarrow G[G1[A]]$

$\alpha ; \alpha'^{\uparrow G} =$
$\alpha'^{\uparrow F} ; \alpha$

- Both compositions again give natural transformations

If we have four natural transformations α, β, α', β' with type signatures

$$\alpha : F^A \to G^A \quad , \quad \beta : G^A \to H^A \quad ,$$
$$\alpha' : F'^A \to G'^A \quad , \quad \beta' : G'^A \to H'^A \quad ,$$

$\alpha \star \alpha' = \alpha ; \alpha'^{\uparrow G}$

we can write the distributive law,

$$(\alpha \fatsemi \beta) \star (\alpha' \fatsemi \beta') = (\alpha \star \alpha') \fatsemi (\beta \star \beta')$$

To prove that these properties hold, write out the naturality laws

$$(\alpha \fatsemi \beta) \star (\alpha' \fatsemi \beta') = (\alpha \star \alpha') \fatsemi (\beta \star \beta')$$

$\alpha \star \alpha' = \alpha ; \alpha'^{\uparrow G}$

l.h.s. $(\alpha ; \beta) \star (\alpha' ; \beta') = (\alpha ; \beta) ; (\alpha' ; \beta')^{\uparrow} = \alpha ; \beta ; \alpha'^{\uparrow} ; \beta'^{\uparrow}$

r.h.s. $(\alpha ; \alpha'^{\uparrow}) ; (\beta ; \beta'^{\uparrow}) = \alpha ; \alpha'^{\uparrow} ; \beta ; \beta'^{\uparrow}$ ✓

$\beta ; \alpha'^{\uparrow} = \alpha'^{\uparrow} ; \beta$

$f^{\uparrow}$??

$\alpha : F^{A} \to G^{A}, \quad \alpha' : F'^{A} \to G'^{A}$

$u = \alpha \star \beta = \alpha ; \alpha'^{\uparrow G} \} : F^{F'^{A}} \to G^{G'^{A}}$.

Naturality law of $\underline{u}$: $f^{\uparrow (F \circ F')} ; u = u ; f^{\uparrow (G \circ G')}$.

lhs $(f^{\uparrow F'})^{\uparrow F} ; \alpha ; \alpha'^{\uparrow G} \overset{?}{=} \alpha ; \alpha'^{\uparrow G} ; (f^{\uparrow G'})^{\uparrow G}$ ✓

$(f^{\uparrow F'})^{\uparrow F} ; \alpha = \alpha ; (f^{\uparrow F'})^{\uparrow G}$

lhs $= \alpha ; (f^{\uparrow F'})^{\uparrow G} ; \alpha'^{\uparrow G} = \alpha ; (f^{\uparrow F'} ; \alpha')^{\uparrow G} = \alpha ; (\alpha' ; f^{\uparrow G'})^{\uparrow G}$

$= \alpha ; \alpha'^{\uparrow G} ; (f^{\uparrow G'})^{\uparrow G}$ ✓

We'd like to compose α with β, which is a natural transformation from functor G to H. The component of β at a is a morphism:

$$\beta_a :: Ga \to Ha$$

These morphisms are composable and their composition is another morphism:

$$\beta_a \circ \alpha_a :: Fa \to Ha$$

We will use this morphism as the component of the natural transformation $\beta \cdot \alpha$ — the composition of two natural transformations β after α:

$$(\beta \cdot \alpha)_a = \beta_a \circ \alpha_a$$

One (long) look at a diagram convinces us that the result of this composition is indeed a natural transformation from F to H:

$$Hf \circ (\beta \cdot \alpha)_a = (\beta \cdot \alpha)_b \circ Ff$$

$\beta \circ \alpha = \alpha ; \beta$

$Hf = f^{\uparrow H}$

$\alpha ; \beta ; f^{\uparrow H} = \underline{f^{\uparrow F} ; \alpha ; \beta}$

$\Downarrow$

$\alpha ; \underline{f^{\uparrow G} ; \beta}$

$\alpha ; \beta ; f^{\uparrow H}$

Notice that we cannot apply vertical composition to this pair, because the target of α is different from the source of β. In fact they are members of two different functor categories: $\mathbf{D}^{\mathbf{C}}$ and $\mathbf{E}^{\mathbf{D}}$. We can, however, apply composition to the functors F' and G', because the target of F' is the source of G' — it's the category $\mathbf{D}$. What's the relation between the functors $G' \circ F'$ and $G \circ F$?

Having α and β at our disposal, can we define a natural transformation from $G \circ F$ to $G' \circ F'$? Let me sketch the construction.

As usual, we start with an object a in $\mathbf{C}$. Its image splits into two objects in $\mathbf{D}$: Fa and $F'a$. There is also a morphism, a component of α, connecting these two objects:

$$\alpha_a :: Fa \to F'a$$

Again, following Mac Lane I use the small circle for horizontal composition, although you may also encounter star in its place.

Here's a categorical rule of thumb: Every time you have composition, you should look for a category. We have vertical composition of natural transformations, and it's part of the functor category. But what about the horizontal composition? What category does that live in?

The way to figure this out is to look at **Cat** sideways. Look at natural transformations not as arrows between functors but as arrows between categories. A natural transformation sits between two categories, the ones that are connected by the functors it transforms. We can think of it as connecting these two categories.

Let's focus on two objects of **Cat** — categories **C** and **D**. There is a set of natural transformations that go between functors that connect **C** to **D**. These natural transformations are our new arrows from **C** to **D**. By the same token, there are natural transformations going between functors that connect **D** to **E**, which we can treat as new arrows going from **D** to **E**. Horizontal composition is the composition of these arrows.

We also have an identity arrow going from **C** to **C**. It's the identity natural transformation that maps the identity functor on **C** to itself. Notice that the identity for horizontal composition is also the identity for vertical composition, but not vice versa.

Finally, the two compositions satisfy the interchange law:

$$(\beta' \cdot \alpha') \circ (\beta \cdot \alpha) = (\beta' \circ \beta) \cdot (\alpha' \circ \alpha)$$

I will quote Saunders Mac Lane here: The reader may enjoy writing down the evident diagrams needed to prove this fact. ??

There is one more piece of notation that might come in handy in the future. In this new sideways interpretation of **Cat** there are two ways of getting from object to object: using a functor or using a natural transformation. We can, however, re-interpret the functor arrow as a special kind of natural transformation: the identity natural transformation acting on this functor. So you'll often see this notation:

$$F \circ \alpha$$

where F is a functor from **D** to **E**, and α is a natural transformation between two functors going from **C** to **D**. Since you can't compose a functor with a natural transformation, this is interpreted as a horizontal composition of the identity natural transformation 1_F after α.

Similarly:

$$\alpha \circ F$$

is a horizontal composition of α after 1_F.

10.5 Conclusion

This concludes the first part of the book. We've learned the basic vocabulary of category theory. You may think of objects and categories as nouns; and morphisms, functors, and natural transformations as verbs. Morphisms connect objects, functors connect categories, natural transformations connect functors.

But we've also seen that, what appears as an action at one level of abstraction, becomes an object at the next level. A set of morphisms turns into a function object. As an object, it can be a source or a target of another morphism. That's the idea behind higher order functions.

Other constructions of natural transformations

Natural transformations can be combined in several other ways

Given natural transformations `a[A]: F[A] => G[A]` and `b[A]: K[A] => L[A]`:

- pair product, `((F[A], K[A])) => (G[A], L[A])`
- pair co-product, `Either[F[A], K[A]] => Either[G[A], L[A]]`
- pair exponential, `(F[A] => K[A]) => (G[A] => L[A])` where `F[A]` and `G[A]` must be contrafunctors
  - are natural transformations implemented by combining `a[A]` and `b[A]`

Also, the identity function `identity[A]: A => A` and the constant unit function of type `A => Unit` are natural transformations

It follows that any purely functional combination of natural transformations is again a natural transformation

No need to verify the naturality law in each case

Example: `(Option[A], B) => Option[(A, B)]`

Summary of the type notation

The short type notation helps in symbolic reasoning about types

Description	Scala examples	Notation
Typed value	`x: Int`	$x^{:\text{Int}}$ or $x : \text{Int}$
Unit type	`Unit`, `Nil`, `None`	$\mathbb{1}$
Type parameter	`A`	A
Product type	`(A, B)` or `case class P(x: A, y: B)`	$A \times B$
Co-product type	`Either[A, B]`	$A + B$
Function type	`A => B`	$A \to B$
Type constructor	`List[A]`	List^A
Universal quantifier	`trait P { def f[A]: Q[A] }`	$P \triangleq \forall A.\, Q^A$
Existential quantifier	`sealed trait P[A]` `case class Q[A, B]() extends P[A]`	$P^A \triangleq \exists B.\, Q^{A,B}$

Example: Scala code `def flm(f: A => Option[B]): Option[A] => Option[B]` is denoted by $\text{flm} : (A \to \mathbb{1} + B) \to \mathbb{1} + A \to \mathbb{1} + B$

Summary of the code notation

The short code notation helps in symbolic reasoning about code

Scala examples	Notation
`()` or `true` or `"abc"` or `123`	1, true, "abc", 123
`def f[A](x: A) = ...`	$f^{A}(x^{:A}) \triangleq \ldots$
`{ (x: A) => expr }`	$x^{:A} \to \text{expr}$
`f(x)` or `x.pipe(f)` (Scala 2.13)	$f(x)$ or $x \triangleright f$
`val p: (A, B) = (a, b)`	$p^{:A\times B} \triangleq a \times b$
`{case (a, b) => expr}` or `p._1` or `p._2`	$a \times b \to \text{expr}$ or $p \triangleright \pi_1$ or $p \triangleright \pi_2$
`Left[A, B](x)` or `Right[A, B](y)`	$x^{:A} + \mathbb{0}^{:B}$ or $\mathbb{0}^{:A} + y^{:B}$

`val q: C = (p: Either[A, B]) match {`
`  case Left(x) => f(x)`
`  case Right(y) => g(y)`
`}`
	$q^{:C} \triangleq p^{:A+B} \triangleright \begin{array}{c\|\|c} & C \\ \hline A & x^{:A} \to f(x) \\ B & y^{:B} \to g(y) \end{array}$

`def f(x) = { ... f(y) ... }`	$f(x) \triangleq \ldots \overline{f}(y) \ldots$
`f andThen g` and `(f andThen g)(x)`	$f \fatsemi g$ and $x \triangleright f \fatsemi g$ or $x \triangleright f \triangleright g$
`p.map(f).map(g)`	$p \triangleright f^{\uparrow F} \triangleright g^{\uparrow F}$ or $p \triangleright f^{\uparrow F} \fatsemi g^{\uparrow F}$

Summary

- Use naturality laws to obtain refactoring guaranteed to be correct
- Recognize and refactor code to use natural transformations
- Naturality laws allow us to reduce the number of type parameters in certain functions

  pure[A] wu : L[Unit]
- Short notation for code helps derive properties via symbolic calculations

  flatMap[A,B] flatten[A]
  - which is more efficient than "staring at diagrams"
- Full details and proofs are in the free upcoming book
  - Draft of the book: https://github.com/winitzki/sofp