

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:

<code>def toList[A]: Stream[A] => List[A]</code>	$\text{toList}^A : \text{Str}^A \rightarrow \text{List}^A$
<code>val f: A => B</code>	$f:A \rightarrow B$
<code>_.map(f) with type List[A] => List[B]</code>	$f^{\uparrow \text{List}}$
<code>_.toList.map(f)</code>	$\text{toList} \circ f^{\uparrow \text{List}}$
<code>f andThen g</code>	$f \circ g$
<code>_.map(f).map(g) == _.map(f andThen g)</code>	$f^{\uparrow \text{List}} \circ g^{\uparrow \text{List}} = (f \circ g)^{\uparrow \text{List}}$

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 $\text{toList}^A : \text{Str}^A \rightarrow \text{List}^A$

$$\begin{array}{ccc} \text{Str}^A & \xrightarrow{\text{toList}^A} & \text{List}^A \\ \downarrow f^{\uparrow \text{Str}} & & \downarrow f^{\uparrow \text{List}} \\ \text{Str}^B & \xrightarrow{\text{toList}^B} & \text{List}^B \end{array}$$

$$_.\text{toList}.\text{map}(f) == _.\text{map}(f).\text{toList}$$

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

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

$$\begin{array}{ccc} \text{Str}^A & \xrightarrow{\text{filt}^A(f \circ p)} & \text{Str}^A \\ \downarrow f^{\uparrow \text{Str}} & & \downarrow f^{\uparrow \text{Str}} \\ \text{Str}^B & \xrightarrow{\text{filt}^B(p)} & \text{Str}^B \end{array}$$

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

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

$$_.\text{map}(f).\text{filter}(p) == _.\text{filter}(x \Rightarrow p(f(x))).\text{map}(f)$$

- 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

$$\begin{array}{ccc} \text{List}^A & \xrightarrow{\text{headOpt}^A} & \text{Opt}^A \\ \downarrow f^{\uparrow \text{List}} & & f^{\uparrow \text{Opt}} \downarrow \\ \text{List}^B & \xrightarrow{\text{headOpt}^B} & \text{Opt}^B \end{array} \quad \begin{array}{l} _.\text{map}(f).\text{headOption} == _.\text{headOption}.\text{map}(f) \\ f^{\uparrow \text{List}} \circ \text{headOpt} = \text{headOpt} \circ f^{\uparrow \text{Opt}} \end{array}$$

- 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; $\text{reverse}^A : \text{List}^A \rightarrow \text{List}^A$

$$\begin{array}{l} _.\text{map}(f).\text{reverse} == _.\text{reverse}.\text{map}(f) \\ (f^{A \rightarrow B})^{\uparrow \text{List}} \circ \text{reverse}^B = \text{reverse}^A \circ (f^{A \rightarrow B})^{\uparrow \text{List}} \end{array}$$

- The pure method, $\text{pure}[A] : A \Rightarrow L[A]$. Notation: $\text{pu}_L : A \rightarrow L^A$

$$\begin{array}{l} \text{pure}(x).\text{map}(f) == \text{pure}(f(x)) \\ \text{pu}^A \circ (f^{A \rightarrow B})^{\uparrow L} = f^{A \rightarrow B} \circ \text{pu}^B \end{array}$$

Natural transformations and their laws

A **natural transformation** is a function t with type signature $F^A \rightarrow G^A$ that satisfies the naturality law $f^{\uparrow F} \circ t = t \circ f^{\uparrow G}$. 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:
 - ▶ For `flatMap`, denote $\text{flm} : (A \rightarrow M^B) \rightarrow M^A \rightarrow M^B$, fix A
 - ★ $\text{flm} : F^B \rightarrow G^B$ where $F^B \triangleq A \rightarrow M^B$ and $G^B \triangleq M^A \rightarrow M^B$
 - ▶ The naturality law $f^{\uparrow F} \circ \text{flm} = \text{flm} \circ f^{\uparrow G}$ then gives the equation

$$\text{flm}(p^{A \rightarrow M^B} \circ f^{\uparrow M}) = \text{flm}(p^{A \rightarrow M^B}) \circ f^{\uparrow M}$$

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

$$f^{\uparrow F} = p^{A \rightarrow M^B} \rightarrow p \circ f^{\uparrow M} \quad , \quad f^{\uparrow G} = q^{M^A \rightarrow M^B} \rightarrow q \circ f^{\uparrow M}$$

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))
```

The type signature is of the form $F[A] \Rightarrow G[A]$ 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
 $K[B] \Rightarrow L[B]$ 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)) }
```

- If we have a functor F and a natural transformation $G^A \rightarrow H^A$, we can implement a natural transformation $F^{G^A} \rightarrow F^{H^A}$
- 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

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

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

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 $F[A] \Rightarrow G[A]$ or $F^A \rightarrow G^A$ if we define $F = \text{List}$ and $G^A = \text{Int}$, so that G^A is a constant functor
- The naturality law gives $f^{\uparrow F} \circ \text{length} = \text{length} \circ f^{\uparrow G}$, but $F^{\uparrow G} = \text{id}$, so $f^{\uparrow F} \circ \text{length} = \text{length}$ for any $f: A \rightarrow B$
- 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 :

$$\begin{array}{ccc} A & \xrightarrow{\text{pu}_L} & L A \\ \downarrow f & & \downarrow f^{\uparrow L} \\ B & \xrightarrow{\text{pu}_L} & L B \end{array}$$

$$\text{pure}(a).\text{map}(f) == \text{pure}(f(a))$$

$$\text{pu}_L \circ f^{\uparrow L} = f \circ \text{pu}_L$$

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

$$\begin{array}{ccc} \mathbb{1} & \xrightarrow{\text{pu}_L} & L \mathbb{1} \\ \downarrow 1 \rightarrow b & & \downarrow (1 \rightarrow b)^{\uparrow L} \\ B & \xrightarrow{\text{pu}_L} & L B \end{array}$$

$$\text{pure}(()).\text{map}(_ \Rightarrow b) == \text{pure}(b)$$

$$\text{pu}_L \circ (1 \rightarrow b)^{\uparrow L} = (1 \rightarrow b) \circ \text{pu}_L$$

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] }
```

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 :

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

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

The naturality law with respect to the type parameter A :

$$\begin{array}{ccc} M^A & & \\ f^{\uparrow M} \downarrow & \searrow \text{flm}(f \circ g) & \\ M^B & \xrightarrow{\text{flm}(g)} & M^C \end{array}$$

$$_.\text{flatMap}(f \text{ andThen } g) == _.\text{map}(f).\text{flatMap}(g)$$

$$\text{flm}(f^{A \rightarrow B} \circ g^{B \rightarrow M^C}) = f^{\uparrow M} \circ \text{flm}(g) \quad .$$

Express flatMap through flatten:

$$_.\text{flatMap}(f) == _.\text{map}(f).\text{flatten}$$

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

Express flatten through flatMap:

$$\text{ftn} = \text{flm}(\text{id}^{M^A \rightarrow M^A})$$

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

Reasoning with naturality III: The covariant Yoneda identity

We have shown that the set of all natural transformations $A \rightarrow L^A$ is equivalent to the set of all values L^1

This property can be generalized to any type Z instead of the unit type (1): The set of all natural transformations $(Z \rightarrow A) \rightarrow L^A$ 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:

$$\begin{aligned} (\forall A. A \rightarrow L^A) &\cong L^1 \\ (\forall A. (Z \rightarrow A) \rightarrow L^A) &\cong L^Z \quad \text{– the covariant Yoneda identity} \end{aligned}$$

To prove:

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

Reasoning with naturality IV: other derivations

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 $\text{ex} : \forall A. F^A \rightarrow A$
- ▶ 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 $\alpha : F^A \rightarrow G^A$ and $\beta : G^A \rightarrow H^A$ is the ordinary function composition $(\alpha \circ \beta) : F^A \rightarrow H^A$

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

- Both compositions again give natural transformations

If we have four natural transformations $\alpha, \beta, \alpha', \beta'$ with type signatures

$$\begin{aligned} \alpha : F^A \rightarrow G^A \quad , \quad \beta : G^A \rightarrow H^A \quad , \\ \alpha' : F'^A \rightarrow G'^A \quad , \quad \beta' : G'^A \rightarrow H'^A \quad , \end{aligned}$$

we can write the distributive law,

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

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

Other constructions of natural transformations

Natural transformations can be combined in several other ways

Given natural transformations $a[A]: F[A] \Rightarrow G[A]$ and $b[A]: K[A] \Rightarrow L[A]$:

- pair product, $((F[A], K[A])) \Rightarrow (G[A], L[A])$
- pair co-product, $\text{Either}[F[A], K[A]] \Rightarrow \text{Either}[G[A], L[A]]$
- pair exponential, $(F[A] \Rightarrow K[A]) \Rightarrow (G[A] \Rightarrow L[A])$ where $F[A]$ and $G[A]$ must be contrafunctors
 - ▶ are natural transformations implemented by combining $a[A]$ and $b[A]$

Also, the identity function $\text{identity}[A]: A \Rightarrow A$ and the constant unit function of type $A \Rightarrow \text{Unit}$ 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: $(\text{Option}[A], B) \Rightarrow \text{Option}[(A, B)]$

Summary of the type notation

The short type notation helps in symbolic reasoning about types

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

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

Summary of the code notation

The short code notation helps in symbolic reasoning about code

Scala examples	Notation
() or <code>true</code> or <code>"abc"</code> or 123	1, true, "abc", 123
<code>def f[A](x: A) = ...</code>	$f^A(x^A) \triangleq \dots$
<code>{ (x: A) => expr }</code>	$x^A \rightarrow \text{expr}$
<code>f(x)</code> or <code>x.pipe(f)</code> (Scala 2.13)	$f(x)$ or $x \triangleright f$
<code>val p: (A, B) = (a, b)</code>	$p^{A \times B} \triangleq a \times b$
<code>{case (a, b) => expr}</code> or <code>p._1</code> or <code>p._2</code>	$a \times b \rightarrow \text{expr}$ or $p \triangleright \pi_1$ or $p \triangleright \pi_2$
<code>Left[A, B](x)</code> or <code>Right[A, B](y)</code>	$x^A + 0^B$ or $0^A + y^B$
<code>val q: C = (p: Either[A, B]) match {</code> <code> case Left(x) => f(x)</code> <code> case Right(y) => g(y)</code> <code>}</code>	$q^C \triangleq p^{A+B} \triangleright \begin{array}{c c} & C \\ \hline A & x^A \rightarrow f(x) \\ B & y^B \rightarrow g(y) \end{array}$
<code>def f(x) = { ... f(y) ... }</code>	$f(x) \triangleq \dots \bar{f}(y) \dots$
<code>f andThen g</code> and <code>(f andThen g)(x)</code>	$f \circ g$ and $x \triangleright f \circ g$ or $x \triangleright f \triangleright g$
<code>p.map(f).map(g)</code>	$p \triangleright f^{\uparrow F} \triangleright g^{\uparrow F}$ or $p \triangleright f^{\uparrow F} \circ g^{\uparrow F}$

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
- Short notation for code helps derive properties via symbolic calculations
 - ▶ 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>