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]`

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]`	$f^{\uparrow\text{List}}$
`_.toList.map(f)`	$\text{toList} \,{}^{\circ}_{\circ}\, f^{\uparrow\text{List}}$
`f andThen g`	$f \,{}^{\circ}_{\circ}\, g$
`_.map(f).map(g) == _.map(f andThen g)`	$f^{\uparrow\text{List}} \,{}^{\circ}_{\circ}\, g^{\uparrow\text{List}} = (f \,{}^{\circ}_{\circ}\, g)^{\uparrow\text{List}}$

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

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

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

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

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

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

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

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

- 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

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

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

$$(f^{:A\to B})^{\uparrow\text{List}} ⨾ \text{headOpt} = \text{headOpt} ⨾ (f^{:A\to B})^{\uparrow\text{Opt}}$$

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

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

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

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

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

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

Natural transformations and their laws

A **natural transformation** is a function t with type signature $F^A \to G^A$ that satisfies the naturality law $f^{\uparrow F} \fcmp t = t \fcmp f^{\uparrow G}$. Notation $t : F \leadsto G$
Mnemonic rule: if $t : F \leadsto G$ 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 $\text{flm} : \left(A \to M^B\right) \to M^A \to M^B$, 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 $f^{\uparrow F} \fcmp \text{flm} = \text{flm} \fcmp f^{\uparrow G}$ then gives the equation

$$\text{flm}\,(p^{:A\to M^B} \fcmp f^{\uparrow M}) = \text{flm}\,(p^{:A\to M^B}) \fcmp f^{\uparrow M}$$

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

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

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] => G[A]` if we define

```
type F[A] = (Option[A], B])
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 parametricity theorem

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 $G^A \to H^A$, we can implement a natural transformation $F^{G^A} \to F^{H^A}$
- 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
- Denote `Try[A]` by $E + A$ where E denotes the type of the exception

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

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 $F^{\uparrow G} = \text{id}$, 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: Simplifying the `pure` method

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

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

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

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

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

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

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

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

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: `flatMap` and `flatten`

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

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

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

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

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

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

$$\mathrm{flm}\,(f^{:A\to B} \,;\, g^{:B\to M^C}) = f^{\uparrow M} \,;\, \mathrm{flm}\,(g) \quad .$$

Express `flatMap` through `flatten`:

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

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

Express `flatten` through `flatMap`:

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

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

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 $L^{\mathbb{1}}$

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

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:

$$\left(\forall A.\, A \rightarrow L^A\right) \cong L^{\mathbb{1}}$$

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

To prove:

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

Reasoning with naturality laws

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 horizontal composition of $\alpha : F^A \to G^A$ and $\beta : G^A \to H^A$ is the ordinary function composition $(\alpha\,;\beta) : F^A \to H^A$

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

- Both compositions again give natural transformations

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

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

we can write the distributive law,

$$(\alpha\,;\beta) \star (\alpha'\,;\beta') = (\alpha \star \alpha')\,;(\beta \star \beta')$$

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 two 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

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

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`	$A \rightarrow B$
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 $\text{flm} : (A \rightarrow \mathbb{1} + B) \rightarrow \mathbb{1} + A \rightarrow \mathbb{1} + B$

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) = ...`	$f^A(x^{:A}) \triangleq \dots$
`{ (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) ... }`	$f(x) \triangleq \dots \overline{f}(y) \dots$
`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

- Naturality laws can be used for guaranteed correct refactoring
  - Naturality laws allow us to reduce the number of type parameters
- Full details and proofs are in the free upcoming book
  - Draft of the book: https://github.com/winitzki/sofp