Parametricity properties of purely functional code

"Theorems for free" demystified. A tutorial, with code examples in Scala

Sergei Winitzki

San Francisco Types, Theorems, and Programming Languages

2020-03-26

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`

$$\text{getRows} \fatsemi \underline{\text{toList} \fatsemi \text{getUserInfo}^{\uparrow\text{List}}} = \text{getRows} \fatsemi \underline{\text{getUserInfo}^{\uparrow\text{Stream}} \fatsemi \text{toList}}$$

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)`

$$\text{getRows} \fatsemi \underline{\text{getUserInfo}^{\uparrow\text{Stream}} \fatsemi \text{filt}\,(\text{isValid})}$$
$$= \text{getRows} \fatsemi \underline{\text{filt}\,(\text{getUserInfo} \fatsemi \text{isValid}) \fatsemi \text{getUserInfo}^{\uparrow\text{Stream}}}$$

- These refactorings are guaranteed to be correct

Refactored code: further examples

Writing the previous examples as equations:

`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
```

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

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

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

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

$$(f^{:A\to B})^{\uparrow\text{Str}} \, ⨾ \, \text{filt}^B(p^{:B\to\mathbb{2}}) = \text{filt}^A(f \, ⨾ \, p) \, ⨾ \, f^{\uparrow\text{Str}}$$

- 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

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

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

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

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

- Lifting f before t equals to lifting f after t
  - Intuition: t rearranges data in a collection regardless of value types

Further examples:

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

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

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

Resoning with naturality: 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))
```

$$f \,{}^\circ_9\, \mathrm{pu}_L = \mathrm{pu}_L \,{}^\circ_9\, f^{\uparrow L}$$

Fix a value $b^{:B}$ and set $f \triangleq 1 \to b$ and $A = \mathbb{1}$ 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)
```

$$\mathrm{pu}_L \,{}^\circ_9\, (1 \to b)^{\uparrow L} = (1 \to b) \,{}^\circ_9\, \mathrm{pu}_L$$

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

Fully parametric code: example

Fully parametric code: works in the same way for all types

- Example of a fully parametric function:

```
def headOpt[A]: List[A] => Option[A] = {
  case Nil => None
  case head :: tail => Some(head)
}
```

- The same code in the matrix notation:

$$\text{headOpt}^{:\text{List}^A\to\mathbb{1}+A} \triangleq \begin{array}{|c||cc|} & \mathbb{1} & A \\ \hline \mathbb{1} & \text{id} & \mathbb{0} \\ A\times\text{List}^A & \mathbb{0} & h\times t\to h \end{array}$$

where $\text{List}^A \triangleq \mathbb{1} + A \times \text{List}^A$ is a recursively defined type constructor

```
final case class List[A](x: Option[(A, List[A]])
```

- The fully parametric function `headOption` is a natural transformation between functors `List` and `Option`

Naturality laws express the programmer's intuition about the properties of fully parametric code

Conditions for code to be fully parametric

- All argument types are combinations of type parameters
- All type parameters are treated as unknown, arbitrary types
- No hard-coded values of specific types (`123: Int` or `"abc": String`)
- No side effects (printing, `var x`, mutating values, writing files, networking, starting or stopping new threads, etc.)
- No `null`, no `throw`ing of exceptions, no run-time type comparison
- No run-time code loading, no external libraries with unknown code

Purely functional code is fully parametric if restricted to using only `Unit` type or type parameters (no specific types or values of specific types)

Purely functional programs are written using the 9 code constructions:

```
def fmap[A, B](f: A => B): List[(A, A)] => List[(B, B)] = { // 3
   case Nil              => Nil
//   8   1                  1,7
   case head :: tail     => (f (head._1), f (head._2)) :: fmap(f)(tail)
//   8       6               2 4      6  5 2 4      6    7   9
}
```

1. Use `Unit` value (or a "named `Unit`"), e.g. `()`, `Nil`, or `None`. Notation: 1
2. Use bound variable (a given argument of the function). Notation: x
3. Create function: `{ x => expr(x) }`. Notation: $x \to \text{expr}\,(x)$
4. Use function: `f(x)`. Notation: $f(x)$ or $x \triangleright f$
5. Create tuple: `(a, b)`. Notation: $a \times b$
6. Use tuple: `p._1`. Notation: $\nabla_1 p$ or $p \triangleright \nabla_1$
7. Create disjunctive value: `Left[A, B](x)`. Notation: $x^{:A} + \mathbb{0}^{:B}$
8. Use disjunctive value: `{ case ... }` (pattern-matching); matrix code
9. Use recursive call: `fmap(f)(tail)`. Notation: $\overline{\text{fmap}_{\text{List}}}(f)(t)$

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

Naturality laws in typeclasses

Another use of naturality laws is when implementing typeclasses

- Typeclasses require type constructors with methods `map`, `filter`, `fold`, `flatMap`, `pure`, and others

To be useful for programming, the methods must satisfy certain laws

- `map`: identity, composition
- `filter`: identity, composition, partial function, naturality
- `fold` (traverse): identity, composition, naturality
- `flatMap`: identity, associativity, naturality
- `pure`: naturality

We need to check the laws when implementing new typeclass instances

Naturality laws and parametricity

- The **parametricity theorem** guarantees that all naturality laws hold as long as the method's code is purely functional
- This saves us time: *no need* to check the naturality laws

Using the parametricity theorem is difficult

- The "theorems for free" (Reynolds; Wadler) approach needs to replace functions (one-to-one or many-to-one) by "relations" (many-to-many)
  - Derive a law with relation variables, then replace them by functions
- Alternative approach: analysis of dinatural transformations derives the naturality laws directly (Bainbridge et al.; Backhouse; de Lataillade)
  - See also a 2019 paper by Voigtländer
- Plan:
  - Introduce profunctors and dinatural transformations
  - Derive the naturality laws for dinatural transformations

Type constructors with two type parameters

In particular: bifunctors and profunctors

- In Scala syntax: `L[A, B]`. Example: `type L[A, B] = Either[(A, B), B]`
- In the type notation: $L^{A,B}$. Example: $L^{A,B} \triangleq A \times B + B$
- If a type constructor is **purely functional**, its type parameters will be either in covariant or in contravariant positions
- **Bifunctors**: both type parameters are always in covariant positions
  - Example: `L[A, B]` defined above is a bifunctor
  - Method `bimap[A, B, C, D](f: A => C, g: B => D): L[A, B] => L[C, D]`
  - Laws: identity and composition for `bimap`
- **Profunctors**: one type parameter contravariant, the other covariant
  - Example: `type P[X, Y] = Option[X] => (Y, Y)` or $P^{X,Y} \triangleq \mathbb{1} + X \rightarrow Y \times Y$
  - Method `xmap[A, B, C, D](f: C => A, g: B => D): P[A, B] => P[C, D]`
  - Laws: identity and composition for `xmap`
- If `L[A, B]` is a functor separately in `A` and `B`, is it a bifunctor?
- If `P[A, B]` is contravariant in `A` and covariant in `B`, is it a profunctor?
- They are but only if all liftings in `A` commute with liftings in `B`
  - These are the "commutativity laws" of bifunctors and profunctors

Applying `map` to bifunctors and profunctors

The `map` method can be applied with respect to only one type parameter

- In a bifunctor `L[A, B]`, fix `B`. Denote the resulting functor by $L^{\bullet,B}$
  - In the Scala syntax with "kind projector": `L[?, B]`
  - Lifting a function $f^{:U\to V}$ is denoted by $f^{\uparrow L^{\bullet,B}}:L^{U,B}\to L^{V,B}$
  - If fixing `A` instead, a lifting is denoted by $f^{\uparrow L^{A,\bullet}}:L^{A,U}\to L^{A,V}$
  - **Commutativity law** for bifunctors: $f^{\uparrow L^{\bullet,B}}\fatsemi(g^{:B\to C})^{\uparrow L^{V,\bullet}}=g^{\uparrow L^{U,\bullet}}\fatsemi f^{\uparrow L^{\bullet,C}}$
- In a profunctor `P[A, B]`, fix `B`. The resulting *contrafunctor* is $P^{\bullet,B}$
  - Lifting a function $f^{:U\to V}$ is denoted by $f^{\downarrow P^{\bullet,B}}:P^{V,B}\to P^{U,B}$
  - If fixing `A` instead, a lifting is denoted by $f^{\uparrow P^{A,\bullet}}:P^{A,U}\to P^{A,V}$
    - ★ For brevity, we may denote these liftings by $f^{\downarrow P}$ and $f^{\uparrow P}$ unambiguously
  - **Commutativity law** for profunctors: $f^{\downarrow P}\fatsemi g^{\uparrow P}=g^{\uparrow P}\fatsemi f^{\downarrow P}$

$$
\begin{tikzcd}
P^{A,B} \arrow[r, "(f^{:C\to A})^{\downarrow P}"'] \arrow[d, "(g^{:B\to D})^{\uparrow P}"'] & P^{C,B} \arrow[d, "(g^{:B\to D})^{\uparrow P}"] \\
P^{A,D} \arrow[r, "(f^{:C\to A})^{\downarrow P}"] & P^{C,D}
\end{tikzcd}
$$

- Commutativity laws hold for *all* purely functional type constructors
  - It is not necessary to verify the bifunctor and profunctor laws!
- Proof is by induction on the type structure of $P^{X,Y}$

Proof of the composition law of `xmap`

If $P^{A,B}$ is a functor in a and a contrafunctor in B, define `xmap` by:

$$\begin{tikzcd}
P^{A,B} \arrow[r, "(f^{:C\to A})^{\downarrow P}"] \arrow[dr, swap, "\text{xmap}(f)(g)\triangleq"] & P^{C,B} \arrow[d, "g^{\uparrow P}"] \\
 & P^{C,D}
\end{tikzcd}
\qquad \text{xmap}\,(f^{:C\to A})\,(g^{:B\to D}) \triangleq f^{\downarrow P} \fatsemi g^{\uparrow P}$$

The `xmap` composition law:

$$\text{xmap}\,(f_1)\,(g_1) \fatsemi \text{xmap}\,(f_2)\,(g_2) = \text{xmap}\,(f_2 \fatsemi f_1)\,(g_1 \fatsemi g_2)$$

Proof uses the commutativity law, $f^{\downarrow P} \fatsemi g^{\uparrow P} = g^{\uparrow P} \fatsemi f^{\downarrow P}$, for f_2 and g_1:

$$\begin{aligned}
&\text{xmap}\,(f_1)\,(g_1) \fatsemi \text{xmap}\,(f_2)\,(g_2) = f_1^{\downarrow P} \fatsemi \underline{g_1^{\uparrow P} \fatsemi f_2^{\downarrow P}} \fatsemi g_2^{\uparrow P} \\
&= \underline{f_1^{\downarrow P} \fatsemi f_2^{\downarrow P}} \fatsemi \underline{g_1^{\uparrow P} \fatsemi g_2^{\uparrow P}} = (f_2 \fatsemi f_1)^{\downarrow P} \fatsemi (g_1 \fatsemi g_2)^{\uparrow P} \\
&= \text{xmap}\,(f_2 \fatsemi f_1)\,(g_1 \fatsemi g_2)
\end{aligned}$$

Natural transformations

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 $t : F \leadsto G$

- 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 $f^{\uparrow F} ; \text{flm} = \text{flm} ; f^{\uparrow G}$ then gives the equation

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

The naturality law for $t^A : F^A \to G^A$ when F^A, G^A are contrafunctors:

$$\begin{tikzcd} F^A \arrow[r, "t^A"] \arrow[d, "(f^{:B\to A})^{\downarrow F}"'] & G^A \arrow[d, "f^{\downarrow G}"] \\ F^B \arrow[r, "t^B"] & G^B \end{tikzcd}$$

$$f^{\downarrow F} ; t = t ; f^{\downarrow G}$$

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

Dinatural transformations and profunctors

Some methods do *not* have the type signature of the form $F^A \to G^A$

- `find[A]: (A => Boolean) => List[A] => Option[A]`
- `fold[A, B]: List[A] => B => (A => B => B) => B` with respect to `B`
  - The type parameter is in contravariant and covariant positions at once
  - This gives us neither a functor nor a contrafunctor
- Solution: use a profunctor $P^{X,Y}$ with equal type parameters: $P^{A,A}$

A **dinatural transformation** is a function t with type signature $P^{A,A} \to Q^{A,A}$ that satisfies the naturality law $f^{\downarrow P} \fcmp t \fcmp f^{\uparrow Q} = f^{\uparrow P} \fcmp t \fcmp f^{\downarrow Q}$ where $P^{X,Y}$ and $Q^{X,Y}$ are suitable profunctors

- *All pure functions* have the type signature of a dinatural transformation
- *All* naturality laws (also for `find`, `fold`) are derived in this way
- The corresponding naturality law is guaranteed by parametricity
- Proof of parametricity theorem is a direct proof that any pure function t satisfies its law, by induction on the code structure of t. The proof depends on the profunctor commutativity law and the lifting codes for $f^{\uparrow P}$ and $f^{\downarrow P}$.

The naturality law for dinatural transformations

Given two profunctors $P^{X,Y}$ and $Q^{X,Y}$ and a function $t^A : P^{A,A} \to Q^{A,A}$
The naturality law is an equation for functions $P^{B,A} \to Q^{A,B}$:

$$f^{\downarrow P^{\bullet,A}} \fatsemi t^A \fatsemi f^{\uparrow Q^{A,\bullet}} \overset{!}{=} f^{\uparrow P^{B,\bullet}} \fatsemi t^B \fatsemi f^{\downarrow Q^{\bullet,B}}$$

Both sides must give the same result when applied to arbitrary $p : P^{B,A}$

$$\begin{tikzcd}
 & P^{A,A} \arrow[r, "t^A"] & Q^{A,A} \arrow[dr, "f^{\uparrow Q^{A,\bullet}}"'] & \\
P^{B,A} \arrow[ur, "f^{\downarrow P^{\bullet,A}}"'] \arrow[dr, "f^{\uparrow P^{B,\bullet}}"] & & & Q^{A,B} \\
 & P^{B,B} \arrow[r, "t^B"] & Q^{B,B} \arrow[ur, "f^{\downarrow Q^{\bullet,B}}"] &
\end{tikzcd}$$

This law reduces to natural transformation laws when P and Q are functors or contrafunctors

Example: writing the naturality law for `filter`

`def filter[A]: (A => Boolean) => F[A] => F[A]` for a filterable functor F

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

Rewrite in the form of a dinatural transformation:

$$\text{filt}^A : P^{A,A} \to Q^{A,A} \quad , \quad P^{X,Y} \triangleq (X \to \mathbb{2}) \quad , \quad Q^{X,Y} \triangleq F^X \to F^Y$$

Write the code for the liftings using the specific types of P and Q:

$$(f^{:A\to B})^{\downarrow P^{\bullet,A}} = p^{:B\to\mathbb{2}} \to f \fatsemi p \quad , \qquad f^{\uparrow P^{B,\bullet}} = \text{id} \quad ,$$
$$(f^{:A\to B})^{\downarrow Q^{\bullet,B}} = q^{:F^B\to F^B} \to f^{\uparrow F} \fatsemi q \quad , \qquad f^{\uparrow Q^{A,\bullet}} = q^{:F^A\to F^A} \to q \fatsemi f^{\uparrow F} \quad .$$

Rewrite the naturality law $f^{\downarrow P^{\bullet,A}} \fatsemi \text{filt}^A \fatsemi f^{\uparrow Q^{A,\bullet}} \overset{!}{=} f^{\uparrow P^{B,\bullet}} \fatsemi \text{filt}^B \fatsemi f^{\downarrow Q^{\bullet,B}}$ as

$$(p \to f \fatsemi p) \fatsemi \text{filt}_F \fatsemi (q \to q \fatsemi f^{\uparrow F}) \overset{!}{=} \text{id} \fatsemi \text{filt}_F \fatsemi (q \to f^{\uparrow F} \fatsemi q) \quad .$$

To simplify the form of the naturality law, apply both sides to an arbitrary value $p^{:P^{B,A}} = p^{:B\to\mathbb{2}}$

Evaluate the results and obtain the naturality law of `filter`,

$$\text{filt}_F(f \fatsemi p) \fatsemi f^{\uparrow F} \overset{!}{=} f^{\uparrow F} \fatsemi \text{filt}_F(p)$$

Uniqueness of functor implementations

Statement 1: For any purely functional type constructor F^A covariant in A, there is a unique lawful and purely functional implementation of `fmap` with type signature `fmap[A, B]: (A => B) => F[A] => F[B]`

Statement 2: For any purely functional type constructor F^A contravariant in A, there is a unique lawful and purely functional implementation of `cmap` with type signature `cmap[A, B]: (B => A) => F[A] => F[B]`

- Note: many typeclasses may admit several lawful, purely functional, but non-equivalent implementations of a typeclass instance for the same type constructor `F[A]`. For example, `Filterable`, `Monad`, `Applicative` instances are not always unique. But instances are unique for the functor and contrafunctor type classes.

Proof of Statement 1 (uniqueness of functor instances)

For a given functor F, we can construct the "standard" fmap (denoted by $...^{\uparrow F}$) that is involved in the naturality laws. Suppose that there exists *another* lawful and purely functional implementation $\text{fmap}'(f)$:

$$\text{fmap}' : (A \rightarrow B) \rightarrow F^A \rightarrow F^B \quad , \qquad \text{fmap}'(f^{:A\rightarrow B}) = ???^{:F^A \rightarrow F^B}$$

We need to show that $\text{fmap}' = \text{fmap}$

By parametricity, fmap' has a naturality law with respect to B:

$$\text{fmap}'(f^{:A\rightarrow B} \,{}^{\circ}_{9}\, g^{:B\rightarrow C}) \overset{!}{=} \text{fmap}'(f) \,{}^{\circ}_{9}\, g^{\uparrow F} = \text{fmap}'(f) \,{}^{\circ}_{9}\, \text{fmap}\,(g)$$

This suggests using the composition law for fmap':

$$\text{fmap}'(f \,{}^{\circ}_{9}\, g) = \text{fmap}'(f) \,{}^{\circ}_{9}\, \text{fmap}'(g) \overset{!}{=} \text{fmap}'(f) \,{}^{\circ}_{9}\, \text{fmap}\,(g)$$

Since $f^{:A\rightarrow B}$ is arbitrary, we may choose $A = B$ and $f = \text{id}^{:B\rightarrow B}$ to obtain

$$\underline{\text{fmap}'(\text{id})} \,{}^{\circ}_{9}\, \text{fmap}'(g) = \text{fmap}'(g) \overset{!}{=} \underline{\text{fmap}'(\text{id})} \,{}^{\circ}_{9}\, \text{fmap}\,(g) = \text{fmap}(g)$$

This must hold for arbitrary $g^{:B\rightarrow C}$, which proves that $\text{fmap}'_F = \text{fmap}_F$

Plan for a proof of commutativity law for profunctors

- Main idea: induction on the type expression of a profunctor $P^{X,Y}$
- A purely functional $P^{X,Y}$ must be a combination of `Unit` type ($\mathbb{1}$), parameters X and Y, products $A \times B$, co-products $A + B$, exponentials $A \rightarrow B$, and type recursion (use of P in its definition)
- For each of these cases, we need to show that the commutativity law holds given that it holds for all sub-expressions
  - Base case: show that the law holds for $P^{X,Y} \triangleq \mathbb{1}$ and $P^{X,Y} \triangleq Y$
  - Induction steps: if the law holds for $P^{X,Y}$ and $Q^{X,Y}$, show that it also holds for $P^{X,Y} + Q^{X,Y}$ and $P^{X,Y} \times Q^{X,Y}$ and $P^{Y,X} \rightarrow Q^{X,Y}$
  - Show that the law holds for a recursively defined $P^{X,Y} \triangleq S^{X,Y,P^{X,Y}}$ for a type constructor $S^{X,Y,R}$ contravariant in X, covariant in Y and R
  - We need to use the code of functor and contrafunctor instances for products, co-products, function types, and recursive types
- Example: For $R^{X,Y} \triangleq P^{X,Y} \times Q^{X,Y}$, the liftings to R are given by $f^{\uparrow R} \triangleq p \times q \rightarrow f^{\uparrow P}(p) \times f^{\uparrow Q}(q)$ and $f^{\downarrow R} \triangleq p \times q \rightarrow f^{\downarrow P}(p) \times f^{\downarrow Q}(q)$
  - Write $f^{\downarrow R} \fatsemi g^{\uparrow R}$ explicitly using $f^{\downarrow P}$, $f^{\downarrow Q}$, $g^{\uparrow P}$, and $g^{\uparrow Q}$, and show that $f^{\downarrow R} \fatsemi g^{\uparrow R} = g^{\uparrow R} \fatsemi f^{\downarrow R}$ by assuming that the same law already holds for P and Q

Plan for a proof of parametricity theorem

- Need to prove the naturality law for $t^A : P^{A,A} \rightarrow Q^{A,A}$ written as

$$(f^{:A\rightarrow B})^{\downarrow P^{\bullet,A}} \fcomp t^A \fcomp f^{\uparrow Q^{A,\bullet}} = f^{\uparrow P^{B,\bullet}} \fcomp t^B \fcomp f^{\downarrow Q^{\bullet,B}}$$

- The code of t must be of the form $p \rightarrow \text{expr}$, where "expr" must be built up from the 9 purely functional code constructions
- Main idea: induction on the code of "expr", assuming that the naturality law holds for all sub-expressions
- Example: induction step for code construction 3 ("create function")
  - The code of t is $p \rightarrow z \rightarrow r$ and $Q^{X,Y} \triangleq Z^{Y,X} \rightarrow R^{X,Y}$
  - Inductive assumption is that any $x \rightarrow r$ satisfies the law; let $x = p \times z$
  - Assume that the law holds for $u \triangleq p \times z \rightarrow r$, $u : P^{A,A} \times Z^{A,A} \rightarrow R^{A,A}$
  - Derive the law for $t = p \rightarrow z \rightarrow u(p \times z)$ by a direct calculation
- There are some technical difficulties (dinatural transformations do not generally compose) but these difficulties can be overcome with tricks

Summary

- Purely functional code enables powerful mathematical reasoning:
  - Naturality laws can be used for guaranteed correct refactoring
  - Naturality laws allow us to reduce the number of type parameters
  - In typeclass instances, all naturality laws hold, no need to check
  - Functor, contrafunctor, and profunctor typeclass instances are unique
  - Bifunctors and profunctors obey the commutativity law
- Full details and proofs are in the free upcoming book (Appendix D)
  - Draft of the book: `https://github.com/winitzki/sofp`