Parametricity properties of purely functional code

Sergei Winitzki

Academy by the Bay

2020-02-01

Parametricity properties and naturality laws

Four main results about *purely functional* programs:

1. Parametricity theorem: programs automatically obey naturality laws
   - Example: `headOption: List[A] => Option[A]` satisfies
     `list.map(f).headOption = list.headOption.map(f)`

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

2. Recipe for writing the naturality law, given a function's type signature
   - Dinatural transformation `t: P[A, A] => Q[A, A]` where `P[X, Y]` and `Q[X, Y]` are profunctors (contravariant in `X` and covariant in `Y`) has the naturality law `t(pba.xmap(f, identity)).xmap(identity, g) == t(pba.xmap(identity, g)).xmap(f, identity)`
3. Liftings with respect to different type parameters always commute
   - Example: `IO[E, A]` and `io.map(f).mapError(g) == io.mapError(g).map(f)`
4. Functors and contrafunctors are uniquely derived from types
   - Example: `type F[A] = Either[(A, Int), String => A]` has functor instance `def fmap[A, B](f: A => B): Either[(A, Int), String => A] => Either[(B, Int), String => B] = { case Left(a, n) => Left(f(a), n); case Right(g) => Right(g andThen f) }`

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) }`
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).
9. Use recursive call: `fmap(f)(tail)`. Notation: $\overline{\text{fmap}_{\text{List}}}(f)(t)$

Naturality laws: motivation

A "naturality law" expresses the programmer's intuitions about the properties of **fully parametric** code:

- Code is written once and works in the same way for all types

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

Fully parametric code:

- 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` assignment, writing files, networking)
- No `null`, no `throw`ing of exceptions, no run-time type comparison
- No run-time code loading, no external libraries with unknown code

Naturality laws: equations

Naturality law for a transformation 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)
```

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

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

More examples:

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

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

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

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

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

$$\text{pu}_L \fatsemi (f^{:A\rightarrow B})^{\uparrow L} = f \fatsemi \text{pu}_L$$

Why do we need to know about naturality laws

Typeclasses: type constructors with methods `map`, `filter`, `fold`, `flatMap`
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, composition, naturality

We need to check the laws when implementing a new typeclass instance

- 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

Proving 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)

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

$$\begin{tikzcd}
P^{A,B} \arrow[r, "\text{xmap}_P(f^{:C\rightarrow A},\text{id})"'] \arrow[d, "\text{xmap}_P(\text{id},g^{:B\rightarrow D})"'] & P^{C,B} \arrow[d, "\text{xmap}_P(\text{id},g^{:B\rightarrow D})"] \\
P^{A,D} \arrow[r, "\text{xmap}_P(f,\text{id})"] & P^{C,D}
\end{tikzcd}$$

- Commutativity laws hold for *all* purely functional type constructors
  - It is not necessary to verify the bifunctor and profunctor laws!

Natural transformations and their generalizations

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

- Many standard methods have the form of a natural transformation
- `pure`, `headOption`, `lastOption`, `reverse`, `swap`, `map`, `flatMap`
- If there are several type parameters, use one at a time:
  - For `flatMap`, denote by $\text{flm}_M : \left(A \to M^B\right) \to M^A \to M^B$, fix A
  - $\text{flm}_M : F^B \to G^B$ where $F^B \triangleq A \to M^B$ and $G^B \triangleq M^A \to M^B$
  - The naturality law is then written as the equation

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

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

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

$$(f^{:B\to A})^{\downarrow F} ; t^B = t^A ; (f^{:B\to A})^{\downarrow G}$$

then t is a natural transformation $F \rightsquigarrow G$

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

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

- *All pure functions* have the type signature of a dinatural transformation
- The corresponding naturality law is guaranteed by parametricity
- *All* naturality laws (also for `find`, `fold`) are derived in this way

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 $p : P^{B,A}$

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

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:

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

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

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

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`,

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

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.

Proof of Statement 1 (uniqueness of functor instances)

For a given functor F, we can construct the "standard" fmap_F that is involved in the naturality laws. Suppose that there exists *another* lawful and purely functional implementation $\text{fmap}'_F(f)$. We need to show that $\text{fmap}'_F = \text{fmap}_F$.

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

Now, fmap'_F has a naturality law with respect to B:

$$\text{fmap}'_F(f^{:A\to B} \fatsemi g^{:B\to C}) \overset{!}{=} \text{fmap}'_F(f) \fatsemi g^{\uparrow F} \quad .$$

Use the composition law for fmap'_F:

$$\text{fmap}'_F(f \fatsemi g) = \text{fmap}'_F(f) \fatsemi \text{fmap}'_F(g) \overset{!}{=} \text{fmap}'_F(f) \fatsemi g^{\uparrow F} \quad .$$

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

$$\underline{\text{fmap}'_F(\text{id})} \fatsemi \text{fmap}'_F(g) = \text{fmap}'_F(g) \overset{!}{=} \underline{\text{fmap}'_F(\text{id})} \fatsemi g^{\uparrow F} = g^{\uparrow F} = \text{fmap}_F(g) \quad .$$

This must hold for arbitrary $g^{:B\to C}$, 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 $P^{Y,X} \rightarrow Q^{X,Y}$; also 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, exponentials, and recursive types.
    - ★ Example: Define $R^{X,Y} \triangleq P^{X,Y} \times Q^{X,Y}$, then the lifting to R is given by $f^{\uparrow R} \triangleq p \times q \rightarrow f^{\uparrow P}(p) \times f^{\uparrow Q}(q)$

Plan for a proof of parametricity theorem

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

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

- The code of t must be of the form $p \to \text{expr}$, 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 $p \to z \to r$ and $Q^{X,Y} \triangleq Z^{Y,X} \to R^{X,Y}$
  - Inductive assumption is that any $x \to r$ satisfies the law; let $x = p \times z$
  - Assume that the law holds for $u \triangleq p \times z \to r$, $u : P^{A,A} \times Z^{A,A} \to R^{A,A}$
  - Derive the law for $t = p \to z \to u(p \times z)$ 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:
  - Any type constructor `F[A]` has the form of a diagonal profunctor `P[A, A]`
  - Functor, contrafunctor, and profunctor instances are unique
  - Any purely functional code obeys a naturality law
  - Bifunctors and profunctors obey a commutativity law
- Full details and proofs are in the upcoming book (Appendix D)
  - Source (LaTeX) for the book: https://github.com/winitzki/sofp