Explaining “Theorems for Free” and parametricity

A tutorial, with code examples in Scala

Sergei Winitzki

Academy By the Bay

2020-10-17

Parametricity: a theory about certain code refactorings

Expected properties of code that manipulates collections:

- 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 "work the same way for all types"
  - ... and satisfy the "naturality laws"

Refactoring code: equations

Writing the previous examples as equations:

- The refactoring involving `toList`:

  ```
  def toList[A]: Stream[A] => List[A]
  ```

  For any function `f: A => B`:

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

- The refactoring involving `filter`:

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

  For any function `f: A => B` and predicate `p: B => Boolean`:

  ```
  _.filter(f andThen p).map(f) == _.map(f).filter(p)
  ```

- For any `f: A => B`, applying `t : F[A] => G[A]` *before* `_.map(f)` equals applying `t: F[B] => G[B]` *after* `_.map(f)`
  - This is called a **naturality law**
  - We expect it to hold if the code works the same way for all types

Naturality laws: examples

Naturality law for a function `t[A]: F[A] => G[A]` is an equation involving an arbitrary function `f: A => B` that permutes the order of `t` and of `_.map(f)`

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

- Lifting `f` before `t` is equal to lifting `f` after `t`
- ... need to use different type parameters for `t[A]`
- Intuition: `t` rearranges data in a collection, "not looking" at values

Further examples:

- Reverse a list: `reverse[A]: List[A] => List[A]`

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

- The `pure` method: `pure[A]: A => L[A]` or `Id[A] => L[A]`

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

- Get length: `length[A]: List[A] => Int` or `List[A] => Const[Int, A]`

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

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
If naturality holds for `flatMap`, `filter`, and `pure` then:

- The methods `flatMap` and `flatten: F[F[A]] => F[A]` are equivalent
- The methods `filter` and `deflate: F[Option[A]] => F[A]` are equivalent
- The methods `pure: A => F[A]` and `unit: F[Unit]` are equivalent
  - Can simplify the definitions of some typeclasses

Naturality laws and "theorems for free"

- "Theorems for free" give naturality laws in two flavors:
  - Naturality laws (most often seen in practice)
  - Dinaturality laws (rarely seen in practice)
- The **parametricity theorem** says:
  - Any **fully parametric** code `t: P[A] => Q[A]` satisfies a (di)naturality law
  - There is a recipe for writing that law
  - One independent law is obtained per type parameter
- Usually, typeclass instances are written in fully parametric code
  - Then it is not necessary to verify the naturality laws
  - Can simplify some typeclass definitions
  - Naturality laws save us time and simplifies code

What did "theorems for free" ever do for us programmers?

- "Theorems for free" guarantee naturality laws for fully parametric code
- To use "theorems for free" in practice, programmers need to:
  - Recognize and write fully parametric code
  - Be able to write the refactoring that follows from naturality laws
  - Be able to implement `_.map` for any covariant functor (as well as `_.contramap` for any contravariant functor)
  - Recognize that the refactoring is guaranteed by parametricity
  - Recognize simplifications in typeclasses

Fully parametric code: example

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

- Example of a fully parametric function:

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

- The code does not use explicit types

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

Example of code that is *not* fully parametric:

- An implementation of `headOpt` that has special code for `Int` type

```
def headOptBad[A]: List[A] => Option[A] = {
  case Nil                        => None
  case (head: Int) :: tail        => Some((head + 100).asInstanceOf[A])
  case head :: tail               => Some(head)
}
```

- The code uses explicit run-time type detection
  - But the code is still purely functional and referentially transparent
  - "Full parametricity" is a stronger restriction on code

The function `headOptBad` fails the naturality law:

```
scala> headOptBad(List(1, 2, 3).map(x => s"value = $x"))
res0: Option[String] = Some(value = 1)

scala> headOptBad(List(1, 2, 3)).map(x => s"value = $x")
res1: Option[String] = Some(value = 101)
```

Full parametricity: The price of "free theorems"

"Free theorems" only apply to **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`, mutating values, writing files, networking, starting or stopping new threads, GUI events, etc.)
- No `null`, no `throw`ing of exceptions, no run-time type comparison
- No run-time code loading, no external libraries with unknown code

"Fully parametric" is a stronger restriction than "purely functional" (referentially transparent)

Purely functional code is fully parametric if restricted to using only `Unit` type or type parameters

- No hard-coded values of specific types, and no run-time type detection

Fully parametric 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
}       // This code has used each of the nine allowed constructions.
```

1. Use `Unit` value (or a "named `Unit`"), e.g. `()`, `Nil`, or `None`
2. Use bound variable (a given argument of the function)
3. Create function: `{ x => expr(x) }`
4. Use function: `f(x)`
5. Create tuple: `(a, b)`
6. Use tuple: `p._1`
7. Create disjunctive value: `Left[A, B](x)`
8. Use disjunctive value: `{ case ... => ... }` (pattern-matching)
9. Use recursive calls: e.g., `fmap(f)(tail)` within the code of `fmap`

Approaches to using and proving the parametricity theorem

Using the parametricity theorem à la Wadler 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
  - No need to use relations
  - For any type signature, can quickly write the naturality law

Dinatural transformations and profunctors

Some methods do *not* have the type signature of the form `F[A] => 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]` (contravariant in `X`, covariant in `Y`) but set equal type parameters: `P[A, A]`

A **dinatural transformation** is a function `t[A]: P[A, A] => Q[A, A]` where `P[X, Y]` and `Q[X, Y]` are some profunctors and `t` satisfies the naturality law

- *All pure functions* have the type signature of a dinatural transformation

The naturality law for dinatural transformations

Given `t[A]: P[A, A] => Q[A, A]` where `P[X, Y]` and `Q[X, Y]` are profunctors
The naturality law requires that for any function `f: A => B`,

```
p.contramapPX(f).pipe(t).mapQY(f) == p.mapPY(f).pipe(t).contramapQX(f)
```

Both sides must give the same result when applied to arbitrary `p: P[B, A]`

- *All* naturality laws (also for `find`, `fold`) are derived in this way
- The code for `map` and `contramap` must be lawful and fully parametric

$$\begin{tikzcd}
& \texttt{P[A, A]} \arrow[rr, "\texttt{t[A]}"] & & \texttt{Q[A, A]} \arrow[dr, "\texttt{_.mapQY(f)}"] & \\
\texttt{P[B, A]} \arrow[ur, "\texttt{_.contramapPX(f)}"] \arrow[dr, "\texttt{_.mapPY(f)}"'] & & & & \texttt{Q[A, B]} \\
& \texttt{P[B, B]} \arrow[rr, "\texttt{t[B]}"] & & \texttt{Q[B, B]} \arrow[ur, "\texttt{_.contramapQX(f)}"'] &
\end{tikzcd}$$

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

Example: deriving the naturality law for `filter`

Use a curried version of `_.filter` for convenience:

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

Rewrite as a dinatural transformation, `filter[A]: P[A, A] => Q[A, A]` with `type P[X, Y] = X => Boolean` and `type Q[X, Y] = F[X] => F[Y]`

Write the code for `map` and `contramap` using the specific types of `P` and `Q`:

```
p.contramapPX(f) == f andThen p

p.mapPY(f) == p

q.contramapQX(f) == _.map(f) andThen q

q.mapQY(f) = q andThen _.map(f)
```

Now write the dinaturality law and simplify: `(f andThen p).pipe(filt) andThen _.map(f)) == _.map(f) andThen p.pipe(filt)`

Rewriting in terms of `_.filter`, we obtain the naturality law of `filter`:

`_.filter(f andThen p).map(f) = _.map(f).filter(p)`

Other parametricity properties

- Bifunctor `map` commutes w.r.t. different type parameters
  For any value `b: B[X, Y]`, and any functions `f: X => P`, `g: Y => Q`, the commutativity law is: `b.mapX(f).mapY(g) == b.mapY(g).mapX(f)`

$$\begin{tikzcd}
\mathrm{B[X, Y]} \arrow[r, "_\mathrm{.mapX(f)}"] \arrow[d, "_\mathrm{.mapY(g)}"'] & \mathrm{B[P, Y]} \arrow[d, "_\mathrm{.mapY(g)}"] \\
\mathrm{B[X, Q]} \arrow[r, "_\mathrm{.mapX(f)}"] & \mathrm{B[P, Q]}
\end{tikzcd}$$

- A given functor's lawful and fully parametric method `map` is unique
  - Note: many typeclasses may admit several lawful, fully parametric, 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.
- Analogous results for `contramap`, contrafunctors, and profunctors

Summary

- Fully parametric 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 Appendix D of the upcoming book
  - Draft of the book: `https://github.com/winitzki/sofp`