

# Relational parametricity and "theorems for free"

A tutorial, with example code in Scala

Sergei Winitzki

Academy by the Bay

2021-09-04

# Motivation for parametricity. “Theorems for free”

**Parametricity:** all fully parametric functions satisfy their naturality laws

- Naturality law: code must work in the same way for all types

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

- **“Fully parametric”** code: use only type parameters, no JVM reflection
  - ▶ Naturality laws are “for free” only if all code is fully parametric
- Naturality law for `headOption`: for any `x: List[A]` and `f: A => B`,  
`headOption(x).map(f) == headOption(x.map(f))`

Parametricity theorems work only if the code is “fully parametric”

Parametricity theorems apply only to a subset of a programming language

- Usually, it is a certain flavor of typed lambda calculus

# Examples of code that fails parametricity

Explicit matching on type parameters using JVM reflection:

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

Using typeclasses: define typeclass `NotInt[A]` returning `true` unless `A = Int`

```
def badHeadOpt[A]: List[A] => Option[A] = {  
  case h :: tail if implicitly[NotInt[A]]() => Some(h)  
  case _ => None  
}
```

Failure of naturality law:

```
scala> badHeadOpt(List(10, 20, 30).map(x => s"x = $x"))  
res0: Option[String] = Some(x = 10)
```

```
scala> badHeadOpt(List(10, 20, 30)).map(x => s"x = $x")  
res1: Option[String] = None
```

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 uses each of the nine allowed constructions.
```

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

# Why we need relational parametricity

“Relational parametricity” is a method for proving parametricity theorems

- Main papers: Reynolds (1983) and Wadler “Theorems for free” (1989)
  - ▶ Those papers are a bit outdated and also hard to understand
- There are very few pedagogical tutorials on relational parametricity
  - ▶ “On a relation of functions” by R. Backhouse (1990)
  - ▶ “The algebra of programming” by R. Bird and O. de Moor (1997)

This tutorial does *not* follow any of the above but derives equivalent results

- Alternative approach: prove “dinaturality” (de Lataillade, Voigtländer)
  - ▶ Dinaturality is a consequence of relational parametricity
  - ▶ In practice, dinaturality laws are sufficient in most cases
  - ▶ But some proofs still need full relational parametricity

# Motivating relational parametricity. I. Naturality laws

Naturality law: applying  $\tau[A] : F[A] \Rightarrow G[A]$  before  $\_.\text{map}(f)$  equals applying  $\tau[B] : F[B] \Rightarrow G[B]$  after  $\_.\text{map}(f)$  for any function  $f : A \Rightarrow B$

Naturality laws need *lifting*  $f : A \Rightarrow B$  to  $F[A] \Rightarrow F[B]$  and  $G[A] \Rightarrow G[B]$

$$\begin{array}{ccc} F[A] & \xrightarrow{\tau[A]} & G[A] \\ \downarrow \text{\_\text{map}(f) for F} & & \downarrow \text{\_\text{map}(f) for G} \\ F[B] & \xrightarrow{\tau[B]} & G[B] \end{array}$$

- Proof of the naturality law requires induction on the code of  $\tau[A]$ 
  - ▶ This code is built up by combining the 9 code constructions
  - ▶ This code may include sub-expressions of types not covariant in  $A$

# Motivating relational parametricity. II. The difficulty

Cannot lift  $f: A \Rightarrow B$  to  $F[A] \Rightarrow F[B]$  when  $F[_]$  is not covariant!

- For covariant  $F[_]$  we lift  $f: A \Rightarrow B$  to  $\text{fmap}(f): F[A] \Rightarrow F[B]$
- For contravariant  $F[_]$  we lift  $f: B \Rightarrow A$  to  $\text{cmap}(f): F[A] \Rightarrow F[B]$

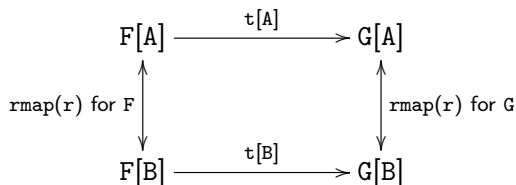
In general,  $F[_]$  will be neither covariant nor contravariant

- Example: `foldLeft` with respect to type parameter  $A$   
`def foldLeft[T, A]: List[T] => (T => A => A) => A => A`
- This is *not* of the form  $F[A] \Rightarrow G[A]$  with covariant  $F[_]$  and  $G[_]$ 
  - ▶ Some occurrences of  $A$  are in covariant positions but other occurrences are in contravariant positions, all mixed up

# Motivating relational parametricity. III. Liftings

The solution involves three nontrivial steps:

- 1 Replace functions  $f: A \Rightarrow B$  by relations  $r: A \Leftrightarrow B$ 
  - Instead of  $b == f(a)$ , we will write:  $(a, b) \text{ in } r$
- 2 Turns out, we can lift  $r: A \Leftrightarrow B$  to  $\text{rmap}(r): F[A] \Leftrightarrow F[B]$
- 3 Reformulate the naturality law of  $t$  via relations: for any  $r: A \Leftrightarrow B$ ,



To read the diagram: the starting values are on the left

For any  $r: A \Leftrightarrow B$ , for any  $fa: F[A]$  and  $fb: F[B]$  such that

$(fa, fb) \text{ in } \text{rmap\_F}(r)$ , we require  $(t(fa), t(fb)) \text{ in } \text{rmap\_G}(r)$



# Definition and examples of relations

In the terminology of relational databases:

- A relation  $r: A \Leftrightarrow B$  is a table with 2 columns ( $A$  and  $B$ )
- Each row  $(a: A, b: B)$  means that the value  $a$  is related to  $b$

Mathematically speaking: a relation  $r: A \Leftrightarrow B$  is a subset  $r \subset A \times B$

- We write  $(a, b)$  in  $r$  to mean  $a \times b \in r$  where  $a \in A$  and  $b \in B$

Relations can be many-to-many while functions  $A \Rightarrow B$  are many-to-one  
A function  $f: A \Rightarrow B$  can be also viewed as a relation  $\text{rel}(f): A \Leftrightarrow B$

- Two values  $a: A, b: B$  are in  $\text{rel}(f)$  if  $b == f(a)$
- $\text{rel}(\text{identity}: A \Rightarrow A)$  defines an **identity** relation  $\text{id}: A \Leftrightarrow A$

Example of a relation that can be many-to-many:

Given two functions  $f: A \Rightarrow C, g: B \Rightarrow C$ , define a “pullback” relation  $\text{pullback}(f, g): A \Leftrightarrow B$  as:  $(a: A, b: B)$  in  $r$  means  $f(a) == g(b)$

- The pullback relation is *not* equivalent to a function  $A \Rightarrow B$  or  $B \Rightarrow A$

# Proof of relational parametricity. I. Relation combinators

## Relation combinators:

- For any relation  $r: A \Leftrightarrow B$ , the **inverse** relation is  $\text{inv}(r): B \Leftrightarrow A$ 
  - ▶ The inverse operation is its own inverse:  $\text{inv}(\text{inv}(r)) == r$
- For any relations  $r: A \Leftrightarrow B$  and  $s: A \Leftrightarrow B$ , get the union ( $r \text{ or } s$ ) and the intersection ( $r \text{ and } s$ ):  
 $(a, b) \text{ in } (r \text{ and } s)$  means  $(a, b) \text{ in } r$  and  $(a, b) \text{ in } s$   
 $(a, b) \text{ in } (r \text{ or } s)$  means  $(a, b) \text{ in } r$  or  $(a, b) \text{ in } s$
- For any relations  $r: A \Leftrightarrow B$  and  $s: B \Leftrightarrow C$  define the **composition** ( $r \text{ compose } s$ ) as a relation  $u: A \Leftrightarrow C$  by  $(a: A, c: C) \text{ in } u$  when there exists  $b: B$  such that  $(a, b) \text{ in } r$  and  $(b, c) \text{ in } s$ 
  - ▶ Composition corresponds to “join” in relational databases
  - ▶ Directionality law:  $\text{inv}(r \text{ compose } s) == \text{inv}(s) \text{ compose } \text{inv}(r)$
  - ▶ Associativity and identity laws with respect to  $\text{id}: A \Leftrightarrow A$
  - ▶ Preserves composition of functions: for  $f: A \Rightarrow B$  and  $g: B \Rightarrow C$ ,  
 $\text{rel}(f \text{ andThen } g) == \text{rel}(f) \text{ compose } \text{rel}(g)$
- The “pullback relation” can be expressed through composition:  
 $\text{pullback}(f, g) == \text{rel}(f) \text{ compose } \text{inv}(\text{rel}(g))$

# Pullback relation expressed through composition of relations

For any  $f: A \Rightarrow C$ ,  $g: B \Rightarrow C$ ,  $a: A$ ,  $b: B$ , to prove:

- $(a, b)$  in  $\text{pullback}(f, g)$  is equivalent to:  
 $(a, b)$  in  $\text{rel}(f) \text{ compose inv}(\text{rel}(g))$

$$A \xleftarrow{\text{rel}(f)} C \xleftarrow{\text{inv}(\text{rel}(g))} B$$

- The first condition is equivalent to:  $f(a) == g(b)$
- The second condition is equivalent to: there exists  $c: C$  such that:  
 $(a, c)$  in  $\text{rel}(f)$  and  $(c, b)$  in  $\text{inv}(\text{rel}(g))$
- This is equivalent to:  $z$  is such that  $c == f(a)$  and  $c == g(b)$
- This is equivalent to the first condition

## Proof of relational parametricity. II. Definition of `rmap`

For a type constructor `F` and `r: A <=> B`, need `rmap(r): F[A] <=> F[B]`

Define `rmap` for `F[A]` by induction over the *type expression* of `F[A]`

There are seven possibilities (assuming that the code is fully parametric):

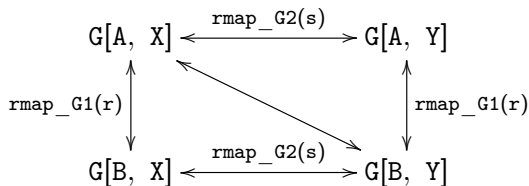
- ① `F[A] = Unit` or another fixed type (say, `T`) not related to `A`
- ② The identity functor: `F[A] = A`
- ③ Product type: `F[A] = (G[A], H[A])`
- ④ Co-product type: `F[A] = Either[G[A], H[A]]`
- ⑤ Function type: `F[A] = G[A] => H[A]`
- ⑥ Recursive type: `F[A] = G[A, F[A]]`
- ⑦ Universally quantified term: `F[A] = [Z] => G[A, Z]`

Define `rmap` similarly to how a functor's `fmap` is defined in these cases

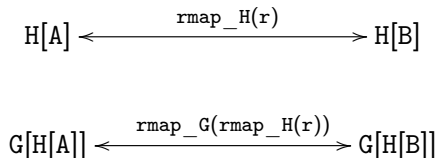
- The inductive assumption is that liftings to `G` and `H` are already defined
- For `G[A, Z]`, need to use two liftings (`rmap_G1` and `rmap_G2`)
- Liftings with respect to different type parameters will commute!
- For `F[A] = G[H[A]]` we expect `rmap_F(r) == rmap_G(rmap_H(r))`

## Some diagrams for clarification

The commutativity theorem for relational liftings: For any type constructor  $G[A, X]$  and any two relations  $r: A \Leftarrow B$  and  $s: X \Leftarrow Y$ :



Relational lifting for a composition of type constructors,  $F[A] = G[H[A]]$ :



# Proof of relational parametricity. II. Definition of `rmap`

Need to define `rmap(r): F[A] <=> F[B]` in these 7 cases:

- ① `F[A] = T` (a fixed type): define `rmap(r) = id: T <=> T`
- ② The identity functor, `F[A] = A`: define `rmap(r) = r: A <=> B`
- ③ When `F[A] = (G[A], H[A])`: define `((g1,h1), (g2,h2)) in rmap(r)` to mean `(g1, g2) in rmap_G(r)` and `(h1, h2) in rmap_H(r)`
- ④ When `F[A] = Either[G[A], H[A]]`: either `(Left(g1), Left(g2)) in rmap(r)` when `(g1, g2) in rmap_G(r)` or `(Right(h1), Right(h2)) in rmap(r)` when `(h1, h2) in rmap_H(r)`
- ⑤ When `F[A] = G[A] => H[A]`: define `(f1, f2) in rmap(r)` to mean `(f1(g1), f2(g2)) in rmap_H(r)` for any `g1: G[A]` and `g2: G[B]` such that `(g1, g2) in rmap_G(r)`
- ⑥ When `F[A] = G[A, F[A]]`: define `rmap(r) = rmap_G1(r) compose rmap_G2(rmap(r))` – the second `rmap(r)` is a recursive call
- ⑦ When `F[A] = [Z] => G[A, Z]`: define `(f1, f2) in rmap(r)` to mean: for any types `Z1` and `Z2`, and for any relation `s: Z1 <=> Z2`, we require `(f1[A][Z1], f2[B][Z2]) in (rmap_G1(r) compose rmap_G2(s))`

# Proof of relational parametricity. III. Examples of using rmap

Use `rmap` to lift a relation `r` to a type constructor

Two main examples of relations generated by functions:

`rel(f)` and `pullback(f, g)`

Three main examples of type constructors (`F[A]`, `G[A]`, `H[A]`):

- If `F[A]` is covariant then:

`rmap(rel(f)) == rel(fmap(f))`

`rmap(pullback(f, g)) == pullback(fmap(f), fmap(g))`

- If `G[A] = A => A` then `(fa, fb) in rmap(rel(f))` means:

when `(a, b) in rel(f)` then `(fa(a), fb(b)) in rel(f)`

or: `f(fa(a)) == fb(f(a))` or: `fa andThen f == f andThen fb`

This relation has the form of a pullback

- If `H[A] = (A => A) => A` then `(fa, fb) in rmap_H(rel(f))` means:

when `(p, q) in rmap_G(rel(f))` then `(fa(p), fb(q)) in rel(f)`

equivalently: if `p andThen f == f andThen q` then `f(fa(p)) == fb(q)`

This is *not* a pullback relation: cannot express `p` through `q`

It is hard to use relations that do not have the form of a pullback

# Proof of relational parametricity. IV. Formulation

Instead of proving relational properties for  $t[A] : P[A] \Rightarrow Q[A]$ , use the function type and the quantified type constructions and get:

- Any fully parametric  $t[A] : P[A]$  satisfies for any  $r : A \Leftrightarrow B$  the relation  $(t[A], t[B]) \text{ in } \text{rmap\_P}(r)$
- Any fully parametric  $t : P[]$  satisfies  $(t, t) \text{ in } \text{rmap\_P}(\text{id})$

It is more convenient to prove a parametricity theorem with a free variable:

- Any fully parametric expression  $t[A](z) : P[A]$  with  $z : Q[A]$  satisfies, for any relation  $r : A \Leftrightarrow B$  and for any  $z1 : Q[A]$ ,  $z2 : Q[B]$ , the law: if  $(z1, z2) \text{ in } \text{rmap\_Q}(r)$  then  $(t[A](z1), t[B](z2)) \text{ in } \text{rmap\_P}(r)$

This applies to expressions containing one free variable ( $z$ )

- Any number of free variables can be grouped into a tuple



# From relational parametricity to naturality laws

Example:  $t[A] = \{ a: A \Rightarrow A \}$  of type  $P[A] = A \Rightarrow A$

Parametricity theorem says:

- For any types  $A$  and  $B$ , and for any relation  $r: A \Leftrightarrow B$ , we have:  
 $(t[A], t[B]) \text{ in } \text{rmap\_P}(r)$  where  $\text{rmap\_P}(r): (A \Rightarrow A) \Leftrightarrow (B \Rightarrow B)$
- $(p, q) \text{ in } \text{rmap\_P}(r)$  means: for any  $a: A, b: B$ , if  $(a, b) \text{ in } r$  then  $p(a), q(b) \text{ in } r$
- So,  $(t[A], t[B]) \text{ in } \text{rmap\_P}(r)$  means: for any  $a: A, b: B$ , if  $(a, b) \text{ in } r$  then  $(t(a), t(b)) \text{ in } r$

Trick: choose  $r = \text{rel}(f)$  where  $f: A \Rightarrow B$  is an arbitrary function

- We get: for any  $a: A, b: B$ , if  $f(a) == b$  then  $f(t(a)) == t(b)$
- Equivalently:  $f(t(a)) == t(f(a))$ , i.e.,  $t$  commutes with all functions
- One can then prove that  $t$  must be an identity function
  - ▶ Choose  $f = \{ \_ : A \Rightarrow b \}$  with a fixed constant  $b: B$

# Proof of relational parametricity. V. Outline

The theorem says that  $t[A](z)$  satisfies its relational parametricity law

Proof goes by induction on the structure of the code of  $t[A](z)$

At the top level,  $t[A](z)$  must have one of the 9 code constructions

Each construction decomposes the code of  $t[A](z)$  into sub-expressions

The inductive assumption is that the theorem holds for all sub-expressions (including the bound variable  $z$ )

# Proof of relational parametricity. VI. Examples

We will show how to prove the first 4 constructions

Constant type: If  $t[A](z) = c$  where  $c$  is a fixed value of a fixed type  $C$ :

- We have  $\text{rmap\_P}(r) == \text{id}$  while  $(c, c) \text{ in id}$  holds

Use argument: If  $t[A](z) = z$  where  $z$  is a value of type  $Q[A]$ :

- If  $(z1, z2) \text{ in rmap\_Q}(r)$  then  $(t(z1), t(z2)) \text{ in rmap\_Q}(r)$

Create function: If  $t(z) = h \Rightarrow s(z, h)$  where  $h: H[A]$  and  $s(z, h): S[A]$ :

- If  $(z1, z2) \text{ in rmap\_Q}(r)$  and  $(h1, h2) \text{ in rmap\_H}(r)$  then  $(s(z1, h1), s(z2, h2)) \text{ in rmap\_S}(r)$

Use function: If  $t(z) = g(z)(h(z))$  where  $g(z): H[A] \Rightarrow P[A]$  and  $h(z): H[A]$  are sub-expressions:

- If  $(z1, z2) \text{ in rmap\_Q}(r)$  then inductive assumption says:  
 $(h(z1), h(z2)) \text{ in rmap\_H}(r)$
- If  $(h1, h2) \text{ in rmap\_H}(r)$  then inductive assumption says:  
 $(g(h1), g(h2)) \text{ in rmap\_P}(r)$

# Summary

- Relational parametricity is a powerful technique
- It has been generalized to many different settings
  - ▶ Gradual typing, higher-kinded types, dependent types, etc.
- Relational parametricity has a steep learning curve
  - ▶ Cannot directly write code that manipulates relations
  - ▶ All calculations need to be done symbolically or with proof assistants
- The result may be a relation that is difficult to interpret as code
- A couple of results in FP do require the relational naturality law
- More details in the free book — <https://github.com/winitzki/sofp>

