

# Proving "theorems for free" via relational parametricity

## A tutorial, with example code in Scala

Sergei Winitzki

Academy by the Bay

2023-01-01

# Outline of the tutorial

- Motivation: practical applications of the parametricity theorem
- What is “fully parametric code”
- Naturality laws and their uses
  - ▶ Example: Covariant and contravariant Yoneda identities
- A complete proof of “theorems for free” in 6 steps
  - ▶ Step 1: Deriving `fmap` and `cmap` methods from types
  - ▶ Step 2: Motivation for the relational approach to naturality laws
  - ▶ Step 3: Definition and examples of relations
  - ▶ Step 4: Definition and properties of the relational lifting (`rmap`)
  - ▶ Step 5: Proof of the relational naturality law
  - ▶ Step 6: Deriving the wedge law from the relational naturality law
- Advanced applications of the parametricity theorem: beyond Yoneda
  - ▶ Church encodings of recursive types
  - ▶ Simplifying universally quantified types where Yoneda fails

# Applications of parametricity. “Theorems for free”

**Parametricity theorem:** any fully parametric function obeys a certain law

Some applications:

Naturality laws for code that works in the same way for all types

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

- Naturality law for `headOption`: for all `x: List[A]` and `f: A => B`,  
`x.headOption.map(f) == x.map(f).headOption`

Uniqueness properties for fully parametric functions

- The `map` and `contramap` methods uniquely follow from types
- There is only one function `f` with type signature `f[A]: A => (A, A)`

Type equivalence for universally quantified types

- The type of functions `pure[A]: A => F[A]` is equivalent to `F[Unit]`
  - ▶ In Scala 3, this type is written as `[A] => A => F[A]`
- The type `[A] => (A, (R, A) => A) => A` is equivalent to `List[R]`
- The type `[A] => ((A => R) => A) => A` is equivalent to `R`

# Requirements for parametricity. Fully parametric code

Parametricity theorem works only if the code is “fully parametric”

- “**Fully parametric**” code: use only type parameters and `Unit`, no run-time type reflection, no external libraries or built-in types
  - ▶ For instance, no `IO`-like monads
- “Fully parametric” is a stronger restriction than “purely functional”

Parametricity theorem applies only to a subset of a programming language

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

# Examples of code that is not fully parametric

Explicit matching on type parameters using type reflection:

```
def badHeadOpt[A]: List[A] => Option[A] = {  
  case Nil => None  
  case (head: Int) :: tail => None // Run-time type match!  
  case head :: tail => Some(head)  
}
```

Using typeclasses: define a typeclass `NotInt[A]` with the method `notInt[A]` that returns `true` unless `A = Int`

```
def badHeadOpt[A: NotInt]: List[A] => Option[A] = {  
  case h :: tail if 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`

# Naturality laws require map

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

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

- Example:  $F = \text{List}$ ,  $G = \text{Option}$ ,  $t = \text{headOption}$

The naturality law of `headOption`: for all  $x: \text{List}[A]$  and  $f: A \Rightarrow B$ ,  
 $x.\text{headOption}.\text{map}(f) = x.\text{map}(f).\text{headOption}$

Naturality laws are formulated using  $\_.\text{map}$  for  $F$  and  $G$

What is the code of `map` for a given  $F[_]$ ?

- Equivalently, the code of  $\text{fmap}[A, B]: (A \Rightarrow B) \Rightarrow F[A] \Rightarrow F[B]$

# Using naturality laws: the Yoneda identities

For covariant  $F[A]$ , the type  $F[R]$  is equivalent to the type of functions

$p[A]: (R \Rightarrow A) \Rightarrow F[A]$  satisfying the naturality law:

$p[A](k).map(f) == p[B](k \text{ andThen } f)$  for all  $f: A \Rightarrow B$

Isomorphism maps:

$inY[A]: F[R] \Rightarrow (R \Rightarrow A) \Rightarrow F[A] = fr \Rightarrow k \Rightarrow fr.map[A](k)$

$outY: ([A] \Rightarrow (R \Rightarrow A) \Rightarrow F[A]) \Rightarrow F[R] = p \Rightarrow p[R](identity[R])$

Proofs of isomorphism:

$outY(inY(fr)) == outY(k \Rightarrow fr.map(k)) == fr.map(identity) == fr$

The other direction:

$inY(outY(p)) == k \Rightarrow outY(p).map(k) == k \Rightarrow p(identity).map(k)$

Use the naturality law:  $p(identity).map(k) == p(identity \text{ andThen } k)$

So:  $inY(outY(p)) == k \Rightarrow p(k) == p$

- The naturality law and the code of `inY` must use *the same* `_.map`

For contravariant  $G[A]$ , the type  $G[R]$  is equivalent to the type of functions

$q[A]: (A \Rightarrow R) \Rightarrow G[A]$  satisfying the appropriate naturality law



# Example applications of the Yoneda identities

Many types can be converted to the form  $[A] \Rightarrow (R \Rightarrow A) \Rightarrow F[A]$  with a covariant  $F$  or to  $[A] \Rightarrow (A \Rightarrow R) \Rightarrow G[A]$  with a contravariant  $G$

Some examples (assume covariant  $F[_]$  and contravariant  $G[_]$ ):

- $[A] \Rightarrow A$  is equivalent to `Nothing`
- $[A] \Rightarrow F[A]$  is equivalent to `F[Nothing]`
- $[A] \Rightarrow G[A]$  is equivalent to `G[Unit]`
- $[A] \Rightarrow A \Rightarrow A$  is equivalent to `Unit`
- $[A] \Rightarrow A \Rightarrow F[A]$  is equivalent to `F[Unit]`
- $[A] \Rightarrow (A, A) \Rightarrow A$  is equivalent to `Boolean`
- $[A] \Rightarrow (A, A) \Rightarrow F[A]$  is equivalent to `F[Boolean]`
- $[A] \Rightarrow (P \Rightarrow A) \Rightarrow Q \Rightarrow A$  is equivalent to `Q => P`
- $[A] \Rightarrow (A \Rightarrow P) \Rightarrow A \Rightarrow Q$  is equivalent to `P => Q`
- $[A] \Rightarrow F[A] \Rightarrow (A \Rightarrow P) \Rightarrow Q$  is equivalent to `F[P] => Q`
- `flatMap` is equivalent to `flatten`: (use Yoneda w.r.t.  $A$ )  

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

# Step 1. Fully parametric type constructors

What is the `fmap` function for a given type constructor `F[_]`?

- If the code of `t[A]: F[A] => G[A]` is fully parametric, then there are only a few ways to build the type constructors `F[_]` and `G[_]`
- Such “fully parametric” type constructors `F[_]` are built as:
  - 1 `F[A] = Unit` or `F[A] = B` where `B` is another type parameter
  - 2 `F[A] = A`
  - 3 `F[A] = (G[A], H[A])` — product types
  - 4 `F[A] = Either[G[A], H[A]]` — co-product types
  - 5 `F[A] = G[A] => H[A]` — function types
  - 6 `F[A] = G[F[A], A]` — recursive types
  - 7 `F[A] = [X] => G[A, X]` — universally quantified types

The recursive type construction (`Fix`) can be defined as:

```
case class Fix[G[_], A](unfix: G[Fix[G[_], A], A], A)
F[A] = Fix[G, A] satisfies the type equation F[A] = G[F[A], A]
```

# Step 1. Deriving fmap from types

- What is the `fmap` function for a covariant type constructor `F[_]`?

`fmap_F[A, B]: (A => B) => F[A] => F[B]`

- 1 If `F[A] = Unit` or `F[A] = B` then `fmap_F(f) = identity`
- 2 If `F[A] = A` then `fmap_F(f) = f`
- 3 If `F[A] = (G[A], H[A])` then we need `fmap_G` and `fmap_H`  
`fmap_F(f) = { case (ga, ha) => (fmap_G(f)(ga),  
fmap_H(f)(ha)) }`
- 4 If `F[A] = Either[G[A], H[A]]` then `fmap_F(f) = {  
case Left(ga) => Left(fmap_G(f)(ga))  
case Right(ha) => Right(fmap_H(f)(ha))  
}`
- 5 If `F[A] = G[A] => H[A]` then we need `cmap_G` and `fmap_H`  
`cmap_G[A, B]: (A => B) => G[B] => G[A]`  
We define `fmap_F(f)(p: G[A] => H[A]) =  
cmap_G(f) andThen p andThen fmap_H(f)`
- 6 If `F[A] = G[F[A], A]` then we need `fmap_G1` and `fmap_G2`  
`fmap_F(f) = fmap_G1(fmap_F(f)) andThen fmap_G2(f)`
- 7 If `F[A] = [X] => G[A, X]` then we need `fmap_G1`  
`fmap_F(f) = p => [X] => fmap_G1(f)(p[X])`

# Step 1. Deriving cmap from types

- When  $F[_]$  is contravariant, we need the `cmap` function  
 $\text{cmap\_G}[A, B]: (A \Rightarrow B) \Rightarrow G[B] \Rightarrow G[A]$
- Use structural induction on the type of  $F[_]$ :
  - ① If  $F[A] = \text{Unit}$  or  $F[A] = B$  then  $\text{cmap\_F}(f) = \text{identity}$
  - ② If  $F[A] = A$  then  $F$  is *not* contravariant!
  - ③ If  $F[A] = (G[A], H[A])$  then we need `cmap_G` and `cmap_H`  
 $\text{cmap\_F}(f) = \{ \text{case } (gb, hb) \Rightarrow (\text{cmap\_G}(f)(gb), \text{cmap\_H}(f)(hb)) \}$
  - ④ If  $F[A] = \text{Either}[G[A], H[A]]$  then  $\text{cmap\_F}(f) = \{$   
     $\text{case Left}(gb) \Rightarrow \text{Left}(\text{cmap\_G}(f)(gb))$   
     $\text{case Right}(hb) \Rightarrow \text{Right}(\text{cmap\_H}(f)(hb))$   
     $\}$
  - ⑤ If  $F[A] = G[A] \Rightarrow H[A]$  then we need `fmap_G` and `cmap_H`  
We define  $\text{cmap\_F}(f)(k: G[B] \Rightarrow H[B]) =$   
     $\text{fmap\_G}(f) \text{ andThen } k \text{ andThen } \text{cmap\_H}(f)$
  - ⑥ If  $F[A] = G[F[A], A]$  then we need `fmap_G1` and `cmap_G2`  
 $\text{cmap\_F}(f) = \text{fmap\_G1}(\text{cmap\_F}(f)) \text{ andThen } \text{cmap\_G2}(f)$
  - ⑦ If  $F[A] = [X] \Rightarrow G[A, X]$  then we need `cmap_G1`  
 $\text{cmap\_F}(f) = k \Rightarrow [X] \Rightarrow \text{cmap\_G1}(f)(k[X])$

## Step 1. Detect covariance and contravariance from types

- The same constructions for `fmap` and `cmap` except for function types
- The function arrow (`=>`) swaps covariant and contravariant positions
- In any fully parametric type expression, each type parameter is either in a covariant position or in a contravariant position

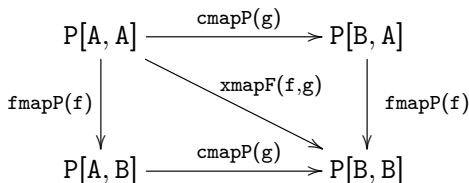
```
type F[A, B] = (A => Either[A, B], A => (B => A) => (A, B))
               -           + +   -           +   -           + +
```

- $F[A, B]$  is covariant w.r.t.  $B$  since  $B$  is always in covariant positions
  - ▶ But  $F[A, B]$  is neither covariant nor contravariant w.r.t.  $A$
  - ▶ We can recognize co(ntra)variance by counting nested function arrows
- Defined in this way, co(ntra)variance is independent of subtyping
- We can generate the code for `fmap` or `cmap` mechanically, from types
- A type expression  $F[A, B, \dots]$  can be analyzed with respect to each of the type parameters separately, and found to be covariant, contravariant, or neither (“invariant”)
- We can write the naturality law for any type signature  $F[A] \Rightarrow G[A]$

# Step 1. “Invariant” type constructors. Profunctors

For “invariant” types, we use a trick: rename contravariant positions

- Example: `type F[A] = Either[A => (A, A), (A, A) => A]`
- Define `type P[X, A] = Either[X => (A, A), (X, X) => A]`
- Then `F[A] = P[A, A]` while `P[X, A]` is contravariant in `X` and covariant in `A`. Such `P[X, A]` are called **profunctors**
- We can implement `cmap` with respect to `X` and `fmap` with respect to `A`  
`def fmapP[X, A, B]: (A => B) => P[X, A] => P[X, B]`  
`def cmapP[X, Y, A]: (X => Y) => P[Y, A] => P[X, A]`
- Then we can compose `cmapP` and `fmapP` to get `xmapF`:  
`def xmapF[A, B]: (A => B, B => A) => P[A, A] => P[B, B] =`  
    `(f, g) => cmapP[A, B, A](g) andThen fmapP[B, A, B](f)`
- What if we compose in another order? A commutativity law holds:



# Step 1. Verifying the functor laws

`fmap` and `cmap` need to satisfy two functor laws

- Identity law:

`fmap(identity) = identity`

`cmap(identity) = identity`

- Composition law: for any `f: A => B` and `g: B => C`,

`fmap(f) andThen fmap(g) = fmap(f andThen g)`

`cmap(g) andThen cmap(f) = cmap(f andThen g)`

- Go through each case and prove that the laws hold

- ▶ Proofs by induction on the type structure

## Step 1. Functor laws: composition law for tuples

- We will prove the composition law for `fmap` in case 3

`fmap_F(f) = { case (ga, ha) => (fmap_G(f)(ga), fmap_H(f)(ha)) }`

For any `f: A => B` and `g: B => C` and values `ga: G[A]`, `ha: H[A]`:

- Apply `fmap_F(f)` and then `fmap_F(g)` to the tuple `(ga, ha)`:

`fmap_F(f)((ga, ha)) == ( fmap_G(f)(ga), fmap_H(f)(ha) )`

`fmap_F(g)((fmap_G(f)(ga), fmap_H(f)(ha)))`  
`== (fmap_G(g)(fmap_G(f)(ga)), fmap_H(g)(fmap_H(f)(ha)))`  
`== ( (fmap_G(f) andThen fmap_G(g))(ga), (fmap_H(f) andThen`  
`fmap_H(g))(ha) )`

- Apply `fmap_F(f andThen g)` to the tuple `(ga, ha)`:

`fmap_F(f andThen g)((ga, ha)) == ( fmap_G(f andThen g)(ga),`  
`fmap_H(f andThen g)(ha) )`

- The law holds for `fmap_F` if it already holds for `fmap_G` and `fmap_H`



## Step 1. Functor laws: composition law for function types

- We will prove the composition law for `cmap` in case 5

`cmap_F(f)(k) == fmap_G(f) andThen k andThen cmap_H(f)`

For any `f: A => B` and `g: B => C` and `kc: G[C] => H[C]`:

Apply `cmap_F(g) andThen cmap_F(f)` to `kc`:

`cmap_F(g)(kc) == fmap_G(g) andThen kc andThen cmap_H(g)`

`cmap_F(f)(fmap_G(g) andThen kc andThen cmap_H(g))`  
`== fmap_G(f) andThen fmap_G(g) andThen kc andThen cmap_H(g)`  
`andThen cmap_H(f)`  
`== fmap_G(f andThen g) andThen kc andThen cmap_H(f andThen g)`

This is the same as `cmap_F(f andThen g)(kc)` by inductive assumption

- The law holds for `cmap_F` if it already holds for `fmap_G` and `cmap_H`

## Step 1. Functor laws: composition law for recursive types

- We will prove the composition law for `fmap` in case 6

`fmap_F(f) = fmap_G1(fmap_F(f)) andThen fmap_G2(f)`

For any `f: A => B` and `g: B => C` and `kc: G[C] => H[C]` and `ga: G[A]:`

LHS: `fmap_F(f) andThen fmap_F(g) == fmap_G1(fmap_F(f)) andThen  
fmap_G2(f) andThen fmap_G1(fmap_F(g)) andThen fmap_G2(g)`

RHS: `fmap_F(f andThen g) == fmap_G1(fmap_F(f andThen g)) andThen  
fmap_G2(f andThen g) == fmap_G1(fmap_F(f) andThen fmap_F(g))  
andThen fmap_G2(f) andThen fmap_G2(g) == fmap_G1(fmap_F(f))  
andThen fmap_G1(fmap_F(g)) andThen fmap_G2(f) andThen fmap_G2(g)`

- LHS equals RHS if the commutativity law holds for `G`
- The law holds for `fmap_F` if the composition laws and the commutativity law already hold for `fmap_G1` and `fmap_G2`

# Step 1. Summary

- `fmap` or `cmap` or `xmap` follow from a given type expression  $F[A]$
- The code of `fmap`, `cmap`, `xmap` is always fully parametric and lawful
  - ▶ That is the “standard” code to be used by naturality laws
- Consistency of the definition of `xmap` requires a commutativity law
- Functor laws for recursive types require a commutativity law
  - ▶ The commutativity laws follow from naturality and will be proved later

## Step 2. Motivation for relational parametricity. I. Papers

Parametricity theorem: any fully parametric function satisfies a certain law  
“Relational parametricity” is a powerful method for proving the parametricity theorem and for using it to prove other laws

- Main papers: Reynolds (1983) and Wadler “Theorems for free” (1989)
  - ▶ Those papers are limited in scope and hard to understand
- There are *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 derives the main results *not* following any of the above
- This tutorial explains a minimum of necessary knowledge and notation

## Step 2. Motivating relational parametricity. II. The difficulty

Naturality laws are formulated via liftings (`fmap`, `cmap`), for example:

```
fmap(f) andThen t == t andThen fmap(f)
```

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  $f: A \Rightarrow B$  to  $\text{cmap}(f): F[B] \Rightarrow F[A]$

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  $F[_]$  and  $G[_]$  being both covariant or both contravariant
  - ▶ Because some occurrences of  $A$  are in covariant and contravariant positions together in function arguments, e.g.,  $(T \Rightarrow A \Rightarrow A) \Rightarrow \dots$
- What law (similar to a naturality law) does `foldLeft` obey with respect to the type parameter  $A$ ?
- We need to formulate a more general naturality law that applies to all type constructors  $F[A]$ , not necessarily covariant nor contravariant

## Step 2. Motivating relational parametricity. III. The solution

The difficulty is resolved using three nontrivial ideas:

- 1 Replace functions  $f: A \Rightarrow B$  by binary relations  $r: A \Leftrightarrow B$ 
  - ▶ The **graph** relation:  $(a, b)$  in  $\text{graph}(f)$  means  $f(a) == b$
  - ▶ Relations are more general than functions, can be many-to-many
  - ▶ Instead of  $f(a) == b$ , we will write  $(a, b)$  in  $r$
- 2 It is *a*lways possible to 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  $r: A \Leftrightarrow B$ ,

$$\begin{array}{ccc} F[A] & \xrightarrow{t[A]} & G[A] \\ \uparrow \text{rmap}(r) \text{ for } F & & \uparrow \text{rmap}(r) \text{ for } G \\ F[B] & \xrightarrow{t[B]} & G[B] \end{array}$$

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

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

The relational naturality law will reduce to the ordinary naturality law when  $F[_]$  and  $G[_]$  are both co(ntra)variant and  $r = \text{graph}(f)$  for any  $f: A \Rightarrow B$

## Step 2. Formulating naturality laws via relations

Ordinary naturality law of  $t[A] : F[A] \Rightarrow G[A]$

$$\begin{array}{ccc} F[A] & \xrightarrow{t[A]} & G[A] \\ \text{fmap}_F(f) \downarrow & & \downarrow \text{fmap}_G(f) \\ F[B] & \xrightarrow{t[B]} & G[B] \end{array}$$

$\forall fa: F[A], fb: F[B]$  if  $fa.map(f) == fb$  then  $t(fa).map(f) == t(fb)$   
Rewrite this via relations: For all  $fa: F[A], fb: F[B]$ , when  $(fa, fb)$  in  $graph(fmap\_F(f))$  then  $(t(fa), t(fb))$  in  $graph(fmap\_G(f))$

We expect:  $graph(fmap(f)) == rmap(graph(f))$ , replace  $graph(f)$  by  $r$ :  
when  $(fa, fb)$  in  $rmap\_F(graph(f))$  then  $(t(fa), t(fb))$  in  $rmap\_G(graph(f))$

when  $(fa, fb)$  in  $rmap\_F(r)$  then  $(t(fa), t(fb))$  in  $rmap\_G(r)$

$$\begin{array}{ccc} F[A] & \xrightarrow{t[A]} & G[A] \\ rmap\_F(r) \updownarrow & & \updownarrow rmap\_G(r) \\ F[B] & \xrightarrow{t[B]} & G[B] \end{array}$$

## Step 3. Definition of relations. Examples

In the terminology of relational databases:

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

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  $f: A \Rightarrow B$  generates the **graph** relation  $\text{graph}(f): A \Leftrightarrow B$

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

Example of a relation that can be many-to-many: given any  $f: A \Rightarrow C$  and  $g: B \Rightarrow C$ , define the **pullback relation**:  $\text{pull}(f, g): A \Leftrightarrow B$ ;

$(a: A, b: B)$  in  $\text{pull}(f, g)$  means  $f(a) == g(b)$

- The pullback relation is *not* the graph of a function  $A \Rightarrow B$  or  $B \Rightarrow A$



## Step 3. Relational combinators

Given two relations  $r: A \Leftrightarrow B$  and  $s: X \Leftrightarrow Y$ , we define new relations:

- Pair product:  $\text{prod}(r, s)$  of type  $(A, X) \Leftrightarrow (B, Y)$   
 $((a, x), (b, y)) \text{ in } \text{prod}(r, s)$  means  $(a, b) \text{ in } r$  and  $(x, y) \text{ in } s$
- Pair co-product:  $\text{psum}(r, s)$  of type  $\text{Either}[A, X] \Leftrightarrow \text{Either}[B, Y]$   
 $(\text{Left}(a), \text{Left}(b)) \text{ in } \text{psum}(r, s)$  if  $(a, b) \text{ in } r$   
 $(\text{Right}(x), \text{Right}(y)) \text{ in } \text{psum}(r, s)$  if  $(x, y) \text{ in } s$
- Pair mapper:  $\text{pmap}(r, s)$  of type  $(A \Rightarrow X) \Leftrightarrow (B \Rightarrow Y)$   
 $(f, g) \text{ in } \text{pmap}(r, s)$  means when  $(a, b) \text{ in } r$  then  $(f(a), g(b)) \text{ in } s$
- Reverse:  $\text{rev}(r)$  has type  $B \Leftrightarrow A$   
 $(b, a) \text{ in } \text{rev}(r)$  means  $(a, b) \text{ in } r$

## Step 4. The relational lifting (`rmap`)

For a type constructor  $F$  and  $r: A \Leftrightarrow B$ , need  $\text{rmap}(r): F[A] \Leftrightarrow F[B]$

Define  $\text{rmap}$  for  $F[A]$  by induction over the *type expression* of  $F[A]$

For a fully parametric  $F[A]$  we have seven cases:

- ①  $F[A] = \text{Unit}$  or  $F[A] = Z$  (a fixed type other than  $A$ ):  $\text{rmap}(r) = \text{id}$
- ②  $F[A] = A$ : define  $\text{rmap}_F(r) = r$
- ③  $F[A] = (G[A], H[A]): \text{rmap}_F(r) = \text{prod}(\text{rmap}_G(r), \text{rmap}_H(r))$
- ④  $F[A] = \text{Either}[G[A], H[A]]$ :  
 $\text{rmap}_F(r) = \text{psum}(\text{rmap}_G(r), \text{rmap}_H(r))$
- ⑤  $F[A] = G[A] \Rightarrow H[A]: \text{rmap}_F(r) = \text{pmap}(\text{rmap}_G(r), \text{rmap}_H(r))$
- ⑥ Recursive type:  $F[A] = G[A, F[A]]$ :  
 $\text{rmap}_F(r) = \text{rmap2}_G(r, \text{rmap}_F(r))$
- ⑦ Universally quantified type:  $F[A] = [X] \Rightarrow G[A, X]$ :  
 $\text{rmap}_F(r) = \text{forall}(X, Y). \text{forall}(s: X \Leftrightarrow Y). \text{rmap2}_G(r, s)$

- The inductive assumption is that liftings to  $G$  and  $H$  are already defined

Define  $\text{rmap2}$  similarly (and  $\text{rmap3}$ ,  $\text{rmap4}$ , ...)

For purely covariant or contravariant  $F[A]$  we will get  $\text{fmap}$  or  $\text{cmap}$

## Step 4. Example: `rmap` for function types

Compare `fmap` and `rmap` for function types

To rewrite `fmap` via relations, introduce intermediate arguments

Let  $F[A] = G[A] \Rightarrow H[A]$  and take any  $p: G[A] \Rightarrow H[A]$ ,  $f: A \Rightarrow B$

Define  $q = \text{fmap\_F}(f)(p) = (gb: G[B]) \Rightarrow \text{fmap\_H}(f)(p(\text{cmap\_G}(f)(gb)))$

Rewrite this via relations:  $(p, q) \text{ in } \text{graph}(\text{fmap\_F}(f))$  means:

for all  $gb: G[B]$  we must have  $q(gb) = \text{fmap\_H}(f)(p(\text{cmap\_G}(f)(gb)))$

Define  $ga: G[A] = \text{cmap\_G}(f)(gb)$ , then:  $q(gb) = \text{fmap\_H}(f)(p(ga))$

But  $ga = \text{cmap\_G}(f)(gb)$  means  $(ga, gb) \text{ in } \text{rev}(\text{graph}(\text{cmap\_G}(f)))$

So, the relational formulation of `fmap_F` is:

$(p, q) \text{ in } \text{graph}(\text{fmap\_F}(f))$  means for all  $ga: G[A]$ ,  $gb: G[B]$  when

$(ga, gb) \text{ in } \text{rev}(\text{graph}(\text{cmap\_G}(f)))$  then:

$(p(ga), q(gb)) \text{ in } \text{graph}(\text{fmap\_H}(f))$

Replace  $\text{graph}(f)$  by an arbitrary relation  $r: A \Leftrightarrow B$ ; replace

$\text{graph}(\text{fmap\_F}(f))$  by  $\text{rmap\_F}(r)$ ;  $\text{rev}(\text{graph}(\text{cmap\_G}(f)))$  by  $\text{rmap\_G}(r)$

Then we get:  $(p, q) \text{ in } \text{rmap}(r)$  means for all  $ga: G[A]$ ,  $gb: G[B]$  when

$(ga, gb) \text{ in } \text{rmap\_G}(r)$  then  $(p(ga), q(gb)) \text{ in } \text{rmap\_H}(r)$

This is the same as  $(p, q) \text{ in } \text{pmap}(\text{rmap\_G}(r), \text{rmap\_H}(r))$

## Step 4. Properties of rmap

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

Two main examples of relations generated by functions:

`graph(f)` and `pull(f, g)`

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

- If  $F[A]$  is covariant then: `rmap(graph(f)) == graph(fmap(f))`
- If  $G[A] = A \Rightarrow A$  then  $(fa, fb)$  in `rmap(graph(f))` means:  
when  $(a, b)$  in `graph(f)` then  $(fa(a), fb(b))$  in `graph(f)`  
or: `f(fa(a)) == fb(f(a))` or: `fa andThen f == f andThen fb`  
This relation between `fa` and `fb` has the form of a pullback
- If  $H[A] = (A \Rightarrow A) \Rightarrow A$  then  $(fa, fb)$  in `rmap_H(graph(f))` means:  
when  $(p, q)$  in `rmap_G(graph(f))` then  $(fa(p), fb(q))$  in `graph(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 are neither a graph nor a pullback

This happens when lifting to a sufficiently complicated type constructor

## Example: applying relational naturality to $[A] \Rightarrow A \Rightarrow A$

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

- The value  $t$  has type  $[A] \Rightarrow A \Rightarrow A$

Relational naturality law says:

- For any types  $A$  and  $B$ , and for any relation  $r: A \Leftrightarrow B$ , we have:

$(t[A], t[B]) \text{ in } \text{rmap\_P}(r)$

For the type  $P[A] = A \Rightarrow A$  we have:

$\text{rmap\_P}(r): (A \Rightarrow A) \Leftrightarrow (B \Rightarrow B)$

$\text{rmap\_P}(r) = \text{pmap}(r, r)$

- $(p, q) \text{ in } \text{pmap}(r, r)$  means: for any  $a: A$  and  $b: B$ , 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  $r$  such that  $(a, b) \text{ in } r$  only when  $a == a_0$  and  $b == b_0$

- Whenever  $a == a_0$  and  $b == b_0$  then  $t(a) == a_0$  and  $t(b) == b_0$
- So,  $t(a_0) == a_0$  and  $t(b_0) == b_0$  for all  $a_0: A$  and  $b_0: B$
- It means that  $t$  must be an identity function

## Step 5. Formulation of relational naturality law

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  $t[A] : F[A]$  satisfies for any  $r : A \Leftrightarrow B$  the relation  $(t[A], t[B]) \text{ in } \text{rmap\_F}(r)$

It is convenient to prove the relational law with a free variable:

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

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

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

## Step 5. Outline of the proof of the relational naturality law

The theorem says that  $t[A](z)$  satisfies its relational naturality 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 free variable  $z$ )

In each inductive case, we choose arbitrary  $z1: P[A]$ ,  $z2: P[B]$  such that  $(z1, z2) \text{ in } rmap\_P(r)$

## Step 5. First four inductive cases of the proof

Constant type:  $t[A](z) = c$  where  $c: C$  has a fixed type  $C$ :

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

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

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

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

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

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

- If  $(z1, z2) \text{ in rmap\_P}(r)$  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:  
 $(g(h1), g(h2)) \text{ in rmap\_Q}(r)$



## Step 5. Next four inductive cases of the proof

Create tuple:  $t[A](z) = (p(z), q(z))$  and\*\*\*:

- We have  $\text{rmap\_P}(r) =$

Use tuple:  $t[A](z) = g[A](z) \cdot \_1$  where  $g[A]$  has type  $(Q[A], L[A])$ :

- If  $(z1, z2)$  in \*\*\*

Create disjunction:  $t[A](z) = \text{Left}[K[A], L[A]](g[A](z))$ :

- If  $(z1, z2)$  \*\*\*

Use disjunction:  $t(z) = \_ \text{match } \{$

case  $\text{Left}(x) \Rightarrow p(z)(x)$

case  $\text{Right}(y) \Rightarrow q(z)(y)$

$\}$

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

## Step 6. From relational naturality to the wedge law

\*\*\*Create tuple:  $t[A](z) = (p(z), q(z))$  and\*\*\*:

## Step 6. From the wedge law to naturality laws

\*\*\*Create tuple:  $t[A](z) = (p(z), q(z))$  and\*\*\*:

# Advanced applications. I. Church encodings

- Recursive types defined by induction:  $T \cong S[T]$  with *covariant*  $S[_]$
- Isomorphism is given by `fix: S[T] => T` and `unfix: T => S[T]`
- `fix andThen unfix == identity; unfix andThen fix == identity`
- Church encoding: `CT = [A] => (S[A] => A) => A` (fully parametric)
- Using Scala 2 traits: `trait CT { def run[A](fix: S[A] => A): A }`
- The Church encoding (`CT`) is equivalent to the inductive definition (`T`)

## Advanced applications. II. Quantified types

- Define  $\text{type } F[R] = [A] \Rightarrow ((A \Rightarrow R) \Rightarrow A) \Rightarrow A$
- This is the Church encoding of an (invalid) recursive type  $T \cong T \Rightarrow R$
- We will use the relational naturality law to prove that  $F[R] \cong R$

# Summary

- “Theorems for free” are laws always satisfied by fully parametric code
- Relational parametricity is a powerful proof technique
- Relational parametricity has a steep learning curve
  - ▶ The result may be a relation that is difficult to interpret as code
  - ▶ Cannot directly write code that manipulates relations
  - ▶ All calculations need to be done symbolically or with proof assistants
- Naturality laws and the wedge law are shortcuts to “theorems for free”
  - ▶ But a few proofs in FP do require the relational naturality law
- More details in the free book — <https://github.com/winitzki/sofp>

