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] => G[A]` *before* `_.map(f)` equals applying `t[B]: F[B] => G[B]` *after* `_.map(f)` for any function `f: A => B`

$$\begin{tikzcd}
\texttt{F[A]} \arrow[r, "\texttt{t[A]}"] \arrow[d, "\texttt{_.map(f) for F}"'] & \texttt{G[A]} \arrow[d, "\texttt{_.map(f) for G}"] \\
\texttt{F[B]} \arrow[r, "\texttt{t[B]}"] & \texttt{G[B]}
\end{tikzcd}$$

- Example: `F = List`, `G = Option`, `t = headOption`
  The naturality law of `headOption`: for all `x: List[A]` and `f: A => B`,
  `x.headOption.map(f) = x.map(f).headOption`

Naturality laws are formulated using `_.map` for `F` and `G`

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

- Equivalently, the code of `fmap[A, B]: (A => B) => F[A] => 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 => A) => F[A]` satisfying the naturality law:
`p[A](k).map(f) == p[B](k andThen f)` for all `f: A => B`
Isomorphism maps:

```
inY[A]: F[R] => (R => A) => F[A] = fr => k => fr.map[A](k)
outY: ([A] => (R => A) => F[A]) => F[R] = p => p[R](identity[R])
```

Proofs of isomorphism:

```
outY(inY(fr)) == outY(k => fr.map(k)) == fr.map(identity) == fr
```

The other direction:

```
inY(outY(p)) == k => outY(p).map(k) == k => p(identity).map(k)
```

Use the naturality law: `p(identity).map(k) == p(identity andThen k)`
So: `inY(outY(p)) == k => 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 => R) => G[A]` satisfying the appropriate naturality law

Example applications of the Yoneda identities

Many types can be converted to the form `[A] => (R => A) => F[A]` with a covariant `F` or to `[A] => (A => R) => G[A]` with a contravariant `G`
Some examples (assume covariant `F[_]` and contravariant `G[_]`):

- `[A] => A` is equivalent to `Nothing`
- `[A] => F[A]` is equivalent to `F[Nothing]`
- `[A] => G[A]` is equivalent to `G[Unit]`
- `[A] => A => A` is equivalent to `Unit`
- `[A] => A => F[A]` is equivalent to `F[Unit]`
- `[A] => (A, A) => A` is equivalent to `Boolean`
- `[A] => (A, A) => F[A]` is equivalent to `F[Boolean]`
- `[A] => (P => A) => Q => A` is equivalent to `Q => P`
- `[A] => (A => P) => A => Q` is equivalent to `P => Q`
- `[A] => F[A] => (A => P) => 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])
  ```

  `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 `cmap_G[A, B]: (A => B) => G[B] => G[A]`
- Use structural indunction on the type of `F[_]`:
  1. If `F[A] = Unit` or `F[A] = B` then `cmap_F(f) = 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`
     `cmap_F(f) = { case (gb, hb) => (cmap_G(f)(gb), cmap_H(f)(hb)) }`
  4. If `F[A] = Either[G[A], H[A]]` then
     ```
     cmap_F(f) = {
       case Left(gb)  => Left(cmap_G(f)(gb))
       case Right(hb) => Right(cmap_H(f)(hb))
     }
     ```
  5. If `F[A] = G[A] => H[A]` then we need `fmap_G` and `cmap_H`
     We define `cmap_F(f)(k: G[B] => H[B]) =`
     `fmap_G(f) andThen k andThen cmap_H(f)`
  6. If `F[A] = G[F[A], A]` then we need `fmap_G1` and `cmap_G2`
     `cmap_F(f) = fmap_G1(cmap_F(f)) andThen cmap_G2(f)`
  7. If `F[A] = [X] => G[A, X]` then we need `cmap_G1`
     `cmap_F(f) = k => [X] => 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, ...]` 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] => 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 commutative law holds:

$$\begin{tikzcd} \mathtt{P[A,A]} \arrow[r, "\mathtt{cmapP(g)}"] \arrow[d, "\mathtt{fmapP(f)}"'] \arrow[dr, "\mathtt{xmapF(f,g)}"] & \mathtt{P[B,A]} \arrow[d, "\mathtt{fmapP(f)}"] \\ \mathtt{P[A,B]} \arrow[r, "\mathtt{cmapP(g)}"] & \mathtt{P[B,B]} \end{tikzcd}$$

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) andThen 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(f))(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 cmapH(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
  - 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 => B` to `F[A] => F[B]` when `F[_]` is not covariant!

- For covariant `F[_]` we lift `f: A => B` to `fmap(f): F[A] => F[B]`
- For contravariant `F[_]` we lift `f: A => B` to `cmap(f): F[B] => 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] => 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 => A => A) =>...`
- 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 => B` by binary relations `r: A <=> B`
   - The **graph** relation: `(a, b) in 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 *always* possible to lift `r: A <=> B` to `rmap(r): F[A] <=> F[B]`
3. Reformulate the naturality law of `t` via relations: for any `r: A <=> B`,

$$
\begin{tikzcd}
\mathrm{F[A]} \arrow[r, "\mathrm{t[A]}"] \arrow[d, leftrightarrow, "\mathrm{rmap(r)\ for\ F}"'] & \mathrm{G[A]} \arrow[d, leftrightarrow, "\mathrm{rmap(r)\ for\ G}"] \\
\mathrm{F[B]} \arrow[r, "\mathrm{t[B]}"] & \mathrm{G[B]}
\end{tikzcd}
$$

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

For any `r: A <=> B`, for any `fa: F[A]` and `fb: F[B]` such that `(fa, fb) in rmap_F(r)`, we require `(t(fa), t(fb)) in 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 = graph(f)` for any `f:A => B`

Step 2. Formulating naturality laws via relations

Ordinary naturality law of `t[A]: F[A] => G[A]`

$$\begin{tikzcd}
\texttt{F[A]} \arrow[r, "\texttt{t[A]}"] \arrow[d, "\texttt{fmap_F(f)}"'] & \texttt{G[A]} \arrow[d, "\texttt{fmap_G(f)}"] \\
\texttt{F[B]} \arrow[r, "\texttt{t[B]}"] & \texttt{G[B]}
\end{tikzcd}$$

$\forall$ `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)`

$$\begin{tikzcd}
\texttt{F[A]} \arrow[r, "\texttt{t[A]}"] \arrow[d, leftrightarrow, "\texttt{rmap_F(r)}"'] & \texttt{G[A]} \arrow[d, leftrightarrow, "\texttt{rmap_G(r)}"] \\
\texttt{F[B]} \arrow[r, "\texttt{t[B]}"] & \texttt{G[B]}
\end{tikzcd}$$

Step 3. Definition of relations. Examples

In the terminology of relational databases:

- A relation `r: A <=> 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 <=> 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 => B` are many-to-one
A function `f: A => B` generates the **graph** relation `graph(f): A <=> B`

- Two values `a: A`, `b: B` are in `graph(f)` if `f(a) == b`
- `graph(identity: A => A)` gives an **identity relation** `id: A <=> A`

Example of a relation that can be many-to-many: given any `f: A => C` and `g: B => C`, define the **pullback relation**: `pull(f, g): A <=> B`;
`(a: A, b: B) in pull(f, g)` means `f(a) == g(b)`

- The pullback relation is *not* the graph of a function `A => B` or `B => A`

Step 3. Relational combinators

Given two relations `r: A <=> B` and `s: X <=> Y`, we define new relations:

- Pair product: `prod(r, s)` of type `(A, X) <=> (B, Y)`

`((a, x), (b, y)) in prod(r, s)` means `(a, b) in r` and `(x, y) in s`

- Pair co-product: `psum(r, s)` of type `Either[A, X] <=> Either[B, Y]`

`(Left(a), Left(b)) in psum(r, s)` if `(a, b) in r`
`(Right(x), Right(y)) in psum(r, s)` if `(x, y) in s`

- Pair mapper: `pmap(r, s)` of type `(A => X) <=> (B => Y)`

`(f, g) in pmap(r, s)` means when `(a, b) in r` then `(f(a), g(b)) in s`

- Reverse: `rev(r)` has type `B <=> A`

`(b, a) in rev(r)` means `(a, b) in r`

Step 4. The relational lifting (`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]`
For a fully parametric `F[A]` we have seven cases:

1. `F[A] = Unit` or `F[A] = Z` (a fixed type other than `A`): `rmap(r) = id`
2. `F[A] = A`: define `rmap_F(r) = r`
3. `F[A] = (G[A], H[A])`: `rmap_F(r) = prod(rmap_G(r), rmap_H(r))`
4. `F[A] = Either[G[A], H[A]]`:
   `rmap_F(r) = psum(rmap_G(r), rmap_H(r))`
5. `F[A] = G[A] => H[A]`: `rmap_F(r) = pmap(rmap_G(r), rmap_H(r))`
6. Recursive type: `F[A] = G[A, F[A]]`:
   `rmap_F(r) = rmap2_G(r, rmap_F(r))`
7. Universally quantified type: `F[A] = [X] => G[A, X]`:
   `rmap_F(r) = forall(X, Y). forall(s: X <=> Y). rmap2_G(r, s)`

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

Define `rmap2` similarly (and `rmap3`, `rmap4`, ...)
For purely covariant or contravariant `F[A]` we will get `fmap` or `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] => H[A]` and take any `p: G[A] => H[A]`, `f: A => B`

Define `q = fmap_F(f)(p) = (gb: G[B]) => fmap_H(f)(p(cmap_G(f)(gb))`

Rewrite this via relations: `(p, q) in graph(fmap_F(f))` means:

for all `gb: G[B]` we must have `q(gb) = fmap_H(f)(p(cmap_G(f)(gb))`

Define `ga: G[A] = cmap_G(f)(gb)`, then: `q(gb) = fmap_H(f)(p(ga))`

But `ga = cmap_G(f)(gb)` means `(ga, gb) in rev(graph(cmap_G(f)))`

So, the relational formulation of `fmap_F` is:

`(p, q) in graph(fmap_F(f))` means for all `ga: G[A]`, `gb: G[B]` when `(ga, gb) in rev(graph(cmap_G(f)))` then:
`(p(ga), q(gb)) in graph(fmap_H(f))`

Replace `graph(f)` by an arbitrary relation `r: A <=> B`; replace `graph(fmap_F(f))` by `rmap_F(r)`; `rev(graph(cmap_G(f)))` by `rmap_G(r)`

Then we get: `(p, q) in rmap(r)` means for all `ga: G[A]`, `gb: G[B]` when `(ga, gb) in rmap_G(r)` then `(p(ga), q(gb)) in rmap_H(r)`

This is the same as `(p, q) in pmap(rmap_G(r), 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 => 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 => A) => 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] => A => A`

Example: `t[A] = identity[A]` of type `P[A] = A => A`

- The value `t` has type `[A] => A => A`

Relational naturality law says:

- For any types `A` and `B`, and for any relation `r: A <=> B`, we have:

`(t[A], t[B]) in rmap_P(r)`

For the type `P[A] = A => A` we have:

```
rmap_P(r): (A => A) <=> (B => B)
rmap_P(r) = pmap(r, r)
```

- `(p, q) in pmap(r, r)` means: for any `a: A` and `b: B`, if `(a, b) in r` then `(p(a), q(b)) in r`
- So, `(t[A], t[B]) in rmap_P(r)` means: for any `a: A`, `b: B`, if `(a, b) in r` then `(t(a), t(b)) in r`

Trick: choose `r` such that `(a, b) in r` only when `a == a0` and `b == b0`

- Whenever `a == a0` and `b == b0` then `t(a) == a0` and `t(b) == b0`
- So, `t(a0) == a0` and `t(b0) == b0` for all `a0: A` and `b0: 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] => 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 <=> B` the relation `(t[A], t[B]) in 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 <=> B` and for any `z1: P[A]`, `z2: P[B]`, the law: if `(z1, z2) in rmap_P(r)` then `(t[A](z1), t[B](z2)) in rmap_Q(r)`
- Equivalently: `(t[A], t[B]) in pmap(rmap_P(r), 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) 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 `rmap_P(r) == id` and `(c, c) in id` holds

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

- If `(z1, z2) in rmap_P(r)` then `(t(z1), t(z2)) in rmap_Q(r)`

Create function: `t(z) = h => s(z, h)` where `h: H[A]` and `s(z, h): S[A]`

- If `(z1, z2) in rmap_P(r)` and `(h1, h2) in rmap_H(r)` then `(s(z1, h1), s(z2, h2)) in rmap_S(r)`

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

- If `(z1, z2) in rmap_P(r)` then inductive assumption says: `(h(z1), h(z2)) in rmap_H(r)`
- If `(h1, h2) in rmap_H(r)` then inductive assumption says: `(g(h1), g(h2)) in rmap_Q(r)`

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

- We have `rmap_P(r) =`

Use tuple: `t[A](z) = g[A](z)._1` where `g[A]` has type `(Q[A],L[A])`:

- If `(z1, z2) in ***`

Create disjunction: `t[A](z) = Left[K[A], L[A]](g[A](z))`:

- If `(z1, z2) ***`

Use disjunction: `t(z) = _ match {`

```
    case Left(x) => p(z)(x)
    case Right(y) => q(z)(y)
}
```

- If `(z1, z2) in 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 `type F[R] = [A] => ((A => R) => A) => A`
- This is the Church encoding of an (invalid) recursive type `T` $\cong$ `T => 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