Relational parametricity and "theorems for free"

A tutorial, with example code in Scala

Sergei Winitzki

Academy by the Bay

2021-09-04

Outline of the tutorial

- Motivation: practical applications of the parametricity theorem
- What is "fully parametric code"
- A complete proof of "theorems for free" in 7 steps
  - Step 1: Deriving `fmap` and `cmap` methods from types
  - Step 2: The commutativity law for bifunctors and profunctors
  - Step 3: Motivation for the relational formulation of naturality laws
  - Step 4: Definition of relational lifting (`rmap`)
  - Step 5: Properties of relational lifting
  - Step 6: Proof of the relational naturality law
  - Step 7: Deriving the wedge law from the relational naturality law
- Applications of the parametricity theorem
  - Yoneda identities
- Advanced applications of the parametricity theorem
  - Church encoding of recursive types
  - Simplifying types with universal quantifiers
  - Equivalence of `foldMap` and `foldLeft` for polynomial functors

Applications of parametricity. "Theorems for free"

Parametricity theorem: any fully parametric function obeys a certain law

Example 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] => (Option[(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 (so, 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 $A = \text{Int}$

```
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`

Step 1. 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`

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

- 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]`

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]`
     `fmap_F(f) = (k: G[A] => H[A]) => (gb: G[B]) => fmap_H(f)(k(cmap_G(f)(gb)))`
  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) = k => [X] => fmap_G1(f)(k[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`
     `cmap_F(f) = (k: G[B] => H[B]) => (ga: G[A]) => cmap_H(f)(k(fmap_G(f)(ga))`
  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 type constructions for `fmap` and `cmap` are the same 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], (B => A) => A => (A, B))
                -            +  +   +    -     -    +  +
```

- `F[A, B]` is covariant w.r.t. `B` since `B` is always in covariant positions
  - We can recognize this just by counting the function arrows
- We can generate the code for `fmap` and `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")

Step 1. "Invariant" types and profunctors

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

- Example: `type F[A] = Either[A => (A, A), (A, A) => A]`
- Define `type G[X, A] = Either[X => (A, A), (X, X) => A]`
- Then `F[A] = G[A, A]` while `G[X, A]` is contravariant in `X` and covariant in `A`. Such `G[X, A]` are called **profunctors**
- We can implement `cmap` with respect to `X` and `fmap` with respect to `A`

```
def fmapG[X, A, B]: (A => B) => G[X, A] => G[X, B]
def cmapG[X, Y, A]: (Y => X) => G[X, A] => G[Y, A]
```

- Then we can compose `cmapG` and `fmapG` to get `xmapF`:

```
def xmapF[A, B]: (A => B) => (B => A) => G[A, A] => G[B, B] =
   f => g => cmapG[A, B, A](g) andThen fmapG[B, A, B](f)
```

- What if we compose in another order? Need a commutativity law:

$$\begin{tikzcd}
\mathtt{G[A, A]} \arrow[r, "\mathtt{cmapG(g)}"] \arrow[d, "\mathtt{fmapG(f)}"'] & \mathtt{G[B, A]} \arrow[d, "\mathtt{fmapG(f)}"] \\
\mathtt{G[A, B]} \arrow[r, "\mathtt{cmapG(g)}"] & \mathtt{G[B, B]}
\end{tikzcd}$$

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)`

Step 1. Summary

- `fmap` or `cmap` or `xmap` follow from a given type expression
- The code of `fmap`, `cmap`, `xmap` is always fully parametric and lawful
- It is precisely that code that we need to use in naturality laws
- Consistency of `xmap` requires to have a commutativity law
  - Commutativity laws are the subject of Step 2

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 outdated and also 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 does *not* follow any of the above but derives all results

Motivating relational parametricity. II. The difficulty

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: B => A` to `cmap(f): F[A] => 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] => 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 => B` by relations `r: A <=> B`
   - Instead of `b == f(a)`, we will write: `(a, b) in r`
2. Turns out, we can 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`,

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

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)`

Definition and examples of relations

In the terminology of relational databases:

- A relation `r: A <=> 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 <=> 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` can be also viewed as a relation `rel(f): A <=> B`

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

Example of a relation that can be many-to-many:
Given two functions `f: A => C`, `g: B => C`, define a "pullback" relation `pullback(f, g): A <=> B` as: `(a: A, b: B) in r` means `f(a) == g(b)`

- The pullback relation is *not* equivalent to a function `A => B` or `B => A`

Proof of relational parametricity. I. Relation combinators

Relation combinators:

- For any relation `r: A <=> B`, the **inverse** relation is `inv(r): B <=> A`
  - The inverse operation is its own inverse: `inv(inv(r)) == r`
- For any relations `r: A <=> B` and `s: A <=> B`, get the union (`r or s`) and the intersection (`r and s`):

  `(a, b) in (r and s)` means `(a, b) in r` and `(a, b) in s`
  `(a, b) in (r or s)` means `(a, b) in r` or `(a, b) in s`
- For any relations `r: A <=> B` and `s: B <=> C` define the **composition** (`r compose s`) as a relation `u: A <=> C` by `(a: A, c: C) in u` when there exists `b: B` such that `(a, b) in r` and `(b, c) in s`
  - Composition corresponds to "join" in relational databases
  - Directionality law: `inv(r compose s) == inv(s) compose inv(r)`
  - Associativity and identity laws with respect to `id: A <=> A`
  - Preserves composition of functions: for `f: A => B` and `g: B => C`, `rel(f andThen g) == rel(f) compose rel(g)`
- The "pullback relation" can be expressed through composition:

  `pullback(f, g) == rel(f) compose inv(rel(g))`

Pullback relation expressed through composition of relations

For any `f: A => C`, `g: B => C`, `a: A`, `b: B`, to prove:

- `(a, b) in pullback(f, g)` is equivalent to:
  `(a, b) in rel(f) compose inv(rel(g))`

$$\mathtt{A} \xleftrightarrow{\mathtt{rel(f)}} \mathtt{C} \xleftrightarrow{\mathtt{inv(rel(g))}} \mathtt{B}$$

- 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 rel(f)` and `(c, b) in inv(rel(g))`
- This is equivalent to: `c` 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 <=> B` and `s: X <=> Y`:

$$
\begin{tikzcd}
\mathtt{G[A,\ X]} \arrow[rr, leftrightarrow, "\mathtt{rmap\_G2(s)}"] \arrow[d, leftrightarrow, "\mathtt{rmap\_G1(r)}"'] \arrow[drr, leftrightarrow] & & \mathtt{G[A,\ Y]} \arrow[d, leftrightarrow, "\mathtt{rmap\_G1(r)}"] \\
\mathtt{G[B,\ X]} \arrow[rr, leftrightarrow, "\mathtt{rmap\_G2(s)}"] & & \mathtt{G[B,\ Y]}
\end{tikzcd}
$$

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

$$
\begin{tikzcd}
\mathtt{H[A]} \arrow[rr, leftrightarrow, "\mathtt{rmap\_H(r)}"] & & \mathtt{H[B]}
\end{tikzcd}
$$

$$
\begin{tikzcd}
\mathtt{G[H[A]]} \arrow[rr, leftrightarrow, "\mathtt{rmap\_G(rmap\_H(r))}"] & & \mathtt{G[H[B]]}
\end{tikzcd}
$$

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] => 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 <=> B` the relation `(t[A], t[B]) in rmap_P(r)`
- Any fully parametric `t: P[]` satisfies `(t, t) in rmap_P(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 <=> B` and for any `z1: Q[A]`, `z2: Q[B]`, the law: if `(z1, z2) in rmap_Q(r)` then `(t[A](z1), t[B](z2)) in 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 => a }` of type `P[A] = A => A`

Parametricity theorem says:

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

`(t[A], t[B]) in rmap_P(r)` where `rmap_P(r): (A => A) <=> (B => B)`

- `(p, q) in rmap_P(r)` means: for any `a: A`, `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 = rel(f)` where `f: A => 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 => 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 `rmap_P(r) == id` while `(c, c) in id` holds

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

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

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

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

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

- If `(z1, z2) in rmap_Q(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_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