Proving "theorems for free" via relational parametricity

A tutorial using the syntax of Scala code

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, (K, A) => A) => A` is equivalent to `List[K]`
- The type `[A] => ((A => K) => A) => A` is equivalent to `K`

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`

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

- 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]`

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]: F[A] => (A => F[B]) => F[B]
def flatten[B]: F[F[B]] => 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]`
     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 induction 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 commutativity 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 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`:

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 used by all naturality laws
- Consistency of the definition of `xmap` requires a commutativity law
- Functor laws for recursive types require a commutativity law
  - Those commutativity laws are naturality laws 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)
  - Parametricity tutorial part 1, part 2, part 3 by E. de Vries (2015)
- Here I derive the main results *not* following any of the above
- I will only explain the minimum 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. Generalize functions `f: A => B` to 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`,

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

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)`

The relational naturality law will reduce to the ordinary naturality laws when `F[_]`, `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]`

$$
\begin{tikzcd}
\mathtt{F[A]} \arrow[r, "\mathtt{t[A]}"] \arrow[d, "\mathtt{fmap\_F(f)}"'] & \mathtt{G[A]} \arrow[d, "\mathtt{fmap\_G(f)}"] \\
\mathtt{F[B]} \arrow[r, "\mathtt{t[B]}"] & \mathtt{G[B]}
\end{tikzcd}
$$

$\forall$ `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)`

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

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: `pprod`, `psum`, `pfun`, `rev`

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

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

`((a, x), (b, y)) in pprod(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 function mapper: `pfun(r, s)` of type `(A => X) <=> (B => Y)`

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

$$\begin{tikzcd} \mathtt{A} \arrow[rr, "\mathtt{f}"] \arrow[d, leftrightarrow, "\mathtt{r}"'] & \arrow[d, phantom, "\mathtt{pfun(r,\ s)}\ \updownarrow"] & \mathtt{X} \arrow[d, leftrightarrow, "\mathtt{s}"] \\ \mathtt{B} \arrow[rr, "\mathtt{g}"'] & & \mathtt{Y} \end{tikzcd}$$

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

`(b, a) in rev(r)` means the same as `(a, b) in r`

Step 4. The relational lifting (rmap)

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

Define `rmap_F` for `F[A]` by induction on the *type expression* of `F[A]`

A fully parametric type `F[A]` must be built up via these seven cases:

1. `F[A] = Unit` or `F[A] = K` (a fixed type): `rmap_F(r) = id`
2. `F[A] = A`: define `rmap_F(r) = r`
3. `F[A] = (G[A], H[A])`: `rmap_F(r) = pprod(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) = pfun(rmap_G(r), rmap_H(r))`
6. Recursive type: `F[A] = G[A, F[A]]`:
   `rmap_F(r) = rmap2_G(r, rmap_F(r))` – recursive definition of `rmap_F`
7. Universally quantified type: `F[A] = [X] => G[A, X]`:
   `rmap_F(r) = ∀(X, Y). ∀(s: X <=> Y). rmap2_G(r, s)`

- The inductive assumption is that liftings to `G` and `H` are already defined
- `rmap_F` translates the type expression `F[A]` into relational combinators

We will define `rmap2` in a similar way

Step 4. Simultaneous relational lifting (`rmap2`)

For a type constructor `F[_, _]` and `r: A <=> B`, `s: P <=> Q`, we define `rmap2_F(r, s): F[A, P] <=> F[B, Q]` by induction on the type `F[A, P]`

1. `F[A, P] = K` (a fixed type): `rmap2_F(r, s) = id`
2. If `F[A, P] = A` then `rmap2_F(r, s) = r`
   If `F[A, P] = P` then `rmap2_F(r, s) = s`
3. `F[A, P] = (G[A, P], H[A, P])`:
   `rmap2_F(r, s) = pprod(rmap2_G(r, s), rmap2_H(r, s))`
4. `F[A, P] = Either[G[A, P], H[A, P]]`:
   `rmap2_F(r, s) = psum(rmap2_G(r, s), rmap2_H(r, s))`
5. `F[A, P] = G[A, P] => H[A, P]`:
   `rmap2_F(r, s) = pfun(rmap2_G(r, s), rmap2_H(r, s))`
6. Recursive type: `F[A, P] = G[A, P, F[A, P]]`:
   `rmap2_F(r, s) = rmap3_G(r, s, rmap2_F(r, s))`
7. Universally quantified type: `F[A, P] = [X] => G[A, P, X]`:
   `rmap2_F(r, s) = ∀(X, Y). ∀(t: X <=> Y). rmap3_G(r, s, t)`

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

Actually, we need to define `rmap`, `rmap2`, `rmap3`, `rmap4`, ..., all at once

This is not a problem: `F[_]` is finitely long, so the induction will stop

Step 4. Example: `rmap` for a covariant type constructor

Consider `P[A] = R => (A, A)` where `R` is a fixed type

Compare `fmap_P` and `rmap_P` defined via the inductive definitions

Case 5: `P[A] = G[A] => H[A]` with `G[A] = R` (case 1), `H[A] = (A, A)`

Case 3: `H[A] = (K[A], L[A])` with `K[A] = A`, `L[A] = A` (case 2)

For `fmap_P`:

```
fmap_P(f)(p) = cmap_G(f) andThen p andThen fmap_H(f)
fmap_H(f) = { case (k, l) => (fmap_K(f)(k), fmap_L(f)(l)) }
```

`cmap_G(f) = identity`; `fmap_K(f) = f`; `fmap_L(f) = f`

```
fmap_P(f)(p) = p andThen { case (k, l) => (f(k), f(l)) }
```

For `rmap_P`:

```
rmap_P(r) = pmap(rmap_G(r), rmap_H(r)) = pmap(id, rmap_H(r))
 = pmap(id, pprod(rmap_K(r), rmap_L(r))) = pmap(id, pprod(r, r))
```

Two values `(p: P[A], q: P[B])` are in `rmapP_(r)` if for ∀ `x: R`, `y: R`, when `(x, y) in id` then `(p(x), q(x)) in pprod(r, r)` or equivalently:

for any `x: R`, `(p(x)._1, q(x)._1) in r` and `(p(x)._2, q(x)._2) in r`

Choose `r = graph(f)` and get for any `x: R`: `f(p(x)._1) == q(x)._1` and `f(p(x)._2) == q(x)._2`

This is the same as `q == fmap_P(f)(p)` or `(p, q) in graph(fmap_P(f))`

Step 4. Example: `rmap` for function types

Compare `fmap` and `rmap` for function types: (`F[A] = G[A] => H[A]`)
To rewrite `fmap_F` via relations, introduce intermediate arguments
Choose any values `p: G[A] => H[A]` and `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 pfun(rmap_G(r), rmap_H(r))`

Step 4. Example: `rmap` for non-covariant type constructors

Consider some type constructors of different complexity:

- If `F[A]` is covariant: `rmap(graph(f)) == graph(fmap(f))`
- If `F[A]` is contravariant: `rmap(graph(f)) == rev(graph(cmap(f)))`
- If `G[A] = A => A` then `(ga, gb) in rmap(graph(f))` means:

when `(a, b) in graph(f)` then `(ga(a), gb(b)) in graph(f)`
or: `f(ga(a)) == gb(f(a))` or: `ga andThen f == f andThen gb`
This relation between `ga` and `gb` has the form of a pullback

- If `H[A] = (A => A) => A` then `(ha, hb) in rmap_H(graph(f))` is:

when `(p, q) in rmap_G(graph(f))` then `(ha(p), hb(q)) in graph(f)`
equivalently: if `p andThen f == f andThen q` then `f(ha(p)) == hb(q)`
This is *not* in the form of a pullback relation: cannot express `p` through `q`

- This happens for sufficiently complicated type constructors
- It is hard to use relations that are neither a graph nor a pullback

Example: applying relational naturality to `[A] => A => A`

Example: `def t[A]: A => A = ...  // Fully parametric.`

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

The 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) = pfun(r, r)
```

- `(t[A], t[B]) in pfun(r, r)` means:
  for any `a: A`, `b: B`, if `(a, b) in r` then `(t(a), t(b)) in r`

Trick: choose `r: A <=> A` such that `(a, b) in r` only if `a == b == a0`

- Whenever `a == b == a0` then `t(a) == t(b) == a0`
- So, `t(a0) == a0` for any fixed `a0: A`
  - It means that `t` must be an identity function

Step 5. Preparing to prove the 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 when `t` has 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 pfun(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 and for 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. The first four cases of the proof

1 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

2 Use argument: `t[A](z) = z` where `z` and `t[A]` have type `P[A]`:

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

3 Create function: `t(z) = h => s(z, h)` where we assume `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)`
  - This is the same as the inductive assumption for `s(z, h)`

4 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 the inductive assumption says: `(h(z1), h(z2)) in rmap_H(r)`
- If `(h1, h2) in rmap_H(r)` then the inductive assumption says: `(g(h1), g(h2)) in rmap_Q(r)`
- Therefore `(t[A](z1), t[B](z2)) in rmap_Q(r)`

Step 5. The next three cases of the proof

5 Create tuple: `t[A](z) = (u(z), v(z))` where `u(z): U[A]`, `v(z): V[A]`
Need `(t[A](z1), t[B](z2)) in rmap_Q(r)` where `Q[A] = (U[A], V[A])`

- As `rmap_Q(r) = pprod(rmap_U(r), rmap_V(r))`, we have `(t[A](z1), t[B](z2)) in rmap_Q(r)` when `(u(z1), u(z2)) in rmap_U(r)` and `(v(z1), v(z2)) in rmap_V(r)`, which hold by inductive assumptions

6 Use tuple: `t[A](z) = g[A](z)._1` with `g[A](z): G[A] = (Q[A], R[A])`

- By inductive assumption, `(g(z1), g(z2)) in rmap_G(r)` while we have `rmap_G(r) = pprod(rmap_Q(r), rmap_R(r))`, so we get `(g(z1)._1, g(z2)._1) in rmap_Q(r)` as required
- The case `t[A](z) = g[A](z)._2` is proved similarly

7 Create a co-product: `t[A](z) = Left[G[A], H[A]](g[A](z))`
Here we set `Q[A] = Either[G[A], H[A]]` and `g[A](z): G[A]`
By the inductive assumption, `(g(z1), g(z2)) in rmap_G(r)` and then:
`(Left(g(z1)), Left(g(z2))) in rmap_Q(r)`

- The case `t[A](z) = Left[G[A], H[A]](g[A](z))` is proved similarly

Step 5. The last two cases of the proof

8 Use a co-product (pattern-matching):

```
t(z) = s(z) match {
    case Left(x) => u(z)(x)
    case Right(y) => v(z)(y)
}
```

- We set `S[A] = Either[G[A], H[A]]`, `s(z): S[A]`, `x: G[A]`, `y: H[A]`, `u(z): G[A] => Q[A]`, and `v(z): H[A] => Q[A]`
- Inductive assumptions: `s(z)`, `u(z)`, `v(z)` already satisfy the law
- if `(z1: P[A], z2: P[B]) in rmap_P(r)` and `(x1: G[A], x2: G[B]) in rmap_G(r)` then `u(z1)(x1), u(z2)(x2)) in rmap_Q(r)`
- `(s[A](z1), s[B](z2)) in rmap_S(r) = psum(rmap_G(r), rmap_H(r))` means `s(z1), s(z2)` are both in `Left` or both in `Right`

If `s(z1) = Left(x1)`, `s(z2) = Left(x2)` then `(x1, x2) in rmap_G(r)` and `(u(z1)(x1), u(z2)(x2)) in rmap_Q(r)`

- The case when both `s(z1), s(z2)` are in `Right` is proved similarly

9 Recursive call: `t(z) = f(z)(t(z)) where f(z):  Q[A] => Q[A]`
Inductive assumptions: the law holds for `f(z)` and for the recursive `t(z)`

- Then `t(z)` satisfies the law because of the "use function" rule

Step 6. From relational naturality to the wedge law

Based on Bartosz Milewski's blog post: The Free Theorem for Ends (2017)
Given:

- a function `f: A => B`
- a fully parametric profunctor `P[X, Y]` with methods `cmapP` and `fmapP`:
  - `cmapP[X, Y, B]: (X => Y) => P[Y, B] => P[X, B]`
  - `fmapP[X, A, B]: (A => B) => P[X, A] => P[X, B]`
- a fully parametric value (without free variables) `t: [A] => P[A, A]`

Then we will prove that the wedge law holds:

- `fmapP[A, A, B](f)(t[A]) == cmapP[A, B, B](f)(t[B])`

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

- Expressed via `xmapP(f, g) = cmapP(g) andThen fmapP(f)`:
  `xmapP(f, id)(t[A]) == xmapP(id, f)(t[B])`
  - We do *not* need to assume the commutativity law for `xmapP`

Step 6. From relational naturality to the wedge law

The relational naturality law holds for `xmapP`:

`xmapP[A, B, X, Y]: (A => B, X => Y) => P[Y, A] => P[X, B]`

For any types `A`, `A'`, `B`, `B'`, `X`, `X'`, `Y`, `Y'`, and for any relations `p: A <=> A'`, `q: B <=> B'`, `r: X <=> X'`, `s: Y <=> Y'` and for any values `f: A => B`, `f': A' => B'`, `g: X => Y`, `g': X' => Y'`, `v: P[Y, A]`, `v': P[Y', A']` such that `(f, f') in pfun(p, q)` and `(g, g') in pfun(r, s)` and `(v, v') in rmap2_P(s, p)` we will have:

`(xmapP(f, g)(v), xmapP(f', g')(v')) in rmap2_P(r, q)`

We need to get the equation `xmapP(f, id)(t[A]) == xmapP(id, f)(t[B])`

This means we need `rmap2_P(r, q)` to be an *identity* relation

Choose `r = id: X <=> X` and `q = id: B <=> B` (here `X' = X`, `B' = B`) and obtain `rmap2_P(r, q) = id` (of type `P[X, B] <=> P[X, B]`)

- This is a version of the "identity extension lemma" of Reynolds
  - Prove it by induction over the cases in the definition of `rmap2`
  - Also need to prove that `rmap2_F(r, id) = rmap_G(r)` etc.
- For the case `P[X, A] = [Y] => Q[X, A, Y]` we need to assume full parametricity for values in the relation `rmap2_P(id, id)`

Step 6. From relational naturality to the wedge law

`xmapP[A, B, X, Y]: (A => B, X => Y) => P[Y, A] => P[X, B]`

We have: `xmapP(f, g )(v  )  == xmapP(f’, g’)(v’ )`

We need: `xmapP(f, id)(t[A]) == xmapP(id, f )(t[B])`

Choose values as `f’ = id`, `g = id`, `g’ = f`, `v = t[A]`, `v’ = t[B]`

Choose types as `A’ = B = B’ = Y’`, `A = X = X’ = Y`

The relational naturality law of `xmapP` also requires us to have:

- `(f, f’) in pfun(p, q)` – this is `(f, id) in pfun(p, id)` – for any `(x: A, y: B) in p` we need `f(x) == y` – this holds if `p = graph(f)`
- `(g, g’) in pfun(r, s)` – this is `(id, f) in pfun(id, s)` – for any `x: A` we need `(x, f(x)) in s` – this holds if `s = graph(f)`
- `(v, v’) in rmap2_P(s, p)` – this is `(t[A], t[B]) in rmap2_P(s, p)` – when `s = p`, this is the relational naturality law of `t` if formulated for the type signature `t[A]: P[A, A]`
- Need to prove: `rmap2_P(p, p) = rmap_F(p)` where `F[A] = P[A, A]`
  - Prove it by induction over the cases in the definition of `rmap2`
  - Also need to prove that `rmap3(p, p, r) = rmap2(p, r)` etc.

With these choices, the relational naturality law reduces to the wedge law

Step 6. From the wedge law to naturality laws

- For type signatures `G[A] => H[A]` where both `G` and `H` are covariant:

Define `P[X, Y] = G[X] => H[Y]`, take any fully parametric `t[A]: P[A, A]`
The wedge law of `t` is: `fmapP(f)(t[A]) == cmapP(f)(t[B])`
For any `f: A => B`, we have: `fmapP(f)(t[A]) = t[A] andThen fmapH(f)`
and `cmapP(f)(t[B]) = fmapG(f) andThen t[B]`
The wedge law gives: `t[A] andThen fmapH(f) == fmapG(f) andThen t[B]`

- This is exactly the naturality law of `t`

Similarly, the naturality law follows when `G` and `H` are both contravariant

Advanced applications. I. Beyond Yoneda

- Consider the type `[A] => (A => A) => Either[E, A]`
  - The Yoneda identities do not apply to that type signature
  - `P[A] = (A => A) => Either[E, A]` does not have naturality laws
  - The wedge law holds but does not give enough information
- Write the relational naturality law of `p[A]: P[A]`
- For any relation `r: A <=> B`, for any `p: P[A]` and `q: P[B]`, we must have `(p, q) in rmap_P(r)`
- The relational lifting: `rmap_P(r) = rfun(rfun(r, r), rsum(id, r))`
- For any `k: A => A`, `l: B => B`, if `(k, l) in rfun(r, r)` then we must have `(p(k), q(l)) in rsum(id, r)`
- Compute the liftings for `r = ∅` (an **empty relation** of type `A <=> B`)
- `(k, l) in rfun(∅, ∅)` means: for any `a: A`, `b: B`, if `(a, b) in ∅` then `(k(a), k(b)) in ∅`
- This holds for *all* `k` and `l` because there are no `(a, b) in ∅`
- The law becomes: for any `k: A => A`, `l: B => B`, we must have either `p(k) == q(l) == Left(e)` with some `e: E`, or `p(k) == Right(x)`, `k(l) == Right(y)` and `(x, y) in ∅`
- `p` and `q` must be equal constant functions returning `Left(e)`

We have proved: `E` $\cong$ `[A] => (A => A) => Either[E, A]`

Advanced applications. II. Church encoding

- Define recursive types by induction: `T` ≅ `S[T]` with *covariant* `S[_]`
- The isomorphism is given by `fix: S[T] => T` and `unfix: T => S[T]`
- `fix andThen unfix == identity`; `unfix andThen fix == identity`
- Example: `T = List[R]`, so `T` ≅ `S[T]` with `S[A] = Option[(R, A)]`
- Church encoding: `CT = [A] => (S[A] => A) => A` (fully parametric)
- With Scala 2 traits: `trait CT { def fold[A](fix: S[A] => A): A }`

Intuition about the types `CT` and `S[A] => A`: consider `T = List[R]`

- A function of type `S[A] => A` is equivalent to: `{`

```
  case None => (aFixedValue: A)
  case Some((r, a)) => (computeNext(r, a): A)
}
```

- The data in `S[A] => A` is equivalent to the type `(A, (R, A) => A)`
- These are exactly the *argument data* of the `List`'s `foldLeft` function `foldLeft[A]: (S[A] => A) => A`

Intuition: we can create a value of type `CT` only if we have a list (of type `List[R]`) that we can then fold using any "fold data" `(A, (R, A) => A)`

Advanced applications. II. Church encoding

The type `CT` is the least fixpoint of the equation `CT` $\cong$ `S[CT]`

- See Wadler's paper "Recursive types for free" (1990)

```
def fix(sct: S[CT]): [A] => (S[A] => A) => A =
  [A] => saa => saa(sct.map(ct => ct[A](saa)))
def unfix(ct: CT): S[CT] = ct[S[CT]](fmap_S(fix))
```

- Relational law of `ct: CT` is: for any `r: A <=> B`, `p: S[A] => A`, `q: S[B] => B` such that (for any `sa: S[A]`, `sb: S[B]` if `(sa, sb) in rmap_S(r)` then `(p(sa), q(sb)) in r`) we will have `(ct[A](p), ct[B](q)) in r`
  - Choose `r = graph(f)` with an arbitrarily chosen `f: A => B`
  - Then the relational law says: for any `p: S[A] => A`, `q: S[B] => B`, when `p andThen f == fmap_S(f) andThen q` then we will have `f(ct[A](p)) == ct[B](q)`

$$\begin{tikzcd} S[A] \arrow[r, "p"] \arrow[d, "fmap\_S(f)"'] & A \arrow[d, "f"] \\ S[B] \arrow[r, "q"] & B \end{tikzcd}$$

- Can prove the isomorphism directly via that law; instead use a trick

Advanced applications. II. Church encoding

The trick is first to prove the "initial algebra" property:
For any "fold data" `q: S[B] => B` there is a *unique* `c(q): CT => B` such that `fix andThen c(q) == fmap_S(c(q)) andThen q`

$$
\begin{tikzcd}
S[CT] \arrow[r, "\mathtt{fix}"] \arrow[d, "\mathtt{fmap\_S(c(q))}"'] & CT \arrow[d, "c(q)"] \\
S[B] \arrow[r, "q"] & B
\end{tikzcd}
$$

The code: `def c[B](q: S[B] => B)(ct: CT): B = ct[B](q)`

- With that code, `c(q)` satisfies the diagram: for any `sct: S[CT]`,
  `c(q)(fix(sct)) == q(sct.map(c(q))) ?`
  `fix(sct)(q) == q(sct.map(ct => ct(q)))` by definition of `fix`
- Use the law with `p = fix`, `f = c(q)` to get `c(q)(ct(fix)) == ct(q)`
  - Equivalently `ct(fix)(q) == ct(q)` for any `q`, so `ct(fix) == ct`
- Use the law with `p = fix` and *any* `f` to get `f(ct(fix)) == ct(q)`
  - So, any `f: CT => B` satisfies `f(ct) == ct(q)`, so `f == c(q)`

Advanced applications. II. Church encoding

- To prove the isomorphism properties, use another trick:
  Consider `fmap_S(fix): S[S[CT]] => S[CT]` as "fold data" for `S[CT]`

The corresponding unique function `u: CT => S[CT]` is `u(ct) = ct(fmap_S(fix)) = unfix(ct)` and so `unfix` satisfies `fix andThen unfix == fmap_S(unfix) andThen fmap_S(fix) == fmap_S(unfix andThen fix)`
Then consider `fix: S[CT] => CT` as "fold data"; the corresponding unique function of type `CT => CT` is identity since `ct(fix) == ct`
But we also have a function `i = unfix andThen fix` of type `CT => CT` satisfying `fix andThen i == fmap_S(i) andThen fix` because:

```
fix andThen unfix andThen fix ==
  fmap_S(unfix andThen fix) andThen fix
```

By uniqueness, we must have `i == identity`
It follows that `unfix andThen fix == identity` and
`fix andThen unfix = fmap_S(i) = identity`

- We proved the isomorphism `CT` $\cong$ `S[CT]`, so `CT` is a fixpoint
- `CT` is the "least fixpoint": for any other fixpoint `T` $\cong$ `S[T]` there is a unique map `CT => T` that preserves the fixpoint structures of `CT` and `T`

Advanced applications. III. A third-order function

- Define `type F[K] = [A] => ((A => Option[K]) => A) => A`
  - This is an attempt to apply the Church encoding to the recursive type definition `T` $\cong$ `T => Option[K]`
  - That recursive type has the form `T` $\cong$ `F[T]` with a *contravariant* `F[_]`
- We will prove that `F[K]` $\cong$ `Option[K]`
- Define isomorphisms `in` and `out`:

```
def in[A](optK: Option[K]): ((A => Option[K]) => A) => A =
  (p: (A => Option[K]) => A) => p(_ => optK)
def out(h: [A] => ((A => Option[K]) => A) => A): Option[K] =
  h[Option[K]] { t: (Option[K] => Option[K]) => t(None) }
```

- We need to prove that `out(in(optK)) == optK` and `in(out(h)) == h`

  First: `out(in(optK) == out(p => p(_ => optK)) ==`
  `(t => t(None))(_ => optK) == (_ => optK)(None) == optK`

  Second: `in(out(h)) == in(h[Option[K]](t => t(None))) ==`
  `{ p => p(_ => h[Option[K]](t => t(None))) }`

  But we expected `in(out(h)) == h == { p => h(p) }` instead of that!
- Need a law for `h` saying that `h(p)` *must* apply `p` to a constant function:

  `h[A](p) == p(_ => h[Option[K]](t => t(None)))`

Advanced applications. III. A third-order function

- Use the *naturality law* for functions `h[A]: F[A] => A` where `F[A]` is defined by `F[A] = (A => Option[K]) => A` and is covariant in `A`: for any `f: X => Y` and `fx: F[X]`: `h[Y](fx.map(f)) == f(h[X](fx))`
- Here `fx.map(f) == (k: Y => Option[K]) => f(fx(f andThen k))`
- Naturality law: `h[Y](k => f(fx(f andThen k))) == f(h[X](fx))`
- We need a law of the form: `h[A](p) == p(_ => h[Option[K]](...))`
- Choose `X = Option[K]`; `Y = A`; `f = optK => p(_ => optK)`; and `fx: F[Option[K] = (t: Option[K] => Option[K]) => t(None)`
- Then LHS: `f(h[X](fx)) == p(_ => h[Option[K]](t => t(None)))`
- The LHS is exactly what we need (with arbitrary `p: F[A]`)
- But the RHS is: `h[Y](k => f(fx(f andThen k))) ==`
  `h[A](k => f((f andThen k)(None))) ==`
  `h[A](k => p(_ => k(p(_ => None))))`
- Instead of that, we need `h[A](p) == h[A](k => p(k))`

We must find a more powerful law of `h` than the naturality law

Advanced applications. III. A third-order function

- Intuition: a function `h: ((A => Option[K]) => A) => A` must apply its argument `p: (A => Option[K]) => A` to a *constant* function of type `A => Option[K]`. So, we expect `h(p) == h(q)` whenever `p(k) == q(k)` for all constant functions `k`. It will follow that `h(p) = p(k)` for some constant function `k = _ => optK`
- To express this intuition via relations, apply the relational naturality law to an "almost-identity" relation `r(a): A <=> A` defined for a fixed `a: A` by: `(x: A, y: A) in r(a)` means `x == y` or `x == a`
- Lift `r(a)` to the type constructor `((A => Option[K]) => A) => A`
- This gives the relational naturality law of `h`:
  (`h(p) == h(q)` or `h(p) == a`) for all `p`, `q` such that (`p(k) == q(l)` or `p(k) == a`) for all `k`, `l` such that { `k(x) == l(y)` for all `x`, `y` such that `x == y` or `x == a` }
  - Suppose `k(x) == l(y)` for all `x`, `y` such that `x == y` or `x == a`
  - It means that `k(a) == l(y)` for all `y: A`, so `l` is a *constant* function
  - And `k(y) == l(y)` for all `y != a`, so `k` and `l` are the same function
- The relational law of `h` is: (`h(p) == h(q)` or `h(p) == a`) for all `p`, `q` such that (`p(k) == q(k)` or `p(k) == a`) for all constant functions `k`

Advanced applications. III. A third-order function

- Use the relational law of `h` to prove that, for any `p: F[A]`, we have:
  `h[A](p) == h[A](k => p(_ => k(p(_ => None))))`
- The relational law says: `h[A](p) == h[A](q)` for all `p`, `q` such that (...)
- Choose `q = (k: A => Option[K]) => p(_ => k(p(_ => None)))`
- We find that the precondition holds for these `p` and `q`: For any *constant* function `k: A => Option[K]` we actually have `p(k) == q(k)`
  - To verify that: Suppose `k = { _ => optK }`, then:
    `q(k) == p(_ => k(p(_ => None))) == p(_ => optK) == p(k)`
- Since the precondition holds, we obtain `h(p) == h(q)` or `h(p) == a`
- This holds for any chosen `a: A` but the definition of `q` does not depend on `a`, so we can rewrite the law as $\forall h \forall p \forall a$ instead of $\forall h \forall a \forall p$
- When the type `A` has at least two different values, choose `a != h(p)`
- The result is `h(p) == h(q)` as required

This completes the proof of `in(out(h)) == h` and of the type isomorphism
`Option[K]` $\cong$ `[A] => ((A => Option[K]) => A) => A`

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"
  - A few proofs in FP do require the relational naturality law
- More details in the free book — https://github.com/winitzki/sofp