

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: a first example
 - ▶ The Church encoding 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.
```

- ❶ Use `Unit` value (or equivalent type), e.g. `()`, `Nil`, `None`
- ❷ Use bound variable (a given argument of the function)
- ❸ Create a function: `{ x => expr(x) }`
- ❹ Use a function: `f(x)`
- ❺ Create a product: `(a, b)`
- ❻ Use a product: `p._1` (or via pattern matching)
- ❼ Create a co-product: `Left[A, B](x)`
- ❽ Use a co-product: `{ case ... => ... }` (pattern matching)
- ❾ Use a recursive call: e.g., `fmap(f)(tail)` within the code of `fmap`

Naturality laws require map

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

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

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

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

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

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

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

Using naturality laws: the Yoneda identities

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

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

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

Isomorphism maps:

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

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

Proofs of isomorphism:

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

The other direction:

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

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

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

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

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

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

Example applications of the Yoneda identities

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

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

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

```
def flatMap[A, B]: 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], A)
F[A] = Fix[G, A] satisfies the type equation F[A] = G[F[A], A]
```

Step 1. Deriving fmap from types

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

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

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

Step 1. Deriving cmap from types

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

Step 1. Detect covariance and contravariance from types

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

$$\text{type } F[A, B] = (A \Rightarrow \text{Either}[A, B], A \Rightarrow (B \Rightarrow A) \Rightarrow (A, B))$$

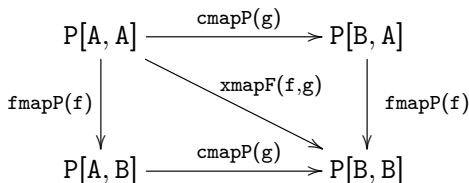
- + + - + - + +

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

Step 1. “Invariant” type constructors. Profunctors

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

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



Step 1. Verifying the functor laws

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

- Identity law:

`fmap(identity) = identity`

`cmap(identity) = identity`

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

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

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

- Go through each case and prove that the laws hold

- ▶ Proofs by induction on the type structure

Step 1. Functor laws: composition law for tuples

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

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

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

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

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

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

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

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

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

Step 1. Functor laws: composition law for function types

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

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

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

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

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

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

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

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

Step 1. Functor laws: composition law for recursive types

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

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

For any `f: A => B` and `g: B => C`:

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

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

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

- Example: `foldLeft` with respect to type parameter A

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

Step 2. Motivating relational parametricity. III. The solution

The difficulty is resolved using three nontrivial ideas:

- 1 Generalize functions $f: A \Rightarrow B$ to binary relations $r: A \Leftrightarrow B$
 - ▶ The **graph** relation: (a, b) in $\text{graph}(f)$ means $f(a) == b$
 - ▶ Relations are more general than functions, can be many-to-many
 - ▶ Instead of $f(a) == b$, we will write (a, b) in r
- 2 It is *a*lways possible to lift $r: A \Leftrightarrow B$ to $\text{rmap}(r): F[A] \Leftrightarrow F[B]$
- 3 Reformulate the naturality law of t via relations: for any $r: A \Leftrightarrow B$,

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

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

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

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

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

Step 2. Formulating naturality laws via relations

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

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

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

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

when (fa, fb) in $rmap_F(r)$ then $(t(fa), t(fb))$ in $rmap_G(r)$

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

Step 3. Definition of relations. Examples

In the terminology of relational databases:

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

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

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

Relations can be many-to-many while functions $A \Rightarrow B$ are many-to-one
A function $f: A \Rightarrow B$ generates the **graph** relation $\text{graph}(f): A \Leftrightarrow B$

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

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

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

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

Step 3. Relational combinators: pprod, psum, pfun, rev

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

- Pair product: $\text{pprod}(r, s)$ of type $(A, X) \Leftrightarrow (B, Y)$

$((a, x), (b, y))$ in $\text{pprod}(r, s)$ means (a, b) in r and (x, y) in s

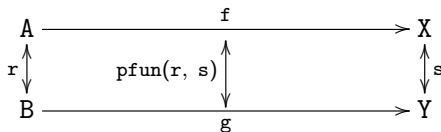
- Pair co-product: $\text{psum}(r, s)$ of type $\text{Either}[A, X] \Leftrightarrow \text{Either}[B, Y]$

$(\text{Left}(a), \text{Left}(b))$ in $\text{psum}(r, s)$ if (a, b) in r

$(\text{Right}(x), \text{Right}(y))$ in $\text{psum}(r, s)$ if (x, y) in s

- Pair function mapper: $\text{pfun}(r, s)$ of type $(A \Rightarrow X) \Leftrightarrow (B \Rightarrow Y)$

(f, g) in $\text{pfun}(r, s)$ means when (a, b) in r then $(f(a), g(b))$ in s



- Reverse: $\text{rev}(r)$ has type $B \Leftrightarrow A$

(b, a) in $\text{rev}(r)$ means the same as (a, b) in r

Step 4. The relational lifting (rmap)

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

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 $F[A] = \text{Unit}$ or $F[A] = K$ (a fixed type): $\text{rmap_F}(r) = \text{id}$
- ② $F[A] = A$: define $\text{rmap_F}(r) = r$
- ③ $F[A] = (G[A], H[A])$: $\text{rmap_F}(r) = \text{pprod}(\text{rmap_G}(r), \text{rmap_H}(r))$
- ④ $F[A] = \text{Either}[G[A], H[A]]$:
 $\text{rmap_F}(r) = \text{psum}(\text{rmap_G}(r), \text{rmap_H}(r))$
- ⑤ $F[A] = G[A] \Rightarrow H[A]$: $\text{rmap_F}(r) = \text{pfun}(\text{rmap_G}(r), \text{rmap_H}(r))$
- ⑥ Recursive type: $F[A] = G[A, F[A]]$:
 $\text{rmap_F}(r) = \text{rmap2_G}(r, \text{rmap_F}(r))$ – recursive definition of rmap_F
- ⑦ Universally quantified type: $F[A] = [X] \Rightarrow G[A, X]$:
 $\text{rmap_F}(r) = \forall(X, Y). \forall(s: X \Leftrightarrow Y). \text{rmap2_G}(r, s)$
 - 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 $r: A \leq B$, $s: P \leq Q$, we define $\text{rmap2_F}(r, s): F[A, P] \leq F[B, Q]$ by induction on the type $F[A, P]$

- ① $F[A, P] = K$ (a fixed type): $\text{rmap2_F}(r, s) = \text{id}$
- ② If $F[A, P] = A$ then $\text{rmap2_F}(r, s) = r$
If $F[A, P] = P$ then $\text{rmap2_F}(r, s) = s$
- ③ $F[A, P] = (G[A, P], H[A, P])$:
 $\text{rmap2_F}(r, s) = \text{pprod}(\text{rmap2_G}(r, s), \text{rmap2_H}(r, s))$
- ④ $F[A, P] = \text{Either}[G[A, P], H[A, P]]$:
 $\text{rmap2_F}(r, s) = \text{psum}(\text{rmap2_G}(r, s), \text{rmap2_H}(r, s))$
- ⑤ $F[A, P] = G[A, P] \Rightarrow H[A, P]$:
 $\text{rmap2_F}(r, s) = \text{pfun}(\text{rmap2_G}(r, s), \text{rmap2_H}(r, s))$
- ⑥ Recursive type: $F[A, P] = G[A, P, F[A, P]]$:
 $\text{rmap2_F}(r, s) = \text{rmap3_G}(r, s, \text{rmap2_F}(r, s))$
- ⑦ Universally quantified type: $F[A, P] = [X] \Rightarrow G[A, P, X]$:
 $\text{rmap2_F}(r, s) = \forall(X, Y). \forall(t: X \leq Y). \text{rmap3_G}(r, s, t)$
 - 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 $P[A] = R \Rightarrow (A, A)$ where R is a fixed type

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

Case 5: $P[A] = G[A] \Rightarrow H[A]$ 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 $\forall x: R, y: R$, 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 $r = \text{graph}(f)$ 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 $q == \text{fmap_P}(f)(p)$ or (p, q) in `graph(fmap_P(f))`

Step 4. Example: rmap for function types

Compare `fmap` and `rmap` for function types: $(F[A] = G[A] \Rightarrow H[A])$

To rewrite `fmap_F` via relations, introduce intermediate arguments

Choose any values $p: G[A] \Rightarrow H[A]$ and $f: A \Rightarrow B$

Define $q = \text{fmap_F}(f)(p) = (gb: G[B]) \Rightarrow \text{fmap_H}(f)(p(\text{cmap_G}(f)(gb)))$

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

for all $gb: G[B]$ we must have $q(gb) = \text{fmap_H}(f)(p(\text{cmap_G}(f)(gb)))$

Define $ga: G[A] = \text{cmap_G}(f)(gb)$, then: $q(gb) = \text{fmap_H}(f)(p(ga))$

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

So, the relational formulation of `fmap_F` is:

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

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

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

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

$\text{graph}(\text{fmap_F}(f))$ by $\text{rmap_F}(r)$; $\text{rev}(\text{graph}(\text{cmap_G}(f)))$ by $\text{rmap_G}(r)$

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

$(ga, gb) \text{ in } \text{rmap_G}(r)$ then $(p(ga), q(gb)) \text{ in } \text{rmap_H}(r)$

This is the same as $(p, q) \text{ in } \text{pfun}(\text{rmap_G}(r), \text{rmap_H}(r))$

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 $G[A] = A \Rightarrow A$ 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 $H[A] = (A \Rightarrow A) \Rightarrow A$ 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 $[A] \Rightarrow A \Rightarrow A$

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

- The value `t` has type $[A] \Rightarrow A \Rightarrow A$
- Denote $P[A] = A \Rightarrow A$

The relational naturality law says:

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

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

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

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

$\text{rmap_P}(r) = \text{pfun}(r, r)$

- $(t[A], t[B])$ in $\text{pfun}(r, r)$ means:
for any $a: A, b: B$, if (a, b) in r then $(t(a), t(b))$ in r

Trick: choose $r: A \Leftrightarrow A$ such that (a, b) in r only if $a == b == a_0$

- Whenever $a == b == a_0$ then $t(a) == t(b) == a_0$
- So, $t(a_0) == a_0$ for any fixed $a_0: A$
 - It means that `t` must be an identity function

Step 5. Preparing to prove the relational naturality law

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

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

It is convenient to prove the relational law when t has a free variable:

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

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

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

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

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

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

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

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

The inductive assumption is that the theorem holds for all sub-expressions and for the free variable z

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

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 $\text{rmap_P}(r) == \text{id}$ and $(c, c) \text{ in id}$ holds
- 2 Use argument: $t[A](z) = z$ where z and $t[A]$ have type $P[A]$:
 - If $(z1, z2) \text{ in rmap_P}(r)$ then $(t(z1), t(z2)) \text{ in rmap_P}(r)$
- 3 Create function: $t(z) = h \Rightarrow s(z, h)$ where we assume $h: H[A]$ and $s(z, h): S[A]$
 - If $(z1, z2) \text{ in rmap_P}(r)$ and $(h1, h2) \text{ in rmap_H}(r)$ then $(s(z1, h1), s(z2, h2)) \text{ in rmap_S}(r)$
 - This is the same as the inductive assumption for $s(z, h)$
- 4 Use function: $t(z) = g(z)(h(z))$ where $g(z): H[A] \Rightarrow Q[A]$ and $h(z): H[A]$ are sub-expressions:
 - If $(z1, z2) \text{ in rmap_P}(r)$ then the inductive assumption says:
 $(h(z1), h(z2)) \text{ in rmap_H}(r)$
 - If $(h1, h2) \text{ in rmap_H}(r)$ then the inductive assumption says:
 $(g(h1), g(h2)) \text{ in rmap_Q}(r)$
 - Therefore $(t[A](z1), t[B](z2)) \text{ in rmap_Q}(r)$

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

If $s(z1) = \text{Left}(x1)$, $s(z2) = \text{Left}(x2)$ then $(x1, x2)$ in $\text{rmap_G}(r)$ and $(u(z1)(x1), u(z2)(x2))$ in $\text{rmap_Q}(r)$

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

9 Recursive call: $t(z) = f(z)(t(z))$ where $f(z): Q[A] \Rightarrow Q[A]$

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

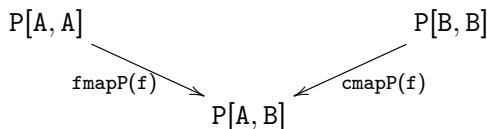
Based on Bartosz Milewski's blog post: [The Free Theorem for Ends](#) (2017)

Given:

- a function $f: A \Rightarrow B$
- a fully parametric profunctor $P[X, Y]$ with methods `cmapP` and `fmapP`:
 - ▶ $\text{cmapP}[X, Y, B]: (X \Rightarrow Y) \Rightarrow P[Y, B] \Rightarrow P[X, B]$
 - ▶ $\text{fmapP}[X, A, B]: (A \Rightarrow B) \Rightarrow P[X, A] \Rightarrow P[X, B]$
- a fully parametric value (without free variables) $t: [A] \Rightarrow P[A, A]$

Then we will prove that the wedge law holds:

- $\text{fmapP}[A, A, B](f)(t[A]) == \text{cmapP}[A, B, B](f)(t[B])$



- Expressed via $\text{xmapP}(f, g) = \text{cmapP}(g) \text{ andThen } \text{fmapP}(f)$:
 $\text{xmapP}(f, \text{id})(t[A]) == \text{xmapP}(\text{id}, f)(t[B])$
 - ▶ 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

$\text{xmapP}[A, B, X, Y]: (A \Rightarrow B, X \Rightarrow Y) \Rightarrow P[Y, A] \Rightarrow P[X, B]$

We have: $\text{xmapP}(f, g)(v) == \text{xmapP}(f', g')(v')$

We need: $\text{xmapP}(f, \text{id})(t[A]) == \text{xmapP}(\text{id}, f)(t[B])$

Choose values as $f' = \text{id}, g = \text{id}, g' = f, v = t[A], v' = t[B]$

Choose types as $A' = B = B' = Y', A = X = X' = Y$

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

- $(f, f') \text{ in } \text{pfun}(p, q)$ – this is $(f, \text{id}) \text{ in } \text{pfun}(p, \text{id})$ – for any $(x: A, y: B) \text{ in } p$ we need $f(x) == y$ – this holds if $p = \text{graph}(f)$
- $(g, g') \text{ in } \text{pfun}(r, s)$ – this is $(\text{id}, f) \text{ in } \text{pfun}(\text{id}, s)$ – for any $x: A$ we need $(x, f(x)) \text{ in } s$ – this holds if $s = \text{graph}(f)$
- $(v, v') \text{ in } \text{rmap2_P}(s, p)$ – this is $(t[A], t[B]) \text{ in } \text{rmap2_P}(s, p)$ – 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: $\text{rmap2_P}(p, p) = \text{rmap_F}(p)$ where $F[A] = P[A, A]$
 - ▶ Prove it by induction over the cases in the definition of rmap2
 - ▶ Also need to prove that $\text{rmap3}(p, p, r) = \text{rmap2}(p, r)$ 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 $G[A] \Rightarrow H[A]$ where both G and H are covariant:

Define $P[X, Y] = G[X] \Rightarrow H[Y]$, 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 $f: A \Rightarrow B$, we have: $fmapP(f)(t[A]) = t[A] \text{ andThen } fmapH(f)$
and $cmapP(f)(t[B]) = fmapG(f) \text{ andThen } t[B]$

The wedge law gives: $t[A] \text{ andThen } fmapH(f) == fmapG(f) \text{ andThen } t[B]$

- 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 $[A] \Rightarrow (A \Rightarrow A) \Rightarrow \text{Either}[E, A]$
 - ▶ The Yoneda identities do not apply to that type signature
 - ▶ $P[A] = (A \Rightarrow A) \Rightarrow \text{Either}[E, A]$ 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 $r: A \Leftrightarrow B$, for any $p: P[A]$ and $q: P[B]$, we must have $(p, q) \text{ in } \text{rmap_P}(r)$
- The relational lifting: $\text{rmap_P}(r) = \text{rfun}(\text{rfun}(r, r), \text{rsum}(\text{id}, r))$
- For any $k: A \Rightarrow A$, $l: B \Rightarrow B$, if $(k, l) \text{ in } \text{rfun}(r, r)$ then we must have $(p(k), q(l)) \text{ in } \text{rsum}(\text{id}, r)$
- Compute the liftings for $r = \emptyset$ (an **empty relation** of type $A \Leftrightarrow B$)
- $(k, l) \text{ in } \text{rfun}(\emptyset, \emptyset)$ means: for any $a: A$, $b: B$, if $(a, b) \text{ in } \emptyset$ then $(k(a), k(b)) \text{ in } \emptyset$
- This holds for *all* k and l because there are no $(a, b) \text{ in } \emptyset$
- The law becomes: for any $k: A \Rightarrow A$, $l: B \Rightarrow B$, we must have either $p(k) == q(l) == \text{Left}(e)$ with some $e: E$, or $p(k) == \text{Right}(x)$, $k(l) == \text{Right}(y)$ and $(x, y) \text{ in } \emptyset$
- p and q must be equal constant functions returning $\text{Left}(e)$

We have proved: $E \cong [A] \Rightarrow (A \Rightarrow A) \Rightarrow \text{Either}[E, A]$

Advanced applications. II. Church encoding

- Define recursive types by induction: $T \cong S[T]$ 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: $T = \text{List}[R]$, so $T \cong S[T]$ with $S[A] = \text{Option}[(R, A)]$
- Church encoding: $\text{CT} = [A] \Rightarrow (S[A] \Rightarrow A) \Rightarrow A$ (fully parametric)
- With Scala 2 traits: `trait CT { def fold[A](fix: S[A] => A): A }`

Intuition about the types `CT` and $S[A] \Rightarrow A$: consider $T = \text{List}[R]$

- A function of type $S[A] \Rightarrow A$ is equivalent to:

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

- The data in $S[A] \Rightarrow A$ is equivalent to the type $(A, (R, A) \Rightarrow A)$
- 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” $(A, (R, A) \Rightarrow A)$

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 $r: A \leq B$, $p: S[A] \Rightarrow A$, $q: S[B] \Rightarrow B$ 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 $r = \text{graph}(f)$ with an arbitrarily chosen $f: A \Rightarrow B$
 - Then the relational law says: for any $p: S[A] \Rightarrow A$, $q: S[B] \Rightarrow B$, when p and $f == fmap_S(f)$ and q then we will have $f(ct[A](p)) == ct[B](q)$

$$\begin{array}{ccc} S[A] & \xrightarrow{p} & A \\ \text{fmap_S}(f) \downarrow & & \downarrow f \\ S[B] & \xrightarrow{q} & B \end{array}$$

- 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” $q: S[B] \Rightarrow B$ there is a *unique* $c(q): CT \Rightarrow B$ such that `fix andThen c(q) == fmap_S(c(q)) andThen q`

$$\begin{array}{ccc} S[CT] & \xrightarrow{\text{fix}} & CT \\ \text{fmap_S}(c(q)) \downarrow & & \downarrow c(q) \\ S[B] & \xrightarrow{q} & B \end{array}$$

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 $sct: S[CT]$, $c(q)(\text{fix}(sct)) == q(sct.\text{map}(c(q)))$?
 $\text{fix}(sct)(q) == q(sct.\text{map}(ct \Rightarrow ct(q)))$ by definition of `fix`
- Use the law with $p = \text{fix}$, $f = c(q)$ to get $c(q)(ct(\text{fix})) == ct(q)$
 - ▶ Equivalently $ct(\text{fix})(q) == ct(q)$ for any q , so $ct(\text{fix}) == ct$
- Use the law with $p = \text{fix}$ and *any* f to get $f(ct(\text{fix})) == ct(q)$
 - ▶ So, any $f: CT \Rightarrow B$ satisfies $f(ct) == ct(q)$, so $f == c(q)$

Advanced applications. II. Church encoding

- To prove the isomorphism properties, use another trick:

Consider $\text{fmap_S}(\text{fix}): S[S[CT]] \Rightarrow S[CT]$ as “fold data” for $S[CT]$

The corresponding unique function $u: CT \Rightarrow S[CT]$ is $u(ct) = ct(\text{fmap_S}(\text{fix})) = \text{unfix}(ct)$ and so unfix satisfies $\text{fix andThen unfix} == \text{fmap_S}(\text{unfix}) \text{ andThen fmap_S}(\text{fix}) == \text{fmap_S}(\text{unfix andThen fix})$. Then consider $\text{fix}: S[CT] \Rightarrow CT$ as “fold data”; the corresponding unique function of type $CT \Rightarrow CT$ is identity since $ct(\text{fix}) == ct$

But we also have a function $i = \text{unfix andThen fix}$ of type $CT \Rightarrow CT$ satisfying $\text{fix andThen i} == \text{fmap_S}(i) \text{ andThen fix}$ because:

$\text{fix andThen unfix andThen fix} ==$
 $\text{fmap_S}(\text{unfix andThen fix}) \text{ andThen fix}$

By uniqueness, we must have $i == \text{identity}$

It follows that $\text{unfix andThen fix} == \text{identity}$ and
 $\text{fix andThen unfix} = \text{fmap_S}(i) = \text{identity}$

- 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 $CT \Rightarrow T$ 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 $T \cong T \Rightarrow \text{Option}[K]$
 - That recursive type has the form $T \cong S[T]$ with a *contravariant* $S[_]$
- We will prove that $F[K] \cong \text{Option}[K]$

- 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 $h[A]: G[A] \Rightarrow A$ where $G[A]$ is defined by $G[A] = (A \Rightarrow \text{Option}[K]) \Rightarrow A$ and is covariant in A : for any $f: X \Rightarrow Y$ and $gx: G[X]: h[Y](gx.\text{map}(f)) == f(h[X](gx))$
- Here $gx.\text{map}(f) == (k: Y \Rightarrow \text{Option}[K]) \Rightarrow f(gx(f \text{ andThen } k))$
- Naturality law: $h[Y](k \Rightarrow f(gx(f \text{ andThen } k))) == f(h[X](gx))$
- We need a law of the form: $h[A](p) == p(_ \Rightarrow h[\text{Option}[K]](...))$
- Choose $X = \text{Option}[K]; Y = A; f = \text{optK} \Rightarrow p(_ \Rightarrow \text{optK});$ and $gx: G[\text{Option}[K] = (t: \text{Option}[K] \Rightarrow \text{Option}[K]) \Rightarrow t(\text{None})$
- Then LHS: $f(h[X](gx)) == p(_ \Rightarrow h[\text{Option}[K]](t \Rightarrow t(\text{None})))$
- The LHS is exactly what we need (with arbitrary $p: G[A]$)
- But the RHS is: $h[Y](k \Rightarrow f(gx(f \text{ andThen } k))) ==$
 $h[A](k \Rightarrow f((f \text{ andThen } k)(\text{None}))) ==$
 $h[A](k \Rightarrow p(_ \Rightarrow k(p(_ \Rightarrow \text{None}))))$
- Instead of that, we need $h[A](p) == h[A](k \Rightarrow p(k))$

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

Advanced applications. III. A third-order function

- Intuition: a function $h: ((A \Rightarrow \text{Option}[K]) \Rightarrow A) \Rightarrow A$ must apply its argument $p: (A \Rightarrow \text{Option}[K]) \Rightarrow A$ to a *constant* function of type $A \Rightarrow \text{Option}[K]$. 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 $k = _ \Rightarrow \text{optK}$
- To express this intuition via relations, apply the relational naturality law to an “almost-identity” relation $r(a): A \Leftrightarrow A$ defined for a fixed $a: A$ by: $(x: A, y: A) \text{ in } r(a) \text{ means } x == y \text{ or } x == a$
- Lift $r(a)$ to the type constructor $((A \Rightarrow \text{Option}[K]) \Rightarrow A) \Rightarrow A$
- This gives the relational naturality law of h :
 $(h(p) == h(q) \text{ or } h(p) == a)$ for all p, q such that $(p(k) == q(l) \text{ or } p(k) == a)$ for all k, l such that $\{ k(x) == l(y) \text{ for all } x, y \text{ such that } x == y \text{ or } x == a \}$
 - ▶ 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 $y \neq a$, so k and l are the same function
- The relational law of h is: $(h(p) == h(q) \text{ or } h(p) == a)$ for all p, q such that $(p(k) == q(k) \text{ or } p(k) == a)$ for all constant functions k

Advanced applications. III. A third-order function

- Use the relational law of h to prove that, for any $p: G[A]$, we have:
 $h[A](p) == h[A](k \Rightarrow p(_ \Rightarrow k(p(_ \Rightarrow None))))$
- The relational law says: $h[A](p) == h[A](q)$ for all p, q such that (...)
- Choose $q = (k: A \Rightarrow Option[K]) \Rightarrow p(_ \Rightarrow k(p(_ \Rightarrow None)))$
- We find that the precondition holds for these p and q : For any *constant* function $k: A \Rightarrow Option[K]$ we actually have $p(k) == q(k)$
 - ▶ To verify that: Suppose $k = \{ _ \Rightarrow optK \}$, then:
 $q(k) == p(_ \Rightarrow k(p(_ \Rightarrow None))) == p(_ \Rightarrow optK) == p(k)$
- 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 $a \neq h(p)$
- The result is $h(p) == h(q)$ as required

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

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>

