# Notes on Category Theory in Scala 3 (Dotty)

December 23, 2019



## 0. Introduction

Learning math can be hard for several reasons, but one of them is that the language used by authors tends to be optimized for specialists:

- symbols and identifiers are heavily overloaded
- math writers abhor boilerplate
- a lot of the context of a given expression is assumed to be available to the reader, etc.

In addition to that, most math is written in plain paper or paper-like mediums such as PDF.

(If you look at this practice from the perspective of a programmer this is a bit strange: it's similar as if programs were written primarily using the plain text editor "Notepad", and without any help from the type checker)

But probably the main difficulty is just abstraction itself. One can spend a lot of time trying to come up with a good mental representation of new concepts.

As it turns out, we can use Scala (or other programming languages) to encode many mathematical definitions and ideas, and get some immediate wins:

- The typechecker can point out many errors.
- We get access to all the tools available when using a modern IDE, such as code completion, navigate to definition, inline documentation, auto-generated code, etc.
- A lot of the ambiguity is removed since we're forced to write every single definition in *painful* detail.

And crucially, we can re-use our existing programming intuition to help develop a purely abstract math intuition.

Of course we'll have to make some compromises in generality, but the reality is that most people don't learn (or do) math at the maximum possible level of generality (in part because that level is not fixed: it normally increases over time). All this is to say that for the ideas that *can* be expressed in our programming language, we *are* doing real math.

Now while Scala can be used to *automate* some theorem proofs (a topic for a different post), it is not a theorem prover such as Lean, or Agda. If you're seriously interested in doing math on your computer at the maximum generality then an automated theorem prover is the best tool.

---

In this article we'll explore how we can encode several definitions from the book Category Theory and Applications by Marco Grandis and translate them into Scala 3 (Dotty).

> *Quoted text like this we'll be reserved for quotes from this book.*

We'll end up by stating and proving the Yoneda lemma and explaining the (in) famous phrase "Monads are just monoids in the category of endofunctors".

Note 1: This article focuses mostly on the process of encoding Algebraic

concepts into Scala, and is not meant to be a full tutorial on Categories.

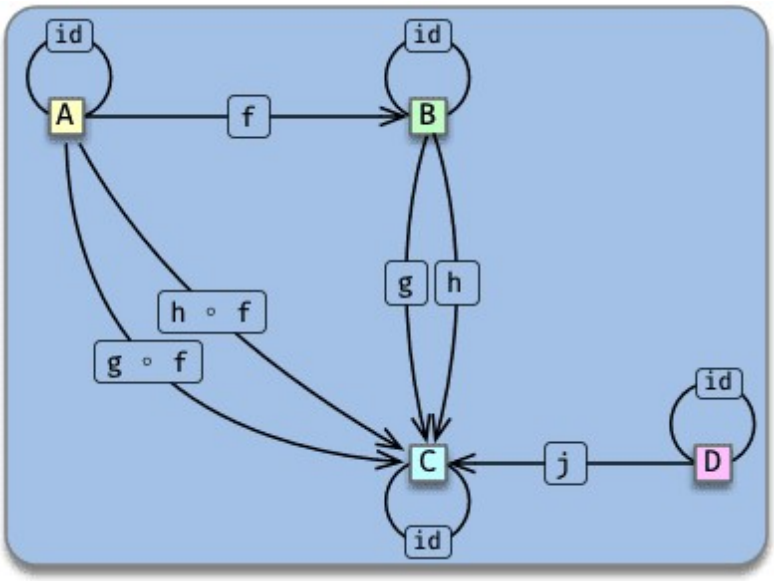
Note 2: If you're not familiar with type functions then I recommend perusing Illustrated guide to Types, Sets and Values.

Audience: Math beginner-ish, Scala intermediate.

Dotty version used: `0.20.0-RC1`

# 1. Categories

Informally, a category is a special kind of directed graph with an operation similar to path concatenation defined on the edges.




> A **Category** $C$ *consists of the following data:*
>
> 1. *a set* $O$ *whose elements are called objects of* $C$*.*
> 2. *for every pair* $(X, Y)$ *of objects, a set* $\mathrm{Hom}(X, Y)$ *(called a hom-set), whose elements are called* **morphisms** *(or maps, or* **arrows***) of* $C$ *from* $X$ *to* $Y$ *and denoted as* $f : X \to Y$*,*
>
> *The set* $\mathrm{Hom}(X, Y)$ *is also written as* $C(X, Y)$
>
> 3. *for every triple* $X, Y, Z$ *of objects on* $C$*, a mapping or composition*
>
> $$\begin{array}{ccc} \mathrm{Hom}(X,Y) \times \mathrm{Hom}(Y,Z) & \to & \mathrm{Hom}(X,Z) \\ (f,g) & \mapsto & g \circ f \end{array}$$
>
> *These data must satisfy the following* **axioms**
>
> 1. *Associativity. Given three consecutive arrows* $f : X \to Y$*,* $g : Y \to Z$*, and* $h : Z \to W$*, the equation* $h \circ (g \circ f) = (h \circ g) \circ f$ *holds,*
> 2. *Identities. Given an object* $X$*, there exists and endomorphism* $e : X \to X$ *which acts as an identity whenever composition makes sense; in other words if* $f : X' \to X$ *and* $g : X \to X''$*, one has* $e \circ f = f$ *and* $g \circ e = g$*.*

> *e is called the identity of $X$ and written as $1_X$ or $idX$.*

## 1.1 Categories over the set of all types $\mathbb{T}$

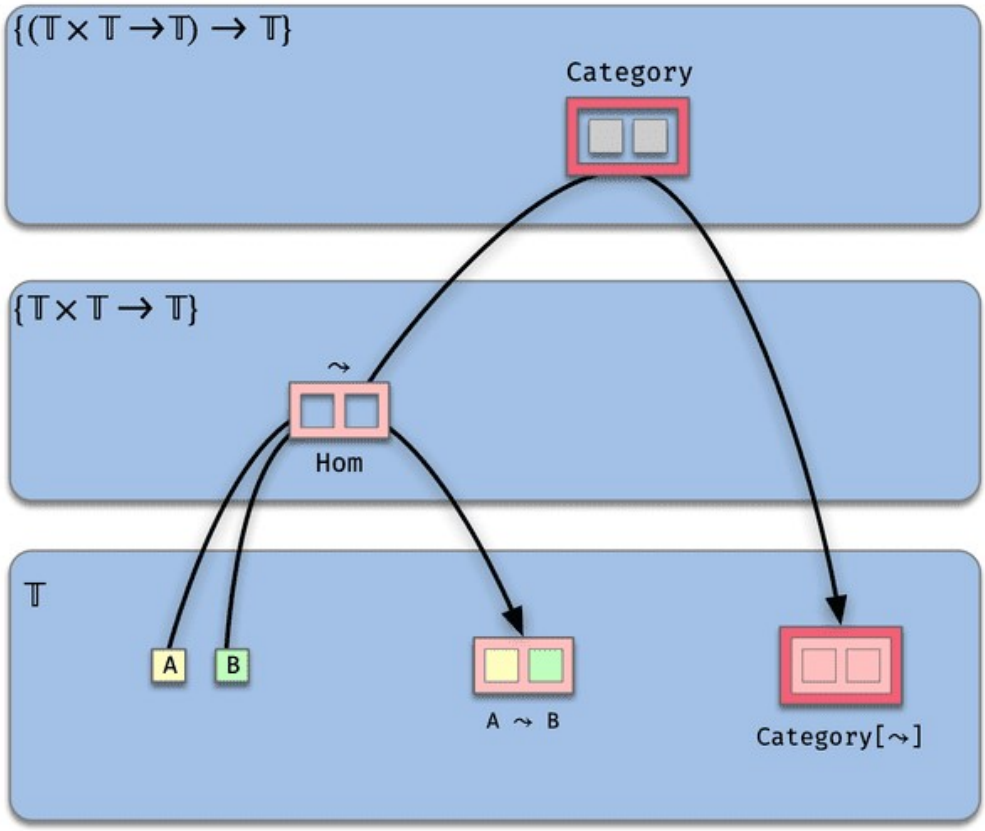
Our approach will be to represents objects as types (i.e. elements of the set $\mathbb{T}$ of all types), and hom-sets as type functions $\mathbb{T} \times \mathbb{T} \to \mathbb{T}$.

```
import discipline._

trait Category[Hom[_, _]]
  type ~> = Hom
  def id[A]: A ~> A

  def [A, B, C] (g: B ~> C) ∘ (f: A ~> B): A ~> C


class CategoryLaws[~>[_, _]](given C: Category[~>])
  def associativity[A, B, C, D](
    f: A ~> B,
    g: B ~> C,
    h: C ~> D
  ) =
    h ∘ (g ∘ f) <-> (h ∘ g) ∘ f

  // show the return type just this time
  def identityRight[A, B](f: A ~> B): IsEq[A ~> B] =
    f ∘ C.id[A] <-> f

  def identityLeft[A, B](f: A ~> B) =
    (C.id[B] ∘ f) <-> f
```

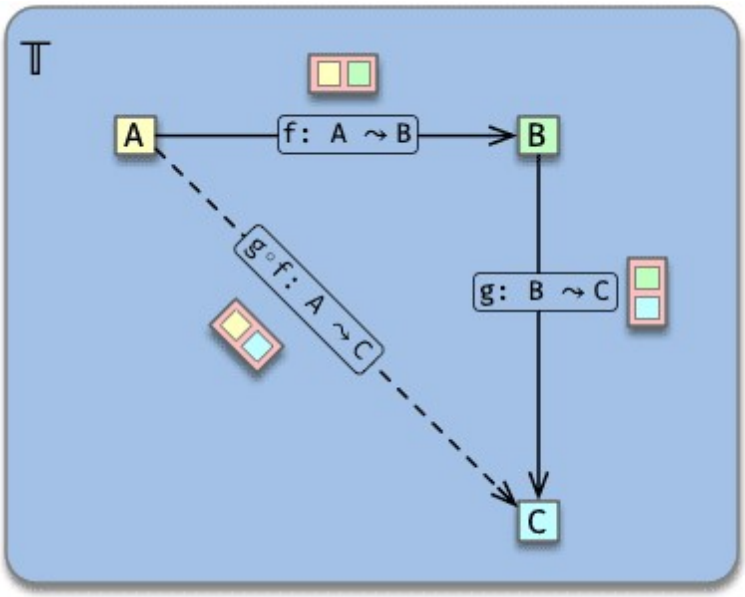
Laws are usually expressed as functions that return an instance of the `case class IsEq(lhs, rhs)`, (the operator `<->` is provided by Cats, it simply creates a new `IsEq`), which is then used in conjunction with the library discipline and Scalacheck to generate random values and try to find *counterexamples*.

(Here's a fully developed example using these laws)

Some of the types involved in this definition are:




with composition:




Notation can get heavy pretty fast, so the arrow notation `A ⇝ B` is very convenient as long as we remember that arrows are not necessarily functions (and as a reminder of this I decided to use `Hom[_, _]` for the parameter name).

In particular notice that there's no evaluation function defined on arrows, which means that everything needs to be done "point-free" when using the `Category` trait.

## Example: Pure Scala functions and types

Probably one of the most elementary categories in math is the **Set** category of sets and functions.

Arguably the **Scala** category (with types as objects and pure functions as arrows) defined below plays a similar role, in the sense that it is the "base" language on top of which everything else is built.

```
type Scala = Function

given Scala: Category[Scala]
  def id[A] = x => x
  def [A, B, C] (g: B => C) ∘ (f: A => B) =
    g compose f

// This satisfies the category laws but
// only for pure functions :)
```

From now on we'll refer to this category simply as **Scala.**

(Dotty feature: given instances).

## Implicit or explicit?

There's no requirement to make the **Scala** category instance an implicit value (I mean... *given*). But in most cases we'll be using only one category instance `C` for a given hom function `Hom[_,_]`. In this situation it's very convenient to be able to just pass the hom function and have the corresponding category instance be looked up by the compiler (and we can always use an explicit instance when needed).

Here's a summary of how we translated the definition into Scala:

| **Math** | **Scala** |
|---|---|
| Algebraic definition | Trait with some abstract members |
| Axioms / Laws | Test suite |
| Concrete Category | Lawful implementation of the Category trait |
| Objects $O$ of the category $C$ | All types (for now) |
| $\mathrm{Hom}(X, Y)$ (morphisms between $X$ and $Y$) | Values of type `Hom[X, Y]` |
| Function $\mathrm{Hom} : O \times O \to \mathrm{Sets}$ | Type Function `Hom[_,_]` |
| Composition operator | Family of binary functions `∘[A, B, C]`, one for each triple $A, B, C \in \mathbb{T}$ |
| Identity morphism $1_X$ for each object $X$ | A value of type `id[X]` for each type `X` |

| Math | Scala |
| --- | --- |
| Function family $X \mapsto 1_X \in \mathrm{Hom}(X, X)$ | Polymorphic function `[X] =>> id[X]: Hom[X,X]` |

The textbook definition allows for the objects $O$ of the category $C$ to be an *arbitrary* set, whereas in our `Category` trait we have no way to restrict the arguments for `Hom` , which means that instances of this trait have a fixed set of objects `o`: the set of all types!

In order to create more general categories we need to find a way to restrict objects to be members of different sets of types.

There are two main ways in which we can specify subsets of types in Scala:

1. Via subtyping
2. Via type classes

Long story short, we'll have to create slightly different `Category` definitions to express different kind of constraints over the arguments of `Hom`. We'll use our original definition when possible though, because other definitions add some boilerplate.

## Type arguments vs type members

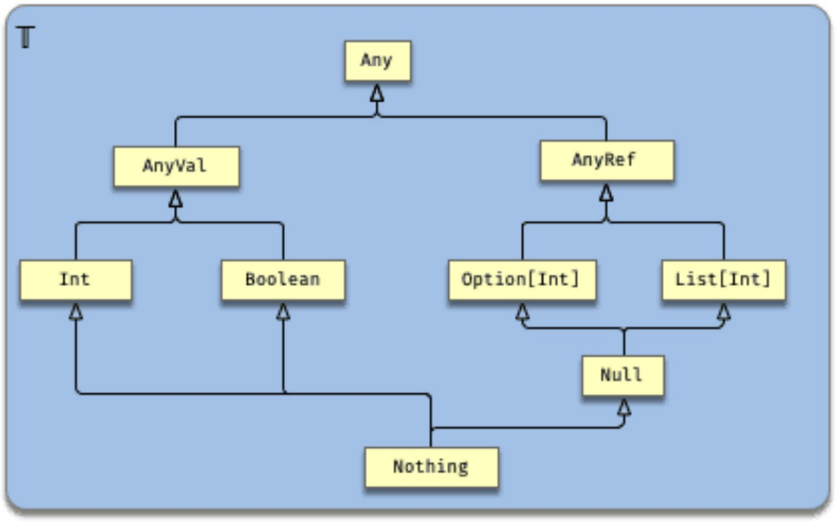
Another encoding choice would be to use an abstract type member like so:

```
trait Category
  type Hom[A, B]
  def id[A]: Hom[A, A]
  def [A, B, C] (g: Hom[B, C]) ∘ (f: Hom[A, B]): Hom[A, C]
```

I find this approach more cumbersome to deal, mainly due to the fact that it can be easy to get the types wrong, and harder to fix. Also you end up having to use things like the `Aux[_]` pattern frequently to reduce boilerplate and prevent errors.

## Example: Category of types and subtyping relationships

We know that types in Scala form a lattice under subtyping:




A natural category to consider would be one with all types as objects and a single morphism between `A` and `B` if `A <: B` (and no morphisms otherwise).

(This is called the Liskov category in scalaz, but I haven't found any other reference to the origin of this name)

Scala provides a data type (`<:<[A, B]`) that encodes the fact that type `A` is a subtype of `B`

```
given Subtypes: Category[<:<]
  def id[A] = <:<.refl
  def [A, B, C] (g: B <:< C) ∘ (f: A <:< B): A <:< C =
    g compose f
```

For each pair of types `A <: B` there is only one arrow, the unique value of type `A <:< B` that can be obtained using `summon` (and none otherwise):

```
trait A
trait B extends A
trait C extends B

Subtypes.id[A] == summon[A <:< A]

import Subtypes.⇝

val f: C ⇝ B = summon[C <:< B]
val g: B ⇝ A = summon[B <:< A]
val h: C ⇝ A = summon[C <:< A]

g ∘ f == h
```

## Example: Product category

> *If $C$ and $D$ are categories, one defines the* **product category** $C \times D$.
>
> *An object is a pair $(X, Y)$ where $X \in C$ and $Y \in D$. A morphism is a pair of morphisms*

> $$(f, g) : (X, Y) \to (X', Y'.)$$
>
> *for* $f \in C(X, X')$ *and* $g \in D(Y, Y')$.

Thus for us objects will be types of the form `(A, B)` and arrows tuples of arrows `(C[A1, A2], D[B1, B2])`.

First we use [match types](#) to define two functions that can only be applied to types of tuples, and can extract the first and second arguments (i.e. the first or second type):

```
type Fst[X] = X match {
  case (a, _) => a
}

type Snd[X] = X match {
  case (_, b) => b
}
```

Morphisms are tuples of morphisms:

```
type ×[⇝[_, _], -->[_, _]] =
  [A, B] =>> (Fst[A] ⇝ Fst[B], Snd[A] --> Snd[B])
```

The operator `×` constructs the product category:

```
// C × D
def [
    C[_, _], D[_, _]
  ](C: Category[C]) × (D: Category[D]): Category[C × D] =
  import C.{∘ => *}
  import D.{∘ => +}

  new Category[C × D] {

    def id[A]: A ⇝ A =
      (C.id[Fst[A]], D.id[Snd[A]])

    def [A, B, C] (g: B ⇝ C) ∘ (f: A ⇝ B): A ⇝ C =
      (g._1 * f._1, g._2 + f._2)
  }
```

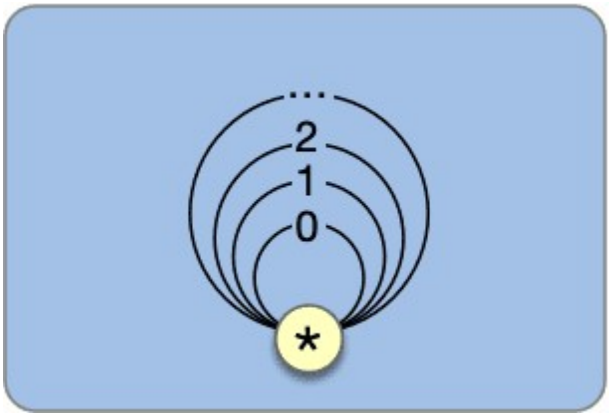
Example:

```
val Scala2 = Scala × Scala

val (f, g) = Scala2.id[(Int, Char)]
// f == identity[Int]
// g == identity[Char]
```

## 1.2 Other encodings

### Example: Monoids as a categories

As an example of a category where the arrows are *not* functions, consider a category with only one (arbitrary) element `*` and whose morphisms are `Ints`.




The previous `Category` definition won't work here because the `Hom` function accepts arguments of *any* type.

One way to restrict the arguments is to add a common upper bound everywhere:

```
trait CategorySub[U, Hom[_ <: U, _ <: U]]
  type ~ = Hom
  def id[A <: U]: A ~ A
  def [A <: U, B <: U, C <: U] (g: B ~ C) ∘ (f: A ~ B): A ~ C

// Category laws modified accordingly
```

Now we can choose an arbitrary value (say `'*'`) and use its singleton type as the upper bound `U`:

```
// The singleton type of the character '*'.
// Any other singleton type will work
// since our goal is to use a type
// with only one inhabitant.
type * = '*'

// this says that the arrows of this category are Ints
type Hom[_ <: *, _ <: *] = Int

object IntCategory extends CategorySub[*, Hom]
  def id[A <: *] = 0
  def [A <: *, B <: *, C <: *] (g: Int) ∘ (f: Int) =
    g + f
```

now we can use the category composition operator `∘` on `Ints`:

```
import IntCategory.∘
```

```
assert(1 ∘ 2 == 3)
```

---

The above definition can be generalized to any `cats.Monoid` instance:

```
// create a new category instance given a cats.Monoid instance
def fromMonoid[M](M: cats.Monoid[M]) =
  type Hom[_ <: *, _ <: *] = M

  new CategorySub[*, Hom] {
    def id[A <: *] = M.empty
    def [A <: *, B <: *, C <: *] (g: M) ∘ (f: M) =
      M.combine(g, f)
  }

// usage:
import cats.implicits._

type H[_ <: *, _ <: *] = String

given StringMonoidCat: CategorySub[*, H] =
  fromMonoid(cats.Monoid[String])

assert( StringMonoidCat.id == "" )
assert( "a" ∘ "b" == "ab" )
```

This construction justifies the following:

> *A single monoid $M$ can be viewed as a category with one formal object $*$.*
>
> *The morphisms $x : * \to *$ are the elements of $M$, composed by the multiplication $xy$ of the monoid, with identity $id(*) = 1$, the unit of the monoid.*
>
> *Grandis, pp 17.*

## Example: Category of Groups

Some of the classic examples of categories are those where the objects are sets with some algebraic structure and the arrows are homomorphisms preserving the algebraic structure.

As an example let's define **Grp**, the category of Groups (i.e. `cats.Group`)

For this we'll have to add a type class constraint `P[_]` to our types:

```
trait CategoryTC[P[_], Hom[_, _]]
  type ⇝ = Hom

  def id[A: P]: A ⇝ A
```

```
  def [A: P, B: P, C: P] (g: B ⇝ C) ∘ (f: A ⇝ B): A ⇝ C
```

In order to represent group homomorphisms we'll have to cheat: a group homomorphism will be just a regular function between groups.

Ideally we would have at least a smart constructor that verifies the given function satisfies the homomorphism property; alas this would be totally impractical, so we'll leave it to the user to verify this either in a test suite or just manually.

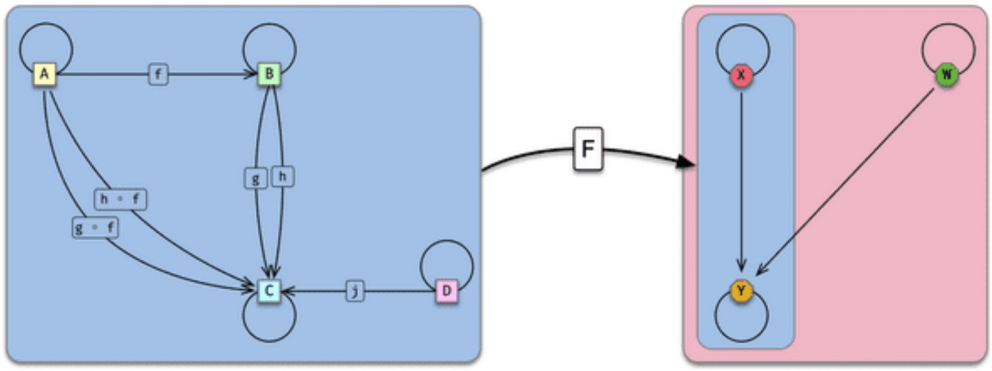
```
import cats.Group
import cats.implicits._
import scala.language.implicitConversions

case class GroupHom[A: Group, B: Group](f: A => B) extends (A => B)
  def apply(a: A) = f(a)

object GroupHomLaws
  def multiplication[A: Group, B: Group](
    f: GroupHom[A, B], x: A, y: A
  ) =
    f(x |+| y) <-> f(x) |+| f(y)
```

Now we can create **Grp**:

```
given Grp: CategoryTC[Group, GroupHom]

  def id[A: Group] = GroupHom(identity)

  def [A: Group, B: Group, C: Group] (g: B ⇝ C) ∘ (f: A ⇝ B): A ⇝ (
    GroupHom(g compose f)
```

# 2. Functors

Functors are graph mappings that preserve arrow composition.




> *If $C$, $D$ are categories, then a (covariant) functor $F$ from $C$ to $D$ is a tuple $(F_0, map)$ where*
>
> 1. $F_0 : Ob(C) \to Ob(D)$

> *(instead of $F_0(X)$ it is common to just write $F_X$).*
>
> 2. *for every pair of objects $X, X'$ in $C$, a function*
>
> $$map_{X,X'} : C(X, X') \to D(F_X, F_{X'})$$
>
> 3. *$F$ preserves composition*
>
> $$F(gf) = F(g).F(f)$$
>
> 4. *$F$ preserves identity morphisms*
>
> $$F(id_X) = id_{F_X}$$

A functor is a data structure that contains a type function `F[_]` and a regular function `map`.

Since now there are two categories involved we're going to use `C[_,_]` for the types of arrows.

```
trait Functor[F[_], C[_, _], D[_, _]](
  given
    Category[C],
    Category[D]
  )
  type ⇝  = C
  type ⟿ = D

  def map  [A, B](f: A ⇝ B): F[A] ⟿ F[B]
  // helper method
  def apply[A, B](f: A ⇝ B) = map(f)


class FunctorLaws[F[_], C[_, _], D[_, _]](F: Functor[F, C, D])(
    given
      C: Category[C],
      D: Category[D],
    )
  type ⇝ = C

  def composition[X, Y, Z](f: X ⇝ Y, g: Y ⇝ Z) =
    F(g ∘ f) <-> F(g) ∘ F(f)

  def identities[X] =
    F(C.id[X]) <-> D.id[F[X]]
```

## 2.1 The identity functor on a category C

```
type Id[A] = A

object Functor
  def identity[C[_, _]: Category] =
```

```
    new Functor[Id, C, C] {
      def map[A, B](f: C[A, B]) = f
    }
```

## 2.2 Endofunctors

A Functor from a category $C$ to itself.

```
type Endofunctor[F[_], C[_, _]] =
  Functor[F, C, C]
```

Later we'll use the following *given* to create `Scala` endofunctors from `cats` instances.

```
import scala.language.implicitConversions

given [F[_]](given F: cats.Functor[F]): Endofunctor[F, Scala]
  def map[A, B](f: A => B) = F.lift(f)

import cats.implicits._
// uses cats.Functor[List] to create our Endofunctor instance
summon[Endofunctor[List, Scala]]
```

## 2.3 Bifunctors

A bifunctor is a functor whose domain is the product category

```
type Bifunctor[F[_, _], Prod[_, _], D[_, _]] =
  Functor[[A] =>> F[Fst[A],  Snd[A]], Prod, D]
```

alternatively:

```
trait Bifunctor2[F[_, _], C1[_, _], C2[_, _], D[_, _]](
    given Category[C1 × C2], Category[D]
  ) extends
    Functor[[A] =>> F[Fst[A],  Snd[A]], C1 × C2, D]
```

## 2.4 The Hom functor

Given a fixed object $X_0$ in a category $C$, the $\mathrm{Hom}$ functor $C \to \mathbb{T}$ sends $A \mapsto \mathrm{Hom}(X_0, A)$.

```
type Hom[C[_, _], X0] = [A] =>> C[X0, A]

def homFunctor[C[_, _]: Category, X0] =
  new Functor[Hom[C, X0], C, Scala] {
    def map[A, B](f: C[A, B]): C[X0, A] => C[X0, B] =
```

```
      f ∘ _
  }
```

## 2.5 Functor composition

```
type *[G[_], F[_]] = [A] =>> G[F[A]]

// G * F
def [G[_], F[_], C[_, _], D[_, _], E[_, _]] (
  G: Functor[G, D, E]) * (
  F: Functor[F, C, D]
  )(given
      Category[C],
      Category[D],
      Category[E]) =
    type ⇝ = C
    type ⟿ = E

    new Functor[G * F, C, E] {
      def map[A, B](f: A ⇝ B): G[F[A]] ⟿ G[F[B]] =
        G(F(f))
    }
```

## 2.6 On notation

Since dotty supports curried type functions an alternative alias for functors could be

```
type ==>[C[_, _], D[_, _]] =
  [F[_]] =>> Functor[F, C, D]
```

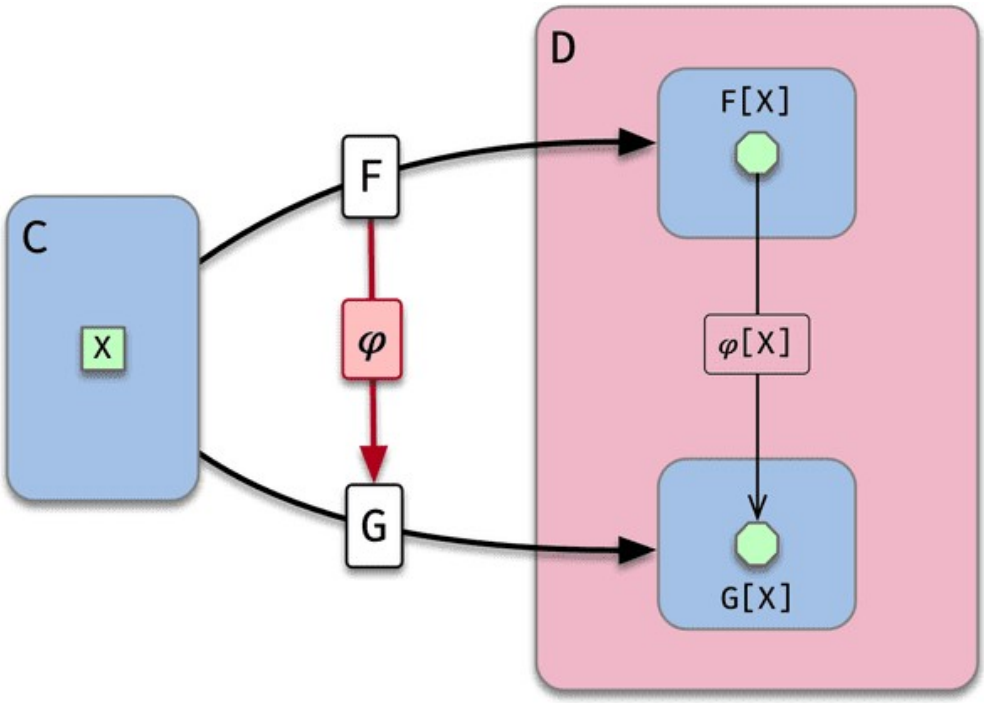
To be used like so:

```
// endofunctors
(C ==> C)[F]

// Bifunctor
(C1 × C2 ==> D)[F]

// etc
```

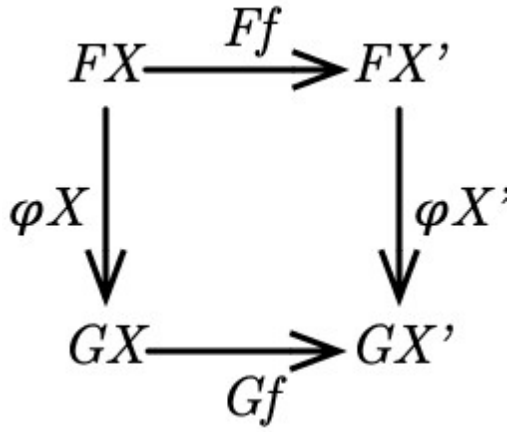
The downside is that it could be hard to keep track of all different kind of arrows.

# 3. Natural Transformations

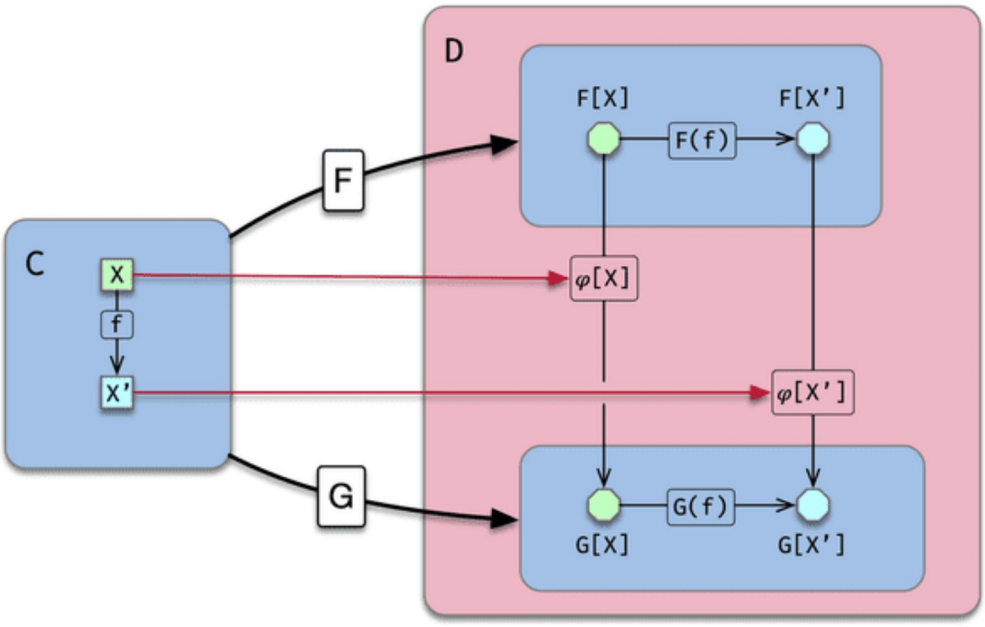
A natural transformation is a family of morphisms between the images of two functors:




> *Given two functors $F, G : C \to D$ between the same categories, a natural transformation $\varphi : F \to G$ consists of the following data:*
>
> *For each object $X$ of $C$ a morphism $\varphi X : FX \to GX$ in $D$ so that, for every arrow $f : X \to X'$ in $C$, we have a commutative square in $D$ (naturality condition of $\varphi$ on $f$)*




> *i.e.*
>
> $$\varphi X'.F(f) = G(f).\varphi X$$




```
trait Nat[F[_], G[_], C[_, _], D[_, _]](
  val source: Functor[F, C, D],
  val target: Functor[G, C, D]
)(given
  Category[C],
  Category[D]
)
  type ⇝ = D
  def apply[X]: F[X] ⇝ G[X]

object NaturalLaws
  def naturality[
    F[_], G[_], C[_, _]: Category, D[_, _]: Category, X, Y
  ](
      F: Functor[F, C, D],
      G: Functor[G, C, D],
      ϕ: Nat[F, G, C, D],
      f: C[X, Y]
  ) =
    ϕ[Y] ∘ F(f) <-> G(f) ∘ ϕ[X]
```

One could also define an alias (double wiggly arrow)

```
type ~~>[F[_], G[_]] =
  [C[_, _], D[_, _]] =>> Nat[F, G, C, D]
```

and use it like so

```
(F ~~> G)[C, D]
```

but same caveats apply regarding “way too many arrows” (TM)

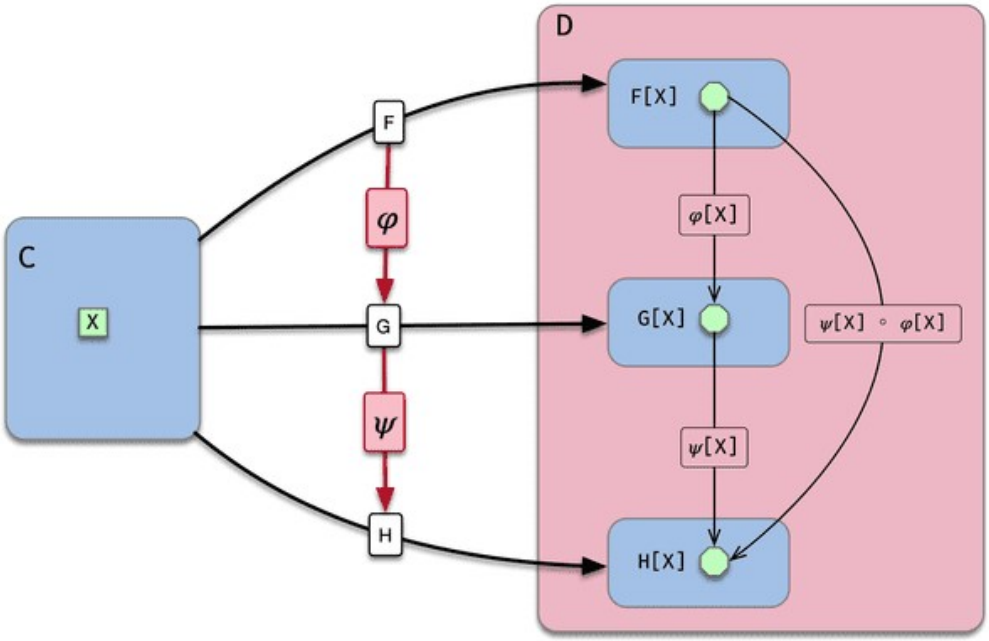
## 3.1 The identity transformation

```
object Nat
```

```
  def identity[F[_], C[_, _]](F: Endofunctor[F, C])(
    given C: Category[C]
  ) =
    new Nat[F, F, C, C](F, F) {
      def apply[X] = C.id[F[X]]
    }
```

## 3.2 Vertical composition

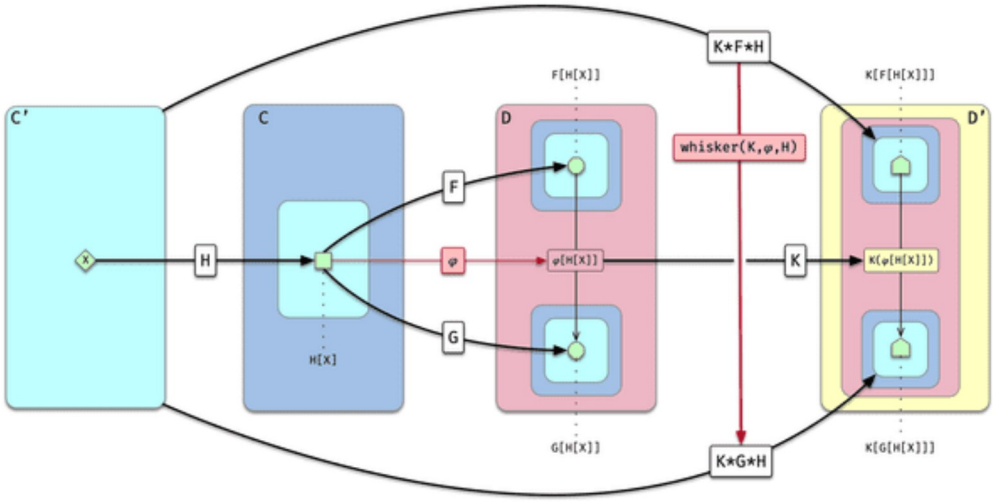
> *Two natural transformations* $\varphi : F \to G$ *and* $\psi : G \to H$ *have a* **vertical composition** $\psi\varphi : F \to H$ *(also written* $\psi.\varphi$*)*
>
> $$(\psi\varphi)(X) = \psi(X).\varphi(X) : FX \to HX$$




```
// vertical composition:
// ψ * ϕ: F ~~> H
def [F[_], G[_], H[_], C[_, _], D[_, _]](
  ψ: Nat[G, H, C, D] ) * (
  ϕ: Nat[F, G, C, D]
)(given
  Category[C],
  Category[D])
 =
  new Nat[F, H, C, D](ϕ.source, ψ.target) {
    def apply[X]: F[X] ⇝ H[X] =
      ψ[X] ∘ ϕ[X]
  }
```

## 3.3 Whisker composition

> *Moreover there is a* **whisker composition** *of natural transformations with functors, or* **reduced horizontal composition**, *written as* $K\varphi H$
>
> $$K\varphi H : KFH \to KGH$$
> $$(K\varphi H)(X) = K(\varphi(HX)$$




```
// whisker(K, ϕ, H): K * F * H ~~> K * G * H
def whisker[
  H[_], F[_], G[_], K[_],
  Cp[_, _], C[_, _], D[_, _], Dp[_, _]
](
  K: Functor[K, D, Dp],
  ϕ: Nat[F, G, C, D],
  H: Functor[H, Cp, C],
)(given
  Category[Cp],
  Category[C],
  Category[D],
  Category[Dp],
): Nat[K * F * H, K * G * H, Cp, Dp] =
  val F = ϕ.source
  val G = ϕ.target
  new Nat[K * F * H, K * G * H, Cp, Dp](
          K * F * H, K * G * H
  ) {
    def apply[X] = K(ϕ[H[X]])
  }
```

> *The binary operations $\varphi H$ and $K\varphi$ are also called whisker compositions (obtained by inserting identity functors in the ternary operation)*

```
// (ϕ *: H): F * H ~~> G * H
def [H[_], F[_], G[_], Cp[_, _], C[_, _], D[_, _]](
  ϕ: Nat[F, G, C, D]) *: (
  H: Functor[H, Cp, C]
)(given
  Category[Cp],
  Category[C],
  Category[D]
): Nat[F * H, G * H, Cp, D] =
  whisker(Functor.identity[D], ϕ, H)

// (K :* ϕ): K * F ~~> K * G
def [F[_], G[_], K[_], C[_, _], D[_, _], Dp[_, _]](
  K: Functor[K, D, Dp]) :* (
  ϕ: Nat[F, G, C, D]
```

```
)(given
  Category[C],
  Category[D],
  Category[Dp]
): Nat[K * F, K * G, C, Dp] =
  whisker(K, ϕ, Functor.identity[C])
```

We've chosen `ϕ *: H` and `K :* ϕ` to have a visual clue of which side is the natural transformation and which side is the functor.

A natural transformation between two functors in **Scala** (`Nat[F, G, Scala, Scala]`) is just a polymorphic function that is capable of transforming one data structure / context into another, without looking at the concrete type argument.

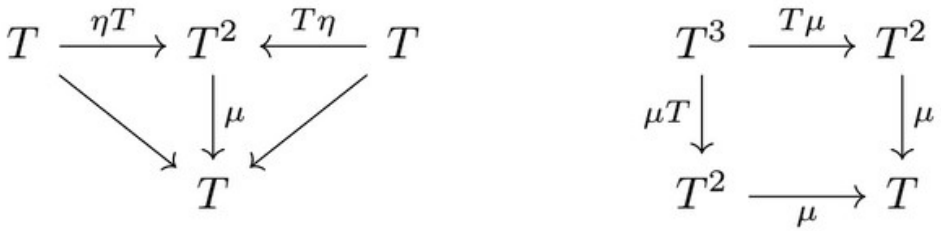
We could say that a functor transforms the *contents* of a data structure from one type into another:

```
F[A] -> F[B]
```

whereas a natural transformation performs a *structural/context* transformation:

```
F[A] -> G[A]
```

# 4. Monads

> *A monad in the category $X$ is a triple $(T, \eta, \mu)$ where $T : X \to X$ is an endofunctor, while $\eta : 1 \to T$ and $\mu : T^2 \to T$ are natural transformations (called the unit and multiplication of the monad) which make the following diagrams commute:*
>
> 
>
> *(Grandis, p133)*

```
trait Monad[T[_], X[_, _]](given X: Category[X])

  type ~~>[F[_], G[_]] = Nat[F, G, X, X]

  def T: Endofunctor[T, X]

  //     η:  1 ~~> T
  def pure: Id ~~> T
```

```
  //     μ    :    T²   ~~> T
  def flatten: (T * T) ~~> T


class MonadLaws[T[_], X[_, _]](m: Monad[T, X])(given X: Category[X])
  import m._

  def unitarity1 =
    (flatten * (pure *: T)) <-> Nat.identity(T)

  def unitarity2 =
    (flatten * (T :* pure)) <-> Nat.identity(T)

  def associativity =
    flatten * (T :* flatten) <-> flatten * (flatten *: T)
```

## 4.1 Monoids in the category of endofunctors

> *In fact, a monad on the category $X$ is an internal monoid in the category $C = \mathrm{End}(X)$ of endofunctors of $X$ and their natural transformations, equipped with the strict (non-symmetric) monoidal structure given by the composition of endofunctors.*

Let's unpack this remark.

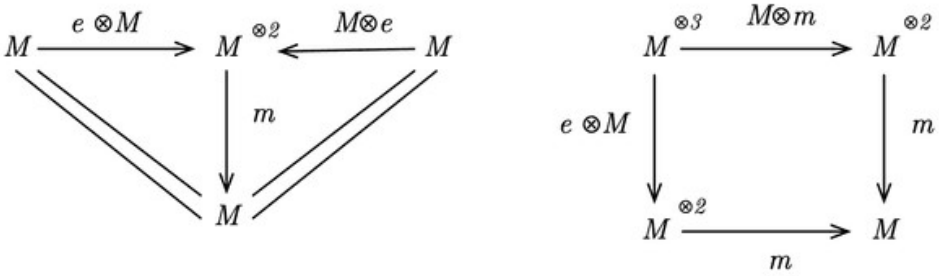
### 4.1.1 Monoidal categories and internal monoids

> A (strict) **monoidal category** $(C, \otimes, E)$ *is a category $C$ equipped with a a bifunctor*
>
> $$\otimes : C \times C \to C$$
>
> *that is associative, and an object $E$ called the unit satisfying*
>
> $$E \otimes A = A$$
> $$A \otimes E = A$$

```
trait Monoidal[C[_, _]] extends Category[C]
  type ⊗[_, _]
  type E

  def tensor: Bifunctor[⊗, C × C, C]

  // tensor gives rise to the canonical function
  // f ⊗ g:
  def [A, B, Ap, Bp](f: A ~> B) ⊗ (g: Ap ~> Bp): (A ⊗ Ap) ~> (B ⊗ Bp
    tensor[(A, Ap), (B, Bp)]((f, g))

  def associativity[A, B, C]:
    (A ⊗ (B ⊗ C)) =:= (A ⊗ B) ⊗ C

  def unitLeft[A]:
    E ⊗ A =:= A
```

```
  def unitRight[A]:
    A ⊗ E =:= A
```

> An **internal monoid** in $C$ is a triple $(M, e, m)$, consisting of an object $M$ and two arrows $e : E \to M, m : M \otimes M \to M$ of $C$, called the unit and multiplication, satisfying:




```
trait InternalMonoid[C[_, _]](given val C: Monoidal[C])
  import C.{⇝, E, ⊗, id}

  type M
  val e: E ⇝ M
  val m: M ⊗ M ⇝ M

  // laws
  def unitarity1 =
    m ∘ (e ⊗ id[M]) <-> id[M]

  def unitarity2 =
    m ∘ (id[M] ⊗ e) <-> id[M]

  def associativity =
    m ∘ (m ⊗ id[M]) <-> m ∘ (id[M] ⊗ m)
```

### 4.1.2 The category of endofunctors and natural transformations

In order to describe this category we'll need a higher order version of `CategoryTC` (that we used to create the category of groups).

```
// A category of type functions with some constraint P
trait CategoryTC1[P[_[_]], Hom[_[_], _[_]]]

  type ⇝ = Hom

  def id[F[_]](given P[F]): F ⇝ F

  def [F[_]: P, G[_]: P, H[_]: P] (m: G ⇝ H) ∘ (n: F ⇝ G): F ⇝ H
```

This means objects will be type functions $\mathbb{T} \to \mathbb{T}$ with some constraint represented by `P[_[_]]` and `Hom` is a function $(\mathbb{T} \to \mathbb{T}, \mathbb{T} \to \mathbb{T}) \to \mathbb{T}$.

```
// Given a Category[X] creates the category whose objects are
// endofunctors of X and whose morphisms are the natural
// transformations between them.

def endo[X[_, _]](given Category[X]) =

  type Hom[H[_], G[_]] = Nat[H, G, X, X]

  type EndoX[F[_]] = Endofunctor[F, X]

  new CategoryTC1[EndoX, Hom] {

    def id[F[_]](given f: EndoX[F]): F ~> F =
        Nat.identity(f)

    def [F[_], G[_], H[_]] (m: G ~> H) ∘ (n: F ~> G): F ~> H =
      m * n
  }
```

For example, this is identity natural transformation for `List` in the category of endofunctors in **Scala**:

```
endo[Scala].id[List]
```

And this is the category of endofunctors for the `Kleisli` category of functions `A => Option[B]`:

```
val optionKleisliEndo = endo[[A, B] =>> Kleisli[Option, A, B]]
```

### 4.1.3 Monoids in the category of endofunctors

> *In fact, a monad on the category $X$ is an internal monoid in the category $C = End(X)$ of endofunctors of $X$ and their natural transformations, equipped with the strict monoidal structure given by the composition of endofunctors.*

I won't rewrite all the definitions needed here in terms of `CategoryTC1` , but instead I'll just ask the reader to compare the structure of `InternalMonoid` vs `Monad`.

| Internal Monoid | Monad |
| --- | --- |
| `trait InternalMonoid[C[_, _]]`<br>`  (given val C: Monoidal[C])`<br>`  import C.{~>, E, ⊗, id}`<br>`  //`<br>`  //`<br>`  type M`<br>`  val e: E ~> M`<br>`  val m: M ⊗ M ~> M` | `trait Monad[T[_], X[_, _]]`<br>`  (given Category[X])`<br>`  type ~~>[F[_], G[_]] =`<br>`  Nat[F, G, X, X]`<br>`  //`<br>`  def T : Endofunctor[T, X]`<br>`  def pure : Id ~~> T`<br>`  def flatten : (T * T) ~~> T` |

| Internal Monoid | Monad |
| --- | --- |
| unitarity1 | |
| `m ∘ (e ⊗ id[M]) <-> id[M]` | `(flatten * (pure *: T)) <-> Nat.identity(T)` |
| unitarity2 | |
| `m ∘ (id[M] ⊗ e) <-> id[M]` | `(flatten * (T :* pure)) <-> Nat.identity(T)` |
| associativity | |
| `m ∘ (m ⊗ id[M]) <->`<br>`m ∘ (id[M] ⊗ m)` | `flatten * (T :* flatten) <->`<br>`flatten * (flatten *: T)` |

## 4.2 Monads in Scala

If we set the category to be `Scala` (of types and functions) several things happen:

- `Endofunctor` becomes the usual `cats.Functor`
- Natural transformations become:

```
trait Nat[F[_]: cats.Functor, G[_]: cats.Functor]
  def apply[A]: F[A] => G[A]
```

- Our specialized monad becomes:

```
trait Monad[T[_]: cats.Functor]
  def pure   : Nat[Id,  T]
  def flatten: Nat[T∘T, T]
```

- Simplifying even more by inlining the `apply` method of both transformations:

```
trait Monad[T[_]: cats.Functor]
  def pure   [A]:     A   => T[A]
  def flatten[A]: T[T[A]] => T[A]

  def flatMap[A, B](a: T[A])(f: A => T[B]): T[B] =
    flatten(cats.Functor[T].map(a)(f))
```

*voila*!

# 5. The Yoneda lemma

> *Let* $F, G : C \to Set$ *be two functors, with* $F = C(X_0, \_)$. *The canonical mapping*

$$\begin{array}{rccc} y: & Nat(F,G) & \to & G_{X_0} \\ & \varphi & \mapsto & \varphi_{X_0}(id_{X_0}) \end{array}$$

*from the set of natural transformations $\varphi : F \to G$ to the set $G(X_0)$ is a bijection.*

*The inverse mapping is given by*

$$\begin{array}{rccc} y': & G_{X_0} & \to & Nat(Hom(X_0,\_),G) \\ & z & \mapsto & y'(z)_X f := G_f(z) \;\; \forall X \\ & & & \qquad\qquad f{:}C(X_0,X) \end{array}$$

*(pp 44)*

A bijection between two types `A` and `B` can be naturally expressed as a instance of the following type

```
case class Biyection[A, B](
  from: A => B,
  to  : B => A
)
  // satisfying laws
  def IdB = (from ∘ to) <-> identity[B]
  def IdA = (to ∘ from) <-> identity[A]

type <=>[A, B] = Biyection[A, B]
```

Since our base category is **Scala** (instead of **Set**) then both $F, G$ will be functors from $C$ to **Scala**.

```
def yonedaLemma[C[_, _], G[_], X0](G: Functor[G, C, Scala])
  (given C: Category[C] ) =
  import C.id

  // The hom-functor at X0 in the category C
  type CX0[A] = C[X0, A]
  val F: Functor[CX0, C, Scala] = homFunctor[C, X0]

  // at this point we know that F, G are functors
  // and C is a category, so we can simplify a bit
  // by working with the natural transformation as a
  // polymorhic function instead of the wrapper type Nat:
  type ~~>[F[_], G[_]] =
    [X] => F[X] => G[X]

  // ----------------
  // The Yoneda lemma
  // ----------------

  // the canonical mapping:
  val y : (CX0 ~~> G) => G[X0] =
              ϕ      => ϕ[X0]( id[X0] )

  // is a bijection with inverse:
  val yp: G[X0] => (CX0 ~~> G) =
```

```
            z   => ([X] => (f: C[X0, X]) => G(f)(z))

// i.e.
// y  ∘ yp <-> identity[G[X0]]
// yp ∘ y  <-> identity[CX0 ~~> G]

// Or more concisely:
val yoneda: (CX0 ~~> G) <=> G[X0] =
    Biyection(y, yp)
```

Proof:

```
// a) yp(y(φ)) == φ
def proofPart1(ϕ: CX0 ~~> G) =

  // by definition of y:
  yp(y(ϕ)) == yp( ϕ[X0] { id[X0] } )

  // by definition of yp:
  yp(y(ϕ)) == ( [X] => (f: C[X0, X]) => G(f) { ϕ[X0] { id[X0] } }

  // which in Scala is the same as:
  yp(y(ϕ)) == ( [X] => (f: C[X0, X]) => (G(f) ∘ ϕ[X0]) { id[X0] }

  // by the naturality condition: G(f) ∘ ϕ[X0] === ϕ[X] ∘ F(f)
  yp(y(ϕ)) == ( [X] => (f: C[X0, X]) => (ϕ[X] ∘ F(f)) { id[X0] } )

  yp(y(ϕ)) == ( [X] => (f: C[X0, X]) => ϕ[X] { F(f) { id[X0] } } )

  // by definition of homFunctor: F(f)( id[X0] ) == f ∘ id[X0]
  // and by definition of composition: f ∘ id[X0] == f
  yp(y(ϕ)) == ( [X] => (f: C[X0, X]) => ϕ[X] { f } )

  // applying eta reduction two times:
  // yp(y(ϕ)) == ( [X] => ϕ[X]  ) == ϕ

  yp(y(ϕ)) == ϕ


// b) y(yp(z)) == z
def proofPart2(z: G[X0]) =

  // by definition of yp:
  y(yp(z)) == y( [X] => (f: C[X0, X]) => G(f)(z) )

  // by definition of y:
  y(yp(z)) == ( [X] => (f: C[X0, X]) => G(f)(z) ) { id[X0] }

  // evaluating the right hand side
  y(yp(z)) == G(id[X0])(z)

  // Since G maps identities to identities:
  // G(id[X0]) == Scala.id[G[X0]] == identity[G[X0]] = x => x
  y(yp(z)) == z
```

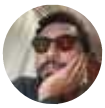
```
// Q.E.D
```

This is pretty much what one would write on paper, except that it is more verbose. The other difference is that the compiler will help us ensure that expressions at least typecheck.

## 6. Bibliography


- M. Grandis, Category Theory And Applications: A Textbook For Beginners, World Scientific, 2018.
- W. Lawvere, Conceptual Mathematics: A First Introduction to Categories, Cambridge University Press; 2 edition, 2009.
- Dotty 0.20.0-RC1 documentation: https://dotty.epfl.ch/docs.


---



Personal blog of **Juan Pablo Romero Méndez**.