A pointfree Yoneda lemma for endofunctors of functional categories

Luc Duponcheel,
mathematician and programmer, cyclist and gardener.

February 24, 2023

Abstract

Category theory [2] is used in almost all areas of mathematics. In mathematics, the Yoneda lemma is a fundamental category theory result that suggests that, instead of studying a category C, one should study the category of all functors from C to $\boldsymbol{Fun}$ (a.k.a. $\boldsymbol{Set}$), the category whose objects are sets, and whose morphisms are functions. This suggestion is now agreed upon to be an important step forward to a better understanding of various categories. Surprisingly, the lemma uses only three concepts: category, functor and natural transformation. On the other hand, kind of inevitably, the lemma is, pointful: it involves elements of sets.

Category theory is also used in other disciplines, ranging from physics to philosophy. This paper uses category in computing science [1] where, typically, objects are called nodes and morphisms are called arrows. Consider a programming language `L`, with a type system that supports higher-kinded type classes [3]. Using `L` one can write a domain specific language library that models programming. It specifies categories `FC`, referred to as functional categories, whose nodes, invariably, are the types of `L` and whose arrows are called programs. Let `Fun` (a.k.a. `Type`) be the category whose arrows are the effectfree functions of `L`. The most important property of a functional category `FC` is the existence a functor from `Fun` to `FC` that is the identity on types. Using that functor the effectfree functions of `L` can be used as programs. Of course, functional categories may also have effectful programs, most notably, programs that perform IO. Functors from `FC` to `Fun` can be composed with the functor from `Fun` to `FC` above to obtain an endofunctor of `FC`.

When defining functional categories FC in mathematics, roughly speaking, `Fun` is replaced by $\boldsymbol{Fun}$. It is natural to try to formulate and prove a Yoneda lemma for endofunctors of functional categories. It is challenging to minimize the amount of concepts involved and to make both the formulation and the proof of the lemma pointfree, only dealing with morphisms of FC.

This paper is the result of advancing insights, obained, most notably, by writing code in `Scala` [5]. The mathematical notation of the paper is almost identical to the programmatic notation of the code. The fact that the `Scala` type system accepts the code implies that propositions and their proofs are syntactically correct, which, in its turn, especially because they are generic, provides some confidence that they are also semantically correct.

1 Fonts

When introducing concepts that are part of the index we use *emphasized* font.

- For *declarations* we use *math* font.
- For *definitions* we use ***boldface math*** font.
- For *code* we use `typewriter` font.

2 Preliminaries

2.1 Definition

The *graph specification*, G, declares

$G_{dec}-0$ G_0, *a class of nodes*, Z, Y $\ldots$,

$G_{dec}-1$ G_1, *sets of arrows*, $Arr_G(Z, Y)$, containing arrows $z \rightarrow y$ with *source* node Z and *target* node Y.

Note that we distinguish classes from sets.

Nodes form a *class*, not necessarily a *set*, cfr. Russell-Zermelo's paradox: sets do not form a set.

2.2 Definition

G_2, *composable arrows* of a graph G are arrows $z \rightarrow y$ and $y \rightarrow x$.

2.2.1 Notation

$Arr_G(Z, Y)$ is also simply denoted $Arr(Z, Y)$ when only one graph is involved.

2.2.2 `Scala` code

```
package plp.specification

trait Graph[Arr[-_, +_]]
```

Specifications are encoded as `trait`'s.

`Graph` does not declare any members.

Programmatically, the class of nodes is, implicitly and invariably, defined as *the class of types*. Only arrows matter.

Of course, nodes being types is a limitation when encoding mathematical concepts *in general*, but, fortunately, for the *specific* purpose of this paper, it will turn out not to be an issue at all.

The *class* of types is a *set* of types, in fact it is a *constructive* set of types.

The sets of arrows are declared as a *binary type constructor parameter*, `Arr[-_, +_]`, constructing *arrow types* `Arr[Z, Y]` ... , containing *arrow values* `` `z-->y` `` ... .

The underscores _ are placeholders for *type parameters*.

`-` and `+` declare *variance*. Just like functions, arrows are allowed to require less and provide more.

2.3 Definition

The *function graph*, ***Fun***, defines

$\boldsymbol{Fun}_{def}-0$ $\boldsymbol{Fun}_0$, the class of nodes, as *the class of sets*, $\boldsymbol{Z}$, $\boldsymbol{Y}$ ... ,

$\boldsymbol{Fun}_{def}-1$ $\boldsymbol{Fun}_1$, the sets of arrows, as *the sets of functions* $\boldsymbol{Fun}(\boldsymbol{Z}, \boldsymbol{Y})$ ... , containing functions $\boldsymbol{z}\Rightarrow\boldsymbol{y}$ with source set $\boldsymbol{Z}$ and target set $\boldsymbol{Y}$.

2.3.1 `Scala` code

```
package plp.implementation.specific

import plp.specification.{Graph}
```

```
type Fun[-Z, +Y] = Z => Y

given functionGraph: Graph[Fun] with {}
```

Implementations are encoded as `given`'s.

`functionGraph` does not define any members.

The sets of functions are defined as a *binary type constructor*, `Fun`, constructing *function types* `Fun[Z, Y]` ... , containing *function values* `` `z=>y` `` ... .

2.4 Definition

The *graph morphism specification*, $M : G \to H$, declares

$M_{dec}-0$ $M_0 : G_0 \to H_0$, a *node part*,

$M_{dec}-1$ $M_1 : G_1 \to H_1$ an *arrow part*, consisting of functions $Arr_G(Z, Y) \to Arr_H(M_0(Z), M_0(Y))$.

2.4.1 Notation

Nodes $M_0(Z)$ are also denoted $M(Z)$.

Arrows $M_1(z{\to}y)$ are also denoted $m(z{\to}y)$ and $m(z){\to}m(y)$.

2.4.2 `Scala` code

```
package plp.specification

trait GraphMorphism[Arr_G[-_, +_]: Graph, Arr_H[-_, +_]: Graph, M[+_]]:

  def lift[Z, Y]: Arr_G[Z, Y] => Arr_H[M[Z], M[Y]]
```

The node part is declared as a *unary type constructor parameter*, `M[+_]`, constructing *lifted types* `M[Z]` ... .

The arrow part is declared as a *function member* `lift`, yielding *lifted arrows*.

The name `lift` can, by need, be made available as `m`.

2.5 Definition

The *category specification*, C, extends the graph specification.

It declares

$C_{dec}{-}0$ $\;composition : C_2 \to C_1$,

$C_{dec}{-}1$ $\;unit : C_0 \to C_1$.

2.5.1 Notation

If $z{\to}y$ and $y{\to}x$ are composable arrows, then the arrow $composition(z{\to}y, y{\to}x)$ is denoted $y{\to}x \circ_c z{\to}y$ and is called the *arrow composition* of $z{\to}y$ and $y{\to}x$, or, simply, the *composition* of $z{\to}y$ and $y{\to}x$.

If Z is a node, then the arrow $unit(Z)$ is denoted $z \to_c z$ and is called the *identity arrow* of Z, or, simply, the *identity* of Z.

2.5.2 Notation

$y{\to}x \circ_c z{\to}y$ is also simply denoted $y{\to}x \circ z{\to}y$ when only one category is involved.

$z \to_c z$ is also simply denoted $z{\to}z$ when only one category is involved.

2.5.3 `Scala` code

```
package plp.specification

trait Category[Arr[-_, +_]] extends Graph[Arr]:
```

```
  def composition[Z, Y, X]: (Arr[Y, X], Arr[Z, Y]) => Arr[Z, X]

  def unit[__]: Arr[__, __]
```

composition is declared as a *function member* `composition`.

unit is declared as an *arrow member* `unit`.

The double underscores, __ , are *unnamed type parameters* that are used if type parameter names do not matter. If they do matter, then it is always possible to somehow introduce named ones.

```
  extension [Z, Y, X](`y-->x`: Arr[Y, X])
    def o(`z-->y`: Arr[Z, Y]): Arr[Z, X] = composition(`y-->x`, `z-->y`)

  def `__-->__`[__]: Arr[__, __] = unit
```

`Category` also defines syntax, `o` resp. `` `__-->__` `` for `composition` resp. `unit`.

2.5.4 Laws

$C_{law}-0$ $(x{\rightarrow}w \circ y{\rightarrow}x) \circ z{\rightarrow}y = x{\rightarrow}w \circ (y{\rightarrow}x \circ z{\rightarrow}y)$ (*associativity law*),

$C_{law}-1$ $y{\rightarrow}y \circ z{\rightarrow}y = z{\rightarrow}y$ (*left identity law*),

$C_{law}-2$ $z{\rightarrow}y \circ z{\rightarrow}z = z{\rightarrow}y$ (*right identity law*).

2.5.5 `Scala` code

Let

```
package plp.notation

case class Law[__](equation: (__, __))

extension [__](lhs: __) def =:(rhs: __): Law[__] = Law(lhs, rhs)
```

in

```
package plp.specification

import plp.notation.{Law, =:}
```

```
class CategoryLaws[Arr[-_, +_]: Category]:
```

```
  def categoryCompositionAssociativityLaw[Z, Y, X, W]
      : (Arr[Z, Y], Arr[Y, X], Arr[X, W]) => Law[Arr[Z, W]] =
    (`z-->y`, `y-->x`, `x-->w`) =>
```

```
      ((`x-->w` o `y-->x`) o `z-->y`) =:
        (`x-->w` o (`y-->x` o `z-->y`))
```

```
  def categoryLeftIdentityLaw[Z, Y]: Arr[Z, Y] => Law[Arr[Z, Y]] =
    `z-->y` =>
```

```
      val `y-->y` = summon[Category[Arr]].`__-->__`[Y]
```

```
      (`y-->y` o `z-->y`) =:
        (`z-->y`)
```

```
  def categoryRightIdentityLaw[Z, Y]: Arr[Z, Y] => Law[Arr[Z, Y]] =
    `z-->y` =>
```

```
      val `z-->z` = summon[Category[Arr]].`__-->__`[Z]
```

```
      (`z-->y` o `z-->z`) =:
        (`z-->y`)
```

2.6 Definition

The *function category* category, $\boldsymbol{Fun}$, defines

$\boldsymbol{Fun}_{def}-0$ *composition* : $\boldsymbol{Fun}_2 \rightarrow \boldsymbol{Fun}_1$ as *function composition*,

$\boldsymbol{Fun}_{def}-1$ *unit* : $\boldsymbol{Fun}_0 \rightarrow \boldsymbol{Fun}_1$ as the *identity function*.

2.6.1 `Scala` code

```
package plp.implementation.specific

import plp.specification.{Category}
```

```
given functionCategory: Category[Fun] with
```

```
  def composition[Z, Y, X]: (Fun[Y, X], Fun[Z, Y]) => Fun[Z, X] =
    (`y=>x`, `z=>y`) => z => `y=>x`(`z=>y`(z))

  def unit[__]: Fun[__, __] = __ => __
```

The double underscores, `__` , are *unnamed function parameters* that are used if function parameter names do not matter. If they do matter, then it is always possible to somehow introduce named ones.

2.6.2 `Scala` code

Below is a proof.

Let

```
package plp.notation

import scala.collection.immutable.Seq
```

```
case class Proof[__](steps: Seq[__])

extension [__](step: __)
  def ==:(proof: Proof[__]): Proof[__] = Proof(step +: proof.steps)

def qed[__]: Proof[__] = Proof(Seq())
```

in

```
import plp.notation.{Proof, ==:, qed}
```

```
class FunctionCategoryProofs:
```

```
def functionCompositionAssociativityProof[Z, Y, X, W]
    : ((Fun[Z, Y], Fun[Y, X], Fun[X, W]) => Z => Proof[W]) =
  (`z=>y`, `y=>x`, `x=>w`) =>
    z =>
```

```
      ((`x=>w` o `y=>x`) o `z=>y`)(z) ==:
        // definition o for Fun
        (`x=>w` o `y=>x`)(`z=>y`(z)) ==:
        // definition o for Fun
        `x=>w`(`y=>x`(`z=>y`(z))) ==:
        // definition o for Fun
        `x=>w`((`y=>x` o `z=>y`)(z)) ==:
        // definition o for Fun
        (`x=>w` o (`y=>x` o `z=>y`))(z) ==:
        // done
        qed
```

```
def functionLeftIdentityProof[Z, Y]: Fun[Z, Y] => Z => Proof[Y] =
  `z=>y` =>
    z =>
```

```
      val `y-->y` = functionCategory.`__-->__`[Y]
```

```
      (`y-->y` o `z=>y`)(z) ==:
        // definition o for Fun
        `y-->y`(`z=>y`(z)) ==:
        // definition `y-->y` for Fun
        (`z=>y`) (z) ==:
        // done
        qed
```

```
def functionRightIdentityProof[Z, Y]: Fun[Z, Y] => Z => Proof[Y] =
  `z=>y` =>
    z =>
```

```
      val `z-->z` = functionCategory.`__-->__`[Z]
```

```
      (`z=>y` o `z-->z`)(z) ==:
        // definition o for Fun
        `z=>y`(`z-->z`(z)) ==:
        // definition `z-->z` for Fun
        (`z=>y`) (z) ==:
        // done
        qed
```

2.7 Definition

The *functor specification* $F : C \to D$, where C and D are categories, extends the graph morphism specification.

2.7.1 Scala code

```
package plp.specification

trait Functor[Arr_C[-_, +_]: Category, Arr_D[-_, +_]: Category, F[+_]]
    extends GraphMorphism[Arr_C, Arr_D, F]
```

2.7.2 Laws

$F_{law}-0$ $f(y{\rightarrow}x \circ_c z{\rightarrow}y) = f(y{\rightarrow}x) \circ_d f(z{\rightarrow}y)$ (*composition law*),

$F_{law}-1$ $f(z \rightarrow_c z) = f(z) \rightarrow_d f(z)$ (*identity law*).

2.7.3 Scala code

```
package plp.specification

import plp.notation.{Law, =:}
```

```
class FunctorLaws[
    Arr_C[-_, +_]: Category,
    Arr_D[-_, +_]: Category,
    F[+_]: [_[+_]] =>> Functor[Arr_C, Arr_D, F]
]:
```

```
  def f[Z, Y]: Arr_C[Z, Y] => Arr_D[F[Z], F[Y]] =
    summon[Functor[Arr_C, Arr_D, F]].lift
```

```
  def functorCompositionLaw[Z, Y, X]
      : (Arr_C[Z, Y], Arr_C[Y, X]) => Law[Arr_D[F[Z], F[X]]] =
    (`z-->y`, `y-->x`) =>
```

```
      extension [Z, Y, X](`y-->x`: Arr_C[Y, X])
        def o_c(`z-->y`: Arr_C[Z, Y]): Arr_C[Z, X] =
          summon[Category[Arr_C]].o[Z, Y, X](`y-->x`)(`z-->y`)

      extension [Z, Y, X](`y-->x`: Arr_D[Y, X])
        def o_d(`z-->y`: Arr_D[Z, Y]): Arr_D[Z, X] =
          summon[Category[Arr_D]].o[Z, Y, X](`y-->x`)(`z-->y`)
```

```
      (f(`y-->x` o_c `z-->y`)) =:
        (f(`y-->x`) o_d f(`z-->y`))
```

```
  def functorIdentityLaw[Z]: Law[Arr_D[F[Z], F[Z]]] =
```

```
    val `z-c->z` = summon[Category[Arr_C]].`__-->__`[Z]

    val `f[z]-d->f[z]` = summon[Category[Arr_D]].`__-->__`[F[Z]]
```

```
    (f(`z-c->z`)) =:
      (`f[z]-d->f[z]`)
```

2.8 Definition

An *endofunctor* is a functor $F : C \to C$.

2.8.1 `Scala` code

```
type EndoFunctor[Arr[-_, +_], F[+_]] = Functor[Arr, Arr, F]
```

2.9 Definition

Given

- categories C, D and E,
- functors $F : C \to D$ and $G : D \to E$,

the *composed functor*, $G \circ F : C \to E$, defines

$(G \circ F)_{def}{-}0 \;\; (G \circ F)(Z) = G(F(Z))$,

$(G \circ F)_{def}{-}1 \;\; (g \circ f)(z{\to}y) = g(f(z{\to}y))$.

2.9.1 Scala code

Let

```
package plp.notation

type O = [G[+_], F[+_]] =>> [__] =>> G[F[__]]
```

in

```
package plp.implementation.generic

import plp.notation.{O}

import plp.specification.{Category, Functor}

import plp.implementation.specific.{functionCategory}
```

```
given composedFunctor[
    Arr_C[-_, +_]: Category,
    Arr_D[-_, +_]: Category,
    Arr_E[-_, +_]: Category,
    F[+_]: [_[+_]] =>> Functor[Arr_C, Arr_D, F],
    G[+_]: [_[+_]] =>> Functor[Arr_D, Arr_E, G]
]: Functor[Arr_C, Arr_E, G O F] with
```

```
  def lift[Z, Y]: Arr_C[Z, Y] => Arr_E[(G O F)[Z], (G O F)[Y]] =
```

```
    val f: Arr_C[Z, Y] => Arr_D[F[Z], F[Y]] =
      summon[Functor[Arr_C, Arr_D, F]].lift

    val g: Arr_D[F[Z], F[Y]] => Arr_E[G[F[Z]], G[F[Y]]] =
      summon[Functor[Arr_D, Arr_E, G]].lift
```

```
    `z-->y` => g(f(`z-->y`))
```

2.9.2 Scala code

Below is a proof.

```
import plp.notation.{Proof, ==:, qed}
```

```
class ComposedFunctorProofs[
    Arr_C[-_, +_]: Category,
    Arr_D[-_, +_]: Category,
    Arr_E[-_, +_]: Category,
    F[+_]: [_[+_]] =>> Functor[Arr_C, Arr_D, F],
    G[+_]: [_[+_]] =>> Functor[Arr_D, Arr_E, G]
]:
```

```
  def f[Z, Y]: Arr_C[Z, Y] => Arr_D[F[Z], F[Y]] =
    summon[Functor[Arr_C, Arr_D, F]].lift

  def g[Z, Y]: Arr_D[F[Z], F[Y]] => Arr_E[G[F[Z]], G[F[Y]]] =
    summon[Functor[Arr_D, Arr_E, G]].lift

  def `gof`[Z, Y]: Arr_C[Z, Y] => Arr_E[G[F[Z]], G[F[Y]]] =
    composedFunctor[Arr_C, Arr_D, Arr_E, F, G].lift
```

```
  def functorCompositionProof[Z, Y, X]
      : Arr_C[Z, Y] => (Arr_C[Y, X] => Proof[Arr_E[(G O F)[Z], (G O F)[X]]]) =
    `z-->y` =>
      `y-->x` =>
```

```
        extension [Z, Y, X](`y-->x`: Arr_C[Y, X])
          def o_c(`z-->y`: Arr_C[Z, Y]): Arr_C[Z, X] =
            summon[Category[Arr_C]].o[Z, Y, X](`y-->x`)(`z-->y`)

        extension [Z, Y, X](`y-->x`: Arr_D[Y, X])
          def o_d(`z-->y`: Arr_D[Z, Y]): Arr_D[Z, X] =
            summon[Category[Arr_D]].o[Z, Y, X](`y-->x`)(`z-->y`)

        extension [Z, Y, X](`y-->x`: Arr_E[Y, X])
          def o_e(`z-->y`: Arr_E[Z, Y]): Arr_E[Z, X] =
            summon[Category[Arr_E]].o[Z, Y, X](`y-->x`)(`z-->y`)
```

```
        (`gof`(`y-->x` o_c `z-->y`)) ==:
          // definition lift for composedFunctor
          (g(f(`y-->x` o_c `z-->y`))) ==:
          // functorCompositionLaw for f
          (g(f(`y-->x`) o_d f(`z-->y`))) ==:
          // functorCompositionLaw for g
          (g(f(`y-->x`)) o_e g(f(`z-->y`))) ==:
          // definition lift for composedFunctor
          (`gof`(`y-->x`) o_e `gof`(`z-->y`)) ==:
          // done
          qed
```

```
  def functorIdentityProof[Z]: Proof[Arr_E[(G O F)[Z], (G O F)[Z]]] =
```

```
    val `z-c->z` = summon[Category[Arr_C]].`__-->__`[Z]

    val `f[z]-d->f[z]` = summon[Category[Arr_D]].`__-->__`[F[Z]]

    val `(gof)[z]-e->(gof)[z]` = summon[Category[Arr_E]].`__-->__`[(G O F)[Z]]
```

```
    (`gof`(`z-c->z`)) ==:
      // definition lift for composedFunctor
      (g(f(`z-c->z`))) ==:
      // functorIdentityLaw for f
      (g(`f[z]-d->f[z]`)) ==:
      // functorIdentityLaw for g
      (`(gof)[z]-e->(gof)[z]`) ==:
      // done
      qed
```

2.10 Definition

The *identity endofunctor* of category C, $\boldsymbol{I}: C \to C$, defines

$\boldsymbol{I}_{def}-0$ $\boldsymbol{I}(Z) = Z$,

$\boldsymbol{I}_{def}-1$ $\boldsymbol{i}(z{\rightarrow}y) = z{\rightarrow}y$.

2.10.1 `Scala` code

Let

```
package plp.notation

type I = [__] =>> __
```

in

```
package plp.implementation.generic

import plp.notation.{I}

import plp.specification.{Category, EndoFunctor}
```

```
given identityEndoFunctor[Arr[-_, +_]: Category]: EndoFunctor[Arr, I] with
```

```
  def lift[Z, Y]: Arr[Z, Y] => Arr[I[Z], I[Y]] =
    `z-->y` => `z-->y`
```

2.10.2 Scala code

Below is a proof.

```
import plp.notation.{Proof, ==:, qed}
```

```
class IdentityEndoFunctorProofs[Arr[-_, +_]: Category]:
```

```
  def i[Z, Y]: Arr[Z, Y] => Arr[Z, Y] = identityEndoFunctor.lift
```

```
  def functorCompositionProof[Z, Y, X]
      : Arr[Z, Y] => (Arr[Y, X] => Proof[Arr[I[Z], I[X]]]) =
    `z-->y` =>
      `y-->x` =>
```

```
        (i(`y-->x` o `z-->y`)) ==:
          // definition i
          identity(`y-->x` o `z-->y`) ==:
          // definition identity
          (`y-->x` o `z-->y`) ==:
          // definition identity
          (identity(`y-->x`) o identity(`z-->y`)) ==:
          // definition i
          (i(`y-->x`) o i(`z-->y`)) ==:
          // done
          qed
```

```
  def functorIdentityProof[Z]: Proof[Arr[I[Z], I[Z]]] =
```

```
    val `z-->z` = summon[Category[Arr]].`__-->__`[Z]
```

```
    (i(`z-->z`)) ==:
      // definition i
      (identity(`z-->z`)) ==:
      // definition identity
      (`z-->z`) ==:
      // done
      qed
```

2.11 Definition

The *Yoneda functor* for node Z of category C, $\boldsymbol{YF}_Z : C \to \boldsymbol{Fun}$, defines

$\boldsymbol{YF}_Z{-}0$ $\boldsymbol{YF}_Z(Y) = Arr(Z, Y)$,

$\boldsymbol{YF}_Z{-}1$ $\boldsymbol{yf}_z(y{\to}x)(z{\to}y) = y{\to}x \circ z{\to}y$.

2.11.1 Scala code

```
package plp.implementation.generic

import plp.specification.{Category, Functor}

import plp.implementation.specific.{Fun, functionCategory}
```

```
type Yoneda = [Arr[-_, +_]] =>> [Z] =>> [Y] =>> Arr[Z, Y]
```

```
given yonedaFunctor[Arr[-_, +_]: Category, Z]: Functor[Arr, Fun, Yoneda[Arr][Z]]
  with
```

```
  type YF = [Z] =>> [__] =>> Yoneda[Arr][Z][__]

  def lift[Y, X]: Arr[Y, X] => Fun[YF[Z][Y], YF[Z][X]] =
    `y-->x` => `z-->y` => `y-->x` o `z-->y`
```

2.11.2 Scala code

Below is a proof

```
import plp.notation.{Proof, ==:, qed}
```

```
class YonedaFunctorProofs[Z, Arr[-_, +_]: Category]:
```

```
  type YF = [Z] =>> [__] =>> Yoneda[Arr][Z][__]

  def yf[Y, X]: Arr[Y, X] => (YF[Z][Y] => YF[Z][X]) = yonedaFunctor.lift
```

```
  def compositionProof[Y, X, W]
      : (Arr[Y, X], Arr[X, W]) => YF[Z][Y] => Proof[YF[Z][W]] =
    (`y-->x`, `x-->w`) =>
      `z-->y` =>
```

```
        (yf(`x-->w` o `y-->x`)(`z-->y`)) ==:
          // definition yf
          ((`x-->w` o `y-->x`) o `z-->y`) ==:
          // categoryAssociativityLaw for Arr
          (`x-->w` o (`y-->x` o `z-->y`)) ==:
          // definition yf
          (`x-->w` o yf(`y-->x`)(`z-->y`)) ==:
          // definition yf
          (yf(`x-->w`)(yf(`y-->x`)(`z-->y`))) ==:
          // definition o for functionCategory
          ((yf(`x-->w`) o yf(`y-->x`))(`z-->y`)) ==:
          // done
          qed
```

```
  def identityProof[Y]: Arr[Z, Y] => Proof[YF[Z][Y]] =
    `z-->y` =>
```

```
      val `y-->y` = summon[Category[Arr]].`__-->__`[Y]
```

```
      (yf(`y-->y`)(`z-->y`)) ==:
        // definition yf
        (`y-->y` o `z-->y`) ==:
        // categoryLeftIdentityLaw for Arr
        (`z-->y`) ==:
        // done
        qed
```

2.12 Definition

The *natural transformation specification* $\tau : F \to G$, where $F : C \to D$ and $G : C \to D$ are functors, declares

$\tau_{dec}{-}0$ arrows $\tau_z \in Arr_D(F(Z), G(Z))$.

2.12.1 `Scala` code

```
package plp.specification

trait NaturalTransformation[
    Arr_C[-_, +_]: Category,
    Arr_D[-_, +_]: Category,
    F[+_]: [_[+_]] =>> Functor[Arr_C, Arr_D, F],
    G[+_]: [_[+_]] =>> Functor[Arr_C, Arr_D, G]
]:
```

```
  def transform[__]: Arr_D[F[__], G[__]]
```

τ is declared as an *arrow member* `transform`.

The name `transform` can, by need, be made available as `tau`.

`Scala` allows using τ but, for `LaTeX` typesetting reasons, we use `tau` instead.

2.12.2 Laws

$\tau_{law}-0\;\; \tau_y \circ_d f(z{\rightarrow}y) = g(z{\rightarrow}y) \circ_d \tau_z$ (*commutativity law*).

2.12.3 `Scala` code

```
import plp.notation.{Law, =:}
```

```
class NaturalTransformationLaw[
    Arr_C[-_, +_]: Category,
    Arr_D[-_, +_]: Category,
    F[+_]: [_[+_]] =>> Functor[Arr_C, Arr_D, F],
    G[+_]: [_[+_]] =>> Functor[Arr_C, Arr_D, G],
    T[_[-_, +_], _[-_, +_], _[+_], _[+_]]: [_[
        _[-_, +_],
        _[-_, +_],
        _[+_],
        _[+_]
    ]] =>> NaturalTransformation[Arr_C, Arr_D, F, G]
]:
```

```
  def naturalTransformationLaw[Z, Y]: Arr_C[Z, Y] => Law[Arr_D[F[Z], G[Y]]] =
    `z-->y` =>
```

```
def tau[_]: Arr_D[F[_], G[_]] =
  summon[NaturalTransformation[Arr_C, Arr_D, F, G]].transform

val f: Arr_C[Z, Y] => Arr_D[F[Z], F[Y]] =
  summon[Functor[Arr_C, Arr_D, F]].lift

val g: Arr_C[Z, Y] => Arr_D[G[Z], G[Y]] =
  summon[Functor[Arr_C, Arr_D, G]].lift

extension [Z, Y, X](`y-->x`: Arr_D[Y, X])
  def o_d(`z-->y`: Arr_D[Z, Y]): Arr_D[Z, X] =
    summon[Category[Arr_D]].o(`y-->x`)(`z-->y`)
```

```
(tau o_d f(`z-->y`)) =:
  (g(`z-->y`) o_d tau)
```

Note that, using programmatic notation, we twice used `tau` while, using mathematical notation, we used τ_z and τ_y. In fact, we could have used `tau[Z]` and `tau[Y]` as in

```
(tau[Y] o_d f(`z-->y`)) =:
  (g(`z-->y`) o_d tau[Z])
```

but the `Scala` type system is clever enough to infer the types `Z` and `Y` involved.

2.12.4 Notation

τ_z is also simply denoted τ if z can be inferred.

2.13 Definition

Given

- a category C,
- endofunctors, $F : C \to C$, $G : C \to C$, $H : C \to C$,
- natural transformations $\alpha : F \to G$ and $\beta : H \to K$,

the *composed natural transformation* $\beta \circ \alpha : C \to C$ defines

$(\beta \circ \alpha)_{def} -0 \ \ (\beta \circ \alpha)_z = \beta_z \circ \alpha_z$.

2.13.1 `Scala` code

```
package plp.implementation.generic
```

```
import plp.specification.{Category, Functor, NaturalTransformation}
```

```
def composedNaturalTransformation[
    Arr[-_, +_]: Category,
    F[+_]: [_[+_]] =>> Functor[Arr, Arr, F],
    G[+_]: [_[+_]] =>> Functor[Arr, Arr, G],
    H[+_]: [_[+_]] =>> Functor[Arr, Arr, H],
    S[_[-_, +_], _[-_, +_], _[+_], _[+_]]: [_[
        _[-_, +_],
        _[-_, +_],
        _[+_],
        _[+_]
    ]] =>> NaturalTransformation[Arr, Arr, F, G],
    T[_[-_, +_], _[-_, +_], _[+_], _[+_]]: [_[
        _[-_, +_],
        _[-_, +_],
        _[+_],
        _[+_]
    ]] =>> NaturalTransformation[Arr, Arr, G, H]
]: NaturalTransformation[Arr, Arr, F, H] =
```

```
  def alpha[__] = summon[NaturalTransformation[Arr, Arr, F, G]].transform[__]

  def beta[__] = summon[NaturalTransformation[Arr, Arr, G, H]].transform[__]
```

```
  new {
    override def transform[__]: Arr[F[__], H[__]] =
      beta o alpha
  }
```

2.13.2 Scala code

Below is a proof

```
import plp.notation.{Proof, ==:, qed}
```

```
class ComposedNaturalTransformationProof[
    Arr[-_, +_]: Category,
    F[+_]: [_[+_]] =>> Functor[Arr, Arr, F],
    G[+_]: [_[+_]] =>> Functor[Arr, Arr, G],
    H[+_]: [_[+_]] =>> Functor[Arr, Arr, H],
    S[_[-_, +_], _[-_, +_], _[+_], _[+_]]: [_[
        _[-_, +_],
        _[-_, +_],
        _[+_],
        _[+_]
    ]] =>> NaturalTransformation[Arr, Arr, F, G],
    T[_[-_, +_], _[-_, +_], _[+_], _[+_]]: [_[
        _[-_, +_],
        _[-_, +_],
        _[+_],
        _[+_]
    ]] =>> NaturalTransformation[Arr, Arr, G, H]
]:
```

```
  def naturalTransformationProof[Z, Y]: Arr[Z, Y] => Proof[Arr[F[Z], H[Y]]] =
```

```
    def alpha[__] = summon[NaturalTransformation[Arr, Arr, F, G]].transform[__]

    def beta[__] = summon[NaturalTransformation[Arr, Arr, G, H]].transform[__]

    def tau[__] =
      composedNaturalTransformation[Arr, F, G, H, S, T].transform[__]
```

```
    val f: Arr[Z, Y] => Arr[F[Z], F[Y]] =
      summon[Functor[Arr, Arr, F]].lift

    val g: Arr[Z, Y] => Arr[G[Z], G[Y]] =
      summon[Functor[Arr, Arr, G]].lift

    val h: Arr[Z, Y] => Arr[H[Z], H[Y]] =
      summon[Functor[Arr, Arr, H]].lift
```

```
    `z-->y` =>
      (tau o f(`z-->y`)) ==:
        // definition tau
        ((beta o alpha) o f(`z-->y`)) ==:
        // categoryCompositionAssociativityLaw for Arr
        (beta o (alpha o f(`z-->y`))) ==:
        // naturalTransformationLaw for S
        (beta o (g(`z-->y`) o alpha)) ==:
        // categoryCompositionAssociativityLaw for Arr
        ((beta o g(`z-->y`)) o alpha) ==:
        // naturalTransformationLaw for T
        ((h(`z-->y`) o beta) o alpha) ==:
        // categoryCompositionAssociativityLaw for Arr
        (h(`z-->y`) o (beta o alpha)) ==:
        // definition tau
        (h(`z-->y`) o tau) ==:
        // done
        qed
```

2.14 Definition

Given

- categories C, D and E,
- functors, $F : C \to D$, $H : D \to E$ and $K : D \to E$,
- natural transformations $\alpha : F \to G$ and $\beta : H \to K$,

natural transformation $\beta F : H \circ F \to K \circ F$ defines

$(\beta F)_{def}\!-\!0\ \ (\beta F)_z = \beta_{f(z)}$

2.14.1 `Scala` code

```
package plp.implementation.generic
```

```
import plp.notation.{O}

import plp.specification.{Category, Functor, NaturalTransformation}
```

```
def leftFunctorComposedNaturalTransformation[
    Arr_C[-_, +_]: Category,
    Arr_D[-_, +_]: Category,
    Arr_E[-_, +_]: Category,
    F[+_]: [_[+_]] =>> Functor[Arr_C, Arr_D, F],
    H[+_]: [_[+_]] =>> Functor[Arr_D, Arr_E, H],
    K[+_]: [_[+_]] =>> Functor[Arr_D, Arr_E, K],
    S[_[-_, +_], _[-_, +_], _[+_], _[+_]]: [_[
        _[-_, +_],
        _[-_, +_],
        _[+_],
        _[+_]
    ]] =>> NaturalTransformation[Arr_D, Arr_E, H, K]
]: NaturalTransformation[Arr_C, Arr_E, H O F, K O F] =
```

```
  given `hof`: Functor[Arr_C, Arr_E, H O F] =
    composedFunctor[Arr_C, Arr_D, Arr_E, F, H]

  given `kof`: Functor[Arr_C, Arr_E, K O F] =
    composedFunctor[Arr_C, Arr_D, Arr_E, F, K]

  def beta[__] = summon[NaturalTransformation[Arr_D, Arr_E, H, K]].transform[__]
```

```
  new {
    override def transform[__]: Arr_E[(H O F)[__], (K O F)[__]] =
      beta[F[__]]
  }
```

2.14.2 Scala code

Below is a proof.

```
import plp.notation.{Proof, ==:, qed}

import plp.implementation.specific.{functionCategory}
```

```
class LeftFunctorComposedNaturalTransformationProof[
    Arr_C[-_, +_]: Category,
    Arr_D[-_, +_]: Category,
    Arr_E[-_, +_]: Category,
    F[+_]: [_[+_]] =>> Functor[Arr_C, Arr_D, F],
    H[+_]: [_[+_]] =>> Functor[Arr_D, Arr_E, H],
    K[+_]: [_[+_]] =>> Functor[Arr_D, Arr_E, K],
    S[_[-_, +_], _[-_, +_], _[+_], _[+_]]: [_[
        _[-_, +_],
        _[-_, +_],
        _[+_],
        _[+_]
    ]] =>> NaturalTransformation[Arr_D, Arr_E, H, K]
]:
```

```
  def naturalTransformationProof[Z, Y]
      : Arr_C[Z, Y] => Proof[Arr_E[(H O F)[Z], (K O F)[Y]]] =
    `z-->y` =>
```

```
      def beta[__] =
        summon[NaturalTransformation[Arr_D, Arr_E, H, K]].transform[__]

      def tau[__] =
        leftFunctorComposedNaturalTransformation[
          Arr_C,
          Arr_D,
          Arr_E,
          F,
          H,
          K,
          S
        ]
          .transform[__]
```

```
    def f[Z, Y]: Arr_C[Z, Y] => Arr_D[F[Z], F[Y]] =
      summon[Functor[Arr_C, Arr_D, F]].lift

    def h[Z, Y]: Arr_D[Z, Y] => Arr_E[H[Z], H[Y]] =
      summon[Functor[Arr_D, Arr_E, H]].lift

    def k[Z, Y]: Arr_D[Z, Y] => Arr_E[K[Z], K[Y]] =
      summon[Functor[Arr_D, Arr_E, K]].lift
```

```
      (tau o (h o f)(`z-->y`)) ==:
        // definition tau
        (beta[F[Y]] o (h o f)(`z-->y`)) ==:
        // definition o for functionCategory
        (beta[F[Y]] o h(f(`z-->y`))) ==:
        // naturalTransformationLaw for S (with f(`z-->y`))
        (k(f(`z-->y`)) o beta[F[Z]]) ==:
        // definition o for functionCategory
        ((k o f)(`z-->y`) o beta[F[Z]]) ==:
        // definition tau
        ((k o f)(`z-->y`) o tau) ==:
        // done
        qed
```

2.15 Definition

Given

- categories C, D and E,
- functors, $H : D \to E$, $F : C \to D$ and $G : C \to D$,
- natural transformations $\alpha : F \to G$ and $\beta : H \to K$,

natural transformation $H\alpha : H \circ F \to H \circ G$ defines

$(H\alpha)_{def}{-0}\ \ (H\alpha)_z = h(\alpha_z)$.

2.15.1 Scala code

```
package plp.implementation
```

```
import plp.notation.{O}

import plp.specification.{Category, Functor, NaturalTransformation}
```

```
def rightFunctorComposedNaturalTransformation[
    Arr_C[-_, +_]: Category,
    Arr_D[-_, +_]: Category,
    Arr_E[-_, +_]: Category,
    H[+_]: [_[+_]] =>> Functor[Arr_D, Arr_E, H],
    F[+_]: [_[+_]] =>> Functor[Arr_C, Arr_D, F],
    G[+_]: [_[+_]] =>> Functor[Arr_C, Arr_D, G],
    T[_[-_, +_], _[-_, +_], _[+_], _[+_]]: [_[
        _[-_, +_],
        _[-_, +_],
        _[+_],
        _[+_]
    ]] =>> NaturalTransformation[Arr_C, Arr_D, F, G]
]: NaturalTransformation[Arr_C, Arr_E, H O F, H O G] =
```

```
  given `hof`: Functor[Arr_C, Arr_E, H O F] =
    composedFunctor[Arr_C, Arr_D, Arr_E, F, H]

  given `hog`: Functor[Arr_C, Arr_E, H O G] =
    composedFunctor[Arr_C, Arr_D, Arr_E, G, H]

  def alpha[__]: Arr_D[F[__], G[__]] =
    summon[NaturalTransformation[Arr_C, Arr_D, F, G]].transform[__]

  def h[Z, Y]: Arr_D[Z, Y] => Arr_E[H[Z], H[Y]] =
    summon[Functor[Arr_D, Arr_E, H]].lift
```

```
  new {
    override def transform[__]: Arr_E[(H O F)[__], (H O G)[__]] =
      h(alpha[__])
  }
```

2.15.2 Scala code

Below is a proof.

```
import plp.notation.{Proof, ==:, qed}

import plp.implementation.specific.{functionCategory}
```

```
class RightFunctorComposedNaturalTransformationProof[
    Arr_C[-_, +_]: Category,
    Arr_D[-_, +_]: Category,
    Arr_E[-_, +_]: Category,
    H[+_]: [_[+_]] =>> Functor[Arr_D, Arr_E, H],
    F[+_]: [_[+_]] =>> Functor[Arr_C, Arr_D, F],
    G[+_]: [_[+_]] =>> Functor[Arr_C, Arr_D, G],
    T[_[-_, +_], _[-_, +_], _[+_], _[+_]]: [_[
        _[-_, +_],
        _[-_, +_],
        _[+_],
        _[+_]
    ]] =>> NaturalTransformation[Arr_C, Arr_D, F, G]
]:
```

```
def naturalTransformationProof[Z, Y]
    : Arr_C[Z, Y] => Proof[Arr_E[(H O F)[Z], (H O G)[Y]]] =
  `z-->y` =>
```

```
    def alpha[__]: Arr_D[F[__], G[__]] =
      summon[NaturalTransformation[Arr_C, Arr_D, F, G]].transform[__]

    def tau[__] =
      rightFunctorComposedNaturalTransformation[
        Arr_C,
        Arr_D,
        Arr_E,
        H,
        F,
        G,
        T
      ]
        .transform[__]
```

```
    def h[Z, Y]: Arr_D[Z, Y] => Arr_E[H[Z], H[Y]] =
      summon[Functor[Arr_D, Arr_E, H]].lift

    def f[Z, Y]: Arr_C[Z, Y] => Arr_D[F[Z], F[Y]] =
      summon[Functor[Arr_C, Arr_D, F]].lift

    def g[Z, Y]: Arr_C[Z, Y] => Arr_D[G[Z], G[Y]] =
      summon[Functor[Arr_C, Arr_D, G]].lift
```

```
    (tau o (h o f)(`z-->y`)) ==:
      // definition tau
      (h(alpha) o (h o f)(`z-->y`)) ==:
      // definition o for functionCategory
      (h(alpha) o h(f(`z-->y`))) ==:
      // functorCompositionLaw for h
      (h(alpha o f(`z-->y`))) ==:
      // naturalTransformationLaw for T
      (h(g(`z-->y`) o alpha)) ==:
      // functorCompositionLaw for h
      (h(g(`z-->y`)) o h(alpha)) ==:
      // definition o for functionCategory
      ((h o g)(`z-->y`) o h(alpha)) ==:
      // definition tau
      ((h o g)(`z-->y`) o tau) ==:
      // done
      qed
```

2.16 Definition

The *pre-triple specification* for category C, (PT, η), declares

$\pi\tau_{dec}-0$ an endofunctor $PT : C \to C$,

$\pi\tau_{dec}-1$ a natural transformation $\eta : \boldsymbol{I} \to PT$.

η is called the *unit* of the pre-triple.

2.16.1 Scala code

```
package plp.specification

import plp.notation.{I}
```

```
trait PreTriple[
    Arr[-_, +_]: Category,
    PT[+_]: [_[+_]] =>> EndoFunctor[Arr, PT]
]:
```

```
  val etaNaturalTransformation: NaturalTransformation[Arr, Arr, I, PT]

  def eta[__]: Arr[I[__], PT[__]] = etaNaturalTransformation.transform
```

PT is declared as a *type constructor parameter* `PT`.

η is declared as a *natural transformation member* `etaNaturalTransformation` and corresponding defined *arrow member* `eta`.

2.17 Definition

The *triple specification* for category C, (T, μ, η), declares

$\tau_{dec}-0$ a pre-triple (T, η)

$\tau_{dec}-1$ a natural transformation $\mu : T \circ T \to T$.

μ is called the *multiplication* of the triple.

2.17.1 Scala code

```
package plp.specification

import plp.notation.{O}
```

```
trait Triple[Arr[-_, +_]: Category, T[+_]: [_[+_]] =>> EndoFunctor[Arr, T]]
    extends PreTriple[Arr, T]:
```

```
  def t[Z, Y]: Arr[Z, Y] => Arr[T[Z], T[Y]] = summon[EndoFunctor[Arr, T]].lift

  val muNaturalTransformation: NaturalTransformation[Arr, Arr, T O T, T]

  def mu[__]: Arr[(T O T)[__], T[__]] = muNaturalTransformation.transform
```

μ is declared as a *natural transformation member* `muNaturalTransformation` and corresponding defined *arrow member* `mu`.

2.17.2 Laws

In the laws below, the unit of a category, when treated as a natural transformation, is denoted υ.

$\tau_{law}-0$ $\mu \circ (\eta T) = \upsilon$ (*left identity law*),

$\tau_{law}-1$ $\mu \circ (T \eta) = \upsilon$ (*right identity law*),

$\tau_{law}-2$ $\mu \circ (T \mu) = \mu \circ (\mu T)$ (*associativity law*).

2.17.3 `Scala` code

```
import plp.notation.{Law, =:}
```

```
class TripleLaws[Arr[-_, +_]: Category, T[+_]: [_[+_]] =>> Triple[Arr, T]]:
```

```
  val c: Category[Arr] = summon[Category[Arr]]

  import c.{o, unit => upsilon}

  val triple: Triple[Arr, T] = summon[Triple[Arr, T]]

  import triple.{t, eta, mu}
```

```
  def tripleLeftIdentityLaw[__]: Law[Arr[T[__], T[__]]] =
    (mu o eta) =:
      (upsilon)
```

```
  def tripleRightIdentityLaw[__]: Law[Arr[T[__], T[__]]] =
    (mu o t(eta)) =:
      (upsilon)
```

```
def tripleAssociativityLaw[__]: Law[Arr[(T O T O T)[__], T[__]]] =
  (mu o mu) =:
    (mu o t(mu))
```

Note that the first and last laws could have been

```
  (mu o eta[T[__]]) =:
    (upsilon)
```

and

```
  (mu o mu[T[__]]) =:
    (mu o t(mu))
```

but the `Scala` type system is clever enough to infer the types `T[__]` involved.

2.18 Definition

The *pre-functional category specification*, *PFC*, extends the category specification.

PFC specifies categories whose nodes are the nodes of ***Fun***, in other words, the class of all sets.

It declares

$PFC_{dec}-0$ *being a functor*, $F2A : \boldsymbol{Fun} \rightarrow PFC$, that is the identity on nodes, turning functions into arrows.

It defines

$PFC_{def}-0$ *being a pre-triple*, $(\boldsymbol{YEF}_U, \eta)$, for PFC, where U is a terminal node (singleton set) of the category ***Fun***, $\boldsymbol{YEF}_Z = F2A \circ \boldsymbol{YF}_Z$ is the *Yoneda endofunctor for* Z, and where $\eta_y \in Arr(\boldsymbol{I}(Y), \boldsymbol{YEF}_U(Y)) = f2a(v2gv)$, where $v2gv \in \boldsymbol{Fun}(\boldsymbol{I}(Y), \boldsymbol{YEF}_Z(Y)) = y \mapsto f2a(z \mapsto y)$ turns values into *global values* of ***Fun***.

Since $v2gv$ is defined in terms of $f2a$, which is declared, we use *math* font for it. Since η is defined in terms of $f2a$ and $v2gv$ we use *math* font for it.

From a functional programming viewpoint, it is instructive to think about arrows $f2a(z \Rightarrow y)$ as *pure functions* (a.k.a *effectfree arrows*) and other arrows as *impure functions* (a.k.a. *effectful arrows*).

2.18.1 `Scala` code

Let

```
package plp.notation

type U = Unit
```

in

```
package plp.specification

import plp.notation.{O, I, U}

import plp.implementation.specific.{Fun}

import plp.implementation.generic.{
  identityEndoFunctor,
  Yoneda,
  yonedaFunctor,
  yonedaEndoFunctor
}
```

```
trait PreFunctionalCategory[
    Arr[-_, +_]: Category: [_[-_, +_]] =>> Functor[Fun, Arr, I]
] extends Category[Arr]
    with PreTriple[Arr, Yoneda[Arr][U]]:
```

```
  def f2a[Z, Y]: Fun[Z, Y] => Arr[Z, Y] = summon[Functor[Fun, Arr, I]].lift

  def v2gv[__]: Fun[__, Arr[U, __]] = __ => f2a(_ => __)

  val etaNaturalTransformation: NaturalTransformation[Arr, Arr, I, YEF[U]] =
    given PreFunctionalCategory[Arr] = this
    new:
      def transform[__]: Arr[I[__], YEF[U][__]] =
        f2a(v2gv)
```

Being a functor *F2A* is declared as a *type class extension* for `Arr` using `[_[-_, +_]] =>> Functor[Fun, Arr, I]`.

Auxiliary members `f2a` and `v2gv`, corresponding with *f2a* and *v2gv* are defined as well.

Being a pre-triple $(\boldsymbol{YEF}_U, \eta)$ is defined as an *object-oriented extension* using `with PreTriple[Arr, Yoneda[Arr][U]]`, and by defining η as a *natural transformation member* `etaNaturalTransformation`, as such also defining `eta`.

```
  type YF = [Z] =>> [__] =>> Yoneda[Arr][Z][__]

  def yf[Z, Y, X]: Arr[Y, X] => Fun[YF[Z][Y], YF[Z][X]] =
    yonedaFunctor[Arr, Z].lift

  type YEF = [Z] =>> [__] =>> Yoneda[Arr][Z][__]

  def yef[Z, Y, X]: Arr[Y, X] => Arr[YEF[Z][Y], YEF[Z][X]] =
    given PreFunctionalCategory[Arr] = this
    yonedaEndoFunctor[Arr, Z].lift
```

Auxiliary type members `YF` and `YEF`, function member `yf` and arrow member `yef` are defined as well.

The code above makes use of `yonedaEndoFunctor` whose code is below.

2.18.2 `Scala` code

```
package plp.implementation.generic

import plp.specification.{EndoFunctor, PreFunctionalCategory}
```

```
given yonedaEndoFunctor[Arr[-_, +_]: PreFunctionalCategory, Z]
    : EndoFunctor[Arr, Yoneda[Arr][Z]] with
```

```
  val pfc = summon[PreFunctionalCategory[Arr]]

  import pfc.{YEF}
```

```
  def lift[Y, X]: Arr[Y, X] => Arr[YEF[Z][Y], YEF[Z][X]] =
    `y-->x` =>
```

```
      import pfc.{f2a, yf, YEF}
```

```
      f2a(yf(`y-->x`))
```

2.18.3 Laws

$PFC_{law} - 0$ $F2A$ is a functor,

$PFC_{law} - 1$ $(\boldsymbol{YEF}_U, \eta)$ is a pre-triple, in other words, $\eta : \boldsymbol{I} \to \boldsymbol{YEF}_U$ is a natural transformation,

$PFC_{law} - 2$ $\eta \circ u{\to}z = v2gv(u{\to}z)$ $(\eta\ law)$.

2.18.4 `Scala` code

```
import plp.notation.{Law, =:}
```

```
class PreFunctionalCategoryLaws[Arr[-_, +_]: PreFunctionalCategory]:
```

```
import plp.implementation.{functionCategory}

val pfc = summon[PreFunctionalCategory[Arr]]
```

```
def preFunctionalCategoryFunctorCompositionLaw[Z, Y, X]
    : Fun[Z, Y] => (Fun[Y, X] => Law[Arr[Z, X]]) =
  `z=>y` =>
    `y=>x` =>
```

```
      import pfc.{f2a}
```

```
      (f2a(`y=>x` o `z=>y`)) =:
        (f2a(`y=>x`) o f2a(`z=>y`))
```

```
def preFunctionalCategoryFunctorIdentityLaw[__]: Law[Arr[__, __]] =
```

```
import functionCategory.{`__-->__` => `__-f->__`}

import pfc.{`__-->__`, f2a}
```

```
f2a(`__-f->__`) =:
  `__-->__`
```

```
import pfc.{YEF}
```

```
def preFunctionalCategoryEtaNaturalTransformationLaw[Z, Y]
    : Arr[Z, Y] => Law[Arr[I[Z], YEF[U][Y]]] =
  `z-->y` =>
```

```
    import pfc.{yef, eta}

    val i: Arr[Z, Y] => Arr[Z, Y] = identityEndoFunctor[Arr].lift
```

```
    (eta o i(`z-->y`)) =:
      (yef(`z-->y`) o eta)
```

```
  def preFunctionalCategoryEtaLaw[__]: YEF[U][__] => Law[(YEF[U] O YEF[U])[__]] =
    `u-->__` =>
```

```
      import pfc.{eta, v2gv}
```

```
      (eta o `u-->__`) =:
        (v2gv(`u-->__`))
```

2.18.5 Properties

The following pre-functional category properties hold

$\boldsymbol{YEF}{-}0$ $f2a(z{\Rightarrow}y) \circ v2gv(z) = v2gv(z{\Rightarrow}y(z))$ $(pointfree\ application)$,

$\boldsymbol{YEF}{-}1$ $\boldsymbol{yef}(y{\rightarrow}x) \circ v2gv(z{\rightarrow}y) = v2gv(y{\rightarrow}x \circ z{\rightarrow}y)$ $(pointfree\ Yoneda)$.

2.18.6 `Scala` code

```
package plp.proposition

import plp.notation.{U, Law, =:}

import plp.specification.{PreFunctionalCategory}

import plp.implementation.specific.{Fun}
```

```
class PreFunctionalCategoryProperties[Arr[-_, +_]: PreFunctionalCategory]:
```

```
  val pfc = summon[PreFunctionalCategory[Arr]]

  import pfc.{YEF}
```

```
  def pointfreeApplicationProperty[Z, Y]: Fun[Z, Y] => (Z => Law[YEF[U][Y]]) =
    `z=>y` =>
      z =>
```

```
        import pfc.{v2gv, f2a}
```

```
        (f2a(`z=>y`) o v2gv(z)) =:
          (v2gv(`z=>y`(z)))
```

```
  def pointfreeYonedaProperty[Z, Y, X]
      : Arr[Z, Y] => (Arr[Y, X] => Law[YEF[U][Arr[Z, X]]]) =
    `z-->y` =>
      `y-->x` =>
```

```
        import pfc.{v2gv, yef}
```

```
        (yef(`y-->x`) o v2gv(`z-->y`)) =:
          (v2gv(`y-->x` o `z-->y`))
```

2.18.7 Scala code

Below is a proof.

```
import plp.notation.{Proof, ==:, qed}

import plp.implementation.specific.{functionCategory}
```

```
class PreFunctionalCategoryProofs[Z, Arr[-_, +_]: PreFunctionalCategory]:
```

```
  val pfc = summon[PreFunctionalCategory[Arr]]

  import pfc.{YEF}
```

```
  def pointfreeApplicationProof[Z, Y]: Fun[Z, Y] => (Z => Proof[YEF[U][Y]]) =
    `z=>y` =>
      z =>
```

```
        import pfc.{f2a, v2gv}
```

```
      (f2a(`z=>y`) o v2gv(z)) ==:
        // definition v2gv
        (f2a(`z=>y`) o f2a(_ => z)) ==:
        // functorCompositionLaw for f2a
        (f2a(`z=>y` o (_ => z))) ==:
        // definition o for functionCategory
        (f2a(_ => `z=>y`(z))) ==:
        // definition v2gv
        (v2gv(`z=>y`(z))) ==:
        // done
        qed
```

```
  def pointfreeYonedaProof[Y, X]
      : Arr[Z, Y] => (Arr[Y, X] => Proof[YEF[U][Arr[Z, X]]]) =
    `z-->y` =>
      `y-->x` =>
```

```
      import pfc.{yf, yef, v2gv}

      def f2a[__] = pfc.f2a[Arr[__, Y], Arr[__, X]]
```

```
      (yef(`y-->x`) o v2gv(`z-->y`)) ==:
        // definition yef and definition f2a
        (f2a[Z](yf(`y-->x`)) o v2gv(`z-->y`)) ==:
        // definition yf
        f2a[Z](`z-->y` => `y-->x` o `z-->y`) o v2gv(`z-->y`))
        // pointfreeApplicationProperty and definition f2a
        (v2gv(`y-->x` o `z-->y`)) ==:
        // done
        qed
```

2.19 Definition

The *functional category specification FC* extends the pre-functional category specification.

It declares

$FC_{dec}-0$ *being a triple*, $(\boldsymbol{YEF}_U, \mu, \eta)$, for *PFC*.

2.19.1 Scala code

```
package plp.specification
```

```
import plp.notation.{U}

import plp.implementation.generic.{Yoneda}
```

```
trait FunctionalCategory[Arr[-_, +_]: Category]
    extends PreFunctionalCategory[Arr]
    with Triple[Arr, Yoneda[Arr][U]]
```

Being a triple $(\boldsymbol{YEF}_U, \mu, \eta)$ is declared as an *object-oriented extension* using `with Triple[Arr, Yoneda[Arr][U]]`.

2.19.2 Laws

$FC_{law}-0$ $(\boldsymbol{YEF}_U, \mu, \eta)$ is a triple.

2.19.3 `Scala` code

```
import plp.notation.{Law, =:}

import plp.implementation.specific.{functionCategory}
```

```
class FunctionalCategoryLaws[Arr[-_, +_]: FunctionalCategory]:
```

```
  val fc = summon[FunctionalCategory[Arr]]

  import fc.{YEF}
```

```
  def functionalCategoryMuNaturalTransformationLaw[Z, Y]
      : Arr[Z, Y] => Law[Arr[(YEF[U] O YEF[U])[Z], YEF[U][Y]]] =
    `z-->y` =>
```

```
      import fc.{yef, mu}
```

```
      (mu o (yef o yef)(`z-->y`)) =:
        (yef(`z-->y`) o mu)
```

```
def functionalCategoryTripleLeftIdentityLaw[__]: Law[Arr[YEF[U][__], YEF[U][__]]] =
```

```
  import fc.{unit => upsilon, eta, mu}
```

```
  (mu o eta) =:
    (upsilon)
```

```
def functionalCategoryTripleRightIdentityLaw[__]: Law[Arr[YEF[U][__], YEF[U][__]]] =
```

```
  import fc.{unit => upsilon, yef, eta, mu}
```

```
  (mu o yef(eta)) =:
    (upsilon)
```

```
def functionalCategoryTripleAssociativityLaw[__]
    : Law[Arr[(YEF[U] O YEF[U] O YEF[U])[__], YEF[U][__]]] =
```

```
    import fc.{yef, mu}
```

```
    (mu o mu) =:
      (mu o yef(mu))
```

2.19.4 Properties

The following functional category property holds

$\mu_{prop}-0 \;\; \mu \circ v2gv(u{\to}(u{\to}z)) = u{\to}(u{\to}z) \;\; (\mu \;\; property)$

2.19.5 Scala code

```
package plp.proposition

import plp.notation.{O, U, Law, =:}

import plp.specification.{FunctionalCategory}
```

```
class FunctionalCategoryProperties[Z, Arr[-_, +_]: FunctionalCategory]:
```

```
  val fc = summon[FunctionalCategory[Arr]]

  import fc.{YEF}
```

```
  def functionalCategoryMuProperty[__]
      : (YEF[U] O YEF[U])[__] => Law[(YEF[U] O YEF[U])[__]] =
    `u-->(u-->__)` =>
```

```
      import fc.{v2gv, mu}
```

```
      (mu o v2gv(`u-->(u-->__)`)) =:
        (`u-->(u-->__)`)
```

2.19.6 Scala code

Below is a proof.

```
import plp.notation.{Proof, ==:, qed}
```

```
class FunctionalCategoryProoof[Z, Arr[-_, +_]: FunctionalCategory]:
```

```
  val fc = summon[FunctionalCategory[Arr]]

  import fc.{YEF}
```

```
  def functionalCategoryMuProof[__]
      : (YEF[U] O YEF[U])[__] => Proof[(YEF[U] O YEF[U])[__]] =
    `u-->(u-->__)` =>
```

```
      import fc.{unit, v2gv, eta, mu}
```

```
      def upsilon[__] = unit[__]
```

```
      (mu o v2gv(`u-->(u-->__)`)) ==:
        // preFunctionalCategoryEtaLaw for YEF[U]
        (mu o (eta o `u-->(u-->__)`)) ==:
        // categoryCompositionAssociativityLaw for Arr
        ((mu o eta) o `u-->(u-->__)`) ==:
        // tripleLeftIdentityLaw for YEF[U]
        (`u-->(u-->__)` o upsilon) ==:
        // definition upsilon
        (`u-->(u-->__)` o unit) ==:
        // categoryRightIdentityLaw for Arr
        (`u-->(u-->__)`) ==:
        qed
```

2.20 Definition

The *function functional category*, ***Fun***, defines

$\boldsymbol{Fun}_{def}-0$ $F2A : \boldsymbol{Fun} \to \boldsymbol{Fun}$ as the *identity function*,

$\boldsymbol{Fun}_{def}-1$ $\mu : \boldsymbol{YEF}_U \circ \boldsymbol{YEF}_U \to \boldsymbol{YEF}_U$ as *application to u*, where u is the unique element of U.

2.20.1 `Scala` code

Let

```
package plp.notation

val u: U = ()
```

in

```
package plp.implementation.specific

import plp.notation.{O, I, U, u}

import plp.specification.{FunctionalCategory, NaturalTransformation}

import plp.implementation.generic.{
  identityEndoFunctor,
  composedFunctor,
  yonedaEndoFunctor
}
```

```
given functionFunctionalCategory: FunctionalCategory[Fun] with
```

```
  def composition[Z, Y, X]: (Fun[Y, X], Fun[Z, Y]) => Fun[Z, X] =
    functionCategory.composition

  def unit[Z]: Fun[Z, Z] = functionCategory.unit

  def lift[Z, Y]: Fun[Z, Y] => Fun[I[Z], I[Y]] = identityEndoFunctor.lift

  val muNaturalTransformation
      : NaturalTransformation[Fun, Fun, YEF[U] O YEF[U], YEF[U]] =
    new:
      def transform[__]: Fun[(YEF[U] O YEF[U])[__], YEF[U][__]] =
        `u-->(u-->__)` => `u-->(u-->__)`(u)
```

2.20.2 Scala code

Below is a proof

```
import plp.notation.{Proof, ==:, qed}

import plp.implementation.specific.{functionCategory}
```

```
class FunctionFunctionalCategoryProofs:
```

```
  def functionPreFunctionalCategoryFunctorCompositionProof[Z, Y, X]
      : Fun[Z, Y] => (Fun[Y, X] => Proof[Fun[Z, X]]) =
    `z=>y` =>
      `y=>x` =>
```

```
        import functionFunctionalCategory.{f2a}
```

```
    f2a(`y=>x` o `z=>y`) ==:
      // definition f2a for identityEndoFunctor for Fun
      (`y=>x` o `z=>y`) ==:
      // definition f2a for identityEndoFunctor for Fun
      (f2a(`y=>x`) o f2a(`z=>y`)) ==:
      // done
      qed
```

```
def functionPreFunctionalCategoryFunctorIdentityProof[__]
    : Proof[Fun[__, __]] =
```

```
  import functionCategory.{`__-->__`}

  import functionFunctionalCategory.{f2a}
```

```
  f2a(`__-->__`) ==:
    // definition f2a for identityEndoFunctor for Fun
    `__-->__` ==:
    // done
    qed
```

```
import functionFunctionalCategory.{YEF}
```

```
def functionPreFunctionalCategoryEtaNaturalTransformationProof[Z, Y]
    : Fun[Z, Y] => Proof[Fun[I[Z], YEF[U][Y]]] =
  `z=>y` =>
```

```
    import functionFunctionalCategory.{o, f2a, v2gv, yf, yef, eta}

    def i[Z, Y]: Fun[Z, Y] => Fun[Z, Y] = identityEndoFunctor.lift
```

```
    (yef(`z=>y`) o eta) ==:
      // definition eta for functionFunctionalCategory
      (yef(`z=>y`) o f2a(v2gv)) ==:
      // definition f2a for identityEndoFunctor for Fun
      (yef(`z=>y`) o v2gv) ==:
      // definition v2gv
      (yef(`z=>y`) o ((z: Z) => f2a(_ => z))) ==:
      // definition f2a for identityEndoFunctor for Fun
      (yef(`z=>y`) o ((z: Z) => (_ => z))) ==:
      // definition yef
      (f2a(yf(`z=>y`)) o ((z: Z) => (_ => z))) ==:
      // definition f2a for identityEndoFunctor for Fun
      (yf(`z=>y`) o ((z: Z) => (_ => z))) ==:
      // definition o for functionCategory
      ((z: Z) => yf(`z=>y`)(_ => z)) ==:
      // definition yf
      ((z: Z) => (`z=>y` o (_ => z))) ==:
      // definition o for functionCategory
      ((z: Z) => (_: U) => `z=>y`(z)) ==:
      // definition definition o for functionCategory (substituting `z=>y`(z) for y)
      (((y: Y) => (_: U) => y) o (z => `z=>y`(z))) ==:
      // lambda calculus eta conversion
      (((y: Y) => ((_: U) => y)) o `z=>y`) ==:
      // definition f2a for identityEndoFunctor for Fun
      (((y: Y) => f2a(_ => y)) o `z=>y`) ==:
      // definition v2gv
      (v2gv o `z=>y`) ==:
      // definition f2a for identityEndoFunctor for Fun
      (f2a(v2gv) o `z=>y`) ==:
      // definition eta
      (eta o `z=>y`) ==:
      // definition i
      (eta o i(`z=>y`)) ==:
      // done
      qed
```

```
def functionFunctionalCategoryMuNaturalTransformationProof[Z, Y]
    : Fun[Z, Y] => Proof[(YEF[U] O YEF[U])[Z] => YEF[U][Y]] =
  `z=>y` =>
```

```
    import functionFunctionalCategory.{o, f2a, yf, yef, mu}
```

```
    (yef(`z=>y`) o mu) ==:
      // definition yef
      (f2a(yf(`z=>y`)) o mu) ==:
      // definition f2a for identityEndoFunctor for Fun
      (yf(`z=>y`) o mu) ==:
      // definition mu for functionFunctionalCategory
      (yf(`z=>y`) o (_(u))) ==:
      // definition o for functionCategory
      (`u=>(u=>z)` => yf(`z=>y`)(`u=>(u=>z)`(u))) ==:
      // definition yf
      (`u=>(u=>z)` => (`z=>y` o `u=>(u=>z)`(u))) ==:
      // eta conversion lambda calculus
      (
          (
              (`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) =>
                ((u: U) => (`z=>y` o `u=>(u=>z)`(u)))(u)
            )
      ) ==:
      // definition o for functionCategory
      (((`u=>(u=>y)`: (YEF[U] O YEF[U])[Y]) => `u=>(u=>y)`(u)) o (
        (`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => (u => `z=>y` o `u=>(u=>z)`(u))
      )) ==:
      // definition o for functionCategory
      (((`u=>(u=>y)`: (YEF[U] O YEF[U])[Y]) => `u=>(u=>y)`(u)) o (
        (`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) =>
          ((`u=>z`: YEF[U][Z]) => `z=>y` o `u=>z`) o (u => `u=>(u=>z)`(u))
      )) ==:
      // eta conversion lambda calculus
      (((`u=>(u=>y)`: (YEF[U] O YEF[U])[Y]) => `u=>(u=>y)`(u)) o (
        (`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) =>
          ((`u=>z`: YEF[U][Z]) => `z=>y` o `u=>z`) o `u=>(u=>z)`
      )) ==:
      // // definition yf
      (((`u=>(u=>y)`: (YEF[U] O YEF[U])[Y]) => `u=>(u=>y)`(u)) o yf(
        (`u=>z`: YEF[U][Z]) => `z=>y` o `u=>z`
      )) ==:
      // definition yf
      (((`u=>(u=>y)`: (YEF[U] O YEF[U])[Y]) => `u=>(u=>y)`(u)) o yf(
        yf(`z=>y`)
      )) ==:
      // definition mu for functionFunctionalCategory
      (mu o yf(yf(`z=>y`))) ==:
      // definition f2a for identityEndoFunctor for Fun
      (mu o f2a(yf(f2a(yf(`z=>y`))))) ==:
      // definition yef
      (mu o yef(yef(`z=>y`))) ==:
      // done
      qed
```

```
def functionFunctionalCategoryEtaProof[Z]
    : YEF[U][Z] => Proof[(YEF[U] O YEF[U])[Z]] =
  `u=>z` =>
```

```
    import functionFunctionalCategory.{o, f2a, v2gv, eta}
```

```
    (eta o `u=>z`) ==:
      // definition eta for functionFunctionalCategory
      (f2a(v2gv) o `u=>z`) ==:
      // definition f2a for identityEndoFunctor for Fun
      (v2gv o `u=>z`) ==:
      // definition v2gv
      (((z: Z) => f2a(_ => z)) o `u=>z`) ==:
      // definition f2a for identityEndoFunctor
      (((z: Z) => ((_: U) => z)) o `u=>z`) ==:
      // eta conversion lambda calculus
      (((z: Z) => ((_: U) => z)) o (u => `u=>z`(u))) ==:
      // definition o for functionCategory
      (u => ((_: U) => `u=>z`(u))) ==:
      // eta conversion lambda calculus
      (_ => `u=>z`) ==:
      // definition f2a for identityEndoFunctor for Fun
      (f2a(_ => `u=>z`)) ==:
      // definition v2gv
      v2gv(`u=>z`) ==:
      // done
      qed
```

```
  def functionFunctionalCategoryTripleLeftIdentityProof[Z]
    : Proof[YEF[U][Z] => YEF[U][Z]] =
```

```
   import functionFunctionalCategory.{unit, f2a, v2gv, eta, mu}

   def upsilon[__] = unit[__]
```

```
   (mu o eta) ==:
     // definition eta for functionFunctionalCategory
     (mu o f2a(v2gv)) ==:
     // definition f2a for identityEndoFunctor for Fun
     (mu o v2gv) ==:
     // definition v2gv
     (mu o (`u=>z` => f2a(_ => `u=>z`))) ==:
     // definition f2a for identityEndoFunctor for Fun
     (mu o (`u=>z` => (_ => `u=>z`))) ==:
     // definition mu for functionFunctionalCategory
     (((`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => `u=>(u=>z)`(u)) o (`u=>z` =>
       (_ => `u=>z`)
     )) ==:
     // definition o for functionCategory
     ((`u=>z` => ((_: U) => `u=>z`)(u))) ==:
     // lambda calculus eta conversion (U has only one value, u)
     (`u=>z` => `u=>z`) ==:
     // definition unit for Fun
     unit ==:
     // definition upsilon
     upsilon[YEF[U][Z]] ==:
     // done
     qed
```

```
  def functionFunctionalCategoryTripleRightIdentityProof[Z]
    : Proof[YEF[U][Z] => YEF[U][Z]] =
```

```
import functionFunctionalCategory.{unit, f2a, v2gv, yef, yf, eta, mu}

def upsilon[__] = unit[__]
```

```
(mu o yef(eta)) ==:
  // definition yef
  (mu o f2a(yf(eta))) ==:
  // definition f2a for identityEndoFunctor for Fun
  (mu o yf(eta)) ==:
  // definition eta for functionFunctionalCategory
  (mu o yf(f2a(v2gv))) ==:
  // definition f2a for identityEndoFunctor for Fun
  (mu o yf(v2gv)) ==:
  // definition v2gv
  (mu o yf(f2a(z => (_ => z)))) ==:
  // definition f2a for identityEndoFunctor for Fun
  (mu o yf(z => _ => z)) ==:
  // definition mu for functionFunctionalCategory
  (((`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => `u=>(u=>z)`(u)) o yf(z =>
    (_ => z)
  )) ==:
  // definition yf
  (((`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => `u=>(u=>z)`(u)) o (`u=>z` =>
    ((z: Z) => ((_: U) => z)) o `u=>z`
  )) ==:
  // definition o for functionCategory
  (`u=>z` => (((z: Z) => ((_: U) => z)) o `u=>z`)(u)) ==:
  // definition o for functionCategory
  (`u=>z` => (_ => `u=>z`(u))) ==:
  // lambda calculus eta conversion (U has only one value, u)
  (`u=>z` => `u=>z`) ==:
  // definition unit for Fun
  unit ==:
  // definition upsilon
  upsilon[YEF[U][Z]] ==:
  // done
  qed
```

```
def functionFunctionalCategoryTripleAssociativityProof[Z]
    : Proof[(YEF[U] O YEF[U] O YEF[U])[Z] => YEF[U][Z]] =
```

```
import functionFunctionalCategory.{f2a, yf, yef, mu}
```

```
(mu o mu) ==:
  // definition mu for functionFunctionalCategory
  (((`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => `u=>(u=>z)`(u)) o (
    (`u=>(u=>(u=>z))`: (YEF[U] O YEF[U] O YEF[U])[Z]) => `u=>(u=>(u=>z))`(u)
  )) ==:
  // definition o for functionCategory
  (
      (
          (`u=>(u=>(u=>z))`: (YEF[U] O YEF[U] O YEF[U])[Z]) =>
            `u=>(u=>(u=>z))`(u)(u)
        )
  ) ==:
  // definition o for functionCategory
  (
      (
          (`u=>(u=>(u=>z))`: (YEF[U] O YEF[U] O YEF[U])[Z]) =>
            ((
                (`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => `u=>(u=>z)`(u)
            ) o `u=>(u=>(u=>z))`)(u)
        )
  ) ==:
  // definition o for functionCategory
  (((`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => `u=>(u=>z)`(u)) o (
    (`u=>(u=>(u=>z))`: (YEF[U] O YEF[U] O YEF[U])[Z]) =>
      (
          (`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => `u=>(u=>z)`(u)
      ) o `u=>(u=>(u=>z))`
  )) ==:
  // definition yf
  (((`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => `u=>(u=>z)`(u)) o yf(
    (`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => `u=>(u=>z)`(u)
  )) ==:
  // definition f2a for identityEndoFunctor
  (((`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => `u=>(u=>z)`(u)) o f2a(
    yf((`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => `u=>(u=>z)`(u))
  )) ==:
  // definition yef
  (((`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => `u=>(u=>z)`(u)) o yef(
    (`u=>(u=>z)`: (YEF[U] O YEF[U])[Z]) => `u=>(u=>z)`(u)
  )) ==:
  // definition mu for functionFunctionalCategory
  (mu o yef(mu)) ==:
  // done
  qed
```

3 Yoneda lemmas

3.1 Yoneda lemma for categories

3.2 Lemma

For all categories C, nodes Z of C and functors G from C to $\boldsymbol{Set}$, natural transformations $\tau : \boldsymbol{YF}_Z \to G$ correspond with elements of $G(Z)$.

On the one hand, if $\tau : \boldsymbol{YF}_Z \to G$ is a natural transformation, and $gz \in G(Z)$ is defined as $\tau(z{\rightarrow}z)$, then $\sigma : \boldsymbol{YF}_Z \to G$, defined as $\sigma = z{\rightarrow}y \mapsto g(z{\rightarrow}y)(gz)$, is equal to τ.

On the other hand, if $gz \in G(Z)$, and $\tau : \boldsymbol{YF}_Z \to G$ is defined as $\tau = z{\rightarrow}y \mapsto g(z{\rightarrow}y)(gz)$, then τ is a natural transformation.

3.2.1 Scala code

```
package plp.proposition

import plp.notation.{Law, =:}

import plp.specification.{Category, Functor, NaturalTransformation}

import plp.implementation.specific.{Fun, functionCategory}

import plp.implementation.generic.{Yoneda, yonedaFunctor}
```

```
class YonedaLemma[
    Arr[-_, +_]: Category,
    Z,
    G[+_]: [_[+_]] =>> Functor[Arr, Fun, G]
]:
```

```
  val c = summon[Category[Arr]]

  def g[Z, Y]: Arr[Z, Y] => Fun[G[Z], G[Y]] =
    summon[Functor[Arr, Fun, G]].lift[Z, Y]

  type YF = [Z] =>> [__] =>> Yoneda[Arr][Z][__]
```

```
  def yonedaLemma1[Y]: (
      NaturalTransformation[Arr, Fun, YF[Z], G] => Law[Fun[YF[Z][Y], G[Y]]]
  ) =
    yfz2g =>
```

```
      import c.{`__-->__`}

      def `z-->z`[Z] = `__-->__`[Z]

      def tau[__]: Fun[YF[Z][__], G[__]] = yfz2g.transform

      val `g[z]` : G[Z] = tau(`z-->z`)

      val yfz_2_g: NaturalTransformation[Arr, Fun, YF[Z], G] =
        new:
          def transform[__]: Fun[YF[Z][__], G[__]] =
            `z-->__` => g(`z-->__`)(`g[z]`)

      def sigma[__]: Fun[YF[Z][__], G[__]] = yfz_2_g.transform
```

```
      tau =:
        sigma
```

```scala
  def yonedaLemma2[Y, X]: G[Z] => (Arr[Y, X] => (Law[Fun[YF[Z][Y], G[X]]])) =
    `g[z]` =>

      val yfz2g: NaturalTransformation[Arr, Fun, YF[Z], G] =
        new:
          def transform[__]: Fun[YF[Z][__], G[__]] =
            `z-->__` => g(`z-->__`)(`g[z]`)

      def tau[__]: Fun[YF[Z][__], G[__]] = yfz2g.transform

      val yf: Arr[Y, X] => Fun[YF[Z][Y], YF[Z][X]] = yonedaFunctor.lift

      `y-->x` =>
        (g(`y-->x`) o tau) =:
          (tau o yf(`y-->x`))
```

3.2.2 Scala code

Below is a proof.

```scala
import plp.notation.{Proof, ==:, qed}
```

```scala
class YonedaLemmaProof[
    Z,
    Arr[-_, +_]: Category,
    G[+_]: [_[+_]] =>> Functor[Arr, Fun, G]
]:
```

```scala
  val c = summon[Category[Arr]]

  import c.`__-->__`
```

```scala
  def g[Z, Y]: Arr[Z, Y] => Fun[G[Z], G[Y]] =
    summon[Functor[Arr, Fun, G]].lift[Z, Y]

  type YF = [Z] =>> [__] =>> Yoneda[Arr][Z][__]

  def yf[Y, X]: Arr[Y, X] => Fun[YF[Z][Y], YF[Z][X]] = yonedaFunctor.lift
```

```scala
  def yonedaLemma1Proof[Y]
      : NaturalTransformation[Arr, Fun, YF[Z], G] => (YF[Z][Y] => Proof[G[Y]]) =
    yfz2g =>
```

```
    def `z-->z`[Z] = `__-->__`[Z]

    def tau[__]: Fun[YF[Z][__], G[__]] = yfz2g.transform

    val `g[z]` : G[Z] =
      tau(`z-->z`)

    val yfz_2_g: NaturalTransformation[Arr, Fun, YF[Z], G] =
      new:
        def transform[__]: Fun[YF[Z][__], G[__]] =
          `z-->__` => g(`z-->__`)(`g[z]`)

    def sigma[__]: Fun[YF[Z][__], G[__]] = yfz_2_g.transform
```

```
    `z-->y` =>
      tau(`z-->y`) ==:
        // rightIdentityLaw for Arr
        tau(`z-->y` o `z-->z`) ==:
        // definition yf
        tau(yf(`z-->y`)(`z-->z`)) ==:
        // definition o for functionCategory
        (tau o yf(`z-->y`))(`z-->z`) ==:
        // naturalTransformationLaw for tau
        (g(`z-->y`) o tau)(`z-->z`) ==:
        // definition o for functionCategory
        g(`z-->y`)(tau(`z-->z`)) ==:
        // definition `g[z]`
        g(`z-->y`)(`g[z]`) ==:
        // definition sigma
        sigma(`z-->y`) ==:
        // done
        qed
```

```
def yonedaLemma2Proof[Y, X]
    : G[Z] => (Arr[Y, X] => (YF[Z][Y] => Proof[G[X]])) =
  `g[z]` =>
    `y-->x` =>
```

```
      val yfz2g: NaturalTransformation[Arr, Fun, YF[Z], G] =
        new:
          def transform[__]: Fun[YF[Z][__], G[__]] =
            `z-->__` => g(`z-->__`)(`g[z]`)

      def tau[__]: Fun[YF[Z][__], G[__]] = yfz2g.transform
```

```
    `z-->y` =>
      (g(`y-->x`) o tau)(`z-->y`) ==:
        // definition o for functionCategory
        g(`y-->x`)(tau(`z-->y`)) ==:
        // definition tau
        g(`y-->x`)(g(`z-->y`)(`g[z]`)) ==:
        // definition o for functionCategory
        (g(`y-->x`) o g(`z-->y`))(`g[z]`) ==:
        // compositionLaw for g
        (g(`y-->x` o `z-->y`))(`g[z]`) ==:
        // definition yf
        (g(yf(`y-->x`)(`z-->y`)))(`g[z]`) ==:
        // definition tau
        tau(yf(`y-->x`)(`z-->y`)) ==:
        // definition o for functionCategory
        (tau o yf(`y-->x`))(`z-->y`) ==:
        // done
        qed
```

3.3 Pointfree Yoneda lemma for pre-functional categories

3.4 Lemma

For all pre-functional categories *PFC*, nodes Z of *PFC*, endofunctors G of *PFC* and natural tranformations $\tau : \boldsymbol{YEF}_Z \rightarrow G$, natural transformations $\eta G \circ \tau : \boldsymbol{YEF}_Z \rightarrow \boldsymbol{YEF}_U \circ G$ correspond with arrows of $(\boldsymbol{YEF}_U \circ G)(Z)$.

On the one hand, if $\tau : \boldsymbol{YEF}_Z \rightarrow G$ is a natural transformation, and $u{\rightarrow}g(z) \in (\boldsymbol{YEF}_U \circ G)(Z)$ is defined as $\tau \circ v2gv(z{\rightarrow}z)$, then $\sigma : \boldsymbol{YEF}_Z \rightarrow \boldsymbol{YEF}_U \circ G$, defined as $\sigma = f2a(z{\rightarrow}y \mapsto g(z{\rightarrow}y) \circ u{\rightarrow}g(z))$, is equal to $\eta G \circ \tau$.

On the other hand, if $u{\rightarrow}g(z) \in (\boldsymbol{YEF}_U \circ G)(Z)$, and $\tau : \boldsymbol{YEF}_Z \rightarrow \boldsymbol{YEF}_U \circ G$ is defined as $\tau = f2a(z{\rightarrow}y \mapsto g(z{\rightarrow}y) \circ u{\rightarrow}g(z))$, then τ is a natural transformation.

Both statements hold modulo composing with a global arrow $v2gv(z{\rightarrow}y)$.

3.4.1 Scala code

```
package plp.proposition

import plp.notation.{O, U, Law, =:}

import plp.specification.{
  PreFunctionalCategory,
  EndoFunctor,
  NaturalTransformation
}

import plp.implementation.generic.{composedFunctor, yonedaEndoFunctor}

import plp.implementation.specific.{functionCategory}
```

```
class PreFunctionalCategoryEndoYonedaLemma[
    Z,
    Arr[-_, +_]: PreFunctionalCategory,
    G[+_]: [G[+_]] =>> EndoFunctor[Arr, G]
]:
```

```
  val pfc = summon[PreFunctionalCategory[Arr]]

  import pfc.{YEF}

  def g[Z, Y]: Arr[Z, Y] => Arr[G[Z], G[Y]] = summon[EndoFunctor[Arr, G]].lift
```

```
  def endoYonedaLemma1[Y]: NaturalTransformation[Arr, Arr, YEF[Z], G] => Law[
    Arr[YEF[Z][Y], (YEF[U] O G)[Y]]
  ] =
    yefz2g =>
```

```
      import pfc.{`__-->__`, v2gv, eta}
```

```
      val `z-->z` = `__-->__`[Z]

      def tau[__]: Arr[YEF[Z][__], G[__]] = yefz2g.transform

      val `u-->g[z]` : (YEF[U] O G)[Z] =
        tau o v2gv(`z-->z`)

      val yefz_2_g: NaturalTransformation[Arr, Arr, YEF[Z], YEF[U] O G] =
        new:
          def transform[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] =
            pfc.f2a(`z--> __` => g(`z--> __`) o `u-->g[z]`)

      def sigma[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] =
        yefz_2_g.transform
```

```
      (eta[G[Y]] o tau) =:
        sigma
```

```
  def endoYonedaLemma2[Y, X]
      : (YEF[U] O G)[Z] => (Arr[Y, X] => Law[Arr[YEF[Z][Y], (YEF[U] O G)[X]]]) =
    `u-->g[z]` =>
      `y-->x` =>
```

```
        import pfc.{yef}
```

```
        val yefz2g: NaturalTransformation[Arr, Arr, YEF[Z], YEF[U] O G] =
          new:
            def transform[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] =
              pfc.f2a(`z--> __` => g(`z--> __`) o `u-->g[z]`)

        def tau[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] = yefz2g.transform
```

```
        ((yef o g)(`y-->x`) o tau) =:
          (tau o yef(`y-->x`))
```

3.4.2 `Scala` code

Below is a proof.

```
import plp.notation.{Proof, ==:, qed}
```

```
class PreFunctionalCategoryEndoYonedaProofs[
    Z,
    Arr[-_, +_]: PreFunctionalCategory,
    G[+_]: [G[+_]] =>> EndoFunctor[Arr, G]
]:
```

```
  val pfc = summon[PreFunctionalCategory[Arr]]

  import pfc.{YEF}
```

```
  def g[Z, Y]: Arr[Z, Y] => Arr[G[Z], G[Y]] = summon[EndoFunctor[Arr, G]].lift
```

```
def endoYonedaProof1[Y]: NaturalTransformation[Arr, Arr, YEF[Z], G] => (
    Arr[Z, Y] => Proof[(YEF[U] O YEF[U] O G)[Y]]
) =
  yefz2g =>
```

```
    import pfc.{`__-->__`, v2gv, yef, eta}
```

```
    val `z-->z` = `__-->__`[Z]

    def tau[__]: Arr[YEF[Z][__], G[__]] = yefz2g.transform

    val `u-->g[z]` : (YEF[U] O G)[Z] =
      tau o v2gv(`z-->z`)

    val yefz_2_g: NaturalTransformation[Arr, Arr, YEF[Z], YEF[U] O G] =
      new:
        def transform[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] =
          pfc.f2a(`z--> __` => g(`z--> __`) o `u-->g[z]`)

    def sigma[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] =
      yefz_2_g.transform

    def f2a[__] = pfc.f2a[Arr[Z, __], (YEF[U] O G)[__]]
```

```
    `z-->y` =>
      (eta[G[Y]] o tau o v2gv(`z-->y`)) ==:
        // categoryRightIdentityLaw for Arr
        (eta o tau o v2gv(`z-->y` o `z-->z`)) ==:
        // pointfreeYonedaProperty
        (eta o tau o yef(`z-->y`) o v2gv(`z-->z`)) ==:
        // naturalTransformationLaw for eta o tau
        ((yef o g)(`z-->y`) o eta o tau o v2gv(`z-->z`)) ==:
        // definition `u-->g[z]`
        ((yef o g)(`z-->y`) o eta o `u-->g[z]`) ==:
        // preFunctionalCategoryEtaLaw
        ((yef o g)(`z-->y`) o v2gv(`u-->g[z]`)) ==:
        // definition o for functionCategory
        ((yef(g(`z-->y`)) o v2gv(`u-->g[z]`))) ==:
        // pointfreeYonedaProperty
        (v2gv(g(`z-->y`) o `u-->g[z]`)) ==:
        // pointfreeApplicationProperty
        (f2a[Y](`z--> __` => g(`z--> __`) o `u-->g[z]`) o v2gv(`z-->y`)) ==:
        // definition sigma
        (sigma o v2gv(`z-->y`)) ==:
        // done
        qed
```

```
def endoYonedaProof2[Y, X]: (YEF[U] O G)[Z] => (
    Arr[Y, X] => (Arr[Z, Y] => Proof[(YEF[U] O YEF[U] O G)[X]])
) =
  `u-->g[z]` =>
    `y-->x` =>
```

```
      import pfc.{v2gv, yef}
```

```
val yefz2g: NaturalTransformation[Arr, Arr, YEF[Z], YEF[U] O G] =
  new:
    def transform[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] =
      pfc.f2a(`z--> __` => g(`z--> __`) o `u-->g[z]`)

def tau[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] = yefz2g.transform

def f2a[__] = pfc.f2a[Arr[Z, __], (YEF[U] O G)[__]]
```

```
`z-->y` =>
  ((yef o g)(`y-->x`) o tau o v2gv(`z-->y`)) ==:
    // definition tau
    ((yef o g)(`y-->x`) o f2a[Y](`z--> __` =>
      g(`z--> __`) o `u-->g[z]`
    ) o v2gv(`z-->y`)) ==:
    // pointfreeApplicationProperty
    ((yef o g)(`y-->x`) o v2gv(g(`z-->y`) o `u-->g[z]`)) ==:
    // definition o for functionCategory
    (yef(g(`y-->x`)) o v2gv(g(`z-->y`) o `u-->g[z]`)) ==:
    // pointfreeYonedaProperty
    v2gv(g(`y-->x`) o g(`z-->y`) o `u-->g[z]`) ==:
    // functorCompositionLaw for g
    v2gv(g(`y-->x` o `z-->y`) o `u-->g[z]`) ==:
    // pointfreeApplicationProperty
    (f2a[X](`z--> __` => g(`z--> __`) o `u-->g[z]`) o v2gv(
      `y-->x` o `z-->y`
    )) ==:
    // definition tau
    (tau o v2gv(`y-->x` o `z-->y`)) ==:
    // pointfreeYonedaProperty
    (tau o yef(`y-->x`) o v2gv(`z-->y`)) ==:
    // done
    qed
```

3.5 Pointfree Yoneda lemma for functional categories

3.6 Lemma

For all functional categories FC, nodes Z of FC and endofunctors G of FC, natural transformations $\tau : \boldsymbol{YEF}_Z \to \boldsymbol{YEF}_U \circ G$ correspond with arrows of $(\boldsymbol{YEF}_U \circ \boldsymbol{YEF}_U \circ G)(Z)$.

On the one hand, if $\tau : \boldsymbol{YEF}_Z \to \boldsymbol{YEF}_U \circ G$ is a natural transformation, and $u{\to}(u{\to}g(z)) \in (\boldsymbol{YEF}_U \circ \boldsymbol{YEF}_U \circ G)(Z)$ is defined as $\tau \circ \mathit{v2gv}(z{\to}z)$, then $\sigma : \boldsymbol{YEF}_Z \to \boldsymbol{YEF}_U \circ G$, defined as $\sigma = \mu G \circ \mathit{f2a}(z{\to}y \mapsto (\boldsymbol{y}_u \circ g)(z{\to}y) \circ u{\to}(u{\to}g(z)))$, is equal to τ.

As a corollary, if $\tau : \boldsymbol{YEF}_Z \to G$ is a natural transformation, and $u{\to}(u{\to}g(z)) \in (\boldsymbol{YEF}_U \circ \boldsymbol{YEF}_U \circ G)(Z)$ is defined as $\eta \circ \tau \circ \mathit{v2gv}(z{\to}z)$, then $\sigma : \boldsymbol{YEF}_Z \to \boldsymbol{YEF}_U \circ G$, defined as $\sigma = \mu G \circ \mathit{f2a}(z{\to}y \mapsto (\boldsymbol{y}_u \circ g)(z{\to}y) \circ u{\to}(u{\to}g(z)))$, is equal to $\eta \circ \tau$.

On the other hand, if $u \to (u \to g(z)) \in (\boldsymbol{YEF}_U \circ \boldsymbol{YEF}_U \circ G)(Z)$, and $\tau : \boldsymbol{YEF}_Z \to \boldsymbol{YEF}_U \circ G$ is defined as $\tau = \mu G \circ \mathit{f2a}(z{\to}y \mapsto (\boldsymbol{y}_u \circ g)(z{\to}y) \circ u{\to}(u{\to}g(z)))$, the τ is a natural transformation.

Both statements hold modulo composing with a global arrow $\mathit{v2gv}(z{\to}y)$.

3.6.1 Scala code

```
package plp.proposition

import plp.notation.{O, U, Law, =:}

import plp.specification.{
  FunctionalCategory,
  EndoFunctor,
  NaturalTransformation
}

import plp.implementation.generic.{composedFunctor, yonedaEndoFunctor}

import plp.implementation.specific.{functionCategory}
```

```
class FunctionalCategoryEndoYonedaLemma[
    Z,
    Arr[-_, +_]: FunctionalCategory,
    G[+_]: [G[+_]] =>> EndoFunctor[Arr, G]
]:
```

```
  val fc = summon[FunctionalCategory[Arr]]

  import fc.{YEF}
```

```
  def g[Z, Y]: Arr[Z, Y] => Arr[G[Z], G[Y]] = summon[EndoFunctor[Arr, G]].lift
```

```
  def endoYonedaLemma1[Y]
      : NaturalTransformation[Arr, Arr, YEF[Z], YEF[U] O G] => Law[
        Arr[YEF[Z][Y], (YEF[U] O G)[Y]]
      ] =
    yz2yu_o_g =>
```

```
      import fc.{`__-->__`, v2gv, yef, mu}
```

```
      val `z-->z` = `__-->__`[Z]

      def tau[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] = yz2yu_o_g.transform

      val `u-->(u-->g[z])` : (YEF[U] O YEF[U] O G)[Z] =
        tau o v2gv(`z-->z`)

      val yz_2_yu_o_g: NaturalTransformation[Arr, Arr, YEF[Z], YEF[U] O G] =
        new:
          def transform[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] =
            mu o fc.f2a(`z-->__` => (yef o g)(`z-->__`) o `u-->(u-->g[z])`)

      def sigma[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] = yz_2_yu_o_g.transform
```

```
    tau =:
      sigma
```

```
def endoYonedaLemma2[Y, X]: (YEF[U] O YEF[U] O G)[Z] => (
    Arr[Y, X] => Law[Arr[YEF[Z][Y], (YEF[U] O G)[X]]]
) =
  `u-->(u-->g[z])` =>
    `y-->x` =>
```

```
      import fc.{yef, mu}
```

```
      val yz2yu_o_g: NaturalTransformation[Arr, Arr, YEF[Z], YEF[U] O G] =
        new:
          def transform[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] =
            mu o fc.f2a(`z-->__` => (yef o g)(`z-->__`) o `u-->(u-->g[z])`)
```

```
      def tau[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] = yz2yu_o_g.transform
```

```
      ((yef o g)(`y-->x`) o tau) =:
        (tau o yef(`y-->x`))
```

```
def endoYonedaLemma1Corollary[Y]
    : NaturalTransformation[Arr, Arr, YEF[Z], G] => Law[
      Arr[YEF[Z][Y], (YEF[U] O G)[Y]]
    ] =
  yef2g =>
```

```
    import fc.{eta}
```

```
    def tau[__]: Arr[YEF[Z][__], G[__]] = yef2g.transform
```

```
    endoYonedaLemma1[Y](
      new {
        override def transform[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] =
          eta o tau
      }
    )
```

3.6.2 Scala code

Below is a proof.

```
import plp.notation.{Proof, ==:, qed}
```

```
class FunctionalCategoryEndoYonedaProof[
    Z,
    Arr[-_, +_]: FunctionalCategory,
    G[+_]: [G[+_]] =>> EndoFunctor[Arr, G]
]:
```

```
  val fc = summon[FunctionalCategory[Arr]]

  import fc.{YEF}
```

```
  def g[Z, Y]: Arr[Z, Y] => Arr[G[Z], G[Y]] = summon[EndoFunctor[Arr, G]].lift
```

```
  def endoYonedaProof1[Y]
      : NaturalTransformation[Arr, Arr, YEF[Z], YEF[U] O G] => (
          Arr[Z, Y] => Proof[(YEF[U] O YEF[U] O G)[Y]]
      ) =
    yz2yu_o_g =>
```

```
      import fc.{`__-->__`, v2gv, yef, eta, mu}
```

```
      val `z-->z` = `__-->__`[Z]

      def tau[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] = yz2yu_o_g.transform

      def `u-->(u-->g[z])` : (YEF[U] O YEF[U] O G)[Z] =
        tau o v2gv(`z-->z`)

      val yz_2_yu_o_g: NaturalTransformation[Arr, Arr, YEF[Z], YEF[U] O G] =
        new:
          def transform[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] =
            mu o fc.f2a(`z-->__` => (yef o g)(`z-->__`) o `u-->(u-->g[z])`)

      def sigma[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] = yz_2_yu_o_g.transform

      def f2a[__] = fc.f2a[Arr[Z, __], (YEF[U] O YEF[U] O G)[__]]
```

```
    `z-->y` =>
      (tau o v2gv(`z-->y`)) ==:
        // categoryRightIdentityLaw for Arr
        (tau o v2gv(`z-->y` o `z-->z`)) ==:
        // pointfreeYonedaProperty
        (tau o yef(`z-->y`) o v2gv(`z-->z`)) ==:
        // naturalTransformationLaw for tau
        ((yef o g)(`z-->y`) o tau o v2gv(`z-->z`)) ==:
        // definition `u-->(u-->g[z])`
        ((yef o g)(`z-->y`) o `u-->(u-->g[z])`) ==:
        // functionalCategoryMuProperty
        (mu o v2gv((yef o g)((`z-->y`)) o `u-->(u-->g[z])`)) ==:
        // pointfreeApplicationProperty
        (mu o f2a[Y](`z-->y` => (yef o g)(`z-->y`) o `u-->(u-->g[z])`) o v2gv(
          `z-->y`
        )) ==:
        // definition sigma
        (sigma o v2gv(`z-->y`)) ==:
        // done
        qed
```

```
def endoYonedaProof2[Y, X]: (YEF[U] O YEF[U] O G)[Z] => (
    Arr[Y, X] => (Arr[Z, Y] => Proof[(YEF[U] O YEF[U] O G)[X]])
) =
  `u-->(u-->g[z])` =>
    `y-->x` =>
```

```
      import fc.{v2gv, yef, mu}
```

```
      val yz2yu_o_g: NaturalTransformation[Arr, Arr, YEF[Z], YEF[U] O G] =
        new:
          def transform[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] =
          mu o fc.f2a(`z-->__` => (yef o g)(`z-->__`) o `u-->(u-->g[z])`)

      def tau[__]: Arr[YEF[Z][__], (YEF[U] O G)[__]] = yz2yu_o_g.transform

      def f2a[__] = fc.f2a[Arr[Z, __], (YEF[U] O YEF[U] O G)[__]]
```

```
    `z-->y` =>
      ((yef o g)(`y-->x`) o tau o v2gv(`z-->y`)) ==:
        // definition tau
        ((yef o g)(`y-->x`) o (mu o f2a[Y](`z-->y` =>
          (yef o g)(`z-->y`) o `u-->(u-->g[z])`
        )) o v2gv(`z-->y`)) ==:
        // pointfreeApplicationProperty
        ((yef o g)(`y-->x`) o (mu o v2gv(
          (yef o g)(`z-->y`) o `u-->(u-->g[z])`
        ))) ==:
        // mu property for (YEF[U], mu, eta)
        ((yef o g)(`y-->x`) o (yef o g)(`z-->y`) o `u-->(u-->g[z])`) ==:
        // functorCompositionProperty for yef o g
        ((yef o g)(`y-->x` o `z-->y`) o `u-->(u-->g[z])`) ==:
        // functionalCategoryMuProperty
        (mu o v2gv((yef o g)(`y-->x` o `z-->y`) o `u-->(u-->g[z])`)) ==:
        // pointfreeApplicationProperty
        (mu o f2a[X](`z-->x` =>
          (yef o g)(`z-->x`) o `u-->(u-->g[z])`
        ) o v2gv(`y-->x` o `z-->y`)) ==:
        // definition tau
        (tau o v2gv(`y-->x` o `z-->y`)) ==:
        // pointfreeYonedaProperty
        (tau o yef(`y-->x`) o v2gv(`z-->y`)) ==:
        qed
```

Index

References

[1] M. Barr and Ch. Wells, *Category theory for computing science*, 2nd ed. Prentice-Hall International Series in Computer Science, 1999.

[2] Mac Lane, Saunders, *Categories for the Working Mathematician*, vol-5, 2nd ed. Springer-Verlag, 1998.

[3] Benjamin C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[4] Simon Peyton Jones, *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[5] Martin Odersky, Lex Spoon, Bill Venners, and Frank Sommers, *Programming in Scala*, 5th ed. Artima, 2021.