A pointfree Yoneda lemma for endofunctors of functional categories

Luc Duponcheel,
mathematician and programmer, gardener and cyclist.

Abstract—Category theory [2] is used in almost all areas of mathematics, and in some areas of programming, more precisely pure effectful functional programming.

In mathematics, the Yoneda lemma suggests that instead of studying a category, one should study the category of all functors from that category to **Set**, the category whose objects are sets, and whose morphisms are functions. The lemma uses only three concepts: category, functor and natural transformation. The lemma is pointful, it deals with elements of sets.

In pure functional programming, one can write domain specific language libraries that model programming. Those libraries model programs as morphisms of categories whose objects are types. Those categories, we call them functional categories, have some specific properties. First, there exists a functor from **Set** to those categories that is the identity on objects. Using that functor effectfree functions can be used as programs. Of course there may also exist effectful programs. Second, programs themselves form a type.

This paper is a contribution to bridging the gap between mathematics and programming using the set-type, resp. function-program correspondence. Category theory, almost inevitably, plays a fundamental role in doing so [2].

Functors from a functional category to **Set** can be composed with the functor above from **Set** to that functional category to obtain an endofunctor of that functional category. It is natural to try to formulate and prove a Yoneda lemma for such endofunctors. 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 that functional category.

This paper is the result of advancing insights, obained by writing a library in `Scala` [5]. Any programming language with a sufficiently powerful type system, for example `Haskell` [4], could have been used instead. Most notably its type system has to support higher kinded type classes [3].

I. Introduction

A. Terminology and font conventions

We use computing science category theory terminology. For example,

- we use *node* resp. *arrow* for *object* resp. *morphism*,
- we use *source* resp. *target* for *domain* resp. *codomain*.

We use the following font conventions

- for category specifications we use math font, like C, and Z, Y, $\dots$, for nodes resp. $z{\longrightarrow}y\ \dots$, for arrows,
- for category implementations, simply called categories, we use boldface font, like **Set**, for the category of sets, and **Z**, **Y**, … , for sets resp. **z⇒y** … , for functions.

B. Notational assumptions and conventions

We use notational assumptions for arrows of C.

- we assume that $z \longrightarrow y\ \dots$, are names of arrows with source node Z and target node Y,
- we assume that $z{\longrightarrow}z\ \dots$, are names of identity arrows with source node Z and target node Z.

We use similar notational assumptions for functions of **Set**.

Moreover,

- we assume that **z**, resp. **y** … , are names of elements of sets **Z**, resp. **Y** … .

We use notational conventions.

- letters A and a, resp. F and f, resp. C and c, resp. Y and y refer to ”arrow”, resp. ”function”, resp. ”constant”, resp. ”Yoneda”.

Function literals $\mathbf{z} \mapsto \mathbf{e}(\mathbf{z})$, where $\mathbf{e}(\mathbf{z})$ is an expression containing **z** once, are, just like is often done when using programmatic notation, simply denoted $\mathbf{e}(\_)$.

II. Preliminaries

A. Definition

The *graph specification* G declares

$G_{dec}{-}0$ G_0, *nodes*, Z, Y, $\dots$,
$G_{dec}{-}1$ G_1, *arrows*, $z{\longrightarrow}y$, $\dots$,
$G_{dec}{-}2$ G_2, *composable arrows* $z{\longrightarrow}y$ and $y{\longrightarrow}x$.

Nodes do not need to form a set. $Arr_G(Z,Y)$, arrows with source node Z and target node Y, form a set.

Luc Duponcheel is a retired mathematician and programmer. Nowadays, besides building a bridge between mathematics and programming, he raises vegetables in his greenhouse and cycles with his wife all over Europe.

1) Notation: $Arr_G(Z,Y)$ is also simply denoted $Arr(Z,Y)$.

2) Assumptions: When using names like $z{\rightarrow}y$ we assume that $z{\rightarrow}y \in Arr(Z,Y)$.

B. Definition

An *endoarrow* is an arrow in $Arr(Z,Z)$.

C. Definition

The **Set** graph defines

$\mathbf{Set}_{def}-0$ $\mathbf{Set}_0$, *sets*, $\mathbf{Z}$, $\mathbf{Y}$, $\dots$,
$\mathbf{Set}_{def}-1$ $\mathbf{Set}_1$, *functions*, $\mathbf{z{\Rightarrow}y}$, $\dots$.

Sets do not form a set. $\mathbf{Fun}(\mathbf{Z},\mathbf{Y})$, functions with source set $\mathbf{Z}$ and target set $\mathbf{Y}$ form a set.

1) Assumptions: When using names like $\mathbf{z{\Rightarrow}y}$ we assume that $\mathbf{z{\Rightarrow}y} \in \mathbf{Fun}(\mathbf{Z},\mathbf{Y})$.

D. Definition

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

$M_{dec}-0$ $M_0 : G_0 \to H_0$,
$M_{dec}-1$ $M_1 : G_1 \to H_1$.

M_0 does not need to be a function.

1) Notation: Nodes $M_0(Z)$ are denoted $m(Z)$. Arrows $M_1(z{\rightarrow}y)$ are denoted $m(z{\rightarrow}y)$.

2) Laws:

$M_{law}-0$ if $z{\rightarrow}y \in Arr_G(Z,Y)$,
then $m(z{\rightarrow}y) \in Arr_H(m(Z),m(Y))$.

All $m : Arr_G(Z,Y) \to Arr_H(m(Z),m(Y))$ are functions.

3) Notation: Arrows $m(z{\rightarrow}y)$ are also denoted $m(z){\rightarrow}m(y)$.

E. Definition

The *category specification* C extends the graph specification. It delares

$C_{dec}-0$ $c : C_2 \to C_1$,
$C_{dec}-1$ $u : C_0 \to C_1$.

c is called *composition*. u is called *unit*.

1) Notation: If $z{\rightarrow}y$ and $y{\rightarrow}x$ are composable arrows, then the arrow $c(z{\rightarrow}y, y{\rightarrow}x)$ is denoted $y{\rightarrow}x \circ_c z{\rightarrow}y$, or, simply, $y{\rightarrow}x \circ z{\rightarrow}y$ and is called the *composite arrow* of $z{\rightarrow}y$ and $y{\rightarrow}x$, or, simply, the *composite* of $z{\rightarrow}y$ and $y{\rightarrow}x$. If Z is a node, then the arrow $u(Z)$ is denoted $z \rightarrow_c z$, or, simply, $z{\rightarrow}z$ and is called the *identity arrow* of Z, or, simply, the *identity* of Z.

2) Assumptions: When using names like $z \rightarrow_c z$ we assume that $z \rightarrow_c z$ is the identity of Z.

3) Laws:

$C_{law}-0$ The source of $y{\rightarrow}x \circ_c z{\rightarrow}y$ is the source of $z{\rightarrow}y$,
$C_{law}-1$ The target of $y{\rightarrow}x \circ_c z{\rightarrow}y$ is the target of $y{\rightarrow}x$,
$C_{law}-2$ The source resp. target of $y{\rightarrow}y$ is Y,
$C_{law}-3$ The source resp. target of $z{\rightarrow}z$ is Z,
$C_{law}-4$ $(x{\rightarrow}w \circ_c y{\rightarrow}x) \circ_c z{\rightarrow}y = x{\rightarrow}w \circ_c (y{\rightarrow}x) \circ_c z{\rightarrow}y)$
(*associativity law*),
$C_{law}-5$ $y{\rightarrow}y \circ_c z{\rightarrow}y = z{\rightarrow}y$ (*left identity law*),
$C_{law}-6$ $z{\rightarrow}y \circ_c z{\rightarrow}z = z{\rightarrow}y$ (*right identity law*).

F. Definition

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

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

G. Definition

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

H. Definition

Given

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

the *composed functor* $G \circ F : C \to E$ is defined as follows:

$(G \circ F)_{def}-0$ $(g \circ f)(Z) = g(f(Z))$,
$(G \circ F)_{def}-1$ $(g \circ f)(z{\rightarrow}y) = g(f(z{\rightarrow}y))$.

I. Definition

The *identity functor* $\mathbf{I} : C \to C$ of category C is defined as follows

$\mathbf{I}_{def}-0$ $\mathbf{i}(Z) = Z$,
$\mathbf{I}_{def}-1$ $\mathbf{i}(z{\rightarrow}y) = z{\rightarrow}y$.

J. Definition

The *Yoneda functor* $\mathbf{YF}_Z : C \to \mathbf{Set}$ for node Z of category C is defined as follows

$\mathbf{YF}_Z-0$ $\mathbf{yf}_z(Y) = Arr(Z,Y)$,
$\mathbf{YF}_Z-1$ $\mathbf{yf}_z(y{\rightarrow}x) = y{\rightarrow}x \circ \_$.

Note that $\mathbf{yf}_z(y{\rightarrow}x)(z{\rightarrow}y) = y{\rightarrow}x \circ z{\rightarrow}y$.

K. Definition

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

$\alpha_{dec}-0$ arrows $\alpha_z \in Arr_D(f(Z),g(Z))$.

1) Laws:

$\alpha_{law}-0$ $\alpha_y \circ_d f(z{\rightarrow}y) = g(z{\rightarrow}y) \circ_d \alpha_z$ (*commutativity law*).

2) Notation: α_z is also, simply, denoted α.

L. Definition

Given

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

the *composed natural transformation* $\beta \circ \alpha : C \rightarrow C$ is defined as follows:

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

M. Definition

Given

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

natural transformation $\beta F : H \circ F \rightarrow K \circ F$ is defined as follows:

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

natural transformation $H\alpha : H \circ F \rightarrow H \circ G$ is defined as follows:

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

N. Definition

The *pre-triple specification* $\pi\tau = (PT, \eta)$ for category C declares

$\pi\tau_{dec}-0$ an endofunctor $PT : C \rightarrow C$,
$\pi\tau_{dec}-1$ a natural transformation $\eta : \mathbf{I} \rightarrow PT$.

η is called *unit*.

O. Definition

The *triple specification* $\tau = (T, \mu, \eta)$ for category C declares

$\tau_{dec}-0$ a pre-triple (T, η)
$\tau_{dec}-1$ a natural transformation $\mu : T \circ T \rightarrow T$.

μ is called *multiplication*.

Note that $\mathbf{I} = T^0$, $T = T^1$ and $T \circ T = T^2$.

1) Laws:

$\tau_{law}-0$ $\mu_z \circ_c (\eta T)_z = t(z){\rightarrow}t(z)$ (*left identity law*),
$\tau_{law}-1$ $\mu_z \circ_c (T\eta)_z = t(z){\rightarrow}t(z)$ (*right identity law*),
$\tau_{law}-2$ $\mu_z \circ_c (T\mu)_z = \mu_z \circ_c (\mu T)_z$ (*multiplication law*).

III. Content

A. Definition

The *pre-functional category specification PFC*, whose nodes are the nodes of **Set**, extends the category specification.

1) Notation: Although nodes of *PFC* are sets, we will, by abuse of notation, denote them as nodes, $Z, Y, \ldots$ (and the same for their elements, z, y, $\ldots$).

It declares

$PFC_{dec}-0$ a functor **F2A** from **Set** to *PFC* that is the identity on nodes.

We will use the following definitions

$PFC_{def}-0$ $\mathbf{Y}_Z$, the *Yoneda endofunctor for* Z, is $\mathbf{F2A} \circ \mathbf{YF}_Z$,
$PFC_{def}-1$ $\mathbf{caf} \in \mathbf{Fun}(Y, Arr(Z, Y))$ is $y \mapsto \mathbf{f2a}(z \mapsto y)$,
$PFC_{def}-2$ $\eta_z \in Arr(\mathbf{I}(Z), \mathbf{Y}_U(Z))$ is $\mathbf{f2a}(\mathbf{caf})$, where U is a terminal node (singleton set) of the category **Set**.

Note that **f2a** stands for function to arrow, and **caf** stands for constant arrow function.

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

2) Notation: **f2a**, mapping functions of $\mathbf{Fun}(Z, Y)$ to arrows of $Arr(Z, Y)$ for node Z, is, when, in our opinion, instructive, explicitly denoted $\mathbf{f2a}_y$.

3) Laws:

$PFC_{law}-0$ $\eta : \mathbf{I} \rightarrow \mathbf{Y}_U$ is a *natural* transformation, in other words, $(\mathbf{Y}_U, \eta)$ is a pre-triple.
$PFC_{law}-1$ $\mathbf{caf}(u{\rightarrow}z) = \eta \circ u{\rightarrow}z$ (η *law*)

4) Property: The following property can easily be proved

app-0 $\mathbf{f2a}(z{\Rightarrow}y) \circ \mathbf{caf}(z) = \mathbf{caf}(z{\Rightarrow}y(z))$
(*pointfree application*).

5) Property: The following property can easily be proved

$\mathbf{Y}-0$ $\mathbf{y}_z(y{\rightarrow}x) \circ \mathbf{caf}(z{\rightarrow}y) = \mathbf{caf}(y{\rightarrow}x \circ z{\rightarrow}y)$
(*pointfree Yoneda*).

B. Pointfree Yoneda lemma for pre-functional categories

We are ready now for our pointfree Yoneda lemma for endofunctors of pre-functional categories.

C. Lemma

For all pre-functional categories *PFC*, nodes Z of *PFC*, and endofunctors G of *PFC*, natural transformations $\eta G \circ \beta : \mathbf{Y}_Z \rightarrow \mathbf{Y}_U \circ G$, for natural tranformations $\beta : \mathbf{Y}_Z \rightarrow G$, correspond with arrows of $(\mathbf{y}_u \circ g)(Z)$.

On the one hand, if $\beta : \mathbf{Y}_Z \rightarrow G$ is a natural transformation, and

- $u{\rightarrow}g(z) \in (\mathbf{y}_u \circ g)(Z)$ is defined as $\beta \circ \mathbf{caf}(z{\rightarrow}z)$, then, $\gamma : \mathbf{Y}_Z \to \mathbf{Y}_U \circ G$, defined as $\gamma = \mathbf{f2a}(g(\_) \circ u \rightarrow g(z))$, is equal to $\eta G \circ \beta$.

On the other hand, if $u \rightarrow g(z) \in (\mathbf{y}_u \circ g)(Z)$, and

- $\gamma : \mathbf{Y}_Z \to \mathbf{Y}_U \circ G$ is defined as $\gamma = \mathbf{f2a}(g(\_) \circ u \rightarrow g(z))$,

then γ, is a natural transformation.

Both statements hold modulo composing with an arrow $\mathbf{caf}(z \rightarrow y)$.

D. Definition

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

$FC_{dec}-0$ natural transformation $\mu : (\mathbf{Y}_U \circ \mathbf{Y}_U) \to \mathbf{Y}_U$.

1) Laws:

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

2) Property: The following property can easily be proved

$\mu_{prop}-0$ $\mu \circ \mathbf{caf}(u \rightarrow (u \rightarrow z)) = u \rightarrow (u \rightarrow z)$ (μ *property*)

E. Pointfree Yoneda lemma for functional categories

We are ready now for our pointfree Yoneda lemma for endofunctors of functional categories.

F. Lemma

For all functional categories FC, nodes Z of FC, and endofunctors G of FC, natural transformations $\beta : \mathbf{Y}_Z \to \mathbf{Y}_U \circ G$ correspond with arrows of $(\mathbf{y}_u \circ \mathbf{y}_u \circ g)(Z)$.

On the one hand, if $\beta : \mathbf{Y}_Z \to \mathbf{Y}_U \circ G$ is a natural transformation, and

- $u \rightarrow (u \rightarrow g(z)) \in (\mathbf{y}_u \circ \mathbf{y}_u \circ g)(Z)$ is defined as $\beta \circ \mathbf{caf}(z \rightarrow z)$,

then, $\gamma : \mathbf{Y}_Z \to \mathbf{Y}_U \circ G$, defined as
$\gamma = \mu G \circ \mathbf{f2a}((\mathbf{y}_u \circ g)(\_) \circ u \rightarrow (u \rightarrow g(z)))$, is equal to β.

On the other hand, if $u \rightarrow (u \rightarrow g(z)) \in (\mathbf{y}_u \circ \mathbf{y}_u \circ g)(Z)$, and

- $\gamma : \mathbf{Y}_Z \to \mathbf{Y}_U \circ G$ is defined as
  $\gamma = \mu G \circ \mathbf{f2a}((\mathbf{y}_u \circ g)(\_) \circ u \rightarrow (u \rightarrow g(z)))$,

then γ, is a natural transformation.

Both statements hold modulo composing with an arrow $\mathbf{caf}(z \rightarrow y)$.

Appendix A
Proof of first lemma

A. Proof

On the one hand, if $z \rightarrow y \in Arr(Z, Y)$, then

$$\begin{aligned}
& \eta G \circ \beta \circ \mathbf{caf}(z \rightarrow y) \\
& \qquad \text{right identity law for } Arr \\
= \; & \eta G \circ \beta \circ \mathbf{caf}(z \rightarrow y \circ z \rightarrow z) \\
& \qquad \text{pointfree Yoneda law for } (\mathbf{Y}_U, \eta) \\
= \; & \eta G \circ \beta \circ \mathbf{y}_z(z \rightarrow y) \circ \mathbf{caf}(z \rightarrow z) \\
& \qquad \text{natural transformation law for } \eta G \circ \beta \\
= \; & (\mathbf{y}_u \circ g)(z \rightarrow y) \circ \eta G \circ \beta \circ \mathbf{caf}(z \rightarrow z) \\
& \qquad \text{definition } u \rightarrow g(z) \\
= \; & (\mathbf{y}_u \circ g)(z \rightarrow y) \circ \eta G \circ u \rightarrow g(z) \\
& \qquad \eta \text{ law for } (\mathbf{Y}_U, \eta) \\
= \; & (\mathbf{y}_u \circ g)(z \rightarrow y) \circ \mathbf{caf}(u \rightarrow g(z)) \\
& \qquad \text{function composition for } \mathbf{y}_u \text{ and } g \\
= \; & \mathbf{y}_u(g(z \rightarrow y)) \circ \mathbf{caf}(u \rightarrow g(z)) \\
& \qquad \text{pointfree Yoneda law for } (\mathbf{Y}_U, \eta) \\
= \; & \mathbf{caf}(g(z \rightarrow y) \circ u \rightarrow g(z)) \\
& \qquad \text{pointfree application law for } (\mathbf{Y}_U, \eta) \\
= \; & \mathbf{f2a}_y(g(\_) \circ u \rightarrow g(z)) \circ \mathbf{caf}(z \rightarrow y) \\
& \qquad \text{definition } \gamma \\
= \; & \gamma \circ \mathbf{caf}(z \rightarrow y)
\end{aligned}$$

On the other hand, if $z \rightarrow y \in Arr(Z, Y)$, then

$$\begin{aligned}
& (\mathbf{y}_u \circ g)(y \rightarrow x) \circ \gamma \circ \mathbf{caf}(z \rightarrow y) \\
& \qquad \text{definition } \gamma \\
= \; & (\mathbf{y}_u \circ g)(y \rightarrow x) \circ \mathbf{f2a}_y(g(\_) \circ u \rightarrow g(z)) \circ \mathbf{caf}(z \rightarrow y) \\
& \qquad \text{pointfree application law for } (\mathbf{Y}_U, \eta) \\
= \; & (\mathbf{y}_u \circ g)(y \rightarrow x) \circ \mathbf{caf}(g(z \rightarrow y) \circ u \rightarrow g(z)) \\
& \qquad \text{function composition for } \mathbf{y}_u \text{ and } g \\
= \; & \mathbf{y}_u(g(y \rightarrow x)) \circ \mathbf{caf}(g(z \rightarrow y) \circ u \rightarrow g(z)) \\
& \qquad \text{pointfree Yoneda law for } (\mathbf{Y}_U, \eta) \\
= \; & \mathbf{caf}(g(y \rightarrow x) \circ g(z \rightarrow y) \circ u \rightarrow g(z)) \\
& \qquad \text{composition law for } g \\
= \; & \mathbf{caf}(g(y \rightarrow x \circ z \rightarrow y) \circ u \rightarrow g(z)) \\
& \qquad \text{pointfree application law for } (\mathbf{Y}_U, \eta) \\
= \; & \mathbf{f2a}_x(g(\_) \circ u \rightarrow g(z)) \circ \mathbf{caf}(y \rightarrow x \circ z \rightarrow y) \\
& \qquad \text{pointfree Yoneda law for } (\mathbf{Y}_U, \eta) \\
= \; & \mathbf{f2a}_x(g(\_) \circ u \rightarrow g(z)) \circ \mathbf{y}_z(y \rightarrow x) \circ ca(z \rightarrow y) \\
& \qquad \text{definition } \gamma \\
= \; & \gamma \circ \mathbf{y}_z(y \rightarrow x) \circ \mathbf{caf}(z \rightarrow y)
\end{aligned}$$

Appendix B
Proof of Second Lemma

A. Proof

On the one hand, if $z{\to}y \in Arr(Z, Y)$, then

$\beta \circ \mathbf{caf}(z{\to}y)$

right identity law for Arr

$= \beta \circ \mathbf{caf}(z{\to}y \circ z{\to}z)$

pointfree Yoneda law for $(\mathbf{Y}_U, \eta)$

$= \beta \circ \mathbf{y}_z(z{\to}y) \circ \mathbf{caf}(z{\to}z)$

natural transformation law for β

$= (\mathbf{y}_u \circ g)(z{\to}y) \circ \beta \circ \mathbf{caf}(z{\to}z)$

definition $u{\to}(u{\to}g(z))$

$= (\mathbf{y}_u \circ g)(z{\to}y) \circ u{\to}(u{\to}g(z))$

μ property for $(\mathbf{Y}_U, \mu, \eta)$

$= \mu G \circ \mathbf{caf}((\mathbf{y}_u \circ g)(z{\to}y) \circ u{\to}(u{\to}g(z)))$

pointfree application law for $(\mathbf{Y}_U, \eta)$

$= \mu G \circ \mathbf{f2a}_y((\mathbf{y}_u \circ g)(\_) \circ u{\to}(u{\to}g(z))) \circ ca(z{\to}y)$

definition γ

$= \gamma \circ \mathbf{caf}(z{\to}y)$

On the other hand, if $z{\to}y \in Arr(Z, Y)$, then

$(\mathbf{y}_u \circ g)(y{\to}x) \circ \gamma \circ \mathbf{caf}(z{\to}y)$

definition γ

$= (\mathbf{y}_u \circ g)(y{\to}x) \circ \mu G \circ \mathbf{f2a}_y((\mathbf{y}_u \circ g)(\_) \circ u{\to}(u{\to}g(z))) \circ \mathbf{caf}(z{\to}y)$

pointfree application law for $(\mathbf{Y}_U, \eta)$

$= (\mathbf{y}_u \circ g)(y{\to}x) \circ \mu G \circ \mathbf{caf}((\mathbf{y}_u \circ g)(z{\to}y) \circ u{\to}(u{\to}g(z)))$

μ property for $(\mathbf{Y}_U, \mu, \eta)$

$= (\mathbf{y}_u \circ g)(y{\to}x) \circ (\mathbf{y}_u \circ g)(z{\to}y) \circ u{\to}(u{\to}g(z))$

composition law for $\mathbf{y}_u \circ g$

$= (\mathbf{y}_u \circ g)(y{\to}x \circ z{\to}y) \circ u{\to}(u{\to}g(z))$

μ property for $(\mathbf{Y}_U, \mu, \eta)$

$= \mu G \circ \mathbf{caf}((\mathbf{y}_u \circ g)(y{\to}x \circ z{\to}y) \circ u{\to}(u{\to}g(z)))$

pointfree application law for $(\mathbf{Y}_U, \eta)$

$= \mu G \circ \mathbf{f2a}_x((\mathbf{y}_u \circ g)(\_) \circ u{\to}(u{\to}g(z))) \circ \mathbf{caf}(y{\to}x \circ z{\to}y)$

definition γ

$= \gamma \circ \mathbf{caf}(y{\to}x \circ z{\to}y)$

pointfree Yoneda law for $(\mathbf{Y}_U, \eta)$

$= \gamma \circ \mathbf{y}_z(y{\to}x) \circ \mathbf{caf}(z{\to}y)$

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.

Luc Duponcheel has worked as a mathematics researcher at various universities of Belgium and The Netherlands. Later on he worked as an international Java programming instructor for Sun Microsystems. At the end of his career he worked as an independent Scala software consultant. All his life his end-of-day hobby was to build a bridge between mathematics and programming.