A Pointfree Yoneda Lemma for Endofunctors of Functional Categories

ICMSP 2023
Madrid

Luc Duponcheel

Previous life

Previous life

- Mathematician

Previous life

- Mathematician
- Programmer

Previous life

- Mathematician
- Programmer
    - Monad and monad transformer contributor 30 years ago with Erik Meijer, Graham Hutton and Doaitse Swierstra at University of Utrecht

Current life

Current life

- Cyclist

Current life

- Cyclist
- Gardener

Current life

- Cyclist
- Gardener

- Mathematician

Current life

- Cyclist
- Gardener

- Mathematician
- Programmer

Main themes

Main themes

- Mathematics $\to$ Programming

Main themes

- Mathematics $\rightarrow$ Programming
- Programming $\rightarrow$ Mathematics

Mathematics $\rightarrow$ Programming

Mathematics $\rightarrow$ Programming

- Category theory

Mathematics $\to$ Programming

- Category theory
    - Composition

Mathematics $\rightarrow$ Programming

- Category theory
    - Composition
    - Additional features

Mathematics $\rightarrow$ Programming

- Category theory
    - Composition
    - Additional features
        - Transformation ...

Mathematics → Programming

- Category theory
  - Composition
  - Additional features
    - Transformation . . .

  - Separation

Mathematics → Programming

- Category theory
    - Composition
    - Additional features
        - Transformation . . .
    - Separation
        - specifications

Mathematics $\rightarrow$ Programming

- Category theory
  - Composition
  - Additional features
    - Transformation ...
  - Separation
    - specifications
    - implementations

Main sub-themes

Main sub-themes

- Composition : components

Main sub-themes

- Composition : components
    - Closed (pointfree) components

Main sub-themes

- Composition : components
  - Closed (pointfree) components
  - Open (pointful) components

Main sub-themes

- Composition : components
  - Closed (pointfree) components
  - Open (pointful) components

- Additional features and separation : side-effects

Main sub-themes

- Composition : components
  - Closed (pointfree) components
  - Open (pointful) components

- Additional features and separation : side-effects
  - Specified (declared) side-effects

Main sub-themes

- Composition : components
    - Closed (pointfree) components
    - Open (pointful) components

- Additional features and separation : side-effects
    - Specified (declared) side-effects
    - Implemented (defined) side-effects

Pointful Effectfree Defined

Pointful Effectfree Defined

- Expressions (pointful effectfree components)

Pointful Effectfree Defined

- Expressions (pointful effectfree components)
- Operational (are evaluated to yield a result)

Pointful Effectfree Defined

- Expressions (pointful effectfree components)
- Operational (are evaluated to yield a result)

  -
    ```
    val zero = 0
    val one = zero + 1
    val two = one + 1
    two
    ```

Pointfree Effectfree Defined

Pointfree Effectfree Defined

- Functions (pointfree effectfree components)

Pointfree Effectfree Defined

- Functions (pointfree effectfree components)
- Denotational (are meaningful)

Pointfree Effectfree Defined

- Functions (pointfree effectfree components)
- Denotational (are meaningful)

  -
    ```
    def incrementF: Function[Int, Int] =
      i => i + 1

    def zeroF: Function[Unit, Int] =
      i => 0
    ```

Pointfree Effectfree Defined

- Functions (pointfree effectfree components)
- Denotational (are meaningful)

  -
    ```
    def incrementF: Function[Int, Int] =
      i => i + 1

    def zeroF: Function[Unit, Int] =
      i => 0
    ```

  - `incrementF o incrementF o zeroF`

Pointful Effectful Defined

Pointful Effectful Defined

- `val readInt: Int = ...`

Pointful Effectful Defined

- `val readInt: Int = ...`

    -
      ```
      val zero = readInt
      val one = zero + 1
      val two = one + 1
      two
      ```

Pointfree Effectful Defined

Pointfree Effectful Defined

- `val readIntF: Function[Unit, Int] = ...`

Pointfree Effectful Defined

- `val readIntF: Function[Unit, Int] = ...`
  - `incrementF o incrementF o readIntF`

Pointful Effectful Declared

Pointful Effectful Declared

- Computations (generalize expressions)

Pointful Effectful Declared

- Computations (generalize expressions)
- Operational (are executed to yield a result)

Pointful Effectful Declared

- Computations (generalize expressions)
- Operational (are executed to yield a result)

- `val readIntC: C[Int]`

Pointful Effectful Declared

- Computations (generalize expressions)
- Operational (are executed to yield a result)

- `val readIntC: C[Int]`

  -
    ```
    readIntC bind { i =>
      result(i + 1) bind { j =>
        result(j + 1) bind { k =>
          result(k)
        }
      }
    }
    ```

Pointfree Effectful Declared

Pointfree Effectful Declared

- Programs (generalize functions)

Pointfree Effectful Declared

- Programs (generalize functions)
    - Functions can be used as programs

Pointfree Effectful Declared

- Programs (generalize functions)
  - Functions can be used as programs
- Denotational (are meaningful)

Pointfree Effectful Declared

- Programs (generalize functions)
  - Functions can be used as programs
- Denotational (are meaningful)

-
  ```
  val readIntP: Program[Unit, Int]

  val incrementP: Int --> Int = incrementF asProgram
  ```

Pointfree Effectful Declared

- Programs (generalize functions)
    - Functions can be used as programs
- Denotational (are meaningful)

-
  ```
  val readIntP: Program[Unit, Int]

  val incrementP: Int --> Int = incrementF asProgram
  ```
    - `incrementP o incrementP o readIntP`

Programming $\rightarrow$ Mathematics

Programming $\rightarrow$ Mathematics

- Type theory (as implemented by type systems)

Programming $\rightarrow$ Mathematics

- Type theory (as implemented by type systems)
  - Proofs syntactic correctness

Programming $\rightarrow$ Mathematics

- Type theory (as implemented by type systems)
  - Proofs syntactic correctness
  - Provides confidence in semantic correctness

Programming $\rightarrow$ Mathematics

- Type theory (as implemented by type systems)
    - Proofs syntactic correctness
    - Provides confidence in semantic correctness
        - Formulations of laws for specifications and proofs of for implementations

Programming $\rightarrow$ Mathematics

- Type theory (as implemented by type systems)
  - Proofs syntactic correctness
  - Provides confidence in semantic correctness
    - Formulations of laws for specifications and proofs of for implementations
    - Formulations and proofs of lemmas, propositions, theorems ... for specifications

Advancing insights

Advancing insights

- Common patterns leading to auxiliary lemmas, propositions, theorems . . .

Advancing insights

- Common patterns leading to auxiliary lemmas, propositions, theorems . . .
    - Genericity (Do not repeat yourself)

Advancing insights

- Common patterns leading to auxiliary lemmas, propositions, theorems . . .
    - Genericity (Do not repeat yourself)
- Recognition of known concepts

Advancing insights

- Common patterns leading to auxiliary lemmas, propositions, theorems ...
  - Genericity (Do not repeat yourself)
- Recognition of known concepts
  - Reusability (Embrace what exists)

Advancing insights

- Common patterns leading to auxiliary lemmas, propositions, theorems . . .
  - Genericity (Do not repeat yourself)
- Recognition of known concepts
  - Reusability (Embrace what exists)
- My experience

Advancing insights

- Common patterns leading to auxiliary lemmas, propositions, theorems . . .
  - Genericity (Do not repeat yourself)
- Recognition of known concepts
  - Reusability (Embrace what exists)
- My experience
  - 25 percent Imagination (idea)

Advancing insights

- Common patterns leading to auxiliary lemmas, propositions, theorems . . .
  - Genericity (Do not repeat yourself)
- Recognition of known concepts
  - Reusability (Embrace what exists)
- My experience
  - 25 percent Imagination (idea)
  - 25 percent Information (knowledge)

Advancing insights

- Common patterns leading to auxiliary lemmas, propositions, theorems . . .
  - Genericity (Do not repeat yourself)
- Recognition of known concepts
  - Reusability (Embrace what exists)
- My experience
  - 25 percent Imagination (idea)
  - 25 percent Information (knowledge)
  - 50 percent Transpiration (the heavy lifting)

Advancing insights

- Common patterns leading to auxiliary lemmas, propositions, theorems . . .
  - Genericity (Do not repeat yourself)
- Recognition of known concepts
  - Reusability (Embrace what exists)
- My experience
  - 25 percent Imagination (idea)
  - 25 percent Information (knowledge)
  - 50 percent Transpiration (the heavy lifting)

- Never be happy with a solution, try to go for the best one