PDBP

Program Description Based Programming

Luc Duponcheel

Intro

Intro

- this talk is about

Intro

- this talk is about
    - a *library*, `PDBP`

Intro

- this talk is about
    - a *library*, `PDBP`
    - the *language* `FP`

John Backus

John Backus

- ACM Turing Award Winner 1977

John Backus

- ACM Turing Award Winner 1977
- *Can programming be liberated from the Von Neumann style?*

What is this?

What is this?

- A *pipe*?

What is this?

- A *pipe*?
- A *painting describing* a pipe?

What is this?

- A *pipe*?
- A *painting describing* a pipe?
- A *slide describing* a painting describing a pipe?

What is this?

- A *pipe*?
- A *painting describing* a pipe?
- A *slide describing* a painting describing a pipe?
- ...

Ceci n'est pas une pipe.

This is not a program.

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

This is not a program.

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

- It is *code describing* a program.

This is not a program.

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

- It is *code describing* a program.
- It can be given different *meanings*.

description versus *meaning*

```
scala> var z = 3
z: Int = 3

scala> while(z > 0) { println(z) ; z = z - 1 }
3
2
1
```

description versus *meaning*

```
scala> trait WhileDescription {
     |   def `while`(b: => Boolean)(u: => Unit): Unit
     | }
defined trait WhileDescription
```

description versus *meaning*

```
scala> object whileMeaning1
     |   extends WhileDescription {
     |   override
     |     def `while`(b: => Boolean)(u: => Unit): Unit =
     |       while(b)(u)
     | }
defined object whileMeaning1
```

description versus *meaning*

```
scala> import whileMeaning1.`while`
import whileMeaning1.`while`

scala> var z = 3
z: Int = 3

scala> `while`(z > 0) { println(z) ; z = z - 1 }
3
2
1
```

description versus *meaning*

```
scala> object whileMeaning2
     |   extends WhileDescription {
     |   override
     |     def `while`(b: => Boolean)(u: => Unit): Unit =
     |       if(b) { u ; `while`(b)(u) } else { }
     | }
defined object whileMeaning2
```

description versus *meaning*

```
scala> import whileMeaning2.‘while‘
import whileMeaning2.‘while‘

scala> var z = 3
z: Int = 3

scala> ‘while‘(z > 0) { println(z) ; z = z - 1 }
3
2
1
```

description versus *meaning*

```
scala> object whileMeaning3
     |   extends WhileDescription {
     |   override
     |     def `while`(b: => Boolean)(u: => Unit): Unit =
     |       if(b) { u ; `while`(b)(u) } else { println("!") }
     | }
defined object whileMeaning3
```

description versus *meaning*

```
scala> import whileMeaning3.`while`
import whileMeaning3.`while`

scala> var z = 3
z: Int = 3

scala> `while`(z > 0) { println(z) ; z = z - 1 }
3
2
1
!
```

FP versus PDBP

`FP` versus `PDBP`

- `FP` and `PDBP` promote *pointfree functional programming*

FP versus PDBP

- FP and PDBP promote *pointfree functional programming*
- FP is a *language*
  PDBP is a *library*

FP versus PDBP

- FP and PDBP promote *pointfree functional programming*
- FP is a *language*
  PDBP is a *library*
    - FP *semantics* is *fixed*
      PDBP *semantics* is *not fixed*

FP versus PDBP

- FP and PDBP promote *pointfree functional programming*
- FP is a *language*
  PDBP is a *library*
  - FP *semantics* is *fixed*
    PDBP *semantics* is *not fixed*
  - FP *capabilities* are *fixed*
    PDBP *capabilities* are *not fixed*

FP versus PDBP

- FP and PDBP promote *pointfree functional programming*
- FP is a *language*
  PDBP is a *library*
  - FP *semantics* is *fixed*
    PDBP *semantics* is *not fixed*
  - FP *capabilities* are *fixed*
    PDBP *capabilities* are *not fixed*
  - FP *effects* are *impure*
    PDBP *effects* are *pure*

Semantics

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

Semantics

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

- *main*

Semantics

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

- *main*
  - *recursion* using *stack*

Semantics

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

- *main*
  - *recursion* using *stack*
  - *recursion* using *heap*

Semantics

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

- *main*
  - *recursion* using *stack*
  - *recursion* using *heap*
- *test*

Semantics

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

- *main*
  - *recursion* using *stack*
  - *recursion* using *heap*
- *test*
  - ...

Capabilities

Capabilities

- *manipulating state*

Capabilities

- *manipulating state*
- *handling failure*

Capabilities

- *manipulating state*
- *handling failure*
- *handling latency*

Capabilities

- *manipulating state*
- *handling failure*
- *handling latency*
- *handling control*

Capabilities

- *manipulating state*
- *handling failure*
- *handling latency*
- *handling control*
- ...

Effects

Effects

- *reading*

Effects

- *reading*
- *writing*

Monads versus arrows

Monads versus arrows

- *monads* are *computations* that generalize *expressions*, they promote *binding* based, *pointful* functional programming in `PDBP` they correspond to `trait Computation`

Monads versus arrows

- *monads* are *computations* that generalize *expressions*, they promote *binding* based, *pointful* functional programming in `PDBP` they correspond to `trait Computation`
  - `{ val z = ez ; { val y = ey ; /* ... */ } }`

Monads versus arrows

- *monads* are *computations* that generalize *expressions*, they promote *binding* based, *pointful* functional programming in `PDBP` they correspond to `trait Computation`
  - `{ val z = ez ; { val y = ey ; /* ...  */ } }`
  - `mz bind { z => my bind { y => /* ...  */ } }`

Monads versus arrows

- *monads* are *computations* that generalize *expressions*, they promote *binding* based, *pointful* functional programming
  in `PDBP` they correspond to `trait Computation`
  - `{ val z = ez ; { val y = ey ; /* ...  */ } }`
  - `mz bind { z => my bind { y => /* ...  */ } }`
- *arrows* are *programs* that generalize *functions*, they promote *composition* based, *pointfree* functional programming
  in `PDBP` they correspond to `trait Program`

Monads versus arrows

- *monads* are *computations* that generalize *expressions*, they promote *binding* based, *pointful* functional programming
  in `PDBP` they correspond to `trait Computation`
  - `{ val z = ez ; { val y = ey ; /* ... */ } }`
  - `mz bind { z => my bind { y => /* ... */ } }`
- *arrows* are *programs* that generalize *functions*, they promote *composition* based, *pointfree* functional programming
  in `PDBP` they correspond to `trait Program`
  - ``val `z=>x` = `z=>y` andThen `y=>x` ``

Monads versus arrows

- *monads* are *computations* that generalize *expressions*, they promote *binding* based, *pointful* functional programming
  in PDBP they correspond to `trait Computation`
  - `{ val z = ez ; { val y = ey ; /* ... */ } }`
  - `mz bind { z => my bind { y => /* ... */ } }`
- *arrows* are *programs* that generalize *functions*, they promote *composition* based, *pointfree* functional programming
  in PDBP they correspond to `trait Program`
  - ``val `z=>x` = `z=>y` andThen `y=>x` ``
  - ``val `z>-->x` = `z>-->y` >--> `y>-->z` ``

Monads versus arrows

Monads versus arrows

- *monads* promote *pointful programming* but can also be programmed *pointfree* (*kleisli arrows*)

Monads versus arrows

- *monads* promote *pointful programming* but can also be programmed *pointfree* (*kleisli arrows*)
- *arrows* promote *pointfree programming* but can also be programmed *pointful* (*arrow calculus*)

Elegance of use

Elegance of use

- *pointfree* programming is sometimes considered to be more *abstruse* than *pointful* programming

Elegance of use

- *pointfree* programming is sometimes considered to be more *abstruse* than *pointful* programming
- `Scala` comes to the rescue

Elegance of use

- *pointfree* programming is sometimes considered to be more *abstruse* than *pointful* programming
- `Scala` comes to the rescue
    - `Scala` is a `Sca`lable `la`nguage

Elegance of use

- *pointfree* programming is sometimes considered to be more *abstruse* than *pointful* programming
- `Scala` comes to the rescue
  - `Scala` is a `Sca`lable `la`nguage
  - `Scala` *library based* language extensions are *type safe*

Elegance of use

- *pointfree* programming is sometimes considered to be more *abstruse* than *pointful* programming
- `Scala` comes to the rescue
  - `Scala` is a `Sca`lable `la`nguage
  - `Scala` *library based* language extensions are *type safe*
  - `Scala` comes with a *program description DSL*

factorial

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

Constraints Liberate, Liberties Constrain

Constraints Liberate, Liberties Constrain

- *monads* are more *concrete* (less *abstract*) than *arrows*

Constraints Liberate, Liberties Constrain

- *monads* are more *concrete* (less *abstract*) than *arrows*
  - *monads* allow more *specification liberty*

Constraints Liberate, Liberties Constrain

- *monads* are more *concrete* (less *abstract*) than *arrows*
  - *monads* allow more *specification liberty*
  - *monads* impose more *implementation constraints*

PDBP's choice

PDBP's choice

- the PDBP libary goes for

PDBP's choice

- the PDBP libary goes for
  - `private[pdbp]` *pointful monad API*
    provides *power of expression* for *library* developers

PDBP's choice

- the PDBP libary goes for
  - `private[pdbp]` *pointful monad API*
    provides *power of expression* for *library* developers
  - `public` *pointfree arrow API*
    provides *elegance of use* for *application* developers

PDBP's choice

- the PDBP libary goes for
  - `private[pdbp]` *pointful monad API*
    provides *power of expression* for *library* developers
  - `public` *pointfree arrow API*
    provides *elegance of use* for *application* developers
- the PDBP can live with

PDBP's choice

- the PDBP libary goes for
  - `private[pdbp]` *pointful monad API*
    provides *power of expression* for *library* developers
  - `public` *pointfree arrow API*
    provides *elegance of use* for *application* developers
- the PDBP can live with
  - corresponding implementation constraints

PDBP

```
//       _______         __    __        _______
//      / ___  /\       / /\  / /\      / ___  /\
//     / /__/ / / _____/ / / / /_/__   / /__/ / /
//    / _____/ / / ___  / / / ___  /\ /____  / /
//   / /\____\/ / /__/ / / / /__/ / / \___/ / /
//  /_/ /      /______/ / /______/ /     /_/ /
//  \_\/       \______\/  \______\/      \_\/
//                                           v1.0
//  Program Description Based Programming Library
//  author        Luc Duponcheel        2017-2018
```

PDBP library design decisions (cfr. Haskell)

PDBP library design decisions (cfr. Haskell)

- *description* separated from *meaning*

PDBP library design decisions (cfr. Haskell)

- *description* separated from *meaning*
- *description*

PDBP library design decisions (cfr. Haskell)

- *description* separated from *meaning*
- *description*
  - `trait`'s *declare* capabilities (*type classes*)

PDBP library design decisions (cfr. Haskell)

- *description* separated from *meaning*
- *description*
  - `trait`'s *declare* capabilities (*type classes*)
- *language level* meaning

PDBP library design decisions (cfr. Haskell)

- *description* separated from *meaning*
- *description*
  - `trait`'s *declare* capabilities (*type classes*)
- *language level* meaning
  - `implicit object`'s *define* capabilities (*extend* type classes)

PDBP library design decisions (cfr. Haskell)

- *description* separated from *meaning*
- *description*
  - `trait`'s *declare* capabilities (*type classes*)
- *language level* meaning
  - `implicit object`'s *define* capabilities (*extend* type classes)
- *library level* meaning

`PDBP` library design decisions (cfr. `Haskell`)

- *description* separated from *meaning*
- *description*
    - `trait`'s *declare* capabilities (*type classes*)
- *language level* meaning
    - `implicit object`'s *define* capabilities (*extend* type classes)
- *library level* meaning
    - *natural transformations*

PDBP library design decisions (cfr. Haskell)

PDBP library design decisions (cfr. Haskell)

- *definitions* in `class`'es that *implicitly* depend on *type classes* use capabilities *declared* in those type classes

`PDBP` library design decisions (cfr. `Haskell`)

- *definitions* in `class`'es that *implicitly* depend on *type classes* use capabilities *declared* in those type classes
- `object`'s that *extend* those `class`'es `import implicit object`'s

`PDBP` library design decisions (cfr. `Haskell`)

- *definitions* in `class`'es that *implicitly* depend on *type classes* use capabilities *declared* in those type classes
- `object`'s that *extend* those `class`'es `import implicit object`'s
- *dependency injection* by `import`

Program (cfr. *arrow*)

```
trait Program[>-->[- _, + _]]
```

Computation (cfr. *monad*)

```
trait Computation[C[+ _]]
```

Liskov Substitution Principle

Liskov Substitution Principle

- *impose less*

Liskov Substitution Principle

- *impose less*
- *provide more*

Internet Robustness Principle

Internet Robustness Principle

- *be liberal in what you receive*

Internet Robustness Principle

- *be liberal in what you receive*
- *be generous in what you send*

PDBP library details

Program

```
trait Program[>-->[- _, + _]]
    extends Function[>-->]
    with Composition[>-->]
    with Construction[>-->]
    with Condition[>-->]

    with Aggregation[>-->]
```

`Function`

Function

- val ‘z>-->y‘ = function(‘z=>y‘)

Function

- `val ‘z>-->y‘ = function(‘z=>y‘)`
- pure functions are *atomic programs*

Function

- `val 'z>-->y' = function('z=>y')`
- pure functions are *atomic programs*
    - up to you to define granularity

Composition

Composition

- `val ‘z>-->x‘ = ‘z>-->y‘ >--> ‘y>-->x‘`

Construction

Construction

- `` val `z>-->y&&x` = `z>-->y` & `z>-->x` ``

Construction

- ```
  val `z>-->y&&x` = `z>-->y` & `z>-->x`
  ```
- ```
  val `z&&y>-->x&&w` = `z>-->x` && `y>-->w`
  ```

Construction

- `val ‘z>-->y&&x‘ = ‘z>-->y‘ & ‘z>-->x‘`
- `val ‘z&&y>-->x&&w‘ = ‘z>-->x‘ && ‘y>-->w‘`
- ```
  val ‘z>-->x‘ =
     ‘let‘ ‘z>-->y‘ ‘in‘ ‘z&&y>-->x‘
  ```


  ```

Condition

Condition

- ```
  val ‘y||x>-->z‘ = ‘y>-->z‘ | ‘x>-->z‘
  ```


  ```

Condition

- `val ‘y||x>-->z‘ = ‘y>-->z‘ | ‘x>-->z‘`
- `val ‘x||w>-->z||y‘ = ‘x>-->z‘ || ‘w>-->y‘`

Condition

- ```
  val ‘y||x>-->z‘ = ‘y>-->z‘ | ‘x>-->z‘
  ```
- ```
  val ‘x||w>-->z||y‘ = ‘x>-->z‘ || ‘w>-->y‘
  ```
- ```
  val ‘y>-->z‘ =
      ‘if‘(‘y>-->b‘) ‘y>-t->z‘ ‘else‘ ‘y>-f->z‘
  ```


  ```

Computation

```
private[pdbp] trait Computation[C[+ _]]
    extends Resulting[C]
    with Binding[C]
    with Program[[-Z, +Y] => Z => C[Y]]

    with Lifting[C]

    with Sequencing[C]
```

Resulting

```
val cz = result(z)
```

`Binding`

Binding

- `` val cy = cz bind { z => `z=>cy`(y) } ``

Binding

- `val cy = cz bind { z => ‘z=>cy‘(y) }`
- `val cy = cz bind { z => result(‘z=>y‘(y)) }`

`Kleisli`

Kleisli

- `type Kleisli[C[+ _]] = [-Z, + Y] => Z => C[Y]`

Kleisli

- `type Kleisli[C[+ _]] = [-Z, + Y] => Z => C[Y]`

- ```
  private[pdbp] trait Computation[C[+ _]]
      extends Resulting[C]
      with Binding[C]
      with Program[Kleisli[C]]

      // ...
  ```


  ```

factorial

```
val isZeroFunction: BigInt => Boolean =
{ i =>
    i == 0
}
def oneFunction[Z]: Z => BigInt =
{ z =>
    1
}
val subtractOneFunction: BigInt => BigInt =
{ i =>
    i - 1
}
val multiplyFunction: (BigInt && BigInt) => BigInt =
{ (i, j) =>
    i * j
}
```

factorial

```
val isZero: BigInt >--> Boolean =
  function(isZeroFunction)

def one[Z]: Z >--> BigInt =
  function(oneFunction)

val subtractOne: BigInt >--> BigInt =
  function(subtractOneFunction)

val multiply: (BigInt && BigInt) >--> BigInt =
  function(multiplyFunction)
```

factorial

```
val factorial: BigInt >--> BigInt =
  ‘if‘(isZero) {
    one
  } ‘else‘ {
    ‘let‘ {
      subtractOne >-->
        factorial
    } ‘in‘ {
      multiply
    }
  }
```

`programMain`

programMain

- `val producer: Unit >--> Z`

programMain

- `val producer: Unit >--> Z`
- `val program: Z >--> Y`

programMain

- `val producer: Unit >--> Z`
- `val program: Z >--> Y`
- `val consumer: Y >--> Unit`

programMain

- `val producer: Unit >--> Z`
- `val program: Z >--> Y`
- `val consumer: Y >--> Unit`

- ```
  val programMain: Unit >--> Unit =
    producer >--> program >--> consumer
  ```


  ```

factorialMain

```
def effectfulReadIntFromConsoleFunction(message: String):
    Unit => BigInt = {
  _ =>
    println(s"$message")
    val i = BigInt(readInt())
    i
}
def effectfulWriteToConsoleFunction[Y](message: String):
    Y => Unit = { y =>
  println(s"$message")
  val u = println(s"$y")
  u
}
```

factorialMain

```
private def effectfulReadIntFromConsole(message: String):
    Unit >--> BigInt =
  function(effectfulReadIntFromConsoleFunction(message))

private def effectfulWriteToConsole[Y](message: String):
    Y >--> Unit =
  function(effectfulWriteToConsoleFunction(message))

val intProducer: Unit >--> BigInt =
  effectfulReadIntFromConsole("please type an integer")

def factorialOfIntConsumer: BigInt >--> Unit =
  effectfulWriteToConsole("the factorial value of the integer is")
```

factorialMain

```
val factorialMain: Unit >--> Unit =
  intProducer >-->
    factorial >-->
    factorialOfIntConsumer
```

activeTypes

```
object activeTypes {

  type Active[+Z] = Z

  type `=>A` = Kleisli[Active]

}
```

activeProgram

```
implicit object activeProgram
    extends Computation[Active]
    with Program[`=>A`] {

  override private[pdbp] def result[Z]: Z => Active[Z] =
    `z=>az`

  override private[pdbp] def bind[Z, Y](
      az: Active[Z],
      `z=>ay`: => (Z => Active[Y])): Active[Y] =
    `z=>ay`(az)

}
```

Main

```
trait Main[>-->[- _, + _]] {

  val mainKleisliProgram: Unit >--> Unit

  val run: Unit

  def main(args: Array[String]): Unit = {

    run

  }

}
```

FactorialMain

```
object FactorialMain extends Main[`=>A`] {

  override val mainKleisliProgram: Unit `=>A` Unit =
    factorialMain

  override val run =
    mainKleisliProgram(())

}
```

Problems and Solutions

Problems and Solutions

- Problem: obvious `factorial` meaning implementing `>-->` as `` `=>A` `` *is not stack safe*

Problems and Solutions

- Problem: obvious `factorial` meaning implementing `>-->` as `‘=>A‘` *is not stack safe*
  - Solution: `FreeTransformation` and `FreeTransformedMeaning` defined using a *natural transformations*

Problems and Solutions

- Problem: obvious `factorial` meaning implementing `>-->` as `` `=>A` `` *is not stack safe*
  - Solution: `FreeTransformation` and `FreeTransformedMeaning` defined using a *natural transformations*
- Problem: `intProducer` and `factorialOfIntConsumer` *execute effects*

Problems and Solutions

- Problem: obvious `factorial` meaning implementing `>-->` as `‘=>A‘` *is not stack safe*
  - Solution: `FreeTransformation` and `FreeTransformedMeaning` defined using a *natural transformations*
- Problem: `intProducer` and `factorialOfIntConsumer` *execute effects*
  - Solution: `Reading` resp. `Writing` extensions of `Program` with members `read` resp. `write` *describe effects*

NaturalBinaryTypeConstructorTransformation

```
trait `~B~>`[`>-F->`[- _, + _], `>-T->`[- _, + _]] {
  def apply[Z, Y]: Z `>-F->` Y => Z `>-T->` Y
}
```

NaturalUnaryTypeConstructorTransformation

```
private[pdbp] trait `~U~>`[F[+ _], T[+ _]]
    extends `~B~>`[Kleisli[F], Kleisli[T]] {

  private[pdbp] def apply[Z](fz: F[Z]): T[Z]

  private type `=>F` = Kleisli[F]

  private type `=>T` = Kleisli[T]

  override def apply[Z, Y]: Z `=>F` Y => Z `=>T` Y = { `z=>fy` =>
    `z=>fy` andThen apply
  }

}
```

ComputationTransformation

```
private[pdbp] trait ComputationTransformation[
    FC[+ _]: Computation, T[+ _]] {

  private[pdbp] val transform: FC `~U~>` T

}
```

FreeTransformed

```
sealed trait Free[C[+ _], +Z]

final case class Transform[C[+ _], +Z]
  (cz: C[Z]) extends Free[C, Z]
final case class Result[C[+ _], +Z]
  (z: Z) extends Free[C, Z]
final case class Bind[C[+ _], -Z, ZZ <: Z, +Y]
  (fczz: Free[C, ZZ], `z=>fcy`: Z => Free[C, Y])
    extends Free[C, Y]

type FreeTransformed[C[+ _]] = [+Z] => Free[C, Z]
```

FreeTransformation

```
private[pdbp]
  trait FreeTransformation[C[+ _]: Computation]
    extends Computation[FreeTransformed[C]]
    with Program[Kleisli[FreeTransformed[C]]]
    with Transformation[C, FreeTransformed[C]] {

  // ...

}
```

FreeTransformation

```
private type FTFC = FreeTransformed[FC]

override private[pdbp] val transform: FC `~U~>` FTFC = new {
  override private[pdbp] def apply[Z](fcz: FC[Z]): FTFC[Z] =
    Transform(fcz)
}

override private[pdbp] def result[Z]: Z => FTFC[Z] =
  Result(_)

override private[pdbp] def bind[Z, Y]
  (ftfcz: FTFC[Z], `z=>ftfcy`: => (Z => FTFC[Y])): FTFC[Y] =
  Bind(ftfcz, `z=>ftfcy`)
```

activeFreeTypes

```
object activeFreeTypes {

  type ActiveFree = FreeTransformed[Active]

  type `=>AF` = Kleisli[ActiveFree]

}
```

activeFreeProgram

```
import ... activeProgram

implicit object activeFreeProgram
    extends Computation[ActiveFree]
    with Program[‘=>AF‘]
    with FreeTransformation[Active]()
    with ComputationTransformation[Active, ActiveFree]()
```

ProgramMeaning

```
trait ProgramMeaning[
  ‘>-FP->‘[- _, + _]: Program, ‘>-T->‘[- _, + _]] {

  private[pdbp] lazy val binaryTransformation:
      ‘>-FP->‘ ‘~B~>‘ ‘>-T->‘

  lazy val meaning: ‘>-FP->‘ ‘~B~>‘ ‘>-T->‘ =
    binaryTransformation

}
```

ComputationMeaning

```
private[pdbp] trait ComputationMeaning[
  FC[+ _]: Computation, T[+ _]]
    extends ProgramMeaning[Kleisli[FC], Kleisli[T]] {

  private[pdbp] val unaryTransformation: FC `~U~>` T

  private type `=>FC` = Kleisli[FC]

  private type `=>T` = Kleisli[T]

  private[pdbp] override lazy val binaryTransformation:
      `=>FC` `~B~>` `=>T` =
    unaryTransformation

}
```

FreeTransformedMeaning

```
private[pdbp] trait FreeTransformedMeaning[
  FC[+ _]: Computation, T[+ _]](
  implicit toBeTransformedMeaning: ComputationMeaning[FC, T])
  extends ComputationMeaning[FreeTransformed[FC], T] {

  // ...

}
```

FreeTransformedMeaning

```
private val implicitComputation = implicitly[Computation[FC]]

import implicitComputation._

private type FTFC = FreeTransformed[FC]

// ...
```

FreeTransformedMeaning

```
override private[pdbp] val unaryTransformation: FTFC `~U~>` T =
  new {
    override private[pdbp] def apply[Z](ftfcz: FTFC[Z]): T[Z] = {
      @annotation.tailrec
      def tailrecFold(ftfcz: FTFC[Z]): FC[Z] = ftfcz match {
        case Transform(fcz) => fcz
        case Result(z) => result(z)
        case Bind(Result(y), y2ftfcz) =>
          tailrecFold(y2ftfcz(y))
        case Bind(Bind(fcx, x2ftfcy), y2ftfcz) =>
          tailrecFold(Bind(fcx, { x =>
            Bind(x2ftfcy(x), y2ftfcz) }))
        case any => sys.error("Impossible ...")
      }
      toBeTransformedMeaning.unaryTransformation(
        tailrecFold(ftfcz))
    }
  }
```

MeaningOfActive

```
private[pdbp] trait MeaningOfActive[
  TR[+ _]: Resulting] extends ComputationMeaning[Active, TR] {

  override private[pdbp] val unaryTransformation:
      Active `~U~>` TR =
    new {
      override private[pdbp] def apply[Z](
        az: Active[Z]): TR[Z] = {
        import implicitly._
        result(az)
      }
    }

}
```

activeMeaningOfActive

```
implicit object activeMeaningOfActive
    extends MeaningOfActive[Active]()
    with ComputationMeaning[Active, Active]()
    with ProgramMeaning[‘=>A‘, ‘=>A‘]()
```

activeMeaningOfActiveFree

```
import ... activeMeaningOfActive

implicit object activeMeaningOfActiveFree
    extends FreeTransformedMeaning[Active, Active]()
    with ComputationMeaning[ActiveFree, Active]()
    with ProgramMeaning[`=>AF`, `=>A`]()
```

FactorialMain

```
import activeMeaningOfActiveFree.meaning
import mainFactorial.factorialMain

object FactorialMain extends Main[`=>A`] {

  override val mainKleisliProgram: Unit `=>A` Unit =
    meaning(factorialMain)

  override val run = mainKleisliProgram(())

}
```

Reading

```
trait Reading[R, >-->[- _, + _]] {
  this: Function[>-->] & Composition[>-->] =>

  private[pdbp] def `u>-->r`: Unit >--> R






}
```

Reading

```
trait Reading[R, >-->[- _, + _]] {
  this: Function[>-->] & Composition[>-->] =>

  private[pdbp] def `u>-->r`: Unit >--> R = `z>-->r`[Unit]

  private[pdbp] def `z>-->r`[Z]: Z >--> R =
    compose(`z>-->u`, `u>-->r`)

  def read[Z]: Z >--> R = `z>-->r`

}
```

ReadingTransformed

```
type `I=>`[-X, +Y] = implicit X => Y

type ReadingTransformed[R, C[+ _]] = [+Z] => R `I=>` C[Z]
```

ReadingTransformation

```
private[pdbp] trait ReadingTransformation[
  R, FC[+ _]: Computation]
    extends ComputationTransformation[
      FC, ReadingTransformed[R, FC]]
    with Computation[ReadingTransformed[R, FC]]
    with Program[Kleisli[ReadingTransformed[R, FC]]]
    with Reading[R, Kleisli[ReadingTransformed[R, FC]]] {

  // ...

}
```

ReadingTransformation

```
private type RTFC = ReadingTransformed[R, FC]
private type `=>RTFC` = Kleisli[RTFC]

import implicitly.{result => resultFC}
import implicitly.{bind => bindFC}
```

ReadingTransformation

```
override private[pdbp] val transform: FC `~U~>` RTFC = new {
  override private[pdbp] def apply[Z](fcz: FC[Z]): RTFC[Z] =
  fcz
}

override private[pdbp] def result[Z]: Z => RTFC[Z] = { z =>
  resultFC(z)
}

override private[pdbp] def bind[Z, Y]
  (rtfcz: RTFC[Z], `z>=rtfcy`: => (Z => RTFC[Y])): RTFC[Y] =
  bindFC(rtfcz, `z>=rtfcy`(_))
```

ReadingTransformation

```
// ...

private[pdbp] override def `u>-->r`: Unit `=>RTFC` R = { _ =>
  resultFC(implicitly)
}
```

factorialMain

```
val factorialMain: Unit >--> Unit =
  read >-->
    factorial >-->
    factorialOfIntConsumer
```

activeReadingTypes

```
object activeReadingTypes {

  type ActiveReading[R] = ReadingTransformed[R, Active]

  type `=>AR`[R] = Kleisli[ActiveReading[R]]

}
```

ActiveReadingProgram

```
private[pdbp] trait ActiveReadingProgram[R]
    extends Computation[ActiveReading[R]]
    with Program[`=>AR`[R]]
    with Reading[R, `=>AR`[R]]
    with ComputationTransformation[Active, ActiveReading[R]]
    with ReadingTransformation[R, Active]
```

activeIntReadingProgram

```
import  ... activeProgram

implicit object activeIntReadingProgram
  extends ActiveReadingProgram[BigInt]()
  with ComputationTransformation[Active, ActiveReading[BigInt]]()
  with ReadingTransformation[BigInt, Active]()
  with Reading[BigInt, `=>AR`[BigInt]]
```

FactorialMain

```
object FactorialOfIntReadMain extends Main[‘=>AR‘[BigInt]] {

  import ... readIntFromConsoleEffect

  private type ‘=>AR[BigInt]‘ = ‘=>AR‘[BigInt]

  override val mainKleisliProgram: Unit ‘=>AR[BigInt]‘ Unit =
    factorialMain

  override val run =
    mainKleisliProgram(())

}
```

factorialMultipliedByIntRead

```
val factorialMultipliedByIntRead: BigInt >--> BigInt =
    (factorial & read) >--> multiply
```

factorialMultipliedByIntReadMain

```
val factorialMultipliedByIntReadMain: Unit >--> Unit =
  intProducer >-->
    factorialMultipliedByIntRead >-->
    factorialOfIntMultipliedByIntReadConsumer
```

Writing

```
trait Writing[W: Writable, >-->[- _, + _]] {
  this: Function[>-->] & Composition[>-->] =>

  private[pdbp] val `w>-->u`: W >--> Unit




}
```

Writing

```
trait Writing[W: Writable, >-->[- _, + _]] {
  this: Function[>-->] & Composition[>-->] =>

  private[pdbp] val `w>-->u`: W >--> Unit = write(identity)

  def write[Z]: (Z => W) `I=>` Z >--> Unit =
    compose(function(implicitly), `w>-->u`)

}
```

Writable

```
private[pdbp] trait Writable[W]
    extends Startable[W]
    with Appendable[W]
    with Lifting[Const[W]] {

  override private[pdbp] def liftFunction[Z, Y](
    `z=>y`: Z => Y): W => W = `w=>w`

}
```

Startable

```
private[pdbp] trait Startable[W]
    extends ObjectLifting[Const[W]] {

  private[pdbp] val start: W

  override private[pdbp] def liftObject[Z](z: Z): W = start

}
```

Appendable

```
private[pdbp] trait Appendable[W]
    extends OperatorLifting[Const[W]] {

  private[pdbp] val append: W && W => W

  override private[pdbp] def liftOperator[Z, Y, X](
      `(z&&y)=>x`: (Z && Y) => X): (W && W) => W = append

}
```

WritingTransformed

```
type WritingTransformed[W, FC[+ _]] = [+Z] => FC[W && Z]
```

WritingTransformation

```
private[pdbp] trait WritingTransformation[
  W: Writable, FC[+ _]: Computation]
    extends ComputationTransformation[
      FC, WritingTransformed[W, FC]]
    with Computation[WritingTransformed[W, FC]]
    with Program[Kleisli[WritingTransformed[W, FC]]]
    with Writing[W, Kleisli[WritingTransformed[W, FC]]] {

  // ...

}
```

WritingTransformation

```
private type WTFC = WritingTransformed[W, FC]
private type `=>WTFC` = Kleisli[WTFC]

private val implicitComputation = implicitly[Computation[FC]]

import implicitComputation.{bind => bindFC}
import implicitComputation.{result => resultFC}

private val implicitWritable = implicitly[Writable[W]]

import implicitWritable._
```

WritingTransformation

```
override private[pdbp] val transform: FC `~U~>` WTFC = new {
  override private[pdbp] def apply[Z](fcz: FC[Z]): WTFC[Z] =
    bindFC(fcz, { z =>
      resultFC((start, z))
    })
}

override private[pdbp] def result[Z]: Z => WTFC[Z] = { z =>
  resultFC((start, z))
}

override private[pdbp] def bind[Z, Y](
  wtfcz: WTFC[Z], `z=>wtfcy`: => (Z => WTFC[Y])): WTFC[Y] =
  bindFC(wtfcz, { (leftW, z) =>
    bindFC(`z=>wtfcy`(z), { (rightW, y) =>
    resultFC(append(leftW, rightW), y)
    })
  })
```

WritingTransformation

```
// ...

private[pdbp] override val `w>-->u`: W `=>WTFC` Unit = { w =>
  resultFC((w, ()))
}
```

factorialMain

```
val factorialMain: (BigInt => ToConsole) `I=>` Unit >--> Unit =
  read >-->
    factorial >-->
    write
```

ToConsole

```
case class ToConsole(effect: Effect)

type Effect = Unit => Unit
```

ToConsole

```
implicit object toConsoleWritable extends Writable[ToConsole] {

  override private[pdbp] val start: ToConsole =
    ToConsole { _ =>
      ()
    }

  override private[pdbp] val append:
      ToConsole && ToConsole => ToConsole = {
    (tc1, tc2) =>
      ToConsole { _ =>
        { tc1.effect(()); tc2.effect(()) }
      }
  }

}
```

activeWritingTypes

```
object activeWritingTypes {

  type ActiveWriting[W] = WritingTransformed[W, Active]

  type `=>AW`[W] = Kleisli[ActiveWriting[W]]

}
```

activeWritingTypes

```
object activeReadingWithWritingTypes {

  type ActiveReadingWithWriting[R, W] =
    ReadingTransformed[R, ActiveWriting[W]]

  type `=>ARW`[R, W] = Kleisli[ActiveReadingWithWriting[R, W]]

}
```

ActiveReadingWithWritingProgram

```
trait ActiveReadingWithWritingProgram[R, W: Writable]
    extends Computation[ActiveReadingWithWriting[R, W]]
    with Program[`=>ARW`[R, W]]
    with Reading[R, `=>ARW`[R, W]]
    with Writing[W, `=>ARW`[R, W]]
    with ComputationTransformation[
      ActiveWriting[W], ActiveReadingWithWriting[R, W]]
    with ReadingWithWritingTransformation[
      R, W, ActiveWriting[W]]
```

activeIntReadingWithWritingToConsoleProgram

```
import ... toConsoleWritable
import ... activeWritingToConsoleProgram

implicit object activeIntReadingWithWritingToConsoleProgram
  extends ActiveReadingWithWritingProgram[BigInt, ToConsole]()
  with ComputationTransformation[ActiveWriting[
    ToConsole], ActiveReadingWithWriting[BigInt, ToConsole]]()
  with ReadingTransformation[BigInt, ActiveWriting[ToConsole]]()
  with ReadingWithWritingTransformation[
    BigInt, ToConsole, ActiveWriting[ToConsole]]()
  with Reading[BigInt, `=>ARW`[BigInt, ToConsole]]()
  with Writing[ToConsole, `=>ARW`[BigInt, ToConsole]]()
```

FactorialOfIntReadWrittenToConsoleMain

```
object FactorialOfIntReadWrittenToConsoleMain
    extends Main[‘=>ARW‘[BigInt, ToConsole]] {

  import ... readIntFromConsoleEffect
  import ... writeFactorialOfIntReadToConsoleEffect

  private type ‘=>ARW[BigInt, ToConsole]‘ = ‘=>ARW‘[BigInt, ToConsole]

  override val mainKleisliProgram:
    Unit ‘=>ARW[BigInt, ToConsole]‘ Unit = factorialMain

  override val run = mainKleisliProgram(()) match {
    case (ToConsole(effect), _) => effect(())
  }

}
```