PDBP

Program Description Based Programming
Scala eXchange

Luc Duponcheel

August 1, 2018

Intro

Intro

- this talk is about

Intro

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

Intro

- this talk is about
  - a `Dotty` *library* `PDBP`
  - inspired by the *function-level programming language* `FP`

John Backus

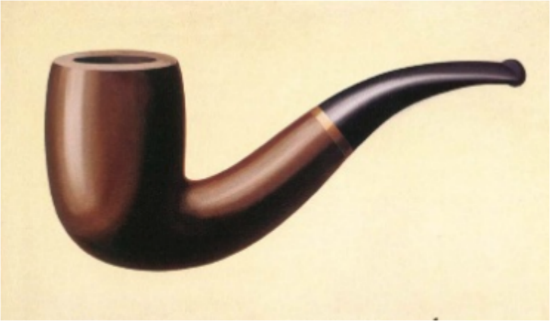
John Backus

- ACM Turing Award Winner 1977

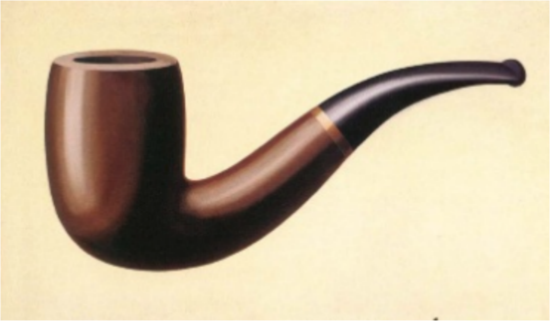
John Backus

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

What is this?

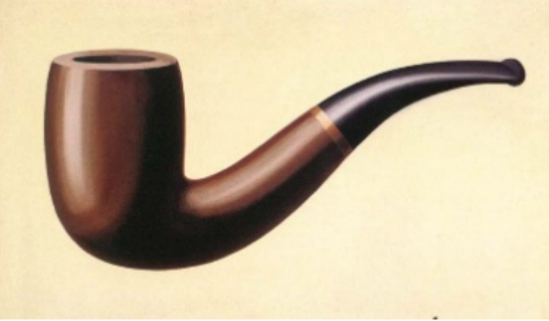


What is this?



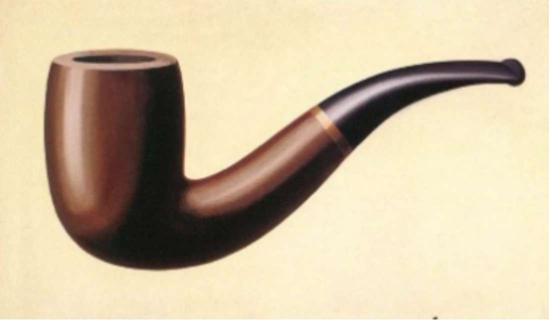
- A *pipe*?

What is this?



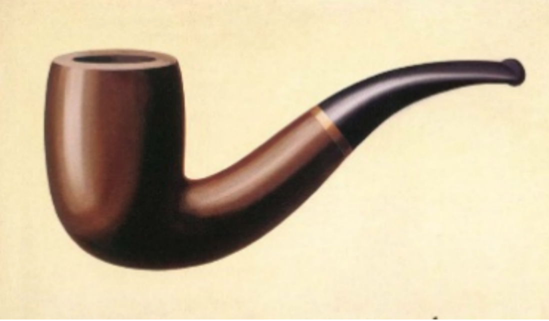
- A *pipe*?
- A *painting describing* a pipe?

What is this?



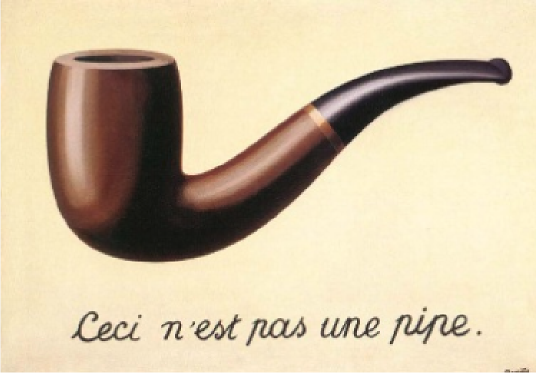
- A *pipe*?
- A *painting describing* a pipe?
- A *slide describing* a painting describing a pipe?

What is this?

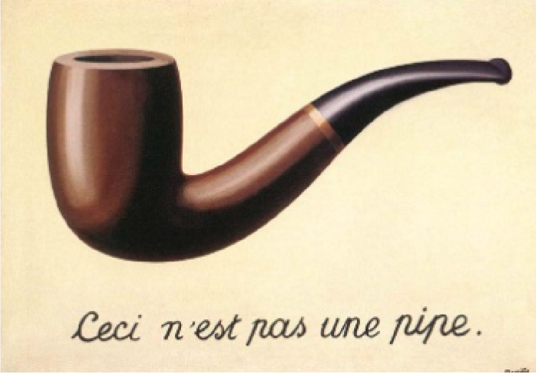


- A *pipe*?
- A *painting describing* a pipe?
- A *slide describing* a painting describing a pipe?
- . . .

Ceci n'est pas une pipe



Ceci n'est pas une pipe



- It is a *painting* by René Magritte *describing* a pipe

This is not a program

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

This is not a program

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

- It is *code describing* a program

`factorial` description

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

`factorial` description

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

- `factorial` *uses* capabilities *declared* in type class `trait Program[>-->[- _, + _]]`

factorial meanings

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

`factorial` meanings

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

- different `implicit object`'s *define* capabilities of type class `trait Program[>-->[- _, + _]]`

program

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

Composition

```
val ‘z>-->y‘: Z >--> Y
val ‘y>-->x‘: Y >--> x

val ‘z>-->x‘: Z >--> X =  ‘z>-->y >--> ‘y>-->x‘
```

mainProgram

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

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

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
    }
  }
```

- *production*

Semantics

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

- *production*
  - *recursion* using *stack*

Semantics

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

- *production*
  - *recursion* using *stack*
  - *recursion* using *heap*

Semantics

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

- *production*
  - *recursion* using *stack*
  - *recursion* using *heap*
- *test*

Semantics

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

- *production*
  - *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*

Foundations

Foundations

- `trait Program[>-->[- _, + _]]` corresponds to *arrows*

Foundations

- `trait Program[>-->[- _, + _]]` corresponds to *arrows*
- `trait Computation[C[+ _]]` corresponds to *monads*

Monads versus arrows

Monads versus arrows

- *arrows* (`trait Program[>-->[- _, + _]]`)
  generalize *functions*

Monads versus arrows

- *arrows* (`trait Program[>-->[- _, + _]]`) generalize *functions*
    - *composition* based, *pointfree* functional programming

Monads versus arrows

- *arrows* (`trait Program[>-->[- _, + _]]`) generalize *functions*
  - *composition* based, *pointfree* functional programming
  - `` val `z=>x` = `z=>y` andThen `y=>x` ``

Monads versus arrows

- *arrows* (`trait Program[>-->[- _, + _]]`) generalize *functions*
  - *composition* based, *pointfree* functional programming
  - `` val `z=>x` = `z=>y` andThen `y=>x` ``
  - `` val `z>-->x` = `z>-->y` >--> `y>-->z` ``

Monads versus arrows

- *arrows* (`trait Program[>-->[- _, + _]]`) generalize *functions*
  - *composition* based, *pointfree* functional programming
  - `` val `z=>x` = `z=>y` andThen `y=>x` ``
  - `` val `z>-->x` = `z>-->y` >--> `y>-->z` ``
- *monads* (`trait Computation[C[+ _]]`) generalize *expressions*

Monads versus arrows

- *arrows* (`trait Program[>-->[- _, + _]]`) generalize *functions*
  - *composition* based, *pointfree* functional programming
  - val ‘z=>x‘ = ‘z=>y‘ andThen ‘y=>x‘
  - val ‘z>-->x‘ = ‘z>-->y‘ >--> ‘y>-->z‘
- *monads* (`trait Computation[C[+ _]]`) generalize *expressions*
  - *binding* based, *pointful* functional programming

Monads versus arrows

- *arrows* (trait Program[>-->[- _, + _]]) generalize *functions*
  - *composition* based, *pointfree* functional programming
  - val ‘z=>x‘ = ‘z=>y‘ andThen ‘y=>x‘
  - val ‘z>-->x‘ = ‘z>-->y‘ >--> ‘y>-->z‘
- *monads* (trait Computation[C[+ _]]) generalize *expressions*
  - *binding* based, *pointful* functional programming
  - { val z = ez ; { val y = ey ; ex(z, y) } }

Monads versus arrows

- *arrows* (`trait Program[>-->[- _, + _]]`) generalize *functions*
  - *composition* based, *pointfree* functional programming
  - ``val `z=>x` = `z=>y` andThen `y=>x` ``
  - ``val `z>-->x` = `z>-->y` >--> `y>-->z` ``
- *monads* (`trait Computation[C[+ _]]`) generalize *expressions*
  - *binding* based, *pointful* functional programming
  - `{ val z = ez ; { val y = ey ; ex(z, y) } }`
  - `mz bind { z => my bind { y => mx(z, y) } }`

Monads versus arrows

- *arrows* (`trait Program[>-->[- _, + _]]`)
  generalize *functions*
  - *composition* based, *pointfree* functional programming
  - `` val `z=>x` = `z=>y` andThen `y=>x` ``
  - `` val `z>-->x` = `z>-->y` >--> `y>-->z` ``
- *monads* (`trait Computation[C[+ _]]`)
  generalize *expressions*
  - *binding* based, *pointful* functional programming
  - `{ val z = ez ; { val y = ey ; ex(z, y) } }`
  - `mz bind { z => my bind { y => mx(z, y) } }`
  - `mz bind { z => my bind { y => result(ex(z, y)) } }`

Monads versus arrows

Monads versus arrows

- *monads* can also be programmed *pointfree* (*kleisli arrows*)

Monads versus arrows

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

Power of expressions

Power of expressions

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

Power of expressions

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

Power of expressions

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

Power of expressions

- *monads* are more *concrete* (less *abstract*) than *arrows*
  - *monads* allow more *specification liberty*
  - *monads* impose more *implementation constraints*
- *Constraints Liberate, Liberties Constrain*

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
- `Dotty` comes to the rescue

Elegance of use

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

Elegance of use

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

Elegance of use

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

Program description DSL

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

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 (*type classes*) *declare* capabilities

PDBP library design decisions (cfr. Haskell)

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

PDBP library design decisions (cfr. Haskell)

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

PDBP library design decisions (cfr. Haskell)

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

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

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

PDBP library design decisions (cfr. Haskell)

PDBP library design decisions (cfr. Haskell)

- *dependency injection* by `implicit import`

PDBP library design decisions (cfr. Haskell)

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

PDBP library design decisions (cfr. `Haskell`)

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

`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]]

      // ...
  ```


  ```

HelperFunctions

```
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
}
```

HelperPrograms

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

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

val multiplyHelper: (BigInt && BigInt) >--> BigInt =
  function(multiplyFunction)

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

AtomicPrograms

```
val isZero: BigInt >--> Boolean =
  isZeroHelper

val subtractOne: BigInt >--> BigInt =
  subtractOneHelper

val multiply: (BigInt && BigInt) >--> BigInt =
  multiplyHelper

def one[Z]: Z >--> BigInt =
  oneHelper
```

factorial

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

factorialMain

```
val factorialMain: Unit >--> Unit =
  producer >-->
    factorial >-->
    consumer
```

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`

Problems and Solutions

- Problem: obvious `factorial` meaning implementing `>-->` as `` `=>A` `` *is not stack safe*
  - Solution: `FreeTransformation` and `FreeTransformedMeaning`
- Problem: obvious `producer` and `consumer` *execute effects*

Problems and Solutions

- Problem: obvious `factorial` meaning implementing `>-->` as `` `=>A` `` *is not stack safe*
  - Solution: `FreeTransformation` and `FreeTransformedMeaning`
- Problem: obvious `producer` and `consumer` *execute effects*
  - Solution: `Reading` resp. `Writing` extensions of `Program` with members `read` resp. `write` that *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]

  // ...
  }

}
```

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]] {

  // unfold
  // transform => Transform
  // result => Result
  // bind => Bind

}
```

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

  // ...

}
```

FreeTransformedMeaning

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

  // tailrecFold
  // Transform(fcz) => fcz
  // Result(z) => result(z)
  // Bind(Result(y), y2ftfcz) => tailrecFold(y2ftfcz(y))
  // Bind(Bind(fcx, x2ftfcy), y2ftfcz) =>
  //   tailrecFold(Bind(fcx, Bind(x2ftfcy, y2ftfcz)))

}
```

activeMeaningOfActiveFree

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

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`)



}
```

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]]] {

  // ...

}
```

factorialMain

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

activeReadingTypes

```
object activeReadingTypes {
  type ActiveReading[R] = ReadingTransformed[R, Active]
  type `=>AR`[R] = Kleisli[ActiveReading[R]]
}
```

activeIntReadingProgram

```
import  ... activeProgram

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

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`)

  // ...

}
```

Writing

```
def writeUsing[Z, Y, X](
    `(z&&y)=>x`: ((Z && Y) => X)):
    (Z >--> Y) => ((X => W) `I=>` Z >--> Y) = { `z>-->y` =>
    val `(z&&y)>-->x` = function(`(z&&y)=>x`)
    val `z>-->(x&&y)` =
      `let` {
        `z>-->y`
      } `in` {
        `let` {
          `(z&&y)>-->x`
        } `in` {
          `(z&&y&&x)>-->(x&&y)`
        }
      }
    compose(compose(`z>-->(x&&y)`, left(write)),
      `(u&&y)>-->y`)
}
```

Writable

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

Startable

```
private[pdbp] trait Startable[W] {

  private[pdbp] val start: W

}
```

Appendable

```
private[pdbp] trait Appendable[W] {

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

}
```

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]]] {

  // ...

}
```

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(()) }
      }
  }

}
```

factorialMain

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

activeIntReadingWithWritingToConsoleProgram

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

implicit object activeIntReadingWithWritingToConsoleProgram
    extends ActiveReadingWithWritingProgram[BigInt, ToConsole]()
    with Computation[ActiveReadingWithWriting[BigInt, ToConsole]]()
    with Program[`=>ARW`[BigInt, ToConsole]]()
    with Reading[BigInt, `=>ARW`[BigInt, ToConsole]]()
    with Writing[ToConsole, `=>ARW`[BigInt, ToConsole]]()
    // ...
```

activeIntReadingWithWritingToConsoleProgram

```
// ...
with ReadingWithWritingTransformation[BigInt,
                                      ToConsole,
                                      ActiveWriting[ToConsole]]()
with ReadingTransformation[BigInt, ActiveWriting[ToConsole]]()
with ComputationTransformation[
  ActiveWriting[ToConsole],
  ActiveReadingWithWriting[BigInt, ToConsole]]()
```

infoUtils

```
def infoFunction[Z, Y](string: String): Z && Y => String = {
  case (z, y) =>
    s"INFO -- $currentCalendarInMilliseconds -- $string($z) => $y"
}

def info[
    W: Writable, Z, Y,
    >-->[- _, + _]: [>-->[- _, + _]] => Writing[W, >-->]]
    (string: String):
    (Z >--> Y) => ((String => W) `I=>` Z >--> Y) = {
  val implicitWriting = implicitly[Writing[W, >-->]]
  implicitWriting.writeUsing(infoFunction(string))
}
```

WritingAtomicPrograms

```
val isZero: (String => W) ‘I=>‘ BigInt >--> Boolean =
  info("isZero") { isZeroHelper }

val subtractOne: (String => W) ‘I=>‘ BigInt >--> BigInt =
  info("subtractOne") { subtractOneHelper }

val multiply: (String => W) ‘I=>‘ (BigInt && BigInt) >--> BigInt =
  info("multiply") { multiplyHelper }

def one[Z]: (String => W) ‘I=>‘ Z >--> BigInt =
  info("one") { oneHelper }
```

WritingFactorial

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

FactorialOfIntRead WritingToConsoleWrittenToConsoleMain

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

  import ... readIntFromConsoleEffect
  import ... writeFactorialOfIntReadToConsoleEffect
  import ... writeToConsoleEffect

  // ...

}
```

FactorialOfIntRead WritingToConsoleWrittenToConsoleMain

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

  // ...

  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(())
}
```

running

```
please type an integer to read
2
INFO -- 2018-08-01 18:00:42.639 -- isZero(2) => false
INFO -- 2018-08-01 18:00:42.645 -- subtractOne(2) => 1
INFO -- 2018-08-01 18:00:42.646 -- isZero(1) => false
INFO -- 2018-08-01 18:00:42.647 -- subtractOne(1) => 0
INFO -- 2018-08-01 18:00:42.647 -- isZero(0) => true
INFO -- 2018-08-01 18:00:42.648 -- one(0) => 1
INFO -- 2018-08-01 18:00:42.648 -- factorial(0) => 1
INFO -- 2018-08-01 18:00:42.649 -- multiply((1,1)) => 1
INFO -- 2018-08-01 18:00:42.649 -- factorial(1) => 1
INFO -- 2018-08-01 18:00:42.650 -- multiply((2,1)) => 2
INFO -- 2018-08-01 18:00:42.650 -- factorial(2) => 2
the factorial value of the integer read is
2
```