LambdaDays @ Krakow
February 26, 2015

Modular Syntax and Semantics

Luc Duponcheel

Great Book

Great Book

- *Functional Programming in Scala*
  (Paul Chiusano, Runar Bjarnason)

Strong[Typed]
Scala Training and Consultancy

Great Book

- *Functional Programming in Scala*
  (Paul Chiusano, Runar Bjarnason)
- this talk is

Great Book

- *Functional Programming in Scala*
  (Paul Chiusano, Runar Bjarnason)
- this talk is
  - inspired by *chapter 13* (effects)

Great Book

- *Functional Programming in Scala*
  (Paul Chiusano, Runar Bjarnason)
- this talk is
  - inspired by *chapter 13* (effects)
  - compatible with *chapter 15* (streaming)

Some History

Some History

- 1993
  Composing Monads
  (Mark Jones, Luc Duponcheel)
  modular semantics

Some History

- 1993
  Composing Monads
  (Mark Jones, Luc Duponcheel)
  modular semantics
- 1995
  Monad transformers and modular interpreters
  (Sheng Liang, Paul Hudak, Mark Jones)
  better modular semantics

Some History

- 1993
  Composing Monads
  (Mark Jones, Luc Duponcheel)
  modular semantics
- 1995
  Monad transformers and modular interpreters
  (Sheng Liang, Paul Hudak, Mark Jones)
  better modular semantics
- 1995
  Using Catamorphisms, Subtypes and Monad Transformers for Writing Modular Functional Interpreters
  (Luc Duponcheel)
  algorithmic approach to modular syntax

Some More History

Some More History

- 2008
  Data types a la carte
  (Wouter Swierstra)
  data structures approach to modular syntax

Some More History

- 2008
  Data types a la carte
  (Wouter Swierstra)
  data structures approach to modular syntax
- 2014
  Composable application architecture with reasonably priced monads
  (Runar Bjarnason)
  Scala implementation

Syntax and Semantics

Syntax and Semantics

- in general, this talk is, roughly speaking, about the separation of

Syntax and Semantics

- in general, this talk is, roughly speaking, about the separation of
- *syntax*

Syntax and Semantics

- in general, this talk is, roughly speaking, about the separation of
- *syntax*
  - the *description* of programs

Syntax and Semantics

- in general, this talk is, roughly speaking, about the separation of
- *syntax*
  - the *description* of programs
  - as *data structures*

Syntax and Semantics

- in general, this talk is, roughly speaking, about the separation of
- *syntax*
  - the *description* of programs
  - as *data structures*
- *semantics*

Syntax and Semantics

- in general, this talk is, roughly speaking, about the separation of
- *syntax*
  - the *description* of programs
  - as *data structures*
- *semantics*
  - the *meaning* of those programs

Syntax and Semantics

- in general, this talk is, roughly speaking, about the separation of
- *syntax*
  - the *description* of programs
  - as *data structures*
- *semantics*
  - the *meaning* of those programs
  - often (but not necessarily) as *executable algorithms*

Effects

Effects

- in particular, this talk is, roughly speaking, about the separation of

Effects

- in particular, this talk is, roughly speaking, about the separation of
- *effects*

Effects

- in particular, this talk is, roughly speaking, about the separation of
- *effects*
  - program syntax *describes* them

Effects

- in particular, this talk is, roughly speaking, about the separation of
- *effects*
  - program syntax *describes* them
- *side effects*

Effects

- in particular, this talk is, roughly speaking, about the separation of
- *effects*
  - program syntax *describes* them
- *side effects*
  - program semantics *executes* them

Somewhat special quote from Linus Torvalds (among others, author of `git`)

Somewhat special quote from Linus Torvalds (among others, author of `git`)

- `git` actually has a simple design, with stable and reasonably well-documented data structures. In fact, I'm a huge proponent of *designing your algorithms around the data, rather than the other way around*, and I think it's one of the reasons `git` has been fairly successful.

Somewhat special quote from Linus Torvalds (among others, author of `git`)

- `git` actually has a simple design, with stable and reasonably well-documented data structures. In fact, I'm a huge proponent of *designing your algorithms around the data, rather than the other way around*, and I think it's one of the reasons `git` has been fairly successful.
- I will, in fact, claim that the difference between a *bad programmer* and a *good one* is whether he *considers his algorithms or his data structures more important*.

Introduction

Id

```
type Id[+Z] = Z
```

Strong[Typed]
Scala Training and Consultancy

Product

```
type **[+Z, +Y] = (Z, Y)
```

Sum

```
type ++[+Z, +Y] = Either[Z, Y]
```

expression02

```scala
def expression02: String = {
  val z = "Hello "
  val y = "LambdaDays "
  z + y
}
```

expression04

```scala
def expression04: String = {
  val (z, y) = ("Hello ", "LambdaDays ")

  z + y
}
```

plus01

```
val plus: String ** String => String = {
  case (z, y) => z + y
}
```

expression05

```
def expression05: String =
  plus apply ("Hello ", "LambdaDays ")
```

expression06

```
def expression06: String =
  ("Hello ", "LambdaDays ") bind plus
```

main01

```
def main(args: Array[String]): Unit = {
  println(expression01)
  println(expression02)
  println(expression03)
  println(expression04)
  println(expression05)
  println(expression06)
}
```

operators03

```
def res[Z](z: => Z): Id[Z] =
  z

implicit class Ops[Z](iz: Id[Z]) {
  def and[Y](iy: Id[Y]): Id[Z ** Y] =
    (iz, iy)
  def bnd[Y](z2iy: Z => Id[Y]): Id[Y] =
    z2iy(iz)
  def end[Y](z2y: Z => Y): Id[Y] =
    z2y(iz)
}
```

bnd01Expression03

```
val bnd01Expression: Id[String] =
  res("Hello ") bnd { case z =>
    res("LambdaDays ") bnd { case y =>
      res(z + y)
    }
  }
```

and01Expression03

```
val and01Expression: Id[String] =
  res("Hello ") and
    res("LambdaDays ") bnd { case (z, y) =>
      res(z + y)

  }
```

bnd02Expression03

```
val bnd02Expression: Id[String] =
  res("Hello ") bnd { case z =>
    res("LambdaDays ") end { case y =>
      z + y
    }
  }
```

and02Expression03

```
val and02Expression: Id[String] =
  res("Hello ") and
    res("LambdaDays ") end { case (z, y) =>
      z + y

  }
```

and04Expression03

```
val and04Expression: Id[String] =
  res("Hello ") and
    res("LambdaDays ") end
      plus
```

Strong[Typed]
Scala Training and Consultancy

main03

```
def main(args: Array[String]): Unit = {
  println(bnd01Expression)
  println(bnd02Expression)
  println(and01Expression)
  println(and02Expression)
  println(and03Expression)
  println(and04Expression)
}
```

Println Program

bnd01ProgramSyntax04

```
res("Hello ") bnd { case z =>
  res("LambdaDays ") bnd { case y =>
    res(z + y) bnd { case x =>
      println(x)
    }
  }
}
```

bnd02ProgramSyntax04

```
res("Hello ") bnd { case z =>
  res("LambdaDays ") bnd { case y =>
    res(z + y)
  }
} bnd { case x =>
  println(x)
}
```

bnd03ProgramSyntax04

```
res("Hello ") bnd { case z =>
  res("LambdaDays ") end { case y =>
    z + y
  }
} bnd { case x =>
  println(x)
}
```

and01ProgramSyntax04

```
res("Hello ") and
  res("LambdaDays ") end { case (z, y) =>
    z + y

} bnd { case x =>
  println(x)
}
```

and02ProgramSyntax04

```
res("Hello ") and
  res("LambdaDays ") end
    plus

  bnd
  println[Sntx]
```

Strong[Typed]
Scala Training and Consultancy

semantics04

```
val bndSemantics01 =
  meaning.apply(bnd01ProgramSyntax)

val bndSemantics02 =
  meaning.apply(bnd02ProgramSyntax)

val bndSemantics03 =
  meaning.apply(bnd03ProgramSyntax)

val andSemantics01 =
  meaning.apply(and01ProgramSyntax)

val andSemantics02 =
  meaning.apply(and02ProgramSyntax)
```

main04

```
def main(args: Array[String]): Unit = {
  resourceSafeExec(bndSemantics01)
  resourceSafeExec(bndSemantics02)
  resourceSafeExec(bndSemantics03)
  resourceSafeExec(andSemantics01)
  resourceSafeExec(andSemantics02)
}
```

End

End

```
trait End[E[+_]] {
  def end[Z, Y](z2y: Z => Y): E[Z] => E[Y]
}

object End {
  def apply[E[+_]: End] = implicitly[End[E]]
}
```

End Law 1

```
     Z
     |
id   |
     |
     v
     Z
```


Strong[Typed]
Scala Training and Consultancy

End Law 1

```
Z                    F[Z]
|                     |
|      End[F].end(id) | id
|                     |
v                     v
Z                    F[Y]
```

End Law 2

```
     Z  --
     |    |
     |    |
z2y  |    |
     v    |
     Y    |  z2y andThen y2x
     |    |
y2x  |    |
     |    |
     v    |
     X <--
```

End Law 2

```
Z  --                      F[Z] --
|    |                       |     |
|    |                       |     |
|    |     End[F].end(z2y)   |     |
v    |                       v     |
Y    |     andThen          F[Y]   | End[F](z2y andThen y2x)
|    |                       |     |
|    |     End[F].end(y2x)   |     |
|    |                       |     |
v    |                       v     |
X <--                      F[X] <--
```

Strong[Typed]
Scala Training and Consultancy

Natural Transformation

NaturalTransformation

```
trait ->[F[+_], H[+_]] { `f->h` =>
  def apply[Z](fz: F[Z]): H[Z]
  // ...
```

Natural Transformation Law

```
    Z
    |
z2y |
    |
    v
    Y
```

Natural Transformation Law

```
                                   apply
Z                   F[Z] -----> H[Z]
|                    |            |
|     End[F].end(z2y) |            | End[H].end(z2y)
|                    |            |
v                    v            v
Y                   F[Y] -----> H[Y]
                                   apply
```

list2option

```
val list2option: List -> Option =
  new (List -> Option) {
    def apply[Z](zs: List[Z]) = zs match {
      case Nil => None
      case z::_ => Some(z)
    }
  }
```

Natural Transformation Law

```
   Int                 List[Int] -----> Option[Int]
    |                      |                 |
sqr |                      |                 |
    |                      |                 |
    v                      v                 v
   Int                 List[Int] -----> Option[Int]
```

Strong[Typed]
Scala Training and Consultancy

idNaturalTransformation

```
def id[F[+_]]: F -> F =
  new (F -> F) {
    override def apply[Z](fz: F[Z]): F[Z] =
      fz
  }
```

andThenNaturalTransformation

```
// ...
def andThen[L[+_]](‘h->l‘: H -> L) =
  new (F -> L) {
    def apply[Z](fz: F[Z]) =
      ‘h->l‘(‘f->h‘(fz))
  }
```

leftNaturalTransformation

```
def left[F[+_], G[+_]] =
  new (F -> ({ type L[+Z] = F[Z] ++ G[Z] })#L) {
    override def apply[Z](fz: F[Z]): F[Z] ++ G[Z] =
      Left(fz)
  }
```

rightNaturalTransformation

```
def right[F[+_], G[+_]] =
  new (G -> ({ type L[+Z] = F[Z] ++ G[Z] })#L) {
    override def apply[Z](gz: G[Z]): F[Z] ++ G[Z] =
      Right(gz)
  }
```

sumNaturalTransformationMethod

```
def ++[G[+_]](‘g->h‘: G -> H) =
  new (({ type L[+Z] = F[Z] ++ G[Z] })#L -> H) {
    override def apply[Z](‘fz++gz‘: F[Z] ++ G[Z]): H[Z] =
      ‘fz++gz‘ match {
        case Left(fz) => ‘f->h‘(fz)
        case Right(gz) => ‘g->h‘(gz)
      }
  }
```

Subtype

Subtype

```
type <=[F[+_], G[+_]] =
  F -> G

implicit def subReflect[F[+_]] =
  id[F]

implicit def subRight[F[+_], G[+_]] =
  right[F, G]

implicit def subTransLeft[F[+_], G[+_], H[+_]]
  (implicit `f<=g`: F <= G) =
  `f<=g` andThen[({ type L[+Z] = G[Z] ++ H[Z] })#L] left
```

Program

Res

```
trait Res[R[+_]] {
  val res: Id -> R
}

object Res {
  def apply[R[+_]: Res] = implicitly[Res[R]]
}
```

And

```
trait And[A[+_]] {
  def and[Z, Y]: A[Z] ** A[Y] => A[Z ** Y]
}

object And {
  def apply[A[+_]: And] = implicitly[And[A]]
}
```

Bnd(trait)

```scala
trait Bnd[B[+_]] {
  def bnd[Z, Y](z2by: Z => B[Y]): B[Z] => B[Y]
  def bnd_[Z, Y](by: B[Y]): B[Z] => B[Y] =
    bnd { (_: Z) =>
      by
    }
  val join: ({ type L[+Y] = B[B[Y]]})#L -> B =
    new (({ type L[+Y] = B[B[Y]]})#L -> B) {
      def apply[Y](bby: B[B[Y]]) =
        bnd { (by: B[Y]) =>
          by
        } (bby)
    }
}
```

Bnd(object)

```
object Bnd {
  def apply[B[+_]: Bnd] = implicitly[Bnd[B]]
}
```

Prg(trait)

```
trait Prg[P[+_]]
  extends End[P] with Res[P] with And[P] with Bnd[P] {
  override def end[Z, Y](z2y: Z => Y): P[Z] => P[Y] =
    bnd { (z: Z) => res(z2y(z)) }
  override def and[Z, Y]: P[Z] ** P[Y] => P[Z ** Y] = {
    case (pz, py) =>
      bnd { (z: Z) =>
        bnd { (y: Y) =>
          res((z, y))
        } (py)
      } (pz)
  }
  override def bnd[Z, Y](z2py: Z => P[Y]): P[Z] => P[Y] =
    pz => join(end(z2py)(pz))
  def end_[Z, Y](y: Y): P[Z] => P[Y] = bnd_(res(y))
}
```

Prg(object)

```
object Prg {
  def apply[P[+_]: Prg] = implicitly[Prg[P]]
}
```

Prgrm

```
class Prgrm[P[+_]: Prg, +Z] { pz: P[Z] =>
  def end[Y](z2y: Z => Y): P[Y] =
    End[P].end(z2y)(pz)
  def and[Y](py: P[Y]): P[Z ** Y] =
    And[P].and((pz, py))
  def bnd[Y](z2py: Z => P[Y]): P[Y] =
    Bnd[P].bnd(z2py)(pz)
  def bnd_[Y](py: P[Y]): P[Y] =
    Bnd[P].bnd_(py)(pz)
  def end_[Y](y: Y): P[Y] =
    Prg[P].end_(y)(pz)
}
```

ProgramSyntax

ProgramSyntax

```
class Program[Sntx[+_]: Res, +Z]
  extends Prgrm[({ type L[+Z] = Program[Sntx, Z] })#L, Z]
```

ProgramSyntaxSubClasses

```
case class ResProgram[Sntx[+_]: Res, +Z](sz: Sntx[Z])
  extends Program[Sntx, Z]

case class AndProgram[Sntx[+_]: Res, +Z, +Y]
  (psz_and_psy: (Program[Sntx, Z], Program[Sntx, Y]))
  extends Program[Sntx, Z ** Y]

case class BndProgram[Sntx[+_]: Res, +Z, ZZ <: Z, +Y]
  (psz: Program[Sntx, ZZ], z2psy: ZZ => Program[Sntx, Y])
  extends Program[Sntx, Y]
```

programSyntaxPrg

```
implicit def programSyntaxPrg[Sntx[+_]: Res] =
  new Prg[({ type L[+Z] = Program[Sntx, Z] })#L] {
    override val res:
      Id -> ({ type L[+Z] = Program[Sntx, Z] })#L =
      new (Id -> ({ type L[+Z] = Program[Sntx, Z] })#L) {
        def apply[Z](z: Id[Z]) =
          ResProgram(Res[Sntx].res(z))
      }
    override def and[Z, Y]:
      Program[Sntx, Z] ** Program[Sntx, Y] =>
      Program[Sntx, Z ** Y] =
       AndProgram(_)
    override def bnd[Z, Y]
     (z2psy: Z => Program[Sntx, Y]):
     Program[Sntx, Z] => Program[Sntx, Y] =
       BndProgram(_, z2psy)
  }
```

programSyntaxSemanticsDeclaration

```
def ‘programSyntax->semantics‘
  [Sntx[+_]: Res, Smntcs[+_]: Prg, Z]
  (‘syntax->semantics‘: Sntx -> Smntcs):
  ({ type L[+Z] = Program[Sntx, Z] })#L -> Smntcs =
```

programSyntaxSemanticsDefinition

```
new (({ type L[+Z] = Program[Sntx, Z] })#L -> Smntcs) {
  lazy val semantics =
    `programSyntax->semantics`(`syntax->semantics`)
  def apply[Z](psz: Program[Sntx, Z]) = psz match {
    case ResProgram(sz) =>
      `syntax->semantics`(sz)
    case AndProgram((psz, psy)) =>
      And[Smntcs].and {
        (semantics(psz), semantics(psy))
      }
    case BndProgram(psz: Program[Sntx, Z], z2psy) =>
      Bnd[Smntcs].bnd { (z: Z) =>
        semantics(z2psy(z))
      } (semantics(psz))
    }
}
```

subSyntaxToProgramSyntax

```
def ‘subSntx->programSyntax‘[SubSntx[+_], Sntx[+_]: Res]
  (‘subSntx<=sntx‘: SubSntx <= Sntx) =
  new (SubSntx -> ({ type L[+Z] = Program[Sntx, Z] })#L) {
    def apply[Z](subSntx: SubSntx[Z]) =
      ResProgram(‘subSntx<=sntx‘(subSntx))
  }
```

Program Resource

PrgRsc(trait)

```
trait PrgRsc[R[+_]] {
  def acquire[Z]: Unit => R[Z]
  def release[Z]: R[Z] => Unit
  def rscSafe[Z, Y](exec: R[Z] => Y): Try[Y] =
    `trying` {
      val rz = acquire[Z](())
      `try` {
        exec(rz)
      } `finally` {
        release(rz)
      }
    }
}
```

PrgRsc(object)

```
object PrgRsc {
  def apply[R[+_]: PrgRsc] = implicitly[PrgRsc[R]]
}
```

Executable

Exc

```
trait Exc[E[+_]] {
  type Result[+Z]
  type R[+Z]
  val prgRsc: PrgRsc[R]
  def exec[Z](ez: E[Z]): R[Z] => Result[Z]
  def rscSafeExec[Z](ez: E[Z]): Try[Result[Z]] =
    prgRsc.rscSafe(exec(ez))
}

object Exc {
  def apply[E[+_]: Exc] = implicitly[Exc[E]]
}
```

ExcPrg

```
trait ExcPrg[EP[+_]]
  extends Exc[EP]
  with Prg[EP] {
}

object ExcPrg {
  def apply[EP[+_]: ExcPrg] = implicitly[ExcPrg[EP]]
}
```

Callable

Callable

```scala
type Callable[-Z] = Z => Unit
```

Syntax

Syntax

```
trait Syntax[+Z] {
  def act: Z
  def rct: Callable[Z] => Unit =
    cz => cz(act)
}
```

Syntax
to
Semantics

syntaxToSemantics

```
def `syntax-res->semantics`[Smntcs[+_]: Prg] =
  new (Syntax -> Smntcs) {
    def apply[Z](syntax: Syntax[Z]) =
      Res[Smntcs].res(syntax.act)
  }
```

IdentitySyntax

IdentitySyntax

```
case class IdentitySyntax[+Z](z: Z)
  extends Syntax[Z] {
  def act = z
}

val identitySyntaxRes =
  new Res[IdentitySyntax] {
    val res: Id -> IdentitySyntax =
      new (Id -> IdentitySyntax) {
        def apply[Z](z: Id[Z]) =
          IdentitySyntax(z)
      }
  }
```

IdentitySyntax
to
Semantics

identitySyntaxToSemantics

```
def ‘identitySyntax-res->semantics‘[Smntcs[+_]: Prg]:
  IdentitySyntax -> Smntcs =
  ‘syntax-res->semantics‘[Smntcs]
    .asInstanceOf[IdentitySyntax -> Smntcs]
```

Identity Semantics

IdentitySemantics

```
case class IdentitySemantics[+Z](value: Z)
```

Println Program Details

PrintWriterPrgResource

```
case class PrintWriterPrgResource[+Z](pw: PrintWriter)
```

printWriterPrgRsc

```
object printWriterPrgRsc
  extends PrgRsc[PrintWriterPrgResource] {
  def acquire[Z]: Unit => PrintWriterPrgResource[Z] = {
    case () =>
      PrintWriterPrgResource(System.console.writer())
  }
  def release[Z]: PrintWriterPrgResource[Z] => Unit = {
    case PrintWriterPrgResource(pw) =>
      pw.close()
  }
}
```

PrintlnSyntax

```
case class PrintlnSyntax[+Z](string: String)
  extends Syntax[Try[Unit]] {
  def act =
    printWriterPrgRsc.rscSafe {
      (rsc: PrintWriterPrgResource[Nothing]) =>
        rsc.pw.println(string)
        rsc.pw.flush()
    }
}
```

println

```
def println[Sntx[+_]: Res](string: String)
  (implicit `printlnSyntax<=sntx`: PrintlnSyntax <= Sntx):
  Program[Sntx, Try[Unit]] =
  `subSntx->programSyntax`(`printlnSyntax<=sntx`)
    .apply(PrintlnSyntax(string))
```

printlnSyntaxToSemantics

```
def `printlnSyntax-res->semantics`[Smntcs[+_]: Prg]:
  PrintlnSyntax -> Smntcs =
  `syntax-res->semantics`[Smntcs]
    .asInstanceOf[PrintlnSyntax -> Smntcs]
```

PrintlnEffectSyntax

```
type EffectSyntax[+Z] =
  IdentitySyntax[Z] ++
  PrintlnSyntax[Z]
```

PrintlnSyntaxToSemantics

```
val ‘effectSyntax->semantics‘:
  EffectSyntax -> IdentitySemantics =
  ‘identitySyntax-res->semantics‘ ++
    ‘printlnSyntax-res->semantics‘
```

PrintlnEffectSyntaxRes

```
implicit val effectSyntaxRes: Res[EffectSyntax] =
  new Res[EffectSyntax] {
    val res: Id -> EffectSyntax =
      new (Id -> EffectSyntax) {
        def apply[Z](z: Id[Z]) =
          subTransLeft
            [IdentitySyntax, IdentitySyntax, PrintlnSyntax]
              .apply(identitySyntaxRes.res(z))
      }
  }
```

ConcretePrintInProgram

```
object ConcreteProgram
  extends AbstractProgram[EffectSyntax] {

  val meaning =
    `programSyntax->semantics`(`effectSyntax->semantics`)

  def resourceSafeExec[Z] =
    (semantics: IdentitySemantics[Try[Z]]) =>
      identitySemanticsPrg.rscSafeExec(semantics)
```

Active Socket

activeSocketProgramThrowSyntaxDefinition

```
res("ubuntu-laptop") bnd { socketHost =>
  print("read port: ") bnd_ {
    readln() bnd { readPortString =>
      val readPort = parseInt(readPortString)
      print("print port: ") bnd_ {
        readln() bnd { printPortString =>
          val printPort = parseInt(printPortString)
          (asyncPrintln("reading") and
            socketReadln(socketHost, readPort) and
            socketReadln(socketHost, readPort)) bnd {
              case ((_, how), who) =>
                val greeting = how + who
                asyncPrintln("printing") and
                  socketPrintln(greeting, socketHost, printPort) and
                  socketPrintln(greeting, socketHost, printPort)
            } end_ (())
          } } } } }
```

activeSocketProgramThrowSyntaxDeclaration

```
def socketProgramThrowSyntax
  (implicit `identitySyntax<=sntx`: IdentitySyntax <= Sntx,
  `printSyntax<=sntx`: PrintSyntax <= Sntx,
  `readlnSyntax<=sntx`: ReadlnSyntax <= Sntx,
  `asyncPrintlnSyntax<=sntx`: AsyncPrintlnSyntax <= Sntx,
  `socketReadLineSyntax<=sntx`: SocketReadlnSyntax <= Sntx,
  `socketPrintlnSyntax<=sntx`: SocketPrintlnSyntax <= Sntx):
  ProgramThrowSyntax[Unit] = {
```

ProgramThrow

Thr

```
trait Thr[T[+_]] {
  def ‘try‘[Z](tz: T[Z]): T[Try[Z]]
  def ‘throw‘[Z](t: Throwable): T[Z]
}

object Thr {
  def apply[T[+_]: Thr] = implicitly[Thr[T]]
}
```

PrgThr

```
trait PrgThr[PT[+_]]
  extends Prg[PT]
  with Thr[PT] {
}

object PrgThr {
  def apply[PT[+_]: PrgThr] = implicitly[PrgThr[PT]]
}
```

PrgrmThrw

```
class PrgrmThrw[PT[+_]: PrgThr, +Z]
  extends Prgrm[PT, Z] { ptz: PT[Z] =>
  def `try`: PT[Try[Z]] =
    Thr[PT].`try`(ptz)
}
```

ThrowTrans

ThrowTrans

```
class ThrowTrans[P[+_]: Prg, +Z](val get: P[Try[Z]])
  extends PrgrmThrw[({ type L[+Z] = ThrowTrans[P, Z] })#L, Z] {
```

throwTransPrgThr

```
implicit def throwTransPrgThr[P[+_]: Prg] =
  new PrgThr[({ type L[+Z] = ThrowTrans[P, Z] })#L] {
```

throwTransRes

```
override val res:
  Id -> ({ type L[+Z] = ThrowTrans[P, Z] })#L =
  new (Id -> ({ type L[+Z] = ThrowTrans[P, Z] })#L) {
    def apply[Z](z: Id[Z]) =
      new ThrowTrans(Res[P].res(‘trying‘(z)))
  }
```

throwTransAnd

```
override def and[Z, Y]:
  (ThrowTrans[P, Z] ** ThrowTrans[P, Y]) =>
    ThrowTrans[P, Z ** Y] = {
    case (ttz, tty) =>
      new ThrowTrans(End[P].end {
        (tz_and_ty: Try[Z] ** Try[Y]) => tz_and_ty match {
        case (Success(z), Success(y)) => success((z, y))
        case (Success(z), Failure(e)) => failure(e)
        case (Failure(e), _) => failure(e)
  } } (And[P].and((ttz.get, tty.get)))) }
```

throwTransBnd

```
override def bnd[Z, Y](z2tty: Z => ThrowTrans[P, Y]):
ThrowTrans[P, Z] => ThrowTrans[P, Y] =
  ttz => new ThrowTrans(Prg[P].bnd {
    (tz: Try[Z]) => tz match {
      case Success(z) => z2tty(z).get
      case Failure(t) => Res[P].res(failure(t))
    } } (ttz.get))
```

throwTransTry

```
def ‘try‘[Z](ttz: ThrowTrans[P, Z]):
  ThrowTrans[P, Try[Z]] =
  new ThrowTrans[P, Try[Z]]({
    End[P].end {
      (tz: Try[Z]) => tz match {
      case Success(z) =>
        success(success(z))
      case Failure(t) =>
        success(failure(t))
    } } (ttz.get)
  })
```

throwTransThrow

```
  def `throw`[Z](t: Throwable): ThrowTrans[P, Z] =
    new ThrowTrans(Res[P].res(failure(t)))
}
```

ProgramThrowSyntax

ProgramThrow

```
type ProgramThrow[Sntx[+_], +Z] =
  ThrowTrans[({ type L[+Z] = Program[Sntx, Z] })#L, Z]
```

ActiveSocketEffectSyntax

```
type Syntax01[+Z] = IdentitySyntax[Z] ++ PrintSyntax[Z]
type Syntax02[+Z] = Syntax01[Z] ++ ReadlnSyntax[Z]
type Syntax03[+Z] = Syntax02[Z] ++ SocketReadlnSyntax[Z]
type Syntax04[+Z] = Syntax03[Z] ++ AsyncPrintlnSyntax[Z]
type EffectSyntax[+Z] = Syntax04[Z] ++ SocketPrintlnSyntax[Z]
```

ActiveSocketEffectSyntaxToActiveFutureSemantics

```
val `effectSyntax->actFtrPrgSemantics`
  : EffectSyntax -> ActiveFutureSemantics =
  `identitySyntax-now->actFtrPrgSemantics` ++
    `printSyntax-now->actFtrPrgSemantics` ++
    `readlnSyntax-now->actFtrPrgSemantics` ++
    `socketReadlnSyntax-activeFuture->actFtrPrgSemantics` ++
    `asyncPrintlnSyntax-activeFuture->actFtrPrgSemantics` ++
    `socketPrintlnSyntax-activeFuture->actFtrPrgSemantics`
```

Reactive Socket

reactiveSocketProgramThrowSyntaxFragment

```
def socketProgramThrowSyntaxFragment
  (socketHost: SocketHost, readPort: SocketPort)
  (implicit ‘identitySyntax<=sntx‘: IdentitySyntax <= Sntx,
  ‘socketAsyncReadln<=sntx‘: SocketAsyncReadlnSyntax <= Sntx):
  ProgramThrowSyntax[String ** String] =
    socketAsyncReadln(socketHost, readPort) and
      socketAsyncReadln(socketHost, readPort)
```

ReactiveSocketEffectSyntax

```
type EffectSyntax[+Z] =
  IdentitySyntax[Z] ++
    SocketAsyncReadlnSyntax[Z]
```

ReactiveSocketEffectSyntaxToActiveFutureSemantics

```
val ‘effectSyntax->rctFtrPrgSemantics‘:
  EffectSyntax -> ReactiveFutureSemantics =
  ‘identitySyntax-now->rctFtrPrgSemantics‘ ++
    ‘socketAsyncReadlnSyntax-reactiveFuture->rctFtrPrgSemantics‘
```

reactiveSocketExpressionFragment

```
def socketExpressionFragment
  (socketHost: SocketHost, readPort: SocketPort) =
  resourceSafeExec(
    socketProgramThrowSyntaxFragment(socketHost, readPort))
```

reactiveSocketProgramThrowSyntaxDefinition

```
res("ubuntu-laptop") bnd { socketHost =>
 print("read port: ") bnd_ {
  readln() bnd { readPortString =>
   val readPort = parseInt(readPortString)
   print("print port: ") bnd_ {
    readln() bnd { printPortString =>
     val printPort = parseInt(printPortString)
     res(socketExpressionFragment(socketHost, readPort)) bnd {
      case Success(Success((how, who))) =>
       val greeting = how ++ who
        socketPrintln(greeting, socketHost, printPort) and
         socketPrintln(greeting, socketHost, printPort)
      case Success(Failure(t)) => res(Right(Left(t)))
      case Failure(t) => res(Left(t))
     } end_ (())
    } } } } }
```

ProcessSyntax

Process(trait)

```
trait Process[F[+_], +O]
  extends Prgrm[({ type L[+O] = Process[F, O] })#L, O] {
```

ProcessSubClasses

```
case class Await[F[+_], I, +O](
  fi: F[I],
  ti2po: Try[I] => Process[F, O])
  extends Process[F, O]

case class Emit[F[+_], +O](
  o: O,
  po: Process[F, O])
  extends Process[F, O]

case class Halt[F[+_], O](t: Throwable)
  extends Process[F, O]
```

ProcessExceptions

```
case object Finished extends Exception

case object Killed extends Exception
```

File Source Process

linesProcessSyntaxDefinition

```
rscSafe
  [ProgramThrowSyntax,
    SourcePrcResource, Nothing, String] { source =>
  iteratorNext(source.getLines()).toProcess.bind {
    case Some(line) => line
  } `while` {
    case Some(_) => true
    case None => false
  }
}
```


Strong[Typed]
Scala Training and Consultancy

linesProgramThrowSyntax

```
val linesProgramThrowSyntax:
ProgramThrow[EffectSyntax, Unit] =
  linesProcessSyntax("/tmp/helloLambdaDays")
    .runLog(
      PrgThr[({ type L[+Z] = ProgramThrow[EffectSyntax, Z] })#L])
    .bnd { lines =>
      println(lines.toString())
    }
```

ProcessSyntax Continued

toProcess(ProcessSyntax)

```
def toProcess[F[+_], I](fi: F[I]): Process[F, I] = {
  await(fi) {
    case Success(i) =>
      emit(i)
    case Failure(t) =>
      halt(t)
  }
}
```

toProcess(ProgramThrowSyntax)

```
def toProcess:
  Process[({ type L[+Z] = ThrowTrans[P, Z] })#L, Z] =
  Process.toProcess[
    ({ type L[+Z] = ThrowTrans[P, Z] })#L, Z](this)
```

Resource Continued

PrcRsc

```
trait PrcRsc[F[+_], R[+_]] {
  def prcAcquire[Z]: Unit => F[R[Z]]
  def prcRelease[Z]: F[R[Z] => Unit]
}

object PrcRsc {
  def apply[F[+_], R[+_]: ({ type L[R[+_]] = PrcRsc[F, R] })#L] =
    implicitly[PrcRsc[F, R]]
}
```

Zipped File Sources Process

zippedLinesProcessSyntaxDefinition

```
rscSafe[ProgramThrowSyntax,
  SourcesPrcResource, Nothing, String ** String] {
  case (source1, source2) =>
    (iteratorNext(source1.getLines()) and
      iteratorNext(source2.getLines())).toProcess
      .bind {
        case (Some(line1), Some(line2)) => (line1, line2)
      } `while` {
        case (Some(_), Some(_)) => true
        case _ => false
      }
}
```

Sinked Zipped File Sources Process

filePrintWriterSink

```
val filePrintWriterSink:
  Sink[ProgramThrowSyntax, String] =
  rscSafe[ProgramThrowSyntax,
    FilePrintWriterPrgResource,
    Nothing,
    String =>
      Process[ProgramThrowSyntax, Unit]] { printWriter =>
      toSink[ProgramThrowSyntax, String] { string =>
        filePrintWriterPrintln[Sntx](printWriter, string)
      }
    }
```

sinkedLinesZippedWithZipperProcessSyntax

```
linesZippedWithZipperProcessSyntax
  .filter { (line: String) => !line.contains("5") }
  .through(filePrintWriterSink)
  .prcDrain
```

Channel

Channel

```
type Channel[F[+_], -I, +O] = Process[F, I => Process[F, O]]

type Source[F[+_], +O] = Channel[F, Unit, O]

type Sink[F[+_], -I] = Channel[F, I, Unit]
```

toChannel

```
def toChannel[F[+_]: Res, Z, Y](z2fy: Z => F[Y]):
  Channel[F, Z, Y] =
  constantly { (z: Z) =>
    toProcess(z2fy(z))
  }

def toSource[F[+_]: Res, Y](u2fy: Unit => F[Y]):
  Source[F, Y] =
  toChannel[F, Unit, Y](u2fy)

def toSink[F[+_]: Res, Z](z2fu: Z => F[Unit]):
  Sink[F, Z] =
  toChannel[F, Z, Unit](z2fu)
```

constantly

```
def constantly[F[+_]: Res, O](o: O): Process[F, O] =
  toProcess(Res[F].res(o)) bnd {
    o =>
      emit(o, constantly(o))
  }
```

Hello LambdaDays

Goodbye LambdaDays