Typed Interaction with Session Types

(using Scala and Libretto)

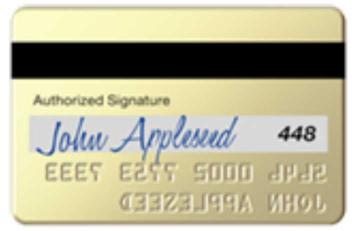
Tomas Mikula

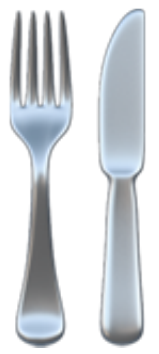
Functional Scala 2022

Model Problem: Canteen

Rules

- Customer proceeds through sections, *in this order:*
  1. soups
  2. main dishes
  3. payment
- Can get any number of items
  - as long as supply lasts, otherwise proceed to the next section
- Eats all of the purchased food
  - in any order, possibly even before paying





First Sketch

Approach I: Naive Objects and Methods

The Interface

```scala
trait Canteen:
  def enter(): Session

trait Session:
  def getSoup(): Option[Soup]

  def getMainDish(): Option[MainDish]

  def payAndClose(card: PaymentCard): Unit
```

```scala
class Soup:
  def eat(): Unit

class MainDish:
  def eat(): Unit
```

Approach I: Naive Objects and Methods

Customer

```
def customer(canteen: Canteen, card: PaymentCard): Unit =
  val session = canteen.enter()

  val soup = session.getSoup()
  val dish = session.getMainDish()

  session.payAndClose(card)

  soup.foreach(_.eat())
  dish.foreach(_.eat())
```

Approach I: Naive Objects and Methods

What Could Possibly Go Wrong (1)

```
def customer(canteen: Canteen, card: PaymentCard): Unit =
  val session = canteen.enter()

  val dish = session.getMainDish()

  val soup = session.getSoup()
  …
```

❌ **Illegal to move from main dish back to soup**

Approach I: Naive Objects and Methods

What Could Possibly Go Wrong (2)

```
def customer(canteen: Canteen, card: PaymentCard): Unit =
  val session = canteen.enter()

  val soup = session.getSoup()

  session.payAndClose(card)

  val dish = session.getMainDish()
  …
```

❌ **Illegal to get more items after paying**

Approach I: Naive Objects and Methods

What Could Possibly Go Wrong (3)

```
def customer(canteen: Canteen, card: PaymentCard): Unit =
  val session = canteen.enter()

  val soup = session.getSoup()

  soup.foreach(_.eat())
```

❌ **Illegal to leave without paying**

Approach I: Naive Objects and Methods

What Could Possibly Go Wrong (4)

```
def customer(canteen: Canteen, card: PaymentCard): Unit =
  val session = canteen.enter()

  val soup = session.getSoup()

  val dish = session.getMainDish()

  session.payAndClose(card)

  dish.foreach(_.eat())
```

❌ **Illegal to waste food (did not eat the soup)**

Approach I: Naive Objects and Methods

What Could Possibly Go Wrong (5)

```
def customer(canteen: Canteen, card: PaymentCard): Unit =
  val session = canteen.enter()

  val soup = session.getSoup()

  session.payAndClose(card)

  soup.foreach(_.eat())
  soup.foreach(_.eat())
```

❌ **Illegal to eat the same item twice**

Approach I: Naive Objects and Methods

What Could Possibly Go Wrong (6)

```
def customer(canteen: Canteen, card: PaymentCard): Unit =
  val session = canteen.enter()

  val soup1: Option[Soup] = session.getSoup() // None (ran out)

  val soup2: Option[Soup] = session.getSoup()
  …
```

❌ **Illegal to *repeatedly* ask for a meal that ran out**

Approach I: Naive Objects and Methods

Canteen Implementation

```
class SessionImpl extends Session:

  enum State:
    case SectionSoup
    case SectionMain
    case SectionPayment
    case Closed

  private var state: State = SectionSoup

  // …
```

Approach I: Naive Objects and Methods

Canteen Implementation: Handling Illegal State (1)

```
override def getSoup(): Option[Soup] =
  this.state match
    case SectionSoup =>
      // …
    case SectionMain | SectionPayment | Closed =>
      throw IllegalStateException()
```

Approach I: Naive Objects and Methods

Canteen Implementation: Handling Illegal State (2)

```
override def getMainDish(): Option[MainDish] =
  this.state match
    case SectionSoup =>
      // …
    case SectionMain =>
      // …
    case SectionPayment | Closed =>
      throw IllegalStateException()
```

Approach I: Naive Objects and Methods

Canteen Implementation: Handling Illegal State (3)

```
override def payAndClose(card: PaymentCard): Unit =
  this.state match
    case SectionPayment | SectionMain | SectionSoup =>
      // …
    case Closed =>
      throw IllegalStateException()
```

Approach I: Naive Objects and Methods

Summary

- canteen handling **illegal state**
- customer getting **runtime errors** and/or **resource leaks**

Moreover

- **bad discoverability** of the correct protocol (relying on documentation)
- **fragile** w.r.t. refactoring or changes in the protocol

Let’s take a step up!

Approach II: Linearity by Convention

The Idea

- use types specific to the stages of interaction (`SectionSoup`, `SectionMain`, …)
  - each having only methods that are legal at that stage
- a method on one stage returns the next stage
- use each object *exactly once* (**linearity**)
  - only a convention
  - but adherence to it can be checked *locally*

Approach II: Linearity by Convention

The Interface

```scala
trait Session:
  def proceedToSoups(): SectionSoup

trait SectionSoup:
  def getSoup(): Either[(Soup, SectionSoup), SectionMain]
  def proceedToMainDishes(): SectionMain

trait SectionMain:
  def getMainDish(): Either[(MainDish, SectionMain), SectionPayment]
  def proceedToPayment(): SectionPayment

trait SectionPayment:
  def payAndClose(card: PaymentCard): Unit
```

Approach II: Linearity by Convention

Customer

```
def customer(session: Session, card: PaymentCard): Unit =
  val sectionSoup          = session.proceedToSoups()
  val (soup, sectionMain) = tryGetSoupAndProceed(sectionSoup)
  val (dish, sectionPay)  = tryGetDishAndProceed(sectionMain)

  sectionPay.payAndClose(card)

  soup.foreach(_.eat())
  dish.foreach(_.eat())
```

Each variable used exactly once.
Linearity ensures adherence to protocol.

Approach II: Linearity by Convention

Canteen Implementation

```
class SectionSoupImpl extends SectionSoup:

  def getSoup(): Either[(Soup, SectionSoup), SectionMain] =
    // …

  def proceedToMainDishes(): SectionMain =
    // …
```

No handling of illegal state.

Trusting the client to uphold linearity.

Approach II: Linearity by Convention

Summary

- handling illegal state avoided (*)
- no runtime errors or leaks (*)
- type driven: the types+convention guide us towards a correct implementation
  - single rule of *linearity* supersedes all the protocol-specific rules
- more robust w.r.t. refactoring or changes in the protocol (*)
- unclear what may be used non-linearly

(*) provided *everyone* upholds linearity

- one defector ruins everything

Linearity Helps

Can we enforce it *before* execution?

Linearity Helps

Can we enforce it *before* execution?

Meet Libretto!

Libretto: The Idea

- Programs as data structures

`p: A -o B`

- **Linear by construction** (non-linear programs *unrepresentable*)
- types `A`, `B` define the *interface* of `p` (*protocol of interaction* with its surroundings)
- Executed by an interpreter
- `IO[A]` ~ Free Monad with extra operations

  `A -o B` ~ Free Category with extra operations

  (closed symmetric bimonoidal, traced, distributive, …, *not* cartesian)

Approach III: Libretto

Canteen: Customer

```
def customer: (Session |*| PaymentCard) -o PaymentCard =
  λ { case (session |*| card) =>
    val soupSection = Session.enter(session)

    val (soup |*| mainSection) = tryGetSoupAndProceed(soupSection)
    val (dish |*| paySection)  = tryGetDishAndProceed(mainSection)

    paySection(card)
      .waitFor(
        joinAll(
          soup .map(eatSoup(_))     .getOrElse(done),
          dish .map(eatMainDish(_)) .getOrElse(done),
        )
      )
  }
```

Approach III: Libretto

Canteen: Customer

```
def customer: (Session |*| PaymentCard) -o PaymentCard =
  λ { case (session |*| card) =>
    val soupSection = Session.enter(session)

    val (soup |*| mainSection) = tryGetSoupAndProceed(soupSection)
    val (dish |*| paySection)  = tryGetDishAndProceed(mainSection)

    paySection(card)
      .waitFor(
        joinAll(
          soup .map(eatSoup(_))      .getOrElse(done),
          dish .map(eatMainDish(_)) .getOrElse(done),
        )
      )
  }
```

Approach III: Libretto

Canteen: Customer

pair

```
def customer: (Session |*| PaymentCard) -o PaymentCard =
  λ { case (session |*| card) =>
    val soupSection = Session.enter(session)

    val (soup |*| mainSection) = tryGetSoupAndProceed(soupSection)
    val (dish |*| paySection)  = tryGetDishAndProceed(mainSection)

    paySection(card)
      .waitFor(
        joinAll(
          soup .map(eatSoup(_))     .getOrElse(done),
          dish .map(eatMainDish(_)) .getOrElse(done),
        )
      )
  }
```

Approach III: Libretto

Canteen: Customer

```
def customer: (Session |*| PaymentCard) -○ PaymentCard =
 λ { case (session |*| card) =>
    val soupSection = Session.enter(session)

    val (soup |*| mainSection) = tryGetSoupAndProceed(soupSection)
    val (dish |*| paySection)  = tryGetDishAndProceed(mainSection)

    paySection(card)
      .waitFor(
        joinAll(
          soup .map(eatSoup(_))     .getOrElse(done),
          dish .map(eatMainDish(_)) .getOrElse(done),
        )
      )
  }
```

Approach III: Libretto

Canteen: Customer

```
def customer: (Session |*| PaymentCard) -○ PaymentCard =
  λ { case (session |*| card) =>
    val soupSection = Session.enter(session)

    val (soup |*| mainSection) = tryGetSoupAndProceed(soupSection)
    val (dish |*| paySection)  = tryGetDishAndProceed(mainSection)

    paySection(card)
      .waitFor(
        joinAll(
          soup .map(eatSoup(_))     .getOrElse(done),
          dish .map(eatMainDish(_)) .getOrElse(done),
        )
      )
  }
```

Approach III: Libretto

Canteen: Customer

```
def customer: (Session |*| PaymentCard) -o PaymentCard =
  λ { case (session |*| card) =>
    val soupSection = Session.enter(session)

    val (soup |*| mainSection) = tryGetSoupAndProceed(soupSection)
    val (dish |*| paySection)  = tryGetDishAndProceed(mainSection)

    paySection(card)
      .waitFor(
        joinAll(
          soup .map(eatSoup(_))      .getOrElse(done),
          dish .map(eatMainDish(_)) .getOrElse(done),
        )
      )
  } throws LinearityViolation at assembly time
```

Approach III: Libretto

Canteen: Customer

Let’s break it!

Approach III: Libretto

Protocol Violation (1)

```
def customer: (Session |*| PaymentCard) -o PaymentCard =
  λ { case (session |*| card) =>
    val soupSection = Session.enter(session)
    val mainSection = SectionSoup.proceedToMainDishes(soupSection)

    val (dish |*| paySection) = tryGetMainDishAndProceed(mainSection)
    val (soup |*| _)          = tryGetSoupAndProceed(soupSection)

    paySection(card)
      .waitFor(
        joinAll(
          soup .map(eatSoup(_))     .getOrElse(done),
          dish .map(eatMainDish(_)) .getOrElse(done),
        )
      )
  }
```

Wrong Order
becomes
Linearity Violation
becomes
Assembly-time error

Approach III: Libretto

Protocol Violation (2)

```
def customer: (Session |*| PaymentCard) -○ PaymentCard =
  λ { case (session |*| card) =>
    val soupSection = Session.enter(session)

    val (soup |*| mainSection) = tryGetSoupAndProceed(soupSection)
    val (dish |*| paySection)  = tryGetDishAndProceed(mainSection)

    // paySection(card)
    card
      .waitFor(
        joinAll(
          soup .map(eatSoup(_))     .getOrElse(done),
          dish .map(eatMainDish(_)) .getOrElse(done),
        )
      )
  }
```

Not Paying
becomes
Linearity Violation
becomes
Assembly-time error

Approach III: Libretto

Protocol Violation (3)

```
def customer: (Session |*| PaymentCard) -o PaymentCard =
  λ { case (session |*| card) =>
    val soupSection = Session.enter(session)

    val (soup |*| mainSection) = tryGetSoupAndProceed(soupSection)
    val (dish |*| paySection)  = tryGetDishAndProceed(mainSection)

    paySection(card)
      .waitFor(
        joinAll(
       // soup .map(eatSoup(_))      .getOrElse(done),
          dish .map(eatMainDish(_)) .getOrElse(done),
        )
      )
  }
```

Wasting Food
becomes
Linearity Violation
becomes
Assembly-time error

Approach III: Libretto

Protocol Violation (4)

```
def customer: (Session |*| PaymentCard) -o PaymentCard =
  λ { case (session |*| card) =>
    val soupSection = Session.enter(session)

    val (soup |*| mainSection) = tryGetSoupAndProceed(soupSection)
    val (dish |*| paySection)  = tryGetDishAndProceed(mainSection)

    paySection(card)
      .waitFor(
        joinAll(
          soup .map(eatSoup(_))     .getOrElse(done),
          soup .map(eatSoup(_))     .getOrElse(done),
          dish .map(eatMainDish(_)) .getOrElse(done),
        )
      )
  }
```

Double-spending
becomes
Linearity Violation
becomes
Assembly-time error

Approach III: Libretto

Catching Linearity Violations

```
test("customer") {

  customer : (Session |*| PaymentCard) -o PaymentCard

}
```

Code is cheap.

Show me the types!

Libretto: Types of Interaction

Session Types in Libretto

$$A \multimap B$$

(in-port) (out-port)

`Done`	signal traveling left-to-right
`Val[A]`	Scala value of type `A`
`A ⊗ B`	(concurrent) pair (`|*|` in code)
`A ⊕ B`	producer choice (`|+|` in code)
`A & B`	consumer choice (`|&|` in code)
`A =o B`	function object
`Rec[F[_]]`	recursive type former

Libretto: Types of Interaction

Inversion

`-[A]`		inverts the data-flow through `A`
`-[Done]`		signal traveling *right-to-left*
`-[Val[A]]`		Scala value traveling *right-to-left*
`-[A ⊗ B]`	≃	`-[A] ⊗ -[B]`
`-[A ⊕ B]`	≃	`-[A] & -[B]`
`-[A & B]`	≃	`-[A] ⊕ -[B]`

Approach III: Libretto

Canteen: Protocol

```
type SectionSoup = Rec[ [SectionSoup] =>>
  |&| [
    (Soup |*| SectionSoup) |+| SectionMainDish,
    SectionMain,
  ]
]
```

Approach III: Libretto

Canteen: Protocol

```
type SectionSoup = Rec[ [SectionSoup] =>>
  |&| [
    (Soup |*| SectionSoup) |+| SectionMain,
    SectionMain,
  ]
]
```

consumer choice:

- get soup
- go to main section

Approach III: Libretto

Canteen: Protocol

```
type SectionSoup = Rec[ [SectionSoup] =>>
  |&| [
    (Soup |*| SectionSoup) |+| SectionMain,
    SectionMain,
  ]
]
```

consumer choice:

- get soup
- go to main section

producer choice:

here's a soup, want another?

out of soup, proceed

Approach III: Libretto

Canteen: Protocol

```
type SectionSoup = Rec[ [SectionSoup] =>>
  |&| [
    (Soup |*| SectionSoup) |+| SectionMain,
    SectionMain,
  ]
]
```

consumer choice:

- get soup
- go to main section

producer choice:

here's a soup, want another? — out of soup, proceed

```
// helper functions to make a choice

def getSoup: SectionSoup -o ((Soup |*| SectionSoup) |+| SectionMain) =
  unpack > chooseL

def proceedToMainDishes: SectionSoup -o SectionMain =
  unpack > chooseR
```

Approach III: Libretto

Canteen: Protocol

```
type SectionSoup = Rec[ [SectionSoup] =>>
  |&| [
    (Soup |*| SectionSoup) |+| SectionMain,
    SectionMain,
  ]
]
```

consumer choice:

- get soup
- go to main section

producer choice:

here's a soup, want another? — out of soup, proceed

```
// factory method to create SectionSoup from A
def from[A](
  onSoupRequest : A -o ((Soup |*| SectionSoup) |+| SectionMain),
  goToMainDishes: A -o SectionMain,
): A -o SectionSoup =
  choice(onSoupRequest, goToMainDishes) > pack
```

Approach III: Libretto

Canteen: Protocol

```
type SectionMain = Rec[ [SectionMain] =>>
  |&| [
    (MainDish |*| SectionMain) |+| SectionPayment,
    SectionPayment,
  ]
]

type SectionPayment =
  PaymentCard =○ PaymentCard
```

Approach III: Libretto

Canteen: Protocol

```
opaque type Session = SectionSoup

object Session:
  def proceedToSoups: Session -o SectionSoup =
    id

  def create: SectionSoup -o Session =
    id
```

Approach III: Libretto

Canteen: Provider

```
def provider: Done -o Session =
  soupSection > Session.create

def soupSection: Done -o SectionSoup =
  rec { soupSection =>
    SectionSoup.from(
      onSoupRequest =
        λ.+ { done =>
          injectL( makeSoup(done) |*| soupSection(done) )
        },
      goToMainDishes =
        mainSection,
    )
  }
```

No handling of illegal state

Approach III: Libretto

Canteen: Provider

```
def provider: Done -o Session =
  soupSection > Session.create

def soupSection: Done -o SectionSoup =
  rec { soupSection =>
    SectionSoup.from(
      onSoupRequest =
        λ.+ { done =>
          injectL( makeSoup(done) |*| soupSection(done) )
        } (using Cosemigroup[Done]),
      goToMainDishes =
        mainSection,
    )
  }
```

No handling of illegal state

Approach III: Libretto

Canteen: Provider

```
def mainSection: Done -○ SectionMainDish =
  rec { mainSection =>
    SectionMainDish.from(
      onDishRequest =
        λ.+ { done =>
          injectL( makeMainDish(done) |*| mainSection(done) )
        },
      goToPayment =
        paymentSection,
    )
  }
```

No handling of illegal state

Approach III: Libretto

Canteen: Provider

```
def paymentSection: Done -o SectionPayment =
  λ { done =>
    λ.closure { card =>
      card.waitFor(done)
    }
  }
```

No handling of illegal state

Approach III: Libretto

Canteen: Provider

```
                 Done -o (PaymentCard =o PaymentCard)
def paymentSection: Done -o SectionPayment =
  λ { done =>
    λ.closure { card =>
      card.waitFor(done)
    }
  }
```

No handling of illegal state

Approach III: Libretto

Canteen: Putting It All Together

```
object Main extends StarterApp:

  override def blueprint: Done -○ Done =
    λ.+ { started =>
      val cardIn  = started > PaymentCard.issue

      val session = provider(started)
      val cardOut = customer(session |*| cardIn)

      PaymentCard.shred(cardOut)
    }
```

Approach III: Libretto

Summary

- handling illegal state avoided
- runtime errors and resource leaks prevented
- type driven: the types guide us towards a correct implementation
  - protocol expressed by types
- robust w.r.t. refactoring or changes in the protocol
- no confusion about what's linear: non-linearity witnessed by a typeclass

There's More in Libretto

- seamless **concurrency**
  - not built on effects
- **streams** expressible using types we have already seen

```
type Stream[A] = Rec[[Self] =>> Done |&| (Done |+| (Val[A] |*| Self))]
```

  - custom combinators need not fall back to effects
- **effects**
- **resource safety**
  - linearity avoids the complexities of managing scopes
- **programs as values** without opaque Scala functions inside
  - whole new world of possibilities

👊 Give it a try

https://github.com/TomasMikula/libretto/

👉