Improving Correctness with Types

Scala Days March 2015
Defensive programming
Fail fast
Design by contract
“Bad programmers worry about the code.

Good programmers worry about data structures and their relationships.”

- Linus Torvalds
What is a function?

Domain \( \xrightarrow{f} \) Range

\( x \)

\( y \)
Never use nulls ... period

“Null references, my billion dollar mistake”
- Tony Hoare, QCon 2009
Almost all data is immutable
Do not throw exceptions
Wrapper Types
case class Customer(name: String, 
  preferredCurrency: String) {
  require(Currency.isValid(preferredCurrency))
}

val customer = Customer(name = "Joe Bloggs", 
  preferredCurrency = "SFO")

This is an airport?
class Currency private (val code: String) extends AnyVal

object Currency {
  val USD: Currency = new Currency("USD")
  val EUR: Currency = new Currency("EUR")

  // ...

  def from(code: String): Option[Currency] = ???
}

case class Customer(name: String,
                     preferredCurrency: Currency)

def creditCardRate(customer: Customer): BigDecimal = {
  if (customer.preferredCurrency == "USD")
    BigDecimal("0.015")
  else
    BigDecimal("0.025")
}
import org.scalactic.TypeCheckedTripleEquals._

case class Customer(name: String,
                      preferredCurrency: Currency)

def creditCardRate(customer: Customer): BigDecimal = {
  if (customer.preferredCurrency === "USD")
    BigDecimal("0.015")
  else
    BigDecimal("0.025")
}
import org.scalactic.TypeCheckedTripleEquals

case class Customer(name: String,
                      preferredCurrency: Currency)

def creditCardRate(customer: Customer): BigDecimal = {
    if (customer.preferredCurrency === Currency.USD)
        BigDecimal("0.015")
    else
        BigDecimal("0.025")
}
val order: Order = ???
val customer: Customer = ???

val creditCardCharge = order.amount + creditCardRate(customer)

Eeek this a bug
class MoneyAmount(val amount: BigDecimal) extends AnyVal {
  def + (rhs: MoneyAmount): MoneyAmount =
    new MoneyAmount(amount + rhs.amount)

  def - (rhs: MoneyAmount): MoneyAmount =
    new MoneyAmount(amount - rhs.amount)

  def * (rhs: Rate): MoneyAmount =
    new MoneyAmount(amount * rhs.size)
}

class Rate(val size: BigDecimal) extends AnyVal {
  def * (rhs: Rate): MoneyAmount = rhs * this
}
val order: Order = ???
val customer: Customer = ???

val creditCardCharge = order.amount + creditCardRate(customer)

Does not compile
Using wrapper types

• Do not abuse primitives
• Control the available values
• Control the available operations
• Move validation to the correct place
Non empty list
def average(items: List[Int]): Int = items.sum / items.size

scala> average(List(5))
res1: Int = 5

scala> average(List(5, 10, 15))
res2: Int = 10

scala> average(List())
java.lang.ArithmeticException: / by zero
  at .average0(<console>:8)
  ... 35 elided
import org.scalactic.Every

def average(items: Every[Int]): Int = items.sum / items.size

scala> average(Every(5))
res1: Int = 5

scala> average(Every(5, 10, 15))
res2: Int = 10

scala> average(Every())
<console>:10: error: not enough arguments for method apply:
(firstElement: T, otherElements: T*)org.scalactic.Every[T] in
object Every.
Unspecified value parameters firstElement, otherElements.
    average(Every())
import org.scalactic.Every

def average(items: Every[Int]): Int = items.sum / items.size

def average(first: Int, rest: Int*): Int =
  average(Every(first, rest: _*))

scala> average(5)
res1: Int = 5

scala> average(5, 10, 15)
res2: Int = 10

scala> average()
<console>:11: error: not enough arguments for method average:
  (first: Int, rest: Int*)Int.
Unspecified value parameters first, rest.
  average()
Non empty lists

• Some lists cannot be empty
• Tell the compiler
• One-plus-var-args idiom
Algebraic data types
## Agents

<table>
<thead>
<tr>
<th>Agent Id</th>
<th>Type</th>
<th>Status</th>
<th>Host</th>
<th>In Use By</th>
</tr>
</thead>
<tbody>
<tr>
<td>A01</td>
<td>1</td>
<td>Active</td>
<td>10.0.0.1</td>
<td></td>
</tr>
<tr>
<td>A02</td>
<td>1</td>
<td>Failed</td>
<td>10.0.0.2</td>
<td>J01</td>
</tr>
<tr>
<td>A03</td>
<td>2</td>
<td>Active</td>
<td>10.0.0.3</td>
<td>J03</td>
</tr>
<tr>
<td>A04</td>
<td>2</td>
<td>Waiting</td>
<td>10.0.0.4</td>
<td></td>
</tr>
</tbody>
</table>

## Jobs

<table>
<thead>
<tr>
<th>Job Id</th>
<th>Type</th>
<th>Status</th>
<th>Submitted By</th>
<th>Processed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>J01</td>
<td>1</td>
<td>Waiting</td>
<td>Fred</td>
<td></td>
</tr>
<tr>
<td>J02</td>
<td>1</td>
<td>Active</td>
<td>Wilma</td>
<td>A01</td>
</tr>
<tr>
<td>J03</td>
<td>2</td>
<td>Complete</td>
<td>Barney</td>
<td>A03</td>
</tr>
</tbody>
</table>
case class Agent(agentId: String,
  jobType: Int,
  host: String,
  port: Int,
  status: String, // Waiting | Active | Failed
  maybeLastAccessed: Option[DateTime],
  inUse: Boolean,
  maybeInUseBy: Option[String])

case class Job(referenceId: String,
  jobType: Int,
  status: String, // Waiting | Active | Complete
  submittedBy: String,
  submittedAt: DateTime,
  maybeStartedAt: Option[DateTime],
  maybeProcessedBy: Option[String],
  maybeCompletedAt: Option[DateTime])
case class Agent(agentId: String,
    jobType: JobType,
    address: AgentAddress,
    status: AgentStatus,
    lastAccessed: Option[DateTime],
    inUse: Boolean,
    maybeInUseBy: Option[String])

case class Job(referenceId: String,
    jobType: JobType,
    status: JobStatus,
    submittedBy: User,
    submittedAt: DateTime,
    maybeStartedAt: Option[DateTime],
    maybeProcessedBy: Option[String],
    maybeCompletedAt: Option[DateTime])
sealed abstract class JobType(val value: Int)
case object SmallJob extends JobType(1)
case object LargeJob extends JobType(2)
case object BatchJob extends JobType(3)

sealed abstract class AgentStatus(val value: String)
case object AgentWaiting extends AgentStatus("Waiting")
case object AgentActive extends AgentStatus("Active")
case object AgentFailed extends AgentStatus("Failed")

sealed abstract class JobStatus(val value: String)
case object JobWaiting extends JobStatus("Waiting")
case object JobActive extends JobStatus("Active")
case object JobCompelete extends JobStatus("Complete")

case class AgentAddress(host: String, port: Int)
case class User(name: String)
case class Agent(agentId: String,
    jobType: JobType,
    address: AgentAddress,
    status: AgentStatus,
    lastAccessed: Option[DateTime],
    inUse: Boolean,
    maybeInUseBy: Option[String])

case class Job(referenceId: String,
    jobType: JobType,
    status: JobStatus,
    submittedBy: User,
    submittedAt: DateTime,
    maybeStartedAt: Option[DateTime],
    maybeProcessedBy: Option[String],
    maybeCompletedAt: Option[DateTime])
import tag.@@

trait Foo
def onlyFoo(value: String @@ Foo): String = s"It a foo: $value"

scala> onlyFoo("simple string")
<console>:13: error: type mismatch; found : String("simple string")
    required: tag.@@[String, Foo]
    (which expands to) String with tag.Tagged[Foo]
    onlyFoo("simple string")
    ^

scala> val foo = tag[Foo]("Foo String")
foo: tag.@@[String, Foo] = Foo String

scala> onlyFoo(foo)
res2: String = It a foo: Foo String

def anyString(value: String): String = s"Just a string: $value"

scala> anyString(foo)
res6: String = Just a string: Foo String
case class Agent(agentId: String, jobType: JobType, address: AgentAddress, status: AgentStatus, lastAccessed: Option[DateTime], inUse: Boolean, maybeInUseBy: Option[String])

case class Job(referenceId: String, jobType: JobType, status: JobStatus, submittedBy: User, submittedAt: DateTime, maybeStartedAt: Option[DateTime], maybeProcessedBy: Option[String], maybeCompletedAt: Option[DateTime])
case class Job(referenceId: String @@ Agent,
jobType: JobType,
status: JobStatus,
submittedBy: User,
submittedAt: DateTime,
maybeStartedAt: Option[DateTime],
maybeProcessedBy: Option[String @@ Job],
maybeCompletedAt: Option[DateTime])
def recordCompletionMetrics(job: Job): Unit = {
  for( startedAt <- job.maybeStartedAt;
      completedAt <- job.maybeCompletedAt ) {

    writeJobEvent(
      event = "Completed",
      time = completedAt,
      referenceId = job.referenceId,
      waitingTime = (startedAt - job.submittedAt),
      executionTime = (completedAt - startedAt))
  }
}
def recordCompletionMetrics(job: Job): Unit = {
    require(job.status = JobComplete)
    require(job.maybeStartedAt.isDefined)
    require(job.maybeCompletedAt.isDefined)

    for (startedAt <- job.maybeStartedAt;
         completedAt <- job.maybeCompletedAt) {

        writeJobEvent {
            event = "Completed",
            time = completedAt,
            referenceId = job.referenceId,
            waitingTime = (startedAt - job.submittedAt),
            executionTime = (completedAt - startedAt)
        }
    }
}
sealed trait Job {
    def referenceId: String @Job
    def status: JobStatus
    def submittedBy: User
    def submittedAt: DateTime
}

case class WaitingJob(referenceId: String @Job, submittedBy: User, submittedAt: DateTime) extends Job {
    val status: JobStatus = JobWaiting
}
sealed trait Job {
  def referenceId: String @@ Job
  def status: JobStatus
  def submittedBy: User
  def submittedAt: DateTime
}

case class ActiveJob(referenceId: String @@ Job,
  submittedBy: User,
  submittedAt: DateTime,
  startedAt: DateTime,
  processedBy: String @@ Agent)
  extends Job {
    val status: JobStatus = JobActive
}
sealed trait Job {
  def referenceId: String @@ Job
  def status: JobStatus
  def submittedBy: User
  def submittedAt: DateTime
}

case class CompleteJob(referenceId: String @@ Job,
  submittedBy: User,
  submittedAt: DateTime,
  startedAt: DateTime,
  processedBy: String @@ Agent,
  completedAt: DateTime)
  extends Job {
    val status: JobStatus = JobComplete
  }
def recordCompletionMetrics(job: CompleteJob): Unit = {
  writeJobEvent(
    event = "Completed",
    time = job.completedAt,
    referenceId = job.referenceId,
    waitingTime = (job.startedAt - job.submittedAt),
    executionTime = (job.completedAt - job.startedAt))
}
Algebraic data types

- Simply sealed traits and case classes
- Exposes the shape of your data
- Use this shape to control possible states
"Smart data structures and dumb code works a lot better than the other way around" - Eric Raymond
Path dependent types
trait Handle {
  def name: Name
  def owner: User
}

trait Data {
  def stream: OutputStream
}

trait Storage {
  def create(name: Name, owner: User, data: OutputStream): Handle

  def find(name: Name): Option[Handle]

  def read(handle: Handle): Try[Data]
}
case class HandleImpl(id: Long,
   name: Name,
   owner: User) extends Handle

case class DataImpl(stream: OutputStream) extends Data

class StorageImpl extends Storage {
  // ...

  def read(entryDesc: Handle): Try[Data] = {
    require(entryDesc.isInstanceOf[HandleImpl])

    val impl = entryDesc.asInstanceOf[HandleImpl]
    dataStore.read(impl.id)
  }
}
val riakStorage: Storage = ???

val maybeData = for {
  handle <- riakStorage.find(someName)
  data <- riakStorage.read(handle).toOption
} yield data
val riakStorage: Storage = ???
val memoryStorage: Storage = ???

val maybeData = for {
  handle <- riakStorage.find(someName)
  data <- memoryStorage.read(handle).toOption
} yield data

IllegalArgumentException
trait HandleLike {
  def name: Name
  def owner: User
}

trait DataLike {
  def stream: OutputStream
}

trait Storage {
  type Handle <: HandleLike
  type Data <: DataLike

  def create(name: Name, owner: User, data: OutputStream): Handle

  def find(name: Name): Option[Handle]

  def read(handle: Handle): Try[Data]
}
private[impl] class StorageImpl extends Storage {
  type Handle = HandleImpl
  type Data = DataImpl

  case class HandleImpl(id: Long,
      name: Name,
      owner: User) extends HandleLike

  case class DataImpl(stream: OutputStream) extends DataLike

  // ...

  def read(entryDesc: Handle): Try[Data] = {
    dataStore.read(entryDesc.id)
  }
}
val riakStorage: Storage = ???
val memoryStorage: Storage = ???

val maybeData = for {
    handle <- riakStorage.find(someName)
    data <- memoryStorage.read(handle).toOption
} yield data

error: type mismatch;
  found  : handle.type (with underlying type riakStorage.Handle)
required: memoryStorage.Handle
    data <- memoryStorage.read(handle).toOption
^
val riakStorage1: Storage = ???
val riakStorage2: Storage = ???

val maybeData = for {
    handle <- riakStorage1.find(someName)
    data <- riakStorage2.read(handle).toOption
} yield data

error: type mismatch;
found  : handle.type (with underlying type riakStorage1.Handle)
required: riakStorage2.Handle
    data <- riakStorage2.read(handle).toOption
^
Path dependent types

- Family polymorphism
- Remove casts
- Bound to instances not classes
More advanced techniques

• Self-recursive types
• Phantom types
• Shapeless
Defensive Programming

Fail Fast

Design by Contract
“A mind is like a parachute, it doesn’t work unless its open”

- Frank Zappa