Let The Compiler Help You: How To Make The Most Of Scala’s Typesystem

Markus Hauck (@markus1189)

Scala.IO 2017
One Evening

LETS PROGRAM ALL THE THINGS
Done!?

```bash
> compile
[success] Total time: 42s, completed Nov 2 14:05:42
> run
...
```
JAVA.LANG.UNSUPPORTEDOPERATIONEXCEPTION: EMPTY.MAX
Done!?

> compile
[success] Total time: 42s, completed Nov 2 14:05:42
> run
...

Markus Hauck (@markus1189)
JAVA.LANG.ILLEGALARGUMENTEXCEPTION

#!@ IS NOT A VALID VERSION STRING
WHAT IF I TOLD YOU
IT DOESN'T HAVE TO BE THIS WAY
A Better Way

> compile

[info] Compiling 1 Scala source to
/home/brain/world-domination/target/scala-2.12/classes ...  
[error] found : Int(-42)  
[error] required: PositiveInt  
[error] requiredAssets(-42)  
[error] ^  
[error] one error found  
[error] (compile:compileIncremental) Compilation failed
NO NEED TO RUN

YOU HAVE AN ERROR THERE
Introduction

- Currently: programming as a fight against the compiler
- Soon: work **together** with the compiler
- Tests: **evidence** it works
- Types: **proof** it works
- You need to communicate with the compiler
The Road Ahead

- Preparations
- Be Honest
- Forbid it
- No Garbage
- Only Valid Ops
Become Friends

• by default the compiler is a little shy
• but once you get friends, you don’t want to go back
• first thing: unmute him!
  • -deprecation
  • -unchecked
  • -Xfuture
  • -Xlint:-unused
  • -Ywarn-unused:imports,privates,locals

• sometimes false positives, see
  • scalac -X
  • scalac -Y
Not: Documentation

- documentation is not for the compiler
- it is meant for humans (which are bad at it)
- avoid: context sensitive reasoning
- think: I know this can’t happen
- use a language that the compiler understands: types
Case Study: Vending Machine

- insert coin (50 cents or 1 euro)
- push button for drink
- abort the transaction and return money
final class VendingMachine(id: Int) {
    require(id > 0 && id < 100, "Invalid identifier")
    private[this] var amount: Int = 0

    def insertMoney(cents: Int): Unit = cents match {
      case 50 | 100 => amount += cents
      case _ => throw new IllegalArgumentException("...")
    }

    def pushButton(): Unit = if (amount == 100) {
      () // eject
    } else {
      throw new IllegalStateException("...")
    }

    def abort(): Int = if (amount > 0) {
      val res = amount; amount = 0; res
    } else {
      throw new IllegalStateException("...")
    }
}
A First Design

- what does the compiler know?
- what do you as a user of this class know without docs?

READ ALL THE DOCS?

AIN'T NOBODY GOT TIME FOR THAT

memegenerator.net
Step 1: Be Honest
Get The Max Out Of Your List

```scala
def max: A
    [use case]
    Finds the largest element.

returns the largest element of this list.

Definition Classes TraversableOnce → GenTraversableOnce

Full Signature
```
Get The Max Out Of Your List

> List[Int](1, 2, 3).max
res0: Int = 3

> List[Int]().max
java.lang.UnsupportedOperationException: empty.max
Be Honest

- not honest $\implies$ no help
- max pretends to always return something, which is not the case!
- tell the compiler this operation can fail
- Option: if no result is a result
- Either: if there can be errors
- Custom ADT: if Either doesn’t cut it
Be Honest

- how does this apply to our case study?
- `insertMoney` can fail
- `pushButton` can fail
- `abort` can fail
- we will fix this later
Step 2: If not allowed, forbid it
Step 2: If not allowed, forbid it

- **every** project has domain classes with invariants
- how to verify those invariants?

```scala
if (input.isValid) {
  ???
} else {
  throw new Exception("whoopsie")
}
```

```scala
class ImportantStuff(stuff: Stuff) {
  require(stuff.isImportant, "Not important!")
}
```

- but the compiler (and others) does not know this!
If not allowed, forbid it

```scala
case class Email(value: String) extends AnyVal {
  require(isValidEmail(value))
}
```

```scala
> Email("markus.hauck@codecentric.de")
res1: Email = ...
```

```scala
> Email("Hello World!")
java.lang.IllegalArgumentException:
    Not a valid email address
```
If not allowed, forbid it

• how can we improve?
• for methods, the **return type** changed
• instantiation doesn’t have one?
• one solution: smart constructors / factories
If not allowed, forbid it

abstract case class Email private (...)

object Email {
    def fromString: Either[ValidationErrorResponse, Email] = ???  // exercise
}

> Email.fromString("markus.hauck@codecentric.de")
res1: Either[ValidationErrorResponse, Email] =
    Right("markus.hauck@codecentric.de")

> Email.fromString("Hello World")
res2: Either[ValidationErrorResponse, Email] =
    Left(InvalidEmail)
Case Study

- Okay, back to our case study
- we want to fix:
  - methods that are not honest
  - forbid invalid instantiation
final class VendingMachine private (id: Int) {
  private[this] var amount: Int = 0

  def insertMoney(cents: Int): Either[InvalidCoin, Unit] =
    cents match {
      case 50 | 100 =>
        amount += cents
        Right(())
      case _ =>
        Left(InvalidCoin)
    }

  def pushButton(): Either[InsufficientFunds, Unit] =
    if (amount == 100) { Right(()) }
    else { Left(InsufficientFunds) }

  def abort(): Either[NoChange, Int] = if (amount > 0) {
    val res = amount; amount = 0; Right(res)
  } else Left(NoChange)
}
object VendingMachine {
    def create(id: Int): Either[InvalidId, VendingMachine] =
        if (id > 0 && id < 100) {
            Right(new VendingMachine(id))
        } else {
            Left(InvalidId)
        }
}

// define left sides of eithers
Case Study: Review

- we got rid of all exceptions
- no way to “forget” that something can fail
- compiler (coworkers?) now knows quite a bit more
- what else can we improve?
Step 3: Don’t accept garbage
Don’t accept garbage input

- another everyday example:

```scala
> List(1, 2, 3).take(-42)
```

- Quiz: what happens?
Don’t accept garbage input

• another everyday example:

```scala
> List(1, 2, 3).take(-42)
```

• Quiz: what happens?

```scala
res1: List[Int] = List()
```
Don’t accept garbage input

- our constructor and methods are quite liberal
- they take almost everything!
- does the validation really belong in our vending machine?
- actually it is the **caller**’s fault!
- better: don’t accept garbage and let the caller to the work
- push validation to the boundaries of your system
- avoid doing it over and over again in program flow
Putting It Into Practice

class MailService {
    def sendEmail(mail: String):
        Either[MailValidationException, MailStatus]
}

class MailService {
    def sendEmail(mail: Email): MailStatus
}
Putting It Into Practice

> `List(1, 2, 3).head`
res1: `Option[Int] = Some(1)`

> `NonEmptyList.of(1, 2, 3).head`
res1: `Int = 1`
Putting It Into Practice

sealed abstract class List[+A] {
  def apply(index: Int): A
}

sealed abstract class List[+A] {
  def apply(index: Natural): A
}
How To Validate

- okay seems like it makes sense to do this
- goal: make core logic more focused, factor out error handling
- how?
  - use a wrapper with smart constructors
  - use phantom types as “tags”
Validation With Smart Constructors

abstract case class Email private (...)

object Email {
   def fromString: Either[Validation Error, Email] =
   ??? // exercise
}

• that works, but becomes cumbersome pretty quick
• I know it will succeed!

> Email.fromString("foo@bar.de")
Phantom Types

- phantom type: not associated to any value
- only exists at compile time
- attach information to values
- example: validation of user input
- instead of repeated validation “to be sure”, enforce with types
Phantom Types: The Idea

sealed trait Validation
final abstract class Valid extends Validation
final abstract class Unknown extends Validation

abstract case class UserInput[A <: Validation] (value: String)

object UserInput {
  def unknown(s: String): UserInput[Unknown] =
    new UserInput[Unknown](s) {}

  def validate(input: UserInput[Unknown]): Option[UserInput[Valid]] = ???
}
Phantom Types for Tags

- Creating new classes for invariants works, but very cumbersome
- But: Whenever input enters your system: require validation
- Avoid validation again and again because of different flows through system and refactorings (context-free!)
- annoying: static input requires validation
- skipping: Tagged to avoid wrappers
Refined

- the refined library to the rescue
- in essence: a `Refined[T, P]`
- actual value $T +$ phantom type for predicate $P$
- no wrapper, no custom phantom type (ADT)
- only define your predicate or use the builtin ones
- killer feature: perform validation for literals as input at compile time
Refined: Predicates

- DSL to define predicates
- many builtins: link
- useful as documentation as well

```scala
String Refined Uri
String Refined Uuid
String Refined MatchesRegex[W."^[0-9]+$".T]

type ZeroToOne = Not[Less[W."0.0".T]]
    And Not[Greater[W."1.0".T]]

type ValidChar =
    AnyOf[Digit :: Letter :: Whitespace :: HNil]
```
Using Refined

```scala
object Refined {
  def index[A](xs: List[A])(
    i: Int Refined Positive): A = xs(i)

  index(List(1, 2, 3))(2)

  index(List(1, 2, 3))(-1)
}
```

[error] index(List(1, 2, 3))(-1) // compile error
[error] ~
[error] one error found
Refined: Dynamic Input

- but the macro works only for literal input
- dynamic values: you still have to take care

```scala
> refineV[Positive](fortyTwo)
res1: Either[String, Int Refined Positive] = Right(...)

> refineV[Positive](negativeInt)
res2: Either[String, Int Refined Positive] = Left(...)```
Case Study

```scala
final class VendingMachine(id: Identifier) {
    private[this] var amount: Int = 0
    def insertMoney(cents: Coin): Unit = ???
    def pushButton(): Either[InsufficientFunds, Unit] = ???
    def abort(): Either[NoChange, Int] = ???
}

object VendingMachine {
    type Identifier = Int Refined
    Interval.Closed[W.‘1‘.T, W.‘100‘.T]
}

sealed trait Coin
case object FiftyCents extends Coin
case object OneEuro extends Coin
```
Case Study

- the **Identifier** is checked by refined
- **insertMoney** only allows valid coins by design
- we can get rid of the **Either** in **insertMoney**
- what else?
Step 4: Restrict Valid Operations
Vending State Machine

- **Idle**
  - Start → Idle
  - 50||-
  - abort||50

- **Half**
  - 50||-

- **Ready**
  - 1||-
  - abort||1
  - pushButton||drink

Markus Hauck (@markus1189) Let The Compiler Help You: How To Make The Most Of Scala's Typesystem 50 / 61
Vending State Machine

- possible:
  - $50 \rightarrow 50 \rightarrow \text{pushbutton}$
  - $50 \rightarrow \text{abort}$
  - $1 \rightarrow \text{pushbutton}$
  - $1 \rightarrow \text{abort}$

- available actions depend on the \textbf{implicit state}

- methods can return \textbf{different types} depending on the state, e.g., abort

- so we need to check those two invariants (via types)!
State Machine: States

**sealed trait VState**
**final abstract class Idle extends VState**
**final abstract class Half extends VState**
**final abstract class Ready extends VState**
State Machine: Vending Machine

```scala
class VendingMachine[S <: VState] private {
  def insertFirst50()(implicit ev: S =:= Idle):
    VendingMachine[Half] = new VendingMachine

  def insertSecond50()(implicit ev: S =:= Half):
    VendingMachine[Ready] = new VendingMachine

  def insertEuro()(implicit ev: S =:= Idle):
    VendingMachine[Ready] = new VendingMachine

  def pushButton()(implicit ev: S =:= Ready):
    (VendingMachine[Idle], Drink) =
    (new VendingMachine[Idle], Drink("Fizz"))

  def abort[T, O]()(implicit ev: Next.Aux[S, T, O]):
    (VendingMachine[T], O) =
    (new VendingMachine[T], ev.coin)
}
```

Markus Hauck (@markus1189)
State Machine: Typeclass

sealed abstract class Next[S <: VState] {
  type Next <: VState
  type Out <: Coin
  def coin: Out
}

final object Next {
  type Aux[S0 <: VState, NO <: VState, OO <: Coin] =
    Next[S0] {
      type Next = NO
      type Out = OO
    }
State Machine: Implicit Evidence

```scala
implicit val halfAbort:
      type Next = Idle; type Out = FiftyCents.type
      override val coin = FiftyCents
    }
```
State Machine: Implicit Evidence

```scala
implicit val readyAbort:
  Next.Aux[Ready, Idle, OneEuro.type] = new Next[Ready] {
  type Next = Idle; type Out = OneEuro.type
  override val coin = OneEuro
}
```
State Machines: Usage

```scala
object VendingMachineExamples {
  val machine = VendingMachine.initial

  // machine.insertSecond50()
  // Cannot prove that

  // machine.abort()
  // no implicit found for Next[Idle, ???, ???]

  machine.insertFirst50().insertSecond50().pushButton()
  machine.insertEuro().pushButton()
}
```
State Machines: Summary

- calling methods only compiles when it makes sense
- eliminates our **Either** return types
- but: restricted to immutability (change type parameters)
- potentially less code than inheritance (and less weird)
Be Honest

Forbid It

No Garbage

Only Valid Ops

Markus Hauck (@markus1189) Let The Compiler Help You: How To Make The Most Of Scala's Typesystem
The Compiler Is There To Help!

THANKS!

https://github.com/markus1189/scala-io-compiler-help
• our vending machine is still quite liberal

```scala
def main() = {
  val machine: VendingMachine[Ready] = 
    VendingMachine.initial.insertEuro()
  val drink1 = machine.pushButton()._2
  val drink2 = machine.pushButton()._2
  // ...
}
```

• how could we restrict this \textbf{with types}?
• hint: limit access to the state machine