11 - Illegal States

Einführung in die Funktionale Programmierung

Prof. Frank Puppe, Felix Herrmann, Alexander Gehrke

Sommersemester 2023

Lehrstuhl für Informatik VI, Uni Würzburg

Overview

In this lecture, let's look at a few techniques to helps us make our programming more robust. Unlike the former lectures, this might be a bit more Scala specific.

- Make illegal states unrepresentable
- Force compiler errors on model changes
- Introduce new types
- Forgo boolean blindness

Making illegal states unrepresentable

(shamelessly ~~stolen~~ adapted from Scott Wlaschin's "Designing With Types")

Let's take a look at the following `case class`:

```
final case class Contact(
  name: String,
  mail: EmailContactInfo,
  address: PostalContactInfo,
)
```

Seems reasonable. We require every contact to have both types of addresses. But what if our business rule is that each contact has to have a postal or an email address?

Making illegal states unrepresentable

Et voilà!

```
final case class Contact(
  name: String,
  mail: Option[EmailContactInfo],
  address: Option[PostalContactInfo],
)
```

Done. This was **easy**!

But does this describe our business rule?

Making illegal states unrepresentable

Et voilà!

```
final case class Contact(
  name: String,
  mail: Option[EmailContactInfo],
  address: Option[PostalContactInfo],
)
```

Done. This was **easy**!

But does this describe our business rule? No it doesn't it is now possible have neither, which violates our business rule.

Let's try to fix that!

Making illegal states unrepresentable

The solution:

```
enum ContactInfo:
  case OnlyMail(info: EmailContactInfo)
  case OnlyPostal(info: PostalContactInfo)
  case Both(
    mail: EmailContactInfo,
    postal: PostalContactInfo
  )

final case class Contact(
  name: String,
  info: ContactInfo
)
```

This models our business rule exactly.

Making illegal states unrepresentable

Note that it is now impossible to write code that compiles and violates the business rule.

There is no need to do any unit test for this rule.

It could be said that our model is now much more complicated than the original. And that's true. It's on you to decide whether the gain is worth it. Here are two things to think about:

- The old model was either wrong, incomplete or did require you and every other programmer to keep the rule in their heads.
- Our business rule is really that complex. This is an inherent complexity and it won't go away. Either we have to keep it in our heads, document and test it or model it correctly.

It's a trade-off.

Making illegal states unrepresentable

Okay, let's look at another example. We want to write a software like Gitlab to manage multiple code repositories.

A repository has a name, a creation date. It has normal users, which may commit to it. It also has administrators which are the only ones who may change permissions, add new administrators, remove users, create branches, delete the project and so on.

We model it like that:

```
final case class Repository(
  projectName:  String,
  creationDate: Instant,
  admins:       List[User],
  users:        List[User],
)
```

Whats wrong here?

Making illegal states unrepresentable

We might run into serious trouble if the last admin leaves. Since they are the only ones who can do a bunch of stuff and the only ones who can promote new admins, the last admin leaving basically locks the project.

How could we change our model to make it impossible to have no admins in the project?

Making illegal states unrepresentable

We might run into serious trouble if the last admin leaves. Since they are the only ones who can do a bunch of stuff and the only ones who can promote new admins, the last admin leaving basically locks the project.

How could we change our model to make it impossible to have no admins in the project?

```
final case class Repository(
  projectName:  String,
  creationDate: Instant,
  admin1:       User,
  admins:       List[User],
  users:        List[User],
)
```

This isn't bad. Now there is no way to create a repository with no admin in it. Can we do better?

Making illegal states unrepresentable

We can't do better with regard to modeling the business rules. We are already spot on. We can do a little better with regard to simplifying our code.

Let's try to factor out the "at least one"-part:

```
final case class NonEmptyList[A](head: A, tail: List[A])
```

Making illegal states unrepresentable

We can't do better with regard to modeling the business rules. We are already spot on. We can do a little better with regard to simplifying our code.

Let's try to factor out the "at least one"-part:

```
final case class NonEmptyList[A](head: A, tail: List[A])
```

Now we can write our model like this:

```
final case class Repository(
  projectName:  String,
  creationDate: Instant,
  admins:       NonEmptyList[User],
  users:        List[User],
)
```

Making illegal states unrepresentable

This has two advantages:

- we only have to learn about `NonEmptyList` once and when we see it, the meaning should be immediately obvious to us, even without looking at other fields of our case class.
- we can define a lot of useful type classes for `NonEmptyList` which we get for free when we use it. `Traverseable` is such an example.

As a side note: We can get a `MonoidK[List]` for concatenating lists. Is the same true for `NonEmptyList`?

Making illegal states unrepresentable

This has two advantages:

- we only have to learn about `NonEmptyList` once and when we see it, the meaning should be immediately obvious to us, even without looking at other fields of our case class.
- we can define a lot of useful type classes for `NonEmptyList` which we get for free when we use it. `Traverseable` is such an example.

As a side note: We can get a `MonoidK[List]` for concatenating lists. Is the same true for `NonEmptyList`?

No, because we can't create an empty list and therefore have no `zero` value.

Forcing compiler errors on model changes

Let's move on.

What if want to add two other contact methods like home and work phone numbers to our contact from the previous example?

We would have to encode that with 15 enum cases. That seems a bit excessive. And it is. Let's go back to our original design for a moment:

```
final case class Contact(
  name: String,
  mail: Option[EmailContactInfo],
  address: Option[PostalContactInfo],
)
```

Forcing compiler errors on model changes

```
final case class Contact(
  name: String,
  mail: Option[EmailContactInfo],
  address: Option[PostalContactInfo],
)
```

Here is a function to work on that data structure:

```
def getReport(c: Contact): String =
  s"${c.name} -- ${c.mail.getOrElse("n/a")} -- ${c.address.getOrElse("n/a")}"
```

Forcing compiler errors on model changes

But this leaves us with a new problem. Let's change our model to include the new types of addresses:

```
final case class Contact(
  name: String,
  mail: Option[EmailContactInfo],
  address: Option[PostalContactInfo],
  homePhone: Option[HomePhoneContactInfo],
  workPhone: Option[WorkPhoneContactInfo],
)
```

Sadly, our old function is now buggy! And still compiles.

```
def getReport(c: Contact): String =
  s"${c.name} -- ${c.mail.getOrElse("n/a")} -- ${c.address.getOrElse("n/a")}"
```

Can we do better?

Forcing compiler errors on model changes

We can.

```
enum ContactInfo:
    case EmailContactInfo(dummy: String)
    case PostalContactInfo(dummy: String)
    case HomePhoneContactInfo(dummy: String)
    case WorkPhoneContactInfo(dummy: String)

final case class Contact(
  name: String,
  infos: NonEmptyList[ContactInfo],
)
```

This also solves our initial problem of having to have at least one contact info.

How does our report function look like now?

Forcing compiler errors on model changes

```
def getReport(c: Contact): String =
  s"${c.name} -- " + c.infos.toList.map({
    case EmailContactInfo(d) => s"mail $d"
    case PostalContactInfo(d) => s"postal $d"
    case HomePhoneContactInfo(d) => s"home phone $d"
    case WorkPhoneContactInfo(d) => s"work phone $d"
  }).mkString(", ")
```

This is much better. Whenever we add a new contact info to our list of possible info types we break our program, thanks to exhaustiveness checks.

Forcing compiler errors on model changes

The lesson:

- Write your model in a way which breaks compilation on incompatible model changes
- We don’t fear breaking our programs as long as we do it at compile time. Never fear, our types are here!

Introducing new types

And now for something completely different.

```
final case class Config(
  user:    String,
  path:    String,
  host:    String,
  retries: Int,
  port:    Int,
)

def readUser:    Either[Throwable, String]
def readPath:    Either[Throwable, String]
def readHost:    Either[Throwable, String]
def readPort:    Either[Throwable, Int   ]
def readRetries: Either[Throwable, Int   ]

def readAll: Either[Throwable, Config] =
  (readUser, readPath, readHost, readPort, readRetries)
    .mapN(Config.apply) // mapN is similar to map2, just more arguments
```

Where's the problem?

Introducing new types

The problem is that `retries` and `port` are used in the wrong order when passed to `Config` in the `mapN` call.

This was only possible because they both have the type `Int`. But should they?

Introducing new types

There are (at least) two very good reasons why `retries` and `port` should not be of type `Int`

- Both variables can be assigned to each other, though a port and the number of retries are obviously not the same thing. This is a strong indication that something is amiss.
- Both of them really have constraints that `Int` doesn't have. A port number should be within the range of valid ports, while the number of retries should never be negative.

We want to fix the first point first. And to keep things simple, we will only concentrate on the `port`.

Introducing new types

So we want `port` and `retries` to have different types, but the underlying type is still an integer. To represent this, we can use an opaque type alias:

```
opaque type Port = Int
object Port:
  def apply(i: Int): Port = i
```

This works like a normal type alias, but outside its definition scope (i.e. in this case outside the `Port` object), it is treated like a completely separate, incompatible type. To get a `Port` value, we now need to use the `apply` method explicitly:

```
val http: Port = Port(80)
// would not compile:
// val http: Port = 80
```

Introducing new types

On to our second problem: Restricting the port range.

As we said, any code outside the `Port` object has to go through the `Port.apply` method, we can add any necessary restrictions there:

```
opaque type Port = Int
object Port:
  def apply(i: Int): Option[Port] =
    if i > 0 && i <= 65535 then Some(i) else None
```

We now return an Option, that only contains valid ports, so no code outside the object can create a `Port` with an invalid integer value.

Introducing new types

We can make it even better for compile time constants by throwing a compile time error:

```
opaque type Port = Int
object Port:
  def apply(i: Int): Option[Port] =
    if i > 0 && i <= 65535 then Some(i) else None
  import scala.compiletime.*
  inline def checked(inline i: Int): Port =
    inline if i > 0 && i <= 65535 then i
    else error("Port not in range: " + codeOf(i))
//val a: Port = Port.checked(-1)
// ↑ Compile time error
```

Introducing new types

```
opaque type Port = Int
object Port:
  def apply(i: Int): Port = i
```

```
final case class Config(
  user:    String,
  path:    String,
  host:    String,
  retries: Int,
  port:    Port,
)
//def readAll: Either[Throwable, Config] =
//  (readUser, readPath, readHost, readPort, readRetries)
//    .mapN(Config.apply) // mapN is similar to map2, just more arguments
```

Now our `readAll` function doesn't compile anymore \o/

Introducing new types

Where does this leave us?

Whenever you have two sets of values which have the same type but are very different semantically, create a new type to wrap them.

If you also require the validation of constraints before creating the type, this is known as the Smart Constructor Pattern.

It might seem overkill, but there is a good chance this habit will save you a lot of headache down the road.

Introducing new types - other languages

How about doing this in other languages?

Opaque type aliases are becoming more common in programming languages. Sometimes they are also called newtypes. E.g. Rust and Haskell both have equivalent constructs.

What if your favorite typed language doesn't have them? An alternative is to use a wrapper type (like a case class or similar).
This comes with more trade-offs: you usually have a runtime overhead for wrapping and there are more pitfalls with making it safe, e.g. you must remember to prevent inheriting, make sure no default constructor exists, prevent any standard language methods to modify the class, etc.

Forgoing boolean blindness

Let's take a look at two code examples:

```
def age: Option[Int] = ???
if age.nonEmpty then transformSomehow(age.get)
else                 someValue
```

```
def getList: List[Int] = ???
if getList.size > 2 then transformSomehow(getList(1))
else                     someValue
```

In both cases, we reduce the information we have to a boolean. And this boolean has no direct connection to whether the expressions guarded by it are now valid or not. It's only that we as the programmer know that being nonempty means that we can call `get`.

Throwing away information in that way is called **boolean blindness**. The general principle goes beyond booleans though and there are many examples of it.

Forgoing boolean blindness

We already know how to do better. Use pattern matching:

```
def age: Option[Int] = ???
age match
  case Some(a) => transformSomehow(a)
  case None    => someValue

def getList: List[Int] = ???
getList match
  case _::x::_ => transformSomehow(x)
  case _       => someValue
```

In both of those cases we use pattern matching to extract the value and treat the value itself as a proof that it's there.

Let's look at two examples which aren't so obvious.

Forgoing boolean blindness

Regex matching. Java style

```
val name: Pattern = Pattern.compile("(\\w+), (\\w+)")
def getFirstAndLastName(s: String): Option[(String, String)] =
  val matcher = name.matcher(s)
  if matcher.matches then Some((matcher.group(1), matcher.group(2)))
  else None
```

Forgoing boolean blindness

Regex matching. Java style

```
val name: Pattern = Pattern.compile("(\\w+), (\\w+)")
def getFirstAndLastName(s: String): Option[(String, String)] =
  val matcher = name.matcher(s)
  if matcher.matches then Some((matcher.group(1), matcher.group(2)))
  else None
```

Scala style

```
val Name = "(\\w+), (\\w+)".r
def getFirstAndLastName(s: String): Option[(String, String)] =
  s match
    case Name(last, first) => Some((last, first))
    case _                 => None
```

Forgoing boolean blindness

What happens if I want to `filter` and `map`?

```
def allStreetsWrong(l: List[Person]): List[String] =
  l.filter(_.address.nonEmpty).map(_.address.get.street)
```

Using `filter`+`map`, we can't do it in one step:

```
def allStreetsHow(l: List[Person]): List[String] =
  l.map(_.address match {
    case Some(a) => a.street
    case None    => ??? // what to do here?
  })
```

Forgoing boolean blindness

We can use `flatMap` to not throw information away between the two calls:

```
def allStreetsRight(l: List[Person]): List[String] =
  l.flatMap(_.address match {
    case None    => Nil
    case Some(a) => a.street :: Nil
  })
```

Here, the information that there is an address is never lost.

Forgoing boolean blindness

To sum up:

Whenever you use an unsafe function because you think you need to, try if you can come up with a solution which doesn't loose the information you need to forgo the calling of the unsafe method.