3 — Handling Errors Without Exceptions

Einführung in die Funktionale Programmierung

Prof. Frank Puppe, Felix Herrmann, Alexander Gehrke

Sommersemester 2023

Lehrstuhl für Informatik VI, Uni Würzburg

Mean of Doubles

How to implement a function which calculates the mean of a list of doubles?

```
def mean1(xs: List[Double]): Double = xs.sum / xs.size
```

Mean of Doubles

How to implement a function which calculates the mean of a list of doubles?

```
def mean1(xs: List[Double]): Double = xs.sum / xs.size
```

What happens if the list is empty? How do we deal with errors?

Mean of Doubles / Sentinel Values

Use sentinel values. `NaN` in this case

```
def mean1(xs: List[Double]): Double = xs.sum / xs.size
```

Has a lot of problems:

- Silently propagates through code. Caller doesn't have to check for NaN.
- If you want to check, you have to put if/else statements everywhere
- Maybe there is no sentinel value
- Requires special policy to call which might be at odds with how some function uses it when passed as a higher order function

Mean of Doubles / Multiple Return Values

```
def mean2(xs: List[Double]): (Boolean, Double) =
  if xs.isEmpty then (true, 0.0)
  else (false, xs.sum / xs.size)
```

Isn’t too bad if you get compiler support, but:

Mean of Doubles / Multiple Return Values

```
func (iter *BatchObjectIter) Next() (...) {
  header, err := iter.f.ReadString('\n')
  if err != nil {
    return OID{}, "", 0, nil, err
  }
  oid, objectType, objectSize, err := parseBatchHeader("", header)
  if err != nil {
    return OID{}, "", 0, nil, err
  }
  data := make([]byte, objectSize+1)
  _, err = io.ReadFull(iter.f, data)
  if err != nil {
    return OID{}, "", 0, nil, err
  }
  data = data[:len(data)-1]
  return oid, objectType, objectSize, data, nil
}
```

Works, can’t be missed (at least in go), but needs to be littered everywhere

I

won’t

I

won’t

even...

Mean of Doubles / Global/Thread-Local Error Variables (**errno**)

```java
public static int getClockedInTime(ResultSet rs) {
  var startTime = rs.getInt("timestamp");
  rs.next();
  var endTime = rs.getInt("timestamp");
  rs.next();

  var warnings = rs.getWarnings();
  if (warnings != null) {
    throw new RuntimeException(warnings);
  }

  return endTime - startTime;
}
```

Mean of Doubles / Global/Thread-Local Error Variables (**errno**)

```java
public static int getClockedInTime(ResultSet rs) {
  var startTime = rs.getInt("timestamp");
  rs.next();
  var endTime = rs.getInt("timestamp");
  rs.next();

  var warnings = rs.getWarnings();
  if (warnings != null) {
    throw new RuntimeException(warnings);
  }

  return endTime - startTime;
}
```

`getWarnings` gets cleared on `rs.next()`, so we are ignoring half of the warnings.

Mean of Doubles / Exceptions

```scala
def mean3(xs: List[Double]): Double =
  if xs.isEmpty then
    throw new ArithmeticException("mean of empty list")
  else xs.sum / xs.size
```

First, the positive aspects:

- Doesn’t clutter our code like in the go example
- Can be handled at one central location

But:

- They are not typesafe: the signature of a function throwing an exception doesn't tell us anything about it's exceptional behavior
- They don't need to be handled. The programmer might forget about them.
- They are not referentially transparent:

Since exceptions break referential transparency, we're back returning the only thing we can: values. We need a new datatype to represent our problem:

Since exceptions break referential transparency, we're back returning the only thing we can: values. We need a new datatype to represent our problem:

```scala
def mean(xs: List[Double]): Option[Double] =
  if xs.isEmpty then None
  else Some(xs.sum / xs.size)
```

- Let `None` represent failure
- Let `Some` represent success

Let's call this data type `Option`

```
enum Option[+A]:
  case Some(get: A)
  case None
```

We have two cases:

- We have a value (`Some`)
- We don't have a value (`None`)

To work with it, we need a few more useful functions:

```
enum Option[+A]:
  case Some(get: A)
  case None

  def map[B](f: A => B): Option[B] = ???
```

Implement the `map` functions, which takes a function and uses it to transform the value within (if there is one).

Solution `map`

```
def map[B](f: A => B): Option[B] = this match
  case None => None
  case Some(a) => Some(f(a))
```

To work with it, we need a few more useful functions:

```
enum Option[+A]:
  case Some(get: A)
  case None

  def map[B](f: A => B): Option[B] = ???
  def getOrElse[B >: A](default: => B): B = ???
```

Implement the `getOrElse` function, which returns the value if it is there and gives you the passed default value otherwise.

For now, you can ignore the `=>` in the parameter type. For referentially transparent code, `=> B` is semantically the same as `B` and just an optimization. We will discuss its effects in detail next week.

Solution `getOrElse`

```
def getOrElse[B>:A](default: => B): B = this match
  case None => default
  case Some(a) => a
```

To work with it, we need a few more useful functions:

```
enum Option[+A]:
  case Some(get: A)
  case None

  def map[B](f: A => B): Option[B] = ???
  def getOrElse[B >: A](default: => B): B = ???
  def flatMap[B](f: A => Option[B]): Option[B] = ???
```

Implement the `flatMap` function, which works like `map` but takes a function which yields an `Option[A]` instead of A.

Solution `flatMap`

```
def flatMap[B](f: A => Option[B]): Option[B] =
  map(f) getOrElse None
```

– or –

```
def flatMap[B](f: A => Option[B]): Option[B] = this match
  case None => None
  case Some(a) => f(a)
```

To work with it, we need a few more useful functions:

```
enum Option[+A]:
  case Some(get: A)
  case None

  def map[B](f: A => B): Option[B] = ???
  def getOrElse[B >: A](default: => B): B = ???
  def flatMap[B](f: A => Option[B]): Option[B] = ???
  def filter(f: A => Boolean): Option[A] = ???
```

Implement the function `filter`, which takes a predicate and yields `None` if the predicate doesn't match and keeps the value otherwise.

Solution `filter`

```scala
def filter(f: A => Boolean): Option[A] = this match
  case Some(a) if f(a) => this
  case _ => None
```

– or –

```scala
def filter(f: A => Boolean): Option[A] =
  flatMap(a => if f(a) then Some(a) else None)
```

So, how do we use this?

```scala
final case class Person(
  name: String,
  department: String,
)

def lookup(name: String): Option[Person] = ???
def getManager(p: Person): Option[Person] = ???

val p = lookup("John")
val pDept = lookup("John").map(_.department)
val pMan = lookup("John").flatMap(getManager)
val pManMan = lookup("John").flatMap(getManager).flatMap(getManager)
val pManManDepNoAcc = lookup("John")
  .flatMap(getManager)
  .flatMap(getManager)
  .map(_.department)
  .filter(!_.contains("Accounting"))
```

We can now chain error handling code. How do we combine multiple results?

```
final case class Person(
  name: String,
  department: String,
)

type Team = (Person, Person)

def lookup(name: String): Option[Person] = ???
def getTeam(name1: String, name2: String): Option[Team] =
```

You do it!

We can now chain error handling code. How do we combine multiple results?

```
final case class Person(
  name: String,
  department: String,
)

type Team = (Person, Person)

def lookup(name: String): Option[Person] = ???
def getTeam(name1: String, name2: String): Option[Team] =
```

```
def getTeam(name1: String, name2: String): Option[Team] =
  lookup(name1).flatMap(p1 =>
      lookup(name2).map(p2 => (p1, p2)))
```

Because nesting `flatMap`s and `map`s is done so often, Scala has syntactic sugar for it:

```
for
  aa <- a
  bb <- b
  cc <- c
  dd <- d
yield (aa + bb + cc + dd)
```

```
a.flatMap(aa =>
    b.flatMap(bb =>
        c.flatMap(cc =>
            d.map(dd =>
                aa + bb + cc + dd))))
```

This allows us to write code with `flatMap`s in a very concise and clear way.

How do we deal with old APIs? Let's say we have a function

```scala
def insuranceRate(age: Int, numberOfTickets: Int): Double
```

We get both `age` and `numberOfTickets` from an HTTP request which sadly has this data encoded as strings.

We need to parse them. `.toInt` to the rescue. But `.toInt` throws on invalid strings.

Introduce new function:

```
def Try[A](a: =>A): Option[A] =
  try Option.Some(a)
  catch
    case e: Exception => Option.None
```

Then we can write:

```
def insuranceRateS(age: String, numberOfTickets: String): Option[Double] =
  for
    a <- Try(age.toInt)
    n <- Try(numberOfTickets.toInt)
  yield insuranceRate(a, n)
```

Okay, so how do we deal with multiple values? What if we get a list of strings and should turn them into a list of integers or fail if one of those isn't valid?

```
val parsed = List("1", "2", "r").map(s => Try(s.toInt))
```

What's wrong here?

Okay, so how do we deal with multiple values? What if we get a list of strings and should turn them into a list of integers or fail if one of those isn't valid?

```
val parsed = List("1", "2", "r").map(s => Try(s.toInt))
```

What's wrong here? This has the wrong type. We get `List[Option[Int]]` but need `Option[List[Int]]`.

Exercise: Write the following function:

```
def sequence[A](as: List[Option[A]]): Option[List[A]] =
```

Exercise: Write the following function:

```
def sequence[A](as: List[Option[A]]): Option[List[A]] =
  as match
    case Nil => Option.Some(Nil)
    case h::t =>
      for
        hh <- h
        tt <- sequence(t)
      yield (hh :: tt)
```

Also possible using `foldRight` and `map2`

This leaves us with:

```scala
val parsedS = sequence(List("1", "2", "r").map(s => Try(s.toInt)))
val parsedT = traverse(List("1", "2", "r"))(s => Try(s.toInt))

def traverse[A, B](l: List[A])(f: A => Option[B]): Option[List[B]] =
  sequence(l.map(f))
```

Where `traverse` is a function which combines `map` and `sequence` because those two often show up together.

We have seen:

- how to use the `Option` data type to represent failure or success
- how to chain multiple functions which might fail
- how to keep multiple results which may be failures and combine them (`map2` or `for ... yield`).
- how to cope with old APIs
- how to cope with multiple values

The Either Datatype

The option datatype has one major drawback: You lose the information what error occurred.

To solve this, we invent a new data type:

```
enum Either[+E, +A]:
  case Left(value: E)
  case Right(value: A)
```

- A value in both cases, `Left` and `Right`
- The `Left` class normally carries the error. This is why the left type is called `E`

We also have a very familiar API:

```
enum Either[+E, +A]:
  case Left(value: E)
  case Right(value: A)

  def map[B](f: A => B): Either[E, B] = ???
  def flatMap[EE >: E, B](f: A => Either[EE, B]): Either[EE, B] = ???
  def map2[EE >: E, B, C](b: Either[EE, B])(f: (A, B) => C): Either[EE, C] = ???
```

Implement the function `map2`. Use the `for` syntax for `flatMap` and `map` for that. You can use `this` to get the current instance, just like in Java.

```
for
  aa <- a
  bb <- b
yield (aa + bb)
```

```
enum Either[+E, +A]:
  case Left(value: E)
  case Right(value: A)

  def map[B](f: A => B): Either[E, B] = ???
  def flatMap[EE >: E, B](f: A => Either[EE, B]): Either[EE, B] = ???
  def map2[EE >: E, B, C](b: Either[EE, B])(f: (A, B) => C): Either[EE, C] = ???
```

Solution **map2**

```
def map2[EE >: E, B, C](b: Either[EE, B])(f: (A, B) => C): Either[EE, C] =
  for
    a  <- this
    b1 <- b
  yield f(a,b1)
```

Why can't we have `filter(f: A => Boolean)` on `Either`, even though we had it on `Option`?

Why can't we have `filter(f: A => Boolean)` on `Either`, even though we had it on `Option`?

```
case Left(value: E)
case Right(value: A)
```

Parametricity strikes again. If the predicate doesn't match, we can't return an `Left` because we don't know what `E` is and so we can't provide a value for it.

The Either Datatype

`Either` is very similar to `Option`.

Let's go through the old problems we solved with either to see how:

Chaining Multiple Functions:

```
final case class Person(
  name: String,
  department: String,
)

type Team = (Person, Person)

def lookup(name: String): Either[String, Person] = ???
def getManager(p: Person): Either[String, Person] = ???

val p = lookup("John")
val pDept = lookup("John").map(_.department)
val pMan = lookup("John").flatMap(getManager)
val pManMan = lookup("John").flatMap(getManager).flatMap(getManager)
```

Using Multiple Results

```
def getTeam(name1: String, name2: String): Either[String, Team] =
  for
    p1 <- lookup(name1)
    p2 <- lookup(name2)
  yield (p1, p2)
```

Dealing With Old API

Defining `Try`

```
def Try[A](a: =>A): Either[Exception, A] =
  try Either.Right(a)
  catch
    case e: Exception => Either.Left(e)
```

and then using it

```
def insuranceRateS(age: String, numTickets: String): Either[Exception, Double] =
  for
    a <- Try(age.toInt)
    n <- Try(numTickets.toInt)
  yield insuranceRate(a, n)
```

```scala
val parsedS = sequence(List("1", "2", "r").map(s => Try(s.toInt)))
val parsedT = traverse(List("1", "2", "r"))(s => Try(s.toInt))
```

Yes, `traverse` and `sequence` also exist for `Either`. How to implement them will be on the exercise sheet.

We have seen:

- how to use the `Either` data type to represent failure or success
- how to chain multiple functions which might fail
- how to keep multiple results which may be failures and combine them (`map2` or `for ... yield`).
- how to cope with old APIs
- how to cope with multiple values
- how the API of `Either` is surprisingly similar to that of `Option`. We will abstract over that in future lectures.