Fuss-free data validation without using exceptions

Scala, Rust, Swift, Haskell

Franklin Chen
http://franklinchen.com/

Pittsburgh Code and Supply

April 9, 2015

Outline

Goals

- Explain concepts using fully worked-out example
- Show real code (it's all up on GitHub)
- Hope you learn something you go out and *use*

Why cover four languages?

Pittsburgh Code and Supply's polyglot advantage

- Opportunity for you to explore a new language, and compare different designs
- More efficient than giving multiple presentation about the exact same concepts

Why Scala, Rust, Swift, Haskell?

- First-class functions
- Tagged union types

But any language with first-class functions can be used:

- JavaScript, etc.

Swift is still changing

Limited Swift code examples: language and compiler not stable (Swift 1.2 was officially released *yesterday* !)

A data validation example task

From Martin Fowler's article, "Replacing Throwing Exceptions with Notification in Validations":

- Goal: create a valid event from a theater booking request
- Given: a date string that is possible null, a number of seats that is possible null
- Validate:
  - Possibly null date string
    - ★ Date string must not be null
    - ★ Date string must actually parse to a date
    - ★ Request date must not be earlier than now
  - Possibly null number of seats
    - ★ Number of seats must not be null
    - ★ Number of seats must be positive

Java code

```
public void check() {
  if (date == null)
    throw new IllegalArgumentException("date is missing");
  LocalDate parsedDate;
  try {
    parsedDate = LocalDate.parse(date);
  }
  catch (DateTimeParseException e) {
    throw new IllegalArgumentException("Invalid format for da
  }
  if (parsedDate.isBefore(LocalDate.now()))
    throw new IllegalArgumentException("date cannot be before
  if (numberOfSeats == null)
    throw new IllegalArgumentException("number of seats canno
  if (numberOfSeats < 1)
    throw new IllegalArgumentException("number of seats must
}
```

Normal execution

- A thread of execution, toward a destination
- Stack of pending operations
- When an operation is complete, pops off the stack

Exceptions considered problematic

Exceptions mean jumping up the stack

- Have to explicitly watch for and catch them
- Tedious to collect more than one error if exceptions are used
- What happens when there is concurrency?

Some languages don't have exceptions

- C
- Go
- Rust
- Swift

Railway-oriented programming

Railway-oriented programming:

- Keeping computation on the tracks.
- Cleanly handle track-switching and merging.

`null`: the unloved second track

- `null` is Tony Hoare's billion-dollar mistake, invented in 1965
- Adds a second track to every single computation involving a reference type
- Null Pointer Exceptions, seg faults

No more mention of `null` here!

- All four languages mentioned have an improvement we take as a starting point.

`Option[T]`

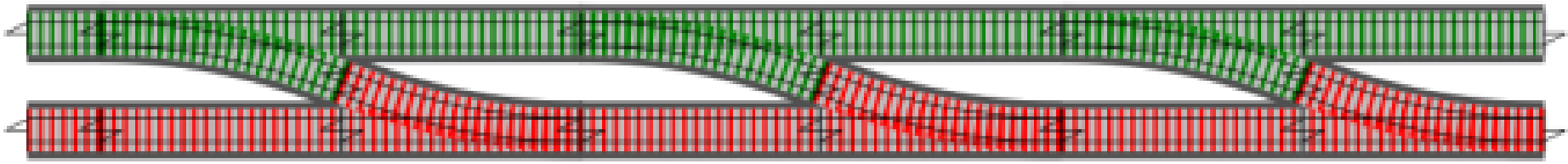
	Scala	Rust	Swift	Haskell
Type	`Option[T]`	`Option<T>`	`Optional<T>`[1]	`Maybe t`
Nonexistence	`None`	`None`	`.None`[2]	`Nothing`
Existence	`Some(x)`	`Some(x)`	`.Some(x)`[3]	`Just x`

[1] Abbreviated `T?`

[2] Abbreviated `nil`

[3] Special syntactic sugar available

Chaining computations over `Option[T]`



Example

Railway chaining values of an `Option` type

- If encountering a `None`:
  - Bail out permanently to the failure track
- Else if encountering a `Some(x)`:
  - Stay on the success track

Scala: chaining syntactic sugar for `Option[T]`

Example

"Find the winner: your best friend's oldest sister's youngest child"

```
/** Assume: bestFriend(), oldestSister(), youngestChild()
    each returns Option[Person] */
def winner(person: Person): Option[Person] = for {
  friend <- person.bestFriend()
  sister <- friend.oldestSister()
  child <- sister.youngestChild()
} yield child
```

- Scala's "`for` comprehensions" inspired by Haskell
- Generic for railway-oriented programming

Scala: non-sugar chaining for `Option[T]`

Example

"Find the winner: your best friend's oldest sister's youngest child"

```
/** Assume: bestFriend(), oldestSister(), youngestChild()
  each returns Option[Person] */
def unsweetWinner(person: Person): Option[Person] =
  person.bestFriend()    .flatMap( friend =>
  friend.oldestSister()  .flatMap( sister =>
  sister.youngestChild()
  ))
```

- Sugar is preprocessed to this code *before compilation*

Swift: chaining syntactic sugar for T?

Example

"Find the winner: your best friend's oldest sister's youngest child"

```
/** Assume: bestFriend(), oldestSister(), youngestChild()
   each return Person? */
func winner(person: Person) -> Person? = {
  return person.bestFriend()?.
                oldestSister()?.
                youngestChild()
}
```

- Swift's special chaining sugar
- Specific to `Optional` only!

Rust: no syntactic sugar for `Option<T>`

Example

"Find the winner: your best friend's oldest sister's youngest child"

```
/// Assume: best_friend(), oldest_sister(), youngest_child()
/// each returns Option<Person>
fn winner(person: Person) -> Option<Person> {
  person.best_friend()       .and_then( |friend|
     friend.oldest_sister() .and_then( |sister|
     sister.youngest_child()
  ))
}
```

- Rust: no syntactic sugar
- Deprecate use of `Option` for error signaling!
- Sugar provided for what Rust recommends instead (next topic)

Haskell: chaining syntactic sugar for `Maybe t`

Example

"Find the winner: your best friend's oldest sister's youngest child"

```haskell
-- | Assume: bestFriend, oldestSister, youngestChild
-- each returns 'Maybe Person'
winner :: Person -> Maybe Person
winner person = do
  friend <- person & bestFriend
  sister <- friend & oldestSister
  sister & youngestChild
```

- Haskell's "`do` notation" invented in 1993

`Option` considered harmful

Warning

An `Option`-chained failure gives *zero information* about *why* and *where* something failed!

When `winner(person)` returns `None`:

- Did the person's best friend's oldest sister not have any children?
- Or did the person's best friend not have any sisters?
- Or did the person not have any friends?

Knowledge is power

"Enquiring minds want to know!"

`Either[E, T]`

	Scala	Swift	Haskell
Type	`Either[E, T]`[4]	`Either<E, T>`[5]	`Either e t`
Bad	`Left(e)`	`.Left(e)`	`Left e`
Good	`Right(x)`	`.Right(x)`	`Just x`

	Rust
Type	`Result<T, E>`[6]
Bad	`Err(e)`
Good	`Ok(x)`

[4] The Scalaz library provides an improved version called `E \/ T` we will prefer

[5] `Either<E, T>` *not* in Swift's standard library, but provided in Swiftx

[6] Rust chose a more informative name, and placed success type param `T` first

Converting between `Option[T]` to `E \/ T`

Conversion is simple

Examples using Scalaz:

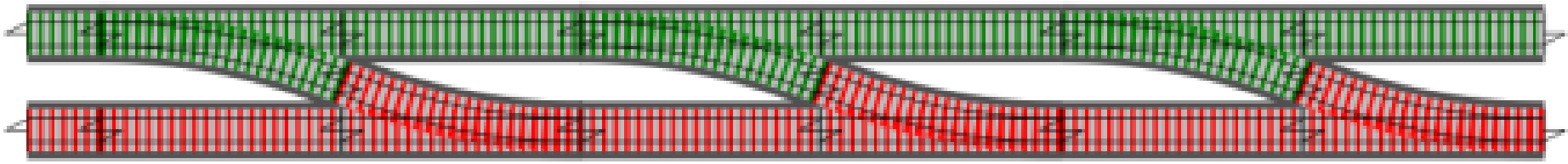
`E \/ T` to `Option` "Forget" an error by replacing it with `None`:

```
optX = eitherX.toOption
```

`Option[T]` to `E \/ T` "Add" an error by replacing `None` with an error:

```
eitherX = optX.toRightDisjunction("some error")
```

Chaining computations over `Either[E, T]`



Exact same concept as with `Option[T]`.

Scala: chaining syntactic sugar for `E \/ T`

Example

"Find the winner: your best friend's oldest sister's youngest child"

```
/** Assume: bestFriend(), oldestSister(), youngestChild()
    each returns MyError \/ Person */
def winner(person: Person): MyError \/ Person = for {
  friend <- person.bestFriend()
  sister <- friend.oldestSister()
  child <- sister.youngestChild()
} yield child
```

- Exact same code as with `Option[T]`!
- We are using Scalaz library's disjunction `E \/ T` because standard Scala's `Either[E, T]` has limitations
- Genericity of railway-oriented code: large topic in itself

Swift: no syntactic sugar for `Either<E, T>`

Example

"Find the winner: your best friend's oldest sister's youngest child"

```
/** Assume: bestFriend(), oldestSister(), youngestChild()
   each return Either<MyError, Person> */
func winner(person: Person) -> Either<MyError, Person> = {
  return person.bestFriend() .flatMap { friend in
    friend.oldestSister()    .flatMap { sister in
    sister.youngestChild()
  }}
}
```

- Use Swiftx library
- Swift does *not* have general railway-oriented syntactic sugar

Rust: chaining syntactic sugar for `Result<T, E>`

Example

"Find the winner: your best friend's oldest sister's youngest child"

```
/// Assume: best_friend(), oldest_sister(), youngest_child()
/// each returns Result<Person, MyError>
fn winner(person: Person) -> Result<Person, MyError> {
  let friend = try!(person.best_friend());
  let sister = try!(friend.oldest_sister());
  sister.youngest_child()
}
```

Rust

- Can use exactly the same non-sugar code as for `Option<T>` if wanted
- Standard library has macro `try!` for use with `Result<T, E>`
- *Does not have exceptions*, so ease of use of `Result<T, E>` is critical!

Haskell: chaining syntactic sugar for `Either e t`

Example

"Find the winner: your best friend's oldest sister's youngest child"

```
-- | Assume: bestFriend, oldestSister, youngestChild
-- each returns 'Either MyError Person'
winner :: Person -> Either MyError Person
winner person = do
  friend <- person & bestFriend
  sister <- friend & oldestSister
  sister & youngestChild
```

- Exact same code as with `Maybe t`!

Constructors considered harmful

- Constructors in many languages do not return a result, so failures are indicated by either:
  - Throwing an exception
  - Return `null` and setting an error object elsewhere
- Alternative: "factory method" that returns an `Either[SomeError, ThingToCreate]`

Constructing Seat: Scala

```scala
case class Seats private(val num: Int) extends AnyVal

object Seats {
  sealed trait Error
  case class BadCount(num: Int) extends Error {
    override def toString =
      s"number of seats was $num, but must be positive"
  }

  def make(num: Int): BadCount \/ Seats = {
    if (num < 0)
      BadCount(num).left
    else
      Seats(num).right
  }
}
```

Notes on clean module design

- The constructor for `Seats`:
  - Is trivial, not bloated: just saves off parameters into fields
  - Is private, to guarantee only approved factory methods can call it
- Errors:
  - Each module defines its own set of errors as a union type, here called `Error`
  - (Here only one, `BadCount`)
- Factory methods:
  - Each `Seats` factory method returns `Error \/ Seats`

Constructing Seats: Rust

```
pub struct Seats {
  num: i32 // private
}

pub enum Error {
  BadCount(i32)
}

impl Seats {
  pub fn make(num: i32) -> Result<Seats, Error> {
    if num <= 0 {
      Err(Error::BadCount(num))
    } else {
      Ok(Seats { num: num })
    }
  }
}
```

Constructing Seats: Haskell

```haskell
-- | Wrapper around 'Int' that ensures always positive.
newtype Seats = Seats { getNum :: Int }

data Error = BadCount Int -- ^ attempted number of seats

instance Show Error where
  show (BadCount seats) = "number of seats was " ++
     show seats ++ ", but must be positive"

-- | Smart constructor for 'Seats' that
-- ensures always positive.
make :: Int -> Validation Error Seats
make seats | seats < 0 = Failure $ BadCount seats
           | otherwise = Success $ Seats seats
```

Constructing Seats: Swift

No example: because didn't want to dive into the flaws of

- failable initializers
- Cocoa factory methods

`Either` considered insufficient

Warning

An `Either`-chained failure returns information *only* about the first failure ("fail fast").
What if we want to chain multiple result-returning computations while collecting *all* failures along the way?

Examples:

- Booking request example: date and number of seats may both be invalid; we want to know about both failures
- Facebook: *concurrently* accesses many data sources, collecting all failures[7]

The goal

"Enquiring minds want to know *everything* !"

[7] Facebook open-sourced their Haskell library Haxl for this

Introduction

Validation libraries:

Scala	Scalaz library
Rust	I wrote my own library, may generalize and publish it
Swift	Swiftz (superset of Swiftx) is based on Scalaz
Haskell	validation library

Differences in naming and design

Because of differences, we will use Scalaz terminology.

Definition

`Validation[E, T]`

Continue to track success/failure in an `Either[E, T]`-like object:

- Leave *sequential* railway-oriented computation model
- Adopt *parallel* computation model

`ValidationList[E, T]`

Just a synonym for `Validation[List[E], T]`:

- Replace individual failure with a collection of failures

Annoying names

In the Scalaz library, the real name for `ValidationList[E, T]` is `ValidationNel[E, T]`:

- "Nel" stands for `NonEmptyList`
- `ValidationNel[E, T]` is a synonym for `Validation[NonEmptyList[E], T]`

All the different `Either` types

	`Either[E, T]`	`E \/ T`	`Validation[E, T]`
Bad	`Left(e)`	`-\/(e)`	`Failure(e)`
Good	`Right(x)`	`\/-(x)`	`Success(x)`
Purpose	symmetric, neutral	railway-oriented	accumulation

Conversion among them is simple: just replacing the tag.

BookingRequest types: Scala

```scala
case class BookingRequest private(
  val date: Date,
  val seats: Seats
)

sealed trait Error

// These wrap errors from other modules.
case class DateError(e: Date.Error) extends Error
case class SeatsError(e: Seats.Error) extends Error

// Our additional errors.
case class DateBefore(date1: Date, date2: Date) extends Error
case class Missing(label: String) extends Error
```

BookingRequest types: Rust

```
pub struct BookingRequest {
  date: Date,
  seats: seats::Seats
}

pub enum Error {
  DateError(date::Error),
  SeatsError(seats::Error),
  DateBefore(Date, Date),
  Missing(String)
}
```

BookingRequest types: Haskell

```haskell
data BookingRequest =
  BookingRequest { getDate :: Date.Date
                 , getSeats :: Seats.Seats
                 }

data Error =
    DateError Date.Error
  | SeatsError Seats.Error
  | Missing String          -- ^ label
  | DateBefore Date.Date -- ^ date that was attempted
               Date.Date -- ^ the current date at attempt
```

Seats creation

```
def makeSeats(optSeats: Option[Int]):
    Error \/ Seats = for {
  num <- optSeats.toRightDisjunction(Missing("seats"))
  validSeats <- Seats.make(num).leftMap(SeatsError)
} yield validSeats
```

Use chaining:

- Convert the `Option[Int]` to `Error \/ Int`
- Use `leftMap` to lift from `Seats.Error \/ Seats` to `Error \/ Seats`

Date validation against now

```
def timelyBookingDate(date: Date, now: Date):
    DateBefore \/ Date = {
  if (!date.isBefore(now))
    date.right
  else
    DateBefore(date, now).left
}
```

A validator that just passes along what comes in if it's OK.

Date creation

```
def makeTimelyBookingDate(now: Date,
  optDateString: Option[String]): Error \/ Date = for {
  dateString <- optDateString.
                toRightDisjunction(Missing("date"))
    date <- Date.parse(dateString).leftMap(DateError)
    timelyDate <- timelyBookingDate(date, now)
} yield timelyDate
```

Use chaining:

- First, get the requested date
- Then validate that against now

BookingRequest factory method

```
def make(
  now: Date,
  optDateString: Option[String],
  optSeats: Option[Int]
): ValidationNel[Error, BookingRequest] = {
  val combinedBuilder =
    makeTimelyBookingDate(now, optDateString).
      validation.toValidationNel |@|
    makeSeats(optSeats).
      validation.toValidationNel

  combinedBuilder(BookingRequest(_, _))
}
```

Combination of techniques:

- Sequential: each of `Seats`, `Date` validated creation is railway-oriented
- Parallel: in principle, the combiner can be parallelized

Technical notes

Concepts covered, in order from more specific to more general:

- Sequential, railway-oriented programming is called *monadic*: `Option` and `Either` are the simplest monads; there is a vast number of more complex monads
- Parallelizable composition is called *applicative*: `Validation` is an applicative functor
- Parallelizable combination is called *monoidal*: `List` is a monoid
- (`NonEmptyList` is a semigroup, a monoid without identity element)

Conclusion

Summary:

- We saw how to break down a messy validation problem
- Design with types to reflect intent
- Use result objects that track failures, force error handling
- Factory methods to create only valid objects
- Sequential railway-oriented chaining of validator functions
- Parallel computations can be expressed by separating independent components
- Minimizing `if`/`else`-style programming feels good!

Slides and code

Slides in source and PDF form:

- https://github.com/FranklinChen/data-validation-demo

Complete code with tests run on Travis CI:

Scala	https://github.com/FranklinChen/data-validation-demo-scala
Rust	https://github.com/FranklinChen/data-validation-demo-rust
Haskell	https://github.com/pittsburgh-haskell/data-validation-demo-haskell
Swift	https://github.com/FranklinChen/data-validation-demo-swift