PROGRAMMATION FONCTIONNELLE

Fabrice Legond-Aubry et Pascal Poizat

Juillet-Août 2024

- Introduction
- Pureté
- Immutabilité
- Ordre supérieur
- Programmes séquentiels
- Absence de valeur et erreurs
- Types de données algébriques
- Effets de bord
- Flux et évaluation paresseuse
- Concurrence
- Test
- Synthèse des notations

ABSENCE DE VALEUR ET ERREURS

EVÈNEMENTS HISTORIQUES

on a un type pour les évènements historiques

```scala
case class Event(name: String, start: Int, end: Int)
```

- le nom doit être une chaîne non vide
- la fin doit être inférieure à 3000
- le début doit être inférieur ou égal à la fin

on souhaite écrire une fonction de parsing

```scala
def parse(name: String, start: Int, end: Int): Event =
  if (name.size > 0 && end < 3000 && start <= end)
    Event(name, start, end)
  else
    null
```

PROBLÈMES

Scala
```scala
def parse(name: String, start: Int, end: Int): Event =
    if (name.size > 0 && end < 3000 && start <= end)
        Event(name, start, end)
    else
        null
```

Scala
```scala
parse("Apollo Program", 1961, 1972)
-> Event("Apollo Program", 1961, 1972)

parse("Apollo Program", 1961, 1972).name
-> "Apollo Program"

parse("", 1939, 1945).name
-> java.lang.NullPointerException: ...
```

- concepts entremelés au niveau de la vérification
- parse ment (absence de valeur), elle n'est pas pure T. Hoare "one-billion dollar mistake"

OPTION

- *Option[A] = Some[A]|None*
- on le retrouve sous différents noms
  (`Option` en Scala, `Maybe` en Haskell, `Optional` en Java)

Scala
```scala
def parse(name: String, start: Int, end: Int): Option[Event] =
    if (name.size > 0 && end < 3000 && start <= end)
        Some(Event(name, start, end))
    else
        None
```

Scala
```scala
parse("Apollo Program", 1961, 1972)
-> Some(Event("Apollo Program", 1961, 1972))

parse("", 1939, 1945)
-> None

parse("Apollo Program", 1961, 1972).name
-> ... (value name is not a member of Option[Event])
```

OPTION

- on peut utiliser certaines fonctions de l'API d'`Option`
  - *isEmpty* : *Option*[*A*] ↪ () → *Boolean*
  - *isDefined* : *Option*[*A*] ↪ () → *Boolean*
  - *get* : *Option*[*A*] ↪ () → *A* (!!)
  - *getOrElse* : *Option*[*A*] ↪ *A* → *A*

Scala

```
parse("Apollo Program", 1961, 1972).isDefined
-> true

parse("Apollo Program", 1961, 1972).get.name
-> Apollo Program

parse("", 1939, 1945).isDefined
-> false

parse("", 1939, 1945).getOrElse(Event("Unknown name", 0, 0))
-> Event(Unknown name, 0, 0)
```

EN JAVA

création avec `Optional.of`, `Optional.ofNullable` et `Optional.empty`

Java

```java
static Optional<Event> parse(String name, int start, int end) {
    if (!name.isEmpty() && end < 3000 && start <= end)
        return Optional.of(new Event(name, start, end));
    else
        return Optional.empty();
}
```

`isEmpty`, `isPresent`, `get` et `orElse` dans l'API

Java

```java
Optional<Event> ev1 = Event.parse("Apollo Program", 1961, 1972);
Optional<Event> ev2 = Event.parse("", 1939, 1945);
ev1.isPresent(); // true
ev1.get().name(); // Apollo Program
ev2.isPresent(); // false
ev2.orElse(new Event("Unknown name", 0, 0));
```

OPTION

- mais on dispose aussi de `map` et `flatMap`
  - *map* : *Option*[*A*] ↪ (*A*→*B*) ↪ *Option*[*B*]
  - *flatMap* : *Option*[*A*] ↪ (*A*→*Option*[*B*]) ↪ *Option*[*B*]

Scala

```
def informe(base: Map[String, String])(event: String): Option[String] = base.get(
val myWiki: Map[String, String] = Map("Apollo Program" -> "programme spatial amér
                                      "Ariane" -> "lanceur européén")
val informeWithMyWiki = informe(myWiki)

parse("Apollo Program", 1961, 1972).map(_.name)
-> Some(Apollo Program)
parse("", 1939, 1945).map(_.name)
-> None
parse("Apollo Program", 1961, 1972).map(_.name).flatMap(informeWithMyWiki)
-> Some(programme spatial américain)
parse("WWII", 1939, 1945).map(_.name).flatMap(informeWithMyWiki)
-> None
parse("", 1939, 1945).map(_.name).flatMap(informeWithMyWiki)
-> None
```

EN JAVA

Java

```java
static Function<Map<String, String>, Function<String, Optional<String>>> informe
    map -> event -> Optional.ofNullable(map.get(event));
```

`map` et `flatMap` dans l'API

Java

```java
final Map<String, String> myWiki = Map.of(
    "Apollo Program", "programme spatial américain",
    "Ariane", "lanceur européén"
);
Function<String, Optional<String>> informeWithMyWiki = informe.apply(myWiki);
Optional<Event> ev1 = Event.parse("Apollo Program", 1961, 1972);
Optional<Event> ev3 = Event.parse("WWII", 1939, 1945);
Optional<Event> ev2 = Event.parse("", 1939, 1945);
System.out.println(ev1.map(Event::name)); // Optional[Apollo Program]
System.out.println(ev2.map(Event::name)); // Optional.empty
System.out.println(ev1.map(Event::name).flatMap(informeWithMyWiki)); // Optional[
System.out.println(ev2.map(Event::name).flatMap(informeWithMyWiki)); // Optional.
System.out.println(ev3.map(Event::name).flatMap(informeWithMyWiki)); // Optional.
```

PARSING COMME PIPELINE

idée du pipeline :

- une fonction pour chaque validation
- une comprehension pour combiner les validations et créer l'évènement

Scala

```scala
def validateName(name: String): Option[String] =
    if (name.size > 0) then Some(name) else None
def validateStart(start: Int, end: Int): Option[Int] =
    if (start <= end) then Some(start) else None
def validateEnd(end: Int): Option[Int] =
    if (end < 3000) then Some(end) else None

def parseValidation(name: String, start: Int, end: Int): Option[Event] =
    for {
        validName <- validateName(name);
        validStart <- validateStart(start, end)
        validEnd <- validateEnd(end)
    } yield Event(validName, validStart, validEnd)
```

PARSING COMME PIPELINE EN JAVA

Java

```java
static Optional<String> validateName(String name) {
    return !name.isEmpty() ? Optional.of(name) : Optional.empty();
}
static Optional<Integer> validateStart(int start, int end) {
    return start <= end ? Optional.of(start) : Optional.empty();
}
static Optional<Integer> validateEnd(int end) {
    return end < 3000 ? Optional.of(end) : Optional.empty();
}

static Optional<Event> parseValidation(String name, int start, int end) {
    return validateName(name).flatMap(n ->
            validateStart(start, end).flatMap(s ->
                validateEnd(end).map(e ->
                    new Event(n, s, e))));
}
```

COMMENT SAVOIR POURQUOI CELA NE MARCHE PAS ?

Scala

```
parseValidation("Apollo Program", 1961, 1972);
parseValidation("WWII", 1939, 1945);
parseValidation("", 1939, 1945);
parseValidation("", 1945, 1939);
```

donne :

Scala

```
Some(Event(Apollo Program,1961,1972))
Some(Event(WWII,1939,1945))
None
None
```

EITHER

- *Either*[*A*,*B*] = *Left*[*A*]|*Right*[*B*] (Right = “all is right” = OK)
- on le retrouve
  (`Either` en Scala, `Either` en Haskell, pas en Java)

Scala

```scala
def parseE(name: String, start: Int, end: Int): Either[String, Event] =
    if (name.size > 0 && end < 3000 && start <= end)
        Right(Event(name, start, end))
    else
        Left("un truc ne va pas")
```

Scala

```scala
parseE("Apollo Program", 1961, 1972);
-> Right(Event(Apollo Program,1961,1972))
parseE("WWII", 1939, 1945);
-> Right(Event(WWII,1939,1945))
parseE("", 1939, 1945);
-> Left(un truc ne va pas)
parseE("", 1945, 1939);
-> Left(un truc ne va pas)
```

EITHER

comme pour `Option`, avec `Either`

- on a une API spécifique
  - `isLeft`, `isRight`, `left`, `right`, `getOrElse`
- mais aussi une API commune
  - `map` et `flatMap`

exercice : reprendre le code avec `Option` en `Either`

EITHER ET PIPELINE

Scala

```scala
def validateNameE(name: String): Either[String, String] =
    if (name.size > 0) then Right(name) else Left("Invalid name")
... // même chose pour validateStartE et validateEndE
def parseValidationE(name: String, start: Int, end: Int): Either[String, Event] =
    for {
        validName <- validateNameE(name);
        validStart <- validateStartE(start, end)
        validEnd <- validateEndE(end)
    } yield Event(validName, validStart, validEnd)
```

Scala

```scala
parseValidationE("Apollo Program", 1961, 1972);
-> Right(Event(Apollo Program,1961,1972))
parseValidationE("WWII", 1939, 1945);
-> Right(Event(WWII,1939,1945))
parseValidationE("", 1939, 1945);
-> Left(Invalid name)
parseValidationE("", 1945, 1939);
-> Left(Invalid name) // la première erreur
```

APPORTS VAVR.IO

librairie fonctionnelle pour Java 8+

documentation ici, API ici et code ici

```
dependencies {
    implementation "io.vavr:vavr:0.10.4"
}
```

- structures de données purement fonctionnelles
  - accès direct à `sorted`, `filter`, `map`, `flatMap` et +
  - version paresseuse : `Stream`
- tuples de 1 à 8 arguments
- fonctions de 0 à 8 arguments
  - composition, lifting, curryfication, application partielle et +
- `Option`, `Try`, `Either`, `Validation` et +

PARSING COMME PIPELINE AVEC VAVR.IO

```java
static Validation<String, String> validateName(String name) {
    return !name.isEmpty()
        ? Validation.valid(name)
        : Validation.invalid("nom incorrect");
}
... // idem pour validateStart et validateEnd
static Validation<Seq<String>, EventWithVavr> parseValidation(String name, int st
    return validateName(name)
            .combine(validateStart(start, end))
            .combine(validateEnd(end))
            .ap(EventWithVavr::new);
}
```

- *combine* : *Val* < *E*, *T1* > ↪ *Val* < *E*, *T2* > → *Builder* < *E*, *T1*, *T2*>
- *combine* : *Builder* < *E*, *T1*, *T2* > ↪ *Val* < *E*, *T3* > → *Builder3* < *E*, *T1*, *T2*, *T3*>
- *ap* : *Builder3* < *E*, *T1*, *T2*, *T3* > ↪ *Function3* < *T1*, *T2*, *T3*, *R* > → *Val* < *Seq* < *E*>,*R*>

```
parseValidation("", 1945, 1939);
-> donne bien les deux informations d'erreur
```

VALIDATION EN SCALA

- non disponible nativement en Scala
  - `Validated` de Cats et `Validation` de ZIO
  - `libraryDependencies += "org.typelevel" %% "cats-effect" % "3.5.4"`

Scala

```
def validateNameV(name: String): Validated[String, String] =
    if (name.size > 0) then Validated.valid(name) else Validated.invalid("Invalid
... // idem pour validateStartV et validateEndV
def parseValidationV(name: String, start: Int, end: Int): Validated[EventValidati
    for {
        validName <- validateNameV(name);
        validStart <- validateStartV(start, end)
        validEnd <- validateEndV(end)
        event <- Validated.valid(Event(validName, validStart, validEnd))
    } yield event
```

ne marche pas

car `Validated` n'a pas de méthode `flatMap`

VALIDATION EN SCALA

définition d'un type plus synthétique (optionel)

Scala
```
type ValidationResult[T] = ValidatedNel[String, T]
// idem que type ValidationResult[T] = Validated[NonEmptyList[String], T]
```

puis

Scala
```
def validateNameV2(name: String): ValidationResult[String] =
    if (name.size > 0) then Validated.validNel(name) else Validated.invalidNel("I

// nécessite aussi import cats.implicits._
def parseValidationV(name: String, start: Int, end: Int): ValidationResult[Event]
    (
      validateNameV2(name),
      validateStartV2(start, end),
      validateEndV2(end)
    ).mapN((name, start, end) => Event(name, start, end))
```

Scala
```
parseValidationV("", 1945, 1939);
-> Invalid(NonEmptyList(Invalid name, Invalid start))
```

LE PETIT ZOO DES TYPES

- Semigroup, Monoid, Functor, Applicative, Monad
- Option, Either, Validation
- ce “Zoo” est celui de Cats
  on le retrouve dans d’autres langages ou librairies
- une petite liste (il y en a d’autres)
  - Java (~Option)
  - Vavr (Java) (Option, Either, Validation)
  - Scala (Option, Either)
  - Cats (Scala) (tous)
  - Haskell (tous)

LE PETIT ZOO DES TYPES (1)

- `Semigroup[A]`
  - *combine* : $(A,A) \rightarrow A$
  - associativité : $combine(x,combine(y,z)) = combine(combine(x,y),z)$

on peut définir

Scala
```scala
implicit val intAdditionSemigroup: Semigroup[Int] = _ + _
```

et ensuite

Scala
```scala
val x = 1; val y = 2; val z = 3

Semigroup[Int].combine(x, y) // res1: Int = 3
Semigroup[Int].combine(x, Semigroup[Int].combine(y, z)) // res2: Int = 6
Semigroup[Int].combine(Semigroup[Int].combine(x, y), z) // res3: Int = 6
```

mais aussi pour `List[_]`, `Set[_]`, `Map[_]`, ...

LE PETIT ZOO DES TYPES (2)

- `Monoid[A]` qui étend `Semigroup[A]`
  - *empty* : *A*
  - élément neutre : *combine(x,empty) = combine(empty,x) = x*

on peut définir

Scala

```scala
def combineAll[A: Monoid](as: List[A]): A =
    as.foldLeft(Monoid[A].empty)(Monoid[A].combine)
```

utilisable pour tout (toute instance de) `Monoid`

Scala

```scala
combineAll(List(1, 2, 3))
// res3: Int = 6
combineAll(List("hello", " ", "world"))
// res4: String = "hello world"
combineAll(List(Map('a' -> 1), Map('a' -> 2, 'b' -> 3), Map('b' -> 4, 'c' -> 5)))
// res5: Map[Char, Int] = Map('b' -> 7, 'c' -> 5, 'a' -> 3)
combineAll(List(Set(1, 2), Set(2, 3, 4, 5)))
// res6: Set[Int] = HashSet(5, 1, 2, 3, 4)
```

LE PETIT ZOO DES TYPES (3)

- `Functor[F[_]]`
  - *map* : *F*[*A*] → (*A*→*B*) → *F*[*B*] ou *lift* : (*A*→*B*) → *F*[*A*] → *F*[*B*]
  - si on a `map` on peut définir *lift*(*f*) = *fa* → *map*(*fa*)(*f*)
  - composition : *fa.map*(*f*).*map*(*g*) = *fa.map*(*f.andThen*(*g*))
  - identité : *fa.map*(*x*→*x*) = *fa*
- `F` est souvent appelé effet
  - exemples : `List[_]` ou `Option[_]`
- les `Functor` se composent
  - si `F[_]` et `G[_]` sont des `Functor` alors `F[G[_]]` aussi
  - exemples : `Option[List[_]]` ou `List[Option[_]]`

Scala

```scala
val listOption = List(Some(1), None, Some(2))

Functor[List].compose[Option].map(listOption)(_ + 1)
// res1: List[Option[Int]] = List(Some(value = 2), None, Some(value = 3))
```

LE PETIT ZOO DES TYPES (4)

- `Applicative[F[_]]` qui étend `Functor[F]`
  - *pure* : *A* → *F*[*A*]
  - *ap* : *F*[*A*→*B*] → *F*[*A*] → *F*[*B*] ou *product* : (*F*[*A*],*F*[*B*]) → *F*[(*A*,*B*)]
  - on peut définir *map*(*fa*)(*f*) = *ap*(*pure*(*f*))(*fa*)
  - associativité : *fa*.*product*(*fb*).*product*(*fc*) = *fa*.*product*(*fb*.*product*(*fc*)).*map*{*case*(*a*,(*b*,*c*)) → ((*a*,*b*),*c*)}
  - identités : *pure*(()).*product*(*fa*).*map*(_._2) = *fa* et *fa*.*product*(*pure*(())).*map*(_._1) = *fa*

LE PETIT ZOO DES TYPES (5)

- `Monad[F[_]]` qui étend `Applicative[F]`
  - *flatten* : *F*[*F*[*A*]] → *F*[*A*] (indirectement on a *flatMap*)

LE PETIT ZOO DES TYPES (6)

- `Option` et `Either` sont des `Monad`
  - on peut utiliser `map` et `flatMap`
- `Validated` est uniquement `Applicative`
  - on peut utiliser `map` (`mapN`) mais pas `flatMap`

RÉSUMÉ

- the “one billion dollar mistake”
- types `Option` et `Either`, et leur APIs
- pipelining et types `Option` et `Either`
- vavr.io et type `Validation`
- cats et type `Validated`
- initiation au “petit zoo des types”