# Cats

Paradigmi di programmazione e sviluppo

Andrea Rettaroli
0000977930
andrea.rettaroli@studio.unibo.it

1 Maggio 2023

# Indice

# Elenco delle figure

# Capitolo 1

# Introduzione

Il progetto che si intende sviluppare è un'indagine accurata e approfondita sulla libreria **Cats**, sia dal punto di vista teorico che pratico. Lo scopo principale è quello di evidenziare i motivi per cui la libreria che fornisce astrazioni per linguaggi funzionali nel linguaggio Scala viene utilizzata. Inoltre, si presta particolare attenzione ai meccanismi relativi alla concorrenza, all'I/O e alla gestione delle risorse, utilizzando il supporto di Cats-Effect, una libreria funzionale per la gestione dei side-effect e delle operazioni asincrone.

La libreria in questione offre numerosi vantaggi, come la capacità di gestire in modo efficiente il multithreading e l'asincronia, nonché di garantire la scalabilità delle applicazioni. Questo la rende una scelta ideale per la realizzazione di applicazioni distribuite e altamente concorrenti. Grazie all'utilizzo di Cats-Effect, la libreria offre una grande flessibilità nella gestione delle risorse, permettendo di ottimizzare le prestazioni e di evitare la comparsa di effetti collaterali indesiderati.

Per dimostrare l'utilità e la versatilità di questa libreria, il progetto prevede la presentazione di numerosi esempi di codice, illustrando le funzionalità principali della libreria e mostrando come queste possano essere utilizzate per risolvere problemi comuni nello sviluppo di applicazioni. Verranno condotti esperimenti più approfonditi, che consentiranno di esplorare le funzionalità avanzate della libreria e di comprendere meglio le modalità di utilizzo.

Infine, come parte del progetto, verranno realizzate delle RestAPI con Cats-Effect, dimostrando così come la libreria possa essere utilizzata in modo pratico per lo sviluppo di applicazioni web. Le RestAPI saranno una dimostrazione pratica delle capacità della libreria. In sintesi, il progetto si propone di approfondire la libreria in questione, mostrandone le funzionalità principali e dimostrando come possa essere utilizzata per risolvere problemi comuni e caratteristici nello sviluppo di applicazioni.

# Capitolo 2

# Cats

## 2.1 Cats

La libreria Cats è nata nel 2014, grazie all'iniziativa di un gruppo di sviluppatori di software funzionale, tra cui Michael Pilquist, Travis Brown, Lars Hupel e altri. Essi hanno avvertito la necessità di creare una libreria che potesse fornire un'implementazione consistente e componibile dei concetti fondamentali della programmazione funzionale.
Cats è stata ispirata dalla libreria Scalaz, una libreria di supporto per la programmazione funzionale in Scala, sviluppata da Tony Morris e dagli altri membri della comunità Scalaz. Tuttavia, rispetto a Scalaz, Cats è stata progettata con un'attenzione maggiore alla modularità e alla compatibilità con altre librerie di Scala.
Il nome "Cats" è un acronimo di "Category Theory Scala", in riferimento alla teoria delle categorie, una branca della matematica che fornisce un linguaggio formale per la descrizione di concetti astratti e relazioni tra di essi. La teoria delle categorie è stata una fonte di ispirazione per la progettazione di Cats, in quanto essa offre un'astrazione potente e componibile per i concetti fondamentali della programmazione funzionale.
Negli anni successivi alla sua nascita, Cats ha avuto un crescente successo all'interno della comunità di sviluppatori Scala, diventando una delle librerie più utilizzate per la programmazione funzionale. Grazie alla sua architettura modulare e alla compatibilità con altre librerie di Scala, essa ha consentito lo sviluppo di applicazioni altamente performanti, robuste e scalabili, permettendo di sfruttare al massimo le potenzialità della programmazione funzionale.
Ad oggi, Cats è una libreria molto utilizzata dalle grandi aziende di sviluppo software che adottano la programmazione funzionale in Scala. Ad esempio, Twitter utilizza Cats all'interno della propria infrastruttura di servizi, sfruttando le funzionalità di concorrenza e di gestione delle risorse per sviluppare applicazioni altamente performanti e scalabili. In particolare, Twitter ha sviluppato una libreria di supporto chiamata "Finagle", che si basa su Cats e su

altre librerie di Scala, per fornire un'architettura di servizi distribuiti altamente efficiente e affidabile.
Anche la società di consulenza tecnologica Netflix utilizza Cats all'interno della propria piattaforma di streaming video, sfruttando le funzionalità di gestione delle risorse e di concorrenza per garantire prestazioni elevate e stabili.
Oltre alle grandi aziende, anche molte startup e imprese di medie dimensioni utilizzano Cats per lo sviluppo di applicazioni web, servizi backend e strumenti di analisi dati. Grazie alla sua architettura modulare e alla flessibilità, Cats è in grado di soddisfare le esigenze di una vasta gamma di applicazioni e di ambienti di sviluppo.
Essa fornisce un'ampia gamma di funzionalità e costrutti, che consentono di scrivere codice più conciso, espressivo e robusto permettendo astrazioni per la programmazione funzionale fornendo supporti per la gestione dell'I/O e della concorrenza. Una delle funzionalità principali di Cats è la gestione degli side-effect e delle operazioni asincrone, che è ottenuta attraverso l'utilizzo di tipi di dati funzionali come i Funtori, i Monoidi, le Applicative che approfondiremo in seguito. Questi tipi di dati consentono di gestire i side-effect e le operazioni asincrone in modo sicuro e componibile. Cats offre una serie di costrutti e funzionalità utili per la gestione delle collezioni di dati, tra cui mappe, insiemi, sequenze e stream. Questi costrutti consentono di manipolare le collezioni di dati in modo elegante e funzionale, fornendo numerosi metodi e funzioni per la trasformazione, l'ordinamento e l'aggregazione dei dati. Inoltre, fornisce anche una serie di funzionalità per la gestione delle eccezioni, compresa la gestione degli errori e la loro propagazione in modo sicuro e componibile. La libreria offre un supporto completo per la programmazione generica, consentendo di scrivere codice parametrico e riutilizzabile. Cats contiene un'ampia gamma di funzionalità per la programmazione asincrona, tra cui il supporto per la gestione degli eventi, la gestione delle risorse e la concorrenza. Queste funzionalità consentono di scrivere applicazioni altamente performanti e scalabili, sfruttando al massimo le capacità di Scala e della programmazione funzionale.
In sintesi, la libreria Cats offre una vasta gamma di funzionalità e costrutti, che consentono di scrivere codice più conciso, espressivo e robusto. Grazie alla sua architettura funzionale, essa è particolarmente adatta per lo sviluppo di applicazioni asincrone, distribuite e altamente concorrenti.
Le seguenti sezioni mostrano i costrutti matematici di programmazione funzionale forniti da Cats.

## 2.2 Costrutti Matematici

Prima di entrare nel dettaglio di Cats e successivamente Cats-effect è necessario definire brevemente i principali costrutti matematici che vengono utilizzati soprattutto da Cats: semigroup, monoidi, monadi e funtori. Tutti costrutti devono aderire a delle laws per essere definiti in modo opportuno.

### 2.2.1 Laws

Concettualmente, tutte le classi di tipo sono dotate di leggi. Queste leggi vincolano le implementazioni per un dato tipo e possono essere sfruttate e utilizzate per ragionare sul codice generico.

### 2.2.2 Semigroup

Si definisce come:

```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}
```

Un semigruppo per un certo tipo A ha una singola operazione combine, che prende due valori di tipo A, e restituisce un valore di tipo A. Questa operazione deve essere garantita come associativa. Vale a dire che:

```
((a combine b) combine c)
```

Deve essere uguale a:

```
(a combine (b combine c))
```

Associatività significa che la seguente uguaglianza deve valere per qualsiasi scelta di x, y, e z.

```
combine(x, combine(y, z)) = combine(combine(x, y), z)
```

Infatti supponendo di usare una combine di Int avremmo che:

```
val x = 1
val y = 2
val z = 3
combine(x, combine(y, z)) == combine(combine(x, y), z))
=> true
```

### 2.2.3 Monoidi

Si definisce come:

```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}

trait Monoid[A] extends Semigroup[A] {
  def empty: A
}
```

Monoid estende la classe di tipo Semigroup, aggiungendo un metodo empty al combine del semigroup. Il metodo empty deve restituire un valore che se combinato con qualsiasi altra istanza di quel tipo restituisce l'altra istanza:

```
(combine(x, empty) == combine(empty, x) == x)
```

Infatti, dato che i monoidi estendono i semigroup hanno anche essi l'associativity, avendo però un valore empty hanno anche l'identity definita come: combine(x, empty) = combine(empty, x) = x. Ad esempio:

```
val empty = 0
combine(1, empty) == combine(empty, 1)
=> true
combine(1, empty) == 1
=> true
```

### 2.2.4 Funtori

Si definisce come:

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

Un Functor è una classe di tipo onnipresente che coinvolge tipi che hanno un "foro", cioè tipi che hanno la forma F[*], come Option, List e Future. (Questo è in contrasto con un tipo come Int che non ha foro, o Tuple2 che ha due fori (Tuple2[*,*])). Il Funtore prevede una singola operazione, denominata map.
Per quanto riguarda i funtori sono presenti due laws: Composition e Identity.

#### Composition

La compositione si definisce come:

```
x.map(f).map(g) = x.map(f.andThen(g))
```

Ovvero mappare con una funzione f poi con una funzione g è l'equivalente di mappare per composizione con f e g. Il seguente codice mostra un esempio di questa equivalenza:

```
val v = List(1, 1)
val g: Int => Int = _ + 1
=> g: Int => Int = <function1>
val f: Int => Int = _ - 1
=> f: Int => Int = <function1>
val l0 = v.map(f).map(g)
=> List(1, 1)
val l1 = v.map(f.andThen(g))
=> List(1, 1)
l0 == l1
=> true
```

**Identity**

L'identità si definisce come:

```
v.map(x => x) = v
```

Questa equivalenza ci dice che la map applicata ad un oggetto con la sua identità restituisce l'oggetto stesso. Infatti:

```
val v0 = List(1, 1)
val v1 = v0.map(x => x)
=> List(1, 2)
v0 == v1
=> true
```

### 2.2.5 Apply

Apply estende la classe di tipo Functor (che presenta la funzione map) con una nuova funzione ap. La funzione ap è simile alla map in quanto stiamo trasformando un valore in un contesto; un contesto è F in F[A]; un contesto può essere Option, List or Future per esempio. Tuttavia, la differenza tra ap e map è che per ap la funzione che si occupa della trasformazione è di tipo F[A ⇒ B], mentre per la mappa è A ⇒ B:

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}

trait Apply[F[_]] extends Functor[F]{
    def ap[A, B](ff: F[(A) => B])(fa: F[A]): F[B]
}
```

Ecco le implementazioni di Apply per i tipi Option e List:

```
import cats._

implicit val optionApply: Apply[Option] = new Apply[Option] {
  def ap[A, B](f: Option[A => B])(fa: Option[A]): Option[B] =
    fa.flatMap(a => f.map(ff => ff(a)))

  def map[A, B](fa: Option[A])(f: A => B): Option[B] = fa map f

}

implicit val listApply: Apply[List] = new Apply[List] {
  def ap[A, B](f: List[A => B])(fa: List[A]): List[B] =
    fa.flatMap(a => f.map(ff => ff(a)))

  def map[A, B](fa: List[A])(f: A => B): List[B] = fa map f
```

```
}
```

Infatti:

```
val intToString: Int => String = _.toString
intToString: Int => String = <function1>
val v = Apply[Option].map(Some(1))(intToString)
=> Some("1")
```

### 2.2.6 Applicative

Si definiscono come:

```
trait Applicative[F[_]] extends Apply[F]{
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]
  def pure[A](a: A): F[A]
  def map[A, B](fa: F[A])(f: A => B): F[B] = ap(pure(f))(fa)
}
```

Le Applicative estendono Apply aggiungendo un singolo metodo chiamato pure, pure eleva qualsiasi valore nel Funtore Applicativo in pratica questo metodo prende qualsiasi valore e restituisce il valore nel contesto del funtore. Ad esempio per Option potrebbe essere Some(_), per Future Future.successful e per List il singleton list.
Quando si parla di applicative la documentazione afferma che ap è un po' complicato da spiegare e motivare, quindi esamineremo una formulazione alternativa ma equivalente tramite product e definisce un Applicativa come:

```
trait Applicative[F[_]] extends Functor[F] {
  def product[A, B](fa: F[A], fb: F[B]): F[(A, B)]

  def pure[A](a: A): F[A]
}
```

Per quanto riguarda le applicative sono presenti tre laws: Associativity, Left Identity e Right Identity.

#### Associativity

L'associatività afferma che: indipendentemente dall'ordine in cui si sommano tre valori, il risultato è isomorfo. Infatti:

- fa.product(fb).product(fc) $\sim$ fa.product(fb.product(fc))

Con map, questo può essere trasformato in un'uguaglianza con:

- fa.product(fb).product(fc) = fa.product(fb.product(fc)).map  case (a, (b, c)) $\Rightarrow$ ((a, b), c)

**Left Identity**

Left Identity afferma che: comprimendo un valore a sinistra con unità si ottiene qualcosa di isomorfo al valore originale. Infatti:

- pure(()).product(fa) $\sim$ fa

Come uguaglianza:

- pure(()).product(fa).map(_._2) = fa

**Right Identity**

Right Identity afferma che: comprimere un valore a destra con l'unità risulta in qualcosa di isomorfo al valore originale. Infatti:

- fa.product(pure(())) $\sim$ fa

Come uguaglianza:

- fa.product(pure(())).map(_._1) = fa

### 2.2.7 Monadi

Si definisce come:

```
trait Monad[F[_]] extends FlatMap[F] with Applicative[F]{
        def flatten[A](ffa: F[F[A]]): F[A]
}
```

Monad estende la Applicative con una nuova funzione flatten. Flatten prende un valore in un contesto annidato (es. F[F[A]]dove F è il contesto) e "unisce" i contesti insieme in modo da avere un unico contesto (es. F[A]). Ad esempio:

```
Option(Option(1)).flatten
// res0: Option[Int] = Some(value = 1)
Option(None).flatten
// res1: Option[Nothing] = None
List(List(1),List(2,3)).flatten
// res2: List[Int] = List(1, 2, 3)
```

La seguente è una definizione più completa dell'interfaccia:

```
trait Monad[F[_]] extends FlatMap[F] with Applicative[F]{
    def flatten[A](ffa: F[F[A]]): F[A]
    def flatMap[A, B](fa: F[A])(f: (A) => F[B]): F[B]
    def pure[A](x: A): F[A]
    def tailRecM[A, B](a: A)(f: (A) => F[Either[A, B]]): F[B]
    //Keeps calling f until a scala.util.Right[B] is returned.

}
```

Se Applicative è già definita e flatten si comporta bene, estendere la Applicative a Monad è banale. Per fornire la prova che un tipo appartiene alla Monad classe type, l'implementazione di cats ci richiede di fornire un'implementazione di pure(che può essere riutilizzata da Applicative) e flatMap. Possiamo usare flatten per definire flatMap: flatMap è una map seguito da flatten. Al contrario, flatten si ottiene solo usando flatMap e la funzione di identità x ⇒ x. Ad esempio flatMap(_)(x ⇒ x).
flatMap è spesso considerata la funzione principale delle Monadi in Cats che segue questa tradizione fornendo implementazioni flatten e map derivati da flatMap e pure. La ragione di questo è che il nome flatMap ha un significato speciale in scala, in quanto le for-comprehension(sequenza di chiamate di uno o più metodi foreach, map, flatMap, ecc..) si basano su questo metodo per concatenare insieme le operazioni in un contesto di monadi.

### tailRecM

Oltre a richiedere flatMap e pure, le Monadi in Cats richiedono anche tailRecM che codifica la ricorsione monadica stack safe, come descritto in Stack Safety for Free di Phil Freeman. Poiché la ricorsione monadica è comune nella programmazione funzionale ma non è sicura nello stack sulla JVM, Cats ha scelto di richiedere questo metodo per tutte le implementazioni della monade invece che solo un sottoinsieme. Tutte le funzioni che richiedono la ricorsione monadica in Cats lo fanno tramite tailRecM.

Un esempio di implementazione di Monad per Option è mostrato qui sotto. Nota la coda ricorsiva e quindi l'implementazione sicura di tailRecM.

```
import cats.Monad
import scala.annotation.tailrec

implicit val optionMonad = new Monad[Option] {
  def flatMap[A, B](fa: Option[A])(f: A => Option[B]): Option[B] = fa.flatMap(f)
  def pure[A](a: A): Option[A] = Some(a)

  @tailrec
  def tailRecM[A, B](a: A)(f: A => Option[Either[A, B]]): Option[B] = f(a) match {
    case None              => None
    case Some(Left(nextA)) => tailRecM(nextA)(f) // continue the recursion
    case Some(Right(b))    => Some(b)            // recursion done
  }
}
```

### FlatMap

FlatMap è una classe di tipo strettamente correlata è identica a Monad, meno il metodo pure. Infatti in Cats Monad c'è una sottoclasse di FlatMap(da cui ottiene flatMap) e Applicative (da cui ottiene pure).

```scala
trait FlatMap[F[_]] extends Apply[F] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}

trait Monad[F[_]] extends FlatMap[F] with Applicative[F]
```

Le law per FlatMap sono solo le leggi di Monad che non menzionano pure.Infatti anche essa è Associativa, Left Identity, Right Identity. Una delle motivazioni per l'esistenza di FlatMap è che alcuni tipi di istanze hanno FlatMap ma non Monad. Un esempio è Map[K, *]. Considera il comportamento di pure per Map[K, A]. Dato un valore di type A, abbiamo bisogno di associarvi qualche arbitrario K ma non abbiamo modo di farlo. Tuttavia, dato che esiste Map[K, A]e Map[K, B](o Map[K, A ⇒ B]), è semplice accoppiare (o applicare funzioni a) valori con la stessa chiave. Quindi Map[K, *]ha un'istanza FlatMap.