Una libreria che fornisce astrazioni per linguaggi funzionali nel linguaggio Scala: Cats

Mattia Achilli
mattia.achilli@studio.unibo.it

Dicembre 2021

Indice

1 Obiettivo del progetto

Il progetto mette insieme aspetti teorici e pratici della libreria: si mette in evidenza il perché della libreria, ciò che la contradistingue, ciò che fornisce, con una particolare attenzione a meccanismi relativi a **concorrenza**, **I/O** e di gestione delle risorse con il supporto di **Cats-Effect**. Verranno mostrati e realizzati snippets code vari insieme ad esperimenti più approfonditi, infine verrà realizzata una **RestAPI** con Cats-Effect.

2 Costrutti matematici functional programming

Prima di introdurre 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.

2.1 Semigroup

Semigroup è un concetto che incapsula l'aggregazione con un'operazione binaria associativa. La classe di tipo Semigroup viene fornita con un metodo **combine** che combina semplicemente due valori dello stesso tipo seguendo il principio dell'associatività.

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

Per i semigroup esiste solo una regola che è l'**associativity**.

L'associativity è definita come: **combine(x, combine(y, z)) = combine(combine(x, y), z)**. **combine** mantiene l'associatività, il che significa che la seguente uguaglianza deve essere valida per qualsiasi scelta di x, y e z.

```
val x = 1
val y = 2
val z = 3

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

2.2 Monoidi

I monoidi estendono i semigroup aggiungendo semplicemente un valore **empty**.

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

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**.

```
val empty = 0

combine(1, empty) == combine(empty, 1)
=> true

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

2.3 Funtori

Il funtore è un costrutto per eseguire una sequenza di azioni e permette di implementare una funzionalità di tipo **map**.

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

Per quanto riguarda i funtori sono presenti due laws: **Composition** e **Identity**.

2.3.1 Composition

La composition è definita come: **x.map(f).map(g) = x.map(f.andThen(g))**. Mappare con una funzione **f** poi con una funzione **g** è lo stesso di mappare per composizione con **f** e **g**.

```
val v = List(1, 2)

val g: Int => Int = _ + 1
=> g: Int => Int = <function1>

val f: Int => Int = _ - 1
=> f: Int => Int = <function1>

val l = v.map(f).map(g)
=> List(1, 2)

val l1 = v.map(f.andThen(g))
=> List(1, 2)

l == l1
=> true
```

2.3.2 Identity

L'identity è definita come: **x.map(v => v) = x**. La map applicata ad un oggetto con la sua identità è uguale all'oggetto stesso.

```
val v = List(1, 2)

val l = v.map(x => x)
=> List(1, 2)

l == v
=> true
```

2.4 Monadi

Una monade (sottotipo dei funtori) a differenza dei funtori implementa una funzionalità di tipo **flatmap**.

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

Come per i funtori anche per le monadi esistono delle regole o laws: **Left identity**, **Right identity** e l'**Associativity**.

2.4.1 Left identity

La prima regola è definita come: **x.flatMap(f) = f(x)**. Applicando all'oggetto la flatMap attraverso la funzione o applicare direttamente la funzione all'oggetto il risultato non cambia.

Prendendo l'esempio di due funzioni che prendono in input un intero e tornano in output una lista:

```
val f: (Int => List[Int]) = x => List(x - 1, x, x + 1)
val g: (Int => List[Int]) = x => List(x, -x)
```

E' possibile verificare la left identity come segue:

```
val v = 2
val l = List(v).flatMap(f)
=> List(1, 2, 3)

val l1 = f(v)
=> List(1, 2, 3)

l == l1
=> true
```

2.4.2 Right identity

La Right identity è definita come: **x.flatMap(v => v) = x**. Un oggetto deve essere uguale alla flatMap applicata all'oggetto dell'oggetto stesso.

Riprendendo sempre l'esempio sopra delle funzioni **f** e **g**:

```
val v = List(2)

val l = v.flatMap(List(_))
=> List(2)

l == v
=> true
```

2.4.3 Associativity

L'ultima regola è la seguente: **x.flatMap(f).flatMap(g) = x.flatMap(v => f(v).flatMap(g))**. Se si applica ad un oggetto la flatMap due volte (con f e g) o si applica la flatMap all'oggetto con la funzione a cui si applica una flatMap il risultato non cambia.

```
val v = List(1, 2)

val l = v.flatMap(f).flatMap(g)
=> List(0, 0, 1, -1, 2, -2, 1, -1, 2, -2, 3, -3)

val l1 = v.flatMap(x => f(x).flatMap(g))
=> List(0, 0, 1, -1, 2, -2, 1, -1, 2, -2, 3, -3)

l == l1
=> true
```

3 Typeclass

Una **typeclass** è strumento utilizzato nella programmazione funzionale per abilitare il polimorfismo ad hoc. Laddove molti linguaggi orientati agli oggetti come **Java** sfruttano il concetto di **ereditarietà** per il codice polimorfico, la programmazione funzionale tende a una combinazione di polimorfismo parametrico (come i generici Java) e polimorfismo ad hoc.

Si può prendere come esempio quello di voler costruire un meccanismo di stampa per visualizzare degli oggetti in un sistema informativo scolastico. In questo caso gli oggetti sono **StudentId**, **StaffId** e **Score**:

```
case class StudentId(id: Int)
case class StaffId(id: Int)
case class Score(s: Double)
```

Ora si può definire la typeclass **Printer** relativa alla stampa come una trait:

```
trait Printer[A] {
  def getString(a: A): String
}
```

La typeclass sopra è parametricamente polimorfica perché viene definita con un tipo **A**, il che significa che si può sostituire A con qualsiasi tipo.

Un trait da solo o usato in altro modo non si qualifica come typeclass.

Ora si può definire la funzione che si vuole rendere poliformica ad-hoc:

```
def show[A](a: A)(implicit printer: Printer[A]): String = printer.getString(a)
```

Il primo parametro è di un tipo A, il secondo parametro richiede di definire la **type variable** per il tipo A che dovrebbe essere un sottotipo della typeclass **Printer** definita:

```
implicit val studentPrinter: Printer[StudentId] = new Printer[StudentId] {
  def getString(a: StudentId): String = s"StudentId: ${a.id}"
}
```

In questo modo si può aumentare il range di tipi che il metodo **show** può gestire, creando il **polimorfismo ad-hoc:**

```
implicit val studentPrinter: Printer[StudentId] = new Printer[StudentId] {
    def getString(a: StudentId): String = s"StudentId: ${a.id}"
}

implicit val staffPrinter: Printer[StaffId] = new Printer[StaffId] {
    def getString(a: StaffId): String = s"StaffId: ${a.id}"
}

implicit val scorePrinter: Printer[Score] = new Printer[Score] {
    def getString(a: Score): String = s"Score: ${a.s}%"
}
```

Come si vedrà a breve tutta la struttura di Cats e Cats-Effect si basa sulle typeclass.

4 Cos'è Cats e perché

Cats è una libreria che fornisce astrazioni per la programmazione funzionale nel linguaggio di programmazione **Scala**.

Scala è un linguaggio con un approccio ibrido che supporta sia la programmazione orientata agli oggetti che quella funzionale il ché non lo rende un linguaggio puramente funzionale. La libreria Cats si sforza di fornire astrazioni di programmazione puramente funzionale che siano soprattutto efficienti. L'obiettivo di Cats è fornire una base per un'ecosistema di librerie pure e tipizzate per supportare la programmazione funzionale in applicazioni Scala.

4.1 Punti di forza di Cats

Le caratteristiche che contradistinguono Cats sono le seguenti:

- **Approachability**: a seguito di molti successi, fallimenti e contributi la libreria Cats è molto accessibile a nuovi sviluppatori.
- **Minimality**: Cats è una libreria modulare, contiene le **typeclasses** e il minimo di strutture dati necessarie.
- **Documentation**: il progetto Cats ha da sempre l'obiettivo di documentare molto e in modo chiaro il codice (con tanto di esempi). Questo permette sempre riguardo all'approachability di avvicinare meglio lo sviluppatore alla libreria.
- **Efficiency**: punto chiave in Cats a cui il progetto tiene molto: mantenere la libreria il più efficiente possibile senza fare a meno di purezza e usabilità.

4.2 Struttura di Cats

La libreria Cats è molto consistente e mette a disposizione molti moduli ricchi di strutture dati e funzionalità, i due moduli di base richiesti sono i seguenti:

- **cats-kernel**: contiene un piccolo insieme di typeclasses.
- **cats-core**: contiene la maggior parte delle typeclasses e delle funzionalità core.

Oltre ai due moduli sopra citati sono presenti tanti altri moduli tra cui **Cats-effect** su cui ci si soffermerà particolarmente nel progetto per la gestione della concorrenza, delle risorse e I/O.

4.3 Casi d'uso di Cats

Cats è una libreria molto versatile e come già detto nella precedente sezione contiene un vasto insieme di moduli. Permette infatti di gestire collezioni, strutture dati algebriche, concorrenza, I/O e tanto altro.

In questo progetto ci si concentra maggiormente sulla parte concorrente e I/O attraverso il framework Cats-Effect.

5 Cats-Effect

Cats Effect è un framework asincrono per la creazione di applicazioni in uno stile puramente funzionale, fornisce uno strumento noto come **Monade IO**, per catturare e controllare le azioni, chiamate **effects**, da eseguire in un contesto tipizzato con supporto alla concorrenza e il coordinamento.

Gli **effects** possono essere asincroni (guidati da callback) o sincroni (restituiscono direttamente i valori).

Cats Effect definisce un insieme di typeclass che definiscono un sistema puramente funzionale.

5.1 Che cos'é un side effect

Una funzione contiene un **side-effect** se non gode della trasparenza referenziale. Vale a dire una funzione che quando riceve lo stesso parametro in input, restituisce sempre lo stesso valore in output. Quindi una funziona ha un **side-effect** quando ad esempio modifica una variabile al di fuori del proprio scoping, quando modifica uno dei suoi argomenti, quando scrive su file o quando invoca altre funzioni con side-effects.

5.2 Monade IO di Cats-Effect

La monade IO consente di:

- acquisire e controllare effetti asincroni basati su callback dietro un'interfaccia pulita e sincrona.
- gestire applicazioni altamente simultanee, come i servizi Web che devono servire decine di migliaia di richieste al secondo. La concorrenza in IO è facilitata dai **fibers**: thread leggeri come le coroutines e interrompibili, sono gestiti a runtime.
- raccogliere informazioni di runtime durante l'esecuzione del programma (**tracing**), rendendo più facile rintracciare l'origine degli errori.
- gestire i cicli di vita delle risorse e garantire che le risorse siano allocate e rilasciate in modo sicuro anche in presenza di eccezioni e interruzioni. La gestione delle risorse è più complesso in applicazioni concorrenti; un piccolo bug potrebbe causare una perdita di dati o persino un deadlock bloccando il sistema.
- scrivere programmi semplici che possono essere composti per formare programmi più complessi, pur mantenendo la capacità di ragionare sul comportamento e sulla complessità.

5.3 IO

Un IO in Cats-Effect è una struttura dati che rappresenta una descrizione di una computazione sincrona o asincrona con side-effects.

Un valore di tipo **IO[A]** è una computazione che, se valutata, può eseguire **effetti** prima di restituire un valore di tipo A. I valori IO sono puri e immutabili e quindi preservano la trasparenza funzionale.

```
val io: IO[Int] = IO.pure(42) // Si utilizza pure quando non c'è un side-effect

val aDelayedIO: IO[Int] = IO.delay({ // Viene valutata quando viene chiamata
    println("Hello World!")
    54
})

/* equivalente di delay */
```

```
val aDelayedIOV2: IO[Int] = IO { // apply
    println("Hello World!")
    54
}
```

Inoltre IO è anche una Monade con cui è possibile concatenere effetti attraverso **flatMap**, ad esempio:

```
val ioConcatenation: IO[String] = IO.pure(42).flatMap(_ => IO.pure(54)
    .flatMap(_ => IO.pure("Hello World!")))
```

Si può notare come gli effetti vengono concatenati con flatMap, tuttavia utilizzando questo approccio con tante computazioni risulta poco leggibile e chiaro. E' possibile utilizzare un approccio più leggibile e compatto attraverso il for-comprehension come segue:

```
val ioConcatenationV2: IO[String] = for {
    _ <- IO.pure(42)
    _ <- IO.pure(54)
    string <- IO.pure("Hello World")
} yield string
```

La logica è la stessa utilizzando flatMap ma il codice diventa molto più comprensibile e leggibile.

5.4 Fibers

Un fiber esegue un'azione **F** che è tipicamente un'istanza **IO**. I fibers sono come thread "leggeri", il che significa che possono essere utilizzati in modo simile ai thread per creare codice concorrente o parallelo. Tuttavia sono differenti per certi aspetti dai thread. Generare nuovi fibers non garantisce che l'azione **F** venga eseguita in carenza di threads. Internamente **Cats-effect** utilizza un pool di thread per i fibers. Quindi, se non è disponibile alcun thread nel pool, l'esecuzione del fiber "aspetta" fino a quando alcuni thread non sono liberi. D'altra parte, quando l'esecuzione di alcuni fibers è bloccata, ad esempio quando si deve attendere il rilascio di una mutex il thread che esegue la fiber è riciclato da Cats-effect e utilizzato per altri fibers. Quando l'esecuzione del fiber può essere ripreso, Cats-effect cercherà un thread libero per continuare l'esecuzione. Cats-effect ricicla anche i threads dei fibers finiti o interrotti. Però il fiber è davvero bloccato da qualche azione esterna come una socket TCP allora in quel caso Cats-effect non potrà in alcun modo recuperare quel thread fino al termine dell'azione (queste azioni andrebbero eseguite come **IO.blocking** per segnalare che l'azione bloccherà il thread). Ultima differenza dai threads è che i fibers sono molto leggeri, si possono generare milioni di fibers senza influire sulle prestazioni.

6 Cats-Effect Typeclasses

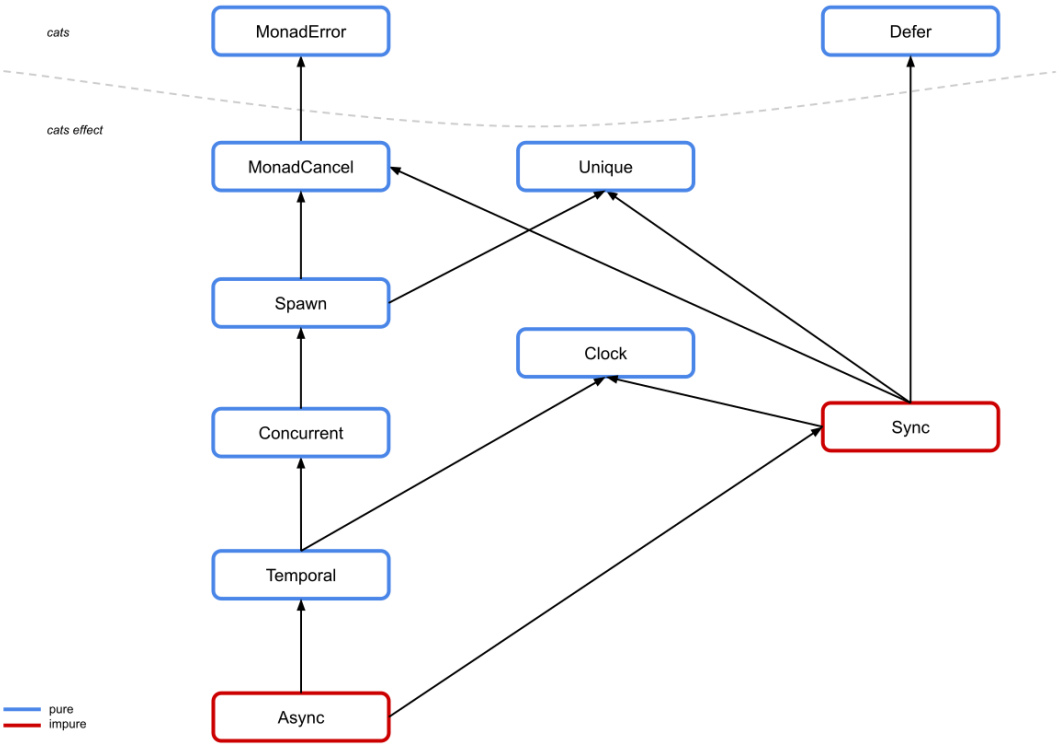

Figure 1: Gerarchia type classes in Cats-effect 3.

La gerarchia di Cats-Effect definisce cosa significa essere un "effetto funzionale". Si possono suddividere le regole per gli effetti funzionali nelle seguenti categorie:

- Safety e interruzione nell'utilizzo di risorse.
- Valutazione parallela di side-effects.
- Stato convidiso tra processi paralleli.
- Interazioni con il tempo.
- Acquisizione sicura dei side-effects che restituiscono valore.
- Acquisizione sicura dei side-effects che invocano callback.

Oltre a quanto detto sopra, alcuni meccanismi sono assunti da Cats-Effect ma definiti in altri moduli di Cats:

- Trasformare il valore all'interno di un effetto mappandolo sopra.

- Mettere il valore in un effetto.
- Comporre calcoli in sequenza, in modo tale che ciascuno dipenda dal precedente.
- Lanciare e gestire errori.

Di seguito vengono mostrate le typeclasses di Cats-effect con le relative API insieme ad esempi di utilizzo. Nelle sezioni successive alle typeclasses verranno mostrati veri e propri esempi anche complessi.

6.1 MonadCancel

Un fiber può terminare in tre diversi stati: **Succeeded**, **Errored** e **Canceled**.

Ciò significa che ad esempio quando si scrive codice per la gestione delle risorse, bisogna tenere conto di eventuali interruzioni (Canceled) ed eccezioni (Errored).

La typeclass **MonadCancel** risolve questo problema estendendo la typeclass di Cats **MonadError**. Usandola, si possono definire effetti che acquisiscono e rilasciano risorse in modo sicuro, ad esempio utilizzando **bracket**:

```
openFile.bracket(f => readFromFile(f))(f => closeFile(f))
```

bracket funziona un pò come l'equivalente FP di **try/finally**: se viene eseguito **openFile** verrà eseguito **closeFile**. Ciò accade anche se **readFromFile** produce un errore o viene interrotto da qualche altro fiber. Inoltre **openFile** è atomico: o non viene valutato per niente o viene valutato completamente permettendo di fare calcoli complessi senza temere che qualcosa di esterno si intrometta.

In aggiunta a **bracket** è presente anche **uncancelable** che consente di definire regioni di codice non interrompibili.

Ad esempio, si immagina di avere un blocco di codice che deve essere protetto da un Semaforo, assicurando accesso esclusivo. Il problema qui è che l'acquisizione del Semaforo, che è una risorsa, può anche comportare il blocco del fiber, e quindi potrebbe essere necessario interromperlo dall'esterno. In generale, l'acquisizione delle risorse deve essere non interrompibile, ma in questo caso particolare si deve consentire l'interruzione, altrimenti potrebbe finire per bloccare la JVM e mandare quindi in deadlock il sistema. **uncancelable** quindi fornisce un meccanismo per raggiungere questo obiettivo:

```
def guarded[F[_], R, A, E](s: Semaphore[F], alloc: F[R])(use: R => F[A])
    (release: R => F[Unit])(implicit F: MonadCancel[F, E]): F[A] =
  F uncancelable { poll =>
    for {
      r <- alloc

      _ <- poll(s.acquire).onCancel(release(r))
      releaseAll = s.release >> release(r)
```

```
      a <- poll(use(r)).guarantee(releaseAll)
    } yield a
}
```

La prima azione che si esegue è **alloc** che esegue un effetto di tipo **R**. Una volta acquisito il valore si tenta di acquisire il Semaforo. Se un altro fiber ha già acquisito il semaforo, il fiber corrente potrebbe bloccarsi per un pò. Si vuole far si che altri fiber siano in grado di interrompere il blocco. Quindi si wrappa **s.acquire** in **poll** (riabilita l'interruzione all'interno di uncancelable per tutto ciò che wrappa). Se l'acquisizione del semaforo viene interrotta, ci si vuole assicurare di rilasciare la risorsa r utilizzando **onCancel**. Infine si passa all'invocazione dell'azione **use(r)** wrappato da poll, indipendentemente dal fatto che **use(r)** si completi naturalmente, generi un errore o venga interrotta viene eseguito **releaseAll** che rilascia l'acquisizione del semaforo e della risorsa.

Inoltre, un aspetto particolare di **MonadCancel** è l'auto-interruzione. Ad esempio:

```
MonadCancel[F].canceled >> fa
```

L'effetto **fa** in questo caso non verrà mai eseguito a meno ché sia racchiuso in un blocco non interrompibile. L'auto-interruzione è in qualche modo simile alla generazione di un errore con **raiseError**.

6.2 Spawn

Questa typeclass fornisce un'astrazione simile a Thread, che può essere implementata per computazioni parallele. Come già detto in precedenza i fibers sono thread leggeri, è infatti possibile averne anche milioni su un'unica macchina.

La typeclass **Spawn** che estende **MonadCancel** mette a disposizione le seguenti funzionalità:

```
trait Spawn[F[_], E] extends MonadCancel[F, E] {
    def start[A](fa: F[A]): F[Fiber[F, Throwable, A]] // Crea un fiber
    def never[A]: F[A] // Ritorna un effetto che non termina mai
    def cede: F[Unit] /* Un effetto "yield", permette di spostare
        l'esecuzione su un altro fiber */

    /*
    I metodi race permettono di eseguire due task concorrentemente:
    Nel metodo race viene ritornato il primo che finisce, il perdente viene interrotto.
    Nel metodo racePair è possibile gestire il task perdente.
    */
    def race[A, B](left: IO[A], right: IO[B]): IO[Either[A, B]]
    def racePair[A, B](left: F[B], right: F[B]): Either[
      (Outcome[F, E, A], Fiber[F, E, B]),
      (Fiber[F, E, A], Outcome[F, E, B])
    ]
}
```

Se ad esempio si vuole eseguire un effetto su un altro fiber è possibile farlo attraverso **start** e aspettare la terminazione con il risultato con **join**:

```
def effectOnSomeThread[F[_]: Spawn, A](fa: F[A]):
    F[Outcome[F, Throwable, A]] = for {
        fib <- fa.start
        result <- fib.join
    } yield result
```

Si può prendere l'esempio di un Server che accetta delle connessioni:

```
trait Server[F[_]] {
  def accept: F[Connection[F]]
}

trait Connection[F[_]] {
  def read: F[Array[Byte]]
  def write(bytes: Array[Byte]): F[Unit]
  def close: F[Unit]
}

object Endpoint {
  def endpoint[F[_]: Spawn](server: Server[F])(body: Array[Byte] => F[Array[Byte]]):
    F[Unit] = {

    def handle(conn: Connection[F]): F[Unit] =
      for {
        request <- conn.read
        response <- body(request)
        _ <- conn.write(response)
      } yield ()

    val handler = MonadCancel[F].uncancelable { poll =>
      poll(server.accept).flatMap { conn =>
        handle(conn).guarantee(conn.close).start
      }
    }

    handler.foreverM // Gestisce le connessioni continuamente
  }
}
```

Per l'handler si utilizza **uncancelable** per evitare perdite di risorse tra quando si ottiene la connessione a quando si imposta la gestione delle risorse. Per ogni connessione la si gestisce con la funzione **handle** e indipendentemente dall'esito la si chiude.

Infine si utilizza **start** per creare un nuovo fiber per ogni connessione che arriva.

6.2.1 Cancelation

Probabilmente il più grosso vantaggio di utilizzare i fibers oltre ad essere leggeri è che a differenza dei thread della JVM sono interrompibili.

Ad esempio:

```
for {
    target <- IO.println("Hello World").foreverM.start
    _ <- IO.sleep(1.second)
    _ <- target.cancel
} yield ()
```

Questo esempio di codice stamperà un numero non deterministico di volte "Hello World" fino a quando il fiber principale è in sleep per un secondo, finito il tempo di sleep interrompe il fiber **target** attraverso **cancel**. Utilizzando Thread è impossibile replicare questo esempio senza costruire il proprio meccanismo per l'interruzione. Con fiber però è gia tutto gestito dalla typeclass Spawn.

6.2.2 Joining

Di solito quando si eseguono fibers paralleli per dei task si desidera attendere che finiscano, accettare i risultati e andare avanti.

```
for {
    /* Stampa per 5 secondi Hello World e in seguito racchiude un intero in IO */
    fiberA <- (IO.println("Hello World Fiber A").foreverM.timeoutTo(5.seconds, IO.unit)
        >> IO.pure(1)).start
    fiberB <- (IO.println("Hello World Fiber B").foreverM.timeoutTo(5.seconds, IO.unit)
        >> IO.pure(2)).start

    /* Attendo il completamento */
    a <- fiberA.joinWithNever
    b <- fiberB.joinWithNever

    _ <- IO.println(s"a: $a, b: $b")
} yield (a, b)
```

Il metodo **joinWithNever** è un metodo conveniente basato su **join**, che è molto più generale.

In particolare, il metodo **Fiber#join** restituisce F[Outcome[F, E, A]] (dove E è il tipo di errore per F).

Outcome ha la seguente forma:

- **Succeeded** contiene un valore di tipo F[A].
- **Errored** contiene un valore di tipo **E** solitamente **Throwable**.

- **Canceled** non contiene niente.

Questi rappresentano i tre possibili stati di terminazione per un fiber e c'è la possibilità di reagire a ciascuno in modo diverso. Ad esempio:

```
fiber.join flatMap {
  case Outcome.Succeeded(fa) =>
    fa
  /* In caso di errore */
  case Outcome.Errored(e) => ???
  /* Quando il fiber viene interrotto */
  case Outcome.Canceled() => ???
}
```

Tornando all'esempio di prima:

```
for {
    fiberA <- (IO.println("Hello World Fiber A").foreverM.timeoutTo(5.seconds, IO.unit)
        >> IO.pure(1)).start
    fiberB <- (IO.println("Hello World Fiber B").foreverM.timeoutTo(5.seconds, IO.unit)
        >> IO.pure(2)).start

    a <- fiberA.join flatMap {
        case Outcome.Succeeded(fa) => fa
        case Outcome.Errored(e) => MonadError[IO, Throwable].raiseError(e)
        /* Nel caso in cui il fiber figlio venga interrotto, si cerca di
        interrompere l'attuale fiber, se non è possibile, deadlock */
        case Outcome.Canceled() => MonadCancel[IO].canceled >> Spawn[IO].never
    }
    b <- fiberB.joinWithNever

    _ <- IO.println(s"a: $a, b: $b")
} yield (a, b)
```

6.3 Unique

È una typeclass che genera token univoci.

```
trait Unique[F[_]] {
  def unique: F[Unique.Token]
}
```

Ogni invocazione di **unique** garantisce un valore distinto di token da quelli attualmente allocati. Se un token viene liberato dal **Garbage Collector** quel token può essere riallocato a seguito di una nuova chiamata a **unique**. La garanzia di unicità è presente

solo all'interno di una singola JVM. Se si ha bisogno di un token univoco in tutto lo spazio e il tempo è meglio usare **UUID**.

Sia **Sync[F]** che **Spawn[F]** estendono **Unique[F]** poiché entrambe le typeclasses hanno la possibilità di creare valori univoci tramite **delay** e **start** (i fibers sono sempre unici).

6.4 Clock

Clock è una typeclass che fornisce un sistema analogo a **System.nanoTime()** e **System.currentTimeMillis()**.

```
for {
      start <- Clock[IO].realTime
      _ <- IO.sleep(5.seconds)
      end <- Clock[IO].realTime
      _ <- IO.println(end - start) // Tempo in millisecondi trascorso
} yield ()
```

6.5 Concurrent

Questa typeclass estende **Spawn** con la capacità di allocare stato simultaneo sotto forma di **Ref** e **Deferred** e di eseguire varie operazioni che richiedono l'allocazione di stato simultaneo come **memoize**.

```
trait Concurrent[F[_]] extends Spawn[F] {
    def ref[A](a: A): F[Ref[F, A]]
    def deferred[A]: F[Deferred[F, A]]
}
```

6.6 Ref e Deferred

Ref è una primitiva che fornisce l'accesso e la modifica concorrente in modo sicuro. Mentre **Deferred** garantisce un meccanismo di sincronizzazione che può essere usato una sola volta per ogni istanza di Deferred.

Ad esempio si può utilizzare Ref e Deferred per creare un contatore con sincronizzazione in cui sono presenti due "agenti", un agente si occupa di incrementare un contatore mentre l'altro attende di essere sbloccato:

```
def notificationAlarm[F[_]: Concurrent]: F[Unit] = {
    def notifyAlarmComplete(signal: Deferred[F, Int]): F[Unit] =
      for {
        _ <- Concurrent[F].pure("[Notifier] Start alarm...")
        value <- signal.get
        _ <- Concurrent[F].pure("[Notifier] time's up!") >>
            Concurrent[F].pure(s"[Notifier] value: $value")
```

```
      } yield ()

    def incrementCounter(contentRef: Ref[F, Int], signal: Deferred[F, Int]): F[Unit] =
      for {
        _ <- Concurrent[F].pure(s"[Incrementer] incrementing value...")
        newValue <- contentRef.getAndUpdate(_ + 1)
        _ <- Concurrent[F].pure(s"[Incrementer] $newValue")
        _ <- if (newValue == 10) signal.complete(newValue) else
            incrementCounter(contentRef, signal)
      } yield ()

    for {
      contentRef <- Concurrent[F].ref(0) // Il contatore viene inizializzato a 0
      signal <- Concurrent[F].deferred[Int] // Utilizzato per la sincronizzazione
      /* Vengono creati due fibers paralleli, uno si occupa di incrementare
      mentre l'altro attende di essere sbloccato */
      notifierFib <- notifyAlarmComplete(signal).start
      incrementerFib <- incrementCounter(contentRef, signal).start
      _ <- notifierFib.join
      _ <- incrementerFib.join
    } yield ()
}
```

Il notifier si mette in uno stato di attesa utilizzando **signal.get**.

L'incrementer utilizza **getAndUpdate** da **Ref** per aggiornare il valore del contatore in maniera thread-safe e ottenere il nuovo valore. Quando il valore del contatore arriva a 10 viene chiamato **signal.complete** per sbloccare l'altro agente.

6.7 Memoization

Si può memorizzare un effetto in modo che venga eseguito solo una volta e il risultato utilizzato ripetutamente. Ad esempio:

```
val action: IO[String] = IO.println("This is only printed once").as("action")
for {
    memoized <- action.memoize
    res1 <- memoized
    res2 <- memoized
    _ <- IO.println(res1 ++ res2)
    /* This is only printed once actionaction */
} yield ()
```

6.8 Temporal

Temporal è una typeclass che estende **Concurrent** e permette di sospendendere un fiber (blocco semantico) per un certo tempo. Ad esempio:

```
val chainOfEffects: IO[String] = IO.println("Before sleeping")
    >> Temporal[IO].sleep(1.second) >> IO.println("After sleeping!")
```

Nota: si potrebbe fare **Sync[F].delay(Thread.sleep(duration))** ma questo è sbagliato poiché blocca il thread. **Temporal[F]#sleep** blocca semanticamente l'esecuzione del fiber. Internamente viene utilizzato uno scheduler per attendere la durata specificata prima di fare il rescheduling del fiber.

6.9 Sync

Sync è una typeclass **FFI** (foreign function interface, in questo caso da Scala con Cats a Scala) per sospendere le operazioni con side-effect. Il significato di "sospendere" dipende se il side-effect che si vuole sospendere è bloccante o meno.

```
trait Sync[F[_]] extends MonadCancel[F, Throwable] with Defer[F] {
    def delay[A](thunk: => A): F[A]
    def blocking[A](thunk: => A): F[A]
}
```

6.9.1 Metodi di sospensione

Se il side-effect non blocca il thread si può utilizzare **Sync[F].delay**:

```
/* "Sospende" la computazione e l'effetto viene eseguito solo nel momento in
cui viene invocato */
val aDelayedIO: IO[Int] = Sync[IO].delay {
    println("I'm an effect")
    42
}

for {
    value <- aDelayedIO // I'm an effect
    _ <- IO.println(s"Value is: $value") // Value is: 42
} yield ()
```

Se il side-effect è bloccante allora bisognerebbe utilizzare **Sync[F].blocking** che non solo sospende il side-effect ma sposta anche l'esecuzione su un pool di thread separato per evitare il blocco del pool di thread di calcolo. L'esecuzione viene spostata di nuovo sul pool di thread di calcolo una volta completata l'operazione. Ad esempio si può utilizzare per leggere da un file:

```
for {
    f <- Sync[IO].blocking(Source.fromFile("file.txt"))
    s <- Sync[IO].blocking(f.mkString)
    _ <- Sync[IO].blocking(f.close()) >> IO.println(s)
} yield ()
```

Uno svantaggio di utilizzare **Sync[F].blocking** è che il fiber che esegue l'effetto bloccante non è interrompibile fino al completamento della chiamata. E' possibile renderlo interrompibile attraverso **Sync[F].interruptible**:

```
/* true significa che si prova a interrompere il thread ripetutamente finché
non viene interrotto */
val aBlockingEffect: IO[Int] = Sync[IO].blocking {
    println("loading...")
    Thread.sleep(1000)
    42
}
val interruptible: IO[Unit] = for {
    operation <- Sync[IO].interruptible(many = true)(aBlockingEffect.foreverM)
    _ <- operation.timeout(2.seconds)
} yield ()
```

Nell'esempio precedente viene eseguito all'infinito l'effetto **aBlockingEffect** ma dopo 2 secondi viene interrotto da **timeout**. Questo grazie anche a **interruptible** che rende l'effetto bloccante interrompibile.

6.10 Async

Async è una typeclass FFI per sospendere le operazioni con side-effects che vengono completate altrove (spesso su un altro pool di thread). Questa typeclass dà la possibilità di eseguire in sequenza operazioni asincrone senza cadere nel **callback hell** e permette di spostare l'esecuzione in altri contesti di esecuzione. Async rappresenta la typeclass più complessa e potente di Cats-effect.

```
trait Async[F[_]] extends Sync[F] with Temporal[F] {
    /* Ritorna il contesto di esecuzione corrente */
    def executionContext: F[ExecutionContext] //

    def async[A](cb: (Either[Throwable, A] => Unit) => IO[Option[IO[Unit]]]): F[A]
    def async_[A](cb: (Either[Throwable, A] => Unit) => Unit): F[A]

    /* Esegue l'effetto su un altro thread pool per poi ritornare su
    quello ritornato da executionContext */
    def evalOn[A](fa: F[A], ec: ExecutionContext): F[A]
```

```
    /* Ritorna un effetto che non termina mai */
    def never[A]: F[A]
}
```

6.10.1 Threadpool shifting

Async ha l'abilità di spostare l'esecuzione su un diverso pool di thread.

Ad esempio si può vedere qual'è il thread pool sul quale vengono eseguiti i side-effects:

```
val printThread: IO[Unit] = Async[IO].executionContext.flatMap(IO.println(_))

val printThreadPoolEffect: IO[Unit] = for {
    _ <- printThread //WorkStealingThreadPool
    _ <- Async[IO].evalOn(printThread, ExecutionContext.global) // ExecutionContextImpl
    _ <- printThread //WorkStealingThreadPool
} yield ()
```

Mentre se si vuole eseguire una computazione su un pool di thread separato è possibile utilizzare **async_**:

```
val threadPool: ExecutorService = Executors.newFixedThreadPool(10) // Un pool di 10 thre

type Callback[A] = Either[Throwable, A] => Unit

val asyncComplex: IO[Int] = Async[IO].async_ {cb: Callback[Int] =>
    threadPool.execute { () =>
        println(s"[${Thread.currentThread().getName}] Computing async")
        cb(Right(42))
    }
}
```

7 Primitive di Cats-Effect

In questa sezione vengono mostrate le principali primitive concorrenti più utilizzate per gestire problemi concorrenti di sincronizzazione come deferred, ref, semafori, barriere, countdown latch e la gestione delle risorse.

7.1 Deferred

E' una primitiva di sincronizzazione che rappresenta un singolo valore che potrebbe non essere ancora disponibile. Una volta creato, un deferred è vuoto. Può essere completato una sola volta e non può essere mai più svuotato.

Mette a disposizione due metodi:

```
abstract class Deferred[F[_], A] {
  def get: F[A]
  def complete(a: A): F[Boolean]
}
```

Se si richiama **get** su un deferred vuoto ci si blocca finché non viene completato con **complete** altrimenti viene ritornato immediatamente il valore. Il metodo **get** è interrompibile fino a quando non è completo.

complete imposta il valore **a** se il deferred è vuoto e ritorna true, altrimenti non modifica il valore e ritorna false.

Deferred può essere utilizzato insieme a **Ref** per creare comportamenti concorrenti e strutture dati come code e semafori. Quindi è utile ogni volta che ci si trova in uno scenario in cui molti processi possono modificare lo stesso valore ma solo il primo ad arrivare può farlo.

Due processi cercheranno di essere completati nello stesso tempo ma solo uno avrà esito positivo, completando esattamente una volta. Il perdente solleverà un errore nel tentativo di completare un deferred già completato e verrà automaticamente interrotto dal meccanismo **IO.race**.

Ad esempio utilizzando **IO.race** è possibile eseguire in parallelo due fibers che cercano di completare il valore di deferred nel seguente modo:

```
object DeferredExample extends IOApp.Simple {
  def raceCompletion(deferred: Deferred[IO, Int]): IO[Unit] = {
    val attemptCompletion: Int => IO[Unit] = n => deferred.complete(n).attempt.void
    val race: IO[Either[Unit, Unit]] = IO.race(attemptCompletion(1),
        attemptCompletion(2))
    (race, deferred.get.flatMap { n => IO(println(s"Result: $n")) })
        .parTupled
        .void
  }

  override def run: IO[Unit] = {
    for {
      deferred <- Deferred[IO, Int]
      _ <- raceCompletion(deferred)
    } yield ()
  }
}
```

Il metodo **parTupled** permette di eseguire due effetti in parallelo, inoltre se uno di quelli fallisce, falliscono anche gli altri.

7.2 Ref

Fornisce l'accesso e la modifica concorrente in modo sicuro, ma nessuna funzionalità per la sincronizzazione, che è invece come è stato visto viene gestita da **Deferred**. Per

questo Ref viene sempre inizializzato con un valore.

Il caso classico per cui utilizzare **Ref** è quello di realizzare un contatore con accesso concorrente da diversi fibers:

```
def counter(): IO[Unit] = {
    def worker(id: Int, ref: Ref[IO, Int]): IO[Unit] =
        for {
          v <- ref.get
          _ <- IO.delay(println(s"Worker $id value $v"))
          newValue <- ref.getAndUpdate(_ + 1) // Thread-safe
          _ <- IO.delay(println(s"Worker $id after increment: $newValue"))
        } yield ()

    for {
      ref <- Ref[IO].of(0) // Inizializzato a 0
      worker1 = worker(1, ref)
      worker2 = worker(2, ref)
      worker3 = worker(3, ref)
      _ <- (worker1, worker2, worker3).parTupled.void
    } yield()
}
```

7.3 Semafori

Un semaforo è utilizzato per la sincronizzazione e viene inizializzato con un valore positivo. E' utile quando più processi tentano di accedere ad una risorsa condivisa e si vuole limitare il numero di accessi a tale risorsa.

L'acquisizione del semaforo riduce il valore e il rilascio del semaforo aumenta il valore. Un'acquisizione che si verifica quando il valore è a 0 risulta bloccata fino a quando qualcuno non rilascia il semaforo.

Il semaforo si definisce come **Semaphore[F]**, per acquisire e rilasciare il semaforo si usano rispettivamente **acquire** e **release**. Un semaforo con un singolo permesso funziona come una **mutex**. Ad esempio dati n utenti si può simulare un sistema in cui un utente alla volta entra nel sistema acquisendo il semaforo, fa qualcosa poi esce rilasciando il semaforo:

```
def doWorkWhileLoggedIn(): IO[Int] = IO.sleep(1.second) >> IO(Random.nextInt(100))
val mutex: IO[Semaphore[IO]] = Semaphore[IO](1)
val users: IO[List[Int]] = mutex.flatMap { sem =>
    (1 to 10).toList.parTraverse { id =>
      for {
        _ <- IO.println(s"[session $id] waiting to log in...")
        _ <- sem.acquire
        // critical section
```

```
        _ <- IO.println(s"[session $id] logged in, working...")
        res <- doWorkWhileLoggedIn()
        _ <- IO.println(s"[session $id] done: $res, logging out...")
        // end of critical session
        _ <- sem.release
      } yield res
    }
}
```

7.4 Count Down Latch

E' una primitiva di concorrenza che blocca tutti i fibers in attesa su di essa. Viene inizializzato con un numero intero positivo **n** e i fibers in attesa vengono bloccati semanticamente fino a quando non vengono rilasciati tutti gli **n** latch.

In Cats-Effect esiste già **CountDownLatch** ma ora che sono stati mostrati Ref e Deffered è possibile decostruire il **CountDownLatch** attraverso queste due primitive:

```
/* Classe astratta con i metodi await e release */
abstract class CountDownLatch {
    def await: IO[Unit]
    def release: IO[Unit]
}

object CountDownLatch {
    sealed trait State
    case object Done extends State
    case class Live(remainingCount: Int, signal: Deferred[IO, Unit]) extends State

    def apply(count: Int): IO[CountDownLatch] = for {
      signal <- Deferred[IO, Unit]
      state <- Ref[IO].of[State](Live(count, signal))
    } yield new CountDownLatch {

      override def await: IO[Unit] = state.get.flatMap {
        case Done => IO.unit // Il latch è finito
        case _ => signal.get // Si blocca semanticamente
      }

      override def release: IO[Unit] = state.modify { // Modifica lo stato
        /* Se il latch è già finito non succede nulla */
        case Done => Done -> IO.unit
        /* Se manca un solo latch si cambia lo stato in Done e si sbloccano
        quelli in attesa */
        case Live(1, signal) => Done -> signal.complete(()).void
```

```
        /* Altrimenti si lascia lo stato in Live e si decrementano il numero di latch */
        case Live(n, signal) => Live(n - 1, signal) -> IO.unit
      }.flatten.uncancelable
    }
}
```

Tra i metodi await e release quello di release non può essere interrotto mentre quello di await si perché si utilizza **signal.get** e c'è rischio di deadlock nel caso in cui nessuno sblocchi.

Utilizzando invece il **CountDownLatch** di Cats-Effect si può definire un **CountDownLatch** inizializzato a 2 e attendere con **await** che il valore arrivi a 0:

```
object CountDownLatchExample extends IOApp.Simple {
  override def run: IO[Unit] = {
    for {
      latch <- CountDownLatch[IO](2)
      fiber <- (latch.await >> IO.println("Latch unlocked")).start
      _ <- latch.release >> IO.println("First release")
      _ <- latch.release >> IO.println("Second release")
      _ <- fiber.join
    } yield()
  }
}
```

7.5 Cyclic Barrier

E' una primitiva di sincronizzazione riutilizzabile che consente a un insieme di fibers di attendere fino a quando non hanno raggiunto tutte lo stesso punto.

Una barriera viene inizializzata con un intero positivo **n** e i fibers che chiamano **await** vengono bloccati fino a quando non viene invocata **await n** volte, a quel punto vengono tutti sbloccati e la barriera viene rilasciata. Quando la barriera viene rilasciata può essere riutilizzata ciclamente con un valore iniziale di **n**.

Anche qua come per il **CountDownLatch** è possibile decostruire la **CyclicBarrier** utilizzando Ref e Deferred nel seguente modo, a differenza del CountDownLatch è presente solo il metodo await:

```
abstract class CyclicBarrier {
    def await: IO[Unit]
  }

  object CyclicBarrier {
    case class State(nWaiting: Int, signal: Deferred[IO, Unit])

    def apply(count: Int): IO[CyclicBarrier] = for {
      signal <- Deferred[IO, Unit]
```

```
      state <- Ref[IO].of[State](State(count, signal))
    } yield new CyclicBarrier {
      override def await: IO[Unit] = Deferred[IO, Unit].flatMap { newSignal =>
        state.modify { // Modifica lo stato
          /* Se c'è solo un waiter si rinializza lo stato come all'inizio e si
          sbloccano tutti */
          case State(1, signal) => State(count, newSignal) -> signal.complete(()).void
          /* Altrimenti si decrementa il numero di waiter e si rimane in attesa */
          case State(n, signal) => State(n - 1, signal) -> signal.get
        }.flatten
      }
    }
}
```

Il metodo **await** è interrompibile.

Utilizzando il CyclicBarrier di Cats-Effect, se si vuole definire una barriera inizializzata a 2 si possono definire due fibers che eseguono in parallelo e alla seconda chiamata di **await** i due fibers vengono sbloccati:

```
object CyclicBarrierExample extends IOApp.Simple {
  override def run: IO[Unit] = {
    for {
      barrier <- CyclicBarrier[IO](2)
      fiberA <- (IO.println("First fiber") >> barrier.await >>
        IO.println("First fiber after barrier")).start
      fiberB <- (IO.println("Second fiber") >> barrier.await >> IO.sleep(1.second) >>
        IO.println("Second fiber after barrier")).start
      _ <- (fiberA.join, fiberB.join).tupled
    } yield ()
  }
}
```

7.6 Resource

Viene utilizzato per la gestione delle risorse, un pattern comune è acquisire una risorsa (ad esempio un file), eseguire un'azione su di essa e infine eseguire un finalizzatore (nel caso del file, chiudere il file), indipendentemente dall'esito dell'azione.

Il modo più semplice per costruire una risorsa è con il metodo **make** e il modo più semplice per utilizzare una risorsa è utilizzare **use**. Ad esempio si può leggere un file riga per riga ogni 100ms:

```
def openFileScanner(path: String): IO[Scanner] =
    IO(new Scanner(new FileReader(new File(path))))

def resourceReadFile(path: String): IO[Unit] = {
```

```
    def readLineByLine(scanner: Scanner): IO[Unit] =
      if (scanner.hasNextLine) IO.println(scanner.nextLine()) >> IO.sleep(100.millis) >>
        readLineByLine(scanner)
      else IO.unit

    Resource.make(openFileScanner(path))(scanner => IO.println("Closing resource") >>
        IO(scanner.close())).use { scanner =>
            readLineByLine(scanner)
        }
}
```

Il file scanner viene chiuso indipendentemente dall'esito della lettura del file. E si nota inoltre che sia l'acquisizione che il rilascio della risorsa non sono interrompibili.

8 Code Snippets

In questa sezione vengono mostrati ed esplicitati pezzi di codice soprattutto relativi alla definizione e utilizzo di semigroups, monoidi, funtori e monadi fondamentali nell'API di Cats e Cats-Effect. Infine vengono mostrati ulteriori esempi sulla gestione dei fibers.

8.1 Semigroup

Utilizzando Cats è possibile definire una proprio **Semigroup** custom ridefinendo il metodo **combine** nel seguente modo:

```
implicit val multiplicationSemigroup: Semigroup[Int] = (x: Int, y: Int) => x * y

print(Semigroup[Int].combine(2, 3)) //6
```

Per gli interi di default il metodo combine somma gli elementi, ridefinendo il metodo (implicitamente) è possibile combinare secondo un altro tipo di operazione, in questo caso la moltiplicazione.

E' anche possibile definire una proprio classe custom per poi combinarla a piacere:

```
final case class CustomClass(value: Int)
object CustomClass {
    implicit val productIntSemigroup: Semigroup[CustomClass] =
        (x: CustomClass, y: CustomClass) => CustomClass(x.value * y.value)
}

print(CustomClass(10) |+| CustomClass(3))
// Equivalente a CustomClass(10).combine(CustomClass(3))
```

8.2 Monoidi

Un monoide è simile ad un semigroup e per definirne una propria è sufficiente ridefinire oltre a combine il metodo **empty**, nel caso degli interi:

```
object MainIntMonoids extends App {
  implicit val multiplicationMonoid: Monoid[Int] = new Monoid[Int] {
    override def empty: Int = 1

    override def combine(x: Int, y: Int): Int = x * y
  }

  print(Monoid[Int].combine(Monoid[Int].empty, 2)) // 2
  print(Monoid[Int].combine(1, 2)) // 2
}
```

E' possibile anche definire una propria classe custom:

```
object MainCustomMonoids extends App {
  final case class CustomClass(value: Int)
  object CustomClass {
    implicit val customMonoid: Monoid[CustomClass] = new Monoid[CustomClass] {
      override def empty: CustomClass = CustomClass(1)

      override def combine(x: CustomClass, y: CustomClass): CustomClass =
        CustomClass(x.value * y.value)
    }
  }

  /* CustomClass(2) */
  print(Monoid[CustomClass].combine(Monoid[CustomClass].empty, CustomClass(2)))
  print(Monoid[CustomClass].combine(CustomClass(3), CustomClass(2))) //CustomClass(6)
}
```

8.3 Funtori

Come per semigroups e monoidi è possibile definire un funtore customizzato attraverso la classe **Functor** ridefinendo il metodo **map**:

```
case class CustomFunctor[A](value: A)

object CustomFunctor {
  implicit val functor: Functor[CustomFunctor] = new Functor[CustomFunctor] {
    def map[A, B](fa: CustomFunctor[A])(f: A => B): CustomFunctor[B] =
        CustomFunctor(f(fa.value))
```

```
  }
}
```

Il main:

```
object MainCustomFunctor extends App{
  print(CustomFunctor(1).map(_ + 1)) //CustomFunctor(2)
}
```

8.4 Monadi

Come già detto precendentemente una monade è un meccanismo che permette di implementare azioni di tipo **flatmap**.

Come per i funtori è possibile definire una propria Monade customizzata (in questo caso chiamata CustomMonad) attraverso la classe **Monad**, ridefinendo i seguenti metodi:

- **pure**: trasforma un valore in option.
- **flatMap**: applica una trasformazione di tipo flatmap.
- **tailRecM**: ottimizzazione usata in Cats per limitare la quantità di spazio utilizzato sullo stack. Se implementata Cats garantisce sicurezza in operazioni di grandi dimensioni.

Un esempio di Monade è il seguente:

```
case class CustomMonad[A](value:A)

object CustomMonad {
    implicit val customMonad: Monad[CustomMonad] = new Monad[CustomMonad] {
        override def pure[A](x: A): CustomMonad[A] = CustomMonad(x)

        override def flatMap[A, B](fa: CustomMonad[A])(f: A => CustomMonad[B]):
            CustomMonad[B] = f(fa.value)

        @tailrec
        override def tailRecM[A, B](a: A)(f: A => CustomMonad[Either[A, B]]):
            CustomMonad[B] = f(a) match {
              case CustomMonad(either) => either match {
                case Left(a) => tailRecM(a)(f)
                case Right(b) => CustomMonad(b)
            }
        }
    }
}
```

Per poi utilizzarla come segue:

```
object MainCustomMonad extends App {
  val endResult = for {
    a <- CustomMonad(1)
    b <- CustomMonad(2)
  } yield a + b
  print(endResult) // CustomMonad(3)
}
```

8.5 Fibers

In questa sezione vengono mostrati ulteriori esempi su come creare, interrompere e gestire i fibers. Come prima cosa prima di procedere è consigliato creare una classe **implicit** di **IO** per estenderla con un metodo **debug** per stampare il nome del **thread** corrente con il valore che incapsula l'IO:

```
implicit class Extension[A](io: IO[A]) {
    def debug: IO[A] = {
      io.map { value =>
        println(s"[${Thread.currentThread().getName}] $value")
        value
      }
    }
}
```

In questo modo è possibile richiamare il metodo **debug** su IO per verificare su quale thread viene eseguito un effetto.

Ad esempio:

```
object FiberExample extends IOApp {
  val intValue: IO[Int] = IO(1)
  val stringValue: IO[String] = IO("Scala")

  override def run(args: List[String]): IO[ExitCode] = {
    intValue.debug *> stringValue.debug *> IO(ExitCode.Success)
  }
}
```

Oppure si può verificare che vengono eseguiti sullo stesso thread nel seguente modo:

```
object FiberExample extends IOApp {
  val intValue: IO[Int] = IO(1)
  val stringValue: IO[String] = IO("Scala")
```

```
  def sameThread(): IO[Unit] = for {
    _ <- intValue.debug
    _ <- stringValue.debug
  } yield ()

  override def run(args: List[String]): IO[ExitCode] = {
    sameThread().as(ExitCode.Success) // as equivale a map
  }
}
```

Dall'output si può verificare che sia **intValue** che **stringValue** vengono valutati dallo stesso thread.

8.5.1 Creazione

Per creare un nuovo fiber diverso da quello del flusso principale è sufficiente utilizzare **start** come già visto precedentemente:

```
object FiberExample extends IOApp {
  val intValue: IO[Int] = IO(1)
  val stringValue: IO[String] = IO("Scala")

  /*
    I tre parametri generici sono:
      - Tipo dell'effetto: in questo caso IO
      - Il tipo dell'errore su cui potrebbe fallire: Throwable
      - Il tipo di dato che ritornerebbe in caso di successo: Int
  */
  val fiber: IO[Fiber[IO, Throwable, Int]] = intValue.debug.start

  def differentThread(): IO[Unit] = {
    for {
      _ <- fiber
      _ <- stringValue.debug
    } yield ()
  }

  override def run(args: List[String]): IO[ExitCode] = {
    differentThread().as(ExitCode.Success)
  }
}
```

Inoltre come già visto in precedenza è possibile attendere il risultato di un fiber con **join**:

```scala
object FiberExample extends IOApp {
  val intValue: IO[Int] = IO(1)

  def runOnAnotherTread[A](io: IO[A]): IO[Outcome[IO, Throwable, A]] = {
    for {
      fib <- io.start // fiber
      result <- fib.join // Risultato in result
      /*
        result è un IO[Outcome[IO, Throwable, A]]:
        1 - success(IO(value))
        2 - errored(e)
        3 - cancelled
       */
    } yield result
  }

  override def run(args: List[String]): IO[ExitCode] = {
    runOnAnotherTread(intValue).debug.as(ExitCode.Success)
    // [io-compute-#] Succeeded(IO(1))
  }
}
```

8.5.2 Interruzione

Come già detto precedentemente a differenza dei Thread è possibile interrompere un fiber attraverso **cancel**:

```scala
object FiberExample extends IOApp {
  def cancelOnAnotherThread(): IO[Outcome[IO, Throwable, String]] = {
    val task = IO("starting").debug *> IO.sleep(1.second) *> IO("done").debug
    for {
      fib <- task.start
      _ <- IO.sleep(500.millis) *> IO("cancelling").debug
      _ <- fib.cancel
      result <- fib.join
    } yield result
  }

  override def run(args: List[String]): IO[ExitCode] = {
    cancelOnAnotherThread().debug.as(ExitCode.Success)
  }
}
```

Nel codice sopra il fiber visualizza ”starting”, aspetta un secondo e poi visualizza ”done”. Nel frattempo il fiber principale attende 500 millisecondi e interrompe il fiber

creato. Nel result viene indicato quindi **Canceled**.

8.5.3 Racing

In questa sezione viene mostrato come eseguire concorrentemente due task (su fiber distinti) attraverso il metodo **race** di IO che ritorna un **IO[Either[A, B]]** dove **A** e **B** sono i tipi di ritorno. Il fiber perdente, cioè quello la cui azione viene completata per seconda viene interrotto. Un esempio utilizzando race è il seguente:

```
object Racing extends IOApp.Simple {
  val valuableIO: IO[Int] = {
    IO("task starting").debug *> IO.sleep(1.second).debug *> IO("task completed").debug
        *> IO(1).debug
  }
  val vIO: IO[Int] = valuableIO.onCancel(IO("task: cancelled").debug.void)
  val timeout: IO[Unit] = {
    IO("timeout: starting").debug *> IO.sleep(500.millis).debug
        *> IO("timeout: finished").debug.void
  }

  def race(): IO[String] = {
    // Il fiber perdente viene interrotto
    // IO.race => IO[Either[A, B]]
    val firstIO: IO[Either[Int, Unit]] = IO.race(vIO, timeout)

    firstIO.flatMap {
      case Left(v) => IO(s"task won: $v")
      case Right(_) => IO("timeout won")
    }
  }

  def run: IO[Unit] = race.debug.void
  /*
    L'output sarà simile al seguente:
        [io-compute-7] timeout: starting
        [io-compute-6] task starting
        [io-compute-2] ()
        [io-compute-2] timeout: finished
        [io-compute-3] task: cancelled
        [io-compute-3] timeout won
  */
}
```

Nell'esempio vengono eseguiti due task concorrentemente, il primo **vIO** visualizza una stringa, aspetta un secondo visualizza un'altra string e ritorna un intero in IO. Nel caso

in cui questo venga interrotto si visualizza **task: cancelled**. Il secondo task stampa, aspetta mezzo secondo e stampa nuovamente. In questo caso il secondo task finisce prima in quanto rimane in sleep per meno tempo rispetto al primo task. Quindi l'output sarà **timeout won**.

Se invece si vogliono eseguire due task concorrenti senza voler interrompere il fiber perdente ma gestendolo diversamente, si può utilizzare **racePair** che ritorna un **IO[Either[(OutcomeIO[A], FiberIO[B]), (FiberIO[A], OutcomeIO[B])]]** dove ognuno delle due tuple contiene l'outcome del task e il fiber dell'altro task da gestire, ad esempio si può decidere comunque di interromperlo:

```
object Racing extends IOApp.Simple {
   def racePair[A](ioA: IO[A], ioB: IO[A]): IO[OutcomeIO[A]] = {
    val pair = IO.racePair(ioA, ioB)
    // IO[Either[(OutcomeIO[A], FiberIO[B]), (FiberIO[A], OutcomeIO[B])]]

    pair.flatMap {
      case Left((outcomeA, fiberB)) => fiberB.cancel *> IO("first task won").debug
        *> IO(outcomeA).debug
      case Right((fiberA, outcomeB)) => fiberA.cancel *> IO("second task won").debug
        *> IO(outcomeB).debug
    }
  }

  // ioA viene completato prima di ioB
  val ioA: IO[Int] = IO.sleep(1.second).as(1).onCancel(IO("first cancelled")
    .debug.void)
  val ioB: IO[Int] = IO.sleep(2.second).as(2).onCancel(IO("second cancelled")
    .debug.void)

  def run: IO[Unit] = racePair(ioA, ioB).void
  /*
    L'output sarà simile al seguente:
        [io-compute-7] second cancelled
        [io-compute-7] first task won
        [io-compute-7] Succeeded(IO(1))
  */
}
```

9 Esperimenti

In questa sezione, dopo aver introdotto le typeclasses di Cats-Effect e le principali primitive vengono realizzati degli esperimenti più complessi realizzati con Cats-Effect. Ci si focalizza in particolare sulla gestione delle risorse, I/O e su problemi di concorrenza che sono i punti cardine inc cui viene utilizzato maggiormente Cats-Effect.

9.1 Gestione delle risorse con IO

In questo primo esperimento l'obiettivo è quello di creare un programma che copi i file. Prima di tutto viene definita una funzione **copy** che prende come parametri il file di origine e quello di destinazione che ritorna un'istanza di **IO** che incapsula tutti i side effects coinvolti (apertura/chiusura e lettura/scrittura). In questa implementazione l'istanza IO restituirà la quantità di byte copiati da un file all'altro. Ovviamente possono verificarsi errori, ma quando si lavora con qualsiasi istanza IO, questi dovrebbero essere incorporati nell'istanza IO. Cioè, nessuna eccezione viene sollevata al di fuori dell'IO e quindi non è necessario utilizzare blocchi **try/catch**.

Per prima cosa si istanziano gli stream per il file di origine e quello di destinazione poi con **bracket** si gestisce la creazione, l'uso e il rilascio della risorsa nel seguente modo:

```
def copy(originFile: File, destinationFile: File): IO[Long] = {
    val inIO: IO[InputStream] = IO(new FileInputStream(originFile))
    val outIO: IO[OutputStream] = IO(new FileOutputStream(destinationFile))

    (inIO, outIO)
      .tupled // Da (IO[InputStream], IO[OutputStream]) a IO[(InputStream, OutputStream)
      .bracket {
        case (in, out) => transfer(in, out) // Usa le risorse
      } {
        case (in, out) => // Rilascia le risorse
          (IO(in.close()), IO(out.close()))
            .tupled // Da (IO[Unit], IO[Unit]) a IO[(Unit, Unit)]
            .handleErrorWith(_ => IO.unit).void // Gestisce in caso d'errore
      }
}
```

Quando si usa **bracket** se c'è un problema nell'ottenere la risorsa la parte di rilascio della risorsa non viene mai eseguita.

Ora si può creare la funzione **transfer** che si occupa di richiamare a sua volta **transmit** che effettua l'operazione di scrittura sul file di destinazione:

```
private def transmit(originFile: InputStream, destinationFile: OutputStream,
    buffer: Array[Byte], acc: Long): IO[Long] = {
        for {
          /* Si utilizza blocking per spostare l'esecuzione su un thread pool dedicato */
          amount <- IO.blocking(originFile.read(buffer, 0, buffer.length))
          /* Richiama ricorsivamente finché non si arriva a EOF, alla fine ritorna il
          totale byte trasferiti */
          count <- if (amount > -1) IO.blocking(destinationFile.write(buffer, 0,
            amount)) >> transmit(originFile, destinationFile, buffer,
                acc + amount) else IO(acc)
        } yield count
```

```
}

private def transfer(originFile: InputStream, destinationFile: OutputStream): IO[Long] =
    transmit(originFile, destinationFile, new Array[Byte](1024 * 10), 0L)
```

Quando si eseguono operazioni di **I/O** come in questo caso la lettura e scrittura da/su file è raccomandato utilizzare **IO.blocking** per aiutare Cats-effect nell'assegnamento dei threads. L'operatore di Cats >> viene utilizzato quando due operazioni si susseguono ma non per forza l'input della seconda deve essere l'output della prima (e l'equivalente di **first.flatMap(x => second)**, nel codice sopra indica che dopo ogni scrittura si deve chiamare ricorsivamente **transmit** accumulando i byte trasferiti.

Il main dell'applicazione estende **IOApp** che è una sorta di equivalente "funzionale" di **App** di Scala. Quando si esegue il main viene eseguito il metodo **run** che prende come argomenti il nome dei due file, copia il contenuto dal file di origine a quello di destinazione e stampa il totale dei byte trasferiti:

```
object MainSimpleCopyFiles extends IOApp {
  override def run(args: List[String]): IO[ExitCode] = {
    for {
      /* Se ci sono meno di due argomenti lancia un'eccezione */
      _ <- if (args.length < 2) IO.raiseError(new IllegalArgumentException
            ("Need origin and destination files")) else IO.unit
      originFile = new File(args.head)
      destinationFile = new File(args.tail.head)
      count <- copy(originFile, destinationFile)
      _ <- IO.println(s"$count bytes copied from ${originFile.getPath}
        to ${destinationFile.getPath}")

    } yield ExitCode.Success
  }
}
```

9.1.1 Versione polimorfica

Tornando al codice creato per copiare i file, si possono creare le funzioni in termini di **F**[_], ad esempio:

```
private val BUFFER_SIZE = 1024 * 10

/* Crea lo stream di input wrappato da Resource */
def inputStream[F[_]: Sync](file: File): Resource[F, FileInputStream] = {
    Resource.make { // Acquisisce la risorsa
      Sync[F].blocking(new FileInputStream(file))
    } { dataInputStream => // Rilascio della risorsa
      Sync[F].blocking(dataInputStream.close()).handleErrorWith(_ => Sync[F].unit)
```

```
    }
}

/* Crea lo stream di output wrappato da Resource */
def outputStream[F[_]: Sync](file: File): Resource[F, FileOutputStream] = {
    Resource.make {
      Sync[F].blocking(new FileOutputStream(file))
    } { dataOutputStream =>
      Sync[F].blocking(dataOutputStream.close()).handleErrorWith(_ => Sync[F].unit)
    }
}

/* Ritorna lo stream di input e output wrappati in Resource */
def inputOutputStreams[F[_]: Sync](inputFile: File, outputFile: File):
    Resource[F, (InputStream, OutputStream)] = {
        for {
          inStream <- inputStream(inputFile)
          outStream <- outputStream(outputFile)
        } yield (inStream, outStream)
}

private def transmit[F[_]: Sync](originFile: InputStream, destinationFile: OutputStream,
    Array[Byte], acc: Long): F[Long] = {
        for {
          amount <- Sync[F].blocking(originFile.read(buffer, 0, buffer.length))
          count <- if (amount > -1) Sync[F].blocking(destinationFile.write(buffer, 0,
            amount)) >> transmit(originFile, destinationFile, buffer, acc + amount) else
                Sync[F].pure(acc)
        } yield count
}

private def transfer[F[_]: Sync](originFile: InputStream, destinationFile: OutputStream,
    bufferSize: Int): F[Long] =
        transmit(originFile, destinationFile, new Array[Byte](bufferSize), 0L)

def copy[F[_]: Sync](originFile: File, destinationFile: File, bufferSize: Option[Int]
    = None): F[Long] = {
        inputOutputStreams(originFile, destinationFile).use { case (in, out) =>
          transfer(in, out, bufferSize.getOrElse(BUFFER_SIZE))
    }
}
```

La logica è praticamente identica a prima con l'unica differenza che si utilizza **Resource** invece di bracket per la gestione delle risorse.

Solo nel main si imposterà **IO** come **F**:

```
object MainPolymorphicCopyFiles extends IOApp{
  override def run(args: List[String]): IO[ExitCode] = {
    for {
      _ <- if (args.length < 2) IO.raiseError(new IllegalArgumentException
        ("Need origin and destination files")) else IO.unit
      originFile = new File(args.head)
      destinationFile = new File(args.tail.head)
      count <- copy[IO](originFile, destinationFile)
      _ <- IO.println(s"$count bytes copied from ${originFile.getPath}
        to ${destinationFile.getPath}")

    } yield ExitCode.Success
  }
}
```

Una versione più estesa e corretta controlla che il file di origine esista, che il file di origine e destinazione siano diversi e che se il file di destinazione esiste già di chiedere all'utente se vuole sovrascrivere il file.

```
object MainAdvancedPolymorphicCopyFiles extends IOApp {

  override def run(args: List[String]): IO[ExitCode] = {
    for {
      _ <- if (args.length < 2) IO.raiseError(new IllegalArgumentException
        ("Need origin and destination files")) else IO.unit
      /* Se il file di origine non esiste */
      _ <- if (!Files.exists(Paths.get(args.head))) IO.raiseError(
        new IllegalArgumentException("Files must be exists!")) else IO.unit
      /* Il file di origine e destinazione devono essere diversi */
      _ <- if (args.head == args.tail.head) IO.raiseError(new IllegalArgumentException
        ("Origin file and destination " + "files must be different!")) else IO.unit
      /* Se il file di destinazione esiste già si chiede se sovrascriverlo */
      _ <- if (Files.exists(Paths.get(args.tail.head))) IO.println
            ("Override destination file (Y/N)?") >>
                IO.readLine.map(_ != "Y").ifM(IO.canceled, IO.unit)
                else IO.unit

      originFile = new File(args.head)
      destinationFile = new File(args.tail.head)

      count <- copy[IO](originFile, destinationFile)
      _ <- IO.println(s"$count bytes copied from ${originFile.getPath}
```

```
        to ${destinationFile.getPath}")

    } yield ExitCode.Success
  }
}
```

9.2 Producer-consumer

Il secondo esperimento mette in evidenza l'utilizzo di primitive concorrenti come **Ref** e **Deferred** per il pattern **producer-consumer**.

Uno o più produttori inseriscono dati su una struttura dati condivisa come una coda mentre uno o più consumatori estraggono dati da essa. Produttori e consumatori eseguono concorrentemente e se la coda è vuota i consumatori si bloccano fino a quando non ci sono dati disponibili, se la coda invece è piena i produttori aspettano che un consumatore liberi uno spazio. Solo un produttore alla volta può aggiungere dati alla coda per garantire consistenza. Inoltre, un solo consumatore può estrarre i dati dalla coda in modo che non ci siano due consumatori che ottengano lo stesso dato.

In questa sezione verranno mostrati alcuni scenari di implementazione del problema attraverso Cats-effect.

9.2.1 Simple Producer-consumer

In questo scenario viene implementatato il pattern producer-consumer attraverso una coda condivisa tra Producer e consumer. E' presente un solo produttore e un solo consumatore. Il produttore genera una sequenza di interi e il consumatore legge la sequenza.

L'accesso alla coda è concorrente, quindi ci vuole un meccanismo di protezione in modo tale che un solo fiber per volta possa accedere alla struttura e modificarla. Il miglior modo di fare questo è tramite **Ref** visto precedentemente. Quando un fiber accede alla struttura tramite Ref tutti gli altri fiber si bloccano.

Il producer quindi viene definito così:

```
def producer[F[_]: Sync: Console](queue: Ref[F, Queue[Int]], counter: Int): F[Unit]
  = {
      for {
        _ <- Console[F].println(s"Produced item: $counter")
        /* Aggiunge un elemento alla coda, solo un fiber alla volta ci può accedere */
        _ <- queue.update(q => q.enqueue(counter + 1))
        _ <- producer(queue, counter + 1)
      } yield ()
}
```

Il metodo non fa altro che stampare l'elemento prodotto, modificare la coda attraverso il metodo **update** di Ref che garantisce l'acesso e la modifica ad un unico fiber per volta e richiamare se stesso.

Il metodo del consumatore è molto simile, inizialmente prende l'elemento dalla coda se esiste (si utilizza **dequeueOption**), poi stampa l'elemento se esisteva:

```
def consumer[F[_]: Sync: Console](queue: Ref[F, Queue[Int]]): F[Unit] = {
    for {
      /* Modifica la coda e se non è vuota ne rimuove uno dalla coda */
      i0 <- queue.modify { q =>
        q.dequeueOption.fold((q, Option.empty[Int])) {
          case (i, q) => (q, Option(i))
        }
      }
      _ <- if (i0.nonEmpty) Console[F].println(s"Consumed item: ${i0.get}")
        else Sync[F].unit
      _ <- consumer(queue)
    } yield ()
}
```

Il main che utilizza i due metodi definiti sopra è il seguente:

```
object MainProducerConsumer extends IOApp {
  override def run(args: List[String]): IO[ExitCode] = {
    for {
      // La coda viene inizializzata vuota
      queue <- Ref.of[IO, Queue[Int]](Queue.empty[Int])
      res <- (consumer(queue), producer(queue, 0))
        .parMapN((_, _) => ExitCode.Success)
        .handleErrorWith { t =>
          Console[IO].errorln(s"Error caught: ${t.getMessage}").as(ExitCode.Error)
        }
    } yield res
}
```

Il metodo **run** istanzia la coda condivisa che viene wrappata da Ref e lancia il producer consumer in parallelo. Per fare questo viene utilizzato il metodo **parMapN** che crea e esegue i fibers che eseguono l'**IO** passato per parametro. In questo caso sia il producer che il consumer eseguono all'infinito.

In alternativa all'utilizzo di parMapN è possibile usare **start** per creare esplicitamente i fibers, infine utilizzare **join** per aspettare il completamento:

```
object MainProducerConsumer extends IOApp {
    override def run(args: List[String]): IO[ExitCode] = {
        for {
          queueR <- Ref.of[IO, Queue[Int]](Queue.empty[Int])
          producerFiber <- producer(queueR, 0).start
          consumerFiber <- consumer(queueR).start
```

```
            _ <- producerFiber.join
            _ <- consumerFiber.join
          } yield ExitCode.Error
    }
}
```

Però c'è un problema, se c'è un errore nei fibers la **join** non viene completata e ritornata. La **parMapN**, invece, permette di gestire possibili errori quindi è preferibile.

La soluzione con la **parMapN** funziona e gestisce bene gli errori ma non è efficiente. Infatti con questa implementazione i produttori producono molto più rapidamente di quanto i consumatori consumano e quindi la coda cresce costantemente. Inoltre i consumatori eseguono indipendentemente dal fatto che ci siano o meno elementi in coda quando invece dovrebbero bloccarsi.

Si può migliorare la soluzione utilizzando **Deferred** e utilizzando diversi produttori e consumatori per bilanciare il carico.

9.2.2 Producer-consumer unbounded

Nel produttore/consumatore precedente si protegge già l'accesso alla coda utilizzando **Ref**. Ora invece di usare **Option** per rappresentare gli elementi recuperati da una coda possibilmente vuota, dovremmo invece bloccare il fiber del consumatore se la coda è vuota finché non viene prodotto un nuovo elemento. Questo può essere fatto come detto precedentemente utilizzando **Deferred**. Le istanza di Deferred vengono create vuote e possono essere riempite solo una volta. Se un fiber tenta di leggere l'elemento da un Deferred vuoto, verrà bloccato fino a quando un altro fiber non lo completa.

Quindi bisogna tenere conto anche delle istanze Deffered create quando la coda era vuota che sono in attesa di elementi disponibili. Per questo viene creato una **case class State** che mantiene la coda di elementi prodotti e la coda di consumatori in attesa:

```
// takers contiene la coda delle istanze di Deffered create quando la coda era vuota.
case class State[F[_], A](queue: Queue[A], takers: Queue[Deferred[F, A]])

object State {
    def empty[F[_], A]: State[F, A] = State(Queue.empty, Queue.empty)
}
```

Sia i producer che consumer accedono all'istanza di State attraverso **Ref**.

Il consumer si comporta in due modi: se la coda non è vuota prende l'elemento dalla testa, se la coda è vuota si istanzia un nuovo Deferred e si aggiunge ai **takers** dell'istanza State bloccando infine il consumer.

Il consumer viene definito quindi come:

```
def consumer[F[_] : Async : Console](id: Int, state: Ref[F, State[F, Int]]): F[Unit] = {
    val consume: F[Int] = {
      Deferred[F, Int].flatMap {
```

```
        taker =>
          state.modify { // Modifica lo stato
            /* Se la coda non è vuota prende l'elemento dalla testa, aggiorna state e
            ritorna l'elemento */
            case State(queue, takers) if queue.nonEmpty =>
              val (i, rest) = queue.dequeue
              State(rest, takers) -> Async[F].pure(i)
            /* Se la coda è vuota aggiunge l'istanza alla coda di takers e si blocca
            aspettando di essere completato */
            case State(queue, takers) => State(queue, takers.enqueue(taker)) ->
                taker.get
          }.flatten
      }
    }

    for {
      i <- consume
      _ <- Console[F].println(s"Consumer $id has got item: $i")
      _ <- consumer(id, state)
    } yield ()
}
```

Il parametro **id** viene usato solo per identificare il consumatore.

Il produttore invece: se la coda dei **takers** è vuota mette semplicemente in coda l'elemento prodotto altrimenti prende un taker dalla testa di takers e lo completa (sblocca):

```
def producer[F[_] : Sync : Console](id: Int, counter: Ref[F, Int],
    state: Ref[F, State[F, Int]]): F[Unit] = {
        def produce(i: Int): F[Unit] =
          state.modify { // Modifica lo stato
            /* Se la coda dei takers non è vuota prende il primo e lo completa
            con il valore */
            case State(queue, takers) if takers.nonEmpty => /
              val (taker, rest) = takers.dequeue
              State(queue, rest) -> taker.complete(i).void
            /* Altrimenti mette in coda l'elemento prodotto */
            case State(queue, takers) =>
              State(queue.enqueue(i), takers) -> Sync[F].unit
          }.flatten

        for {
          i <- counter.getAndUpdate(_ + 1) // Aggiorna il contatore
          _ <- produce(i)
          _ <- Console[F].println(s"Producer $id product item: $i")
```

```
          _ <- producer(id, counter, state)
        } yield ()
}
```

Al producer viene passato anche un counter inizialmente a 1 wrappato da un **Ref**. Infine il main:

```
object MainProducerConsumer extends IOApp {
  override def run(args: List[String]): IO[ExitCode] = {
    for {
      /* Istanzia state che viene wrappato da Ref */
      state <- Ref.of[IO, State[IO, Int]](State.empty[IO, Int])
      /* Inizializza il contatore a 1 wrappato da Ref */
      counter <- Ref.of[IO, Int](1)
      /* 10 produttori e 10 consumatori, viene passato come id i valori 1..10 */
      producers = List.range(1, 11).map(producer(_, counter, state))
      consumers = List.range(1, 11).map(consumer(_, state))
      res <- (producers ++ consumers) // Concatenzazione di liste
        /* Esegue sia i produttori che consumatori in parallelo  */
        .parSequence.as(ExitCode.Success)
        .handleErrorWith {
          t => Console[IO].errorln(s"Error caught: ${t.getMessage}").as(ExitCode.Error)
        }
    } yield res
  }
}
```

Con l'utilizzo di Deferred i consumatori aspettano che ci siano elementi nel buffer per consumare un elemento e utilizzando più consumatori e produttori si migliora l'equilibrio tra essi, ma comunque, la coda tende ad aumentare di dimensioni con il tempo non essendoci un limite. Per risolvere questo problema bisognerebbe aggiungere una dimensione limitata alla coda, in questo modo i produttori si bloccano come fanno i consumatori quando la coda è vuota.

9.2.3 Producer-consumer bounded

Avere una coda di elementi limitata implica che i produttori rimangano bloccati quando la coda è piena, e che vengano sbloccati quando si libera uno spazio. Per fare questo viene aggiunto a State una nuova coda di **Deferred** che tiene conto dei produttori bloccati come si fa già per i consumatori.

```
case class State[F[_], A](capacity: Int, queue: Queue[A], takers: Queue[Deferred[F, A]],
                          offerers: Queue[(A, Deferred[F, Unit])])

object State {
```

```
    /* Aggiunta la capacità massima della coda */
    def empty[F[_], A](capacity: Int): State[F, A] = State(capacity, Queue.empty,
        Queue.empty, Queue.empty)
}
```

Il consumatore può trovare quattro tipi di scenari che dipendono dalla coda di elementi e dalla coda di **offerers** (produttori bloccati):

- Se la coda di elementi non è vuota e non ci sono produttori bloccati si estrae un elemento dalla testa della coda.
- Se la coda di elementi non è vuota e c'è almeno un produttore bloccato si consuma un elemento dalla coda, si toglie il primo produttore dalla coda e si aggiunge l'elemento (che aveva il produttore) alla coda, infine si sblocca il producer bloccato.
- Se la coda di elementi è vuota e non ci sono produttori bloccati si aggiunge il consumatore alla coda dei consumatori bloccati e lo si blocca.
- Se la coda di elementi è vuota e quella dei produttori bloccati non è vuota si estrae il primo produttore dalla coda e lo si sblocca.

Il codice del consumatore diventa quindi:

```
def consumer[F[_]: Async: Console](id: Int, state: Ref[F, State[F, Int]]): F[Unit] = {
    val consume: F[Int] =
      Deferred[F, Int].flatMap { taker =>
        state.modify {
          /* Se la coda di elementi non è vuota e non ci sono produttori bloccati */
          case State(capacity, queue, takers, offerers) if queue.nonEmpty &&
          offerers.isEmpty =>
            val (i, rest) = queue.dequeue
            State(capacity, rest, takers, offerers) -> Async[F].pure(i)
          /* Se la coda di elementi non è vuota e c'è almeno un produttore bloccato */
          case State(capacity, queue, takers, offerers) if queue.nonEmpty =>
            val (i, rest) = queue.dequeue
            val ((_, release), tail) = offerers.dequeue
            State(capacity, rest, takers, tail) -> release.complete(()).as(i)
          /* Se la coda di elementi è vuota e non ci sono produttori bloccati */
          case State(capacity, queue, takers, offerers) if offerers.nonEmpty =>
            val ((i, release), rest) = offerers.dequeue
            State(capacity, queue, takers, rest) -> release.complete(()).as(i)
          /* Altrimenti */
          case State(capacity, queue, takers, offerers) =>
            State(capacity, queue, takers.enqueue(taker), offerers) -> taker.get
        }.flatten
      }
```

```
    for {
      i <- consume
      _ <- Console[F].println(s"Consumer $id has got item: $i")
      _ <- consumer(id, state)
    } yield ()
}
```

Il produttore invece ha tre possibili scenari:

- Se c'è qualche consumatore in attesa viene sbloccato passandogli l'elemento prodotto.
- Se non c'è alcun consumatore in attesa e la coda di elementi non è piena allora l'elemento viene prodotto e messo nella coda.
- Se non ci sono consumatori in attesa e la coda è piena si blocca il produttore.

```
def producer[F[_]: Async: Console](id: Int, counter: Ref[F, Int],
    state: Ref[F, State[F, Int]]): F[Unit] = {
        def produce(i: Int): F[Unit] =
          Deferred[F, Unit].flatMap { offerer =>
            state.modify {
              /* Se c'è almeno un consumatore bloccato */
              case State(capacity, queue, takers, offerers) if takers.nonEmpty =>
                val (taker, rest) = takers.dequeue
                State(capacity, queue, rest, offerers) -> taker.complete(i).void
              /* Se non c'è nessun consumatore bloccato e la coda di elementi
              non è piena */
              case State(capacity, queue, takers, offerers) if queue.size < capacity =>
                State(capacity, queue.enqueue(i), takers, offerers) -> Async[F].unit
              /* Altrimenti */
              case State(capacity, queue, takers, offerers) =>
                State(capacity, queue, takers, offerers.enqueue(i -> offerer)) ->
                    offerer.get
            }.flatten
          }

        for {
          i <- counter.getAndUpdate(_ + 1) // Aggiorna il contatore
          _ <- produce(i)
          _ <- Console[F].println(s"Producer $id product item: $i")
          _ <- producer(id, counter, state)
        } yield ()
}
```

Il main è identico a prima con la differenza della capacità massima della coda:

```
object MainProducerConsumer extends IOApp {
  override def run(args: List[String]): IO[ExitCode] = {
    for {
      /* Capacità massima della coda è di 10 elementi */
      state <- Ref.of[IO, State[IO, Int]](State.empty[IO, Int](capacity=10))
      counter <- Ref.of[IO, Int](1)
      producers = List.range(1, 11).map(producer(_, counter, state))
      consumers = List.range(1, 11).map(consumer(_, state))
      res <- (producers ++ consumers)
        .parSequence.as(ExitCode.Success)
        .handleErrorWith {
          t => Console[IO].errorln(s"Error caught: ${t.getMessage}").as(ExitCode.Error)
        }
    } yield res
  }
}
```

Ora tutti i problemi precedenti sono stati risolti, ma cosa succede se uno dei fiber che gestisce un consumatore o un produttore viene interrotto? Lo stato diventa inconsistente? Si può gestire questa cosa grazie al metodo **uncancelable** e **Poll**.

9.2.4 Cancelation-safe Producer-consumer

Si può utilizzare il metodo **uncancelable** per delimitare una regione di codice che non può essere interrotta. Ma quando l'operazione è **offerer.get** c'è un problema poiché si bloccherà fino al completamento. Quindi il fiber non potrà progredire, ma allo stesso tempo si è impostato quell'operazione all'interno di una regione che non può essere interrotta. Si può risolvere questo problema utilizzando Poll[F], che viene passato come parametro da F.uncancelable. **Poll[F]** viene utilizzato per definire del codice interrompibile all'interno della regione di codice non interrompibile. Quindi, se l'operazione da eseguire era offerer.get, si incorpora quella chiamata all'interno del Poll[F], garantendo così l'interruzione del fiber bloccato. Il codice del produttore diventa quindi:

```
def producer[F[_]: Async: Console](id: Int, counter: Ref[F, Int],
    state: Ref[F, State[F, Int]]): F[Unit] = {
        def produce(i: Int): F[Unit] =
          Deferred[F, Unit].flatMap { offerer =>
            /* Regione di codice non interrompibile */
            Async[F].uncancelable { poll =>
              state.modify {
                /* Se c'è almeno un consumatore bloccato */
                case State(capacity, queue, takers, offerers) if takers.nonEmpty =>
                  val (taker, rest) = takers.dequeue
                  State(capacity, queue, rest, offerers) -> taker.complete(i).void
                /* Se non c'è nessun consumatore bloccato e la coda di elementi non
```

```
                è piena */
              case State(capacity, queue, takers, offerers) if queue.size < capacity =>
                State(capacity, queue.enqueue(i), takers, offerers) -> Async[F].unit
              /* Altrimenti */
              case State(capacity, queue, takers, offerers) =>
                /* Si rimuove il produttore nel caso in cui ci sia un'interruzione */
                val cleanup = state.update { s => s.copy(offerers = s.offerers.
                  filter(_._2 ne offerer))}
                /* Si incorpora offerer.get in poll */
                State(capacity, queue, takers, offerers.enqueue(i -> offerer)) ->
                  poll(offerer.get).onCancel(cleanup)
            }.flatten
          }
        }

      for {
        i <- counter.getAndUpdate(_ + 1)
        _ <- produce(i)
        _ <- Console[F].println(s"Producer $id product item: $i")
        _ <- producer(id, counter, state)
      } yield ()
}
```

Il codice del consumatore si comporta nello stesso modo nel caso di **offerer.get**:

```
def consumer[F[_]: Async: Console](id: Int, state: Ref[F, State[F, Int]]): F[Unit] = {
    val consume: F[Int] =
      Deferred[F, Int].flatMap { taker =>
        Async[F].uncancelable { poll =>
          state.modify {
            /* Se la coda di elementi non `e vuota e non ci sono produttori bloccati */
            case State(capacity, queue, takers, offerers) if queue.nonEmpty
                && offerers.isEmpty =>
                  val (i, rest) = queue.dequeue
                  State(capacity, rest, takers, offerers) -> Async[F].pure(i)
            /* Se la coda di elementi non `e vuota e c'`e almeno un produttore
                bloccato */
            case State(capacity, queue, takers, offerers) if queue.nonEmpty =>
              val (i, rest) = queue.dequeue
              val ((_, release), tail) = offerers.dequeue
              State(capacity, rest, takers, tail) -> release.complete(()).as(i)
            /* Se la coda di elementi è vuota e non ci sono produttori bloccati */
            case State(capacity, queue, takers, offerers) if offerers.nonEmpty =>
              val ((i, release), rest) = offerers.dequeue
```

```
            State(capacity, queue, takers, rest) -> release.complete(()).as(i)
          /* Altrimenti */
          case State(capacity, queue, takers, offerers) =>
            /* Si rimuove il consumatore nel caso in cui ci sia un'interruzione */
            val cleanup = state.update { s => s.copy(takers = s.takers.
              filter(_ ne taker))}
            /* Si incorpora taker.get in poll */
            State(capacity, queue, takers.enqueue(taker), offerers) -> poll(taker.get)
        }.flatten
      }
    }
  for {
    i <- consume
    _ <- Console[F].println(s"Consumer $id has got item: $i")
    _ <- consumer(id, state)
  } yield ()
}
```

Il main rimane invece identico a prima.

10 RestAPI

In questa sezione viene sviluppata attraverso la libreria **http4s** (una delle librerie più utilizzate in Scala per interazioni client-server HTTP) e **Cats-effect** una **RestAPI** per la gestione di film, registi e attori. Vengono messe a disposizione le classiche operazioni **CRUD (Create, Remove, Update e Delete)** oltre ad alcune operazioni parallele grazie l'utilizzo di Cats-Effect il quale semplifica di molto la complessità. Lo scopo è anche quello di scrivere la RestAPI il più funzionale possibile grazie a Cats.

I dati vengono memorizzati in delle strutture dati, in un caso reale ovviamente i dati andrebbero memorizzati su un database.

10.1 Entità

Le entità della RestAPI sono i film, i registi e gli attori.

E' stato realizzato un object **Entities** contenente le entità:

```
object Entities {
  sealed trait Person {
    def firstName: String

    def lastName: String
  }

  case class Actor(firstName: String, lastName: String, movies: Int) extends Person {
```

```
    override def toString: String = s"$firstName $lastName"
  }

  case class Director(firstName: String, lastName: String, nationality: String,
    moviesManaged: Int) extends Person {
        override def toString: String = s"$firstName $lastName"
  }

  case class Movie(title: String, year: Int, actors: List[Actor], director: Director,
    genres: List[String], takings: Long, oscars: Int)

  case class MovieWithId(id: String, movie: Movie)
}
```

Oltre a **Movie** è stata aggiunta anche una case class **MovieWithId** per modellare un film con id, in questo modo nelle richieste POST di aggiunta di un film si utilizza Movie come modello senza id, mentre l'id può essere utilizzato per ottenere determinate informazioni su uno specifico film.

Per quanto riguarda la gestione dei film è stato creato un type **State** che non è altro che una lista di **MovieWithId** e una classe **MoviesRepository** che contiene le operazione dell'applicazione e lo stato wrappato da Ref:

```
class MoviesRepository[F[_] : Async](private val stateRef: Ref[F,
    MoviesRepository.State]) {
  /* Movies */
  def getAllMovies: F[List[MovieWithId]] = stateRef.get

  def findMovieById(id: UUID): F[Option[MovieWithId]] = stateRef.get.
    map(_.find(_.id == id.toString))

  def findMoviesByDirectorName(director: String): F[List[MovieWithId]] = stateRef.get.
    map(state =>
    director.split(" ") match {
      case Array(name, lastName) =>
        state.filter(movieWithId => movieWithId.movie.director.firstName == name
          && movieWithId.movie.director.lastName == lastName)
      case _ => Nil
    }
  )

  def addMovie(movie: Movie): F[String] = for {
    uuid <- Sync[F].delay(UUID.randomUUID().toString)
    movieToAdd = MovieWithId(uuid, movie)
    _ <- stateRef.update(state => (movieToAdd :: state))
```

```
} yield uuid

def updateMovie(id: UUID, movie: Movie): F[Unit] = stateRef.update(state =>
  state.find(_.id == id.toString) match {
    case Some(_) => MovieWithId(id.toString, movie) ::
      state.filterNot(_.id == id.toString)
    case None => state
  }
)

def deleteMovie(id: UUID): F[Unit] = stateRef.update(state =>
  state.filterNot(_.id == id.toString)
)

def findMoviesByYear(year: Int): F[List[MovieWithId]] =
  stateRef.get.map(_.filter(_.movie.year == year))

def findMoviesByGenre(genre: String): F[List[MovieWithId]] =
  stateRef.get.map(_.filter(_.movie.genres.contains(genre)))

def findMoviesByActor(actor: String): F[List[MovieWithId]] = stateRef.get.map(state =>
  actor.split(" ") match {
    case Array(name, lastName) =>
      state.filter(movieWithId => movieWithId.movie.actors.map(actor =>
        (actor.firstName, actor.lastName))
        .contains((name, lastName)))
    case _ => Nil
  })

/* Directors */
def getAllDirectors: F[List[Director]] = stateRef.get.map(_.map(_.movie.director))

def findDirectorByName(director: String): F[Option[Director]] =
  getAllDirectors.map(directors =>
    director.split(" ") match {
      case Array(name, lastName) => directors.find(d => d.firstName == name
          && d.lastName == lastName)
      case _ => None
    }
  )

def replaceDirectorFrom(movieId: String, newDirector: Director): F[Unit] =
  stateRef.update(state =>
      state.find(_.id == movieId) match {
```

```
          case Some(MovieWithId(id, movie)) =>
            val newMovie: Movie = movie.copy(director = newDirector)
            MovieWithId(id, newMovie) :: state.filterNot(_.id == movieId)
          case None => state
        }
    )

    /* Actors */
    def getAllActors: F[List[Actor]] = stateRef.get.map(_.flatMap(_.movie.actors))

    def findActorByName(actor: String): F[Option[Actor]] = getAllActors.map(actors =>
      actor.split(" ") match {
        case Array(name, lastName) => actors.find(a => a.firstName == name
          && a.lastName == lastName)
        case _ => None
      }
    )
}

object MoviesRepository {
    type State = List[MovieWithId] // Stato

    def apply[F[_] : Async](stateRef: Ref[F, State]): MoviesRepository[F] =
      new MoviesRepository[F](stateRef)

    def empty[F[_] : Async]: F[MoviesRepository[F]] = Ref.of[F, State](Nil)
      .map(MoviesRepository[F])

    private val seedState: State = List(
      MovieWithId("9127c44c-7c72-44a8-8bcb-088e3b659eca", Movie(
        "Titanic",
        1997,
        List(Actor("Kate", "Winslet", 57), Actor("Leonardo", "DiCaprio", 44),
          Actor("Billy", "Zane", 63)),
        Director("James", "Cameron", "Canadian", 44),
        List("Romance", "Drama", "Epic", "Disaster"),
        2_195_170_204L,
        11
      )
      ),
      MovieWithId("af9ce051-8541-42a9-88c4-e36d5036ad1e", Movie(
        "Top Gun",
        1986,
        List(Actor("Tom", "Cruise", 73), Actor("Kelly", "McGillis", 10),
```

```
        Actor("Val", "Kilmer", 25)),
      Director("Tony", "Scott", "British", 55),
      List("Action", "Romance", "Drama", "Adventure"),
      356_800_000L,
      0
    )
    )
  )

  def createWithSeedData[F[_] : Async]: F[MoviesRepository[F]] = Ref.of[F, State]
    (seedState).map(MoviesRepository[F])
}
```

Lo stato viene creato o vuoto da dati seed nell'entry point dell'applicazione dove si crea il server e la classe MoviesRepository viene passata ai diversi endpoints dell'applicazione.

10.2 Endpoints

Gli endpoints dell'applicazione sono i seguenti:

- **/movies**: endpoint che permette di ottenere tutti i film (anche attraverso filtri), aggiungere un film, aggiornare un film, cancellare un film, ottenere il miglior film da un API esterna.
- **/directors**: permette di ottenere tutti i registi e aggiornare un regista di un determinato film.
- **/actors**: ottenere tutti gli attori.

Per le richieste di lettura dei dati viene utilizzato il metodo HTTP GET, per quelle di scrittura POST e per gli aggiornamenti PUT.

Per ognuno degli endpoint delle entità è stato definito un **Object** con un metodo **route** che gestisse tutte le path di quel determinato endpoint, ad ogni endpoint è necessario passare il **movieRepository** che rappresenta il "controller" dell'applicazione per eseguire le diverse operazioni sullo stato. Ad esempio l'endpoint relativo agli attori:

```
object ActorRoutes {
  def route[F[_] : Async](moviesRepository: MoviesRepository[F]): HttpRoutes[F] = {
    val dsl = Http4sDsl[F]
    import dsl._

    HttpRoutes.of[F] {
      // Get all actors
      case GET -> Root / "actors" =>
        moviesRepository.getAllActors.flatMap {
          case actors if actors.nonEmpty => Ok(actors.asJson)
```

```
          case _ => NoContent()
        }
    }
  }
}
```

10.3 Operazioni parallele

Attraverso Cats-Effect è possibile eseguire più semplicemente operazioni parallele, ad esempio è possibile ottenere il miglior film tra quelli presenti nello stato a seconda dei punteggi ottenibili dal database **IMDB** che contiene informazioni relative ai film. Questo è possibile farlo utilizzando l'API di Cats-Effect che mette a disposizione metodi come **parTraverse** che permette di eseguire computazini in parallelo su una sequenza di dati:

```
case GET -> Root / "movies" / "ratings" =>
for {
    movies <- moviesRepository.getAllMovies // Tutti i film
    titles = movies.map(_.movie.title) // Titoli dei film
    /* In parallelo viene richiamato il metodo per ottenere il rating di
    ogni film */
    moviesAndRatings <- titles.parTraverse(title => moviesRepository
        .getRatingByMovie(title, client).map(r => (title, r)))
    (bestTitle, score) = moviesAndRatings.maxBy(_._2) // Si ottiene il film migliore
    response <- Ok(s"${moviesAndRatings.map(t => s"${t._1}: ${t._2}").mkString("; ")}.
        The best movie is $bestTitle with a score of $score")
} yield response
```

Mentre il metodo in MoviesRepository non fa altro che ottenere tramite delle richieste all'API di IMDB il rating di un dato film:

```
def getRatingByMovie(title: String, client: Client[F])
    (implicit jsonDecoder: EntityDecoder[F, Json] = jsonOf[F, Json]) : F[Double] = {
    val informationMovieUrl = IMDB.getInformationMovieUrl(title.replaceAll(" ", "%20"))
    for {
      informationMovieJson <- client.expect[Json](informationMovieUrl)
      movieId = root.results.index(0).id.string.getOption(informationMovieJson).get
      urlRating = IMDB.getRatingUrl(movieId)
      ratingMovieJson <- client.expect[Json](urlRating)
      rating = root.imDb.string.getOption(ratingMovieJson).get.toDouble
    } yield rating
}
```

10.4 Creazione del server

Il server viene creato utilizzando **BlazeServerBuild** in ascolto su **localhost** su porta **8080**, il server necessita di una **HttpApp** che è l'insieme di routes del server (MovieR-

outes, DirectorRoutes e ActorRoutes):

```
object WebServer extends IOApp {
  def createServer(app: HttpApp[IO]): IO[ExitCode] =
    BlazeServerBuilder[IO](runtime.compute)
      .bindHttp(8080, "localhost")
      .withHttpApp(app)
      .resource
      .use(_ => IO.never)
      .as(ExitCode.Success)

  def buildHttpApp[F[_] : Async](moviesRepository: MoviesRepository[F]): HttpApp[F] =
    (MovieRoutes.route(moviesRepository)
      <+> DirectorRoutes.route(moviesRepository)
      <+> ActorRoutes.route(moviesRepository)).orNotFound

  override def run(args: List[String]): IO[ExitCode] = for {
    repository <- MoviesRepository.createWithSeedData[IO]
    httpApp = buildHttpApp(repository)
    exitCode <- createServer(httpApp)
  } yield exitCode
}
```

Il server una volta eseguito rimane in ascolto sulla porta 8080 su localhost e attraverso un client HTTP come ad esempio Postman è possibile interagire con la RestAPI.

11 Testing in Cats

Dopo aver studiato e utilizzato la libreria Cats e in particolare Cats-effect è opportuno dedicare una sezione relativa al testing. Esistono diverse librerie tra cui **MUnit** e framework come **Weaver**, in questo caso è stata scelto di utilizzare **MUnit** per la sua semplicità di utilizzo.

11.1 Munit

La trait **munit.CatsEffectSuite** permette di scrivere test che ritornano **IO**.

Ad esempio:

```
class ExampleSuite extends CatsEffectSuite {
    test("tests can return IO[Unit] with assertions expressed via a map") {
        IO(42).map(it => assertEquals(it, 42))
    }
}
```

Oppure utilizzando **assertIO** è più leggibile:

```
class ExampleSuite extends CatsEffectSuite {
    test("alternatively, asertions can be written via assertIO") {
        assertIO(IO(42), 42)
    }

    test("map an IO") {
        assertIO(IO(42).map(_ * 2), 84)
    }
}
```

E' inoltre possibile concatenare gli IO come nell'API e utilizzare assertion multipli:

```
class MultipleAssertionsExampleSuiteTests extends CatsEffectSuite {
  test("multiple IO-assertions should be composed") {
    assertIO(IO(42), 42) *>
      assertIO(IO("Hello World!"), "Hello World!")
  }
}
```

Oppure utilizzando il for-comprehension per rendere il codice più compatto e leggibile:

```
class MultipleAssertionsExampleSuiteTests extends CatsEffectSuite {
  test("multiple IO-assertions should be composed via for-comprehension") {
    for {
      _ <- assertIO(IO(42), 42)
      _ <- assertIO(IO(42).map(_ * 2), 84)
    } yield ()
  }
}
```

11.2 Testing copia dei file

Utilizzando MUnit è stata testata la copia dei file implementata nel primo degli esperimenti con Cats-Effect.

Viene in particolare testata l'effettiva copia da un file ad un altro, per tale scopo viene utilizzato **beforeAll** e **afterAll** per creare prima dei test ed eliminare alla fine dei test due file temporanei, uno viene utilizzato come sorgente e l'altro come destinazione. Per testare la copia dei file viene utilizzato il for-comprehension per migliorare la scrittura e leggibilità del codice:

```
class CopyFilesTest extends CatsEffectSuite {
  var sourceFile: Option[File] = None
  var destinationFile: Option[File] = None

  override def beforeAll(): Unit = {
```

```
    sourceFile = Some(Files.createTempFile("tmp", ".txt").toFile)
    destinationFile = Some(Files.createTempFile("tmp", ".txt").toFile)
    /* Nel file sorgente viene scritto del testo */
    Files.write(sourceFile.get.toPath, "Hello World".getBytes(StandardCharsets.UTF_8))
  }

  override def afterAll(): Unit = {
    Files.deleteIfExists(sourceFile.get.toPath)
    Files.deleteIfExists(destinationFile.get.toPath)
  }

  test("Destination file is empty") {
    assertIO(getAmountOfBytesFromFile(destinationFile.get), -1L)
  }

  test("Origin file is not empty") {
    getAmountOfBytesFromFile(sourceFile.get).map(bytes => assert(bytes > 0))
  }

  test("Copy from source file to destination file must be effective") {
    for {
      bytesTransferred <- copy(sourceFile.get, destinationFile.get)
      bytesDestinationFile <- getAmountOfBytesFromFile(destinationFile.get)
      _ <- assertIO(getAmountOfBytesFromFile(sourceFile.get), bytesDestinationFile)
      _ <- assertIO(getAmountOfBytesFromFile(destinationFile.get), bytesTransferred)
    } yield ()
  }
}
```

11.3 Testing RestAPI

E' stata inoltre testata la **RestAPI**, in particolare vengono testate le rotte della RestAPI verificando che tutte le operazioni si comportino come previsto. Anche in questo caso viene utilizzato **beforeAll** per creare il client prima attraverso **Client.fromHttpApp**:

```
class RestAPITest extends CatsEffectSuite {
  private val TITANIC_ID = "9127c44c-7c72-44a8-8bcb-088e3b659eca"
  private val TOP_GUN_ID = "af9ce051-8541-42a9-88c4-e36d5036ad1e"
  private val movieToAdd: Movie = Movie(
    "The Terminator",
    1984,
    List(Actor("Arnold", "Schwarzenegger", 10)),
    Director("James", "Cameron", "Canadian", 44),
    List("Action", "Horror", "Thriller", "Fantasy"),
```

```
  50_000_000,
  3
)
private val directorUpdate: Director = Director("Steven", "Spielberg", "American", 20)
private var client: Option[Client[IO]] = None

override def beforeAll(): Unit = {
  val moviesRepository = MoviesRepository.createWithSeedData[IO].unsafeRunSync()
  client = Some(Client.fromHttpApp(buildHttpApp[IO](moviesRepository)))
}

private def getUriFromPath(path: String): Uri =
  Uri.fromString(s"http://localhost:8080/$path").toOption.get

/* Movies */
test("Get all movies") {
  val request: Request[IO] = Request(method=Method.GET, uri =
      getUriFromPath("movies"))
  for {
    json <- client.get.expect[Json](request)
    _ <- assertIO(IO(json.asArray.fold(0)(_.size)), 2)
    _ <- assertIO(IO(json.asArray.get.head.hcursor.downField("movie")
      .get[String]("title").toOption), Some("Titanic"))
  } yield ()
}

test("Get movie by id") {
  val request: Request[IO] = Request(method=Method.GET, uri =
      getUriFromPath(s"movies/$TITANIC_ID"))
  for {
    json <- client.get.expect[Json](request)
    _ <- assertIO(IO(json.hcursor.downField("movie")
      .get[String]("title").toOption), Some("Titanic"))
  } yield ()
}

test("Get all movies by genre") {
  val request: Request[IO] = Request(method=Method.GET, uri =
      getUriFromPath(s"movies?genre=Adventure"))
  for {
    json <- client.get.expect[Json](request)
    _ <- assertIO(IO(json.asArray.fold(0)(_.size)), 1)
    _ <- assertIO(IO(json.asArray.get.head.hcursor
      .downField("movie").get[String]("title").toOption), Some("Top Gun"))
```

```
  } yield ()
}

test("Get all movies by actor") {
  val request: Request[IO] = Request(method=Method.GET, uri =
      getUriFromPath("movies?actor=Leonardo%20DiCaprio"))
  for {
    json <- client.get.expect[Json](request)
    _ <- assertIO(IO(json.asArray.fold(0)(_.size)), 1)
    _ <- assertIO(IO(json.asArray.get.head.hcursor
      .downField("movie").get[String]("title").toOption), Some("Titanic"))
  } yield ()
}

test("Get all movies by director") {
  val request: Request[IO] = Request(method=Method.GET, uri =
      getUriFromPath("movies?director=James%20Cameron"))
  for {
    json <- client.get.expect[Json](request)
    _ <- assertIO(IO(json.asArray.fold(0)(_.size)), 1)
    _ <- assertIO(IO(json.asArray.get.head.hcursor
      .downField("movie").get[String]("title").toOption), Some("Titanic"))
  } yield ()
}

test("Get all movies by year") {
  val request: Request[IO] = Request(method=Method.GET, uri =
      getUriFromPath("movies?year=1997"))
  for {
    json <- client.get.expect[Json](request)
    _ <- assertIO(IO(json.asArray.fold(0)(_.size)), 1)
    _ <- assertIO(IO(json.asArray.get.head.hcursor
      .downField("movie").get[String]("title").toOption), Some("Titanic"))
  } yield ()
}

test("Get the best movie by rating") {
  val request: Request[IO] = Request(method=Method.GET, uri =
      getUriFromPath("movies/ratings"))
  for {
    json <- client.get.expect[Json](request)
    _ <- IO(json.asString.get).map(s => assert(s contains "Titanic"))
  } yield ()
}
```

```
test("Add a new movie") {
  val request: Request[IO] = Request(method=Method.POST, uri =
      getUriFromPath("movies")).withEntity(movieToAdd.asJson)
  for {
    json <- client.get.expect[Json](request)
    _ <- IO(json.asString.get).map(s =>
      assert(s contains "Created movie with id"))
  } yield ()
}

test("Update a movie") {
  val request: Request[IO] = Request(method=Method.PUT, uri =
      getUriFromPath(s"movies/$TITANIC_ID"))
    .withEntity(movieToAdd.asJson)
  for {
    json <- client.get.expect[Json](request)
    _ <- IO(json.asString.get).map(s =>
      assert(s contains "Successfully updated movie with id"))
  } yield ()
}

test("Delete a movie") {
  val request: Request[IO] = Request(method=Method.DELETE, uri =
      getUriFromPath(s"movies/$TITANIC_ID"))
  val requestAllMovies: Request[IO] = Request(method=Method.GET,
      uri = uri"/movies")
  for {
    json <- client.get.expect[Json](request)
    _ <- IO(json.asString.get).map(s =>
      assert(s == "Successfully deleted movie with id " +
      "9127c44c-7c72-44a8-8bcb-088e3b659eca"))
    jsonAllMovies <- client.get.expect[Json](requestAllMovies)
    _ <- assertIO(IO(jsonAllMovies.asArray.fold(0)(_.size)), 2)
  } yield ()
}

/* Actors */
test("Get all actors") {
  val request: Request[IO] = Request(method=Method.GET, uri =
      getUriFromPath("actors"))
  for {
    json <- client.get.expect[Json](request)
    _ <- IO(json.asArray.fold(0)(_.size)).map(size => assert(size > 0))
```

```
    } yield ()
  }

  /* Directors */
  test("Get all directors") {
    val request: Request[IO] = Request(method=Method.GET, uri =
        getUriFromPath("directors"))
    for {
      json <- client.get.expect[Json](request)
      _ <- IO(json.asArray.fold(0)(_.size)).map(size => assert(size > 0))
    } yield ()
  }

  test("Update director into a movie") {
    val request: Request[IO] = Request(method=Method.PUT, uri =
        getUriFromPath(s"directors?movieId=$TOP_GUN_ID"))
        .withEntity(directorUpdate.asJson)
    for {
      json <- client.get.expect[Json](request)
      _ <- IO(json.asString.get).map(s => assert(s contains "Updated successfully"))
    } yield ()
  }
}
```

12 Conclusioni

In conclusione, posso ritenermi molto soddisfatto del lavoro svolto. Inizialmente non è stato per niente facile approcciarmi a Cats e in particolare a Cats-Effect in quanto sono librerie abbastanza complesse, inoltre la libreria presenta un'API molto vasta e su internet non sono presenti così tanti esempi, quindi inizialmente mi sono trovato abbastanza in difficoltà.

Penso però che queste conoscenze mi potranno tornare davvero molto utili in futuro in quanto non solo Scala è un linguaggio molto potente e versatile ma soprattutto la conoscenza e l'utilizzo di una libreria puramente funzionale come lo è Cats è fondamentale per lavorare ad alti livelli.

Grazie alla scelta di questo progetto ho anche avuto l'opportunità di conoscere persone della community di Scala Italy e di Typelevel (contributors di Cats) che mi hanno vivamente consigliato di approfondire queste tecnologie per il mio futuro. Quindi in conclusione posso ritenermi molto soddisfatto del lavoro svolto.

13 Sviluppi futuri

Sicuramente lo studio di Cats non andrà a finire nel dimenticatoio ma anzi verranno sfruttate le conoscenze apprese e la libreria per progetti futuri, sia personali che non.