Approccio funzionale alla programmazione concorrente: Cats & Cats-effect

Paradigmi di programmazione e sviluppo

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

1 Maggio 2023

Indice

Elenco delle figure

Capitolo 1

Introduzione

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

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

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

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

Capitolo 2

Cats

2.1 Storia

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

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

2.2 Costrutti Matematici

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

2.2.1 Laws

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

2.2.2 Semigroup

Si definisce come:

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

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

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

Deve essere uguale a:

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

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

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

Infatti supponendo di usare una combine di Int avremmo che:

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

2.2.3 Monoidi

Si definisce come:

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

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

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

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

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

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

2.2.4 Funtori

Si definisce come:

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

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

Composition

La compositione si definisce come:

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

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

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

Identity

L'identità si definisce come:

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

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

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

2.2.5 Apply

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

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

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

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

```
import cats._

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

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

}

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

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

```
}
```

Infatti:

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

2.2.6 Applicative

Si definiscono come:

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

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

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

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

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

Associativity

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

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

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

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

Left Identity

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

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

Come uguaglianza:

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

Right Identity

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

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

Come uguaglianza:

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

2.2.7 Monadi

Si definisce come:

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

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

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

La seguente è una definizione più completa dell'interfaccia:

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

}
```

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

tailRecM

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

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

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

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

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

FlatMap

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

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

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

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

2.3 Typeclass

Una Typeclass è un potente strumento utilizzato nella programmazione funzionale per abilitare il polimorfismo ad hoc, più comunemente noto come overloading. Laddove molti linguaggi orientati agli oggetti sfruttano l'ereditarietà per il codice polimorfico, la programmazione funzionale tende verso una combinazione di polimorfismo parametrico (pensa parametri di tipo, come i generici Java) e polimorfismo ad hoc.
Prendiamo come esempio quello di voler costruire un meccanismo di per il calcolo dell'area di alcune forme geometriche, avremo quindi le seguenti classi:

```scala
// definizione delle case class per le forme geometriche
case class Circle(radius: Double)
case class Square(side: Double)
case class Rectangle(width: Double, height: Double)
```

Ora si può definire la typeclass Area relativa al calcolo dell'area come un trait:

```scala
// definizione della typeclass Area
trait Area[A] {
  def getArea(a: A): Double
}
```

La typeclass sopra è parametricamente polimorfica perchè 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. Definiamo ora la funzione area che vogliamo rendere rendere poliformica ad-hoc:

```scala
// definizione della funzione per calcolare l'area polimorfica ad hoc
def area[A](a: A)(implicit area: Area[A]): Double = area.getArea(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 Area definita precedentemente. Definiamo quindi i seguenti sottotipi:

```
// definizione delle istanze della typeclass Area per le forme geometriche

implicit val circleArea: Area[Circle] = new Area[Circle] {
  def getArea(c: Circle): Double = math.Pi * c.radius * c.radius
}

implicit val squareArea: Area[Square] = new Area[Square] {
  def getArea(s: Square): Double = s.side * s.side
}

implicit val rectangleArea: Area[Rectangle] = new Area[Rectangle] {
  def getArea(r: Rectangle): Double = r.width * r.height
}
```

In questo modo si aumenta il range di tipi che il metodo area può gestire, creando il polimorfismo ad-hoc. Ciò è molto importante in quanto il concetto di typeclass è alla base di Cats e Cats-Effect.

2.4 Caratteristiche e struttura

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

2.4.1 Perchè Cats

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

2.4.2 Caratteristiche

In primo luogo, Cats è molto accessibile a nuovi sviluppatori grazie al lavoro svolto per renderla intuitiva e user-friendly. Inoltre, la libreria è caratterizzata da:

- **Accessibilità:** uno dei principi della libreria è quello legato alla trasmissione dei concetti della stessa, la raccolta di successi e fallimenti in questo processo ha portato Cats ad essere e voler continuare ad essere una libreria accessibile e di facile approccio per i nuovi sviluppatori.

- **Minimalità:** la libreria punta ad essere modulare. Per farlo punta su nucleo compatto che contiene solo le typeclasses, il minimo indispensabile di strutture dati necessarie per supportarle e istanze di typeclasses per tali strutture dati e tipi di libreria standard.
- **Documentazione:** il team ritiene che avere molta documentazione sia un obiettivo molto importante in quanto ritenuto un grande passo verso il loro obiettivo di accessibilità. C'è anche un grande sforzo nel documentare molto e in modo chiaro il codice con esempi.
- **Efficienza:** un 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. Laddove è necessario trovare il compromesso tra questi due aspetti a volte in contrasto, l'obbiettivo è renderlo evidente e ben documentato.

2.4.3 Struttura

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 a questi due moduli la libreria si compone di:

- **cats-laws:** Leggi per il test delle typeclass e delle istanze.
- **cats-free:** Strutture libere come la monade libera e supporto alle typeclass.
- **cats-testkit** Libreria per scriere test sulle istanze typeclass e sulle laws.
- **algebra:** Typeclasses che rappresentano strutture algebriche.
- **alleycats-core:** istanze e classi di senza laws.

Esistono poi altri moduli separati da quelli contenuti nel repository principale di Cats, tra cui Cats-Effect su cui ci si soffermerà particolarmente nel progetto per la gestione della concorrenza, delle risorse e I/O.

Capitolo 3

Cats-Effect

3.1 Storia

Cats-effect è una libreria funzionale per la gestione degli side-effects in Scala, che si basa sui concetti della programmazione funzionale e della teoria delle categorie. Il progetto è stato sviluppato a partire dal 2017 e ha ricevuto una grande attenzione da parte della comunità Scala, diventando uno dei punti di riferimento per la gestione degli side-effects in modo sicuro e componibile. Cats-effect è stato influenzato dalle precedenti libreria come scalaz e monix, ma ha portato nuovi sviluppi in materia di side-effects, come la gestione asincrona e l'astrazione del contesto dell'effetto. Grazie alla sua ampia adozione e alla collaborazione di numerosi sviluppatori, cats-effect è diventato uno strumento fondamentale nella costruzione di applicazioni scalabili e robuste in Scala.

3.2 Cos'è 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 al coordinamento. Gli effects possono essere asincroni (callback-driven) o sincroni (restituiscono direttamente i valori). Cats Effect definisce un insieme di typeclass che definiscono un sistema puramente funzionale.
Ancora più importante è il fatto che: Cats Effect definisce un insieme di typeclasses che definiscono cosa significa essere un sistema di runtime puramente funzionale. Queste astrazioni alimentano un ecosistema fiorente costituito da framework di streaming, livelli di database JDBC, server e client HTTP, client asincroni per sistemi come Redis e MongoDB e molto altro ancora! Inoltre, è possibile sfruttare queste astrazioni all'interno della propria applicazione per sbloccare potenti funzionalità con poche o nessuna modifica del codice, ad esem-

pio risolvere problemi come l'iniezione di dipendenze, canali di errore multipli, stato condiviso tra moduli, tracciamento e altro ancora.

Cats-Effect è quello che viene chiamato un asyncronous framework, ovvero un framework che ti permette di scrivere applicazioni che possono/devono essere asincrone o che traggono un vantaggio dall'effettuare alcune operazioni concorrentemente. Di framework di programmazione asincrona ne esistono diversi, in diversi linguaggi, su JVM abbiamo RxJava, Vert.x, Netty e Akka, mentre in Rust abbiamo ad esempio Tokio . Alcuni di essi, ad esempio Akka e Netty sono implementati usando le ThreadPools, ovvero insiemi di Thread a cui posso "inviare" una computazione ed aspettarmi che uno di quei thread eventually esegua quel task e che me ne restituisca il risultato. Cats-effect è un runtime che nasce la gestione di Green Thread (thread che nasce nativamente sulla VM e non sul sistema operativo sottostante), che su JVM è stato di recente implementato grazie a progetto Loom (Loom vuol dire telaio, per correlazione con le fibre), concetto che in rust è stato implementato da tokio e da tokio è stato copiato in Cats-Effect. Entriamo più nel dettaglio del framework esplorandone i sui costrutti.

3.3 Side Effect

Per definizione 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 livello di scoping, quando modifica uno dei suoi argomenti, quando scrive su file o quando invoca altre funzioni con side-effects.

3.4 Elementi Principali

Nelle seguenti sezioni definiamo quelli che sono i principali elementi che costituiscono la libreria.

3.4.1 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 side effect prima di restituire un valore di tipo A. I valori IO sono puri e immutabili e quindi preservano la trasparenza funzionale.

```
val num: IO[Int] = IO.pure(1)  //caso in cui non ho side effect

val delayIO: IO[Int] = IO.delay({
    println("Hello World!")
    10
```

```
})  //viene valuta quando viene chiamata

//può essere anche scritta come:

val v1DelayIO: IO[Int] = IO {  //sfruttando apply
    println("Hello World!")
    10
}
```

Ma uno dei aspetti più importanti di IO è che consente lo stile monadico. Infatti IO è anche una Monade in dove ogni riga viene concatenata alla successiva usando l'operatore flatMap, il cui nome in haskell é bind e in typescript chain e spesso viene indicato con ¿¿=, ad esempio:

```
val p: Person = ???
def getName(person:Person): IO[String] = ???
def getSurname(person:Person): IO[String] = ???

//Possiamo definire la concatenazione di nome e cognome come:

val completeName: IO[String] = getName(p).flatMap(name => getSurname(p)
                                                    .map(surname => s"$name $surname"))
```

Si noti che questo approccio può essere riscritto utilizzando la for-comprehension come segue:

```
val completeName: IO[String] =
  for {
    name    <- getName(p)
    surname <- getSurname(p)
  } yield s"$name $surname"
```

3.4.2 Fibers

Un fiber è l'astrazione di un Green Thread, che su JVM è stato di recente implementato grazie a progetto Loom (Loom vuol dire telaio, per correlazione con le fibre), concetto che in rust è stato implementato da tokio e da tokio è stato copiato in Cats-Effect. Le fibre non sono altro che una implementazione delle API di green thread. L'idea alla base è basata sul fatto che che sono dei thread che puoi istanziare in gran numero e che vengono gestiti da un sistema di schedulazione. Quindi si pongono come un livello di astrazione superiore a quello dei thread e gestiscono le così dette ThreadPools, possiamo paragonarli agli Executors in Java. Un'entità (nel caso di cats-effect il suo runtime) alloca una thread pool (la cui dimensione e tipologia sono scelte secondo una logica

ottimale presente nella libreria) che viene utilizzata come se ogni singolo thread fosse una cpu e su questi thread, lo schedulatore presente nel runtime di cats-effect, schedula i "pezzi di programma" o azioni da eseguire. Che forma hanno questi pezzi di programma? Sono degli IO, infatti un fiber esegue un'azione F che è tipicamente un'istanza IO. I thread che necessita il runtime di cats-effect sono vari e si occupano principalmente di:

- gestire i comportamenti e elementi interni, come lo schedulatore;
- gestire i IO compute intensive;
- gestire le operazioni bloccanti (come la scrittura su file e le comunicazioni di rete). Il numero di thread in questa thread pool è circa (numero di CPU -1), in modo da gestire al massimo n-1 task bloccanti su n-1 cpu e di riservarne una per lo scheduler.

Gran parte di questa complessità è mascherata dalla libreria e di fatto la libreria ti permette di usare il suo runtime in modo invisibile, è sufficiente creare degli IO e usare flatmap. I fibers sono molto leggeri rispetto ai thread quindi si possono generare milioni di fibers senza influire sulle prestazioni.

3.5 Stile monadico

Nel 1991 Eugenio Moggi, professore di Informatica all'Università degli Studi di Genova, pubblicò un saggio, nel quale considerava l'utilizzo della monade, attraverso la semantica del Lambda calcolo, per gestire gli effetti collaterali, gli input e gli output, le eccezioni ed altro. Le monadi sono un'entità matematica che rispetta determinate leggi, chiamate leggi monadiche, e nella programmazione funzionale le si utilizza molto. Questa pubblicazione ha influito sul lavoro di molti informatici, tra i quali Simon Peyton Jones, il quale, poi le implementerà nel linguaggio puramente funzionale Haskell (al posto di utilizzare delle lazy-list per gestire l'IO). Come abbiamo visto Nella programmazione funzionale, una monade è una struttura che esprime una classe di computazioni concatenabili. Con l'ausilio di questa struttura, è possibile compiere, anche nei linguaggi puramente funzionali, operazioni in sequenza, un po' come avviene nei linguaggi imperativi o procedurali. Infatti, possono essere espresse, con l'ausilio delle monadi, fenomeni come gli effetti collaterali, l'IO, gestione dell'eccezioni.
In Haskell, la classe Monad viene espressa come segue:

```
class Monad m where
  (>>=)  :: m a    -> (a  -> m b) -> m b
  (>>)   :: m a    -> m b -> m b            -- non strettamente necessaria
  return :: a      -> m a
  fail   :: String -> m a                   -- non strettamente necessaria
```

Eugenio Moggi si accorse che una struttura della teoria delle categorie, la monade era molto efficace nel descrivere catene di programmi che potrebbero non terminare e con la caratteristica che il primo programma che si rompe

rompe anche la catena questo perchè le monadi sono un'entità matematica che rispetta determinate leggi, chiamate leggi monadiche, scrivere delle strutture dati monadiche che rispettano queste leggi permette di scrivere programmi in stile monadico, che utilizzando gli operatori che tutte le monadi hanno a disposizione per combinare diversi programmi tra di loro o per applicare funzioni ai valori contenuti al loro interno.

Una diretta conseguenza è l'importanza che assume la monade IO di cats-effect che consente di:

- Controllare l'esecuzione di programmi asincroni, che possono anche produrre side effects, utilizzando un'interfaccia pulita.
- Gestire applicazioni altamente simultanee concorrenti, Cats-effect non è formidabile nel gestire programmi compute intensive, ma eccelle nel gestire programmi che fanno molto I/O di rete o scrittura su più files.
- La concorrenza in IO è costruita con le fibers: un'implementazione dei green thread, come le couroutine sospendibili, gestite a runtime.
- 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. Questo è possibile grazie al fatto che IO è stato costruito come una monade, e quindi puoi usare tutti gli operatori e le proprietà di una monade.

3.6 TypeClasses

La gerarchia delle classi di tipo all'interno di Cats Effect definisce, fondamentalmente, cosa significa essere un effetto funzionale. Questo si traduce nel dire, rispettiamo una serie di regole(laws) che ci dicono cosa significa essere un numero o cosa significa avere un metodo flatMap. Lo schema seguente mostra la gerarchia di type classes di Cats Effects.

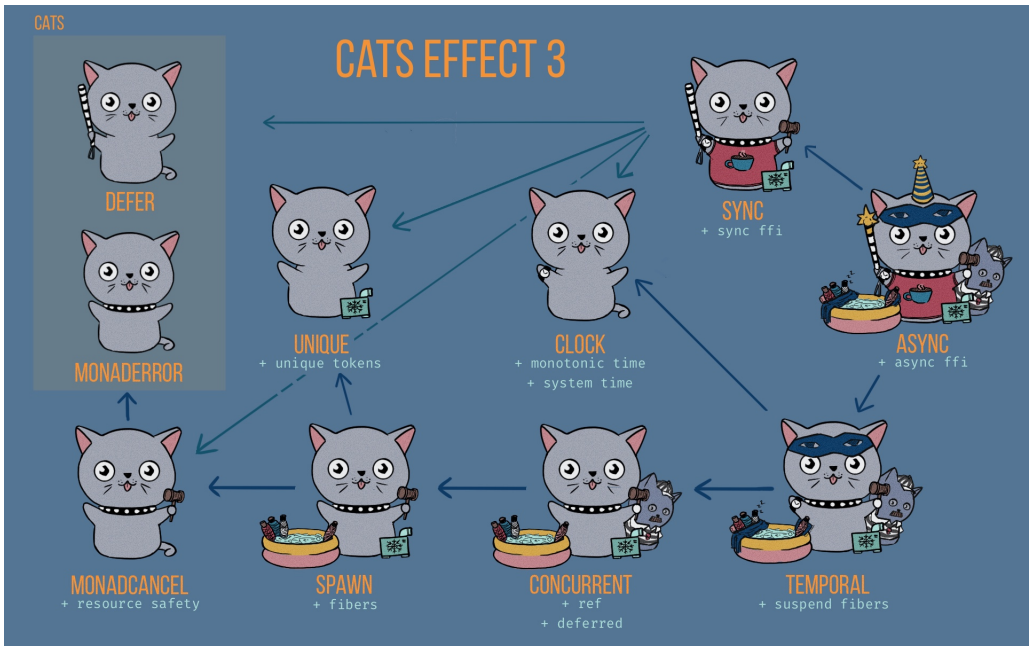

Figura 3.1: Gerarchia type classes in Cats-effect 3.

In generale, le regole per gli effetti funzionali possono essere suddivise nelle seguenti categorie elencate in ordine di potenza crescente :

- Sicurezza e interruzione/cancellazione nell'utilizzo delle risorse;
- Valutazione parallela di side-effects;
- Condivisione dello stato tra processi paralleli;
- Interazioni con il tempo;
- Acquisizione sicura dei side-effect che restituiscono valore;
- Acquisizione sicura dei side-effects che hanno una callback;

Alcune funzionalità sono assunte da Cats Effect ma definite all'interno di Cats e i suoi moduli. Tali capacità sono le seguenti:

- Trasformare il valore all'interno di un effetto mappandolo sopra di esso.
- Inserire un valore in un effetto.
- Comporre più calcoli efficaci insieme in sequenza, in modo tale che ognuno dipenda dal precedente.
- Generare e gestire gli errori.

Nel loro insieme, tutte queste capacità definiscono cosa significa essere un effetto. Ciò ti consente di comprendere e rifattorizzare il tuo codice in base a regole e astrazioni, piuttosto che dover pensare a ogni possibile dettaglio di implementazione e caso d'uso. Inoltre, consente a te e ad altri di scrivere codice molto generico che si compone insieme facendo ipotesi minime. Questa è la base dell'ecosistema Cats Effect. Di seguito vengono mostrate le typeclasses di Cats-effect con le relative API insieme ad esempi di utilizzo anche complessi.

3.6.1 MonadCancel

Tipicamente un fiber termina con tre stati sottotipi di Outcome che sono Succeeded, Errored e Canceled.

```
sealed trait Outcome[F[_], E, A]
final case class Succeeded[F[_], E, A](fa: F[A]) extends Outcome[F, E, A]
final case class Errored[F[_], E, A](e: E) extends Outcome[F, E, A]
final case class Canceled[F[_], E, A]() extends Outcome[F, E, A]
```

Ciò significa che quando si scrive codice sicuro per le risorse, dobbiamo preoccuparci dell'annullamento e delle eccezioni. La typeclass MonadCancel risolve questo problema estendendo MonadError(definito in Cats) per fornire la capacità di garantire il funzionamento dei finalizzatori quando una fibra viene cancellata. Usandolo, possiamo definire effetti che acquisiscono e rilasciano risorse in modo sicuro come ad esempio **bracket** che è l'equivalente di programmazione funzionale di try/finally:

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

Bracket fa si che se viene eseguito openFile verrà di certo 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. Oltre a bracket, MonadCancel fornisce anche un'operazione di livello inferiore, **uncancelable** che consente di eseguire azioni estremamente complesse e sensibili all'annullamento in modo sicuro e componibile.. Si pensi ad un blocco di codice protetto da un Semaforo, a cui si deve garantire la mutua esclusione ci si troverebbe in un problema legato all'acquisizione del Semaforo, che è una risorsa, che può anche comportare il blocco del fiber, e quindi potrebbe essere necessario interromperlo dall'esterno. In generale, vorremmo che l'acquisizione delle risorse sia non interrompibile, ma in questo caso particolare si deve consentire l'interruzione, altrimenti potrebbe finire per bloccare la JVM e entrare in uno stato di deadlock del sistema. L'esempio seguente mostra come Uncancelable fornisce una soluzione 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 //acquisizione della risorsa  r
      // viene utilizzato il metodo poll per eseguire l'operazione di acquisizione
      // del semaforo s in modo cancellabile
      //nel caso in cui viene interrotta viene effettuata la releaseAll per il
      //rilascio della risorsa r
      _ <- poll(s.acquire).onCancel(release(r))
      releaseAll = s.release >> release(r)
        // guarantee fa si che il rilascio sia sempre garantito, anche in caso
        //di eccezioni o interruzioni durante l'utilizzo della risorsa.
      a <- poll(use(r)).guarantee(releaseAll)
    } yield a
  }
```

Abbiamo che l'esecuzione è avvolta in uncancelable, il che significa che non dobbiamo preoccuparci che altre fibre ci interrompano nel mezzo di ciascuna di queste azioni. La primissima azione che eseguiamo è alloc, che alloca un valore di tipo R. Una volta che questo è stato completato con successo, tentiamo di acquisire il Semaphore. Se un altro fiber ha già acquisito il semaforo, il fiber corrente potrebbe bloccarsi per un po'. Vogliamo che altre fibre siano in grado di interrompere questo blocco, quindi avvolgiamo l'acquisizione in poll che riabilita l'interruzione all'interno di uncancelable per tutto ciò che avvolge. 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) che indipendentemente dal suo risultato (successed, canceled, errored) fa si che venga eseguita la releaseAll che che rilascia l'acquisizione del semaforo e della risorsa.

Un altra particolarità di MonadCancel è la capacità di auto-annullarsi. Per esempio:

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

Quanto sopra si tradurrà in un auto-annullamento e fa non verrà mai eseguita, a condizione che non sia racchiusa in un blocco uncancelable.

3.6.2 Spawn

Questa typeclass fornisce un'astrazione simile a Thread, che può essere implementata per computazioni parallele. Come già detto in precedenza i fibers sono Green thread, è infatti possibile averne anche milioni su un'unica macchina. Immaginiamo ad esempio, un'applicazione che implementa un microservizio probabilmente desidererebbe almeno un'azione simultanea per richiesta in un dato

momento, ma se il numero di azioni simultanee è limitato al numero di thread disponibili (che a sua volta è limitato al numero di processori disponibili!), è improbabile che il servizio si ridimensioni particolarmente bene; ad oggi le applicazioni risolvono questo problema attraverso l'uso di thread pool e ad e altre tecniche manuali che sebbene funzionino ragionevolmente bene, sono abbastanza facili da sbagliare e molto limitate in termini di funzionalità che si può costruire su di esse. È necessaria un'astrazione migliore, che consenta al framework e al codice utente di generare semplicemente azioni semantiche secondo necessità (ad esempio per gestire una richiesta in arrivo), mentre il runtime sottostante si occupa della mappatura di tali azioni a thread del kernel reali in un moda ottimale. Prendiamo ad esempio un server che accetta delle connessioni:

```
import cats.effect.{MonadCancel, Spawn}
import cats.effect.syntax.all._
import cats.syntax.all._

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]
}

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 in maniera continua
}
```

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, successivamente si utilizza start per creare un nuovo fiber per ogni richiesta di connessione che arriva.

Cancelation

Il vantaggio di utilizzare i fibers oltre ad essere leggeri è che a differenza dei thread della JVM sono interrompibili. Ciò significa che è possiile utilizzare **cancel** su un fiber in esecuzione per avviare il processo di ripulitura delle risorse allocate e fermarsi. Possiamo verificare questa proprietà con una monade IO come segue:

```
for {
  target <- IO.println("Catch me if you can!").foreverM.start
  _ <- IO.sleep(1.second)
  _ <- target.cancel
} yield ()
```

Il comportamento di questo codice è il seguente:

- la fibra targer stamperà un numero non deterministico di volte la frase "Catch me if you can!"
- quando la fibra principale smette di dormire per un secondo la fibra targer viene annullata. Tecnicamente, l'annullamento potrebbe non riflettersi istantaneamente nella fibra target ma appena la fibra target viene cancellata/annullata, la stampa viene interrotta e il programma termina. Utilizzando Thread è impossibile replicare questo esempio senza costruire il proprio meccanismo per l'interruzione. Con fiber per'è già tutto gestito dalla typeclass Spawn.

Joining

Di solito quando si eseguono delle operazioni in parallelo si desidera attendere che finiscano, accettare i risultati e andare avanti. L'astrazione dei Thread in Java ha l'operatore join, nelle fibre abbiamo qualcosa di simile definito come segue:

```
def both[F[_]: Spawn, A, B](fa: F[A], fb: F[B]): F[(A, B)] =
  for {
    fiberA <- fa.start
    fiberB <- fb.start

    a <- fiberA.joinWithNever
    b <- fiberB.joinWithNever
  } yield (a, b)
```

Il metodo **joinWithNever** è un metodo semplificato 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 {
    /* In caso di successo */
    case Outcome.Succeeded(fa) =>
        fa
    /* In caso di errore */
    case Outcome.Errored(e) => ???
    /* Quando il fiber viene interrotto */
    case Outcome.Canceled() => ???
}
```

Il seguente codice è un implementazione di quanto descritto sopra che stampa per 3 secondi Hello I am Fiber A e Hello I am Fiber B e in seguito racchiude un intero in IO:

```
for {
    fiberA <- (IO.println("Hello I am Fiber A").foreverM.timeoutTo(3.seconds, IO.unit)
        >> IO.pure(1)).start
    fiberB <- (IO.println("Hello I am Fiber B").foreverM.timeoutTo(3.seconds, IO.unit)
        >> IO.pure(1)).start

    /* Attendo il completamento */
    a <- fiberA.joinWithNever
    b <- fiberB.join flatMap {
        case Outcome.Succeeded(fb) => fb
        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 si cade in deadlock */
        case Outcome.Canceled() => MonadCancel[IO].canceled >> Spawn[IO].never
    }

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

3.6.3 Unique

Unique è una typeclass che genera token univoci. Il suo uso avviene principalmente a basso livello, perchè tipicamente è indatta ad identificare entità a livello di logica di applicazione come UUID. Si definisce come:

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

L'unicità è garantita dal confronto di uguaglianza tra istanze Unique.Token che non sono serializzabili o convertibili attualmente allocate. 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. Entramble le typeclasses Sync[F] che Spawn[F] estendono Unique[F] poichè entrambe possono creare valori univoci tramite delay e start (i fibers sono sempre unici).

```
    delay(new Unique.Token())
    start(new Unique.Token())
```

3.6.4 Clock

Clock è una typeclass che fornisce un sistema analogo a System.nanoTime() e System.currentTimeMillis() per la gestione del tempo. E' definito come:

```
trait Clock[F[_]] {
  def monotonic: F[FiniteDuration]
  def realTime: F[FiniteDuration]

}
```

Il seguente codice mostra un uso di realTime di clock per calcolare il tempo trascorso da un dato momento ad un altro:

```
for {
    start <- Clock[IO].realTime
    _ <- IO.sleep(3.seconds)
    end <- Clock[IO].realTime
    _ <- IO.println(end - start) // Tempo trascorso in millisecondi => 3000
} yield ()
```

3.6.5 Concurrent

Concurrent estende **Spawn** aggiungendo la capacità di allocare lo stato simultaneo sotto forma di **Ref** e **Deferred** e di eseguire varie operazioni che

richiedono l'allocazione dello stato simultaneo, inclusi **memoize** e **parTraverseN**. Ref e Deferred sono le primitive concorrenti necessarie per implementare macchine a stati concorrenti arbitrariamente complicate. Concurrent si definisce come:

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

Ref

Ref è una struttura dati simultanea che rappresenta una variabile mutabile. Viene utilizzato per mantenere uno stato accessibile e modificato in modo sicuro da molte fibre concorrenti. Diamo un'occhiata alla sua API di base:

```
trait Ref[A] {
  def get: IO[A]
  def set(a: A): IO[Unit]
  def update(f: A => A): IO[Unit]
}
```

Nello specifico abbiamo che:

- **get** legge e restituisce il valore corrente di Ref;
- **set** imposta il valore corrente di Ref;
- **update** legge e imposta atomicamente il valore corrente del file Ref.

Il seguente codice mostra un esempio dell'uso di ref dove la fibra principale avvia 100 fibre che tentano di aggiornare contemporaneamente lo stato incrementandone atomicamente di uno il valore. Una volta incrementato il valore la fibra generata si riunisce alla fibra principale, in attesa che tutte le fibre abbiano terminato. Una volta che tutte le fibre anno terminato il loro lavoro la fibra principale recupera lo stato finale producendo come output 100.

```
  override def run: IO[Unit] =
    for {
      state <- IO.ref(0)
      fibers <- state.update(_ + 1).start.replicateA(100)
      _ <- fibers.traverse(_.join).void
      value <- state.get
      _ <- IO.println(s"The final value is: $value") // => The final value is: 100
    } yield ()
```

Deferred

Deferred è una struttura dati simultanea che rappresenta una variabile di condizione. Viene utilizzato per bloccare semanticamente le fibre fino a quando non viene soddisfatta una condizione arbitraria. Diamo un'occhiata alla sua API di base:

```
trait Deferred[A] {
  def complete(a: A): IO[Unit]
  def get: IO[A]
}
```

Nello specifico abbiamo che:

- **get** blocca tutte le fibre chiamanti fino a quando Deferred non è stato completato con un valore, dopodichè restituirà quel valore.
- **complete** completa il Deferred, sbloccando tutte le fibre in attesa.

Ricordiamo che Deferred non può essere completato più di una volta.
Il seguente codice mostra un esempio di un programma dove la fibra principale genera una fibra che avvia un conto alla rovescia. Quando il conto alla rovescia arriva a 5, attende un Deferred che viene completato 5 secondi dopo dalla fibra principale. La fibra principale quindi attende il completamento del conto alla rovescia prima di uscire.

```
def countdown(n: Int, pause: Int, waiter: Deferred[IO, Unit]): IO[Unit] =
    IO.println(n) *> IO.defer {
      if (n == 0) IO.unit
      else if (n == pause) IO.println("paused...") *> waiter.get *>
                                                        countdown(n - 1, pause, waiter)
      else countdown(n - 1, pause, waiter)
    }

override def run: IO[Unit] =
    for {
      waiter <- IO.deferred[Unit]
      f <- countdown(10, 5, waiter).start
      _ <- IO.sleep(5.seconds)
      _ <- waiter.complete(())
      _ <- f.join
      _ <- IO.println("off!")
    } yield ()
```

Il programma produce il seguente output:

```
10
9
```

```
8
7
6
5
paused...
4
3
2
1
0
off!
```

Memoize

Permette di memorizzare un effetto e eseguirlo una sola volta utilizzando il risultato ripetutamente. Si definisce come:

```
def memoize[A](fa: F[A]): F[F[A]]
```

Il seguente codice mostra un uso di Memoize.

```
    val action: IO[String] = IO.println("This text is only printed once").as("action")

    val f: IO[String] = for {
        memoized <- action.memoize
        res1 <- memoized
        res2 <- memoized
    } yield res1 ++ res2

    f.unsafeRunSync()
    // => res0: String = "actionaction"
```

3.6.6 Temporal

Temporal estende Concurrent aggiungendo la capacità di sospendere una fibra per una specifica durata, mettendola in sleep. A differenza di **Sync[F].delay(Thread.sleep(duration))** che blocca il thread nel pool di calcolo **Temporal[F]#sleep** assegnata la propria classe di tipo ed è una primitiva dell'implementazione che blocca semanticamente l'esecuzione della fibra chiamante deschedulandola. Internamente viene utilizzato uno scheduler per attendere la durata specificata prima di ripianificarla. Il seguente codice mostra un esempio dell'uso di Temporal.

```
val f: IO[String] = IO.println("Before go sleep")
    >> Temporal[IO].sleep(5.second) >> IO.println("After wake up!")
```

3.6.7 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. Sync si definisce come segue:

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

Metodi di sospensione

Sync ha vari metodi di sospensione. Se il side effect non blocca il thread si usa **Sync[F].delay**

```
val counter = new AtomicLong()

val inc: F[Long] = Sync[F].delay(counter.incrementAndGet())
```

Se il side-effect è bloccante allora bisognerebbe utilizzare **Sync[F].blocking** che non solo sospende il side-effect ma sposta anche la valutazione del side-effect 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. Il seguente codice mostra l'uso di blocking in fase di lettura di un file.

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

Lo svantaggio di utilizzare **Sync[F].blocking** è che la fibra che esegue l'effetto bloccante non è interrompibile fino al completamento della chiamata. Per renderlo interrompibile si usa **Sync[F].interruptible**. Se si dispone di un'operazione di blocco di lunga durata, è consigliabile sospenderla utilizzando **Sync[F].interruptible** o **Sync[F].interruptibleMany**. Questo si comporta come blocking ma tenterà di interrompere l'operazione di blocco tramite un thread interrupt in caso di annullamento.

```
//interruptibleMany prova ad interrompere il thread ripetutamente
//finche l'operazione di blocking non viene interrotta

val operation: F[Unit] = F.interruptibleMany(longRunningOp())

val run: F[Unit] = operation.timeout(30.seconds)
```

Nell'esempio precedente viene eseguita la funzione operation che è un operazione a lungo termine interrompibile. La funzione run fa si che questa operazione venga interrotta da timeout dopo 30 secondi.

3.6.8 Async

Async è una typeclass Foreign Function Interface (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 evitando il callback hell e permette di spostare l'esecuzione in altri contesti di esecuzione. Async rappresenta la typeclass più complessa e potente di Cats-effect. L'esempio seguente definisce la più semplice FFI asincrona:

```
trait Async[F[_]] {
  def async_[A](k: (Either[Throwable, A] => Unit) => Unit): F[A]
}
```

Un esempio d'uso è la funzione **Async[F].fromFuture** che utilizza **Future#onComplete** per richiamare la callback fornita.

```
def fromFuture[A](fut: F[Future[A]]): F[A] =
  flatMap(fut) { f =>
    flatMap(executionContext) { implicit ec =>
      async_[A](cb => f.onComplete(t => cb(t.toEither)))
    }
  }
```

`async_` è comunque un po' vincolato. Non possiamo eseguire alcun effetto F nel processo di registrazione della richiamata e non possiamo nemmeno registrare un finalizzatore per annullare l'attività asincrona nel caso in cui l'esecuzione della fibra `async_` venga annullata. Per questo Async fornisce anche async, infatti:

```
trait Async[F[_]] {
  def async[A](k: (Either[Throwable, A] => Unit) => F[Option[F[Unit]]]): F[A] = {
}
```

Questa volta possiamo eseguire effetti sospesi in F. Il Option[F[Unit]] ci permette di restituire un finalizzatore da invocare se la fibra viene cancellata. Prendiamo un esempio di **Async[F].fromCompletableFuture** in versione semplificata:

```
def fromCompletableFuture[A](fut: F[CompletableFuture[A]]): F[A] =
  flatMap(fut) { cf =>
    async[A] { cb =>
      delay {
        //Invoca la callback con il risultato della completable future
        val stage = cf.handle[Unit] {
          case (a, null) => cb(Right(a))
```

```
          case (_, e) => cb(Left(e))
        }

        //Cancella la completable future se la fibra viene cancellata
        Some(void(delay(stage.cancel(false))))
      }
    }
  }
```

Più in generale possiamo considerare il Async definito come:

```
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]
}
```

Threadpool shifting

Async fornisce la possibilità 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 ()
```

Per eseguire una computazione su un pool di thread separato è possibile utilizzare `async_` come nell'esempio seguente:

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

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(1))
    }
}
```

3.7 Standard di libreria

In questa sezione vengono mostrate le principali primitive utilizzate per gestire problemi concorrenti e di sincronizzazione come deferred, ref, semafori, resource.

3.7.1 Atomic Cell

Un riferimento sincronizzato, simultaneo, mutabile. Fornisce accesso simultaneo sicuro e modifica dei suoi contenuti, garantendo che solo una fibra alla volta possa operare su di essi. Pertanto, tutte le operazioni possono bloccare semanticamente la fibra chiamante. Si definisce come:

```
abstract class AtomicCell[F[_], A] {
  def get: F[A]
  def set(a: A): F[Unit]
  def modify[B](f: A => (A, B)): F[B]
  def evalModify[B](f: A => F[(A, B)]): F[B]
  def evalUpdate(f: A => F[A]): F[Unit]
  // ... e altro
}
```

Atomic cell può essere utilizzato al posto di Semaphore e Ref come nell'esempio seguente.

```
import cats.effect.{IO, Ref}
import cats.effect.std.Semaphore

trait State
class Service(sem: Semaphore[IO], ref: Ref[IO, State]) {
  def modify(f: State => IO[State]): IO[Unit] =
    sem.permit.surround {
      for {
        current <- ref.get
        next <- f(current)
        _ <- ref.set(next)
      } yield ()
    }
}
```

La classe Service ha due parametri di input: sem e ref. Sem è un semaforo e viene utilizzato per gestire l'accesso esclusivo all'interno dello stato, ref rappresenta lo stato iniziale della classe Service, Modify viene utilizzata per modificare lo stato della classe, modify prende come argomento una funzione f che riceve lo stato corrente e restituisce un nuovo stato come effetto IO. La funzione modify acquisisce un permesso dal semaforo sem attraverso il metodo permit e quindi esegue la funzione f all'interno di un blocco surround. All'interno di questo blocco, la funzione f viene eseguita per ottenere il nuovo stato next. Successivamente, il nuovo stato viene impostato all'interno della classe Service utilizzando il metodo set della classe Ref. Tutto questo può essere replicato utilizzando un Atomic Cell come segue:

```
import cats.effect.IO
import cats.effect.std.AtomicCell

trait State
class Service(cell: AtomicCell[IO, State]) {
  def modify(f: State => IO[State]): IO[Unit] =
    cell.evalUpdate(current => f(current))
}
```

3.7.2 Deferred

Una primitiva di sincronizzazione puramente funzionale che rappresenta un singolo valore che potrebbe non essere ancora disponibile. Quando viene creato, a Deferred è vuoto. Può quindi essere completato esattamente una volta e non essere mai più vuoto. Si definisce come:

```
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 finchè 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. Il codice seguente mostra un esempio dell'uso di Deferred.

```
import cats.effect.{IO, Deferred}
import cats.syntax.all._

def start(d: Deferred[IO, Int]): IO[Unit] = {
```

```
  val attemptCompletion: Int => IO[Unit] = n => d.complete(n).void

  List(
    IO.race(attemptCompletion(1), attemptCompletion(2)),
    d.get.flatMap { n => IO(println(show"Result: $n")) }
  ).parSequence.void
}

val program: IO[Unit] =
  for {
    d <- Deferred[IO, Int]
    _ <- start(d)
  } yield ()
```

In questo codice due processi proveranno a completarsi contemporaneamente ma solo uno avrà successo, completando la primitiva Deferred esattamente una volta. Il perdente otterrà un false quando tenterà di completare un Deferred già completato o verrà automaticamente annullato dal meccanismo IO.race .

3.7.3 Ref

Un riferimento mutabile simultaneo.

```
abstract class Ref[F[_], A] {
  def get: F[A]
  def set(a: A): F[Unit]
  def updateAndGet(f: A => A): F[A]
  def modify[B](f: A => (A, B)): F[B]
  // ... and more
}
```

Fornisce accesso simultaneo sicuro e modifica del suo contenuto, ma nessuna funzionalità per la sincronizzazione, che è invece 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:

```
import cats.effect.{IO, IOApp, Sync}
import cats.effect.kernel.Ref
import cats.syntax.all._

class Worker[F[_]](id: Int, ref: Ref[F, Int])(implicit F: Sync[F]) {

  private def putStrLn(value: String): F[Unit] =
    F.blocking(println(value))

  def start: F[Unit] =
```

```
    for {
      c1 <- ref.get
      _  <- putStrLn(show"Worker #$id >> $c1")
      c2 <- ref.updateAndGet(x => x + 1) // Thread-safe
      _  <- putStrLn(show"Worker #$id >> $c2")
    } yield ()
}

object RefExample extends IOApp.Simple {

  val run: IO[Unit] =
    for {
      ref <- Ref[IO].of(0) // ref nizializzato a 0
      w1 = new Worker[IO](1, ref)
      w2 = new Worker[IO](2, ref)
      w3 = new Worker[IO](3, ref)
      _ <- List(
        w1.start,
        w2.start,
        w3.start,
      ).parSequence.void // parallelizza l'esecuzione degli elementi nella sequenza
    } yield ()
}
```

3.7.4 Semaphore

Un semaforo ha un numero non negativo di permessi disponibili. L'acquisizione di un permesso diminuisce il numero attuale di permessi e il rilascio di un permesso aumenta il numero attuale di permessi. Un'acquisizione che si verifica quando non ci sono permessi disponibili comporta il blocco della fibra fino a quando non diventa disponibile un permesso. Un semaforo si definisce come:

```
abstract class Semaphore[F[_]] {
  def available: F[Long]
  def acquire: F[Unit]
  def release: F[Unit]
  // ... altro
}
```

Ricordiamo che un semaforo con un singolo permesso funziona come una **mutex**. Immaginiamo il seguente scenario dove tre processi stanno tentando di accedere a una risorsa condivisa contemporaneamente, ma solo uno alla volta avrà accesso e il processo successivo dovrà attendere fino a quando la risorsa non sarà nuovamente disponibile (la disponibilità è una come indicato dal contatore del semaforo). Il seguente programma rappresenta la gestione del problema definito precedentemente, con l'utilizzo di un semaforo.

```scala
import cats.effect.{IO, Temporal}
import cats.effect.std.{Console, Semaphore}
import cats.implicits._
import cats.effect.syntax.all._

import scala.concurrent.duration._

class PreciousResource[F[_]: Temporal](name: String, s: Semaphore[F])
(implicit F: Console[F]) {
  def use: F[Unit] =
    for {
      x <- s.available
      _ <- F.println(s"$name >> Availability: $x")
      _ <- s.acquire // critical section
      y <- s.available
      _ <- F.println(s"$name >> Started | Availability: $y")
      _ <- s.release.delayBy(3.seconds) // end of critical session
      z <- s.available
      _ <- F.println(s"$name >> Done | Availability: $z")
    } yield ()
}

val program: IO[Unit] =
  for {
    s  <- Semaphore[IO](1) //mutex case, only one can access
    r1 = new PreciousResource[IO]("R1", s)
    r2 = new PreciousResource[IO]("R2", s)
    r3 = new PreciousResource[IO]("R3", s)
    _  <- List(r1.use, r2.use, r3.use).parSequence.void
  } yield ()
```

3.7.5 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. Spesso viene utilizzato anche per evitare annidamenti di Bracket. Il modo più semplice per costruire a Resource è con **Resource#make** e il modo più semplice per consumare una risorsa è con **Resource#use**. Le azioni arbitrarie possono anche essere revocate alle risorse con **Resource#eval**.

```scala
object Resource {
  def make[F[_], A](acquire: F[A])(release: A => F[Unit]): Resource[F, A]

  def eval[F[_], A](fa: F[A]): Resource[F, A]
```

```
}

abstract class Resource[F, A] {
  def use[B](f: A => F[B]): F[B]
}
```

Osserviamo questo esempio dove viene utilizzato Bracket per leggere e chiudere i file:

```
val concat: IO[Unit] = IO.bracket(openFile("file1")) { file1 =>
  IO.bracket(openFile("file2")) { file2 =>
    IO.bracket(openFile("file3")) { file3 =>
      for {
        bytes1 <- read(file1)
        bytes2 <- read(file2)
        _ <- write(file3, bytes1 ++ bytes2)
      } yield ()
    }(file3 => close(file3))
  }(file2 => close(file2))
}(file1 => close(file1))
```

Questo esempio può essere riscritto utilizzando Resource come segue.

```
def file(name: String): Resource[IO, File] = Resource.make(openFile(name)))(file => close(fi

val concat: IO[Unit] =
  (
    for {
      in1 <- file("file1")
      in2 <- file("file2")
      out <- file("file3")
    } yield (in1, in2, out)
  ).use { case (file1, file2, file3) =>
    for {
      bytes1 <- read(file1)
      bytes2 <- read(file2)
      _ <- write(file3, bytes1 ++ bytes2)
    } yield ()

  }
```

Si noti che le risorse vengono rilasciate in ordine inverso rispetto all'acquisizione e che entrambe acquire e release sono non interrompibili e quindi sicure in caso di cancellazione. Le risorse esterne verranno rilasciate indipendentemente dal fallimento nel ciclo di vita di una risorsa interna. Inoltre la finalizzazione avviene non appena il blocco use termina e quindi quanto segue genererà un errore poiché il file è già chiuso quando tentiamo di leggerlo:

```
open(file1).use(IO.pure).flatMap(readFile)
```

Ciò significa che la risorsa viene acquisita ogni volta che use viene invocata.
Quindi:

```
file.use(read) >> file.use(read) // apre il file due volte
file.use { file => read(file) >> read(file) } //lo aprirà solo una volta
```

Capitolo 4

Snippets

In questa sezione verranno presentati alcuni dei pezzi di codice sviluppati al fine di utilizzare e sperimentare con di semigroups, monoidi, funtori e monadi fondamentali nell'API di Cats e Cats-Effect. Infine vengono mostrati ulteriori esempi sulla gestione dei fibers.

4.1 Semigroup

Utilizzando il polimorfismo ad hoc di Cast è possibile ridefinire un proprio Semigroup custom andando a ridefinire il comportamento del metodo combine. Infatti nel seguente snippet viene mostrata una ridefinizione del comportamento combine che di base fa una somma quando passiamo elementi di tipo intero; nel nostro caso effettua una moltiplicazione.

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

  print(Semigroup[Int].combine(3, 3)) //=> 9
```

Nel seguente esempio viene anche mostrato come sia possibile definire una classe custom e combinarla a piacere. Si noti inoltre che l'operatore —+— rappresenta il metodo combine.

```
object SemigroupCustomExample extends App {
  final case class CustomClass(value: Int)

  object CustomClass {

    implicit val productIntSemigroup: Semigroup[CustomClass] =
      (x: CustomClass, y: CustomClass) => CustomClass(x.value * y.value)
  }

  print(CustomClass(3) |+| CustomClass(3)) //=> CustomClass(9)
```

```
}
```

4.2 Monoidi

Come abbiamo visto i Monoidi non sono altro che un estensione di Semigroup che aggiunge il metodo **empty**. L'esempio seguente mostra una ridefinizione del metodo empty nel caso di interi dove tipicamente è 0.

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

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

  println(Monoid[Int].combine(Monoid[Int].empty, 2)) // => 2
  println(Monoid[Int].combine(1, 2)) // => 2
}
```

Anche in questo caso possiamo definire una classe custom e applicare i metodi del monoide.

```
object CustomMonoidExample extends App {
  final case class CustomClass(value: Int)

  object CustomClass {
    implicit val customMonoid: Monoid[CustomClass] = new Monoid[CustomClass] {
      override val empty: CustomClass = CustomClass(1)

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

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

4.3 Funtori

Anche i funtori si comportano come i monoidi e i semigroup con la possibilità di definire un metodo map in più rispetto ai precedenti. Il seguente esempio mostra la definizione di un Funtore custom.

```
object CustomFunctorExample extends App {
  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))
    }
  }

  print(CustomFunctor(5).map(_ + 1)) //=> CustomFunctor(6)
}
```

4.4 Monadi

Ricordiamo che la monade è una typeclass che permette di implementare azioni di tipo flatmap. Come per i funtori è possibile definire una propria Monade customizzata 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 che comportano l'uso di grandi dimensioni di spazio.

Il seguente codice mostra un esempio della definizione di una monade custom.

```
object CustomMonad {

  implicit val customMonadInstance: 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]
        fa.flatMap(f)

      @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)
            }
        }
    }
}
```

```scala
object CustomMonadExample extends App {
  val res = for {
    a <- CustomMonad(1)
    b <- CustomMonad(2).flatMap(x => CustomMonad(x + 1))
  } yield a + b
  println(res) // CustomMonad(4)
}
```

4.5 Fiber

Le fibre sono una delle typeclass più importanti di Cats. Approfondiamo alcuni esempi con l'aiuto della definizione di un metodo **printThread** che estende **IO** come strumento di supporto per il debug dei comportamenti delle fibre; il metodo è definito come segue.

```scala
implicit class Extension[A](io: IO[A]) {

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

Utilizziamo questo metodo di IO per verificare su quale thread viene effettivamente eseguito un side effect.

```scala
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.printThread *> stringValue.printThread *> IO(ExitCode.Success)

}
```

Procediamo verificando come due side effect possano essere eseguiti dallo stresso thread.

```scala
object FiberExample extends IOApp {

  val intValue: IO[Int] = IO(1)
  val stringValue: IO[String] = IO("Scala")
```

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

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

L'output mostra come intValue e stringValue siano valutati dallo stesso thread.

4.5.1 Creation

Il seguente codice mostra come creare un nuovo fiber diverso da quello del flusso principale.

```
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.printThread.start

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

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

4.5.2 Join

L'operatore join permette di attender il risultato di una fibra. Il seguente snippet mostra un esempio d'uso di questo operatore.

```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).printThread.as(ExitCode.Success)
    //=> [io-compute-#] Succeeded(IO(1))

}
```

4.5.3 Interruption

Come già detto precedentemente a differenza dei Thread è possibile interrompere un fiber attraverso **cancel** come nel codice seguente.

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

  override def run(args: List[String]): IO[ExitCode] = {
    cancelOnAnotherThread().printThread.as(ExitCode.Success)
  }
  /*output
    [io-compute-4] starting
    [io-compute-6] cancelling
    [io-compute-0] Canceled()
```

```
  */
}
```

In questo snippet la fibra dovrebbe visualizzare "starting", aspettare un secondo e poi visualizzare "done", ma nel frattempo la fibra principale attende 500 millisecondi e interrompe la fibra creata. Nel result infatti viene indicato Canceled e done non viene stampato in tempo.

4.5.4 Racing

Come abbiamo visto le fibre sono l'elemento principale per la concorrenza; gli snippet di questa sezione mostrano come eseguire concorrentemente due task su fibre diverse attraverso il metodo race. Race di IO ritorna un IO[Either[A, B]] dove A e B sono i tipi di ritorno. La fibra perdente è quella la cui azione viene completata per seconda e viene interrotta. Il seguente snippet mostra un esempio d'uso di race.

```
object Racing extends IOApp.Simple {

  val valuableIO: IO[Int] = {
    IO("task starting").printThread *> IO.sleep(1.second).printThread >> IO(
      "task completed"
    ).printThread *> IO(1).printThread
  }

  val vIO: IO[Int] = valuableIO.onCancel(IO("task: cancelled").printThread.void)

  val timeout: IO[Unit] = {
    IO("timeout: starting").printThread >> IO.sleep(500.millis).printThread >> IO(
      "timeout: finished"
    ).printThread.void
  }

  def race(): IO[String] = {
    // The losing fiber gets canceled
    val firstIO: IO[Either[Int, Unit]] =
      IO.race(vIO, timeout) // IO.race => IO[Either[A, B]]

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

  def run: IO[Unit] = race().printThread.void
```

```
 /* output
    [io-compute-7] task starting
    [io-compute-5] timeout: starting
    [io-compute-5] ()
    [io-compute-5] timeout: finished
    [io-compute-5] task: cancelled
    [io-compute-1] timeout won
 */

}
```

In questo esempio vengono eseguiti due task:

- 1. **vIO:** che visualizza una stringa, aspetta un secondo, visualizza un'altra stringa e ritorna un intero in IO. Nel caso di interruzione visualizzerà **task: cancelled**
- 2. **timeout:** che visualizza una stringa, aspetta un mezzo secondo e visualizza un'altra stringa.

La competizione viene vinta da timeout che produce l'output **timeout won** in quanto il suo comportamento prevede di dormire per mezzo secondo.

Immaginiamo di non voler interrompere la fibra perdente ma di volerla gestire diversamente. Lo snippet seguente mostra un esempio dove la fibra viene comunque interrotta ma per scelta, aprendo però ad altri comportamenti di gestione.

```
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").printThread *> IO(outcomeA).printThread
      case Right((fiberA, outcomeB)) =>
        fiberA.cancel *> IO("second task won").printThread *> IO(outcomeB).printThread
    }
  }

  val ioA: IO[Int] =
    IO.sleep(1.second).as(1).onCancel(IO("first cancelled").printThread.void)
  val ioB: IO[Int] =
```

```
    IO.sleep(2.second).as(2).onCancel(IO("second cancelled").printThread.void)

  def run: IO[Unit] = racePair(ioA, ioB).void

  /*output
    [io-compute-1] second cancelled
    [io-compute-1] first task won
    [io-compute-1] Succeeded(IO(1))

  */
}
```

Lo snippet seguente mostra che racePair che ritorna un IO[Either[(OutcomeIO[A],FiberIO[B]), (FiberIO[A], OutcomeIO[B])]] dove ognuna delle due tuple contiene l'outcome del task e il fiber dell'altro task da gestire.

Capitolo 5

Use Cases

Dopo aver introdotto le typeclasses di Cats-Effect e le principali primitive, in questa sezione vengono descritti degli esperimenti più complessi realizzati con Cats-Effect. Ci si focalizza in particolare sulla gestione delle risorse, I/O e su alcuni dei classici problemi di concorrenza che sono i principali use cases in cui viene utilizzato Cats-Effect.

5.1 Gestione delle risorse con IO

Questo use case ha come obiettivo 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 ovvero apertura/chiusura e lettura/scrittura. L'implementazione fa si che l'istanza IO restituirà la quantità di byte copiati da un file all'altro. Non dimentichiamo che potrebbero verificarsi degli errori, ma quando si lavora con qualsiasi istanza IO, questi dovrebbero essere incorporati nell'istanza IO stessa. Ciò implica che nessuna eccezione viene sollevata al di fuori dell'IO e quindi non è necessario l'utilizzo di 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 come mostrato nel codice seguente.

```
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) //prende le risorse
      .tupled // da (IO[InputStream], IO[OutputStream]) a IO[(InputStream, OutputStream)]
      .bracket {
        case (in, out) => transfer(in, out) // usa le risorse
      } {
        case (in, out) => // libera le risorse
```

```
          (IO(in.close()), IO(out.close()))
            .tupled // da (IO[Unit], IO[Unit]) a IO[(Unit, Unit)]
            .handleErrorWith(_ => IO.unit).void //gestisce l'errore
      }
}
```

Ricordiamo che quando si usa bracket se si verifica un problema nell'ottenere la risorsa la parte di rilascio della risorsa non viene mai eseguita.

Osserviamo ora 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 { //con blocking spostiamo l'esecuzione su un thread pool dedicato
      amount <- IO.blocking(originFile.read(buffer, 0, buffer.length))
      /* Si richiama ricorsivamente finchè 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.pure(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. Ricordiamo che 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 come nel caso di first.flatMap(_⇒second), nel codice precedente indica che dopo ogni scrittura si deve chiamare ricorsivamente transmit accumulando i byte trasferiti. Il main dell'applicazione estende **IOApp** quando lo si esegue 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, come mostrato nel codice seguente.

```
object SimpleCopyFile extends IOApp {
  override def run(args: List[String]): IO[ExitCode] = {
    for {
    /* Se gli argomenti non sono 2 lancia un eccezione*/
      _ <- if (args.length != 2) IO.raiseError(new IllegalArgumentException("Add origin and
      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.
```

```
    } yield ExitCode.Success
  }
}
```

5.1.1 Versione Polimorfa

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

```
object PolymorphicUtils {
  private val BUFFER_SIZE = 1024 * 10
    /* Crea lo stream di input wappato da Resource */
  def inputStream[F[_] : Sync](file: File): Resource[F, FileInputStream] = {
    Resource.make { // acquisisce la risorsa
      Sync[F].blocking(new FileInputStream(file))
    } { dataInputStream => // libera la risorsa
      Sync[F].blocking(dataInputStream.close()).handleErrorWith(_ => Sync[F].unit)
    }
  }
  /* Crea lo stream di output wappato 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)
  }
  /* funzioni viste in precedenza*/
  private def transmit[F[_] : Sync](originFile: InputStream,
  destinationFile: OutputStream, buffer: 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 in copy. L'unica differenza è nel main dove si imposta IO come F, come mostrato nel codice seguente.

```
object PolymorphicCopyFile extends IOApp {
  override def run(args: List[String]): IO[ExitCode] = {
    for {
      _ <- if (args.length != 2) IO.raiseError(
      new IllegalArgumentException("Add origin and destination files as args"))
      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
  }
}
```

5.1.2 Avanced version

Una versione più avanzata 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 PolymorphicCopyFile extends IOApp {
  override def run(args: List[String]): IO[ExitCode] = {
    for {
      _ <- if (args.length != 2) IO.raiseError(
        new IllegalArgumentException("Add origin and destination files as args"))
      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
  }
}
```

5.2 Concorrenza

Per evidenziare alcune peculiarità della librerià si è deciso di affrontare alcuni dei classici problemi di riferimento presenti in letteratura legati alla concorrenza.

5.2.1 Produttori consumatori

Il problema prevede che 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. Questo problema mette in evidenza l'utilizzo di primitive concorrenti come **Ref** e **Deferred**. In questa sezione verranno mostrati alcuni scenari di implementazione del problema attraverso Cats-effect.

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 viene quindi definito come segue.

```
  def producer[F[_]: Sync: Console](queue: Ref[F, Queue[Int]], counter: Int): F[Unit] = {
    for {
      _ <- if(counter % 10000 == 0) Console[F].println(s"Produced item $counter ") else Sync
/*  Aggiunge un elemento alla coda, solo un fiber alla volta ci può accedere */
      _ <- queue.getAndUpdate(_.enqueue(counter + 1))
      _ <- producer(queue, counter + 1)
    } yield ()
  }
```

Il metodo non fa altro che stampare l'elemento prodotto, modificare la coda attraverso il metodo getAndUpdate di Ref che fornisce la coda corrente, quindi usiamo .enqueue per inserire il valore successivo counter+1. Notare che il % 10000 è per rallentare il produttore in quanto produce molto velocemente. Il metodo del consumatore è molto simile, inizialmente prende l'elemento dalla coda se esiste tramite dequeueOption, poi lo stampa come mostra il codice seguente.

```
  def consumer[F[_]: Sync: Console](queue: Ref[F, Queue[Int]]): F[Unit] = {
    for {
     /* rimuove un elemento dalla coda se non è vuota */
      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].uni
      _ <- consumer(queue)
    } yield ()
  }
```

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

```
object ProducerConsumerExample extends IOApp {
  override def run(args: List[String]): IO[ExitCode] = {
    for {
      queue <- Ref.of[IO, Queue[Int]](Queue.empty[Int])
      res <- (consumer(queue), producer(queue, 0))
        .parMapN((_, _) => ExitCode.Success) /*Avvia produttori e consumatori in parallelo*/
        .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. In questo ultimo scenario se c'è un errore nei fibers la join non viene completata e ritornata. Invece parMapN, permette di gestire possibili errori quindi è preferibile. Si noti che 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 diversi produttori e consumatori per bilanciare la produzione e il tasso di consumo.

Unbounded Producer-consumer

Nell'implementazione 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 anche bloccare il fiber del consumatore se la coda è vuota finchè 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 riempirà. Quindi bisogna tenere conto anche delle istanze Deffered create quando la coda era vuota, 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, il codice seguente mostra come è definito State.

```
/* dove 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 come segue.

```
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 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** serve solo ad identificare il consumatore. Il produttore invece:

- se la coda dei takers è vuota mette semplicemente in coda l'elemento prodotto
- se è presente un taker, prende un taker dalla testa di takers e lo completa sbloccandolo

Il produttere si definisce come segue.

```
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 {
      /* 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 ()
```

Il main è definito come segue. Si noti che al producer viene passato anche un counter inizialmente a 1 protetto da un Ref.

```
object ProducerConsumerExample 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)
      producers = List.range(1, 11).map(producer(_, counter, state)) //-> 10 produttori
      consumers = List.range(1, 11).map(consumer(_, state)) //-> 10 consumatori
      res <- (producers ++ consumers)
        .parSequence.as(ExitCode.Success) /* Esegue sia i produttori che consumatori in par
        .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 che produttori si migliora l'equilibrio tra essi, nonostante questo 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.

Bounded Producer-Consumer

Come detto in precedenza per ottimizzare il problemma vanno limitati i produttori. Avere una pila di elementi limitata implica che i produttori rimangano bloccati quando la pila 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 seguendo lo stesso meccanismo dei consumatori.

```
/* Aggiunta la capacità massima della coda */
case class State[F[_], A](capacity: Int, queue: Queue[A], takers: Queue[Deferred[F, A]],
                          offerers: Queue[(A, Deferred[F, Unit])])

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

In questo nuovo scenario un consumatore imabattersi in quattro tipi di casistiche che dipendono dalla coda di elementi e dalla coda dei produttori bloccati(offerers):

- 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 come mostrato in seguito.

```
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.isEmpt
          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.

Il codice del produttore diventa il seguente.

```
  def producer[F[_]: Async: Console](id: Int, counter: Ref[F, Int], state: Ref[F, State[F, I
    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' nessun consumatore bloccato e la coda di elementi
        non  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) // Update the item
      _ <- produce(i)
      _ <- Console[F].println(s"Producer $id product item: $i")
      _ <- producer(id, counter, state)
    } yield ()
  }
```

Il main non subisce variazioni. L'unico cambiamento è legato all' aggiunta della capacità massima della coda.

```
object ProducerConsumerExample 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)) //->10 produttori
      consumers = List.range(1, 11).map(consumer(_, state)) //10->consumatori
      res <- (producers ++ consumers)
        .parSequence.as(ExitCode.Success)
        .handleErrorWith {
          t => Console[IO].errorln(s"Error caught: ${t.getMessage}").as(ExitCode.Error)
        }
    } yield res
  }
}
```

In questo scenario tutti i problemi precedenti sono stati risolti ma potrebbe verificarsi che uno dei fiber che gestisce un consumatore o un produttore venga interrotto, queste implementazioni non gestiscono questa possibilità. Per gestire le interruzioni bisogna utilizzare i metodi **uncancelable** e **Poll**.

Cancelation-safe Producer-Consumer

In questo esempio vedremo come è gestire il problema delle interruzioni. E' possibile 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 poichè 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 viene modificato come segue.

```
  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**, l'esempio è descritto dal blocco seguente.

```
def consumer[F[_]: Async: Console](id: Int, state: Ref[F, State[F, Int]]): F[Unit] = {
  val consume: F[Int] =
    Deferred[F, Int].flatMap { taker =>
    /* Regione di codice non interrompibile */
      Async[F].uncancelable { poll =>
        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.isEm
            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) =>
              /* 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).onCancel(cleanup)
          }.flatten
        }
      }
    for {
      i <- consume
      _ <- Console[F].println(s"Consumer $id has got item: $i")
      _ <- consumer(id, state)
    } yield ()
  }
```

5.2.2 problema dei filosofi a cena

L'esempio fu descritto nel 1965 da Dijkstra, che se ne servì per esporre un problema di sincronizzazione. Cinque filosofi siedono ad una tavola rotonda con un piatto di spaghetti davanti e una forchetta a sinistra. Ci sono dunque cinque filosofi, cinque piatti di spaghetti e cinque forchette. Si immagini che la vita di un filosofo consista di periodi alterni di mangiare e pensare, e che ciascun filosofo abbia bisogno di due forchette per mangiare, ma che le forchette vengano prese una per volta. Dopo essere riuscito a prendere due forchette il filosofo mangia per un po', poi lascia le forchette e ricomincia a pensare. Il problema consiste nello sviluppo di un algoritmo che impedisca lo stallo (deadlock) o la morte d'inedia (starvation). Il deadlock può verificarsi se ciascuno dei filosofi tiene in mano una forchetta senza mai riuscire a prendere l'altra. Il filosofo F1 aspetta di prendere la forchetta che ha in mano il filosofo F2, che aspetta la forchetta che ha in mano il filosofo F3, e così via in un circolo vizioso. La situazione di starvation può verificarsi indipendentemente dal deadlock se uno dei filosofi non riesce mai a prendere entrambe le forchette. La soluzione implementata prevede di numerare le forchette ed esigere che vengano prese in ordine numerico crescente, analogamente ad un caso di allocazione gerarchica delle risorse. In questa soluzione i filosofi sono denominati F1, F2, F3, F4 e F5, mentre le forchette alla loro sinistra sono rispettivamente f1, f2, f3, f4 e f5. Il primo filosofo F1 dovrà prendere la prima forchetta f1 prima di poter prendere la seconda forchetta f2. I filosofi F2, F3 e F4 si comporteranno in modo analogo, prendendo sempre la forchetta fi prima della forchetta fi+1. Rispettando l'ordi-

ne numerico ma invertendo l'ordine delle mani, il filosofo F5 prenderà prima la forchetta f1 e poi la forchetta f5. Si crea così un'asimmetria che serve ad evitare i deadlock. Si è deciso quindi di risolvere il problema utilizzando Cats-effect e **Semaphore** per garantire la mutua esclusione. Di seguito viene riportato il codice di implementazione del comportamento di un filosofo.

```
object Philosopher {

  class Philosopher(
      val id: Int,
      val leftFork: Semaphore[IO],
      val rightFork: Semaphore[IO]
  ) {
    def think: IO[Unit] =
      IO(println(s"Philosopher $id is thinking")) *> IO.sleep(2.seconds)

    def eat: IO[Unit] = IO(println(s"Philosopher $id is eating")) *>
      IO.sleep(1.seconds) *> IO(println(s"Philosopher $id end eating"))

    def acquireForks: IO[Unit] =
      for {
        _ <- leftFork.tryAcquire.ifM(
          IO(println(s"Philosopher $id acquired left")) *>
            rightFork.tryAcquire.ifM(
              IO(println(s"Philosopher $id acquired right with left")), //true case
              IO(println(s"Philosopher $id release left")) *> leftFork.release //false case
            ),
          acquireForks
        )
      } yield ()

    def releaseForks: IO[Unit] =
      for {
        _ <- IO(println(s"Philosopher $id release left")) *> leftFork.release
        _ <- IO(println(s"Philosopher $id release right")) *> rightFork.release

      } yield ()

    def dine: IO[Unit] =
      (think *> acquireForks *> eat *> releaseForks).foreverM
  }
}
```

Rappresentando le forchette come semafori di mutua esclusione possiamo notare che il fulcro di questa implementazione sta nel metodo acquireForks che sfrutta tryAcquire che viene utilizzato per acquisire le due forchette; tryAcquire fa si

che il filosofo provi a prendere la forchetta sinistra, se ci riesce prova a prendere la destra altrimenti riprova ad acqusire le forchette. Una volta che il filosofo ha la forchetta di sinistra prova a prendere la forchetta di destra, se ci riesce mangia se non è disponibile libera la forchetta sinistra (questo sblocca altri filosofi che stanno aspettando quella stessa forchetta per mangiare e evita stati di starvation o deadlock. Di seguito viene mostrato il codice Main di questa implementazione. Si noti che l'ultimo filosofo come specificato in questa soluzione che non prevede ticket avrà come forchetta di sinistra quella alla sua destra e come forchetta di destra la forchetta sinistra.

```
object DiningPhilosophersExample extends IOApp {
  override def run(args: List[String]): IO[ExitCode] = {
    val numPhilosophers = 5
    for {
      s <- Semaphore[IO](1)
      forks = List.fill(numPhilosophers)(s)
      philosophers = (0 until numPhilosophers - 1).map { i =>
        new Philosopher(
          i,
          forks(i),
          forks((i + 1) % numPhilosophers)
        ) // id, leftFork, rightFork
      }
      lastPhilosopher = new Philosopher(
        numPhilosophers - 1,
        forks(0),
        forks(numPhilosophers - 1)
      )

      res <- (philosophers :+ lastPhilosopher)
        .map(_.dine)
        .toList
        .parSequence
        .as(
          ExitCode.Success
        ) // Run producer and consumer in parallel until done
        .handleErrorWith { t =>
          Console[IO]
            .errorln(s"Error caught: ${t.getMessage}")
            .as(ExitCode.Error)
        }
    } yield res
  }
}
```