Realizzazione di un Framework con Cats Effect IO

Corso di Paradigmi di Programmazione e Sviluppo

Daniele Commodaro
daniele.commodaro@studio.unibo.it

Gennaio 2022

Cats Effect é uno dei sistemi di effetti piú popolari di Scala. L'obbiettivo di questo progetto é quello di approfondire alcune delle caratteristiche piú importanti di questa libreria, e di utilizzare alcuni degli strumenti forniti per la realizzazione di un piccolo framework che permetta di gestire gli effetti di un'applicazione, e che fornisca alcune delle operazioni necessarie per risoluzione di problemi di concorrenza e per l'IO.

Indice

1 Introduzione

1.1 Che cos'é un'effetto

L'effetto é una descrizione di un'azione, piuttosto che l'azione stessa. Descrive cosa puó accadere quando l'azione viene eseguita.

1.2 Qual'é la necessitá di un sistema di effetti

Un sistema di effetti aiuta gli sviluppatori a controllare l'esecuzione. Possiamo descrivere gli eventi in dettaglio e comporre piú effetti per creare una lunga catena di effetti correlati. Possiamo quindi eseguire gli effetti solo quando é necessario. Ció significa che i sistemi di effetti ci consentono di gestire gli eventi in modo pigro. Questo non é possibile utilizzando altri strumenti come le Future di Scala. I sistemi di effetti come Cats Effect aiutano a descrivere un calcolo e ad eseguirlo quando necessario.

2 La monade IO

Poiché IO é una monade, possiamo utilizzare su di essa le operazioni comuni come map e flatMap per concatenare le varie IO. In alternativa si puó anche usare il for-comprehension. É possibile usare il metodo flatten per evitare IO nidificati; Cats Effect fornisce inoltre un altro metodo di utilitá per eseguire questa operazione utilizzando defer, che in caso di nidificazione restituisce un IO appiattito. Altri metodi semplici e utili sono void per scartare il risultato di un IO, mentre per sostituire il valore di un IO con un altro valore, possiamo usare as invece di map. Per creare un effetto che stampi il contenuto sulla console, si puó utilizzare il metodo IO.println

2.1 Conversione di tipi comuni in IO

IO fornisce alcuni metodi di utilitá per trasferire i tipi di Scala piú comuni in IO:

- IO.fromOption: per passare da Option[A] a IO[A]
- IO.fromTry: da Try[A] a IO[A]
- IO.fromEither: da Both[Throwable, A] a IO[A]

Si puó anche portare Future in IO utilizzando il metodo fromFuture. Tuttavia, per sospendere l'esecuzione di una Future occorre prima wrappare la Future in un IO.

2.2 Concatenare IO

Si possono usare map, flatMap e for-comprehension per concatenare gli IO. Ci sono peró altri combinatori disponibili per concatenare diversi IO, e hanno forma infissa. Il combinatore *> esegue l'IO di sinistra, poi quello di destra e scarterá il risultato del primo IO eseguito. Allo stesso modo il combinatore <* esegue gli IO nello stesso ordine, ma mantiene il risultato del primo IO e ignore il risultato del secondo. C'é un altro combinatore » che é simile a *>. L'unica differenza é che viene valutato pigramente e quindi é sicuro per lo stack, ad esempio nei casi di ricorsione. Esiste un'altra versione di combinatori &> e <& che sono simili a » e «, ma eseguono gli IO in parallelo invece che in sequenza.

2.3 IO sleep e never

Cats Effect fornisce un modo molto pulito per far dormire in modo asincrono un IO per un tempo specificato. Questo metodo prende il nome di blocco semantico. Bisogna notare che a differenza di Thread.sleep, IO.sleep non blocca il thread corrente. Questo perché Cats gestisce internamente i thread ed esegue la sospensione in modo asincrono. É possibile creare un IO che non termina mai usando il metodo IO.never.

3 Esecuzione di un effetto

Per eseguire un effetto abbiamo principalmente 3 modi possibili:

3.1 Utilizzo dell'app Scala e del metodo main

Possiamo scrivere un normale oggetto Scala ed estenderlo con App o implementare il metodo main. Quindi possiamo usare il metodo unsafeRunSync() sulla monade IO per eseguire l'effetto. Tuttavia, occorre fornire un'istanza IORunTime implicita. Puó essere fornita importanto cats.effect.unsafe.implicits.global.

3.2 Estensione con IOApp

Invece di utilizzare il metodo principale predefinito, é possibile utilizzare IOApp. Bisogna estendere l'app principale con IOApp e sovrascrivere il metodo run. Poiché il metodo run deve restituire IO[ExitCode], possiamo eseguire map sulla monade IO per restituire il codice necessario.

3.3 Estensione con IOApp.Simple

Invece di estendere IOApp, é possibile estendere IOApp.Simple. Una volta esteso si puó sovrascrivere il metodo run. L'unica differenza tra il metodo run di IOApp e quello di IOApp.Simple é che il metodo run di IOApp.Simple restituisce IO[Unit], che é piú comune in generale.

4 Gestione delle eccezioni

Poiché la monade IO descrive un effetto, é possibile che la valutazione non riesca in fase di esecuzione. Il tipo di dato IO puó acquisire gli errori. É possibile usare raiseError per sollevare un'eccezione e far fallire un IO.

4.1 Creazione di IO con errori

Se si vuole creare un IO che fallisca sotto determinate condizioni, si puó usare IO.raiseWhen. Questo metodo solleva l'eccezione fornita se la condizione corrisponde altrimenti restituisce IO[Unit]. Ció é particolarmente utile per gestire alcune situazioni indesiderate. Allo stesso modo, IO.raiseUnless solleva un'eccezione quando la condizione non corrisponde.

4.2 Gestione degli errori

Possiamo gestire gli errori di un IO usando i metodi handleError ed handleErrorWith. L'handler presente in handleErrorWith dovrebbe restituire un altro IO. É possibile anche utilizzare il metodo redeem per gestire insieme i casi di successo e fallimento.

5 Attraversamento di IO

In questa parte vengono esaminati i diversi modi per combinare ed attraversare molteplici IO. La differenza rispetto ai metodi visti in precedenza sta nel fatto essi restituiscono solo uno dei risultati degli IO partecipanti, a meno che non vengano gestiti in modo esplicito.

5.1 Utilizzo dei metodi tupled e mapN

Si puó ottenere il risultato di tutti gli IO partecipanti utilizzando il metodo mapN di Cats. É necessario aggiungere l'istruzione import cats.syntax.apply._ per portare il metodo nell'ambito.

Se c'é bisogno di combinare due IO in una tupla senza fare alcuna trasformazione, é possibile usare il metodo tupled. Entrambi i metodi mapN e tupled eseguono gli IO in sequenza.

5.2 Esecuzione parallela utilizzando parMapN e parTupled

Per eseguire gli IO in parallelo, é possibile usare parMapN. É simile a &> ma aiuta ad applicare piú trasformazioni sui risultati. Per utilizzare il comando bisogna aggiungere la dichiarazione di importazione: import cats.syntax.parallel._. Se si vuole calcolare entrambi gli IO in parallelo e ottenere il risultato come una tupla, si puó usare parTupled.

5.3 Conversione di IO utilizzando traverse e sequence

Nel caso si abbia una raccolta di IO, puó diventare difficile utilizzare tale raccolta insieme ad altri IO. Quello che si fa in questo caso é convertire List[IO] in IO[List]. Per fare questo, in primo luogo si porta un'istanza della typeclass della raccolta richiesta dalla libreria di Cats. Dopodicé si applica il metodo sequence alla raccolta di IO. Esiste anche il metodo traverse molto potente che prende in ingresso due parametri curried. Il primo é la raccolta di IO, mentre il secondo é una funzione che elabora ciascuno degli IO all'interno della raccolta.

5.4 Esecuzione parallela per traverse e sequence

Simile a mapN, esiste una versione parallela per traverse e sequence. I comandi sono denominati parTraverse e parSequence. Questi metodi sono disponibili come metodi di estensione di Cats e devono essere importati per poter essere utilizzati attraverso import cats.syntax.parallel._

6 Concorrenza e parallelismo

Il parallelismo é un modo in cui i calcoli vengono eseguiti contemporaneamente. Questo generalmente utilizza piú core del processore per eseguire ogni calcolo. I metodi parSequence, parTraverse etc utilizzano questo approccio. La concorrenza non implica necessariamente che i calcoli vengano eseguiti contemporaneamente. Invece, sono iterfogliati. Significa che piú attivitá vengono eseguite dallo stesso thread passando da una all'altra ogni volta che é possibile. In altre parole, puó essere considerato come un destreggiarsi tra compiti diversi senza perdere tempo libero.

6.1 Fibre

Le fibre sono come un thread molto leggero. Sono gli elementi costitutivi del modello di concorrenza in Cats. Possiamo creare tutte le fibre necessarie senza preoccuparci del thread e del pool di thread. Queste fibre possono essere create, unite o cancellate. Il runtime di Cats Effect si occupa di programmare queste fibre sui thread reali. Quindi uno sviluppatore non deve preoccuparsi dei thread sottostanti o del modo di gestirli. Le fibre implementano il blocco semantico, il che significa che i thread sottostanti non vengono bloccati dal runtime di Cats Effect. Inoltre, una fibra non é bloccata su un particolare thread. Pertanto, una fibra puó essere eseguita su piú thread prima che l'attivitá sia completata a seconda della disponibilitá dei thread. Queste fibre possono essere vagamente paragonate agli attori Akka, anche se il modello dell'attore é un paradigma completamente diverso.

6.1.1 Utilizzo

É possibile invocare il metodo .start su un IO per creare una fibra. Possiamo usare il metodo .join su una fibra per attendere il risultato. Su puó usare il metodo di estensione trace per stampare il nome del thread su cui viene eseguito l'IO. In questo modo é posssibile verificare l'esecuzione concorrente delle fibre. Possiamo invocare .join sull'handler di una fibra per ottenere il risultato

della sua esecuzione. Dall'esecuzione una fibra restituisce un tipo Outcome, che contiene le diverse possibilitá di esecuzione. Queste sono Succeeded, Errored e Canceled; possiamo eseguire un pattern match sul risultato per gestire i risultati.

6.1.2 Cancellazione IO

Una delle caratteristiche piú potenti di una fibra é la capacitá di annullarne l'esecuzione. Si puó invocare il metodo cancel sull'handler della fibra per annullare la fibra in corso. É possibile anche allegare un cancel hook ad un IO usando onCancel, in questo modo quando la fibra viene cancellata, richiama l'hook(detto anche finalizzatore) onCancel. Bisogna tenere presente che onCancel crea un nuovo tipo di IO per il finalizzatore e questo non avrá alcun impatto sull'IO originale.

7 Gestione delle risorse

Le risorse sono generalmente cose come file, socket, connessioni db, etc.. che vengono utilizzate per leggere e scrivere da fonti esterne. L'approccio generale consiste nell'acquisire l'accesso a una risorsa, eseguire un'azione e chiudere la risorsa. La gestione impropria di queste risorse come la mancata chiusura dopo l'utilizzo etc.. puó causare situazioni molto gravi in qualsiasi applicazione. Ad esempio, se stiamo aprendo un file da leggere e ci dimentichiamo di chiuderlo al termine, ció porterá a perdite di memoria e all'eventuale arresto anomalo dell'applicazione. Quindi, é molto importante che queste risorse siano gestite con cura. Tuttavia, a volte, lo sviluppatore potrebbe dimenticarsi di farlo. In generale, ci sono tre parti per qualsiasi gestione delle risorse:

- acquisizione della risorsa
- utilizzo della risorsa ed esecuzione delle operazioni
- chiusura/rilascio della risorsa

La libreria di Cats offre diversi modi per assicurarsi che le risorse siano gestite correttamente. In questa sezioni questi metodi vengono approfonditi in dettaglio.

7.1 Pattern Bracket

Il modello Bracket é un modo con cui Cats Effect impone di seguire le buone pratiche. I tre passaggi principali per la gestione delle risorse sono quelli sopra citati: acquisizione, utilizzo e chiusura/rilascio della risorsa. Il metodo bracket viene applicato sull'IO acquisito. Prende due parametri curried in ingresso. Il primo é per l'utilizzo della risorsa mentre la seconda parte riguarda il rilascio. In questo modo, siamo costretti a fornire il blocco di rilascio per qualsiasi risorsa utilizzando il modello Bracket. Questo puó essere paragonato al blocco try-catch-finally di Java. Anche se si verifica un errore durante l'elaborazione del contenuto della risorsa, il metodo bracket si assicura di richiamare il metodo predisposto per la chiusura della risorsa. Se si vuole gestire diversamente il successo, il fallimento e l'annullamento della risorsa, é possibile usare bracketCase invece di bracket. Tuttavia, quando ci sono piú risorse coinvolte e vengono utilizzate come risorse nidificate, il codice diventa poco pulito. In questo caso, Cats fornisce un altro modo migliore per gestire le risorse.

7.2 Resource

Resource in Cats Effect é una struttura dati molto potente per gestire qualsiasi tipo di risorsa. Rimuove tutti i problemi del modello Bracket nel caso in cui sono coinvolte piú risorse. Con Resource possiamo creare una risorsa usando il metodo make. Durante la creazione della risorsa, si fornisce sia l'acquisione che l'implementazione del rilascio della risorsa. In questo modo quando

la risorsa é pronta sia l'apertura e sia il rilascio della risorsa sono giá configurate. Successivamente si puó utilizzare il metodo use sulla risorsa per elaborare i contenuti. É possibile anche combinare facilmente piú fonti poiché Resource é una monade e si puó utilizzare il for-comprehension su di essa. Le risorse vengono acquisite nell'ordine con cui sono state scritte, mentre dopo l'utilizzo vengono rilasciate nell'ordine inverso rispetto all'acquisizione

7.3 Finalizzazione con garantee

Oltre a Bracket e a Resource, Cats Effect fornisce un altro modo per applicare dei finalizzatori. Se si vuole eseguire un blocco di codice al completamento di un IO, indipendentemente dal fatto che sia riuscito con successo, fallito o annullato, possiamo utilizzare guarantee. Il metodo guarantee non distingue tra successo, fallimento e casi di annullamento. Per tutti e tre gli scenari viene applicato lo stesso finalizzatore. Se si vuole trattare i vari casi in modo diverso, é possibile utilizzare warrantyCase invece.

8 Cancellazione di IO/fibre

In questa sezione viene esaminata la cancellazione di IO/fibre in esecuzione.

8.1 Cancellazione

Una delle caratteristiche piú preziose delle fibre é la possibilitá di cancellarle se non sono piú necessarie. In questo modo, é possibile risparmiare risorse ed evitare anche l'utilizzo della CPU delle fibre indesiderate. Come giá discusso in precedenza si pó usare il metodo cancel sull'handle di una fibra per cancellare la fibra in esecuzione. In questo modo, si puó annullare manualmente l'esecuzione di una fibra. Tuttavia, ci sono altri modi in cui é possibile eseguire la cancellazione delle fibre. Alcuni di questi metodi utilizzano il semplice metodo di annullamento "under the hood"(sotto il cofano), che rende ancora piú semplice per gli sviluppatori non preoccuparsi della cancellazione manuale.

8.2 Race di fibre

Invece di gestire manualmente il ciclo di vita di una fibra, possiamo usare il metodo race su due IO e prendere il risultato del primo IO completato. In questo modo, il runtime di Cats gestisce automaticamente il ciclo di vita delle fibre. Gli sviluppatori non devono preoccuparsi della cancellazione dell'IO che ha eseguito la computazione piú lentamente e della sua unione con quello piú veloce. Quando race viene chiamato su due IO, il primo risultato completato viene restituito come risultato di entrambi. Se il primo dei due IO viene completato per primo, il risultato Either viene completato con un valore a sinistra, altrimenti con un valore a destra. In questo caso, il completamento non significa necessariamente un'esecuzione riuscita. Se un IO fallisce rapidamente, l'altro viene annullato immediatamente.

8.3 IO racePair

RacePair é una versione piú generica di race. Invece di annullare la fibra piú lenta, restituisce l'handler di quella fibra insieme all'esito di quella che ha completato l'esecuzione. Lo sviluppatore puó quindi decidere se annullare immediatamente o eseguire altre operazioni prima di annullare o non annullare affatto. Invece di restituire un semplice Either con il risultato, racePair ritorna una tupla con l'outcome della fibra completata e l'handler dell'altra fibra. Questo consente un controllo piú preciso delle dibre durante la loro esecuzione.

8.4 Timeout IO

La libreria di Cats implementa il metodo di timeout su IO per utilizzare la stessa cancellazione della fibra. Possiamo annullare un'esecuzione IO se richiede piú di una durata desiderata utilizzando il metodo timeout. Questo metodo é molto utile nella gestione di operazioni di durata rigorosa. Esiste un'altra variazione di timeot come timeoutTo, che consente di eseguire un IO di fallback nel caso in cui si verifichi il timeout.

8.5 Uncancellable

Finora abbiamo esaminato i diversi modi in cui viene annullato un IO o una fibra. A volte peró bisogna assicurarsi che una parte dell'esecuzione non venga annullata. Ad esempio, dopo un'operazione di database se si sta aggiornando una cache, bisogna essere sicuri che l'operazione di aggiornamento venga completata e che non sia possibile annullarla per non lasciare la cache in uno stato di inconsistenza. É possibile rendere un blocco di codice esente da cancellazione utilizzando il metodo uncancellable. Usando uncancellable, si puó contrassegnare una parte di codice come non cancellabile. Anche se viene sollevata una richiesta di annullamento, l'annullamento di qualsiasi codice all'interno del blocco non annullabile viene negato. Una parte del codice contrassegnato come non cancellabile puó essere reso cancellabile usando il metodo unmask all'interno del blocco di codice uncancellable. Se si vuole rendere l'intero IO non cancellabile basta invocare uncancellable direttamente sull'handler dell'IO. É possibile creare piú regioni di IO non annullabili ed usare il metodo unmask per wrappare gli IO che si vuole rendere cancellabili. In questo modo solo le parti di codice non racchiuse in unmask rimangono non cancellabili. Il blocco unmask aiuta quindi a cancellare una parte di una catena di IO che altrimenti sarebbe non cancellabile.

9 Sincronizzazione e accesso simultaneo

In questa sezione vengono esaminate le primitive di sincronizzazione disponibili in Cats Effect 3

9.1 Sincronizzazione e concorrenza

Quando abbiamo piú fibre/thread che lavorano sugli stessi set di dati, diventa importante avere un accesso simultaneo sicuro e sincronizzazioni agli stati condivisi. Cats Effect offre un'ampia varietá di opzioni per gestire tali casi.

9.2 Ref

Ref fornisce un accesso sumultaneo thread-safe e la modifica degli stati condivisi. Sotto il cofano, utilizza AtomicReference di Java, ma fornisce un modo funzionale per gestirlo. Si possono usare principalmente due modi diversi per creare un Ref che sono Ref[IO].of(value) e IO.ref(value). Viene usato il metodo get per ottenere il valore da una Ref. Allo stesso modo, possiamo usare il metodo set per impostare il valore di una variabile Ref. Il metodo getAndSet restituisce il valore corrente e imposta anche un nuovo valore alla variabile Ref. Sono anche forniti alcuni metodi aggiuntivi come update, updateAndGet e getAndUpdate. Sono simili ai metodi get e set, ma permettono di passare una funzione per modificare lo stato. Esiste un metodo chiamato modify, che funziona in modo simile ad update, ma la modifica consente di restituire un tipo di ritorno diverso dopo che lo stato é stato aggiornato.

9.3 Deferred

Deferred é simile al concetto di Promise. Memorizza un valore che non é ancora disponibile. Durante la creazione un elemento Deferred é vuoto, puó quindi essere completato una volta

ottenendo un valore. Ha due metodi importanti, get e complete. Il metodo get blocca semanticamente finché non é disponibile il valore nella variabile differita. Il metodo complete viene utilizzato per completare una variabile differita e tutte le fibre che hanno invocato il metodo get riceveranno una notifica del valore pronto. Si puó creare una variabile Deferred utilizzando il metodo deferred su un IO.

9.4 Semaphore

Il semaforo é un concetto di sincronizzazione della concorrenza in cui si puó limitare il numero di thread che possono accedere a una risorsa contemporaneamente. Il semaforo contiene un numero positivo di permessi. Eventuali fibre/thread che devono accedere alla risorsa custodita devono acquisire un permesso. Una volta effettuato l'accesso, il permesso viene rilasciato. Se i permessi disponibili sono zero, tutte le fibre in entrata verranno bloccate in modo semantico fino a quando qualcosa non sará disponibile. Semaphore in Cats definisce i metodi acquire, acquireN per la richiesta dei permessi e release, releaseN per rilasciarli.

9.5 Count Down Latch

Count Down Latch é una primitiva di sincronizzazione che attente fino a quando un numero predefinito di fibre é in attesa su di essa. Il blocco del conto alla rovescia é un numero positivo. Quando viene invocato il rilascio il numero del blocco(latch) viene ridotto di uno. Quando il valore del latch diventa zero, il latch viene aperto e l'esecuzione riprende. Ció significa che l'esecuzione continuerá solo se il numero desiderato di fibre viene rilasciato sul blocco.

9.6 Cyclic Barrier

Il Count Down Latch é un metodo di sincronizzazione monouso. Una volta aperto il latch, non é possibile riutilizzare quel blocco. La barriera ciclica é simile quasi simile al meccanismo del conto alla rovesci, ma é riutilizzabile. Nella barriera ciclica si puó utilizzare il metodo await per attendere il numero desiderato di fibre. Quando il numero desiderato di fibre invocano il metodo await su Cyclic Barrier l'esecuzione puó continuare.

10 Operatori comuni

In questa parte vengono discussi gli operatori e metodi comuni disponibili in Cats Effect che non sono stati discussi nei punti precedenti.

10.1 Blocking

Come discusso nelle sezioni precedenti, Cats Effect gestisce le fibre e le alloca automaticamente sui thread. Cats gestisce il threadpool utilizzato per tutte queste operazioni. Tuttavia, a volte é necessario utilizzare operazioni di blocco. Quindi, invece di utilizzare lo stesso threadpool utilizzato dalle fibre di Cats, é meglio utilizzare un threadpool separato. É possibile farlo avvolgendo le operazioni nel blocco IO.blocking. Ció garantirá che Cats utilizzi un threadpool diverso per questa operazione e non influirá sul threadpool principale.

10.2 Interruptible

IO.interruptible é simile a IO.blocking, ma tenta di interrompere l'operazione di blocco se l'operazione viene annullata. In questo caso, l'interruzione viene tentata una sola volta. Esiste un'altra variante come IO.interruptMany, che ritenta l'interruzione finché l'operazione di blocco non viene completata o terminata

10.3 Async

Async é una typeclass che fornisce il supporto per sospendere le operazioni degli effetti collaterali che vengono completate altrove, cioé su un altro threadpool. Questa typeclass consente di eseguire delle operazioni asincrone in sequenza e permette di spostare l'esecuzione in altri contesti di esecuzione. Ad ogni esecuzione viene assegnata una callback che viene richiamata dal runtime di Cats ottenere il risultato quando questo é pronto. Tale calcolo potrebbe non riuscire, quindi la callback é di tipo Either[Throwable, A]. Questa attesa é solo semantica: nessun thread viene bloccato, la fibra corrente viene semplicemente annullata fino al completamento della richiamata.[1]

10.4 IO.attempt

Il metodo attempt converte un IO in Either in base al risultato quando viene eseguito.

10.5 IO.option

Simile ad attempt, é possibile convertire un IO[A] in IO[Option[A]] usando il metodo option. Se l'IO eseguito genera un errore, questo verrá sostituito con None.

11 Monade Eval

Durante la creazione di software, si puó avere bisogno di eseguire alcune azioni immediatamente, mentre é meglio ritardare l'esecuzione di alcune azioni in un secondo momento per ottenere prestazioni migliori. Possiamo controllare le esecuzioni in Scala usando val, lazy val e def. Tuttavia, non é possibile generalizzare facilmente queste diverse valutazioni. É qui che Eval di Cats viene in aiuto. Eval aiuta a controllare le valutazioni sincrone e a gestire tutti i diversi tipi allo stesso modo. In questa sezione si parla della monade Eval di Cats.

11.1 Valutazione

Eval é un tipo di dato per il controllo della valutazione sincrona. La sua implementazione é progettata per fornire sempre la sicurezza dello stack utilizzando una tecnica chiamata trampolino. Ci sono due diversi fattori che giocano nella valutazione: memoizzazione e pigrizia. La valutazione memoizzata valuta un'espressione solo una volta e poi ricorda(memorizza) quel valore. La valutazione lazy si riferisce a quando l'espressione viene valutata. Si parla di valutazione desiderosa se l'espressione viene valutata immediatamente quando definita e di valutazione pigra se l'espressione viene valutata quando viene utilizzata per la prima volta. Ad esempio, in Scala, lazy val é sia pigro che memorizzato, una definizione di metodo def é pigra, ma non memorizzata, poiché il corpo verrá valutato a ogni chiamata. Una normale val é valutata con entusiasmo e memorizza anche il risultato. Eval é in grado di esprimere tutte queste strategie di valutazione e ci consente di concatenare i calcoli utilizzando la sua istanza di Monade.

11.1.1 Eval.now

La prima delle strategie é la valutazione impaziente. Possiamo costruire una valutazione Eval impaziende usando Eval.now. Possiamo eseguire il calcolo utilizzando la strategia di valutazione data in qualsiasi momento utilizzando il metodo value.

11.1.2 Eval.later

Se vogliamo una valutazione pigra, possiamo usare Eval.later. Eval.later é diverso dall'usare lazy val in diversi modi. In primo luogo, consente al runtime di eseguire il Garbage Collection del blocco dopo la valutazione, consentendo di liberare puú memoria in precedenza. In secondo

luogo, quando lazy val viene valutata, al fine di preservare la sicurezza dei thread, il compilatore Scala bloccherá l'intera classe circostante, mentre con Eval.later si bloccherá solo l'Eval.

11.1.3 Eval.always

Se vogliamo una valutazione pigra, ma senza memoizzazione simile a Function0, possiamo usare Eval.always

11.2 Concatenare i calcoli lazy

Una delle applicazioni piú utili di Eval é la sua capacitá di concatenare i calcoli in modo stack-safe. Poiché Eval garantisce la sicurezza dello stack , possiamo concatenare molti calcoli insieme con flatMap senza timore di far saltare in aria lo stack.[2]

12 Tool ausiliari

A supporto del processo agile impiegato nello sviluppo di questo progetto vengono impiegati i seguenti strumenti per migliorare l'efficienza e per consentire una maggiore concentrazione verso le attivitá di sviluppo. Vengono utilizzati i seguenti tool:

- SBT: come strumento di automazione della build
- Git: come repository per la gestione del codice sorgente
- Scalatest: per la scrittura ed esecuzione di test automatizzati

13 Requisiti

13.1 Business

I requisiti di business individuati sono:

- Sperimentare la libreria di Scala Cats concentrando il focus sulle astrazioni fornite per la programmazione funzionale e i supporti per la gestione di I/O e concorrenza
- Realizzare un piccolo framework che fornisca le operazioni necessarie a gestire:
  - gli effetti di un'applicazione
  - i problemi di concorrenza
  - le operazioni di input/output
  - un'interfaccia grafica
- Realizzare un videogioco semplice come caso di studio in cui utilizzare le operazioni fornite dal framework implementato

13.2 Utente

Sono i requisiti che vengono percepiti dall'utente, essi prevedono di:

- Leggere e scrivere su file
- Leggere gli input inseriti attraverso la tastiera
- Fornire un'astrazione che permetta di rappresentare lo stato di un'applicazione grafica con ScalaFX
- Gestire gli effetti di un'applicazione attraverso i costrutti e forniti dalla libreria di Cats

- Fornire le operazioni necessarie per poter gestire le risorse che richiedono un accesso atomico e simultaneo di un'applicazione funzionale
- Gestire gli errori attraverso i costrutti forniti dalla libreria di Cats
- Fornire uno stato condivisibile su cui leggere e scrivere dei valori e su cui poter sincronizzare gli effetti che vi accedono
- Creare un'interfaccia grafica
- Controllare i thread su cui vengono eseguiti gli effetti dell'applicazione

13.3 Funzionali

I requisiti funzionali sono le funzionalitá che il sistema deve supportare. Queste sono:

- Leggere e scrivere su file attraverso gli strumenti forniti dalla libreria di Cats: Bracket e Resource
- Fornire dei listener per poter leggere i caratteri e le stringhe immessi attraverso la tastiera
- Creare un'interfaccia che permetta di realizzare una scena. Lo stato di una scena deve prevedere: un contenuto che viene rappresentato attraverso delle forme geometriche, e il supporto per la creazione di un nuovo stato
- Costruire una GUI con cui poter renderizzare una scena e il suo stato
- Gesire gli effetti che compongono un'applicazione attraverso i costrutti forniti dalla libreria di Cats, in maniera:
  - sequenziale
  - sincrona
  - asincrona
  - ciclica
  - parallela
  - concorrente
  - sincronizzata
  - a scadenza
  - cancellabile
  - incancellabile
  - con finalizzatori
  - bloccante
  - rieseguibile
- Gestire le referenze atomiche con le operazioni di: creazione, lettura e aggiornamento
- Gestire gli errori sfruttando alcune tra le varie possibilitá fornite dalla librearia di Cats:
  - sollevamento di un'eccezione
  - sollevamento di un'eccezione al verificarsi di una certa condizione
  - gestione di un'eccezione con un IO
  - gestione dell'eccezione di un IO con un altro IO
  - gestione dei casi di successo ed errore con dei finalizzatori

- Creare uno stato condivisibile composto da due code: una utilizzata per poter leggere e scrivere le informazioni ed una per poter mettere in attesa e risvegliare i processi che vogliono accedere alla prima coda
- Rendere possibile poter identificare i thread che eseguono gli effetti dell'applicazione

13.4 Non funzionali

I requisiti non funzionali previsti sono i seguenti e prevedono di:

- Realizzare un progetto software che sia estendibile e riutilizzabile
- Ottenere un framework le cui componenti risultino di facile comprensione e semplici da utilizzare

13.5 Implementazione dei requisiti

La logica del sistema viene realizzata in linguaggio Scala e sfrutta alcuni dei supporti per la programmazione funzionale forniti dalla libreria di Cats. Per la parte grafica viene utilizzato ScalaFX. Il software prodotto viene testato con ScalaTest per garantire la manutenzione, la qualitá e la corretta integrazione del codice.

14 Architettura

14.1 Pattern architetturale

Il pattern architetturale scelto per lo sviluppo di questo progetto é il Model-View-Controller(MVC), é stato scelto principlamente perché per applicazioni che si basano sull'interazione con l'utente il pattern MVC é uno dei piú noti e flessibili. Il componente centrale del MVC, il modello, cattura il comportamento dell'applicazione in termini di dominio del problema, indipendentemente dall'interfaccia utente. Il modello gestisce direttamente i dati, la logica e le regole dell'applicazione. La view si occupa di presentare le informazioni mentre la terza parte, il controller, accetta l'input e lo converte in comandi per il modello e/o la vista. Questo schema implica la tradizionale separazione fra la logica applicativa, a carico del controller e del model, e l'interfaccia utente a carico della view.

14.2 Architettura complessiva

Il modello dell'applicazione incapsula tutte le operazioni previste per la gestione degli effetti, per la loro sincronizzazione, l'accesso ai dati simultaneo, la gestione degli errori e la condivisione di uno stato. La view viene impiegata per fornire informazioni relative ai thread che eseguono le operazioni dell'applicazione e per renderizzare il modello fornitogli dal controller. Il controller si occupa della gestione delle operazioni di input/output, di fornire il modello da renderizzare alla vista, di aggiornare tale modello e di esporre all'esterno del framework tutte le funzionalitá offerte. L'idea che sta alla base di questo sistema é quella di un controller che utilizza uno dei loop definiti nel model per ciclare i vari effetti che compongono l'applicazione ed eseguire l'aggiornamento dell'istanza di modello che funge da oggetto osservato. La vista che si comporta come un osservatore nei confronti dell'istanza di modello si aggiorna automaticamente quando questo viene modificato.

15 Design di dettaglio

In questa sezione viene effettutato uno studio sul design di dettaglio del progetto. Vengono quindi evidenziati aspetti interni ai componenti, senza toccarne la reale implementazione.

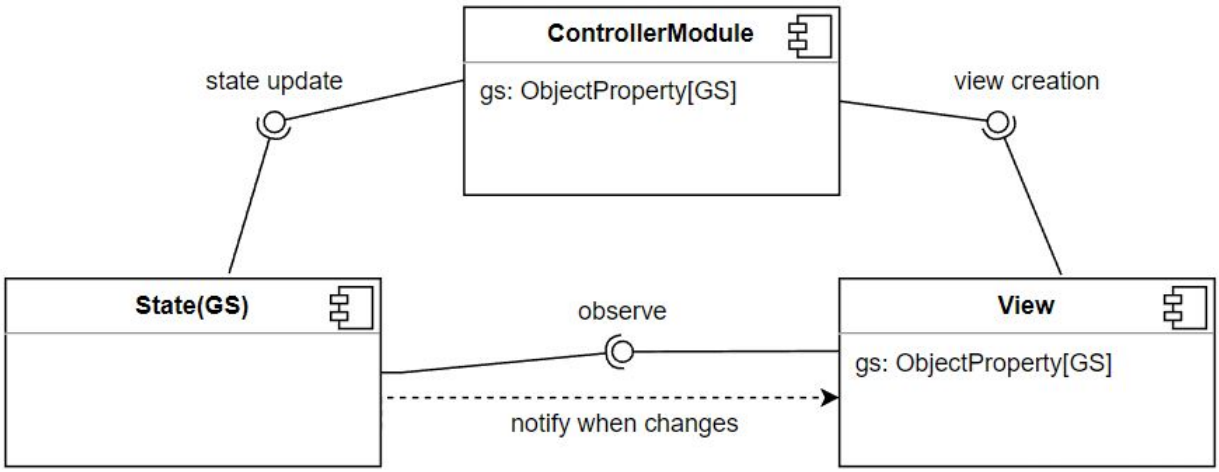

Figura 1: Diagramma delle componenti del framework

15.1 Design del Model

15.1.1 Effetti: sequenziali, paralleli, ciclici e asincroni

Viene creata un'astrazione per modellare un comportamento comune previsto dai vari metodi per la gestione degli effetti che si vuole implementare. La classe astratta prevede l'esecuzione di un numero indefinito di effetti in sequenza e/o applicando una funzione su ogni effetto della sequenza. Partendo da questo comportamento comune vengono poi definiti altri tipi di effetti piú specifici come sequenze di effetti sincrone e sequenze sincrone parallele, effetti asincroni, asincroni paralleli, loop di effetti e loop di effetti paralleli.

Le entitá realizzate nel modello del framework hanno l'obiettivo di gestire gli effetti di un'applicazione e svolgono i seguenti compiti:

- AbsEffects: comportamento generale comune ai tipi di effetto sequenziale, parallelo, asincrono e asincrono parallelo, ciclico e ciclico parallelo
- SequenceEffects: implementa l'esecuzione sequenziale di una serie di effetti applicando eventualmente una funzione ad ognuno di essi. L'esecuzione avviene in maniera sincrona.
- SequenceParEffects: implementa l'esecuzione parallela di una serie di effetti applicando eventualmente una funzione ad ognuno di essi. L'esecuzione avviene in maniera sincrona.
- AsyncEffects: implementa l'esecuzione sequenziale di una serie di effetti applicando eventualmente una funzione ad ognuno di essi. L'esecuzione avviene in maniera asincrona.
- AsyncParEffects: implementa l'esecuzione parallela di una serie di effetti applicando eventualmente una funzione ad ognuno di essi. L'esecuzione avviene in maniera asincrona.
- LoopEffects: implementa un'esecuzione sequenziale ciclica di una serie di effetti applicando eventualmente una funzione ad ognuno di essi. Ad ogni ciclo viene valutata una condizione di terminazione wrappata all'interno di una referenza atomica. L'esecuzione avviene in maniera asincrona.
- LoopParEffects: implementa un'esecuzione parallela ciclica di una serie di effetti applicando eventualmente una funzione ad ognuno di essi. Ad ogni ciclo viene valutata una condizione di terminazione wrappata all'interno di una referenza atomica. L'esecuzione avviene in maniera asincrona.
- AtomicRef: fornisce le operazioni per la gestione di contenuti atomici che necessitano di un accesso simultaneo e sicuro

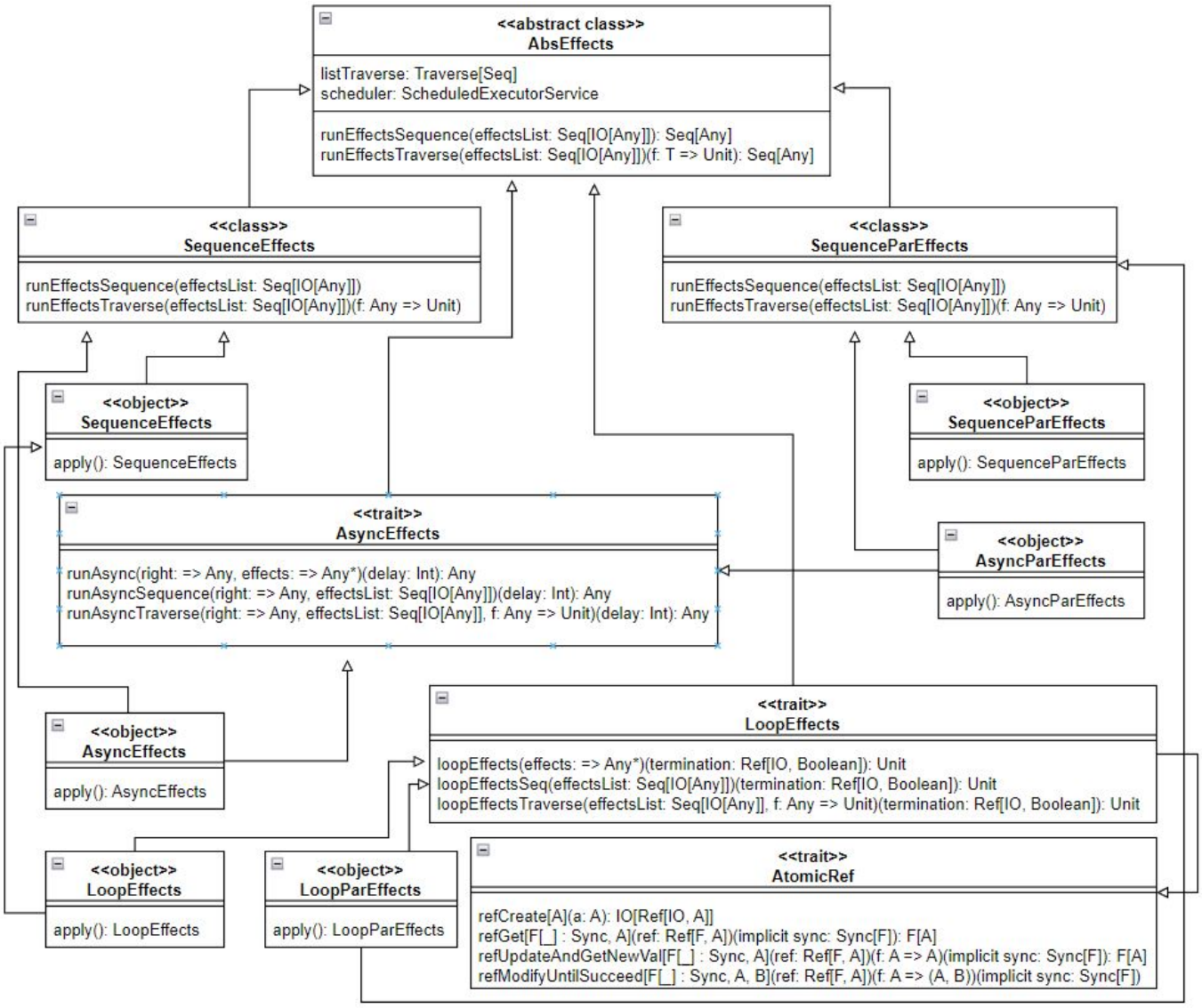

Figura 2: Diagramma delle classi per la gerarchia degli effetti: sequenziali, paralleli, cicli e asincroni

15.1.2 Effetti sincronizzati

Sono gli effetti per i quali si necessita l'utilizzo di strumenti di sincronizzazione. Gli strumenti di sincronizzazione che si é deciso di impiegare sono: Semaphore, Deferred e CountDownLatch. Anche in questo caso viene individuato un comportamento comune da utilizzare nell'implementazione di questo tipo di effetti. L'astrazione implementata riconosce tre tipi di comportamento comuni da utilizzare:

- comportamento di attesa: esegue delle operazioni poi si mette in attesa sullo strumento di sincronizzazione. Quando viene risvegliato dall'attesa esegue delle altre operazioni prima di completare la sua esecuzione
- comportamento di risveglio: esegue delle operazioni, dopodiché risveglia la fibra in attesa sullo strumento di sincronizzazione
- comportamento di esecuzione: esegue i due comportamenti di attesa e risveglio ma non é indispensabile per il loro utilizzo

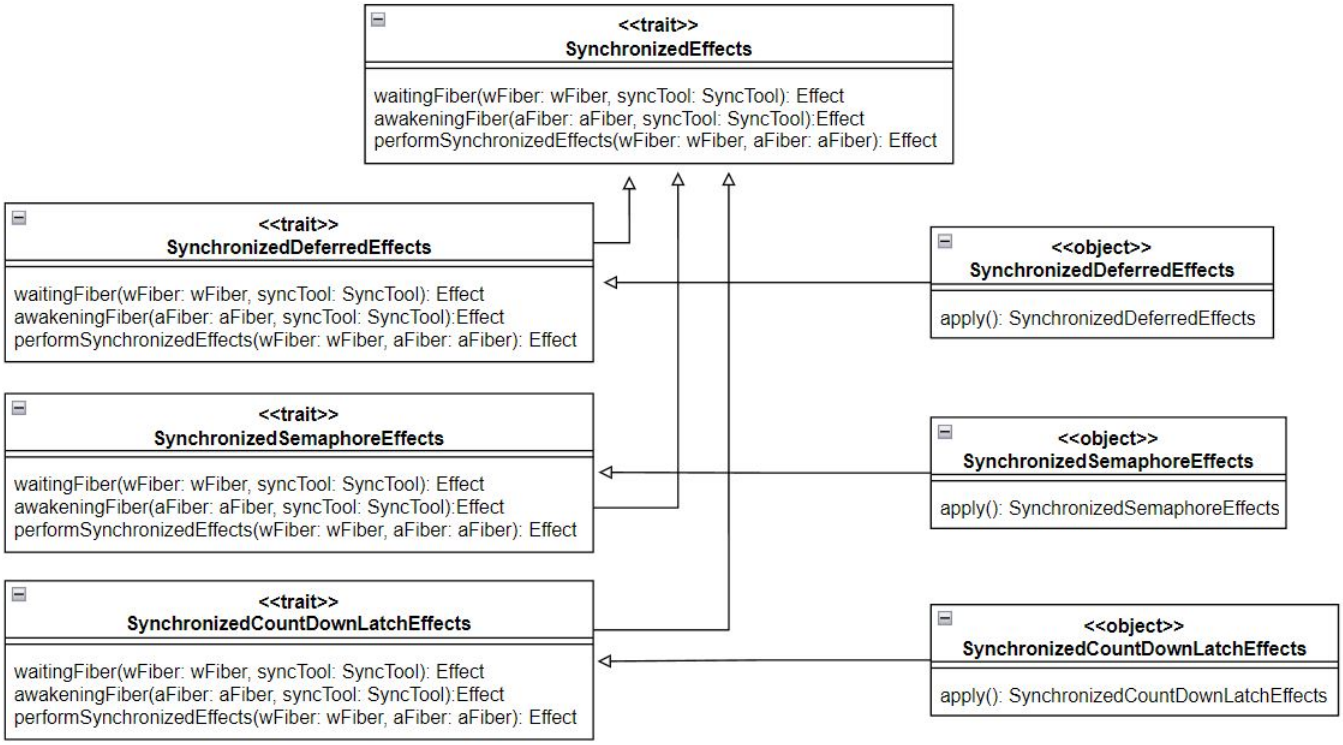

Figura 3: Diagramma delle classi per la gerarchia degli effetti sincronizzati

15.1.3 Effetti non correlati

In questa rappresentazione vengono raccolte classi di effetti diversi tra loro per le quali risulta difficile trovare dei punti in comune. Per questo tipo di effetti viene creata un'astrazione molto generale che ha comunque l'obiettivo di creare una base comune da poter riusare. Le classi di effetti che sono state realizzate per questa sezione sono:

- ExpirationEffect: classe per gli effetti che vengono eseguiti con tempo di espirazione, al termine del quale interrompono la loro esecuzione ed eseguono un effetto di fallback prima di terminare
- FiberWithGuarantee: rappresenta il comportamento di una fibra che viene lanciata in esecuzione in maniera sincrona e per la quale si attende la fine della sua esecuzione. Quando viene creata la fibra verifica una condizione che se non é soddisfatta porta la fibra alla cancellazione. Vengono implementati dei finalizzatori da utilizzare nei vari casi possibili per l'esito dell'esecuzione di questa fibra.
- UncancellableEffects: classe usata per generare effetti di tipo incancellabile e cancellabile. Prevede anche un'esecuzione parallela di tipo lazy.
- BlockingEffects: é la classe utilizzata per generare le fibre che implementano le operazioni di blocco. Come gi'a detto in precedenza per questo tipo di operazioni é meglio utilizzare un threadpool diverso da quello utilizzato da Cats per eseguire le Fibre. Questa classe permette di utilizzare un threadpool diverso per questo tipo di operazioni in modo da non influire sul threadpool principale.
- RetryEffects: questa classe viene usata per gli effetti che per qualche motivo potrebbero non essere eseguiti con successo e per cui si vuole pertanto riprovare l'esecuzione, fino a un certo numero di volte e impostando un tempo di attesa da eseguire prima di ogni tentativo.
- RaceEffects: la classe RaceEffects viene utilizzata per lanciare in esecuzione in maniera concorrente due tipi di effetti. I due effetti vengono eseguiti e viene restituito solo il

risultato dell'effetto che conclude prima la sua esecuzione, ignorando quello che termina per ultimo. Viene implementata anche una variante racePair che prevede di restituire l'handler della fibra che non ha terminato per prima, in modo tale da poter decidere come volerla gestire.

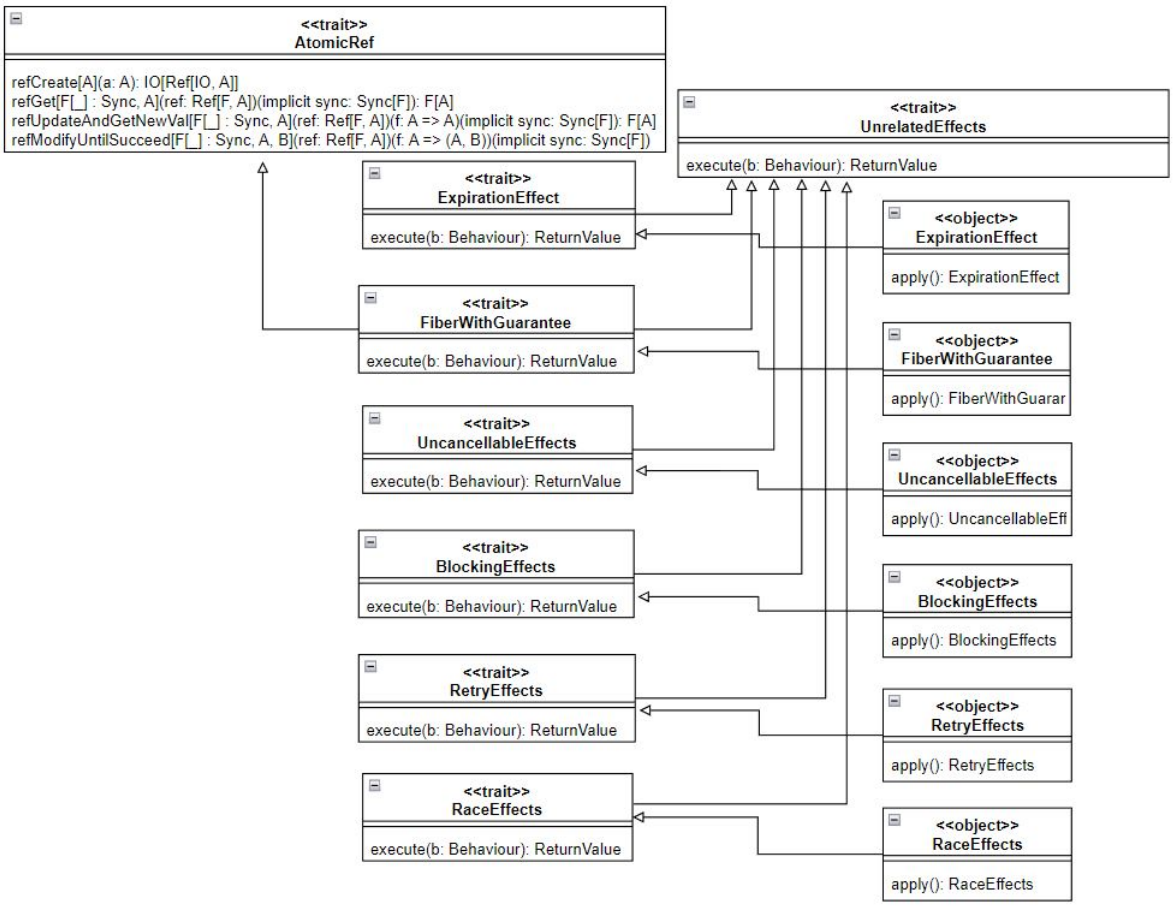

Figura 4: Diagramma delle classi per la gerarchia degli effetti non correlati

Oltre ai vari tipi di effetti e alle referenze atomiche, il model del framework prevede una classe dedicata alla gestione degli errori, la quale viene implementata sfruttando alcuni dei costrutti forniti dalla libreria di Cats. Sempre il model fornisce uno stato condivisibile SharedState composto da due code: la prima da usare per condividere dei valori mentre la seconda coda viene usata per contenere elementi di tipo Deferred che é stata pensata per poter sincronizzare le fibre che vogliono accedere alla prima coda in lettura oppure in scrittura. Il model del framework prevede anche delle operazioni necessarie per eseguire un certo numero di effetti in parallelo e a restituirne il risultato sotto forma ti tupla.

15.2 Design della View

La vista dell'applicazione é progettata per ricevere dal controller un'istanza del modello dell'applicazione da renderizzare che funge da oggetto osservabile. In seguito alle modifiche che avvengono sullo stato di questo modello da parte del controller, la vista che funge da osservatore si aggiorna automaticamente. La view del framework prevede anche una funzionalitá che permetta di stampare a schermo informazioni relative al threadpool sul quale vengono eseguite le fibre dell'applicazione.

15.3 Design del Controller

Il controller del framework si occupa di:

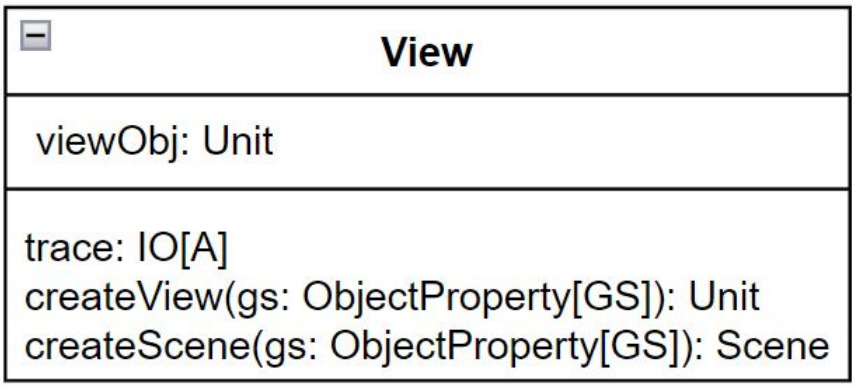

Figura 5: Interfaccia della View

- leggere gli input ricevuti attraverso la tastiera
- leggere e scrivere su file mediante i costrutti di Cats Resource e Bracket
- esporre i metodi e le istanze delle classi presenti nel framework
- fornire un'interfaccia per la creazione e la gestione di uno stato dell'applicazione che sia renderizzabile dalla view
- aggiornare lo stato che viene fornito alla view

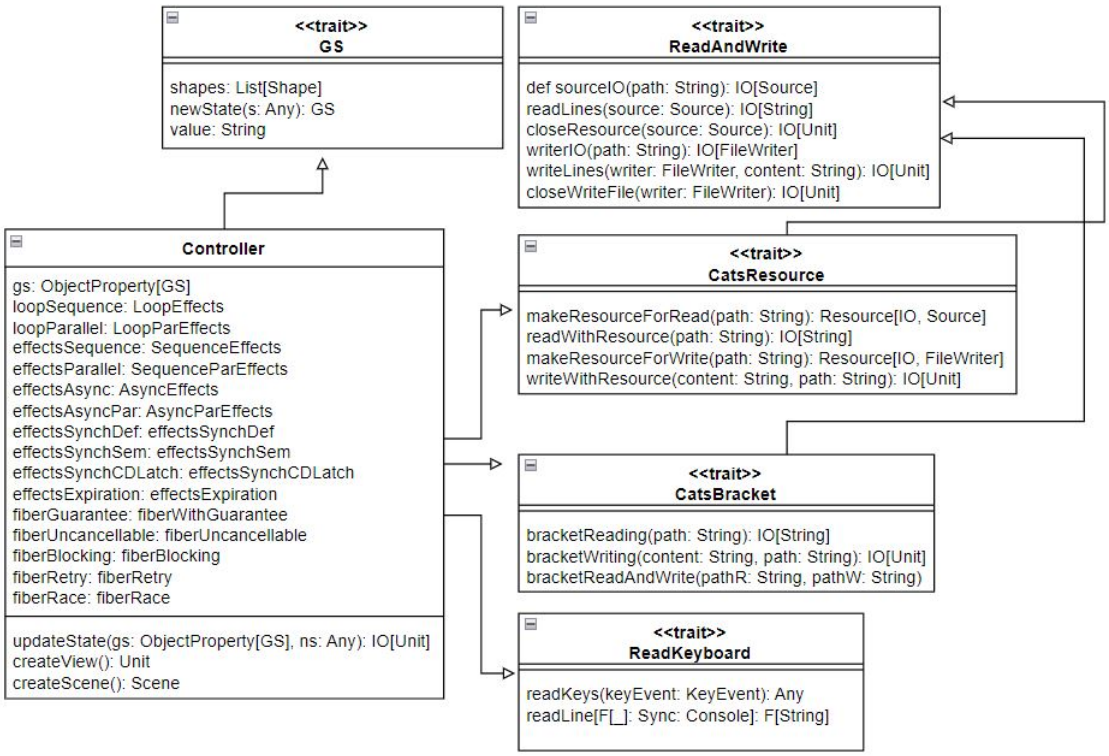

Figura 6: Controller del framework

16 Implementazione

In questa sezione vengono analizzati gli aspetti implementativi del sistema.

16.1 Effetti: sequenziali, paralleli, ciclici, asincroni e referenze atomiche

Nella modellazione di questi effetti é stato individuato un comportamento comune dichiarato attraverso una classe astratta AbsEffects composta da due metodi, i quali prevedono l'esecuzione di una serie di effetti ed eventualmente l'esecuzione con una funzione applicata ad ogni elemento. L'implementazione di questi metodi viene realizzata in modo da eseguire gli effetti in maniera sequenziale e parallela nelle classi SequenceEffects e SequenceParEffects. Per la realizzazione dei comportamenti ciclici e asincroni si utilizza la tecnica dei mixin creando dei tratti AsyncEffects per gli effetti asincroni e LoopEffects per gli effetti ciclici che estendono la classe astratta AbsEffects. Nella definizione dei propri metodi questi tratti non conoscono come sono implementate le operazioni che gli vengono fornite dalla classe astratta AbsEffects, il tipo di implementazione da usare viene deciso all'atto della creazione dell'oggetto relativo al tratto, che puó essere di tipo sequenziale oppure parallelo. Grazie ai mixin possono essere create anche le varianti AsyncParEffects e LoopParEffects semplicemente modificando l'argomento della direttiva extends all'atto della creazione degli oggetti, cosí da non dover replicare il codice scritto. Per la creazione degli oggetti viene usata un'implementazione di tipo factory con metodo apply.

```
1
2  private class SequenceEffects extends AbsEffects:
3
4     type T = Any
5
6     def runEffectsSequence(effectsList: Seq[IO[Any]]) =
7        listTraverse.sequence(effectsList).unsafeRunSync()
8
9     def runEffectsTraverse(effectsList: Seq[IO[Any]])(f: Any => Unit) =
10       listTraverse.traverse(effectsList)(io => io.map(f)).unsafeRunSync()
11
12 object SequenceEffects:
13    def apply(): SequenceEffects =
14       SequenceEffectsImpl()
15    private class SequenceEffectsImpl extends SequenceEffects
16
17
18 private class SequenceParEffects extends AbsEffects:
19
20    type T = Any
21
22    def runEffectsSequence(effectsList: Seq[IO[Any]]) =
23       effectsList.parSequence.unsafeRunSync()
24
25    def runEffectsTraverse(effectsList: Seq[IO[Any]])(f: T => Unit) =
26       effectsList.parTraverse(io => io.map(f)).unsafeRunSync()
27
28 object SequenceParEffects:
29    def apply(): SequenceParEffects =
30       ParallelEffectsImpl()
31    private class ParallelEffectsImpl extends SequenceParEffects
32
33
34 private trait AsyncEffects extends AbsEffects:
35
36    def runAsync(right: => Any, effects: => Any*)(delay: Int): Any =
37       IO.async_[Unit] { cb =>
38          scheduler.schedule(new Runnable {
39             def run = cb(Right(right))
40          }, delay, TimeUnit.MILLISECONDS)
41          effects
```

```
42    }.handleError(error => error.getMessage).unsafeRunSync()
43
44    def runAsyncSequence(right: => Any, effectsList: Seq[IO[Any]])(delay: Int): Any =
45       IO.async_[Unit] { cb =>
46          scheduler.schedule(new Runnable {
47             def run = cb(Right(right))
48          }, delay, TimeUnit.MILLISECONDS)
49       runEffectsSequence(effectsList)
50    }.handleError(error => error.getMessage).unsafeRunSync()
51
52    def runAsyncTraverse(right: => Any, effectsList: Seq[IO[Any]], f: Any =>
          Unit)(delay: Int): Any =
53       IO.async_[Unit] { cb =>
54          scheduler.schedule(new Runnable {
55             def run = cb(Right(right))
56          }, delay, TimeUnit.MILLISECONDS)
57       runEffectsTraverse(effectsList)(f)
58    }.handleError(error => error.getMessage).unsafeRunSync()
59
60 object AsyncEffects:
61    def apply(): AsyncEffects =
62       AsyncEffectsImpl()
63    private class AsyncEffectsImpl extends SequenceEffects with AsyncEffects
64
65
66 object AsyncParEffects:
67    def apply(): AsyncEffects =
68       AsyncParEffectsImpl()
69    private class AsyncParEffectsImpl extends SequenceParEffects with AsyncEffects
```

```
1  private trait LoopEffects extends AbsEffects with AtomicRef:
2
3     def loopEffects(effects: => Any*)(termination: Ref[IO, Boolean]): Unit =
4     refGet(termination).unsafeRunSync() match {
5        case false =>
6        IO {
7           effects.map(_ => loopEffects(effects)(termination))
8        }.handleError(error => error.getMessage).unsafeRunAsync(_ => ())
9        case true => IO.unit
10    }
11
12    def loopEffectsSeq(effectsList: Seq[IO[Any]])(termination: Ref[IO, Boolean]): Unit =
13    refGet(termination).unsafeRunSync() match {
14       case false =>
15       IO {
16          runEffectsSequence(effectsList)
17          .map(_ => loopEffectsSeq(effectsList)(termination))
18       }.handleError(error => error.getMessage).unsafeRunAsync(_ => ())
19       case true => IO.unit
20    }
21
22    def loopEffectsTraverse(effectsList: Seq[IO[Any]], f: Any => Unit)(termination:
          Ref[IO, Boolean]): Unit =
23    refGet(termination).unsafeRunSync() match {
24       case false =>
25       IO {
26          runEffectsTraverse(effectsList)(f)
27          .map(_ => loopEffectsTraverse(effectsList, f)(termination))
28       }.handleError(error => error.getMessage).unsafeRunAsync(_ => ())
```

```
      case true => IO.unit
   }

object LoopEffects:
   def apply(): LoopEffects =
      LoopEffectsImpl()
   private class LoopEffectsImpl extends SequenceEffects with LoopEffects


object LoopParEffects:
   def apply(): LoopEffects =
      LoopParEffectsImpl()
   private class LoopParEffectsImpl extends SequenceParEffects with LoopEffects
```

Gli effetti di tipo ciclico sfruttano una condizione di terminazione per capire quando devono concludere la loro esecuzione. Questa condizione di terminazione é una referenza atomica in quanto ne viene previsto l'accesso simultaneo. Alcuni dei costrutti forniti dalla libreria di Cats che operano sulle referenze di tipo atomico vengono racchiusi in un apposito tratto definito all'interno del model del progetto. Le operazioni implementate riguardano le creazione, la lettura, l'aggiornamento e l'aggiornamento fino al successo.

```
trait AtomicRef:

   def refCreate[A](a: A): IO[Ref[IO, A]] = Ref[IO].of(a)

   def refGet[F[_] : Sync, A](ref: Ref[F, A])(implicit sync: Sync[F]): F[A] = ref.get

   def refUpdateAndGetNewVal[F[_] : Sync, A](ref: Ref[F, A])(f: A => A)(implicit sync:
       Sync[F]): F[A] = ref.updateAndGet(f)

   def refModifyUntilSucceed[F[_] : Sync, A, B](ref: Ref[F, A])(f: A => (A,
       B))(implicit sync: Sync[F]): F[B] = ref.modify(f)
```

16.2 Effetti sincronizzati

Questi tipi di effetti sfruttano degli strumenti per la sincronizzazione che sono Semaphore, Deferred e CountDownLatch. Per la loro implementazione si é scelto di realizzare un tratto composto da tipi generici e metodi astratti che sono stati usati per realizzare un'astrazione comune per questo tipi di effetti. Il tratto astratto SynchronizedEffects é cosí definito:

```
private trait SynchronizedEffects:

   type SyncTool
   type Effect
   type wFiber
   type aFiber

   def waitingFiber(wFiber: wFiber, syncTool: SyncTool): Effect
   def awakeningFiber(aFiber: aFiber, syncTool: SyncTool): Effect
   def performSynchronizedEffects(wFiber: wFiber, aFiber: aFiber): Effect
```

Per ogni tipo di effetto che estende questa interfaccia viene usato un tratto che definisce come sono fatti i tipi precedentemente dichiarati e l'implementazione dei metodi a seconda delle proprie necessiá. La creazione delle istanze di ognuno di questi tratti viene poi realizzata sempre con un implementazione di tipo factory

```
private sealed trait SynchronizedDeferredEffects extends SynchronizedEffects:
```

```
2
3     type Effect = IO[A]
4     type SyncTool = Deferred[IO, A]
5
6     type A = Any
7     type wFiber = (Effect, Effect)
8     type aFiber = (Effect, A)
9
10    def waitingFiber(wFiber: wFiber, syncTool: SyncTool): Effect = for {
11       _ <- wFiber._1.start
12       defValue <- syncTool.get
13       _ <- wFiber._2.start
14    } yield defValue
15
16    def awakeningFiber(aFiber: aFiber, syncTool: SyncTool): Effect = for {
17       ret <- IO(aFiber._1.unsafeRunSync())
18       _ <- syncTool.complete(aFiber._2)
19    } yield ret
20
21    def performSynchronizedEffects(wFiber: wFiber, aFiber: aFiber): Effect = for {
22       deferred <- IO.deferred[A]
23       _ <- waitingFiber((wFiber._1, wFiber._2), deferred).start
24       _ <- awakeningFiber(aFiber, deferred).start
25    } yield ()
26
27 object SynchronizedDeferredEffects:
28    def apply(): SynchronizedDeferredEffects =
29       SynchronizedDeferredEffectsImpl()
30    class SynchronizedDeferredEffectsImpl extends SynchronizedDeferredEffects
```

```
1  private sealed trait SynchronizedSemaphoreEffects extends SynchronizedEffects:
2
3     type Effect = IO[A]
4     type SyncTool = Semaphore[IO]
5     type A = Any
6     type B = Long
7
8     type permitsInit = B
9     type permitsRelease = B
10
11    type wFiber = (Effect, Effect)
12    type aFiber = (Effect, permitsRelease, permitsInit)
13
14    def waitingFiber(wFiber: wFiber, syncTool: SyncTool): Effect = for {
15       _ <- wFiber._1.start
16       _ <- syncTool.acquire
17       _ <- wFiber._2.start
18    } yield ()
19
20    def awakeningFiber(aFiber: aFiber, syncTool: SyncTool): Effect = for {
21       ret <- IO(aFiber._1.unsafeRunSync())
22       _ <- syncTool.releaseN(aFiber._2)
23    } yield ret
24
25    def performSynchronizedEffects(wFiber: wFiber, aFiber: aFiber): Effect = for {
26       sem <- Semaphore[IO](aFiber._3)
27       fib1 <- waitingFiber(wFiber, sem).start
28       fib2 <- awakeningFiber(aFiber, sem).start
29       _ <- fib1.join
```

```
      _ <- fib2.join
   } yield ()

object SynchronizedSemaphoreEffects:
   def apply(): SynchronizedSemaphoreEffects =
      SynchronizedSemaphoreEffectsImpl()
   class SynchronizedSemaphoreEffectsImpl extends SynchronizedSemaphoreEffects
```

```
trait SynchronizedCountDownLatchEffects extends SynchronizedEffects:

   type Effect = IO[A]
   type SyncTool = CountDownLatch[IO]

   type A = Any
   type wFiber = (Effect, Effect)
   type aFiber = (Effect, Approvals)
   type Approvals = Int

   def waitingFiber(wFiber: wFiber, syncTool: SyncTool) = for {
      _ <- wFiber._1
      _ <- syncTool.await
      _ <- wFiber._2
   } yield ()

   def awakeningFiber(aFiber: aFiber, syncTool: SyncTool) = for {
      _ <- aFiber._1
      _ <- syncTool.release
   } yield ()

   def performSynchronizedEffects(wFiber: wFiber, aFiber: aFiber) = for {
      approvals <- CountDownLatch[IO](aFiber._2)
      _ <- waitingFiber(wFiber, approvals).start
      _ <- IO(
         for (x <- 1 to aFiber._2) {
            awakeningFiber(aFiber, approvals).unsafeRunAsync(_ => ())
         }
      )
   } yield ()

object SynchronizedCountDownLatchEffects:
   def apply(): SynchronizedCountDownLatchEffects =
      SynchronizedCountDownLatchEffectsImpl()
   class SynchronizedCountDownLatchEffectsImpl extends
       SynchronizedCountDownLatchEffects
```

16.3 Effetti non correlati

Questo gruppo di tratti raccoglie effetti di vario tipo per i quali é stato realizzato un comportamento comune. Nel tratto UnrelatedEffects vengono dichiarati dei tipi comuni a tutti gli effetti e un metodo per la loro esecuzione:

```
private sealed trait UnrelatedEffects:
   type Effect
   type ReturnValue
   type Behaviour

   def execute(b: Behaviour): ReturnValue
```

Per ogni tipo di effetto viene creato un tratto che estende l'astrazione UnrelatedEffects e ne definisce l'implementazione. Per la creazione delle istanze di ognuno dei tratti realizzati viene fornita un implementazione di tipo factory con metodo apply. Il tratto ExpirationEffect esegue una fibra per una certa durata, allo scadere del tempo interrompe quello che sta facendo e conclude la sua esecuzione con un finalizzatore che prevede l'esecuzione di un effetto di fallback.

```
private sealed trait ExpirationEffect extends UnrelatedEffects:

   type Effect = IO[Unit]
   type Behaviour = (Effect, Int, Effect)
   type ReturnValue = Unit

   def execute(b: Behaviour): ReturnValue =
      val io: Effect = b._1
      val duration: Int = b._2
      val fallback: Effect = b._3

      io.timeoutTo(duration.millis, fallback)

object ExpirationEffect:
   def apply(): ExpirationEffect =
      ExpirationEffectImpl()
   class ExpirationEffectImpl extends ExpirationEffect
```

Il seguente tratto prevede la creazione di una fibra che valuta la condizione contenuta all'interno di una referenza atomica e se questa é soddisfatta esegue le operazioni previste, l'esecuzione continua in maniera sincrona e si attende che la fibra completi la sua attivitá. Quando la condizione invece non é soddisfatta é previsto che la fibra cancelli la sua esecuzione. Attraverso il costrutto di Cats guaranteeCase vengono applicati dei finalizzatori ai vari casi possibili dell'esecuzione della fibra, che sono successo, cancellazione o errore.

```
private sealed trait FiberWithGuarantee extends UnrelatedEffects with AtomicRef:

   type Effect = IO[Unit]
   type Behaviour = (IO[Any], Ref[IO, Boolean], Effect, Effect, Effect)
   type ReturnValue = IO[Outcome[IO, Throwable, Any]]

   def execute(b: Behaviour): ReturnValue =
      val io: IO[Any] = b._1
      val cond: Ref[IO, Boolean] = b._2
      val ioSucc: Effect = b._3
      val ioErr: Effect = b._4
      val ioCanc: Effect = b._5

      for {
         fiber <- io.start
         _ <- IO {
            if (!refGet(cond).unsafeRunSync()) fiber.cancel
         }
         result <- fiber.join
         _ <- io.guaranteeCase {
               case Succeeded(success) =>
                  success.flatMap(msg =>
                     IO("IO successfully completed with value: " + msg) >> ioSucc
                  )
               case Errored(ex) =>
                  IO("Error occurred while processing, " + ex.getMessage) >> ioErr
               case Canceled() => IO("Processing got cancelled in between") >> ioCanc
```

```
28            }
29          } yield result
30
31  object FiberWithGuarantee:
32     def apply(): FiberWithGuarantee =
33        FiberWithGuaranteeImpl()
34     class FiberWithGuaranteeImpl extends FiberWithGuarantee
```

Il tratto di tipo UncancellableEffects esegue una coppia di effetti dove uno é di tipo non cancellabile mentre il secondo é cancellabile, reso tale attraverso il costrutto di Cats unmask da cui viene wrappato all'interno del costrutto uncancellable per gli effetti incancellabili. Questo tratto fornisce anche un secondo metodo che prevede l'esecuzione dei due tipi di effetto in maniera lazy e parallela, sfruttando l'operatore infisso &>

```
1   private sealed trait UncancellableEffects extends UnrelatedEffects:
2
3      type Effect = IO[Any]
4      type Behaviour = (Effect, Effect)
5      type ReturnValue = Unit
6
7      def execute(b: Behaviour): ReturnValue =
8         val cancellableIO: Effect = b._1
9         val uncancellableIO: Effect = b._2
10
11        IO.uncancelable(unmask => unmask(cancellableIO) >> uncancellableIO)
12
13        def execUncancellableLazyParIO(b: Behaviour): ReturnValue =
14        val cancellableIO: Effect = b._1
15        val uncancellableIO: Effect = b._2
16
17        IO.uncancelable(unmask => unmask(cancellableIO) &> uncancellableIO)
18
19  object UncancellableEffects:
20     def apply(): UncancellableEffects =
21        UncancellableEffectsImpl()
22     class UncancellableEffectsImpl extends UncancellableEffects
```

In questo tratto viene realizzata l'implementazione di effetti di tipo bloccante. Gli effetti di tipo bloccante vengono resi tali dal costrutto blocking di Cats e richiedono che la loro esecuzione venga spostata su un threadpool diverso da quello utilizzato dal runtime di Cats Effect. Sono queste le operazioni che vengono svolte per questa implementazione.

```
1   private sealed trait BlockingEffects extends UnrelatedEffects:
2
3      type Effect = IO[Any]
4      type Behaviour = Effect
5      type ReturnValue = Unit
6
7      val customThreadPool = scala.concurrent.ExecutionContext.global
8
9      def execute(b: Behaviour): ReturnValue =
10        val io: Effect = b
11
12        val blockingPoolExec = for {
13           _ <- IO.blocking { io.start }
14        } yield ()
15
16        blockingPoolExec.evalOn(customThreadPool).unsafeRunSync()
17
```

```
object BlockingEffects:
   def apply(): BlockingEffects =
      BlockingEffectsImpl()
   class BlockingEffectsImpl extends BlockingEffects
```

Il tratto RetryEffects si occupa di fornire un meccanismo di retry degli effetti attraverso cui sia possibile riprovare ad eseguire un effetto qual'ora dovesse andare in errore durante la sua esecuzione. Viene impostato un periodo di attesa da eseguire prima di riprovare un'esecuzione dopo un fallimento, e un numero massimo di tentativi per cui si vuole riprovare ad eseguire l'effetto. Il tratto é cosí implementato:

```
private sealed trait RetryEffects extends UnrelatedEffects:
   type Effect = IO[Any]
   type Behaviour = (Effect, Int, FiniteDuration)
   type ReturnValue = Unit

   def retryLoop(io: Effect, times: Int, sleep: FiniteDuration): Effect =
      io.handleErrorWith { case ex =>
         if (times != 0)
            IO.println("Will retry in " + sleep)
            *> IO.sleep(sleep) >> retryLoop(io, times - 1, sleep)
         else
            IO.println("Exhausted all the retry attempts") >> IO.raiseError(ex)
      }

   def execute(b: Behaviour): ReturnValue =
      val io: Effect = b._1
      val times: Int = b._2
      val sleep: FiniteDuration = b._3

      retryLoop(io, times, sleep)

object RetryEffects:
   def apply(): RetryEffects =
      RetryEffectsImpl()
   class RetryEffectsImpl extends RetryEffects
```

Il tratto RaceEffects prevede l'esecuizone di due effetti in maniera concorrente. All'interno del metodo execute fornito dall'interfaccia estesa é utilizzato il costrutto di Cats race il quale prevede di lanciare in esecuzione due effetti per restituire poi il risultato di quello che termina prima la sua esecuzione, dimenticandosi dell'altro. In questo tratto viene definito un ulteriore metodo execRacePair(b: Behaviour): ReturnValue che mediante l'uso del costrutto racePair lancia in esecuzione due effetti e restituisce il risultato di quello che termina prima la sua esecuzione e l'handler della fibra che ha eseguito piú lentamente in maniera tale da poter decidere come gestirla.

```
private sealed trait RaceEffects extends UnrelatedEffects:

   type Effect = IO[Any]
   type Behaviour = (Effect, Effect)
   type ReturnValue = Any

   def execute(b: Behaviour): ReturnValue =
      val io1: Effect = b._1
      val io2: Effect = b._2

      IO.race(io1, io2).map {
         _.match {
```

```
13            case Right(res) => IO.println(s"io2 finished first: `${res}` ") >>
                  IO.pure(res)
14            case Left(res) => IO.println(s"io1 finished first: `${res}` ") >>
                  IO.pure(res)
15         }
16      }
17
18   def execRacePair(b: Behaviour): ReturnValue =
19      val io1: Effect = b._1
20      val io2: Effect = b._2
21
22      IO.racePair(io1, io2).map {
23         _.match {
24            case Right(out, fib) => IO.println(s"io2 finished first: `${out}` ") >>
                  IO.pure(out, fib)
25            case Left(out, fib) => IO.println(s"io1 finished first: `${out}` ") >>
                  IO.pure(out, fib)
26         }
27      }
28
29object RaceEffects:
30   def apply(): RaceEffects =
31      RaceEffectsImpl()
32   class RaceEffectsImpl extends RaceEffects
```

Tutti i tratti per la gestione degli effetti presenti all'interno del sistema sono definiti nel modello e vengono forniti al controller attraverso le istanze di ogni tratto che vengono generate con un'implementazione di tipo factory attraverso un metodo apply. Un oggetto Effects all'interno del modello incapsula tutte le istanze e le rende disponibili al controller del framework:

```
1 object ModelModule:
2    sealed trait Component extends SharedState with ErrorHandling
3       with AtomicRef with ParTupledEffects:
4
5       protected[framework] object Effects:
6          def loopSeq = LoopEffects.apply()
7          def loopPar = LoopParEffects.apply()
8          def effectsSeq = SequenceEffects.apply()
9          def effectsPar = SequenceParEffects.apply()
10         def effectsAsync = AsyncEffects.apply()
11         def effectsAsyncPar = AsyncParEffects.apply()
12         def effectsSynchDef = SynchronizedDeferredEffects.apply()
13         def effectsSynchSem = SynchronizedSemaphoreEffects.apply()
14         def effectsSynchCDLatch = SynchronizedCountDownLatchEffects.apply()
15         def effectsExpiration = ExpirationEffect.apply()
16         def fiberWithGuarantee = FiberWithGuarantee.apply()
17         def fiberUncancellable = UncancellableEffects.apply()
18         def fiberBlocking = BlockingEffects.apply()
19         def fiberRetry = RetryEffects.apply()
20         def fiberRace = RaceEffects.apply()
21
22    trait Interface extends Component
```

16.4 Altre componenti del modello

16.4.1 SharedState

Il tratto SharedState consiste in una case class State usata per generare un oggetto condivisibile su cui poter sincronizzare gli effetti. L'oggetto state é formato da una coda sulla quale leggere e scrivere dei valori ed una seconda coda di tipo Deferred che permette di mettere in attesa e risvegliare le fibre che cercano di accedere all'oggetto condiviso e che necessitano di essere sincronizzate. Il tratto fornisce un metodo empty con il quale inizializzare l'oggetto State, e due metodi per poter leggere e modificare l'oggetto condiviso.

16.4.2 ErrorHandling

Questo tratto é dedicato alla gestione degli errori e sfrutta i costrutti forniti dalla libreria di Cats. I metodi forniti dal tratto permettono di:

- sollevare un'eccezione quando un effetto fallisce
- sollevare un'eccezione quando si verifica una certa condizione
- gestire l'eccezione di un effetto di tipo IO
- gestire l'eccezione di un effetto di tipo IO con un altro effetto di tipo IO
- decidere come gestire i casi di successo ed errore di un effetto IO con dei finalizzatori

16.4.3 ParTupledEffects

Il tratto ParTupledEffects consente di eseguire un certo numero di effetti IO in parallelo per poi restituire il loro risultato sotto forma di tuple. All'interno di questa classe viene utilizzato un meccanismo di conversione implicita per consentire di utilizzare i costrutti forniti da Cats anche con delle sequenze di effetti le quali non sono previste come input dai costrutti parTupled della libreria:

```
1
2  trait ParTupledEffects:
3
4     def parTupled2[A0, A1](io1: IO[A0], io2: IO[A1]): IO[(A0, A1)] = (io1,
          io2).parTupled
5
6     implicit def toTuple2[A](x: Seq[IO[A]]): (IO[A], IO[A]) =
7        x match {
8           case Seq(a, b) => (a, b)
9        }
10
11    def parTupled2Seq[A](s: Seq[IO[A]]): IO[(A, A)] =
12       val seq: (IO[A], IO[A]) = s
13       seq.parTupled
14
15    def parTupled3[A0, A1, A2](io1: IO[A0], io2: IO[A1], io3: IO[A2]): IO[(A0, A1, A2)]
          = (io1, io2, io3).parTupled
16
17    implicit def toTuple3[A](x: Seq[IO[A]]): (IO[A], IO[A], IO[A]) =
18       x match {
19          case Seq(a, b, c) => (a, b, c)
20       }
21
22    def parTupled3Seq[A](s: Seq[IO[A]]): IO[(A, A, A)] =
23       val seq: (IO[A], IO[A], IO[A]) = s
```

```
24      seq.parTupled
25
26    def parTupled4[A0, A1, A2, A3](io1: IO[A0], io2: IO[A1], io3: IO[A2], io4: IO[A3]):
          IO[(A0, A1, A2, A3)] = (io1, io2, io3, io4).parTupled
27
28    implicit def toTuple4[A](x: Seq[IO[A]]): (IO[A], IO[A], IO[A], IO[A]) =
29      x match {
30        case Seq(a, b, c, d) => (a, b, c, d)
31      }
32
33    def parTupled4Seq[A](s: Seq[IO[A]]): IO[(A, A, A, A)] =
34      val seq: (IO[A], IO[A], IO[A], IO[A]) = s
35      seq.parTupled
```

16.5 Gestione delle risorse con Bracket e Resource

I passaggi coinvolti nella gestione delle risorse sono:

- acquisizione della risorsa: che in questo caso risulta essere un file
- utilizzo della risorsa: lettura o scrittura del file
- chiusura della risorsa: quindi la chiusura del file

Vengono creati dei metodi per questi tre diversi passaggi nel tratto ReadAndWrite all'interno del controller del framework.

Cats Effect fornisce diversi tipi di approcci per la gestione delle risorse, nel framework realizzato vengono usati i seguenti approcci:

16.5.1 Modello delle parentesi: Bracket

I metodi creati vengono combinati con il modello delle parentesi fornito da Cats. Viene invocato il metodo Bracket sull'acquisizione delle risorse e vengono invocati i metodi per leggere o scrivere e chiudere la risorsa come parametri di essa. Nel fare questo Cats Effect obbliga lo sviluppatore a fornire l'implementazione per rilasciare le risorse quando ne viene effettuato l'utilizzo. Fare questo serve ad evitare problemi di memory leak durante l'utilizzo delle risorse. Il modello delle parentesi risulta essere efficace quando é presente un solo livello di risorse da manipolare, ma se bisogna gestire diversi livelli nidificati di risorse, questo puó diventare piú complesso e meno leggibile.

16.5.2 Gestione delle risorse con Resource

Questo modello segue lo stesso approccio in tre fasi di acquisizione, uso e rilascio visto nel punto precedente. Tuttavia, questo modello separa chiaramente la parte di acquisizione e rilascio dalla parte di utilizzo. Le parti di acquisizione e rilascio sono legate insieme utilizzando il metodo make che restituisce un Resource, e si utilizza quindi il metodo use sulla risorsa. In questo modo la gestione e l'utilizzo della risorsa sono separati in modo netto. Ogni volta che il metodo use viene invocato il runtime di Cats Effects acquisisce automaticamente le risorse e le rilascia al completamento. L'utilizzo di Resource risulta anche essere piú indicato nel caso di risorse nidificate, per questioni di leggibilitá.

```
1
2   private[controller] trait ReadAndWrite:
3
4     def sourceIO(path: String): IO[Source] = IO(Source.fromFile(path))
```

```
5     def readLines(source: Source): IO[String] =
          IO(source.getLines().mkString(``\textbackslash{}n'')) <* IO.println(``file
          reading completed'')
6     def closeResource(source: Source): IO[Unit] = IO(source.close())
7
8     def writerIO(path: String): IO[FileWriter] =
9        IO.println(``Acquiring file to write'') >> IO(new FileWriter(path))
10
11    def writeLines(writer: FileWriter, content: String): IO[Unit] =
12       IO.println(``Writing the contents to file'') >> IO(writer.write(content)) <*
             IO.println(``writing completed'')
13
14    def closeWriteFile(writer: FileWriter): IO[Unit] =
15       IO.println(``Closing the file writer'') >> IO(writer.close())
16
17
18 private[controller] trait CatsBracket extends ReadAndWrite:
19
20    def bracketReading(path: String): IO[String] =
21       sourceIO(path).bracket(src => readLines(src))(src => closeResource(src))
22
23    def bracketWriting(content: String, path: String): IO[Unit] =
24       writerIO(path).bracket(fw => writeLines(fw, content))(fw => closeWriteFile(fw))
25
26    def bracketReadAndWrite(pathR: String, pathW: String) = sourceIO(pathR).bracket {
          src =>
27       val contentsIO = readLines(src)
28       writerIO(pathW).bracket(fw =>
29          contentsIO.flatMap(contents => writeLines(fw, contents))
30       )(fw => closeWriteFile(fw))
31    } { src =>
32    closeResource(src)
33    }
34
35
36 private[controller] trait CatsResource extends ReadAndWrite:
37
38    def makeResourceForRead(path: String): Resource[IO, Source] =
39       Resource.make(sourceIO(path))(src => closeResource(src))
40
41    def readWithResource(path: String): IO[String] =
42       makeResourceForRead(path).use(src => readLines(src))
43
44    def makeResourceForWrite(path: String): Resource[IO, FileWriter] =
45       Resource.make(writerIO(path))(fw => closeWriteFile(fw))
46
47    def writeWithResource(content: String, path: String): IO[Unit] = for {
48       _ <- IO.println("entrato")
49       _ <- makeResourceForWrite(path).use(fw => writeLines(fw, content))
50    } yield ()
```

16.6 Lettura degli input da tastiera

Per la lettura degli input da tastiera il framework utilizza un listener ad eventi per leggere i singoli tasti mentre adopera il costrutto di Cats Console[F].readLine per poter leggere le stringhe che vengono inserite.

16.7 Gestione del GameState(GS) e implementazione del controller

Lo stato dell'applicazione che deve essere renderizzato viene dichiarato all'interno di un tratto GS presente nel controller il quale prevede al suo interno due metodi per creare un nuovo stato e per fornire alla vista i contenuti da renderizzare. Il controller dichiara tra le sue componenti un valore immutabile GS che viene poi fornito ai metodi della view come oggetto osservabile attraverso le proprietá fornite dalla classe ObjectProperty di ScalaFX. Oltre alle istanze relative alla gestione degli effetti che gli vengono fornite dal modello, il controller definisce al suo interno un metodo per l'aggiornamento dello stato fornito alla view, e richiama i metodi della vista che ne prevedono la sua gestione:

```
object ControllerModule:

   sealed trait Component extends ReadKeyboard
      with CatsResource with CatsBracket

   trait Interface extends Component with ModelModule.Interface with
       ViewModule.Interface:

      val gs: ObjectProperty[GS]
      def updateState(gs: ObjectProperty[GS], ns: Any): IO[Unit] =
          IO(gs.update(gs.value.newState(ns)))
      def createView(): Unit = createView(gs)
      def createScene(): Scene = createScene(gs)

      val loopSequence = Effects.loopSeq
      val loopParallel = Effects.loopPar
      val effectsSequence = Effects.effectsSeq
      val effectsParallel = Effects.effectsPar
      val effectsAsync = Effects.effectsAsync
      val effectsAsyncPar = Effects.effectsAsyncPar
      val effectsSynchDef = Effects.effectsSynchDef
      val effectsSynchSem = Effects.effectsSynchSem
      val effectsSynchCDLatch = Effects.effectsSynchCDLatch
      val effectsExpiration = Effects.effectsExpiration
      val fiberGuarantee = Effects.fiberWithGuarantee
      val fiberUncancellaBLe = Effects.fiberUncancellable
      val fiberBlocking = Effects.fiberBlocking
      val fiberRetry = Effects.fiberRetry
      val fiberRace = Effects.fiberRace
```

16.8 Implementazione della vista

La vista fornisce un'estensione trace al tipo IO che viene utilizzata per stampare il nome dei thread sui quali vengono eseguiti gli effetti di tipo IO. La vista all'interno del tratto View definisce i metodi che servono per la sua creazione e per la creazione della scena della quale vengono renderizzati i contenuti. Il metodo createScene(gs: ObjectProperty[GS]): Scene riceve in input dal controller un'istanza dell'oggetto osservabile di tipo GS sul quale attraverso il metodo onChange della classe ObjectProperty si mette in ascolto per aggiornare automaticamente il proprio contenuto ogni volta che l'oggetto viene modificato.

```

object ViewModule:

   sealed trait Component extends View:

      val viewObj: Unit = View.apply()
```

```
7
8        extension[A] (io: IO[A])
9           def trace: IO[A] = for{
10             res <- io
11             _ = println(s"[${Thread.currentThread.getName}]" + res)
12          } yield res
13
14    trait Interface extends Component
15
16
17 trait View:
18
19    def createView(gs: ObjectProperty[GS]): Unit = new PrimaryStage {
20       title = "ScalaFX App"
21       scene = createScene(gs)
22    }
23
24    def createScene(gs: ObjectProperty[GS]): Scene =
25    new Scene {
26       root = new StackPane {
27          content = gs.value.shapes
28
29          gs.onChange(Platform.runLater {
30             content = gs.value.shapes
31          })
32       }
33    }
34
35 object View:
36    def apply(): Unit =
37       ViewImpl()
38    private class ViewImpl extends View
```

16.9 Il caso di studio: Snake

Il seguente caso di studio viene creato per mettere in pratica le operazioni messe a disposizione dal framework realizzato, il quale viene esteso per essere utilizzato.

16.9.1 Requisiti utente

- Rappresentare in una GUI con delle semplici forme geometriche:
  - un serpente
  - il cibo
  - il cibo cattivo, che deve essere evitato
- Muovere il serpente con i tasti: w, a, s, d
- Mangiare il cibo
- Generare del cibo quando il serpente ha mangiato
- Fare crescere il serpente quando mangia il cibo
- Fare collidere il serpente con i bordi della GUI, col cibo, col cibo cattivo e con il serpente stesso
- Ripristinare le dimensioni del serpente allo stato di partenza quando collide con i limiti della scena, con il cibo cattivo o con se stesso

- Mettere in pausa la partita premendo il tasto p
- Riprendere la partita in pausa quando viene premuto il tasto p
- Fare scomparire e riapparire il cibo cattivo con una certa frequenza
- Stampare nella GUI il nome del gioco
- Salvataggio e lettura del punteggio massimo raggiunto
- Mostrare nella GUI il punteggio massimo che é stato raggiunto

16.9.2 Requisiti funzionali

- Rappresentare i due tipi di cibo come delle forme geometriche rettangolari
- Rappresentare il serpente con una lista di rettangoli
- La direzione seguita dal serpente viene decisa come segue:
  - movimento verso l'alto: tasto w
  - movimento verso sinistra: tasto a
  - movimento verso il basso: tasto s
  - movimento verso destra: tasto d
- Le operazioni che compongono il modello del gioco vengono eseguite ad ogni cambio di frame, i frame vengono aggiornati ogni 70 millisecondi. Ad ogni cambio di frame si aggiorna lo stato del gioco come segue:
  - si aggiorna lo stato della scena
  - la nuova direzione del serpente in coordinate x, y viene calcolata sulla base dell'ultimo tasto premuto
  - la testa del serpente si sposta in base alla direzione calcolata
  - l'ultimo elemento del serpente(la coda) viene rimosso
  - quando il serpente collide con il cibo, il cibo diventa la nuova testa del serpente
- Quando il serpente ha mangiato viene generato un nuovo elemento di tipo cibo
- Gli elementi di tipo cibo vengono posizionati con delle coordinate x, y generate casualmente
- Gli elementi di tipo cibo non possono essere generati nei punti giá occupati dagli elementi del serpente o al di fuori della scena
- Calcolare la collisione del serpente con i bordi della scena
- Calcolare la collisione del serpente con i due tipi di cibo e con se stesso sulla base dell'uguaglianza delle coordinate x e y degli elementi coinvolti
- Nel campo da gioco sono presenti sei elementi di tipo cibo cattivo che vengono fatti apparire uno alla volta in sequenza e poi fatti scomparire tutti assieme, ogni 200 frame
- Il serpente ha una dimensione iniziale di tre rettangoli
- Quando il serpente collide con i bordi dello schermo, con la sua coda e con il cibo cattivo, viene riportato alla dimensione iniziale
- Quando viene premuto il tasto p il gioco viene messo in uno stato di pausa

- Quando il gioco si trova in pausa e viene premuto il tasto p la partita riprende
- Memorizzare il punteggio piú alto che viene raggiunto, calcolato sulla base della dimensione raggiunta dal serpente
- Nella scena viene stampato il punteggio piú alto raggiunto
- Nella scena viene stampato il nome del gioco
- Quando una nuova partita viene lanciata si legge da un file il punteggio massimo raggiunto
- Quando il serpente viene ripristinato se il punteggio raggiunto supera il massimo mai raggiunto allora questo viene memorizzato scrivendolo su un file

16.9.3 Controller

Il controller del progetto Snake svolte le seguenti operazioni

- lettura degli input: questo viene fatto ridefinendo il metodo readKeys fornito dal framework attraverso una sovrascrittura. I tasti che vengono letti sono quelli per direzionare il serpente: w, a, s, d
- gestione della pausa del gioco: viene utilizzato l'oggetto SharedState del framework e viene eseguito l'override del suo metodo modifyState per la sua modifica. Con il metodo readLine importato dal framework si legge l'inserimento del carattere previsto per la pausa p dalla console. Con modifyState viene inserito e rimosso un valore nella coda di valori dell'oggetto SharedState ogni volta che il carattere p viene inserito nella console come ultimo carattere. In base alla presenza o meno di valori nella coda il gioco viene messo in pausa oppure ripreso
- crea un'istanza dello stato del gioco che si compone di:
  - un serpente iniziale
  - un blocco di cibo
  - una lista di elementi di cibo da evitare wrappata in una referenza atomica
  - il risultato di un effetto generato attraverso il metodo readWithResource importato dal framework che rappresenta il punteggio massimo ottenuto nel gioco che viene letto da un file esterno
  - una proprietá osservabile che descrive il numero di frame
- per ogni frame verifica che il gioco non sia in pausa, e se non lo é utilizza il metodo importato dal framework updateState per aggiornare lo stato del gioco
- definisce una proprietá direction che viene usata dirigere il serpende

---

```
1
2  trait Controller extends Model:
3
4    override def readKeys(keyEvent: KeyEvent): Unit =
5      keyEvent.getText match {
6        case "w" | "W" => if(direction.value != 2) direction.value = 1
7        case "s" | "S" => if(direction.value != 1) direction.value = 2
8        case "a" | "A" => if(direction.value != 4) direction.value = 3
9        case "d" | "D" => if(direction.value != 3) direction.value = 4
10       case _ => IO.unit
11     }
12
```

```
13  def handlePause(sState: Ref[IO, State[IO, Int]]): IO[Unit] =
14    for {
15      _ <- modifyState(sState)
16        .handleErrorWith { t =>
17          Console[IO].errorln(s"Error caught: ${t.getMessage}").as(ExitCode.Error)
18        }
19    } yield ()
20
21  val direction = IntegerProperty(4) // 4 = right
22
23  val frame = IntegerProperty(0)
24
25  val gs: ObjectProperty[GS] = ObjectProperty(GameState(initialSnake,
        randomFood(List(EvilFood(-1, -1)), initialSnake),
        refCreate(initialEvilFood).unsafeRunSync(),
        readWithResource(cfg.scoreFilePath).unsafeRunSync(), frame))
26
27  def frameUpdate(stateR: Ref[IO, State[IO, Int]]): Unit =
28
29    frame.onChange {
30
31      val program: IO[Unit] =
32        for {
33          ret <- checkState(stateR)
34          _ <- if (ret) updateState(gs, direction.value)
35            else IO.unit
36        } yield ()
37      program.unsafeRunSync()
38    }
39
40  private def checkState[F[_] : Sync : Console](stateR: Ref[F, State[F, Int]]):
        F[Boolean] =
41    stateR.modify {
42      case State(queue, deferredQ)
43        if queue.nonEmpty => State(queue, deferredQ) -> false
44
45      case State(queue, deferredQ) =>
46        State(queue, deferredQ) -> true
47    }
48
49  def readKey[F[_] : Sync : Console]: F[Char] =
50    for {
51      n <- readLine
52    } yield if(n.nonEmpty)n.last else 'x'
53
54  override def modifyState[F[_] : Sync : Console](sState: Ref[F, State[F, Int]]):
        F[Boolean] =
55    def edit: F[Unit] =
56      sState.modify {
57        case State(queue, deferredQ) if queue.nonEmpty =>
58          val (elem, nQueue) = queue.dequeue
59          State(nQueue, deferredQ) -> true
60        case State(queue, deferredQ) =>
61          State(queue.enqueue(1), deferredQ) -> true
62      }
63
64    for {
65      ch <- readKey
66      _ <- ch match {
```

```
67          case 'p' | 'P' => edit
68          case _ => Sync[F].unit
69        }
70        _ <- modifyState(sState)
71      } yield true
```

16.9.4 Model

Il tratto Model estende il framework al fine di poterlo utilizzare. Vengono create all'interno del tratto le seguenti classi case:

- Snake: per rappresentare il serpente
- Food: per rappresentare il cibo da mangiare
- EvilFood: per il cibo da schivare
- GameState: per rappresentare lo stato del gioco. Questa classe case estende il tratto GS fornito dal framework e ne implementa i metodi. Questo stato viene renderizzato dalla vista dell'applicazione.

All'interno della classe case GameState viene definito il metodo newState dichiarato all'interno del framework e chiamato attraverso il metodo updateState del controller. Il metodo newState(dir: Any): GameState restituisce ad ogni sua invocazione un nuovo stato del gioco e svolge le seguenti operazioni:

- sulla base della direzione inserita calcola la nuova posizione della testa del serpente
- aggiorna il serpente:
  - se il serpente collide con se stesso o con il cibo da evitare allora sempre il serpente viene ripristinato alla situazione di partenza
  - quando il serpente torna allo stato iniziale verifica se il punteggio massimo raggiunto é stato superato, se é cosí scrive su file il nuovo punteggio
  - se il serpente mangia aggiunge il nuovo elemento in testa al serpente
  - se il serpente non mangia e non collide rimuove l'ultimo elemento dalla sua coda
- esegue tre effetti in parallelo con il metodo parTupled3 importato dal frawork attraverso il quale:
  - invoca le operazioni sopra descritte per l'aggiornamento del serpente
  - crea il nuovo cibo se il serpente ha mangiato
  - con il metodo effectsSynchCDLatch importato dal framework invoca due fibre sincronizzate che attraverso una serie di operazioni fanno comparire in sequenza diversi elementi di cibo da evitare e poi li fanno scomparire. Questo comportamento avviene ad ogni 200 frame trascorsi
- restituisce un nuovo stato immutabile composto da: nuovo serpente, cibo, lista di cibo cattivo, punteggio massimo e numero di frame

Altri metodi appartenenti al tratto Model riguardano:

- la creazione del contenuto da renderizzare per la View attraverso il metodo shapes: List[Rectangle] dichiarato nel framework e definito in questo tratto. Gli elementi da raffigurare in questo caso sono il serpente e i vari tipi di cibo.

- il metodo efoodCollision(efoodRef: Ref[IO, List[EvilFood]], newx: Double, newy: Double): Boolean per il controllo della collisione con il cibo da evitare
- un metodo square(xr: Double, yr: Double, color: Color) per la creazione delle forme geometriche rappresentate nel contenuto della scena
- un metodo per il calcolo della nuova direzione della testa
- metodi per generare il cibo e il serpente allo stato iniziale
- con writeNewHighScore viene richiamato il metodo writeWithResource del framework per scrivere su un file esterno

---

```
1
2  trait Model extends ControllerModule.Interface:
3
4    case class Snake(list: List[(Double, Double)])
5
6    case class Food(value: (Double, Double))
7    case class EvilFood(value: (Double, Double))
8    case class GameState(snake: Snake, food: Food, efoodRef: Ref[IO, List[EvilFood]],
         value: String, frame: IntegerProperty) extends GS:
9
10     def newState(dir: Any): GameState =
11       def calcNewHead(snakeRef: Ref[IO, List[(Double, Double)]]): IO[(Double, Double)]
             =
12         for {
13           snake <- refGet(snakeRef)
14           head <- IO(snake.head)
15           (x, y) = head
16           nHead <- compDirection(dir, x, y)
17           (newx, newy) = nHead
18         } yield (newx, newy)
19
20       def newSnake(snakeRef: Ref[IO, List[(Double, Double)]], newx: Double, newy:
             Double): IO[List[(Double, Double)]] =
21         for {
22           newSnake <- if (newx < 0 || newx >= 600 || newy < 0 || newy >= 600
23             || refGet(snakeRef).unsafeRunSync().tail.contains(newx, newy) ||
                   efoodCollision(efoodRef, newx, newy))
24             for {
25               initSnake <- IO(initialSnake) <& writeNewHighScore(value.toInt,
                     refGet(snakeRef).unsafeRunSync().size - 3)
26             } yield (initSnake.list)
27           else if (food.value == (newx, newy))
28             for {
29               newSnake <- refUpdateAndGetNewVal(snakeRef)(snake => food.value :: snake)
30             } yield (newSnake)
31           else
32             refUpdateAndGetNewVal(snakeRef)(snake => (newx, newy) :: snake.init)
33         } yield newSnake
34
35       def snakeLogic: IO[(Snake, Food)] =
36         for {
37           snakeRef <- refCreate(snake.list)
38           newHead <- calcNewHead(snakeRef)
39           (newx, newy) = newHead
40           ret <- parTupled3(newSnake(snakeRef, newx, newy),
41             if (food.value == (newx, newy))
```

```
42              IO(randomFood(refGet(efoodRef).unsafeRunSync(), snake)) //crea cibo
43            else IO(food), IO(if (frame.value % 200 == 0)
                IO(effectsSynchCDLatch.performSynchronizedEffects((
44              IO.unit, efoodRef.set(initialEvilFood)), (IO(
45              refGet(efoodRef)
46                .flatMap(efoodList => refUpdateAndGetNewVal(efoodRef)(_ =>
                    efoodList.updated(0, randomEfood(efoodList, food, snake))))
47                .flatMap(efoodList => IO.sleep(Random.between(500, 1500).millis) *>
                    refUpdateAndGetNewVal(
48                 efoodRef)(_ => efoodList.updated(1, randomEfood(efoodList, food,
                      snake))))
49                .flatMap(efoodList => IO.sleep(Random.between(500, 1500).millis) *>
                    refUpdateAndGetNewVal(
50                 efoodRef)(_ => efoodList.updated(2, randomEfood(efoodList, food,
                      snake))))
51                .flatMap(efoodList => IO.sleep(Random.between(500, 1500).millis) *>
                    refUpdateAndGetNewVal(
52                 efoodRef)(_ => efoodList.updated(3, randomEfood(efoodList, food,
                      snake))))
53                .flatMap(efoodList => IO.sleep(Random.between(500, 1500).millis) *>
                    refUpdateAndGetNewVal(
54                 efoodRef)(_ => efoodList.updated(4, randomEfood(efoodList, food,
                      snake))))
55                .flatMap(efoodList => IO.sleep(Random.between(500, 1500).millis) *>
                    refUpdateAndGetNewVal(
56                 efoodRef)(_ => efoodList.updated(5, randomEfood(efoodList, food,
                      snake)))
57                 <* IO.sleep(5.seconds))
58            ).unsafeRunSync(), 1)).unsafeRunSync()).unsafeRunSync()
59          )
60        )
61        (nSnake, nFood, ()) = ret
62        _ <- handledErrorIO(raisedConditionIO(nSnake.isEmpty, new Exception()),
63          _ => refModifyUntilSucceed(snakeRef)(current => (current, initialSnake ::
                current)))
64      } yield (Snake(nSnake), nFood)
65
66    val (nSnake, nFood) = snakeLogic.unsafeRunSync()
67
68    GameState(nSnake, nFood, efoodRef, value, frame)
69
70  def shapes: List[Rectangle] = square(food.value._1, food.value._2, Red)
71    :: snake.list.map {
72    case (x, y) => square(x, y, Green)
73  }.concat(refGet(efoodRef).unsafeRunSync().map(x => square(x.value._1, x.value._2,
      Purple)))
74
75  def efoodCollision(efoodRef: Ref[IO, List[EvilFood]], newx: Double, newy: Double):
      Boolean =
76    refGet(efoodRef).unsafeRunSync().find(x => x.value == (newx, newy)) match {
77      case None => false
78      case _ => true
79    }
80
81  def square(xr: Double, yr: Double, color: Color) = new Rectangle:
82    x = xr
83    y = yr
84    width = 25
85    height = 25
```

```
86        fill = color
87
88    def compDirection(dir: Any, x: Double, y: Double): IO[(Double, Double)] = IO(
89      dir match {
90        case 1 => (x, y - 25)
91        case 2 => (x, y + 25)
92        case 3 => (x - 25, y)
93        case 4 => (x + 25, y)
94        case _ => (x, y)
95      }
96    )
97
98    def randomFood(evilFood: List[EvilFood], snake: Snake): Food =
99      val food = Food(Random.nextInt(24) * 25, Random.nextInt(24) * 25)
100     dropFood(evilFood, food, snake)
101
102   def dropFood(evilFood: List[EvilFood], food: Food, snake: Snake): Food =
103     if (snake.list.contains(food.value)
104       || evilFood.contains(food.value)) randomFood(evilFood, snake)
105     else
106       food
107
108   def randomEfood(evilFood: List[EvilFood], food: Food, snake: Snake): EvilFood =
109     val efood = EvilFood(Random.nextInt(24) * 25, Random.nextInt(24) * 25)
110     dropEFood(evilFood, efood, food, snake)
111
112   def dropEFood(evilFood: List[EvilFood], efood: EvilFood, food: Food, snake: Snake):
        EvilFood =
113     if (snake.list.contains(efood.value)
114       || evilFood.contains(efood.value)
115       || food.value.equals(efood.value)) randomEfood(evilFood, food, snake)
116     else efood
117
118   def initialSnake: Snake = Snake(List(
119     (250, 200),
120     (225, 200),
121     (200, 200))
122   )
123
124   def initialEvilFood: List[EvilFood] = List(
125     EvilFood((-50, -50)),
126     EvilFood((-50, -50)),
127     EvilFood((-50, -50)),
128     EvilFood((-50, -50)),
129     EvilFood((-50, -50)),
130     EvilFood((-50, -50))
131   )
132
133   def writeNewHighScore(hScore: Int, nScore: Int) = IO(
134     if(hScore < nScore)
135       writeWithResource(nScore.toString, cfg.scoreFilePath).unsafeRunSync()
136   )
```

16.9.5 View

Nel tratto View del progetto Snake viene eseguito l'override del metodo createView(gs: ObjectProperty[GS]) del framework. Viene quindi creata una vista e il content della scena viene impostato per essere aggiornato ad ogni cambiamento dello stato del modello. Il punteggio

massimo raggiunto che viene letto da file attraverso il controller viene mostrato nel campo title dell'applicazione JFXApp accanto al nome del gioco.

```

trait View extends JFXApp3 with Controller:

  override def createView(gs: ObjectProperty[GS]) = new JFXApp3.PrimaryStage {
    width = 600
    height = 635

    scene = new Scene {
      title = "SNAKE | Highscore: " + gs.value.value

      fill = Black
      content = gs.value.shapes
      onKeyPressed = key => readKeys(key)

      gs.onChange(Platform.runLater {
        content = gs.value.shapes
      })
    }
  }
```

16.9.6 Main

Nel file main del progetto il gioco viene lanciato in esecuzione. Le operazioni effettuate sono le seguenti:

- si crea la view dell'applicazione
- si crea uno stato condiviso da usare nei metodi per la gestione della pausa
- attraverso alcuni dei metodi presenti nel modello del framework viene eseguito il motore del gioco che é cosí definito:
  - viene lanciato un loop di effetti sequenziale che in maniera asincrona restitisce subito il controllo dell'edecuzione al thread chiamante
  - all'interno del loop viene lanciato un metodo per l'esecuzione asincrona degli effetti
  - all'interno del metodo per l'esecuzione asincrona si esegue l'aggiornamento dei frame del videogioco incrementandoli ogni 70 millisecondi
  - il tempo di scheduling delle operazioni viene definito con il ritardo previsto dal metodo per l'esecuzione asincrona degli effetti, il quale prevede come parametro in ingresso un tempo di attesa da eseguire prima restituire il controllo al thread che ha chiamato l'effetto
  - il loop di effetti viene eseguito in maniera asincrona pertando restituisce subito il controllo al resto dell'applicazione che prosegue nella sua esecuzione invocando attraverso il metodo frameUpdate un listener sul parametro frame attraverso il quale, ad ogni cambio di frame verrá eseguito un aggiornamento dello stato del gioco, se questo non é in pausa

```

object SnakeFx extends View:

  override def start(): Unit =

```

```
6     val game: IO[Unit] =
7       for {
8         _ <- IO.pure(createView(gs))
9         sState <- Ref.of[IO, State[IO, Int]](State.empty[IO, Int])
10        _ <- IO(handlePause(sState).unsafeRunAsync(_ => ()))
11        _ <- IO(loopSequence.loopEffectsSeq(List(IO(effectsAsync.runAsyncSequence((),
              List(IO(frame.update(frame.value + 1))))
12        (70))))(refCreate(false).unsafeRunSync()))
13        _ <- IO(effectsAsync.runAsyncSequence((), List(IO(frameUpdate(sState))))(0))
14          .handleErrorWith { t =>
15            failedIO(t).as(ExitCode.Error)
16          }
17      } yield ()
18
19    game.unsafeRunSync()
```

Riassumendo i punti fondamentali dell'esecuzione di questo progetto, essi sono:

- ogni 70 millisecondi viene incrementato il numero di frame
- ad ogni incremento del numero di frame viene eseguito un aggiornamento dello stato del gioco, creandone uno nuovo immutabile
- ad ogni aggiornamento di stato la componente View aggiorna il suo contenuto e lo renderizza

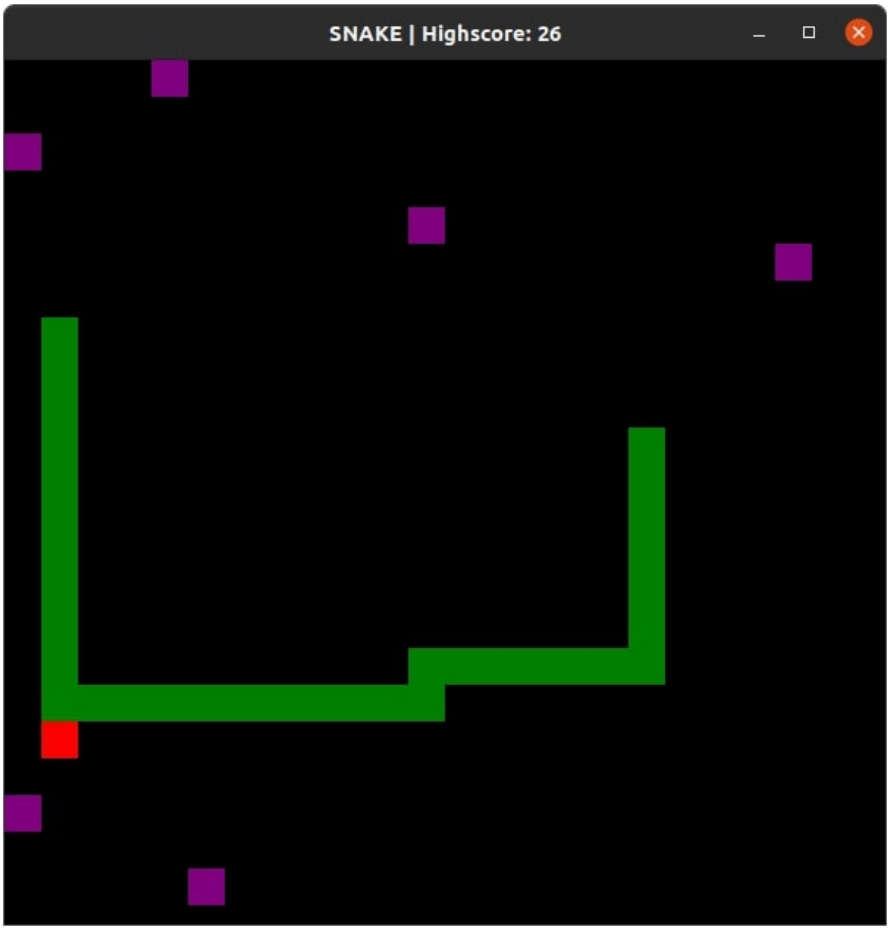

Figura 7: Gameplay di Snake: prima che il cibo da evitare scompaia

17 Retrospettiva

In questa ultima sezione vengono discussi a posteriori quelli che sono stati i lati positivi e negativi riscontrati durante il processo di sviluppo di questo progetto.

17.1 Commenti finali

Lo sviluppo di questo progetto ha sicuramente permesso allo sviluppatore di approfondire la propria conoscenza del linguaggio Scala. Affrontare un progetto come questo in autonomia non é mai facile e doversi cimentare nella realizzazione di un piccolo framework utilizzando una libreria complessa come Cats ha sicuramente aumentato il livello di difficoltá. Apprendere le funzionalitá offerte dalla libreria, renderle disponibili attraverso il framework ed applicarle in maniera soddisfacente al caso di studio non é stato semplice. Ci si puó ritienere soddisfatti dei risultati raggiunti nel progetto ma si é consapevoli che durante questa esperienza si sono maturate le capacitá per fare sicuramente meglio in futuro.

Riferimenti bibliografici

[1] typelevel.org. Async. `https://typelevel.org/cats-effect/docs/typeclasses/async`.

[2] typelevel.org. Eval. `https://typelevel.org/cats/datatypes/eval.html`.