ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

FRP-SCALA: STUDIO E SPERIMENTAZIONE DEL PARADIGMA FUNCTIONAL REACTIVE PROGRAMMING

Christian D'Errico - christian.derrico@studio.unibo.it

Indice

Capitolo 1

Introduzione

La programmazione reattiva è stata proposta come il paradigma di programmazione per applicazioni reattive, *event-driven* e fortemente interattive, ed è profondamente radicata nel paradigma funzionale. Negl'anni, i ricercatori hanno arricchito i vari linguaggi *mainstream* con astrazioni sempre più potenti e applicato la programmazione reattiva a parecchi domini, quali GUI, animazioni, applicazioni Web, robotica e sistemi IoT. La progettazione di sistemi reattivi implica necessariamente l'utilizzo di codice asincrono e la programmazione reattiva (RP) offre al programmatore semplici meccanismi per la gestione di questi aspetti.

1.1 Reactive Programming

Parlare di FRP (*Functional Reactive Programming*) significa ragionare su tipi di dato che rappresentano un valore nel tempo. La programmazione imperativa convenzionale cattura questi valori dinamici solo indirettamente, attraverso stati e mutazioni: tale paradigma è infatti temporalmente discreto. Al contrario, FRP si adatta a queste tipologie di valori direttamente e non ha difficoltà a trattare con tipi che evolvono continuamente. FRP è anche inusuale nel modo in cui affronta problematiche di concorrenza, senza incorrere nelle difficoltà teoriche e pratiche che affligono la programmazione imperativa. Semanticamente, la concorrenza per FRP è *fine-grained, determinate,* e *continuous*. Per riassumere, la comprensione di FRP parte da queste idee:

- valori dinamici in evoluzione (*over time*) sono i principali componenti. Possono essere definiti e combinati, passati come input o output dalle funzioni. Vengono chiamati **behaviors**.

- I *behavior* sono costruiti a partire da poche primitive valutate nel tempo, combinate sequenzialmente e parallelamente. N *behaviors* possono essere associati applicando una funzione n-aria (su valori statici), *point-wise*, cioè continuamente nel tempo.

- I fenomeni discreti sono concepiti come **eventi**, ognuno dei quali ha uno *stream* (finito o infinito) di occorrenze, ognuna delle quali ha associati un tempo ed un valore.

1.1.1 La propagazione del cambiamento

La programmazione reattiva funzionale affronta lo sviluppo di applicazioni *event-driven* provvedendo astrazioni per esprimere dichiarativamente (ovvero nei termini di che cosa devono fare) i programmi, consentendo ai linguaggi di gestire automaticamente il flusso di tempo (concettualmente supportando la simultaneità), e dipendenze di computazioni e dati. In questo paradigma, i cambiamenti di stato sono automaticamente ed efficientemente propagati attraverso la rete di computazioni dipendenti dal sottostante modello d'esecuzione. Di seguito proviamo a spiegare questa dinamica attraverso un semplice esempio di somma di due variabili.
Es:

var1 = 1
var2 = 2
var3 = var1 + var2

In un contesto reattivo, il valore della variabile var3 è sempre mantenuto aggiornato:

- è automaticamente ricomputato nel tempo, quando il valore di var1 e var2 cambia

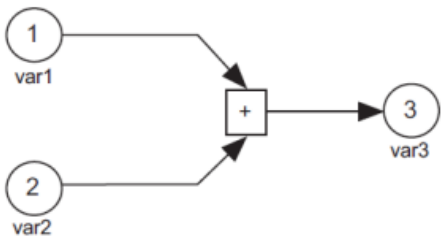
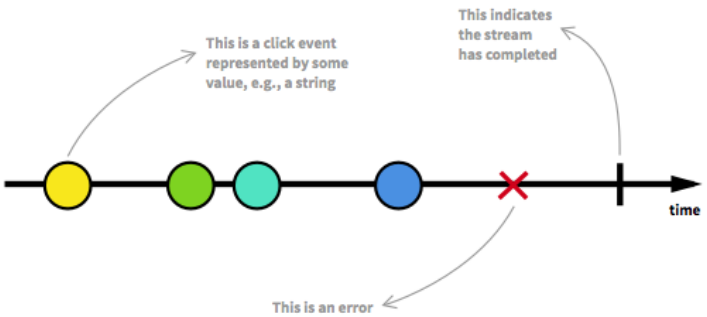


Figura 1.1: Rappresentazione grafica delle dipendenze di un'espressione in un programma reattivo

Il contenuto delle variabili può mutare nel tempo e quando ciò si verifica tutte le computazioni dipendenti sono automaticamente re-eseguite. Questo concetto di gestione automatica delle dipendenze avviene anche all'interno di un qualsiasi foglio di calcolo elettronico; qualora venisse scritta la formula dell'esempio sopra, ad ogni cambiamento delle celle che contengono i primi due valori, verrebbe automaticamente aggiornato anche il valore della cella contenente la somma. La programmazione reattiva può essere considerata un foglio elettronico incorporato in un linguaggio di programmazione.

1.1.2 Flussi asincroni di dati e eventi

Non è scorretto definire FRP come programmazione con flussi asincroni di dati. Bus degl'eventi o tipicamente eventi generati da click sono realmente flussi di dati asincroni, che possono essere osservati o a partire dai quali si possono performare alcuni *side effects*. *Reactive* significa questo: poter essere in grado di creare *stream* di eventi di qualsiasi cosa, non solo a partire da click, ma anche da variabili, input dell'utente, strutture dati.. Aderire a FRP significa poter maneggiare un *toolbox* di funzioni per combinare, creare e filtrare ognuno di questi *streams*. Qui subentra la magia funzionale, poichè uno *stream* può diventare l'input per un altro (così come anche multipli *streams*), oppure se ne possono fondere due, filtrarne uno per ottenerne un altro solo per quegl'eventi di cui si è interessati, si possono mappare i valori di uno *stream* in uno completamente nuovo.
Un tale flusso di dati può essere definito come una **sequenza di eventi nel tempo**, che possono essere catturati solo **asincronicamente** da un **subscriber** che si mette in ascolto sulla o sulle sorgenti, modellando nella pratica l' *Observer Design Pattern*.



1.1.3 Casi d'uso

Il funzionamento delle moderne applicazioni è guidato da ogni sorta d'evento che si verifica all'interno o all'esterno di questi sistemi, che pertanto hanno la necessità di mantenere un'interazione continua con il proprio ambiente, processando eventi e performando task corrispondenti. La parte più interattiva di tali applicazioni è solitamente la GUI, che tipicamente necessita di reagire e coordinare multipli input. Sistemi del genere sono difficilmente pensabili usando approcci sequenziali convenzionali, poichè è impossibile predire o controllare l'ordine di arrivo di eventi ed eseguire *handler* al cambiamento inaspettato dell'ambiente esterno (*inverted control*, il flusso di controllo di un programma è guidato da eventi esterni e non nell'ordine specificato). Una tale gestione manuale delle modifiche allo stato e alle dipendenze dei dati è complessa e soggetta ad errori. Soluzioni di programmazione tradizionali pensano le applicazioni interattive come tipicamente costruite intorno alla nozione di *callback* asincrone (*event handlers*). Sfortunatamente, coordinarne l'esecuzione può essere un compito molto arduo anche per i programmatori più esperti, poiché vi possono essere funzioni in concorrenza sugli stessi dati, e il loro ordine di esecuzione è imprevedibile. In letteratura, il problema della gestione delle chiamate è noto come *Callback Hell*. Il paradigma reattivo funzionale cerca di ovviare a tutto questo: propone un modello dichiarativo che astrae oltre la gestione del tempo, esattamente come fanno i *garbage collectors* che permettono al programmatore di non preoccuparsi della gestione delle risorse. Le prime ricerche su questo paradigma hanno condotto a **Fran**, un *domain specific language* concepito alla fine degl'anni '90 per facilitare la costruzione di applicazioni grafiche e interattive. Partendo da qui, l'interesse per FRP si è esteso anche ad altri domini applicativi, come la programmazione web, la modellizzazione e la simulazione di sistemi, la robotica, la computer vision e perfino alcune applicazioni di *stage lighting* (illuminazione scenica).

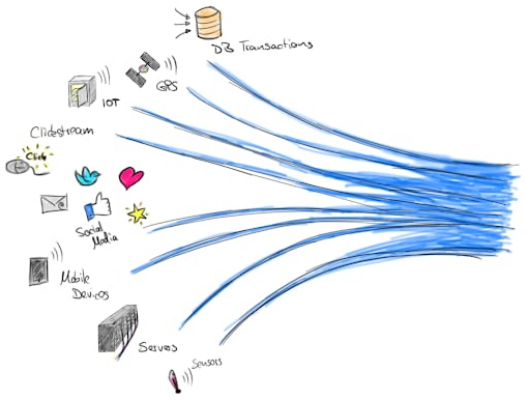


Figura 1.2: *Everything is a stream*

1.1.4 RP Abstractions

Abbiamo già parlato dei due principali tipi di astrazione che la programmazione funzionale reattiva definisce:

- **Event**
  - modella un valore discreto che occorre in un determinato momento nel tempo.
  - è un astrazione asincrona per il flusso progressivo di sequenze di dati intermittenti che provengono da una situazione ricorrente (come ad esempio gli eventi provenienti dal movimento del *mouse*).

- **Behavior**
  - valore che cambia continuamente nel tempo.
  - rappresenta un flusso ininterrotto di dati uscenti da una sorgente statica (ad esempio un *timer*).

Delle due, i *behaviors* rappresentano una sfida implementativa significativa, poichè la loro natura continuamente mutevole implica che quando venga incrementato il *rate* di emissione dei dati, siano prodotti valori più precisi. Nei linguaggi di natura *lazy* inoltre (**Haskell** per esempio), maneggiare questa tipologia di sorgente dati permette di calcolare solo su richiesta i valori prodotti, mentre per i restanti linguaggi, il *behavior*, per il programmatore, è in continuo cambiamento, poichè per ogni istante di tempo emette un valore differente, ma che deve essere regolarmente campionato.

1.1.5 Modello di valutazione

Il modello di valutazione di un linguaggio di programmazione reattivo è legato a come i cambiamenti vengono propagati nel grafo delle dipendenze. Il cambiamento di un valore dovrebbe essere automaticamente passato a tutte le computazioni dipendenti, che devono essere notificate per innescare un ricalcolo. A livello di linguaggio, la decisione di *design* che necessita di essere presa riguarda chi debba incominciare la diffusione dell'aggiornamento, ovvero se la sorgente debba inviare i nuovi dati ai *listeners* o se siano questi a dover richiedere le informazioni. Il paradigma reattivo prevede due diversi modelli:

- **Pull-Based**: la computazione che richiede un valore deve "rivolgersi" alla sorgente: la propagazione pertanto è guidata dalla domanda di nuovi dati (*demand driven*). Questo modello offre flessibilità alla computazione che richiede il valore, poichè ciò può essere fatto solo in caso di effettiva

necessità. L'obiezione più ricorrente rivolta a questo scenario riguarda la possibile latenza significativa fra l'occorrenza di un evento e il momento in cui si verifica la reazione.

- **Push-Based**: quando la sorgente ha un nuovo dato, lo invia a tutte le sue computazioni dipendenti. La propogazione è guidata dalla disponibilità di nuovi elementi. I linguaggi che adottano il modello *push based* (ovvero pressochè tutti i linguaggi non *lazy*) devono adottare una soluzione efficiente per evitare il problema dei calcoli inutili, poichè questi vengono richiesti ogni volta che la sorgente input cambi.

Esistono linguaggi di programmazione reattiva che adottano entrambi i modelli e quindi possono usufruire dei vantaggi offerti da entrambi: per il modello *push-based*, efficienza e bassa latenza, per il modello *pull-based*, la flessibilità di richiamare valori all'occorrenza.

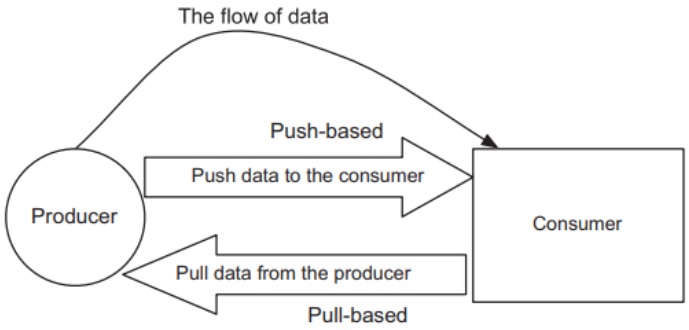


Figura 1.3: Modello *Push* vs Modello*Pull*

1.1.6 Glitch

L'**Assenza di glitch** è un'altra proprietà che necessita di essere considerata da un linguaggio reattivo. I *glitches* sono aggiornamenti inconsistenti che possono verificarsi durante la propagazione dei cambiamenti. Quando una computazione viene eseguita prima di tutte le altre espressioni dipendenti, può risultare in valori fresh che vengono combinati con valori statici, portando ad un *glitch*. Ciò si può verificare solamente in modelli di valutazione *push-based*.
Es:

var1 = 1
var2 = var1 * 1
var3 = var1 + var2

Il valore di var2 è sempre uguale a var1. Qualora il valore di quest'ultima cambiasse, e prima del calcolo di var2, venisse aggiornato il valore di var3 con il nuovo valore fresh di var1 e con quello vecchio di var2, avremmo un valore inesatto di var3, che non sarebbe uguale al doppio di var1 (scenario corretto). Le più recenti implementazioni evitano i *glitch* eseguendo su un singolo pc, ma non in programmi distribuiti, dove errori di rete, ritardi e la mancanza di un *clock* globale potrebbero rendere il tutto molto più difficile.

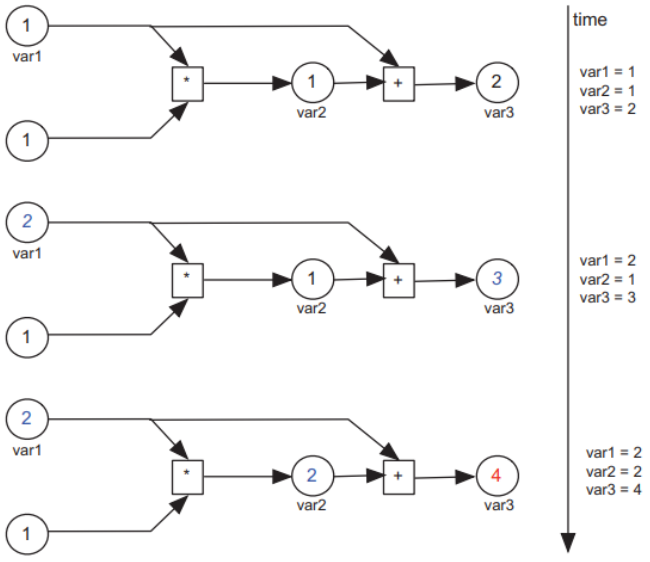


Figura 1.4: Glitches: visualizzazione momentanea dello stato inconsistente di un programma e ricalcolo

1.1.7 Lifting

Quando la programmazione reattiva è inclusa in linguaggi *host* (sia come libreria o estensione), operatori esistenti e funzioni definite dagl'utenti o metodi devono essere convertite per poter lavorare sulle variabili reattive. Questa conversione si definisce **lifting**. Ciò può avvenire implicitamente: un operatore tradizionale, che viene applicato ad un *behavior*, viene automaticamente convertito in un operatore in grado di lavorare con il *behavior* stesso; oppure esplicitamente: il linguaggio fornisce una serie di primitive che consentono di trasformare l'operazione ordinaria e permetterle di lavorare con valori che cambiano nel tempo. Alternativamente a ciò, potrebbere non essere previsti operatori di *lifting*, e pertanto il programmatore potrebbe essere costretto a ottenere i risultati, che nel frattempo potrebbero essere mutati, manualmente.

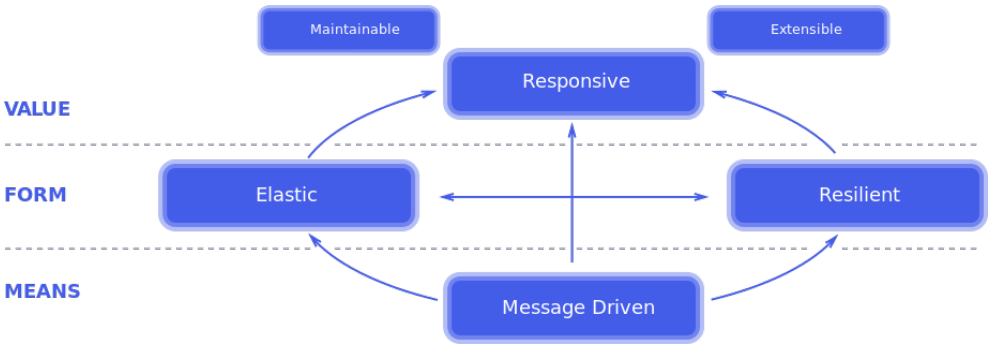
1.2 Reactive Manifesto

Ma quali caratteristiche arichitetturali deve avere un sistema per essere considerato reattivo? Enti ed aziende operanti in vari settori se lo sono chiesto e hanno stabilito che fosse necessario un approccio coerente all'architettura di tali sistemi*software*, per i quali gli elementi chiave fossero già individuabili singolarmente. Un sistema, infatti, è reattivo se **responsivo**, **resiliente**, **elastico** e **orientato ai messaggi**:

- **Responsivo**: il sistema, se è generalmente possible dare una risposta ai client, lo fa tempestivamente. La responsività è la pietra miliare dell'usabilità e dell'utilità; essa presuppone che i problemi vengano identificati velocemente e gestiti in modo efficace. I sistemi responsivi sono focalizzati a minimizzare il tempo di risposta, individuando per esso un limite massimo prestabilito in modo da garantire una qualità del servizio consistente nel tempo. Il comportamento risultante è quindi predicibile, il che semplifica la gestione delle situazioni di errore, genera fiducia negli utenti finali e predispone ad ulteriori interazioni.
- **Resiliente**: il sistema resta responsivo anche in caso di guasti. Ciò riguarda non solo i sistemi ad alta disponibilità o *mission-critical*: infatti, accade che ogni sistema che non è resiliente si dimostrerà anche non responsivo in seguito ad un guasto. La resilienza si acquisisce tramite replica, contenimento, isolamento e delega. I guasti sono relegati all'interno di ogni componente, isolando ognuno di questi dagl'altri e quindi garantendo che il guasto delle singole porzioni del sistema non ne comprometta l'intero funzionamento. Il recupero di ogni componente viene delegato ad un altro (esterno) e l'alta disponibilità viene assicurata

tramite replica laddove necessario. I *client* di un componente vengono dunque sollevati dal compito di gestirne i guasti.

- **Elastico**: Il sistema rimane responsivo sotto carichi di lavoro variabili nel tempo. I Sistemi Reattivi possono adattarsi alle variazioni nella frequenza temporale degli input incrementando o decrementando le risorse allocate al processamento degli stessi. Questo porta ad architetture che non hanno né sezioni contese né colli di bottiglia, favorendo così la distribuibilità o la replica dei componenti e la ripartizione degli input su di essi. I Sistemi Reattivi permettono l'implementazione predittiva, oltre che Reattiva, di algoritmi scalabili perchè fondati sulla misurazione *real-time* della *performance*. Tali sistemi, raggiungono l'elasticità in maniera *cost-effective* su *commodity hardware* e piattaforme *software* a basso costo.
- **Orientati ai messaggi**: I Sistemi Reattivi si basano sullo scambio di messaggi asincrono per delineare per ogni componente il giusto confine che possa garantirne il basso accoppiamento con gli altri, l'isolamento e la trasparenza sul dislocamento e per permettere di esprimere i guasti del componente sotto forma di messaggi al fine di delegarne la gestione. L'utilizzo di uno scambio esplicito di messaggi permette migliore gestibilità del carico di lavoro, elasticità e controllo dei flussi di comunicazione mediante *setup* e monitoraggio di code all'interno del sistema e con l'applicazione di *feedback* laddove necessari. Uno scambio di messaggi trasparente rispetto al dislocamento rende possibile, ai fini della gestione dei guasti, l'utilizzo degli stessi costrutti e semantiche sia su *cluster* che su singoli *host*. Uno stile di comunicazione non bloccante fa sì che l'entità ricevente possa solo consumare le risorse, il che porta ad un minor sovraccarico sul sistema.



Capitolo 2

Reactive Frameworks

Varie sono state le soluzioni proposte per fornire un supporto alla programmazione reattiva. Nonostante ogni *framework* segua le sue logiche e faccia le sue scelte progetturali, un minimo comune denominatore è da ricercare in:

- **Composizionalità** e **Leggibilità**: raggiungono grandi risultati nell'orchestrazione di multiple computazioni asincrone, in cui i risultati dei *task* precedenti vengono utilizzati come input per quelli successivi, e, in quanto dichiarativi, il codice risulta nella maggioranza dei casi più comprensibile, permettendone una lettura più agile e orientata all'acquisizione di una maggior consapevolezza del funzionamento della *business logic*.
- **Flussi di dati manipolati grazie ad un ricco vocabolario di operatori**: esiste un ampio numero di operatori, ognuno dei quali dota un *publisher* di un "comportamento" che influenza la costituzione del flusso lungo i passaggi composizionali. L'intera struttura diventa così linkata, che i dati originano da un primo *step* e si "trasformano" nello spostamento lungo la catena: tante sono le operazioni possibili, da semplici *map* e *filter* a complesse orchestrazioni e gestioni degl'errori.
- **Nulla accade fino alla sottoscrizione**: una sorgente non emette dati fino al verificarsi di una sottoscrizione a questa. Fino a questo momento, quello che si ha è una descrizione astratta di processi asincroni (*configuration stage*). Poi, è questa operazione che determina il *binding* fra *publisher* e *subscriber* e che innesca il flusso di dati nell'intera catena.
- **Backpressure**: con questa espressione si intende un meccanismo di *feedback* retropropagato alla sorgente di emissione che si verifica qualora il *consumer* non riesca a reggere la velocità di lavoro del *producer*. In particolare, un *subscriber* può lavorare in modalità *unbounded* e permettere

alla *source* di emettere tutti i suoi dati alla maggior velocità raggiungibile, oppure può avvalersi del meccanismo di *backpressure* per indicare che è pronto a processare al massimo n elementi. Ciò rende un modello *push* un modello ibrido *push-pull*: *downstream* possono essere elaborati al più n elementi provenienti dall'*upstream*, se questi sono già disponibili, ma qualora non lo siano, si ottengono una volta prodotti.

2.1 Reactive Monix

Fatta la premessa iniziale, è tempo di discutere di **Reactive Monix**, un sottoprogetto di Monix, un libreria Scala per comporre programmi asincroni *event-based*. Monix nasce come un'implementazione di **ReactiveX**, con forti influenze provenienti dalla programmazione funzionale ed è stata progettata per interagire con la *standard library* di Scala, e per rimanere compatibile con **Reactive Streams**. Successivamente si è poi ampliata per includere astrazioni per la gestione dei *side effects* e delle risorse, essendo uno dei "genitori" di **Cats Effect**. Il sottoprogetto *reactive* ruota attorno alla definizione del tipo di dato **Observable** e le sue *utilities*, un'astrazione ad alte performances che è un'implementazione idiomatica di *ReactiveX* per Scala.

2.1.1 Observable ed Observer

Observable è un tipo di dato che modella *stream* di eventi asincroni e reattivi. Potremmo pensare ad un *Observable* come l'equivalente asincrono di un *Iterable* o di uno *Stream*, in un'implementazione che scala a problemi complessi, appoggiandosi sulla *Functional Reactive Programming* e modellando interazioni fra *producers* e *consumers*, e costituendo una potente alternativa al modello ad attori.

Parlando dell'astrazione *Observable* offerta da Monix possiamo affermare che:

- modella *streaming* di eventi asincroni in modo *lazy*.
- si tratta di un'astrazione fortemente componibile, che amplia ed estende rielaborando, in accordo con i concetti introdotti dalla programmazione reattiva, il pattern *Observer*, un pattern canonico per la gestione di molti sistemi ad eventi, e il pattern *Iterator*, in quanto la sequenza osservabile emette i suoi valori in ordine, come un iterator, e lo fa non appena disponibili.
- modella relazioni *producer-consumer*, in cui un singolo *producer* può passare dati a uno o più *consumers*: *observable* assume infatti un ruolo

simile a quello che riveste l'entità *producer* per il pattern *observer*, che emette i valori ai suoi *listener*, che in questo caso abbiamo chiamato *observer*.

A fronte di tutto ciò che rappresenta questo tipo di astrazione, nella pratica, lavorare con un tale modello asincrono di elaborazione dei dati significa:

- Definire una callback per il valore di ritorno della chiamata asincrona che viene eseguita dal *consumer*, che nel nostro contesto definiamo **Observer**.
- La chiamata asincrona stessa è espressa come *Observable*, *producer* del risultato
- Le due entità vengono collegate dall'atto di sottoscrizione dell'*observer* sull' *observable* (azione che avvia l'emissione dei dati e le operazioni dell'*observable*).
- Quando la chiamata asincrona ritorna, l'*observer* inizierà a lavorare con i valori disponibili - gli elementi emessi dall'*observable*.

La sottoscrizione di un *observer* ad un *observable* fa si che quest'ultima entità rispetti il contratto dell'*observer* durante il passaggio degl'elementi, contratto che prevede l'implementazione di un *subset* dei seguenti metodi:

- **onNext**: viene invocato quando l'*observable* emette un elemento. Prende come parametro l'elemento emesso dall'*observable*.
- **onError**: l'*Observable* chiama questo metodo quando si è verificato un fallimento nella generazione dei dati, o ha incontrato qualche errore.
- **onCompleted**: invocato successivamente all'ultima chiamata *onNext* se tutte le operazioni si sono svolte correttamente.

Per soddisfare le necessità di un *observer*, un *observable* implementa **subscribe**, metodo che viene invocato nel passaggio degl'elementi al *consumer*, concretizzato nelle chiamate ad *onNext*, e, al termine, *onComplete*. Il metodo *onNext* ritorna una *future* con il messaggio di *acknowledge* (*continue* per indicare di proseguire con l'emissione o *stop* per arrestare il flusso), che, per preservare l'idea di *backpressure* asincrona, costringe l'*observable* ad attenderne il risultato prima di passare l'elemento successivo. Questo aspetto è fondamentale e deriva dal fatto che *Reactive Monix* è compatibile e interoperabile con *Reactive Streams*, un'iniziativa che è all'origine di molti *framework* reattivi moderni e che ha cercato di creare uno standard per il *processing* di *stream* asincroni di dati con una *backpressure* non bloccante. Un problema fondamentale infatti che è comportato dalla gestione di *stream* di dati, specialmente *live data* di cui non è noto a priori il volume, richiede un'attenzione speciale in sistemi

asincroni, in particolare per quanto riguarda il consumo di risorse, in maniera tale per cui una sorgente dati veloce non sovrasti la destinazione dello stream. Asincronia è necessaria per permettere l'uso parallelo di risorse di calcolo, su reti di *host* o multipli *core* all'interno di una singola macchina. L'idea è governare lo scambio di *data stream* in un contesto asincrono, assicurandosi però che il lato ricevente non venga forzato al *buffering* di un ammontare arbitrario di dati. La *back pressure* risulta dunque una parte integrale del modello nel modo in cui permette alle code che mediano fra i *threads* di essere *bounded*. I benefici del *processing* asincrono sarebbero altrimenti negati se la comunicazione della *back pressure* fosse sincrona. Quando si crea un *observable*, non accade nulla fino alla chiamata *subscribe*. Il *processing* è innescato solo in questa determinata situazione, che comincia in *background*, e ritorna un'istanza di *Cancelable* che può essere utilizzata per stoppare lo *streaming*. In un contesto prettamente funzionale, qualora si volesse combinare i risultati degli *stream*, un *observable* si può convertire in un **Task**, una specifica made in Monix per computazioni *lazy* o asincrone, che quando eseguite produrranno un risultato, insieme a possibili *side-effects*: ciò è possibile o utilizzando un **Consumer** - una tipologia di *subscriber* che può essere pensata come una funzione che effettua appunto questa conversione - oppure utilizzando i **foldLeft Methods**, metodi il cui nome presenta un suffisso in L che indica la possibilità di trasformare *observables* in *tasks*.

```
trait Observable[+A] {
  def subscribe(o: Observer[A]): Cancelable
}
```

Figura 2.1: *Observable contract*

```
trait Observer[-T] {
  def onNext(elem: T): Future[Ack]

  def onError(ex: Throwable): Unit

  def onComplete(): Unit
}
```

Figura 2.2: *Observer contract*

2.1.2 Operators

La maggior parte degl'operatori messi a disposizione da *Reactive Monix* lavora con un *observable* e produce un *observable*. Ciò permette di applicare questi operatori a catena, uno dopo l'altro. Ogni operatore restituisce un nuovo *observable* che risulta dall'operazione del precedente operatore: l'ordine, pertanto, con cui questi vengono applicati, risulta fondamentale ai fini del risultato che si vuole ottenere. Esistono operatori che coprono ogni aspetto del ciclo di vita di un *observable*:

- **Creazione**: operatori che generano nuovi *observable*
- **Trasformazione**: operatori che trasformano gli elementi emessi da un *observable*
- **Filtering**: operatori che selettivamente emettono elementi da una sorgente osservabile
- **Combinazione**: operatori che lavorano con multiple sorgenti per crearne una singola composta
- **Condizionali e operatori booleani**: operatori che valutano uno o più *observable* o gli elementi emessi da questi
- **Matematici e operatori di aggregazione**: operatori che operano su intere sequenze di elementi emessi da un *observable*
- **Backpressure**: strategie per lavorare con *observable* che producono elementi più rapidamente di quanto non li consumino i loro *observer*.

2.1.3 Creating Observable

Questi metodi sono disponibili via *observable companion object*. Ecco alcuni esempi:

Observable.pure

Effettua il *lifting* di un valore nel contesto *observable*:

```
def getObservableFromSingleValue[A](value: A): Observable[A] =
    Observable.pure(value)
```

Observable.fromIterable

Converte un *Iterable* in un *observable*:

```
def getObservableFromIterable[A](elems: Iterable[A]): Observable[A] =
    Observable.fromIterable(elems)
```

Observable.range

Converte un *range* in un *observable*:

```
def getRangeObservable(from: Long, to: Long): Observable[Long] =
    Observable.range(from, to)
```

Observable.eval

Costruisce un *observable* a partire da una espressione *non-strict*:

```
def getObservableFromNotStrictValue[A](exp: => A): Observable[A] =
    Observable.eval(exp)
```

Observable.create

Costruisce un *observable* a partire da un *subscriber*. Rappresenta il modo sicuro di costruire *observables* da fonti di dati che non possono essere sottoposte a *backpressure*:

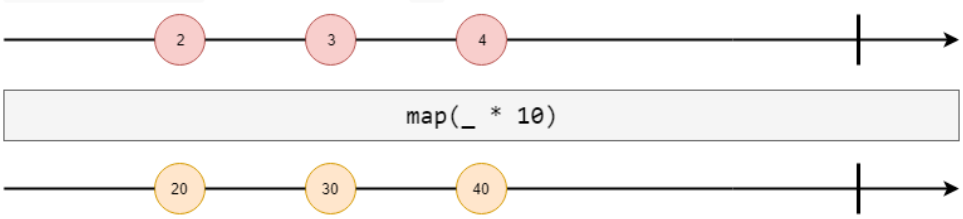
```
Observable.create[Int](OverflowStrategy.Unbounded){ source =>
    for(i <- 0 until elemNumber)
      source.onNext(i)
    source.onComplete()

    Cancelable()
  }
```

2.1.4 Working with Observable

Nota: Vengono mostrati solo alcuni esempi di operatori, dato il cospicuo numero di questi. Altri ne sono stati utilizzati e ne è stato sperimentato il funzionamento nella costruzione della versione *Monix* del *Conway's Game of Life*.

Map: trasforma ogni elemento della sequenza e come parametro di ingresso accetta la funzione di trasformazione.



Nell'esempio che segue, per mezzo dell'operatore *map* ogni elemento del seguente *observable* viene convertito nel suo doppio:

```
Observable.range(0, 10)
         .map(_ * 2)
```

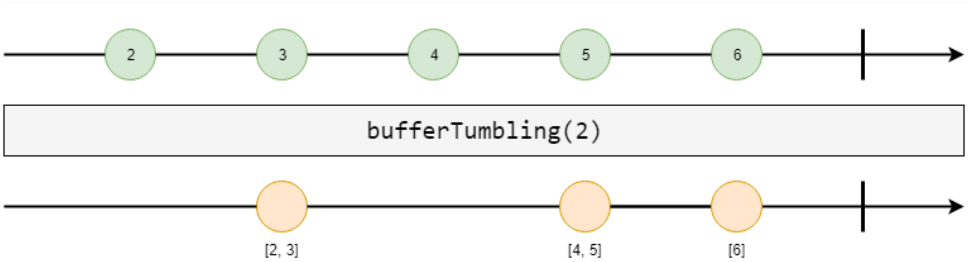
Filter: emette soltanto gl'elementi di un *observable* che passa il test di un predicato. Riprendendo l'esempio sopra, si decide di far passare solo gli elementi pari:

```
Observable.range(0, 10)
         .filter(_ % 2 == 0)
         .map(_ * 2)
```

Scan: applica una funzione ad ogni elemento di un *observable*, sequenzialmente, ed emette elementi ogni volta.
Applicato al precedente *observable*, *scan* produce la somma dell'ultimo elemento con gli elementi precedentemente emessi:

```
Observable.range(0, 10)
         .filter(_ % 2 == 0)
         .map(_ * 2)
         .scan(0L)((acc, next) => acc + next)
```

BufferTumbling: raccoglie periodicamente gli elementi in uscita da un *observable* in sequenze ed emette queste invece degl'elementi uno alla volta.



Applicato all'*observable* costruito in precedenza, gli elementi emessi da *scan* sono raccolti in una sequenza di 5 elementi.

```
Observable.range(0, 10)
          .filter(_ % 2 == 0)
          .map(_ * 2)
          .scan(0L)((acc, next) => acc + next)
          .bufferTumbling(5)
```

2.1.5 Combining Operators

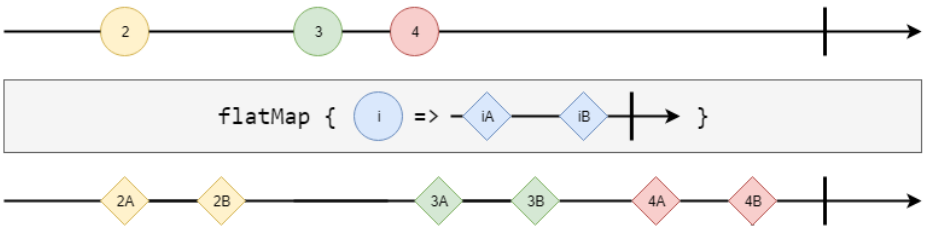
I seguenti operatori permettono di combinare multipli *observables* per costruire singoli *observable* composti:

ZipMap2: Crea un nuovo *observable* da due *observables* (è possibile la stessa operazione anche con più *observable*) combinando i loro elementi a coppie in una *strict sequence* applicando la funzione passata come input.

Nell'esempio viene prodotto un *observable* che emette elementi provenienti da un *iterable* ad intervalli di tempo scanditi da un altro *observable*.
Observable.interval infatti costruisce un *observable* che emette, incrementando di un'unità ad ogni emissione, numeri naturali distanziati da un determinato intervallo di tempo.

```
Observable.zipMap2(
  getObservableFromIterable(iterable),
  Observable.interval(interval)
)((first, _) => first)
```

FlatMap (alias per ConcatMap): applica una funzione a ogni elemento emesso dall'*observable* d'origine, funzione che restituisce un *observable* che viene concatenato alla prima sorgente producendo poi un *observable* finale, risultante da questa associazione, di cui vengono emessi gli elementi.



Nell'esempio, partendo da due sorgenti dati si costruisce un'unica sorgente che produce tuple di valori provenienti dai due *observables* di partenza.

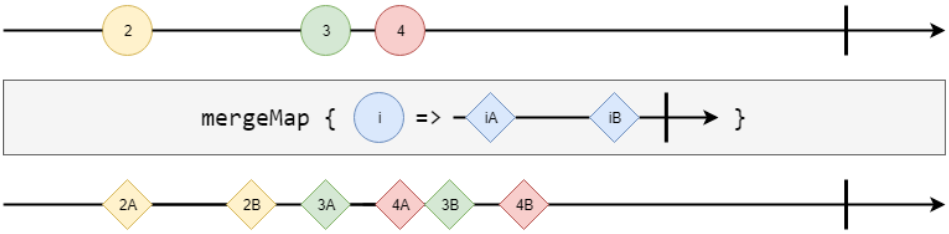
```
val sourceA = getStrictTimedSource(List(0, 1, 2, 3), 0.3.seconds)
val sourceB = getStrictTimedSource(List(5, 6, 7), 0.8.seconds)

val concatObs = sourceA.concatMap(v1 => sourceB.map(v2 => (v1, v2)))

/*val concatObs = for {
  v1 <- sourceA;
  v2 <- sourceB
} yield (v1, v2)*/
...
val expectedResult = List((0, 5), (0, 6), (0, 7),
                          (1, 5), (1, 6), (1, 7),
                          (2, 5), (2, 6), (2, 7),
                          (3, 5), (3, 6), (3, 7))
```

Il risultato, come si può vedere, è costituito da tutte le possibili associazioni degl'elementi nell'ordine con cui vengono emessi dalle sorgenti.

MergeMap: accetta una funzione con la quale produce un unico *observable* risultante dal *merge* degl'elementi prodotti da due sorgenti.

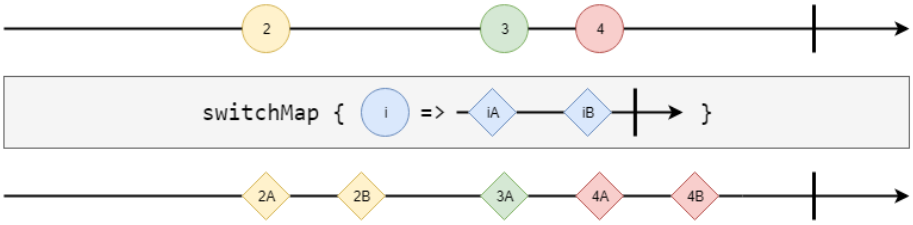


Ancora una volta, nell'esempio viene prodotto un *observable* che emette tuple di valori a partire da due *observable* differenti.

```
val sourceA = getStrictTimedSource(List(0, 1, 2, 3), 0.3.seconds)
val sourceB = getStrictTimedSource(List(5, 6, 7), 0.8.seconds)
...
val mergedObs = sourceA.mergeMap(v => sourceB.map(elem => (v, elem)))
...
val expectedResult = List((0, 5), (1, 5), (2, 5),
                          (0, 6), (3, 5), (1, 6),
                          (2, 6), (0, 7), (3, 6),
                          (1, 7), (2, 7), (3, 7))
```

Il risultato questa volta è però una sequenza di elementi combinati a partire dall'ordine di arrivo degl'elementi prodotti dalle due sorgenti.

SwitchMap: è un operatore simile a *mergeMap*, ma nella combinazione, dà precedenza all'*observable* che ha emesso l'elemento più recente, e cancella l'*inner stream* (l'*observable* passato come input) di modo che vi sia solamente una sottoscrizione attiva alla volta.



A partire dalle medesime sorgenti degl'esempi precedenti, viene prodotto un *observer* utilizzando *switchMap*

```
val sourceA = getStrictTimedSource(List(0, 1, 2, 3), 0.3.seconds)
val sourceB = getStrictTimedSource(List(5, 6, 7), 0.8.seconds)
...
val switchedObs = sourceA.switchMap(v => sourceB.map(elem => (v,
    elem)))
...
val expectedResult = List((0, 5), (1, 5), (2, 5),
                          (3, 5), (3, 6), (3, 7))
```

Il risultato dell'applicazione di *switchMap* è simile a quanto prodotto da *mergeMap*, con l'importante differenza che, la prima sorgente, avendo un *rate* più alto della seconda, ha la precedenza nella combinazione, e quest'ultima fonte viene "presa in considerazione" solo al termine del passaggio dei dati della prima sorgente.

2.1.6 Cold and Hot Observables

Reactive Monix distingue due ampie categorie di stream: **hot** e **cold**. Questa distinzione principalmente riguarda come lo stream reattivo reagisce ai *subscribers*:

- uno *stream cold* inizia un nuovo *stream* per ogni *subscriber*, emettendo segnali soltanto se presente almeno uno di questi. Il modello in questo caso è *pull*, poichè gli elementi sono provvisti in modalità *lazy*.

Es:

```
val obsFromZippedSources = Observable.zipMap2(
    Observable.fromIterable(List(1,2,3,4)),
    Observable.fromIterable(List(1,2,3,4))
)(_+_)
...
obsFromZippedSources.toListL.runToFuture

checkResults(expected = List(2,4,6,8), obtained = result)
```

Il risultato in questo caso sarà uguale ad ogni eventuale sottoscrizione e corrisponderà sempre alla alla lista contenente le somme dei dati delle due sorgenti di cui è effettuato lo *zip*.

- al contrario, uno *stream hot* non parte da zero per ogni *subscriber*:
  - questi ricevono segnali emessi dopo che loro si sono sottoscritti
  - alcuni *hot reactive streams* possono memorizzare o replicare la storia delle emissioni totalmente o parzialmente.
  - da una prospettiva generale, una sequenza *hot* può emettere anche quando nessun *subscriber* è in ascolto (un'eccezione a "nulla accade prima di una sottoscrizione").
  - si tratta di un *push model*: vi è un totale disaccoppiamento fra il flusso di controllo che genera gli elementi e i *subscribers*.
  - possono essere utilizzate tecniche di *backpressure*, qualora l'*observer* non sia in grado di consumare elementi velocemente quanto questi vengano prodotti da un *observable*.

Es:

```
val begin = 0
val end = 10_000

val testList = (begin until end).toList
...
val throttleTime = 0.1.seconds
val delayTime = 1.seconds

val nElementsToTake = 10
val index = ((delayTime / throttleTime) - 1).toInt

val source = Observable(testList).throttle(throttleTime, 1)
                                 .publish
source.connect()

val result = getFirstElem(source.take(nElementsToTake)
                                .delayExecution(delayTime))

checkResults(testList(index), result)
```

Throttle è un operatore che regola l'emissione dei dati in un intervallo di tempo prestabilito che viene fornito in input.

Take applicato ad una sorgente fa si che di questa vengano presi i primi n elementi, dove n è il parametro passato all'operatore in questione.

Data la natura dell'*hot stream* creato, ci aspettiamo che questo emetta un elemento ogni decimo di secondo; pertanto, il primo elemento catturato da un'eventuale sottoscrizione che avviene con un delay pari ad un secondo, sarà il decimo elemento della sequenza di numeri compresa fra 0 e 10.000, proprio perchè l'*hot stream* (che nell'ecosistema *Reactive Monix* è definito dal tipo *ConnectableObservable*) avvia la trasmissione dei dati con una chiamata al metodo *connect*, indipendentemente dalle connessioni di eventuali *subscribers*.

2.2 Backpressure

Nel passaggio fra i vari stadi di composizione di un flusso possono essere fatte molte cose e a velocità differenti. Per evitare problemi quali consumo di memoria o perdita di informazioni, poichè è necessario effettuare *buffering* o *dropping* degl'eventi, vengono applicati i già citati meccanismi di *Backpressure*.

- Si tratta di una forma di controllo di flusso, in cui gli step composizionali possono esprimere quanti elementi sono in grado di processare.

- questo permette la limitazione nell'uso di memoria dei flussi di dati in situazioni in cui non c'è generalmente modo di sapere quanti elementi verranno inviati dalla sorgente di ingresso.

E questo è esattamente ciò che si verifica in situazioni di questo genere:

```
val bufferCapacity = 100
val rangeLimit = 1000
val highVelocityEmitter = getRangeObservable(from = 0, to =
    rangeLimit)

val emitFunction: Subscriber.Sync[Long] => Cancelable = producer => {
    highVelocityEmitter.map(e => producer.onNext(e.toInt))
                       .toListL
                       .runToFuture
                       .onComplete(_ => producer.onComplete())
    Cancelable()
  }

val source = createSourceWithBackPressurePolicy[Long](emitFunction)(_)
```

Observable.create permette proprio di costruire una sorgente dati a partire da fonti non *backpressurable*, poichè imprevedibili nel "comportamento" a priori.

Alcuni operatori già citati come **throttle** o **bufferTumbling** possono essere messi in campo come strategie per regolare la velocità di emissione o la quantità di elementi processabili, ma nello stesso tempo sono messe a disposizione degli sviluppatori diverse *policies* con cui affrontare problemi di questa specie.

Nell'esempio sopra, si è costruita una sorgente che emette, senza limitazioni, 1000 elementi. Un nuova sorgente viene poi costruita a partire da un *subscriber* che produce elementi a partire dalla prima sorgente.

Quello che ci si aspetta è che il *subscriber* non sia in grado di mantenere la velocità sostenuta dalla prima fonte dati, e che quindi saranno necessarie diverse strategie per regolare questa discrepanza.

Si è scelto di dotare la seconda sorgente di un *buffer* di 100 elementi per conservare le emissioni della prima variabile reattiva in attesa di consumazione: in ogni esempio, quello che si fa è stabilire una policy di gestione dell'*overflow* di tale *buffer*.

In questa prima situazione, al riempimento del buffer viene lanciata una *BufferOverflowException*: con questa strategia non viene, infatti, ammessa la possibilità per cui il buffer possa riempirsi.

```
val obsOfFail = source(OverflowStrategy.Fail(bufferCapacity))

val errorDetector =
    createSourceWithBackPressurePolicy[Option[Throwable]] {
  producer =>
    obsOfFail.subscribe(
      _ => Future(Continue),
      errorFn = e => {
        producer.onNext(Some(e));
        producer.onComplete()
      }
    )
}(OverflowStrategy.Unbounded)

errorDetector.headL
            .runToFuture
            .map(e => assertThrows[BufferOverflowException](throw
                e.get))
}
```

In questo secondo scenario, se pieno, il buffer viene svuotato: non si verifica alcun errore, tuttavia si perde una parte dei dati emessi dalla prima sorgente, a causa delle ripetute cancellazioni degl'elementi.

```
...
val obsOfClearBuffer = sourceWithEmitFunction(
    OverflowStrategy.ClearBuffer(bufferCapacity)
)

val result = getElementsFromSource(obsOfClearBuffer)
```

```
checkCondition[List[Long]](list => list.size < rangeLimit, result)
}
```

Infine, viene esaminato anche il caso in cui non si utilizzi alcun *buffer* limitato per la gestione delle sorgenti: ciò implica che con una fonte di dati veloce, potrebbe verificarsi una *lack of memory* e l'intero processo potrebbe andare in *crash*.

```
...
val obsOfUnbounded =
    sourceWithEmitFunction(OverflowStrategy.Unbounded)
val result = getElementsFromSource(obsOfUnbounded)

checkCondition[List[Long]](list => list.size == rangeLimit, result)

}
```

2.3 Akka Streams

Con le stesse premesse che hanno portato alla nascita del paradigma FRP è venuto alla luce il progetto **Akka Streams**.

Akka è un *toolkit* della compagnia *Lightbend*, adottato da team di sviluppo per costruire applicazioni *cloud native* e *pipeline* di dati globalmente distribuite. *Akka* è noto ai più come una soluzione implementativa per l'*actor model*, un modello matematico di computazione concorrente che tratta gli *attori* come una primitiva universale per la programmazione distribuita.

Un attore è un'entità alla quale può essere chiesto anche di maneggiare *stream* di dati: invia e riceve serie di messaggi allo scopo di trasferire conoscenza (o dati) da un posto ad un altro. Tuttavia, ciò non sarebbe abbastanza, poichè ogni volta sarebbe necessario anche in questo caso avere a che fare con *buffers* o *mail boxes* limitati.

Per queste ragioni, il team di sviluppo ha deciso di fornire una soluzione a tutti questi problemi creando l'API *Akka Streams*, considerando anche il fatto che *Akka* è, inoltre, riconosciuto come uno dei membri fondatori della già citata iniziativa **Reactive Streams**. L'API *Akka Streams*, tuttavia, è completamente disaccoppiata dai contratti proposti dalle *Reactive Streams interfaces*: si focalizza infatti sulla formulazione delle trasformazioni operanti su *data streams*, e utilizza lo standard *Reactive* per passare i dati fra i diversi operatori.

2.3.1 Principi di design

Il credo di *Akka* è **favorire la trasparenza sulla "magia"** e lo scopo è distribuire API minimali e consistenti. Questo, concretamente, si traduce in alcune caratteristiche fondanti di *Akka Streams*:

- tutte le *feature* sono **esplicite**.
- elevata composizionalità: caratteristica ricercata dai *reactive framework*, le funzionalità derivano dalla combinazione di pezzi differenti.
- modello di dominio esaustivo per il processing di *bounded stream*.

Akka Streams provvede tutti gli strumenti necessari ad esprimere *streams* di qualsiasi topologia, gestendo tutti gli aspetti (*back-pressure*, *buffering*, *transformations*..) e qualsiasi cosa l'utente scelga di costruire è riutilizzabile in un contesto più ampio.

2.3.2 Concetti chiave

Akka Streams lega la sua essenza all'elaborazione e al trasferimento di sequenze di elementi utilizzando *bounded buffers*. Pertanto, è possibile esprimere una catena, o per utilizzare il gergo di *Akka Streams*, un grafo, di entità, ognuna delle quali esegue indipendentemente (e possibilmente concorrentemente) dalle altre, effettuando il *buffering* di un limitato numero di elementi ogni volta. Un **grafo** è quindi la descrizione della topologia di uno **stream**, inteso come un processo attivo che include il passaggio e la trasformazione dei dati. Un **elemento** è l'unità minima di uno *stream*: tutte le operazioni trasformano e trasferiscono elementi e le dimensioni dei *buffer* sono sempre espresse nei termini della loro cardinalità, indipendentemente dalla loro reale dimensione. Il nome comune di tutti i blocchi di costruzione dei grafi è **operatore**. Come già detto in precedenza, ne esistono di innumerevoli ed è inoltre possibile definirne di propri andando ad estendere *GraphStage*, un'astrazione che permette appunto di creare nuove tipologie di operatori non previsti da *Akka Streams* (con anche un numero a piacere di porte di entrata e di uscita) e nei termini della quale sono definiti anche operatori di base, come ad esempio *map* e *filter*. In *Akka Streams*, le *pipelines* possono essere espresse utilizzando alcune astrazioni principali:

- **Source**: un operatore con esattamente un *output*, che emette dati quando vi è una qualche altra entità pronta a riceverli.
- **Sink**: un operatore con esattamente un *input*, che richiede e accetta elementi e, qualora fosse necessario, può rallentare la sorgente in ingresso.
- **Flow**: un operatore con esattamente un *input* e un *output*, che connette *Sources* e *Sinks*, operando trasformazioni sugl'elementi che vi transitano attraverso.

La combinazione di tutti questi "pezzi elementari" compone un **RunnableGraph**, un *Flow* collegato ad una *Source* ed un *Sink* che può essere messo in esecuzione. Esattamente come per gli altri *framework* reattivi, vale il concetto per cui "nulla accade fino allo sottoscrizione": anche dopo aver costruito un'istanza di *RunnableGraph* con tutte le sue connessioni, nessun dato viene prodotto fino alla **materializzazione**, il processo di allocazione di tutte le risorse necessarie ad eseguire la computazione descritta dal grafo.

2.3.3 Working with Sources, Sinks and Flows

Sink e Sources

Volendo mettere a confronto le entità dell'ecosistema *Akka Streams* con i componenti principali del già trattato ambiente *Reactive Monix*, possiamo

paragonare *Source* e *Sink* ad *Observable* e *Consumer* rispettivamente.
Anche essi ammettono varie possibili modalità di costruzione; ecco alcuni esempi:

- Riprendendo l'analogia con *Reactive Monix* e ricordando il forte legame tra le collezioni e la "rappresentazione reattiva" che ne viene fornita mediante il concetto di *Observable*, consegue che sia immediato pensare di poter costruire una *source* partendo da un'*iterable*, e così è; utilizzando poi *Sink.seq* (analogo di *Consumer.toList* e *toListL*), quello che si ottiene è una *Future* che contiene la sequenza di elementi prodotta dallo *stream*. Vale la pena sottolineare che poichè gl'operatori sono *Immutable*, connetterli ritorna un nuovo operatore, invece di modificare un'istanza già esistente. Cosa analoga accade per la materializzazione di uno *stream*, che può avvenire multiple volte, con conseguente ricalcolo di un valore *fresh* ad ogni occasione.
- Il metodo *runWith* permette di agganciare una *Source* ad un *Sink* e costituisce una *shortcut* per l'espressione:

  $$toMat(sink)(Keep.right).run()$$

  Nello specifico, il *RunnableGraph* risultante dall'operazione di *bind* dei due *building block* fondamentali viene lanciato e ne viene materializzato il valore risultante. Ogni operatore può produrre un valore materializzato, ed è responsabilità dello sviluppatore combinarli in un nuovo tipo. *toMat* indica la volonà di trasformare il valore materializzato da *Source* e *Sink*, e *Keep.right* dice che d'interesse è solo l'elemento prodotto dall'ultimo dei due operatori.

```
val rangeStart = 1
val rangeLimit = 10_000
val source = Source(rangeStart to rangeLimit)
val materializedValue = source.runWith(Sink.seq)

val rangeSeq = awaitForResult(materializedValue)

rangeSeq shouldBe (rangeStart to rangeLimit)
```

In questo caso invece, si mostra come creare una *Source* che emette un singolo elemento, che viene materializzato da un *Sink* che restituisce il primo elemento emesso dallo stream:

```
val source = Source.single(0)
val materializedValue = source.runWith(Sink.head)
```

```
val zero = awaitForResult(materializedValue)

zero shouldBe 0
```

Più interessante è l'esempio seguente:

- *Akka Streams* implementa lo *standard Reactive Streams* per l'elaborazione asincrona di *stream* con *back pressure* non bloccante; pertanto, considerando anche l'introduzione di *Reactive Streams* in Java 9, offre in questo senso interoperabilità. Le due interfacce più importanti in *Reactive Streams* sono **Publisher** e **Subscriber**. Da ciò deriva la possibilità di costruire una *pipeline* a partire da un determinato *Publisher*, inteso come un *producer* di un numero potenzialmente illimitato di elementi sequenziati, e un *Subscriber* associato, che si connette al *publisher* e avvia le emissioni.

- Una *source* del genere emette elementi senza alcuna limitazione, pertanto un *Sink* chiamato a materializzare un tale flusso si trova a lavorare ad un ritmo non sostenibile ed effettua una *backpressure* che arresta la sorgente. *Akka Streams* implementa, infatti, un meccanismo di *backpressure* asincrono: l'utente non deve scrivere alcun codice aggiuntivo per gestire questo aspetto, è tutto provvisto dagl'operatori di *Akka Streams*. È possibile definire operatori con *overflow strategies*, come per *Monix*, che possono influenzare il comportamento dello *stream*. La modalità con cui opera la *backpressure* in *Akka Streams* può essere colloquialmente descritta come *dynamic push / pull mode*, poichè ibrida fra modello *push* e *pull* a seconda della capacità del *consumer* di sostenere la velocità di produzione.

- Utilizzando l'operatore *throttle* (analogo, come tanti altri, all'omonimo visto in *Reactive Monix*), possiamo regolare l'afflusso di dati e permettere al *Sink* di lavorare ad una velocità consona.

```
val rangeStart = 0
val rangeLimit = 10

val sourceFromPublisher = Source.fromPublisher((s: Subscriber[_ >:
    Int]) => {
for (i <- rangeStart until rangeLimit) {
    s.onNext(i)
}
s.onComplete()
})
```

```
//publisher doesn't backpressure source
assertThrows[IllegalStateException](
    awaitForResult(getElementsFromSource(sourceFromPublisher))
)

//timed-based flow control
val materializedValues = getElementsFromSource(
    sourceFromPublisher.throttle(1, 100.millis)
)
val rangeSeq = awaitForResult(materializedValues)

rangeSeq shouldBe (rangeStart until rangeLimit)
```

Streams come combinazione di Sources, Flows and Sinks

Una *Source* combinata con un *Sink* costituisce già, in realtà, un possibile *RunnableGraph*. A ciò si possono aggiungere uno o più *Flows* (aka *Pipes*) per creare *streams* più complessi e strutturati. Per un *Flow* si specifica il tipo di dato in ingresso e le operazioni che devono operare una trasformazione sui dati in transito. Nell'esempio sotto, si costruisce un grafo a partire da una sorgente che emette gli elementi di un *range*, passati ad un *flow*, da cui vengono elevati al quadrato, e materializzati da un *sink* che produce l'ultimo elemento dello *stream*. Quando si costruiscono flussi e grafi in *Akka Streams* quello che viene fatto è tipicamente tracciare un piano di esecuzione. La materializzazione vera e propria dello *stream* si verifica con l'avvio di attori che alimentano l'elaborazione ed è performata sincronicamente sul *thread* che materializza da un *ActorSystem global Materializer*.

```
implicit val testKit: ActorSystem[Nothing] = ActorTestKit().system
...
val startRange = 1
val endRange = 200

val source = Source[Int](startRange to endRange)
...
val squarePow = 2

val square = Flow[Int].map(v => Math.pow(v, squarePow).toInt)
val sink = Sink.last[Int]
```

```
val materializedValue =
    source.via(square).toMat(sink)(Keep.right).run()

val lastSquare = awaitForResult(materializedValue)

lastSquare shouldBe Math.pow(endRange, squarePow)
```

Poichè ogni operatore in *Akka Streams* può provvedere un valore materializzato, è necessario in qualche modo esprimere come questi valori dovrebbero essere composti in un valore finale quando vengono congiunti insieme. Per questo, molti operatori hanno varianti che prendono un argomento aggiuntivo, una funzione, *Keep*, che viene usata per combinare i valori risultanti.

```
val timedSource = Source.tick(0.seconds, 0.1.seconds, empty)
val sum = Flow[Int].fold(0)((acc, value) => acc+value)
                  .zipWith(timedSource)(Keep.left)
val sink = Sink.head[Int]

val materializedValue = source.viaMat(sum)(Keep.left)
                            .toMat(sink)(Keep.right).run()

val sumOfNumbersUntilEndRange = awaitForResult(materializedValue)

sumOfNumbersUntilEndRange shouldBe sumOfFirstNNumbers(endRange)
```

2.4 Working with Graphs

2.4.1 Costruire grafi

In *Akka Streams*, i grafi non sono espressi usando un DSL fluente come le computazioni lineari, ma in un DSL apposito il cui scopo è codificare i disegni dei grafi (dalle note prese in seguito a discussioni di design, o illustrazioni delle specifiche di un protocollo). I grafi sono costruiti da semplici *flow* che fungono da connessioni lineari e *junctions* che sono punti di entrata e uscita per i *flows*; di queste ultime, ne esistono di diversi tipi, a seconda del numero di uscite o di entrate o di tipi di concatenazione che sono in grado di supportare:

```
val sink = Sink.head[Int]

val start = 100
```

```
val graph = RunnableGraph.fromGraph(
GraphDSL.createGraph(sink){ implicit builder =>
    sink =>

    import GraphDSL.Implicits._

    val sourceA = Source[Int](0 to 100)
    val sourceB = Source[Int](start to 200)

    val flowA = Flow[Int].map(_ * 2)
    val stepTwo = Flow[Int].map(_ / 2)

    val zip = builder.add(ZipWith[Int, Int, Int](Keep.right))
    val outputPorts = 2
    val broadcastA = builder.add(Broadcast[Int](outputPorts))

    val flowC = Flow[Int].filter(_ % 2 == 0)

    val broadcastB = builder.add(Broadcast[Int](outputPorts))

    val stepThree = Flow[Int].map(_ * 5)
    val flowE = Flow[Int].map(_.toString)

    val zip2 = builder.add(ZipWith[Int, String, Int](Keep.left))
    val zip3 = builder.add(ZipWith[Int, Int, Int](Keep.right))

    sourceA ~> flowA  ~> zip.in0
    sourceB ~> stepTwo ~> zip.in1
    zip.out ~> broadcastA.in
               broadcastA.out(0) ~> broadcastB.in
               broadcastA.out(1) ~> stepThree ~> zip2.in0
               broadcastB.out(0) ~> flowC   ~> zip3.in0
               broadcastB.out(1) ~> flowE   ~> zip2.in1

                                               zip2.out ~> zip3.in1
                                               zip3.out ~> sink


ClosedShape
```

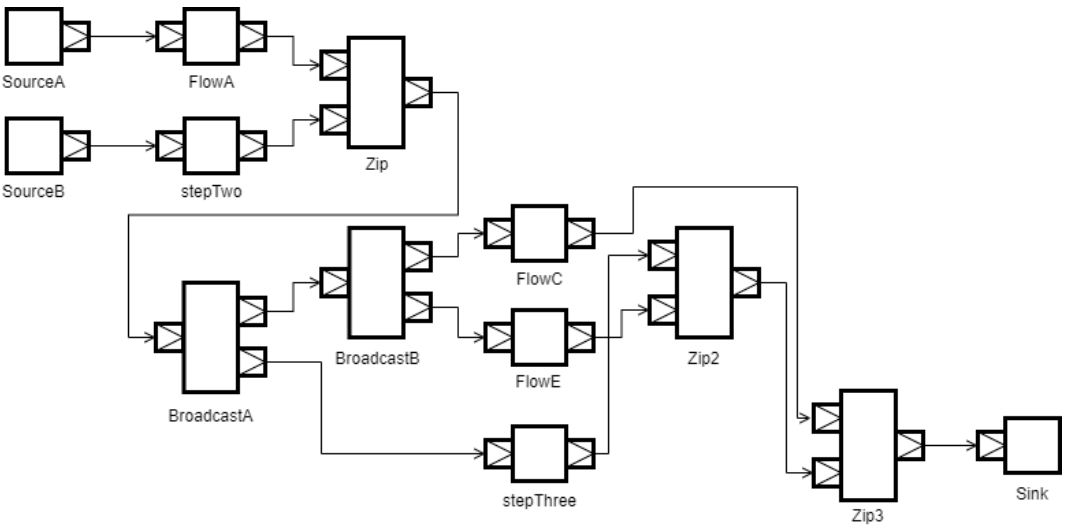


Figura 2.3: Questo è il grafo prodotto nell'esempio

Quelle che abbiamo chiamato *junctions* in questo esempio sono *zip* e *broadcast*: la prima unisce (come l'operatore *zip* di *Monix*) due input in un unico output, risultato dell'applicazione di una funzione passata al costruttore dell'operatore; la seconda prende il suo solo flusso d'input e lo duplica in un numero n di uscite (nel nostro esempio 2). Gli altri componenti del grafo sono costituiti da un paio di *source*, che avviano la computazione, e cinque flussi, di cui due sono nominati in base all'ordine di applicazione nel percorso compiuto dalla *pipeline*. Infine un *sink* materializza il primo elemento dello *stream*. Il costruttore fornito da *GraphDSL* prende come parametro un oggetto *GraphDSL.Builder* mutabile. Questo viene utilizzato (implicitamente) dall'*arrow operator*, fornito dal modulo *Implicits* di *GraphDSL*: il motivo di questa scelta di design è fornire la possibilità di creare in maniera più semplice grafi complessi, che possono anche contenere cicli. Una volta costruita però, l'istanza di *GraphDSL* è immutabile, *thread-safe* e liberamente condivisibile.

2.4.2 Costruire e combinare grafi parziali

Non è sempre possibile costruire l'intera computazione in un unico posto, ma può verificarsi la necessità di lavorare con differenti "parti più piccole" da assemblare e connettere poi in un unico grafo completo. C'è un aspetto infatti di cui non è stato spiegato alcun dettaglio nell'esempio precedente, il concetto di *Shape*. Un *RunnableGraph*, come quello costruito in precedenza, è tale dal momento che tutte le sue porte, sia in entrata che in uscita, sono connesse, ovvero tutti i suoi componenti vengono maneggiati nel rispetto delle loro caratteristiche da un punto di vista topografico: le *source* hanno il loro

output connesso, i *flow output* e *input*, l'unico *sink* il canale d'ingresso e le *junctions* le porte con le quali sono state create. Una tale *shape* è definita *ClosedShape* e permette di lanciare la computazione.

Tuttavia, come abbiamo anticipato, questa non è l'unica "forma" che può essere assegnata ad un grafo: ne esistino di vari tipi, dalle più semplici alle più complesse, con l'importante differenza, rispetto a *ClosedShape*, di portare alla costruzione di quelli che definiamo **grafi parziali**, topologie di *stream* che hanno ingressi e uscite per i quali sono stati designati dei canali non "bindati" ad ulteriori operatori che vadano a completare il percorso del flusso in modellazione. Questo, però, non significa esclusivamente che un grafo parziale sia costituito solamente da un reticolo di *flows* e *junctions*, come chiunque in un primo momento potrebbe pensare, ma può (ed è questo probabilmente il grande pregio di *Akka Streams*), essere rappresentato in una struttura più semplice, come risultato della composizione ancora una volta di *Source*, *Sink* e *Flow*, pensati non come *building block* semplici, ma, in questo caso, come grafi parziali.

Nello specifico:

- Una *Source* può essere pensata in questi termini come un grafo parziale con esattamente **un output**, con forma **SourceShape**.
- Un *Sink* è un grafo parziale con esattamente **un input**, con forma **SinkShape**
- Un *Flow* è un grafo parziale con **un input** e **un output**, con forma **FlowShape**.

La possibilità di nascondere grafi complessi in elementi semplici come *Sink*/*Flow*/*Source* permette di gestirli esattamente come semplici operatori di composizione.

Nell'esempio che segue si costruisce una *Source* composta da due *Source* iniziali, il cui output viene passato ad uno *Zip*, che esegue la somma dei valori in entrata e produce un *output* che viene girato ad un *Flow*, che restituisce il doppio del valore in ingresso ad un *Merge* (*junction* che esegue il *merge* di vari *stream*, emettendo elementi nell'ordine di arrivo). L'output di questa sorgente è costituito dall'output di quest'ultimo componente:

```
val complexSource = Source.fromGraph(GraphDSL.create(){
implicit builder =>
    import GraphDSL.Implicits._

    val upperBound = 100
    val n = Random.nextInt(upperBound)
    val evenNumber = 2 * n
    val oddNumber = 2 * n + 1

    val sourceA = Source.single(evenNumber)
    val sourceB = Source.single(oddNumber)

    val zip = builder.add(ZipWith[Int, Int, Double](_ + _))
    val flow = Flow[Double].map(_ * 2)
    val merge = builder.add(Merge[Double](1))

    sourceA ~>    zip.in0
    sourceB ~>    zip.in1
                  zip.out ~> flow ~> merge

    SourceShape(merge.out)
})
```

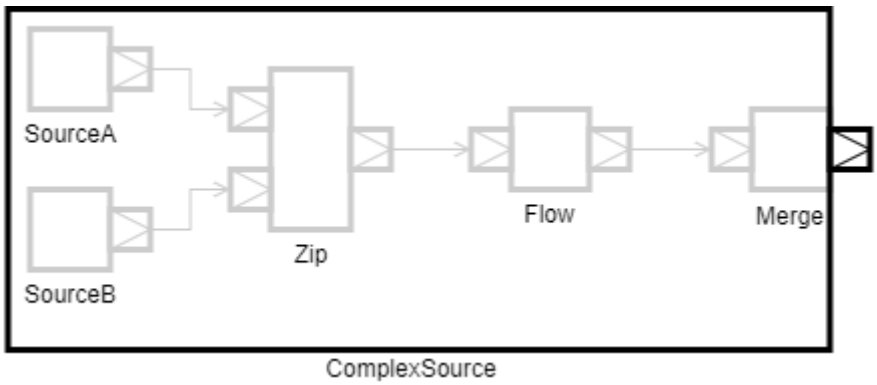


Figura 2.4: La *Source* composta che abbiamo creato

In quest'altro *snippet*, si elabora un *Flow* composto: l'input è costituito dal canale di ingresso di un *Broadcast* che "sdoppia" il flusso dello *stream* in due rami che passano a loro volta attraverso due *Flow*; il primo filtra solo i valori pari e li moltiplica per zero, il secondo raddoppia i valori in entrata. L'output è quello di uno *Zip* che raccoglie i valori dei due *Flow* e permette il passaggio

del minore dei due valori.

```
val complexFlow = Flow.fromGraph(GraphDSL.create(){ implicit builder
    =>
    import GraphDSL.Implicits._

    val outPorts = 2
    val broadcast = builder.add(Broadcast[Double](outPorts))
    val checkEvenCondition: Double => Boolean = _ % 2 == 0

    val flowA = Flow[Double].filter(checkEvenCondition).map(_ * 0)
    val flowB = Flow[Double].map(_ * 2)

    val zip = builder.add(ZipWith[Double, Double, Double](Math.min))

    broadcast.out(0) ~> flowA ~> zip.in0
    broadcast.out(1) ~> flowB ~> zip.in1

    FlowShape(broadcast.in, zip.out)
})
```

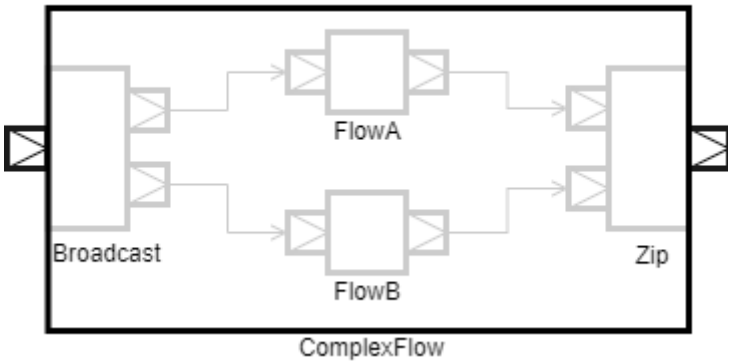
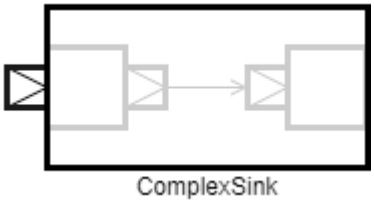


Figura 2.5: Il *Flow* composto

Definiamo anche un *Sink* composto, questa volta senza ricorrere al *GraphDSL*. Il valore prodotto è il risultato del valore in ingresso moltiplicato per un fattore randomico.

```
//fluid DSL
val complexSink = {
  val mult = Random.nextInt()
  Flow[Double].map(_ * mult)
    .toMat(Sink.head[Double])(Keep.right)
}
```



Nella costruzione del grafo finale accade proprio quello che è stato anticipato: i collegamenti fra i tre operatori vengono effettuati ignorando che dietro vi siano implementazioni più articolate, nascoste all'utilizzatore.

```
val materializedValue =
    complexSource.via(complexFlow).toMat(complexSink)(Keep.right).run()
```

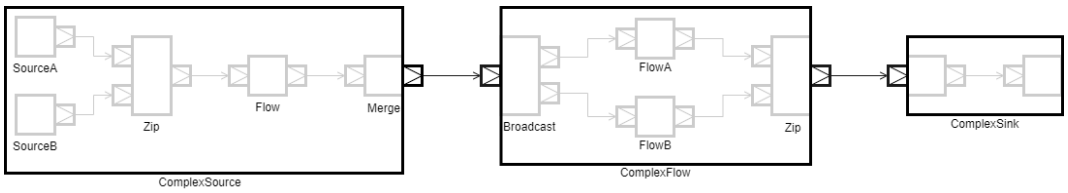


Figura 2.6: Questo è il grafo risultante

Ogni operatore usato in *Akka Streams*, a questo punto, può essere immaginato come un "box" con porte di input e output, in cui gli elementi processati arrivano e lasciano l'operatore. Questo fa si che questi componenti siano riutilizzabili per costruire reti di elaborazione dei dati complesse.

La costruzione di un grafo si riduce alla connessione delle porte esposte da queste "box", intese come architetture modulari, che nascondono la loro logica

interna all'utilizzatore: questo è esattamente il principio che *Akka* applica quando propone *Akka Http*, un modulo che implementa un *full stack HTTP client* e *server side* realizzato mediante *Akka Streams* e, in particolare, grazie a questi aspetti del *framework*.

2.4.3 Grafi con cicli

Un grafo che presenta un ciclo necessita di speciali considerazioni per evitare problemi di *deadlock*.

In questo esempio, si costruisce un grafo a partire da due *Flow* e un *Sink*: il primo dei due *Flow* (nominato *increment*) computa il successivo dell'elemento in ingresso), mentre il secondo, (*gate*), rappresenta la condizione per uscire dal ciclo. Il *Sink* consuma ed emette il primo elemento emesso dallo *stream*.

```
val sink = Sink.head[A]
...
RunnableGraph.fromGraph(
GraphDSL.createGraph(f1, f2, sink)((_, _, s) => s){ implicit builder
    =>
      (add, gate, s) =>
        import GraphDSL.Implicits._

        val source = Source.single(start)

        val broad = builder.add(Broadcast[A](2))
        val merge = builder.add(Merge[A](2))

        source ~> merge.in(0)
                  merge.out           ~>   broad.in
                                           broad.out(0) ~> gate ~> s
                  merge.in(1) <~ add <~    broad.out(1)

        ClosedShape
    })
```

La sorgente iniziale emette un singolo elemento che viene passato ad una delle entrate di *Merge*, che fornisce il suo output alla *junction Broadcast* che, duplica le proprie emissioni, inviandole al *Flow* che filtra gli elementi e li porta al *Sink*, e al *Flow* che effettua l'incremento e ritorna al *merge* formando un ciclo.

Questo grafo tuttavia non emette elementi: ogni volta che si compie il *loop*,

Merge effettua il suo compito adducendo al *feedback* ritornatogli dal *Flow*, che effettua l'incremento, anche l'elemento singolo della sorgente, andando ogni volta a replicare quest'ultimo. Ciò fa si che ogni iterazione vada ad incrementare il numero degl'elementi dello stream, che vanno progressivamente a saturare i *buffer* interni degl'operatori, costringendo questi alla *backpressure* della sorgente, che diviene poi permanentemente inattiva (condizione di *Deadlock*).

Un ciclo tale risulta potenzialmente sbilanciato, poichè il numero degl'elementi circolanti è illimitato: un approccio risolutivo per questo tipo di problema consiste nel *drop* degl'elementi sul ramo che riporta i dati indietro al *Merge*. Si utilizza l'operatore *buffer* (simile ad altri omonimi di altri *framework* reattivi) e una strategia di *drop* degl'elementi in eccesso, mirata a ridurne il numero, in questo caso *dropHead*, che libera spazio eliminando ogni volta il più vecchio fra i memorizzati. Questo risolve il problema, ma la natura sbilanciata del grafo permane e per alcuni domini applicativi può non essere la scelta ideale (come si potrà osservare nell'implementazione *Akka* del *Game of Life*).

```
increment.buffer(bufferDim, OverflowStrategy.dropHead)
```

Capitolo 3

Game of Life

Una mini-applicazione che si è prestata bene come simulazione di svilppo mediante *FRP* è il *John Conway's Game of Life*. Non si tratta di un vero e proprio gioco, ma di un automa cellulare sviluppato dal matematico inglese John Horton Conway nel 1970. Il suo scopo è quello di mostrare come comportamenti simili alla vita possano emergere da regole semplici e interazioni a molti corpi, principio che è alla base dell'ecobiologia, la quale si rifà alla teoria della complessità. Dal punto di vista teorico è interessante perchè ha la potenzialità di una **Macchina di Turing universale**: ogni cosa che può essere elaborata algoritmicamente può essere espressa nel contesto di *Game of Life*, rendendolo di fatto Turing equivalente.

3.1 Descrizione

Si tratta di un gioco senza giocatori: la sua evoluzione è determinata dal suo stato iniziale, senza necessità di alcun input da parte di giocatori umani. Si svolge su una griglia di caselle quadrate (celle) che, ipoteticamente, si estende all'infinito in tutte le direzioni; questa è detta **mondo**. Ogni cella ha 8 vicini, che sono le celle ad essa adiacenti, includendo quelle in senso diagonale. Ogni cella può trovarsi in due stati: **viva** o **morta** (o accesa e spenta, on e off). Lo stato della griglia evolve in intervalli di tempo discreti, cioè scanditi in maniera netta. Gli stati di tutte le celle in un dato istante sono usati per calcolare lo stato delle celle all'istante successivo. Tutte le celle del mondo vengono quindi aggiornate nel passaggio da un istante a quello successivo: passa così una **generazione**.

Le disposizioni che regolano il *Game of Life* sono:

- qualsiasi cella viva con meno di due vicini vivi muore, come per **sottopopolazione**.
- qualsiasi cella viva con due o tre vicini vivi **sopravvive** alla generazione successiva.
- qualsiasi cella viva con più di tre vicini vivi muore, come per **sovrappopolazione**.
- qualsiasi cella morta con esattamente tre vicini vivi diventa viva, come per effetto di **riproduzione**.

3.2 Monix Reactive Version

La versione realizzata di questo tipo di applicativo, utilizzando *Reactive Monix*, ha un' "ossatura" **MVC**:

- Il *Model* è descritto utilizzando Algebric Data Type, definendo in un modulo a parte la logica applicativa.
- Il *Controller* implementa un "*Game Loop* reattivo" e coordina *Model* e *View* fungendo da *Adapter* per le due entità che non comunicano direttamente fra loro.
- La *View* è modellata con l'avallo dell'astrazione *Task* fornita da *Monix* per la gestione dei *Side Effect*

3.2.1 View

L'utente può interagire con l'applicazione decidendo, per ogni iterazione del gioco, quali celle siano vive o meno premendo i pulsanti della griglia e avviando la computazione del ciclo di generazioni con il tasto *Start-Stop*.

Dal punto di vista di progettazione dei flussi di dati, occorre, pertanto, modellare questi aspetti come delle sorgenti:

- Ogni pulsante è modellato come un *observable*. In particolare, viene costruita una sorgente di base a partire dal *lifting*, in questa versione reattiva, delle celle alla quale ne è concatenata un'altra che emette eventi alla pressione. La sequenza di tutti i pulsanti costituisce infine un unico *observable*.

```
private lazy val tilesInput: Observable[Seq[Tile]] =
```

```
Observable.combineLatestList(
tiles.map(tile => tile.liftToObservable ++
    onPressedAsObservable(tile)
) : _*)
```

La sorgente per ogni pulsante dedicata agl'eventi di pressione da parte dell'utente, ad ogni emissione colora o decolora il tasto di riferimento, eseguendo una *callback* asincrona registrata con l'operatore *doOnNext*, che prende appunto una funzione che genera un *Task* contenente una computazione da eseguire alla produzione dell'elemento successivo da parte della sorgente.

```
private def onPressedAsObservable(tile: Tile):
    Observable[Tile] =
tile.button.ActionEventAsObservable
            .doOnNext(_ => paintButton(tile.button))
            .map(_ => tile)
```

- La pressione del tasto *Start/Stop* è un evento che analogamente merita di essere osservato:

```
    private lazy val switchInput: Observable[ActionCommand] =
switch.ActionEventAsObservable
      .doOnNext(executeOnActionCommand)
      .map(_.getActionCommand)
```

Come per le celle, anche in questo caso viene registrata una *callback* in occasione della successiva emissione, orientata a cambiare il testo del pulsante (se *Start* in *Stop* o viceversa). Gli eventi vengono convertiti in comandi impartiti dall'utente (avvia la computazione oppure ferma il gioco).

Oltre a ciò, *observable* è anche il *text field* che mostra la generazione corrente: si costruisce a partire da un *observable* dello stato iniziale del campo concatenato ad una sorgente che produce un output in occasione del cambio di contenuto che si verifica con il progredire delle generazioni e la sovrascrittura del testo del *text field*.

```
private lazy val labelInput: Observable[Int] =
textField.getText
        .toInt
        .liftToObservable ++
           textField.onTextChangeAsObservable.map(_.toInt)
```

Questi tre *stream* di dati di input vengono combinati fra loro in un unico *observable*, che emette elementi continuamente, a partire dalla prima emissione di ogni fonte, andando a replicare, qualora non emergano nuovi valori, quelli prodotti per ultimi. Questo comportamento viene ottenuto grazie all'operatore *combineLatest*; oltre a ciò, il risultato dell'applicazione di questa unione viene convertito in unico elemento che esprime il comando impartito dall'utente legato alla computazione ciclica del *Game of Life*.

```
Observable.combineLatest3(switchInput, tilesInput,
    labelInput).map {
  case (cmd, tiles, gen) => CycleComputationRequest(cmd, tiles,
      gen)
}
```

3.2.2 Controller

Questo componente si prefigge di realizzare un *Game Loop* reattivo, come preannunciato:

- un *observable* che emette elementi entro un intervallo di tempo per cui si abbiano circa 60 aggiornamenti al secondo ne costituisce il motore.

  ```
  Observable.interval(TIME_INTERVAL).doOnStart(_ =>
      view.display)
  ```

- **processInput**: l'*observable* emesso dalla *view* è convertito in richieste che verranno elaborate dalla funzione che aggiorna il *Model*, tramite l'operatore *collect*:

  ```
  private def processInput: Observable[ModelInput] =
  view.emit.collect {
    case request: CycleComputationRequest => request.cmd match
      {
  ```

```
        case View.START =>
            UpdateRequest(GameOfLife.Generation(request.genNumber,
            request.tiles))
        case _ => StopRequest
      }
    }
```

- **updateModel**: il flusso così elaborato viene utilizzato per l'applicazione della logica di gioco e l'avanzamento delle generazioni: a partire dal comando impartito dall'utente (ovvero *Start* o *Stop*), si costruisce un nuovo flusso che, qualora l'ultimo input sia *Start*, emetta la generazione successiva a quella corrente, oppure, in caso di *Stop* ritorni la generazione attuale. Questo nuovo *stream* di eventi viene fatto confluire nel flusso principale in entrata, in maniera tale per cui l'intera costruzione sia reattiva a possibili nuove indicazioni impartite dall'utente con la pressione del pulsante di *Start* o *Stop*. Perfetto in questo senso si rivela l'operatore *SwitchMap*, che dà la precedenza al *most-recently-emitted-observable* e, quindi è in grado di produrre un aggiornamento reattivamente coerente con quanto richiesto dall'utente (se si preme *Stop* deve essere interrotto il susseguirsi delle generazioni immediatamente):

```
        private def updateModel(): Observable[GameOfLife] =
            processInput.switchMap(UpdateGameState(_))
```

- **render**: al flusso risultante è applicata, ancora una volta con l'operatore *doOnNext*, una *callBack* asincrona con cui la versione più aggiornata del *model* (costituita dalla generazione più recente) viene passata alla *view* per essere disegnata.

```
        proactiveLoop.doOnNext(model =>
            view.render(model)).completedL
```

Due sono le versioni prodotte del *Game Loop*:

- Una versione "**proattiva**", che disaccoppia il ritmo esecutivo del *Game Engine* dal ritmo con cui sono scandite le varie generazioni (una nuova ogni secondo, durante il normale svolgimento del gioco), ottenuta mediante l'applicazione dell'operatore *combineLatest* e dell'operatore *distinctUntilChanged*, che va a sopprimere i tanti elementi duplicati emessi dalla sorgente.

```
gameLoopEngine.combineLatestMap(updateModel())((_,
    updatedModel) => updatedModel)
              .distinctUntilChanged
```

- Una versione "**reattiva**", in cui l'andamento del ciclo è dettato dal progredire del gioco e dall'aggiornamento dello stato della griglia, prodotto dall'applicazione dell'operatore *zipMap*:

```
gameLoopEngine.zipMap(updateModel())((_, updatedModel) =>
    updatedModel)
```

3.2.3 Model

Game of Life è logicamente rappresentato dal seguente tipo di dato:

```
sealed trait GameOfLife {
  def world: Board
  def generationNumber: Int
}
```

Pertanto, applicare la logica di gioco significa ogni volta calcolare la generazione successiva a partire dalla generazione corrente e dallo stato dell'automa; anche tale processo è stato reso in maniera completamente reattiva:

- qualora il gioco sia "*Started*", vengono calcolati i nuovi stati delle celle a partire dalla loro configurazione corrente: per ognuna vengono individuati i vicini, si calcola il numero totale dei vicini in vita e si applicano le regole del gioco, grazie alle quali siamo in grado di dire se sarà attiva o meno nella prossima generazione. L'operatore *mergeMap* consente di andare a lavorare singolarmente su ogni cella, effettuare i calcoli necessari, e poi, mediante *bufferTumbling*, si compatta in un unica sequenza il risultato.

```
private def update(currentGeneration: GameOfLife):
    Observable[GameOfLife] = {
def countIfAlive(position: Position): Int =
    currentGeneration.world(position) match {
  case Live => 1
  case _ => 0
}
```

```
  val rows = GameOfLife.gridDimensions.rows
  val columns = GameOfLife.gridDimensions.columns

  Observable.fromIterable(currentGeneration.world)
    .mergeMap {
      case (position, status) =>
      Observable.fromIterable(
      getNeighboursPositions(position)(GameOfLife.gridDimensions))
          .foldLeft(0)((nAlive, neighbourPosition) => nAlive +
              countIfAlive(neighbourPosition))
          .map(nAliveNeighbours =>
              applyGameOfLifeRulesBy(nAliveNeighbours, status))
    }
    .bufferTumbling(rows * columns)
    .map(newStatus =>
        getNextGeneration(currentGeneration)(newStatus))
    .sample(GameOfLife.INTERVAL_BETWEEN_GENERATION)
}
```

- se il gioco è in stato di *Stop*, viene sospeso l'andamento delle generazioni e pertanto la successiva è sempre uguale alla precedente:

```
private def identity: Observable[GameOfLife] =
    Observable.empty[GameOfLife]
```

L'*observable empty* è un'*observable* vuoto, che non rappresenta alcun aggiornamento da applicare allo stato della griglia di gioco.

3.3 Akka Streams Version

La versione del gioco realizzata mediante *Akka Streams* è una versione *lite* rispetto a quella precedente. A differenza della versione *Monix*, non è stata realizzata una *View* analoga, ma si mostrano le varie generazioni su *consolle* e quella *seed* dell'automa, la configurazione iniziale, è generata randomicamente. Il gioco è stato concepito come un *RunnableGraph* composto dalle seguenti componenti:

- **loopEngine**: *Source* che scandisce il susseguirsi delle iterazioni, emettendo iterativamente un elemento ad ogni secondo

```
    val loopEngine = Source.repeat().throttle(1, 1.seconds)
```

- **printer**: *Sink* che effettua la stampa della generazione corrente. Concretamente, prepara le celle del *world* ordinandole e assegnando un simbolo "#" per le vive e un "." per quelle morte; infine, raggruppa per righe e produce l'output finale che mostra la griglia.

```
val printer = Sink.foreach[Generation](i => {
  def compareTwoPositions(pos1: Position, pos2: Position):
      Boolean = {
    def map2DPositionTo1DValue(position: Position): Int =
      position.row * gridDimension.rows + position.column

    map2DPositionTo1DValue(pos1) <
        map2DPositionTo1DValue(pos2)
  }

  println("GENERATION: " + i.generationNumber)

  Source(i.world.toList.sortWith((first, second) => (first,
      second) match {
    case ((pos1, _), (pos2, _)) => compareTwoPositions(pos1,
        pos2)
  })).map {
    case (_, status) if status == Live => "#"
    case _ => "."
  }.grouped(gridDimension.rows)
    .map(s => "".concat(s))
    .runForeach(println)
})
```

- **firstGenInjector**: *Source* che produce la prima generazione:

```
val firstGenInjector = Source.single(Generation(0,
    initialState))
```

- **doGeneration**: *Flow* che a partire dalla generazione corrente, computa quella successiva; la logica è la medesima già vista per la versione *Monix* e perfino il nome degl'operatori è più o meno analogo.

```
val doGeneration =
  Flow[Generation].flatMapConcat(previousGeneration =>
  Source(previousGeneration.world).flatMapConcat {
    case (cellPosition, cellStatus) =>
    Source(getNeighboursPositions(cellPosition)(gridDimension))
```

```
            .fold(0)((nOfAliveNeigh, neighbourPosition) =>
                nOfAliveNeigh + (if
                    (previousGeneration.world(neighbourPosition)
                    == Live) 1 else 0))
            .map(nOfAliveNeighbours =>
                applyGameOfLifeRulesBy(nOfAliveNeighbours,
                cellStatus))
        }
          .grouped(gridDimension.rows * gridDimension.columns)
          .map(getNextGeneration(previousGeneration)))
```

Le restanti componenti sono *junctions* utilizzate per costruire un grafo con un ciclo bilanciato: si sono già esaminati, infatti, quali sarebbero i problemi legati alla gestione di una topologia contenente cicli, e si sono già proposte delle soluzioni che includono il *drop* degl'elementi per evitare *Backpressure* permanente della sorgente e *Deadlock*.

Tuttavia, in questo caso, quella non sarebbe una via percorribile: droppare elementi altererebbe il regolare svolgimento del *Game of Life*.

Pertanto, la risoluzione del problema risiede nella topologia ideata:

```
loopEngine ~> zip.in0
              zip.out ~> broadcast ~> printer
              zip.in1 <~ concat <~ firstGenInjector
                         concat <~ doGeneration <~ broadcast
```

- La sorgente che emette la prima generazione è passata alla *junction Concat*, che effettua la concatenazione di uno o più *stream*;
- Questa emette l'output ad uno *Zip*, che riceve anche gli impulsi prodotti ogni secondo dal *loopEngine*.
- Lo *Zip* emette ad un *Broadcast*, che invia elementi al *printer*, che produce l'output visualizzabile, e al *Flow doGeneration* che computa la generazione successiva, il quale a sua volta dirige la sua uscita sul *Concat*.
- *Concat* non replica più la generazione di *firstGenInjector*, ma invia allo *Zip* solamente l'output prodotto da *doGeneration*, dando vita ad un ciclo bilanciato.

Capitolo 4

Conclusioni

Per concludere, sono soddisfatto di quanto imparato da questa esperienza. La Programmazione Reattiva mi ha affascinato da subito e ho voluto impegnarmi a fondo nel cercare di esplorare la maggior parte dei suoi aspetti in quanti più *framework* fosse possibile per il tempo a mia disposizione. Stessa cosa per Scala, che è stato un linguaggio che mi ha affascinato da subito per la sua potenza e per la sua grande capacità espressiva, e spero vivamente di avere la possibilità in futuro di continuare a lavorare con la programmazione funzionale per poter perfezionare ogni giorno le mie *skills* e la qualità del mio codice. Grazie a questo progetto ho avuto la possibilità di conoscere persone appartenenti alla community di Scala Italy, trovando un ambiente che mi ha fatto sentire a mio agio nonostante il mio status di *beginner*. Tutto ciò mi dà grande motivazione per il futuro e mi fa guardare fiducioso verso il mio avvenire.

4.1 Sviluppi futuri

Mi piacerebbe consolidare le conoscenze di *Akka Streams*, che rimane un progetto molto vasto e ricco di aspetti da testare e padroneggiare, ed esplorare anche altri orizzonti della FRP, come ad esempio *Monix Iterant* oppure *FS2*, altro progetto molto importante per questo tipo di approccio allo sviluppo di applicazioni basate su stream di dati e informazioni.