ALMA MATER STUDIORUM UNIVERSITY OF BOLOGNA
(Campus Cesena)

Functional Reactive Programming in Scala

Paradigmi di Programmazione e Sviluppo
Laurea Magistrale Ingegneria e Scienze Informatiche

Noureddine Slimani (noureddine.slimani@studio.unibo.it)

AA 2020-2021

Cesena

Table of Contents

Capitolo 1:

Programmazione reattiva funzionale

1. Introduzione

La programmazione reattiva sta vivendo un bel po' di clamore. Sono state sviluppate numerose librerie per praticamente tutti i principali linguaggi di programmazione che facilitano la programmazione reattiva. Tra le più importanti ci sono le librerie ReactiveX che esistono per Javascript, Python, Scala e poche altre.

Sebbene queste librerie siano ottime, non devono essere confuse con la programmazione reattiva funzionale. Come è affermato nella documentazione di ReactiveX stesso:

"It is sometimes called "functional reactive programming" but this is a misnomer. ReactiveX may be functional, and it may be reactive, but "functional reactive programming" is a different animal."

Il principale punto di differenza tra le librerie reattive come ReactiveX e Functional Reactive Programming è che queste librerie guardano principalmente agli eventi e non ai comportamenti. Gli eventi sono valori discreti emessi nel tempo, come i clic del mouse. I comportamenti sono valori continui che hanno sempre un valore corrente, come la posizione del mouse.

Un clic del mouse di per sé non ha un valore: è solo un evento che viene attivato ogni volta che l'utente fa clic da qualche parte. Una posizione del mouse, d'altra parte, ha sempre un valore corrente, ma non viene "attivato" a determinati punti nel tempo.

2. Demistificazione della programmazione reattiva funzionale

La programmazione reattiva ha guadagnato molta attenzione negli ultimi anni. Il manifesto reattivo[1] sostiene una progettazione software reattiva ed è stato firmato migliaia di volte. Librerie reattive come ReactiveX ora esistono praticamente per tutti i principali linguaggi di programmazione.

Con l'avvento di Big Data e Spark, anche la programmazione funzionale sta vivendo un vero e proprio clamore. Portando a ancora più confusione, la programmazione reattiva funzionale viene gettata nel mix e spesso viene utilizzata per descrivere librerie reattive come ReactiveX.

In questo capitolo voglio ottenere una sospensione di tutte le parole d'ordine. Quindi cos'è la programmazione funzionale, la programmazione reattiva e la programmazione reattiva funzionale, come si relazionano tra loro e perché dovrebbe interessarti?

1 Programmazione funzionale

La programmazione funzionale è tutta una questione di funzioni? In realtà si tratta di funzioni *pure*. Le funzioni pure sono funzioni senza effetti collaterali. In altre parole, non scrivono in un database, non cambiano il valore di nulla al di fuori del loro ambito e non stampano nulla.

Questo ha una conseguenza importante: le funzioni pure devono sempre restituire qualcosa. Una funzione che non cambia stato e non restituisce nulla sarebbe praticamente inutile.

Immagina questo:

```
def useless_function_times_two(x: Int): Unit = {
  val useless_result = x * 2
}
```

Ovviamente, questa funzione non fa nulla di utile. La funzione non restituisce il suo risultato e non modifica alcun valore esterno.

La trasparenza referenziale

Inoltre, le funzioni pure si basano solo sugli argomenti passati loro come argomento. Non accedono a nulla dal mondo esterno.

Questo ha un grande vantaggio: *la trasparenza referenziale*.

Quando si chiama una funzione pura con gli stessi argomenti, restituirà sempre lo stesso risultato.

Questo rende molto più facile ragionare sul codice. Se hai testato le tue singole funzioni e sei sicuro che funzionino come dovrebbero, puoi anche essere sicuro che quelle funzioni che interagiscono tra loro ti diano i risultati che ti aspetti.

Immagina la seguente funzione impura:

```
var value = 2
def value_times_two(): Int = value * 2 // Impure function - bad idea
```

Non c'è modo di prevedere cosa restituirà questa funzione se non sai:

a) che dipende da value e

b) qual è il valore di value.

Anche se questo esempio potrebbe essere un po' artificioso, probabilmente ogni programmatore esperto ha già affrontato qualcosa di simile e si è chiesto da dove diamine provenisse il loro valore di ritorno. La maggior parte delle funzioni è un po' più complicata di quella sopra e se hanno

dipendenze esterne, la modifica di queste dipendenze potrebbe portare a risultati completamente imprevisti.

La seguente è generalmente un'idea molto migliore:

```
1    def times_two(x: Int): Int = x * 2
```

Questo è l'unico modo giusto per scrivere funzioni e alcuni linguaggi di programmazione funzionale non ti permetteranno di scrivere funzioni impure. Che tu sia un programmatore funzionale hardcore o meno, penso che la trasparenza referenziale sia sempre un obiettivo da raggiungere.

Immutability

La concorrenza porta a un altro requisito per le funzioni pure: le variabili passate come input alle funzioni devono essere immutabili. Altrimenti un altro thread potrebbe modificare il valore di una variabile dopo averla passata alla funzione, interrompendo la trasparenza referenziale.

Per inciso, dal momento che vuoi essere in grado di passare l'output di una funzione come input a un'altra funzione, anche il risultato di ritorno di una funzione pura deve essere immutabile.

Alcuni linguaggi di programmazione funzionale arrivano al punto di non consentire a nulla di essere mutevole. Vuoi raddoppiare i valori di una lista? Passalo a una funzione e fallo restituire invece un nuovo elenco. Vuoi scorrere un elenco e hai bisogno di una variabile per tenere traccia di dove ti trovi nell'elenco? Usa invece la ricorsione[2].

Ciò presenta vantaggi considerevoli quando si ha a che fare con la concorrenza e i sistemi distribuiti che devono essere in grado di recuperare da un errore. Se non puoi mutare nulla, non devi preoccuparti delle condizioni di gara o dei deadlock. Inoltre, se si dispone di un sistema distribuito che esegue alcuni calcoli pesanti e uno dei nodi si guasta, è semplice ripristinarlo se si dispone ancora dei dati immutabili originali e si conoscono i calcoli da eseguire. (Apache Spark, un framework popolare per l'elaborazione di grandi quantità di dati, funziona esattamente in questo modo.)

2 Reactive Programming

La programmazione reattiva viene spesso spiegata con un'analogia con un foglio di calcolo: immagina una cella che calcola l'input di altre due celle. Una volta modificato uno degli input, anche la somma viene aggiornata. La cellula reagisce ai cambiamenti e si aggiorna.

Un modo per implementare un comportamento reattivo è con Futures (o Promises in altre lingue) che forniscono un callback. Si consideri ad esempio il seguente codice:

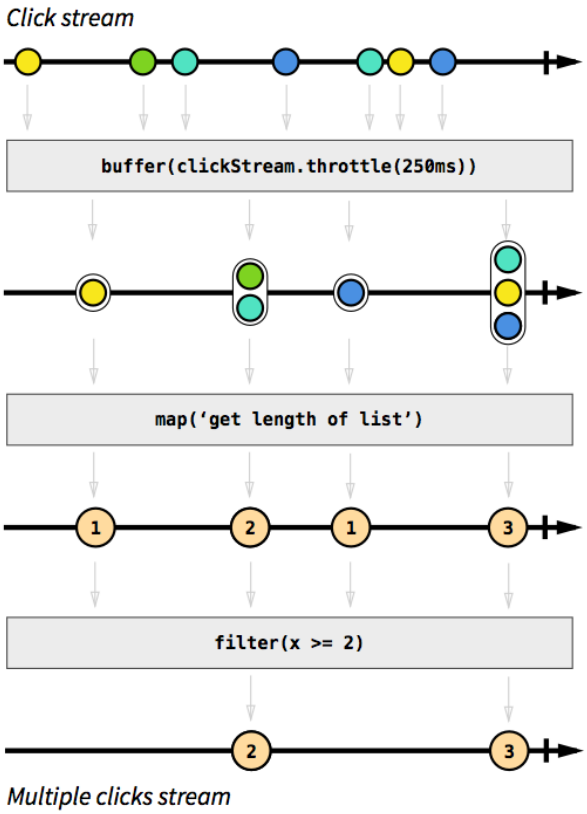
```scala
import scala.concurrent.{Future}
import scala.util.{Failure, Success}
import scala.concurrent.ExecutionContext.Implicits.global

val expensive_computation = Future{
  Thread.sleep( millis = 1000) // Simulate long-running computation
  2 * 2
}

expensive_computation.onComplete{
  case Success(value) => println(s"Result of expensive computation: $value")
  case Failure(e) => e.printStackTrace
}
```

Futures fornisce un'astrazione che consente di eseguire facilmente codice di lunga durata contemporaneamente e consente di gestire i risultati come se fossero già disponibili. Una volta completata la parte di codice di lunga durata, viene eseguita la richiamata fornita, che reagisce al completamento.

Sebbene i Futures siano un ottimo modo per gestire attività semplici e una tantum che devono essere eseguite contemporaneamente, possono diventare disordinate se vuoi fare qualcosa di più complicato. Ad esempio, immagina di implementare un'applicazione per fogli di calcolo con Futures. Sebbene sia possibile combinare Futures in scala con for-comprensions e comporli in altri modi, le cose si metteranno presto in disordine.

A differenza di Futures, le librerie reattive più popolari come Rx visualizzano gli eventi come flussi di dati. (Sono anche chiamati "osservabili", ma ci atterremo a "flussi" perché mi sembra più intuitivo.) Ti permettono di trasformare quei flussi di eventi e incollare insieme le azioni risultanti. Mentre Futures può essere utilizzato una volta (il callback viene chiamato una volta completata l'esecuzione del codice), i flussi di eventi possono essere utilizzati per attivare un'azione ogni volta che si verifica un evento. Quindi i flussi di eventi sono simili a Futures, ma più potenti perché aggiungono la dimensione del tempo.

Facciamo un esempio per rendere le cose più chiare. Supponiamo che tu voglia attivare un evento ogni volta che un utente fa doppio clic. Visualizzare questo come un flusso appare come segue:



(Borrowed from Andre Staltz's highly recommended introduction to Reactive Programming)

Quando si visualizzano gli eventi di clic come flusso, è possibile utilizzare una semplice API dichiarativa per trasformare questo flusso in un altro flusso di eventi per i doppi clic. Ora iscriviti semplicemente a questo flusso con un callback che esegue qualunque cosa tu voglia eseguire quando un utente fa doppio clic.

Reactive Systems

Ora, come si collega tutto questo ai sistemi reattivi come descritto nel manifesto reattivo[1]? Il manifesto vuole che tu abbracci la programmazione reattiva e scriva solo codice in uno stile reattivo?

Non proprio. Il manifesto parla più di sistemi reattivi che di programmazione reattiva. Questo a volte si confonde e porta alla confusione.

I sistemi reattivi sono conformi a determinati principi di progettazione architettonica. Questi principi di progettazione hanno lo scopo di portare a sistemi che siano reattivi, scalabili e tolleranti ai guasti nonostante i crescenti requisiti di oggi.

Il mezzo principale per raggiungere questo scopo è il passaggio di messaggi. Mentre le applicazioni reattive (come nella programmazione reattiva) si concentrano sugli eventi, i sistemi reattivi si concentrano sui messaggi.

Il manifesto reattivo descrive questa distinzione come segue:

Un messaggio è un elemento di dati che viene inviato a una destinazione specifica. Un evento è un segnale emesso da un componente al raggiungimento di un determinato stato. In un sistema guidato dai messaggi, i destinatari indirizzabili attendono l'arrivo dei messaggi e reagiscono ad essi, altrimenti rimangono inattivi. In un sistema basato sugli eventi, i listener di notifica sono collegati alle origini degli eventi in modo tale da essere richiamati quando l'evento viene emesso. Ciò significa che un sistema basato su eventi si concentra su origini di eventi indirizzabili mentre un sistema basato su messaggi si concentra su destinatari indirizzabili.

Questa attenzione ai messaggi ti consente di ridimensionare facilmente e porta alla trasparenza della posizione. Quando i tuoi singoli pezzi di codice gestiscono solo i messaggi, non importa se invii questi messaggi alla stessa macchina o a una macchina dall'altra parte del mondo.

I sistemi reattivi possono essere costituiti da applicazioni reattive, ma non è necessario che lo siano. In genere potrebbe essere una buona idea utilizzare per impostazione predefinita uno stile di programmazione reattivo per i sistemi reattivi, ma è anche possibile conformarsi ai principi di progettazione dei sistemi reattivi senza di esso.

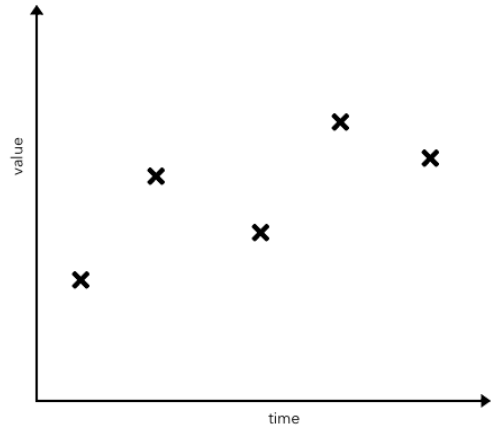
3 Functional Reactive Programming

Quindi, infine, come si collega questo alla programmazione reattiva funzionale? ReactiveX e librerie simili per la programmazione reattiva vengono talvolta etichettate come "Programmazione reattiva funzionale", ma in realtà questo non è del tutto corretto. ReactiveX è reattivo e utilizza molti elementi noti dalla programmazione funzionale come funzioni anonime e metodi come mappa, filtro e molti altri. Tuttavia, la programmazione reattiva funzionale è stata chiaramente definita come qualcos'altro. Come dice la documentazione di ReactiveX:

[ReactiveX] è talvolta chiamato "programmazione reattiva funzionale", ma questo è un termine improprio. ReactiveX può essere funzionale e può essere reattivo, ma la "programmazione reattiva funzionale" è un animale diverso. Un punto di differenza principale è che la programmazione reattiva funzionale opera su valori che cambiano continuamente nel tempo, mentre ReactiveX opera su valori discreti che vengono emessi nel tempo.

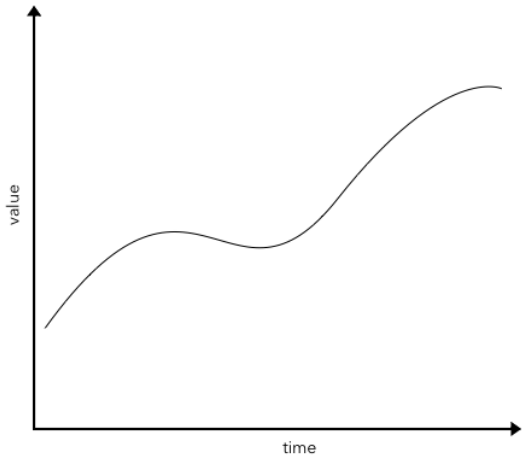
La programmazione reattiva funzionale è stata definita originariamente più di 20 anni fa da Conal Elliott e Paul Hudak nel loro articolo Animazione reattiva funzionale. Hanno fatto una distinzione tra "comportamenti" ed "eventi". Gli eventi sono fondamentalmente come i flussi che abbiamo visto in precedenza. Sono “valori discreti che vengono emessi nel tempo”.

Se li metti in un grafico, appariranno così:



Events

I comportamenti, invece, sono valori continui che cambiano nel tempo. Questa è una differenza sottile ma importante. Se inserisci un comportamento in un grafico, sarà simile a questo:



Behaviour

Un comportamento ha sempre un valore. Un evento ha solo un'ultima occorrenza (e forse un valore associato). Quindi, ad esempio, la posizione di un mouse è un comportamento perché puoi chiedere la sua posizione attuale. I clic del mouse, tuttavia, sono un flusso: non puoi chiedere il valore corrente di un clic del mouse, solo per la sua ultima occorrenza.

ReactiveX modella solo eventi discreti ma non valori continui. Allora come mai se la cava?

È possibile abbandonare del tutto la distinzione. Quando chiedi il valore di un evento, puoi semplicemente prendere il valore della sua ultima occorrenza. Quando si ha un comportamento continuo come un movimento del mouse, è possibile campionarlo e trattarlo come un flusso di eventi.

Tuttavia, questo può portare a una serie di problemi. Ad esempio, è possibile combinare flussi di eventi semplicemente aggiungendo gli eventi da un flusso all'altro, ma questo non ha senso per i valori continui. (Forse puoi riassumere i valori di due comportamenti se questo ha senso nella tua applicazione, ma è una cosa diversa dall'aggiungere gli eventi di un flusso alla sequenza temporale dell'altro flusso.) Se non fai una distinzione nel flusso API, non c'è modo per l'API di impedirti di fare qualcosa di stupido.

Per tornare al nostro esempio originale di un foglio di lavoro con una cella che riassume i valori di altre due celle: I valori delle celle sono comportamenti. Hanno sempre un valore corrente e il valore della cella che calcola la somma dipende dai valori delle altre due celle. Tuttavia, se si modifica il valore di una cella, questa modifica è un evento.

Se ora volessi scrivere un'applicazione per fogli di lavoro, potresti procedere in due modi: puoi allegare un callback all'evento di modifica per aggiornare il valore della somma. Tuttavia, se un altro valore dipendesse da tale somma, sarà necessario allegare anche un'altra richiamata. Puoi vedere come questo può diventare ingombrante piuttosto veloce.

D'altra parte, potresti modellare i valori delle celle come comportamenti, dove il valore della somma è una funzione degli altri due valori da cui dipende. Questo sembra un approccio più naturale per risolvere questo problema che porterebbe a un codice più elegante.

Capitolo 2: Scala.Rx 0.4.1

Una libreria sperimentale per la programmazione reattiva funzionale in Scala

1. Motivazione

In questo capitolo tutti gli esempi sono già implementati su GitHub sotto il percorso *project>src>main>scala>tests>exampleX.scala* dove *exampleX.scala* indica il titolo dell'esempio che coincide con l'*object* di scala su github.

La prima motivazione perché scala Rx è utile? Per rispondere a questa domanda consideriamo il seguente esempio qui:

```
var a = 1; var b = 2
val c = a + b
println(c)  //3
a = 4
println(c)  //3
```

example01.scala

L'esempio considera 3 variabili *a*, *b* e *c* dove *c* dipende da *a* e *b*. Il risultato di *c* è 3. Dopo l'aggiornato di a=4, il risultato è ancora uguale a 3. All'inizio, quando si calcola a+b, questo non significa a+b solo in quel momento, ma significa lo voglio per sempre indipendentemente ai nuovi valori di a.

Le variabili non sono sincronizzate tra di loro. Una soluzione per risolvere questo problema è convertire *c* da val a def. Questo ci dà una sorta di comportamento corretto. Ora il risultato di *c* prende il risultato aspettabile 6.

```
var a = 1; var b = 2
def c: Int = a + b
println(c)  //3
a = 4
println(c)  //6
```

example02.scala

Ma questo ha qualche limitazione, ad esempio se *c* non è a+b, se *c* è un'operazione molto costosa di *a* e *b*?

Ora ogni volta che chiediamo il valore di *c*, l'operazione costosa verrà eseguita anche se il valore di *a* e *b* non cambia. Quindi questo non è l'ideale.

```
var a = 1; var b = 2
def c = VeryExpensiceOperation(a, b)
println(c)  //3
a = 4
println(c)  //6
```

C'è un altro problema più difficile da risolvere, ed è quello che succede se vogliamo che *c* sia a+b ora e per sempre e avere una sorta di ascoltatore *onchange* da attivare se *c* cambia e ogni volta *c* cambia voglio fare qualcosa: voglio rendere un'interfaccia utente, voglio parlare con I server web remoti.

```
var a = 1; var b = 2
def c: Int = a + b
// OnChange(c, ()-->...)
a = 4
```

Non puoi davvero farlo in Scala usando *def* perché una definizione di una funzione come questa è completamente opaca. Ad ogni cambiamento il sistema continua ad eseguire questa funzione. ripetendo quest'operazione frequentemente, il sistema introduce una nuova complessità per qualcosa che dovrebbe essere relativamente integrata per natura nel sistema.

Quindi cosa ti offre Scala RX?

Scala.Rx è una libreria sperimentale di propagazione del cambiamento per Scala. Scala.Rx fornisce le variabili reattive (Rxs)[3], che sono variabili intelligenti che si aggiornano automaticamente quando i valori da cui dipendono cambiano. L'implementazione sottostante è un FRP [4] basato su push basato sulle idee in Deprecating the Observer Pattern[5].

Un semplice esempio che dimostra il comportamento è:

```
import rx._

val a = Var(1); val b = Var(2)
val c = Rx{ a() + b() }
println(c.now) // 3
a() = 4
println(c.now) // 6
```

example03.scala

L'idea è che il 99% delle volte, quando ricalcoli una variabile, la ricalcoli nello stesso modo in cui l'hai calcolata inizialmente. Inoltre, lo si ricalcola solo quando uno dei valori da cui dipende cambia. Scala.Rx lo fa automaticamente per te e gestisce tutta la noiosa logica di aggiornamento per consentirti di concentrarti su altre cose più interessanti!

Oltre alla propagazione delle modifiche di base, Scala.Rx fornisce una serie di altre funzionalità, come un set di combinatori per costruire facilmente il grafico del flusso di dati, controlli del tempo di compilazione per un alto grado di correttezza e interoperabilità senza soluzione di continuità con

il codice Scala esistente. Ciò significa che può essere facilmente incorporato in un'applicazione Scala esistente.

2. Inizio

Scala.Rx è disponibile su Maven Central[6]. Per iniziare, aggiungi semplicemente quanto segue al tuo *build.sbt*:

```
libraryDependencies += "com.lihaoyi" %% "scalarx" % "0.4.1"
```

Successivamente, aprire la console sbt e incollare l'esempio sopra nella console dovrebbe funzionare.

3. ScalaJS

Oltre a funzionare sulla JVM, Scala.Rx compila anche su Scala-Js[7]. Questo artefatto è attualmente su Maven Central e può essere utilizzato tramite il seguente frammento SBT:

```
libraryDependencies += "com.lihaoyi" %%% "scalarx" % "0.4.1"
```

Ci sono alcune piccole differenze tra l'esecuzione di Scala.Rx sulla JVM e in Javascript, in particolare per quanto riguarda le operazioni asincrone, il modello di parallelismo e il modello di memoria. In generale, tuttavia, tutti gli esempi forniti nella documentazione di seguito funzioneranno perfettamente se compilati in modo incrociato in javascript ed eseguiti nel browser.

Scala.rx 0.4.1 è compatibile solo con ScalaJS 0.6.5+.

4. Utilizzo di Scala.Rx

Le operazioni primarie richiedono solo *import rx._* prima di essere utilizzate, con operazioni aggiuntive che richiedono anche *import rx.ops._*. Alcuni degli esempi seguenti utilizzano anche varie importazioni da *scala.concurrent* o *scalatest*.

1 Utilizzo di base

```
import rx._

val a = Var(1); val b = Var(2)
val c = Rx{ a() + b() }
println(c.now) // 3
a() = 4
println(c.now) // 6
obs(c){.... do something .....}
```

L'esempio sopra è un programma eseguibile. In generale, *import rx._* è sufficiente per iniziare con Scala.Rx e verrà assunto in tutti gli altri esempi.

Le entità di base di cui devi preoccuparti sono Var[8], Rx e Obs[9]:

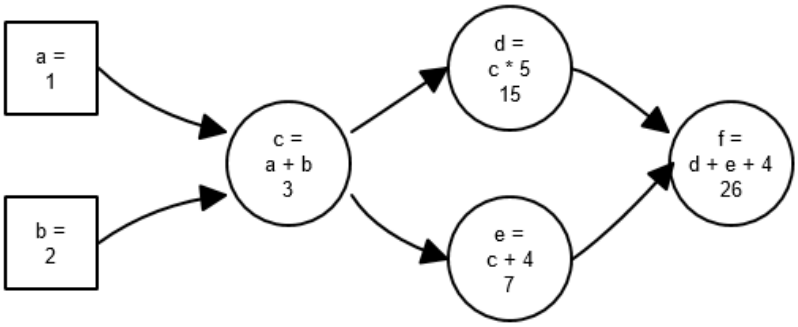
- *Var*: una variabile intelligente che puoi ottenere usando *a()* e impostare usando *a() = ...*. Ogni volta che il suo valore cambia, esegue il ping di qualsiasi entità a valle che deve essere ricalcolata.
- *Rx*: una definizione reattiva che cattura automaticamente qualsiasi *Var* o altro *Rx* che viene chiamato nel suo corpo, contrassegnandoli come dipendenze e ricalcolandoli ogni volta che uno di essi cambia. Come un *Var*, puoi usare la sintassi *a()* per recuperare il suo valore, ed esegue anche il ping delle entità a valle quando il valore cambia.
- *Obs*: un osservatore su uno o più *Var* s o *Rx* s, che esegue qualche effetto collaterale quando il nodo osservato cambia valore e gli invia un ping.

Utilizzando questi componenti, è possibile costruire facilmente un grafico del flusso di dati e mantenere aggiornati i vari valori all'interno del grafico del flusso di dati quando gli input al grafico cambiano:

```
val a = Var(1) // 1   3

val b = Var(2) // 2

val c = Rx{ a() + b() } // 3
val d = Rx{ c() * 5 } // 15
val e = Rx{ c() + 4 } // 7
val f = Rx{ d() + e() + 4 } // 26

println(f.now) // 26
a() = 3
println(f.now) // 38
```

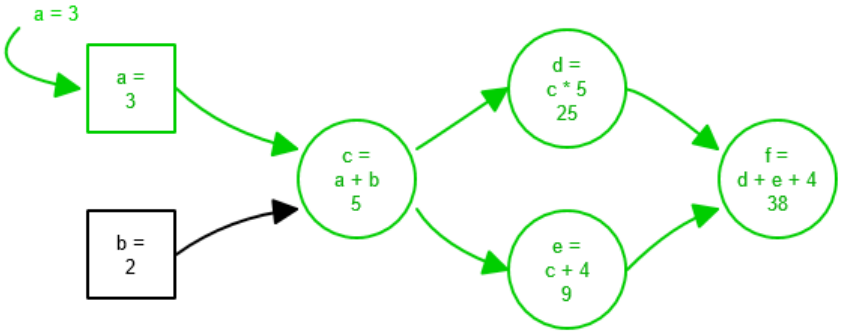
example04.scala

Il grafico del flusso di dati per questo programma ha il seguente aspetto:



Dove le *Var* sono rappresentate da quadrati, le *Rx* da cerchi e le dipendenze da frecce. Ogni *Rx* è etichettato con il suo nome, il suo corpo e il suo valore.

La modifica del valore di *a* fa sì che le modifiche si propaghino attraverso il grafico del flusso di dati



Come si può vedere sopra, la modifica del valore di *a* fa sì che la modifica si propaghi attraverso *c* *d* *e* fino a *f*. Puoi usare *Var* e *Rx* ovunque tu usi una variabile normale.

Le modifiche si propagano attraverso il grafico del flusso di dati in *onde*. Ogni aggiornamento di un *Var* innesca una propagazione, che spinge le modifiche da quel *Var* a qualsiasi *Rx* che è (direttamente o indirettamente) dipendente dal suo valore. Nel processo, è possibile ricalcolare una *Rx* più di una volta.

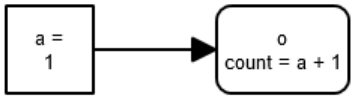
Observers

Come accennato, gli *Obs* s possono essere creati da *Rx* s o *Var* s ed essere utilizzati per eseguire effetti collaterali quando cambiano:

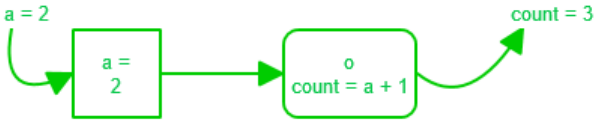
```
val a = Var(1)
var count = 0
val o = a.trigger {
  count = a.now + 1
}
println(count) // 2
a() = 4
println(count) // 5
```

example05.scala

Questo crea un grafico del flusso di dati che assomiglia a:



Quando *a* viene modificato, l'osservatore *o* eseguirà l'effetto collaterale:



Il corpo di *Rxs* dovrebbe essere privo di effetti collaterali, poiché possono essere eseguiti più di una volta per propagazione. Dovresti usare *Obs* s per eseguire i tuoi effetti collaterali, poiché è garantito che vengano eseguiti solo una volta per propagazione dopo che i valori per tutti gli *Rx* si sono stabilizzati.

Scala.Rx fornisce un comodo combinatore *.foreach()*, che fornisce un modo alternativo per creare un *Obs* da un *Rx*:

```
val a = Var(1)
var count = 0
val o = a.foreach{ x =>
  count = x + 1
}
println(count) // 2
a() = 4
println(count) // 5
```

example06.scala

Questo esempio fa la stessa cosa del codice sopra.

Nota che il corpo degli *Obs* viene eseguito una volta inizialmente quando viene dichiarato. Ciò corrisponde al modo in cui ogni *Rx* viene calcolato una volta quando viene dichiarato inizialmente. ma è ipotizzabile che si voglia un *Obs* che si attivi per la prima volta solo quando l'*Rx* che sta ascoltando cambia. Puoi farlo usando la sintassi alternativa *triggerLater*:

```
val a = Var(1)
var count = 0
val o = a.triggerLater {
  count = count + 1
}
println(count) // 0
a() = 2
println(count) // 1
```

example07.scala

Un *Obs* agisce per incapsulare il callback che esegue. Possono essere passati in giro, archiviati in variabili, ecc. Quando *Obs* viene raccolto, il callback smetterà di attivarsi. Pertanto, un *Obs* dovrebbe essere memorizzato nell'oggetto che interessa: se il callback riguarda solo quell'oggetto, non importa quando l'*Obs* stesso viene raccolto spazzatura, poiché accadrà solo dopo che l'oggetto che lo tiene diventa irraggiungibile, nel qual caso i suoi effetti non possono essere osservati comunque. Un *Obs* può anche essere disattivato attivamente, se è necessaria una garanzia più forte:

```
val a = Var(1)
val b = Rx{ 2 * a() }
var target = 0
val o = b.trigger {
  target = b.now
}
println(target) // 2
a() = 2
println(target) // 4
o.kill()
a() = 3
println(target) // 4
```

example08.scala

Dopo aver chiamato manualmente *.kill()*, *Obs* non si attiva più. Oltre a *.kill()*ing *Obs*s, puoi anche uccidere *Rx*s, che impedisce ulteriori aggiornamenti.

In generale, Scala.Rx ruota attorno alla costruzione di grafici del flusso di dati che mantengono automaticamente le cose sincronizzate, con cui puoi interagire facilmente da un codice imperativo esterno. Ciò comporta l'utilizzo di:

- *Var*s come input al grafico del flusso di dati dal mondo imperativo

- *Rx*s come nodi intermedi nei grafici del flusso di dati
- *Obs*s come output dal grafico del flusso di dati nel mondo imperativo

Complex Reactives

Le Rx non sono limitate a *Int*s . *String*s , *Seq[Int]*s , *Seq[String]*s, qualsiasi cosa può andare all'interno di un *Rx*:

```
val a = Var(Seq(1, 2, 3))
val b = Var(3)
val c = Rx{ b() +: a() }
val d = Rx{ c().map("omg" * _) }
val e = Var("wtf")
val f = Rx{ (d() :+ e()).mkString }

println(f.now) // "omgomgomgomgomgomgomgomgomgwtf"
a() = Nil
println(f.now) // "omgomgomgwtf"
e() = "wtfbbq"
println(f.now) // "omgomgomgwtfbbq"
```

example09.scala

Error Handling

Poiché il corpo di un *Rx* può essere qualsiasi codice Scala arbitrario, può generare eccezioni. Propagare l'eccezione nello stack di chiamate non avrebbe molto senso, poiché il codice che valuta l'*Rx* probabilmente non ha il controllo del motivo per cui non è riuscito. Invece, qualsiasi eccezione viene rilevata dalla stessa *Rx* e memorizzata internamente come *Try*.

Questo può essere visto nel seguente unit test:

```
val a = Var(1)
val b = Rx{ 1 / a() }
println(b.now) // 1
println(b.toTry) // Success(1)
a() = 0
intercept[ArithmeticException]{
  b()
}
assert(b.toTry.isInstanceOf[Failure])
```

example10.scala

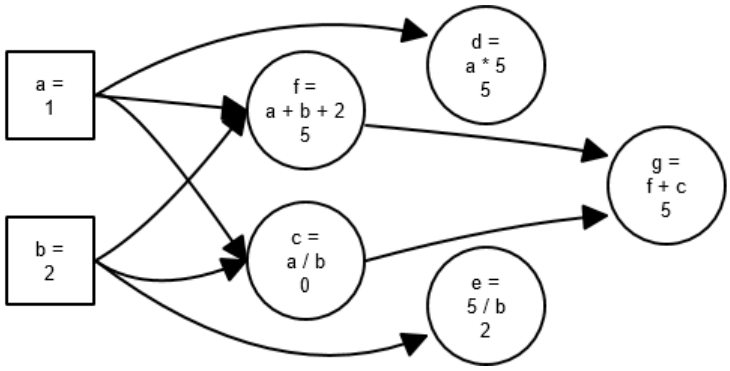
Inizialmente, il valore di *a* è *1* e quindi anche il valore di *b* è *1*. Puoi anche estrarre il *Try* interno usando *b.toTry*, che all'inizio è *Success(1)*.

Tuttavia, quando il valore di *a* diventa *0*, il corpo di *b* genera un'*ArithmeticException*. Questo viene catturato da *b* e rilanciato se si tenta di estrarre il valore da *b* usando *b()*. È possibile estrarre l'intero *Try* usando *toTry* e il pattern match su di esso per gestire sia il caso *Success* che il caso *Failure*.

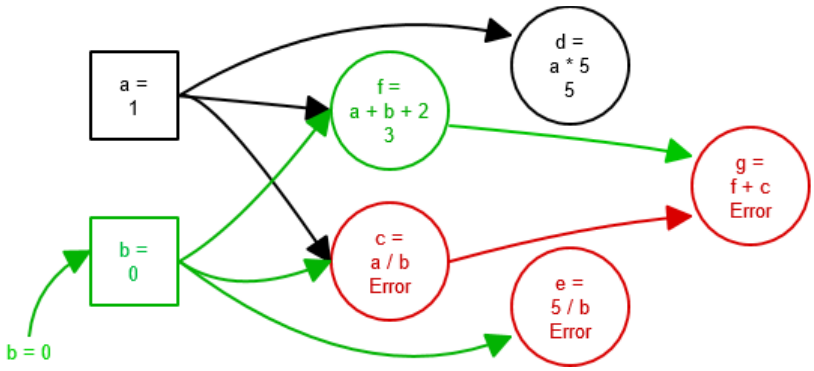
Quando si hanno molti *Rx* concatenati, le eccezioni si propagano in avanti seguendo il grafico delle dipendenze, come ci si aspetterebbe. Il seguente codice:

```
val c = Rx{ a() / b() }
val d = Rx{ a() * 5 }
val e = Rx{ 5 / b() }
val f = Rx{ a() + b() + 2 }
val g = Rx{ f() + c() }

inside(c.toTry){case Success(0) => () }
inside(d.toTry){case Success(5) => () }
inside(e.toTry){case Success(2) => () }
inside(f.toTry){case Success(5) => () }
inside(g.toTry){case Success(5) => () }

b() = 0

inside(c.toTry){case Failure(_) => () }
inside(d.toTry){case Success(5) => () }
inside(e.toTry){case Failure(_) => () }
inside(f.toTry){case Success(3) => () }
inside(g.toTry){case Failure(_) => () }
```

Crea un grafico delle dipendenze simile al seguente:



In questo esempio, inizialmente tutti i valori per a, b, c, d, e, f e g sono ben definiti. Tuttavia, quando b è impostato su 0:



c ed e risultano entrambi in eccezioni e l'eccezione da c si propaga a g. Il tentativo di estrarre il valore da g utilizzando *g.now*, ad esempio, genererà nuovamente l'ArithmeticException. Anche in questo caso, l'utilizzo di *toTry* funziona.

Nesting

L e *Rx* possono contenere altre *Rx*, arbitrariamente in profondità. Questo esempio mostra gli *Rx* annidati a due livelli di profondità:

```
val a = Var(1)
val b = Rx{
  (Rx{ a() }, Rx{ math.random })
}
val r = b.now._2.now
a() = 2
println(b.now._2.now) // r
```

example11.scala

In questo esempio, possiamo vedere che, sebbene abbiamo modificato *a*, questo riguarda solo l'*Rx* interno sinistro, né l'*Rx* interno destro (che assume un valore diverso e casuale ogni volta che viene ricalcolato) o l'*Rx* esterno (che farebbe ricalcolare l'intera cosa) sono interessati. Un esempio un po' meno forzato potrebbe essere:

```scala
var fakeTime = 123
trait WebPage{
  def fTime = fakeTime
  val time = Var(fTime)
  def update(): Unit  = time() = fTime
  val html: Rx[String]
}
class HomePage(implicit ctx: Ctx.Owner) extends WebPage {
  val html = Rx{"Home Page! time: " + time()}
}
class AboutPage(implicit ctx: Ctx.Owner) extends WebPage {
  val html = Rx{"About Me, time: " + time()}
}

val url = Var("www.mysite.com/home")
val page = Rx{
  url() match{
    case "www.mysite.com/home" => new HomePage()
    case "www.mysite.com/about" => new AboutPage()
  }
}

println(page.now.html.now) // "Home Page! time: 123"

fakeTime = 234
page.now.update()
println(page.now.html.now) // "Home Page! time: 234"

fakeTime = 345
url() = "www.mysite.com/about"
println(page.now.html.now) // "About Me, time: 345"

fakeTime = 456
page.now.update()
println(page.now.html.now) // "About Me, time: 456"
```

example12.scala

In questo caso, definiamo una pagina web che ha un valore *html* (a *Rx[String]*). Tuttavia, a seconda dell'*URL*, potrebbe essere una *HomePage* o una *AboutPage*, quindi il nostro *page* oggetto è un *Rx[WebPage]*.

Avere una *Rx[WebPage]*, dove la *WebPage* ha una *Rx[String]* all'interno, sembra naturale e ovvio, e Scala.Rx ti permette di farlo in modo semplice e naturale. Questo tipo di situazione di oggetti all'interno degli oggetti si presenta in modo molto naturale quando si modella un problema in modo orientato agli oggetti. La capacità di Scala.Rx di gestire con grazia i corrispondenti Rx all'interno di Rxs gli consente di adattarsi con grazia a questo paradigma.

2 Ownership Context

Nell'ultimo esempio sopra, abbiamo dovuto introdurre il concetto di proprietà in cui viene utilizzato *Ctx.Owner*. Infatti, se tralasciamo *(implicit ctx: Ctx.Owner)*, otterremmo il seguente errore di compilazione:

error: This Rx might leak! Either explicitly mark it unsafe (Rx.unsafe) or ensure an implicit RxCtx is in scope!

```
        val html = Rx{"Home Page! time: " + time()}
```

Per capire la *ownership* è importante capire il problema che risolve: *leaks*. Ad esempio, considera questa leggera modifica al primo esempio:

```
var count = 0
val a = Var(1); val b = Var(2)
def mkRx(i: Int) = Rx.unsafe { count += 1; i + b() }
val c = Rx{
  val newRx = mkRx(a())
  newRx()
}
println(c.now, count) //(3,1)
```

example13.scala

In questa versione è stata aggiunta la funzione *mkRx*, ma per il resto il valore calcolato di *c* rimane invariato. E la modifica di *a* sembra comportarsi come previsto:

```
a() = 4
println(c.now, count) //(6,2)
```

example14.scala

Ma se modifichiamo *b* potremmo iniziare a notare qualcosa che non va:

```
b() = 3
println(c.now, count) //(7,5) -- 5??

(0 to 100).foreach { i => a() = i }
println(c.now, count) //(103,106)

b() = 4
println(c.now, count) //(104,211) -- 211!!!
```

example15.scala

In questo esempio, anche se *b* viene aggiornato solo poche volte, il valore del conteggio inizia a salire man mano che *a* viene modificato. Questo è *mkRx* che perde! Cioè, ogni volta che *c* viene ricalcolato, costruisce un *Rx* completamente nuovo che rimane e continua a valutare, anche dopo che non è più raggiungibile come dipendenza dai dati e dimenticato. Quindi, dopo aver eseguito quell'istruzione *(da 0 a 100).foreach*, ci sono oltre 100 *Rx* che si attivano tutti ogni volta che *b* viene modificato. Questo chiaramente non è desiderabile.

Tuttavia, aggiungendo un esplicito *owner* (e rimuovendo *unsafe*), possiamo correggere la perdita:

```
var count = 0
val a = Var(1); val b = Var(2)
def mkRx(i: Int)(implicit ctx: Ctx.Owner) = Rx { count += 1; i + b() }
val c = Rx{
  val newRx = mkRx(a())(...)
  newRx()
}
println(c.now,count) // (3,1)
a() = 4
println(c.now,count) // (6,2)
b() = 3
println(c.now,count) // (7,4)
(0 to 100).foreach { i => a() = i }
println(c.now,count) //(103,105)
b() = 4
println(c.now,count) //(104,107)
```

example16.scala

L'ownership corregge le perdite consentendo a un genitore *Rx* di tenere traccia della sua *Rx* nidificata "owned". Cioè ogni volta che un *Rx* ricacola, prima uccide tutte le sue dipendenze di proprietà, assicurandosi che non perdano. In questo esempio, *c* è il proprietario di tutti gli *Rx* creati in *mkRx* e li uccide automaticamente ogni volta che *c* ricalcola.

3 Data Context

Dato un *Rx* o un *Var* usando *()* (aka *apply*) scarta il valore corrente e si aggiunge come dipendenza a qualunque *Rx* che sta attualmente valutando. In alternativa, *.now* può essere utilizzato semplicemente per scartare il valore e saltare la dipendenza dai dati:

```
val a = Var(1); val b = Var(2)
val c = Rx{ a.now + b.now } //not a very useful `Rx`
println(c.now) // 3
a() = 4
println(c.now) // 3
b() = 5
println(c.now) // 3
```

example17.scala

Per comprendere la necessità di un *Data* context e in che modo i *Data* contexts differiscono dai *owner* contexts, si consideri il seguente esempio:

```
def foo()(implicit ctx: Ctx.Owner): Any = {
  val a = rx.Var(1)
  a()
  a
}

val x = rx.Rx{val y = foo()(...); y() = y() + 1; println("done!") }
```

example18.scala

Con il concetto di proprietà, se *a()* è autorizzato a creare una dipendenza dati dal suo proprietario, entrerebbe in una ricorsione infinita e esploderebbe lo stack! Invece, il codice sopra fornisce questo errore di compilazione:

```
<console>:17: error: No implicit Ctx.Data is available here!
           a()
```

Possiamo "correggere" l'errore consentendo esplicitamente le dipendenze dei dati (e vedere che lo stack esplode):

```
def foo()(implicit ctx: Ctx.Owner, data: Ctx.Data): Any = {
  val a = rx.Var(1)
  a()
  a
}
val x = rx.Rx{val y = foo()(...); y() = y() + 1; println("done!") }
...
at rx.Rx$Dynamic$Internal$$anonfun$calc$2.apply(Core.scala:180)
at scala.util.Try$.apply(Try.scala:192)
at rx.Rx$Dynamic$Internal$.calc(Core.scala:180)
at rx.Rx$Dynamic$Internal$.update(Core.scala:184)
at rx.Rx$.doRecalc(Core.scala:130)
at rx.Var.update(Core.scala:280)
at $anonfun$1.apply(<console>:15)
  at $anonfun$1.apply(<console>:15)
    at rx.Rx$Dynamic$Internal$$anonfun$calc$2.apply(Core.scala:180)
    at scala.util.Try$.apply(Try.scala:192)
    ...
```

example18.scala

Il *Data* context è il meccanismo che un *Rx* utilizza per decidere quando ricacolare. L'*ownership* risolve il problema delle perdite. Mescolare i due può portare a una ricorsione infinita: quando qualcosa è sia di proprietà che una dipendenza dai dati dello stesso genitore Rx.

Fortunatamente però è quasi sempre il caso che sia necessario solo l'uno o l'altro contesto. quando si tratta di grafici dinamici, è quasi sempre il caso che sia necessario solo il contesto di proprietà, ovvero le funzioni più spesso hanno la forma:

```
def f(...)(implicit ctx: Ctx.Owner) = Rx { ... }
```

Il *Data* context è necessario meno spesso ed è utile, ad esempio, nel caso in cui sia desiderabile ASCIUGARE del codice *Rx* ripetuto. Tale funzione avrebbe questa forma:

```
def f(...)(implicit data: Ctx.Data) = ...
```

Ciò consentirebbe di estrarre una certa dipendenza dai dati condivisi dal corpo di ogni *Rx* e nella funzione condivisa.

Suddividendo i concetti ortogonali di *ownership* e *data* dependencies , il problema della ricorsione infinita, come descritto sopra, è notevolmente limitato. Le dipendenze esplicite dei dati rendono anche più chiaro quando l'uso di un *Var* o *Rx* è inteso come una dipendenza dei dati e non solo una semplice lettura del valore corrente (ad esempio *.now*). Senza questa distinzione, è più facile introdurre dipendenze di dati "accidentali" inattese e non volute.

4 Operazioni aggiuntive

Oltre ai mattoni di base di *Var*/*Rx*/*Obs*, Scala.Rx fornisce anche un insieme di combinatori che ti permettono di trasformare facilmente i tuoi *Rx*; ciò consente al programmatore di evitare di riscrivere costantemente la logica per le modalità comuni di costruzione del grafo del flusso di dati. I cinque combinatori di base: *map()*, *flatMap()*, *filter()*, *reduce()* e *fold()* sono tutti modellati sulla libreria delle collezioni scala e forniscono un modo semplice per trasformare i valori che escono da un *Rx*.

Map

```
val a = Var(10)
val b = Rx{ a() + 2 }
val c = a.map(_*2)
val d = b.map(_+3)
println(c.now) // 20
println(d.now) // 15
a() = 1
println(c.now) // 2
println(d.now) // 6
```

example19.scala

map fa quello che ti aspetteresti, creando un nuovo *Rx* con il valore del vecchio *Rx* trasformato da qualche funzione. Ad esempio, *a.map(_*2)* è essenzialmente equivalente a *Rx{ a() * 2 }*, ma in qualche modo più comodo da scrivere.

FlatMap

```
val a = Var(10)
val b = Var(1)
val c = a.flatMap(a => Rx { a*b() })
println(c.now) // 10
b() = 2
println(c.now) // 20
```

example20.scala

flatMap è analogo a flatMap dalla libreria delle collezioni in quanto consente di unire *Rx* nidificati di tipo *Rx[Rx[_]]* in un singolo *Rx[_]*.

Questo in combinazione con il map combinator consente alla sintassi di comprensione di scala di funzionare con *Rx* e *Var* :

```
val a = Var(10)
val b = for {
  aa <- a
  bb <- Rx { a() + 5}
  cc <- Var(1).map(_*2)
} yield {
  aa + bb + cc
}
```

example21.scala

Filter

```
val a = Var(10)
val b = a.filter(_ > 5)
a() = 1
println(b.now) // 10
a() = 6
println(b.now) // 6
a() = 2
println(b.now) // 6
a() = 19
println(b.now) // 19
```

example22.scala

Il *filter* ignora le modifiche al valore della *Rx* che non superano il predicato.

Si noti che nessuno dei metodi di filtro è in grado di filtrare il primo valore iniziale di un Rx, poiché non esiste un valore "più vecchio" a cui ricorrere. Quindi questo:

```
val a = Var(2)
val b = a.filter(_ > 5)
println(b.now)
```

example23.scala

stamperà "2".

Reduce

```
val a = Var(1)
val b = a.reduce(_ * _)
a() = 2
println(b.now) // 2
a() = 3
println(b.now) // 6
a() = 4
println(b.now) // 24
```

example24.scala

L'operatore di *reduce* combina insieme i valori successivi di un *Rx*, a partire dal valore iniziale. Ogni modifica alla *Rx* originale viene abbinata al valore precedentemente memorizzato e diventa il nuovo valore della *Rx* ridotta.

Fold

```
val a = Var(1)
val b = a.fold(List.empty[Int])((acc,elem) => elem :: acc)
a() = 2
println(b.now) // List(2,1)
a() = 3
println(b.now) // List(3,2,1)
a() = 4
println(b.now) // List(4,3,2,1)
```

example25.scala

Fold consente l'accumulo in modo simile per ridurre, ma può accumularsi in un tipo diverso da quello della sorgente Rx.

Ciascuno di questi cinque combinatori ha una controparte nello spazio dei nomi *.all* che opera su *Try[T]* anziché su *T*, nel caso in cui sia necessaria la flessibilità aggiuntiva per gestire i *Failure* in qualche modo speciale.

5 Combinatori asincroni

Questi sono combinatori che fanno di più che semplicemente trasformare un valore da uno all'altro. Questi hanno effetti asincroni e possono modificare spontaneamente il grafico del flusso di dati e iniziare cicli di propagazione senza alcun trigger esterno. Sebbene ciò possa sembrare alquanto inquietante, la funzionalità fornita da questi combinatori è spesso necessaria e scrivere manualmente la logica attorno a qualcosa come Debouncing, ad esempio, è molto più soggetto a errori rispetto al semplice utilizzo dei combinatori forniti.

Nota che nessuno di questi combinatori sta facendo qualcosa che non può essere fatto tramite una combinazione di *Obs*s e *Vars*; incapsulano semplicemente i modelli comuni, risparmiandoti di scriverli manualmente più e più volte e riducendo il potenziale di bug.

Future

```scala
import scala.concurrent.Promise
import scala.concurrent.ExecutionContext.Implicits.global
import rx.async._

val p = Promise[Int]()
val a = p.future.toRx(10)
println(a.now) //10
p.success(5)
println(a.now) //5
```

example26.scala

Il combinatore *toRx* si applica solo ai *Future[_]*. Prende un valore iniziale, che sarà il valore della *Rx* fino al completamento del *Future*, a quel punto il valore diventerà il valore del *Future*.

Questo *async* può creare *Future*s tutte le volte necessarie. Questo esempio mostra la creazione di due *Future*s distinti:

```scala
import scala.concurrent.Promise
import scala.concurrent.ExecutionContext.Implicits.global
import rx.async._

var p = Promise[Int]()
val a = Var(1)

val b: Rx[Int] = Rx {
  val f =  p.future.toRx(10)
  f() + a()
}
println(b.now) //11
p.success(5)
println(b.now) //6

p = Promise[Int]()
a() = 2
println(b.now) //12

p.success(7)
println(b.now) //9
```

example27.scala

Il valore di *b()* si aggiorna come ci si aspetterebbe quando la serie di *Future*s viene completata (in questo caso, manualmente utilizzando *Promise*s).

Questo è utile se il grafico delle dipendenze contiene alcuni elementi asincroni. Ad esempio, potresti avere un *Rx* che dipende da un altro *Rx*, ma richiede una richiesta web asincrona per calcolare il suo valore finale. Con *async*, i risultati della richiesta web asincrona verranno automaticamente riportati nel grafico del flusso di dati al completamento di *Future*, avviando un'altra esecuzione di propagazione e aggiornando convenientemente il resto del grafico che dipende dal nuovo risultato.

Timer

```
import rx.async._
import rx.async.Platform._
import scala.concurrent.duration._

val t = Timer(100 millis)
var count = 0
val o = t.trigger {
  count = count + 1
}

println(count) // 3
println(count) // 8
println(count) // 13
```

example28.scala

Un *Timer[10]* è un *Rx* che genera eventi su base regolare. Nell'esempio sopra, l'utilizzo di println nella console mostra che il valore *t()* è aumentato nel tempo.

L'attività pianificata viene annullata automaticamente quando l'oggetto *Timer* diventa irraggiungibile, quindi può essere Garbage Collection. Ciò significa che non devi preoccuparti della gestione del ciclo di vita del *Timer*. D'altra parte, questo significa che il programmatore dovrebbe assicurarsi che il riferimento al *Timer* sia tenuto dallo stesso oggetto di quello che tiene qualsiasi *Rx* che lo ascolta. Ciò assicurerà che il momento esatto in cui il *Timer* viene raccolto immondizia non avrà importanza, poiché a quel punto l'oggetto che lo contiene (e qualsiasi *Rx* che potrebbe influenzare) sono entrambi irraggiungibili.

Delay

```
import rx.async._
import rx.async.Platform._
import scala.concurrent.duration._

val a = Var(10)
val b = a.delay(250 millis)

a() = 5
println(b.now) // 10
eventually{
  println(b.now) // 5
}

a() = 4
println(b.now) // 5
eventually{
  println(b.now) // 4
}
```

example29.scala

Il combinatore *delay(t)* crea una versione ritardata di un *Rx* il cui valore è in ritardo rispetto all'originale di una durata *t*. Quando la *Rx* cambia, la versione ritardata non cambierà fino a quando non sarà passato il ritardo *t*.

Questo esempio mostra il ritardo applicato a un *Var*, ma potrebbe essere facilmente applicato anche a un *Rx*.

Debounce

```scala
import rx.async._
import rx.async.Platform._
import scala.concurrent.duration._

val a = Var(10)
val b = a.debounce(200 millis)
a() = 5
println(b.now) // 5

a() = 2
println(b.now) // 5

eventually{
  println(b.now) // 2
}

a() = 1
println(b.now) // 2

eventually{
  println(b.now) // 1
}
```

example30.scala

Il combinatore *debounce(t)* crea una versione di un *Rx* che non si aggiornerà più di una volta ogni periodo di tempo *t*.

Se si verificano più aggiornamenti in un breve lasso di tempo (meno di *t* l'uno dall'altro), il primo aggiornamento verrà eseguito immediatamente e il secondo verrà eseguito solo dopo che sarà trascorso il tempo *t*. Ad esempio, questo può essere usato per limitare la velocità con cui viene ricalcolato un risultato costoso: potresti essere disposto a lasciare che il valore calcolato sia obsoleto di alcuni secondi se ti consente di risparmiare sull'esecuzione del calcolo costoso più di una volta ogni pochi secondi.

6 Considerazioni sul design

Semplice da usare

Ciò significava che la sintassi per scrivere programmi in un modo dependency-tracking doveva essere il più leggera possibile, che i programmi scritti usando FRP dovevano apparire come le loro controparti normali, antiquate e imperative. Ciò significava utilizzare *DynamicVariable* invece di impliciti per passare automaticamente gli argomenti, sacrificando l'ambito lessicale appropriato per una bella sintassi.

Ho escluso l'utilizzo di uno stile puramente monadico (come reactive-web), poiché sebbene sarebbe stato molto più semplice implementare la libreria in questo modo, sarebbe stato molto più doloroso utilizzarla effettivamente. Inoltre, non volevo dover dichiarare manualmente le dipendenze, poiché ciò viola DRY[11] quando dichiari le tue dipendenze due volte: una volta nell'intestazione della Rx e un'altra volta quando la usi nel corpo.

L'obiettivo era essere in grado di scrivere codice, spruzzare alcuni *Rx{}* in giro e far funzionare il monitoraggio delle dipendenze e la propagazione delle modifiche. Nel complesso, credo che abbia avuto abbastanza successo in questo!

Semplice da ragionare su

Questo significa molte cose, ma soprattutto significa non avere globali. Ciò semplifica notevolmente molte cose per chi utilizza la libreria, poiché non è più necessario ragionare sulle diverse parti del programma che interagiscono attraverso la libreria. L'utilizzo di Scala.Rx in diverse parti di un programma di grandi dimensioni va benissimo; sono completamente indipendenti.

Un'altra decisione di progettazione in quest'area è stata quella di lasciare il parallelismo e la pianificazione della propagazione principalmente a un *ExecutionContext* implicito e avere l'impostazione predefinita per eseguire semplicemente l'onda di propagazione su qualsiasi thread abbia effettuato l'aggiornamento al grafico del flusso di dati.

- Il primo significa che chiunque sia abituato a scrivere programmi paralleli in Scala/Akka ha già familiarità con come gestire la parallelizzazione di Scala.Rx
- Quest'ultimo rende molto più facile ragionare su quando avvengono le propagazioni, almeno nel caso predefinito: accade semplicemente subito e nel momento in cui la funzione *Var.update()* è tornata, la propagazione è stata completata.

Nel complesso, limitare la gamma di effetti collaterali e rimuovere lo stato globale rende Scala.Rx facile da ragionare e significa che uno sviluppatore può concentrarsi sull'utilizzo di Scala.Rx per costruire grafici di flussi di dati piuttosto che preoccuparsi di interazioni imprevedibili di vasta portata o colli di bottiglia delle prestazioni .

Interoperabilità semplice

Ciò significava che doveva essere facile per un programmatore entrare e uscire dal mondo FRP. Molti articoli descrivevano sistemi che funzionavano brillantemente da soli e avevano alcune proprietà sorprendenti, ma richiedevano che l'intero programma fosse scritto in una variante oscura di un linguaggio oscuro. Non è stato dato alcun pensiero all'interoperabilità con linguaggi o paradigmi esistenti, il che rende impossibile l'introduzione incrementale di FRP in una base di codice esistente.

Quindi, Scala.Rx:

- È scritto in Scala: un linguaggio non comune, ma probabilmente meno oscuro di Haskell[12] o Scheme[13]
- È una biblioteca: è semplice-vecchia-scala. Non c'è alcuna trasformazione da sorgente a sorgente, nessun runtime speciale necessario per usare Scala.Rx. Scarichi il codice sorgente nel tuo progetto Scala e inizi a usarlo
- Ti consente di utilizzare qualsiasi costrutto di linguaggio di programmazione o funzionalità di libreria all'interno del tuo Rxs: Scala.Rx individuerà le dipendenze senza che il programmatore debba preoccuparsene, senza limitarsi a qualche scomodo sottoinsieme del linguaggio
- Consente di utilizzare Scala.Rx all'interno di un progetto più ampio senza troppi problemi. È possibile incorporare facilmente i grafici del flusso di dati all'interno di un universo orientato agli oggetti più ampio e interagire con essi impostando *Var*s e ascoltando *Obs*s

Molti dei documenti recensiti mostrano un bellissimo nuovo universo FRP in cui potremmo programmare, se solo portassi tutto il tuo codice su FRP-Haskell e ti limitassi al piccolo insieme di combinatori utilizzati per creare grafici di flussi di dati. D'altra parte, consentendo di incorporare frammenti FRP ovunque all'interno del codice esistente, utilizzando idee FRP in progetti esistenti senza un impegno totale e consentendo una facile interoperabilità tra il codice FRP e non FRP, Scala.Rx mira a portare i vantaggi FRP in l'universo sporco e disordinato in cui stiamo programmando oggi.

Limitazioni

Scala.Rx ha una serie di limitazioni significative, alcune delle quali derivano da compromessi nella progettazione, altre dalle limitazioni della piattaforma sottostante.

- **No "Empty" Reactives**

L'API di Rxs in Scala.Rx cerca di seguire l'API delle collezioni il più possibile: puoi mappare, filtrare e ridurre sugli Rx, proprio come puoi sulle collezioni. Tuttavia, attualmente è impossibile avere un Rx vuoto nel modo in cui una raccolta può essere vuota: il filtraggio di tutti i valori in un Rx lascerà comunque almeno il valore iniziale (anche se fallisce il predicato) e Async Rx deve essere dato un valore iniziale per iniziare.

Questa limitazione nasce dalla difficoltà di unire Rx eventualmente vuoti con una buona esperienza d'uso. Ad esempio, se ho un grafico del flusso di dati:

```
val a = Var()
val b = Var()
var c = Rx{
  .... a() ...
  ... some computation ...
  ... b() ...
  result
}
```

Dove *a* e *b* sono inizialmente vuoti, ho fondamentalmente quattro opzioni:

- Blocca il thread corrente che sta calcolando *c*, in attesa che *a* e poi *b* diventino disponibili.
- Genera un'eccezione quando vengono richiesti *a()* e *b()*, interrompendo il calcolo di *c* ma registrandolo per essere riavviato quando *a()* o *b()* diventano disponibili.
- Riscrivi questo in uno stile monadico usando per la comprensione.
- Usa il plugin delle continuazioni delimitate per trasformare automaticamente il codice sopra in codice monadico.

La prima opzione è un problema di prestazioni: i thread sono generalmente estremamente pesanti sulla maggior parte dei sistemi operativi. Non puoi ragionevolmente creare più di qualche migliaio di thread, che è un numero esiguo rispetto alla quantità di oggetti che puoi creare. Quindi, anche se il blocco sarebbe il più semplice, è disapprovato in molti sistemi (ad esempio in Akka, su cui è costruito Scala.Rx) e non sembra una buona soluzione.

La seconda opzione è un problema di prestazioni in un modo diverso: con *n* dipendenze diverse, che possono iniziare tutte vuote, potrebbe essere necessario avviare e interrompere il calcolo di *c* *n* volte prima di essere completato anche solo una volta. Sebbene questo non blocchi alcun thread, sembra estremamente costoso.

La terza opzione non è possibile dal punto di vista dell'esperienza utente: richiederebbe cambiamenti di vasta portata nella base del codice e nello stile di codifica per trarre vantaggio dalla propagazione del cambiamento, che non sono disposto a richiedere.

L'ultima opzione è problematica a causa dei bug del plugin per le continuazioni delimitate. Sebbene in teoria dovrebbe essere in grado di risolvere tutto, un gran numero di piccoli bug (incasinare l'inferenza di tipo, interferire con la risoluzione implicita) combinato con alcuni problemi

fondamentali hanno fatto sì che anche su un progetto su piccola scala (meno di 1000 righe di codice reattivo) stava diventando doloroso da usare.

- **No Automatic Parallelization at the Start**

Come accennato in precedenza, Scala.Rx può eseguire la parallelizzazione automatica degli aggiornamenti che si verificano nel grafico del flusso di dati: è sufficiente fornire un ExecutionContext appropriato e gli Rx indipendenti avranno i loro aggiornamenti distribuiti su più core.

Tuttavia, questo funziona solo per gli aggiornamenti, e non quando il grafico del flusso di dati viene inizialmente definito: in tal caso, ogni Rx valuta il suo corpo una volta per ottenere il suo valore predefinito e tutto avviene in serie sullo stesso thread. Questa limitazione deriva dal fatto che non abbiamo un buon modo per lavorare con Rx "vuoti" e non sappiamo cosa siano le dipendenze Rxs prima della prima volta che lo valutiamo.

Quindi, non possiamo iniziare tutti i nostri Rx valutando in parallelo poiché alcuni potrebbero finire prima di altri da cui dipendono, che sarebbero quindi vuoti, il loro valore iniziale ancora in fase di calcolo. Inoltre, non possiamo scegliere di parallelizzare quelli che non hanno dipendenze l'uno dall'altro, poiché prima dell'esecuzione non sappiamo quali siano le dipendenze!

Quindi, non abbiamo altra scelta che far sì che le definizioni iniziali di Rxs avvengano in serie. Se necessario, un programmatore può creare manualmente Rx indipendenti in parallelo utilizzando Futures[14].

- **Glitchiness and Redundant Computation**

Nel contesto di FRP, un problema tecnico è un'incoerenza temporanea nel grafico del flusso di dati. A causa del fatto che gli aggiornamenti non avvengono istantaneamente, ma richiedono tempo per essere calcolati, i valori all'interno di un sistema FRP potrebbero essere momentaneamente fuori sincrono durante il processo di aggiornamento. Inoltre, a seconda della natura del sistema FRP, è possibile che i nodi vengano aggiornati più di una volta in una propagazione.

Questo può o non può essere un problema, a seconda di come l'applicazione è tollerante di dati incoerenti occasionali obsoleti. In un sistema a thread singolo, può essere evitato in diversi modi

- Rendi statico il grafico del flusso di dati ed esegui un ordinamento topologico per classificare i nodi nell'ordine in cui devono essere aggiornati. Ciò significa che un nodo viene sempre aggiornato dopo le sue dipendenze, il che significa che non vedranno mai dati obsoleti
- Sospende l'aggiornamento di un nodo quando tenta di richiamare una dipendenza che non è stata aggiornata. Questo potrebbe essere fatto bloccando il thread, ad esempio, e riprendendo solo dopo che la dipendenza è stata aggiornata.

Tuttavia, entrambi questi approcci presentano problemi. Il primo approccio è estremamente restrittivo: un grafico del flusso di dati statico significa che una grande quantità di comportamenti utili, ad es. è vietato creare e distruggere sezioni del grafico in modo dinamico in fase di esecuzione. Questo va contro l'obiettivo di Scala.Rx di consentire al programmatore di scrivere codice "normalmente" senza limiti e lasciare che il sistema FRP lo capisca.

Il secondo caso è un problema per i linguaggi che non consentono facilmente di mettere in pausa i calcoli. In Java, e per estensione Scala, i thread utilizzati sono thread del sistema operativo (OS) che sono estremamente costosi. Quindi, il blocco di un thread del sistema operativo è disapprovato. Anche le coroutine e le continuazioni potrebbero essere utilizzate per questo, ma Scala non ha entrambe queste strutture.

L'ultimo problema è che entrambi questi modelli hanno senso solo nel caso di codice sequenziale a thread singolo. Come menzionato nella sezione Concurrency and Parallelism, Scala.Rx consente di utilizzare più thread per parallelizzare la propagazione e consente di avviare le propagazioni da più thread contemporaneamente. Ciò significa che un rigoroso divieto di glitch è impossibile.

Scala.Rx mantiene un modello un po' più flessibile: il corpo di ogni *Rx* può essere valutato più di una volta per propagazione e Scala.Rx promette solo di fare il "best-effort" per ridurre il numero di aggiornamenti ridondanti. Supponendo che il corpo di ogni *Rx* sia puro, ciò significa che gli aggiornamenti ridondanti dovrebbero influenzare solo il tempo impiegato e il calcolo richiesto per il completamento della propagazione, ma non influenzare il valore di ciascun nodo una volta terminata la propagazione.

Inoltre, Scala.Rx fornisce gli *Obs*s, che sono nodi terminali speciali garantiti per l'aggiornamento solo una volta per propagazione, destinati a produrre qualche effetto collaterale. Ciò significa che sebbene una propagazione possa causare la temporanea fuori sincronismo dei valori degli *Rx*s all'interno del grafico del flusso di dati, gli effetti collaterali finali della propagazione si verificheranno solo una volta completata l'intera propagazione e tutti gli *Obs*s attiveranno i loro effetti collaterali.

Se si verificano più propagazioni in parallelo, Scala.Rx garantisce che ogni *Obs* si attiverà al massimo una volta per propagazione, e almeno una volta nel complesso. Inoltre, ogni *Obs* si attiverà almeno una volta dopo che l'intero grafico del flusso di dati si è stabilizzato e le propagazioni sono state completate. Ciò significa che se ti affidi a *Obs*, ad esempio, per inviare aggiornamenti sulla rete a un client remoto, puoi essere sicuro di non avere chiacchiere inutili trasmesse sulla rete e quando il sistema è inattivo il client remoto avrà gli aggiornamenti che rappresentano la versione più aggiornata del grafico del flusso di dati.

Scala.Rx non è stato creato nel vuoto e prende in prestito idee e ispirazione da una serie di progetti esistenti.

Bibliografia

[1] [online] Available: https://www.reactivemanifesto.org/

[2] [online] Available: https://en.wikipedia.org/wiki/Recursion_%28computer_science%29

[3] [online] Available: https://www.lihaoyi.com/scala.rx/#rx.core.Rx

[4] [online] Available: https://en.wikipedia.org/wiki/Functional_reactive_programming

[5] [online] Available:

http://archive.www6.in.tum.de/www6/pub/Main/TeachingWs2013MSE/2_OderskyDeprecatingObservers.pdf

[6] [online] Available: https://search.maven.org/artifact/com.scalarx/scalarx_2.10/0.1/jar

[7] [online] Available: http://www.scala-js.org/

[8] [online] Available: https://www.lihaoyi.com/scala.rx/#rx.core.Var

[9] [online] Available: https://www.lihaoyi.com/scala.rx/#rx.core.Obs

[10] [online] Available: https://www.lihaoyi.com/scala.rx/#rx.ops.Timer

[11] [online] Available: https://en.wikipedia.org/wiki/Don't_repeat_yourself

[12] [online] Available: https://wiki.haskell.org/Haskell

[13] [online] Available: https://racket-lang.org/

[14] [online] Available: https://stackoverflow.com/questions/12923429/easy-parallel-evaluation-of-tuples-in-scala