Università di Bologna

Corso di Laurea Magistrale in
Ingegneria e Scienze Informatiche

Paradigmi di Programmazione e Sviluppo

PPS-18-PrologZ

Bombardi Tommaso

16 giugno 2020

Indice

Capitolo 1

Scalaz

Scalaz è una libreria open source per la programmazione funzionale, che mette a disposizione un grande numero di strutture dati puramente funzionali create per integrare quelle della Scala Standard Library. In Scalaz sono presenti diversi nuovi elementi, definiti principalmente sotto forma di type class, ma anche delle estensioni per le strutture dati standard.

Scalaz rappresenta un progetto molto ampio, tanto che la prima release risale al 2010 e l'ultima a pochi giorni fa e che si sono sviluppati diversi progetti a partire da esso (come ad esempio Scalaz IO, poi trasformatosi nello standalone ZIO, una libreria per la programmazione asincrona e concorrente). Perciò sarà approfondita soltanto una parte delle strutture dati disponibili in Scalaz, con un duplice obiettivo: fornire una panoramica significativa della libreria ed introdurre gli elementi che saranno utilizzati per l'implementazione del caso d'uso presentato in questo elaborato.

Non esiste una documentazione ufficiale in grado di illustrare in modo esaustivo i costrutti presenti in Scalaz e le motivazioni che hanno portato alla loro implementazione. Perciò lo studio svolto e riportato di seguito si basa su diverse fonti, le più importanti delle quali sono: la Scaladoc e gli esempi che accompagnano la libreria [1], i post del blog *learning Scalaz* scritto da Eugene Yokota [2] e il libro *Functional Programming for Mortals with Scalaz* [3].

1.1 Library introduction

Scalaz aggiunge alla libreria standard strutture puramente funzionali, molte delle quali introdotte dal linguaggio Haskell, e lo fa utilizzando ampiamente tecniche avanzate quali polimorfismo ad hoc e method injection.

La versione di Scalaz studiata e usata in questo elaborato è la *7.2.30*, disponibile sia tramite l'apposito file jar sia usando SBT e specificando nel file di build *libraryDependencies += "org.scalaz" %% "scalaz-core" % "7.2.30"*.

1.1.1 Ad hoc polymorphism

In informatica, il termine polimorfismo viene usato per fare riferimento ad espressioni che possono rappresentare valori di diversi tipi [4]. Si può ottenere un comportamento polimorfico in vari modi, e si parla infatti di:

- **Polimorfismo parametrico**

  Il polimorfismo parametrico permette di definire genericamente una funzione o un tipo di dati, così che i valori possano essere gestiti nello stesso modo indipendentemente dal loro tipo. Tali funzioni e tipi di dati sono detti rispettivamente funzioni generiche e tipi generici.

  Per esempio, la funzione che restituisce l'elemento in testa ad una lista può essere definita in modo generico e sarà utilizzata una variabile di tipo corrispondente al tipo generico di elementi contenuti nella lista.

```
def head[A](xs: List[A]): A = xs(0)
head(1 :: 2 :: Nil) // 1
head("a" :: "b" :: "c" :: Nil)  // "a"
```

- **Polimorfismo per inclusione**

  Il polimorfismo per inclusione, detto anche polimorfismo del sottotipo, è fortemente legato al concetto di ereditarietà. Si tratta di una forma di polimorfismo di tipo in cui un sottotipo è un tipo di dati correlato ad un altro tipo (il supertipo) da una nozione di sostituibilità. Ciò significa che funzioni o altri elementi del programma scritti per operare su elementi del supertipo possono farlo anche su elementi del sottotipo.

  Rispetto al polimorfismo parametrico, quello per inclusione consente di fornire implementazioni diverse per un tipo generico A. Il principale svantaggio è costituito dal fatto che le implementazioni devono essere fornite al momento della definizione della struttura dati, perciò non è possibile definire metodi per tipi primitivi quali Int e String.

```
trait Plus[A] {
  def plus(a2: A): A
}
def plus[A <: Plus[A]](a1: A, a2: A): A = a1.plus(a2)

case class MyInt(value: Int) extends Plus[MyInt] {
  override def plus(a2: MyInt): MyInt = MyInt(value+a2.value)
}
plus(MyInt(1), MyInt(2))  // MyInt(3)
```

- **Polimorfismo ad hoc**

  Il polimorfismo ad hoc prevede che le funzioni polimorfiche possano essere applicate ad argomenti di tipo diverso, perché ognuna di esse può indicare implementazioni distinte e potenzialmente eterogenee a seconda del tipo di argomento a cui è applicata. Perciò questo polimorfismo si definisce *ad hoc* ed è in contrasto col parametrico, caratterizzato da una sola implementazione per ogni funzione polimorfica.

  In Scala, il polimorfismo ad hoc può essere realizzato sfruttando strumenti avanzati quali conversioni implicite o parametri impliciti.

```
trait Plus[A] {
  def plus(a1: A, a2: A): A
}
def plus[A: Plus](a1: A, a2: A): A =
  implicitly[Plus[A]].plus(a1, a2)

implicit object MyInt extends Plus[Int] {
  override def plus(a1: Int, a2: Int): Int = a1 + a2
}
plus(1, 2)  // 3
```

Scalaz utilizza il polimorfismo ad hoc perché consente di fornire definizioni di funzioni separate per diversi tipi di A (possibilità non data dal polimorfismo parametrico), di mettere a disposizione definizioni di funzioni per dei tipi (come Int nell'esempio precedente) senza dover accedere al loro codice sorgente (potenzialità non presente nel polimorfismo per inclusione) e di abilitare o disabilitare le definizioni delle funzioni in diversi ambiti.

1.1.2 Method injection

Un'altra tecnica avanzata che Scalaz usa per fornire le proprie astrazioni funzionali è la cosiddetta *method injection*. Facendo riferimento all'esempio riportato per il polimorfismo ad hoc, per un tipo X per cui è disponibile un'istanza di Plus[X] è possibile usare il metodo *plus*. Se volessimo invece avere un metodo che può essere richiamato direttamente da un'istanza di X?

Applicando il meccanismo di *method injection* nel caso di Plus[X], sarebbe definito un trait PlusOp[X] che contiene un valore di tipo X ed espone il metodo *plus* (o un suo alias). L'implementazione di quest'ultimo richiamerebbe il metodo *plus* visto in precedenza, quello che accetta due parametri. Una semplice conversione implicita dei tipi per cui è definito Plus[X] in PlusOp[X] fornirebbe loro il metodo *plus*, realizzando così l'iniezione di metodo.

1.1.3 Type system

Il type system è un sistema logico che comprende un insieme di regole, assegna una proprietà chiamata tipo ai vari costrutti del programma (variabili, espressioni, funzioni, ...) e permette quindi di verificare automaticamente l'assenza di determinati errori di comportamento in un programma.

In un linguaggio tipato come Scala, questo controllo è svolto in fase di compilazione e il programma viene compilato con successo solo in assenza di errori. Il type system di Scala è uno dei più sofisticati in assoluto, ma pensiamo ad un semplice esempio: *1 == "1"*. Questa espressione restituisce *false* ma, dato che confronta due valori di tipi diversi (Int e String), in un'implementazione ideale non dovrebbe poter essere compilata con successo.

Scalaz fornisce supporto al type system di Scala mettendo a disposizione la type class Equal[A], un'alternativa type safe alla classica implementazione di ==. Per tutti i valori di tipi per cui è definita un'istanza di Equal, grazie alla *method injection*, Scalaz fornisce gli operatori ===, =/= e assert_===. I primi due sono versioni typesafe degli operatori == e !=, mentre il terzo è un metodo typesafe con lo stesso comportamento di *assert(val1==val2)*.

```
trait Equal[A] {
  def equal(a1: A, a2: A): Boolean
}

1 === 1 // true
1 == "1"  // false
1 === "1" // error:type mismatch; found:String("1"); required:Int
1.some =/= 2.some // true
1.some =/= "2".some // error:type mismatch; found:Option[String];
   required:Option[Int]
1 assert_=== 1  // ok
1 assert_=== 2  // java.lang.RuntimeException: 1 != 2
1 assert_=== "1"  // cannot prove that String <:< Int
```

CODICE 1.1: Definizione di Equal e esempi di utilizzo degli operatori iniettati che abilita

1.2 Type classes for functional programming

Scalaz fornisce molte delle type class appartenenti alla gerarchia definita nel linguaggio Haskell. Esse rappresentano concetti ricorrenti nella programmazione funzionale, che non sono però espressi esplicitamente nella Scala Standard Library. Saranno quindi approfondite le astrazioni più diffuse.

1.2.1 Semigroup and Monoid

In matematica, un semigruppo è definito come una struttura dati algebrica costituita da un insieme di elementi munito di un'operazione binaria associativa (funzione che associa a due elementi di un tipo un terzo elemento dello stesso tipo). Analizzando questo concetto nell'ambito della programmazione, un semigruppo può essere costituito, ad esempio, da un insieme di stringhe e dall'operazione di concatenazione oppure da un insieme di interi e dall'operazione di addizione, moltiplicazione o elevamento a potenza.

Scalaz offre il trait Semigroup[S], e ogni sua implementazione (type class) rappresenta un semigruppo [5]. Esso soddisferà due regole:

- *Chiusura*: $\forall$ a, b $\in$ S, append(a, b) $\in$ S. È garantita dal type system.
- *Associatività*: $\forall$ a, b, c $\in$ S, append(append(a,b),c) = append(a,append(b,c)).

Inoltre Scalaz definisce l'operatore *mappend* e il suo alias simbolico |+| che, tramite il meccanismo di method injection (e quindi usando gli impliciti), possono essere richiamati su un oggetto per il cui tipo è stata definita un'istanza di Semigroup e restituiscono la "combinazione" dei due oggetti.

```
trait Semigroup[S] {
  def append(s1: S, s2: => S): S
}
trait SemigroupOps[A] extends Ops[A] {
  final def |+|(other: => A): A = A.append(self, other)
  final def mappend(other: => A): A = A.append(self, other)
}
```

CODICE 1.2: Definizione di Semigroup e operatore *mappend*

In sintesi, è possibile affermare che un Semigroup consente di definire una politica con cui combinare gli elementi di un determinato tipo. Prendiamo però un semplice esempio: data una lista di elementi di un certo tipo e un semigruppo che consente di combinarli, si vuole ottenere un singolo elemento corrispondente alla combinazione di tutti gli elementi della lista (ad esempio, partendo dal primo e arrivando all'ultimo o viceversa).

Il problema dato potrebbe essere risolto semplicemente combinando progressivamente un "accumulatore" con i vari elementi della lista, ma da dove iniziare? Sarebbe necessario avere a disposizione l'elemento neutro dell'operazione di combinazione, che non è però definito in un Semigroup.

Partendo da questo tipo di esigenza Scalaz fornisce la propria definizione di monoide, Monoid[M], ossia un semigruppo dotato di un valore che funge

da identità rispetto alla funzione binaria associativa. Questo valore è detto zero e può essere, ad esempio, una stringa vuota nel caso della concatenazione, 0 nel caso dell'addizione o 1 per la moltiplicazione. Tutti i monoidi devono rispettare quanto previsto per i semigruppi e due regole aggiuntive:

- *Identità sinistra*: $\forall$ a $\in$ M, append(M.zero, a) = a.
- *Identità destra*: $\forall$ a $\in$ M, append(a, M.zero) = a.

Scalaz fornisce istanze implicite predefinite di semigruppi e monoidi, ad esempio per stringhe, collezioni (liste, set, mappe, ...) e numeri interi, in modo che per questi tipi l'operatore *mappend* e lo zero nel caso dei monoidi siano disponibili una volta importata Scalaz. La vera potenzialità di queste astrazioni è però la possibilità di definire delle politiche di combinazione per ogni tipo, ed applicarle semplicemente usando l'operatore *mappend*.

```
trait Zero[Z] {
  val zero: Z
}
trait Monoid[M] extends Zero[M] with Semigroup[M]

List(1, 2, 3) |+| List(4, 5, 6) // List(1, 2, 3, 4, 5, 6)
"one" mappend "two" // "onetwo"
```

CODICE 1.3: Definizione di Monoid e semplice esempio di utilizzo delle istanze predefinite presenti in Scalaz

1.2.2 Functor

Dato un semplice valore, è banale applicare ad esso una funzione. Spesso però questo valore è racchiuso all'interno di un contesto: in questo caso, la funzione non può essere applicata direttamente, il risultato dipende dal contesto ed entrano in gioco i funtori (un'altra astrazione funzionale).

Scalaz fornisce la propria definizione di funtore attraverso il trait Functor[F[_]], le cui implementazioni sono type class destinate agli oggetti che possono essere mappati. Il funtore definisce un metodo *map* in cui viene esplicitato come, dato un valore in un contesto F[A], applicare una funzione A -> B in modo da ottenere un nuovo valore all'interno di un contesto F[B].

Questa definizione è familiare se si considera il metodo *map* delle collezioni, che restituisce lo stesso tipo di collezione contenente i vari elementi della collezione dopo l'applicazione della funzione data, o quello degli option, che applica la funzione data soltanto se il valore è presente.

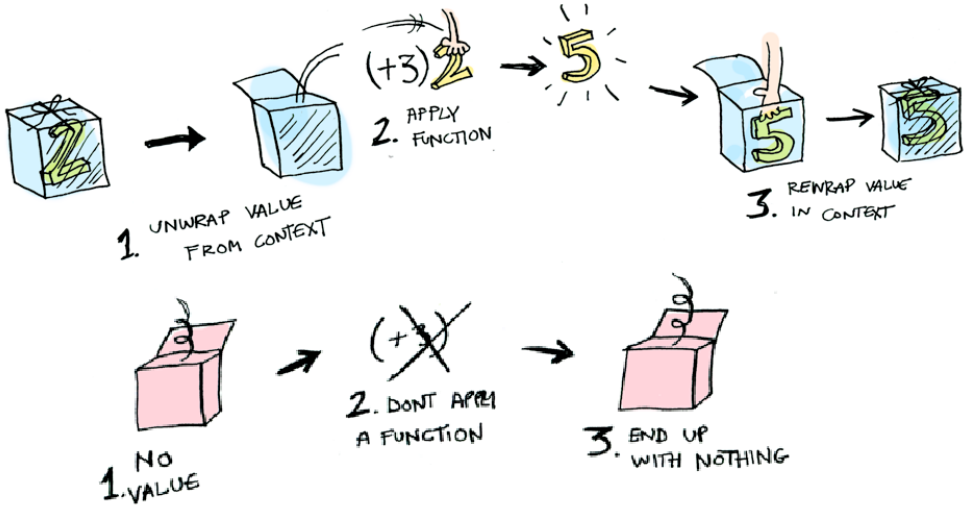

FIGURA 1.1: Comportamento della funzione *map* definita da Functor[Option] [6]

Oltre a fornire la propria definizione del concetto di funtore, per i tipi per cui è implementato un funtore Scalaz abilita alcuni operatori iniettati che rendono disponibile il metodo *map* e altri in grado di modificare la struttura dati di partenza come *as*, *fpair*, *fproduct*, *strengthL*, *strengthR*, e *void*.

Un aspetto molto interessante è costituito dal fatto che Scalaz definisce un'istanza di funtore anche per le funzioni, che in questo caso sono viste come una mappa infinita dal dominio al codominio. Così, usando il metodo *map*, è possibile comporre due o più funzioni, anche se con un'anomalia non trascurabile: l'ordine è invertito rispetto alla classica composizione.

Tradizionalmente, la composizione di f e g porta ad ottenere una funzione corrispondente all'esecuzione di g seguita dall'esecuzione di f. Al contrario nei Functor di Scalaz, dato che *map* è un metodo iniettato in F[A], la prima funzione eseguita è f e sul suo risultato viene poi applicata g.

```
trait Functor[F[_]]  {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
trait FunctorOps[F[_],A] extends Ops[F[A]] {
  implicit def F: Functor[F]
  final def map[B](f: A => B): F[B] = F.map(self)(f)
}

List(1, 2, 3) map {_ + 1} // List(2, 3, 4)
(((_: Int) * 3) map {_ + 100}) (1) // 103
(((x: Int) => x + 1) map {_ * 7}) (3) // 28
```

CODICE 1.4: Definizione di Functor e semplici esempi d'uso

1.2.3 Apply and Applicative

Un funtore consente di applicare una funzione ad un valore all'interno di un contesto, ma se anche la funzione fosse racchiusa in un contesto? Per consentire l'applicazione della funzione nel caso descritto Scalaz introduce Apply[F[_]], un'estensione di Functor[F] che espone un metodo *ap*. Si tratta di una versione potenziata di *map*, poiché estrae una funzione da un contesto e successivamente applica map su un valore usando la funzione estratta.

Inoltre Scalaz abilita gli operatori iniettati <*>, alias simbolico per il metodo *ap*, *> e <*, variazioni che restituiscono solo la parte destra o sinistra.

Scalaz raffina ulteriormente il concetto espresso con Apply, attraverso la definizione del trait Applicative[F[_]]. Gli applicativi, oltre ad essere istanze di Apply, introducono il metodo *point* (e il suo alias *pure*) che prende un valore di qualsiasi tipo e lo restituisce all'interno del contesto per cui è stato definito l'applicativo. Inoltre, Scalaz inietta il metodo *point* in tutti i tipi di dati consentendo così di trasformare un valore di tipo A in F[A].

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

9.some <*> {(_: Int) + 3}.some // Some(12)
3.some <*> { 9.some <*> {(_: Int) + (_: Int)}.curried.some }
  // Some(12)
List(1, 2, 3) <*> List((_: Int) * 0, (_: Int) + 100, x => x * x)
  // List(0, 0, 0, 101, 102, 103, 1, 4, 9)
List(3, 4) <*> { List(1, 2) <*> List({(_: Int) + (_: Int)}.curried,
    {(_: Int) * (_: Int)}.curried) }
  // List(4, 5, 5, 6, 3, 4, 6, 8)
```

CODICE 1.5: Definizione di Apply e Applicative e esempi di utilizzo dell'operatore <*>

Seguendo lo stile introdotto con gli applicativi, Scalaz definisce anche una nuova notazione ^(valore1, valore2, ...) { funzione } che estrae i valori dai loro contesti e li applica ad una singola funzione (può essere utile perché consente di comportarsi come con gli applicativi senza dover inserire la funzione nel contesto). Questo nuovo stile presenta però un problema: non è in grado di gestire applicativi che accettano due parametri di tipo.

Per fornire lo stesso supporto anche a questi applicativi è stato introdotto *Applicative Builder*, un'implementazione alternativa dello stile ispirato agli applicativi che usa la sintassi (valore1 |@| valore2 |@| ...) { funzione }.

```
^(3.some, 5.some) {_ + _} // Option[Int] = Some(8)
^(3.some, none[Int]) {_ + _} // None
(3.some |@| 5.some) {_ + _} // Option[Int] = Some(8)
(List("ha", "heh", "hmm") |@| List("?", "!", ".")) {_ + _}
  // List(ha?, ha!, ha., heh?, heh!, heh., hmm?, hmm!, hmm.)
```

CODICE 1.6: Esempi di *Applicative Style* e *Applicative Builder*

1.2.4 Monad

Nei funtori il valore a cui viene applicata la funzione si trova in un contesto, negli applicativi anche la funzione stessa è all'interno del contesto e, infine, le monadi sono una naturale estensione degli applicativi nata per risolvere i casi in cui il valore è nel contesto e deve essere applicata una funzione che accetta un valore normale e ritorna un valore all'interno del contesto.

A differenza degli applicativi, le monadi dovranno essere in grado di estrarre il valore dal loro contesto per poi applicare la funzione data.

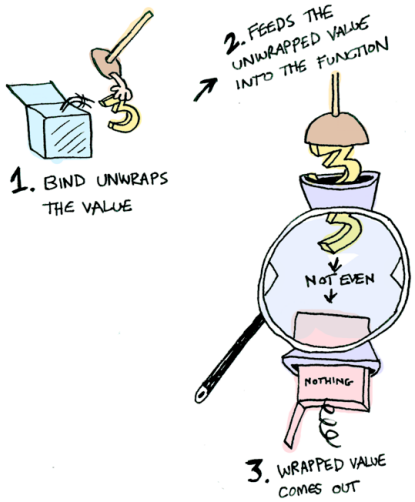

FIGURA 1.2: Comportamento di una monade [6]

Il trait Monad[F[_]] mette a disposizione la definizione di monade di Scalaz, da cui si evince che tutte le monadi debbano essere istanze sia di Applicative[F] sia di Bind[F]. Quest'ultimo rappresenta ciò che le monadi hanno in più rispetto ai classici applicativi ed espone un metodo *bind*, in grado di applicare una funzione A -> F[B] ad un valore nel suo contesto F[A]. Tutte le monadi dovranno inoltre rispettare le seguenti regole:

- *Identità sinistra*: ∀ a, f, f(a) = bind(pure(a), f). Quindi se si prende un valore, si inserisce in un contesto con *point* e si usa *bind*, il risultato è lo stesso che si avrebbe applicando direttamente la funzione al valore.
- *Identità destra*: ∀ a, a = bind(a, x => pure (x)). L'elemento neutro della *bind* è una funzione che restituisce il valore dato nel contesto.
- *Associatività*: ∀ a, f, g, bind(a, x => bind(f(x), g)) = bind(bind(a, f), g). Data una catena di applicazioni con funzioni monadiche, non importa come esse sono nidificate ma conta soltanto il loro ordine.

Come per gli altri costrutti visti, Scalaz fornisce ai tipi per cui è definita una monade il metodo *bind* tramite l'operatore *flatmap* e il suo alias »=.

```
trait Bind[F[_]] extends Apply[F] {
  def bind[A, B](fa: F[A])(f: A => F[B]): F[B]
}
trait BindOps[F[_],A] extends Ops[F[A]] {
  def flatMap[B](f: A => F[B]) = F.bind(self)(f)
  def >>=[B](f: A => F[B]) = F.bind(self)(f)
}
trait Monad[F[_]] extends Applicative[F] with Bind[F]

Monad[Option].point("WHAT") // Some(WHAT)
3.some flatMap { x => (x + 1).some } // Some(4)
(none: Option[Int]) flatMap { x => (x + 1).some } // None
9.some flatMap { x => Monad[Option].point(x * 10) } // Some(90)
3.some >>= { x => "!".some >>= { y => (none: Option[String]) } }
  // None
```

CODICE 1.7: Definizione di Monad e dell'operatore *flatMap*

Un aspetto da tenere in grande considerazione quando si parla di monadi è il fatto che questo tipo di astrazioni, abilitando l'operatore *flatMap*, possano essere usate in una *for comprehension*. Così facendo, sarà possibile concatenare in modo semplice e leggibile una serie di applicazioni di funzione.

```
3.some >>= { x => "!".some >>= { y => (none: Option[String]) } }

for {
  x <- 3.some
  y <- "!".some
  z <- none
} yield z
```

CODICE 1.8: Semplice esempio di composizione monadica senza e con l'utilizzo della *for comprehension*

1.3 Type classes for error handling

La programmazione funzionale prevede che il flusso di esecuzione del programma sia costituito da una serie di valutazioni matematiche, e il suo punto di forza è la mancanza di side effect. In caso di fallimento nell'esecuzione di una funzione (ad esempio, il mancato soddisfacimento di una condizione), è retaggio della programmazione imperativa l'utilizzo di eccezioni.

Il fatto che venga lanciata un'eccezione genera però un side effect, perché l'esecuzione del programma viene interrotta e continua in un luogo diverso rispetto a quello in cui si prevede che la funzione ritorni. Affinché sia completamente rispettato il paradigma funzionale, un linguaggio dovrebbe mettere a disposizione delle strutture dati in grado di incapsulare sia il valore atteso dalle funzioni sia un valore che rappresenta una condizione di errore.

Nella Scala Standard Library sono disponibili alcune astrazioni di questo tipo, che Scalaz va ad arricchire fornendo la propria implementazione dei costrutti già presenti e definendo delle nuove type class [7].

1.3.1 Option

La prima type class in grado di rappresentare un fallimento nell'applicazione di una funzione è Option[A], definita nella libreria standard e ampiamente usata, che offre la possibilità di ritornare o meno il risultato di una computazione. Scalaz fornisce un'estensione della sintassi disponibile per Option[A], aggiungendo metodi di costruzione alternativi e alcuni operatori unari.

In particolare, per i valori di qualsiasi tipo Scalaz fornisce il metodo *some* con cui è possibile creare un option contenente quel valore. Il metodo none[A] consente invece di creare un option vuoto specificando già in fase di creazione di quale tipo sarà (per questo si differenzia da *None.apply*).

```
trait OptionFunctions {
  final def some[A](a: A): Option[A] = Some(a)
  final def none[A]: Option[A] = None
}

1.some // Some(1)
val os = List(Some(42), None, Some(20))
os.foldLeft(None) { (acc, o) => acc orElse o }
  // error:type mismatch; found:Option[Int]; required:None.type
os.foldLeft(None:Option[Int]){(acc, o) => acc orElse o} // Some(42)
os.foldLeft(none[Int]) { (acc, o) => acc orElse o } // Some(42)
```

CODICE 1.9: Metodi introdotti per la costruzione di Option

Scalaz definisce inoltre alcuni operatori per le istanze di Option, tra cui:

- `|`, che costituisce un alias per il metodo *getOrElse*.
- $\sim$, ossia un alias per il metodo *getOrElse* chiamato con argomento *zero*.
- ? a | b, statement che verifica se è presente un valore nell'Option, in tal caso restituisce a e in caso contrario b.

1.3.2 Either \/

Mentre Option[A] consente di rappresentare l'assenza o la presenza di un errore, la type class Either[A, B] è in grado di inserire valori di successo in un contesto di possibile errore e di allegare un valore in caso di fallimento, in modo da poter fornire informazioni utili sull'errore che si è verificato.

Nella Scala Standard Library è definito Either[A, B], ma Scalaz fornisce la propria versione attraverso la definizione del trait \/[A, B]. Per convenzione, la parte sinistra di \/[A, B] viene usata per gli errori (il tipo A rappresenta un errore) e la parte destra è riservata agli esiti positivi. Le istanze di \/[A, B] possono essere create con i metodi *left* e *right*, iniettati in tutti i tipi.

Scalaz accompagna la definizione di \/[A, B] con metodi quali:

- *isRight* e *isLeft*, che controllano il lato da cui si trova il valore.
- *getOrElse* e il suo alias simbolico `|`, che restituiscono il valore nel lato destro o un valore di default in caso di errore.
- *swap* e il suo alias simbolico $\sim$, che invertono lato sinistro e destro.
- *map*, che è in grado di modificare il valore del lato destro.

Inizialmente \/[A, B] era preferibile rispetto a Either[A, B], poiché l'implementazione di Scalaz è monadica ed espone il metodo *flatMap*. Questo risulta essere molto comodo nella gestione degli errori, poiché abilita l'uso della *for comprehension*. Tuttavia, nelle ultime versioni della libreria standard Either è diventata monadica ed ora la differenza è prettamente stilistica.

```
for {
  e1 <- "event 1 ok".right
  e2 <- "event 2 failed!".left[String]
  e3 <- "event 3 failed!".left[String]
} yield (e1 |+| e2 |+| e3)  // -\/("event 2 failed!")
```

CODICE 1.10: Combinazione di \/ con una *for comprehension*: fallimento in corrispondenza del primo valore sinistro

1.3.3 Validation

Either, o \/ nella versione implementata da Scalaz, è uno strumento molto utile perché in grado di fornire una rappresentazione del fallimento. Pensiamo però ad uno scenario molto comune, in cui si verificano diversi errori nel corso della computazione e ognuno di essi è ritenuto importante.

Ad esempio, si consideri un form di registrazione in cui sono richiesti nome utente, password e conferma password. Affinché l'input dell'utente sia valido, il nome utente deve essere formato da un minimo di *m* caratteri e non contenere caratteri speciali, la password deve essere composta da un minimo di *n* caratteri e contenere almeno una lettera maiuscola e il campo di conferma deve essere uguale alla password. Nel caso in cui un utente commetta più errori, un buon sistema dovrebbe essere in grado di segnalarli tutti così che l'utente possa correggerli contemporaneamente.

Benché diversi valori di \/ possano essere facilmente composti con una *for comprehension*, la composizione assume un comportamento *first-fail* poiché in caso di fallimento il valore restituito rappresenta solo il primo errore. Quindi \/ non è adatto all'esempio: se l'utente commettesse un errore sia nel nome utente sia nella password, il secondo non sarebbe mostrato.

Partendo da questi presupposti, Scalaz introduce il trait Validation[E, A] e le sue implementazioni Success[E, A] e Failure[E, A]. Il costrutto Validation[E, A] è isomorfo a \/[E, A] ed è infatti possibile convertirlo in esso tramite il metodo *disjunction* (la conversione inversa è disponibile grazie al metodo *validation*). Le istanze di Validation[E, A] possono essere create con i metodi *success[E]* e *failure[A]*, iniettati da Scalaz in tutti i tipi di dati.

A differenza di \/[E, A], Validation[E, A] non è una monade ma, quando per E (il tipo che rappresenta l'errore in Validation[E, A]) è definito un semigruppo, esiste un Applicative[Validation[E, A]] in grado di accumulare gli errori. La sintassi più comune usata per unire le validazioni è quella proposta dall'*Applicative Builder*, che consente di specificare solo la funzione con cui combinare i valori in caso di successo. Gli eventuali errori sono invece accumulati automaticamente secondo la politica stabilita dal semigruppo.

```
"event 1 ok".success[String]  // Success("event 1 ok")
"event 1 failed!".failure[String] // Failure("event 1 failed!")
"event 1 ok".success[String] |@| "event 2 failed!".failure[String]
   |@| "event 3 failed!".failure[String]) {_ + _ + _}
  // Failure("event 2 failed!event 3 failed!")
```

CODICE 1.11: Creazione di Validation e loro composizione (possibile perché è definito un semigruppo per le stringhe)

1.3.4 ValidationNel

Validation[E, A] consente di rappresentare nello stesso costrutto sia un possibile successo sia un insieme di fallimenti, ma se non fosse possibile combinare gli errori? O se si volesse mantenere la loro separazione? Ad esempio, dato un insieme di Validation[String, A], sarebbe possibile rappresentare gli errori come una concatenazione di stringhe, ma tutti i messaggi di errore sarebbero uniti in una stringa. Di certo, non si tratta della struttura dati ideale e sarebbe più naturale usare una collezione, come una semplice lista.

In questo contesto è utile il trait NonEmptyList[A], definito in Scalaz e a cui si fa riferimento con l'abbreviazione *Nel*, che costituisce un wrapper per una lista, garantisce che essa non sia vuota e fa sì che il metodo *head* sia sempre disponibile. Per la creazione delle istanze di NonEmptyList Scalaz mette a disposizione il metodo *wrapNel*, iniettato in tutti i tipi di dati.

```
sealed trait NonEmptyList[A] {
  val head: A
  val tail: List[A]
  def <:[AA >: A](b: AA): NonEmptyList[AA] = nel(b, head :: tail)
}

1.wrapNel // NonEmptyList(1)
"error message".wrapNel // NonEmptyList("error message")
```

CODICE 1.12: Definizione di NonEmptyList e costruzione usando il metodo *wrapNel*

Utilizzare *Nel* e non una semplice lista come tipo per rappresentare gli errori in una validazione fa in modo che possa essere evitato, grazie al type system, il caso intrinsecamente ambiguo di *Failure(List())*. Esso potrebbe infatti rappresentare infatti sia un fallimento sia l'assenza di errori.

Perciò Scalaz mette a disposizione ValidationNel[E, A], un type alias per Validation[NonEmptyList[E], A], *successNel[E]* e *failureNel[A]*, metodi iniettati in ogni tipo per la creazione di istanze di ValidationNel[E, A] [8].

```
"event 1 ok".successNel[String]
  // Success("event 1 ok")
"event 1 failed!".failureNel[String]
  // Failure(NonEmptyList("event 1 failed!"))
("event 1 ok".successNel[String] |@| "event 2 failed!".failureNel[
    String] |@| "event 3 failed!".failureNel[String]) {_ + _ + _}
  // Failure(NonEmptyList("event 2 failed!", "event 3 failed!"))
```

CODICE 1.13: Esempi di creazione e composizione di ValidationNel

1.4 Useful features for everyday usage

In aggiunta alle type class presentate nelle sezioni precedenti, Scalaz offre molti altri costrutti che possono essere un valido supporto per la programmazione. Di seguito saranno approfonditi alcuni di essi, ritenuti significativi e rivelatisi utili nell'implementazione del caso d'uso presentato.

1.4.1 Tagging

Il type system di Scala ha diverse potenzialità, come la possibilità di definire type alias in grado di rendere il codice più leggibile, ma non consente di limitare il range di supporto di un tipo. Si supponga ad esempio di voler esprimere una massa con il chilogrammo: si potrebbe usare un Double, ma come distinguerlo da altri Double che rappresentano concetti diversi?

La soluzione più semplice, e probabilmente la più naturale, consiste nel definire una case class *KiloGram* contenente un solo parametro *value* corrispondente alla massa (espressa attraverso un Double). In questo modo si ottiene la sicurezza dei tipi, ma si presentano i seguenti problemi:

- È necessario chiamare *object.value* per estrarre il valore.
- A runtime si verificheranno boxing e unboxing del valore, che comportano un maggiore costo di esecuzione. Sarà infatti allocato nello heap un oggetto wrapper (istanza della case class) contenente il valore, mentre sarebbe sufficiente mantenere quest'ultimo sullo stack.

Una soluzione alternativa è quella basata sui tipi taggati messi a disposizione da Scalaz [9]. Il tagging consente di aggiungere ad un tipo dato un riferimento ad un altro tipo, andando così a definire un nuovo tipo. Inoltre, è stato introdotto l'alias di tipo @@ che, dati due tipi A e B, prevede che A with Tagged[B] possa essere rappresentato nella forma compatta A @@ B.

Scalaz fornisce il metodo *Tag[A, B].apply*, con cui è possibile taggare un valore di tipo A con il tipo B. Per ottenere il valore presente nel tipo taggato è invece necessaria un'operazione di cast, disponibile grazie a *Tag.unwrap*.

Facendo ancora riferimento alla rappresentazione del chilogrammo, sarà necessario definire un tipo *KiloGram* (solitamente con un sealed trait) che sarà usato per taggare il tipo dei valori (Double). Double @@ KiloGram, che è un tipo vero e proprio e non un alias di tipo, sarà usato per rappresentare la massa. Qualora venga richiesto un valore di questo tipo, l'utente è obbligato a taggare il dato e in questo modo si ottiene la sicurezza dei tipi.

```
type Tagged[U] = { type Tag = U }
type @@[T, U] = T with Tagged[U]

sealed trait KiloGram
def KiloGram[A](a: A): A @@ KiloGram = Tag[A, KiloGram](a)
val mass = KiloGram(20.0)
2 * Tag.unwrap(mass) // 40.0
```

CODICE 1.14: Definizione e esempio di tipo taggato (la funzione KiloGram tagga un valore di un qualsiasi tipo)

Essendo necessario un cast esplicito per ottenere il valore presente all'interno del tipo taggato, in merito a questo aspetto non vi è un vantaggio concreto rispetto all'uso di case class (in cui è necessario chiamare *object.value* per accedere al valore). Tuttavia, i tipi taggati portano i seguenti benefici:

- Non causano mai il boxing di un valore, perché il tipo taggato costituisce semplicemente il tipo che sarà associato al valore sullo stack.
- Consentono di scrivere facilmente codice generico per i tipi taggati. Per esempio, Double @@ A rappresenta tutti i Double taggati con un qualche tipo ed è possibile definire una funzione generica per essi. Questo non sarebbe possibile se i Double fossero incapsulati in delle case class.

1.4.2 Tree and TreeLoc

Nella Scala Standard Library non è disponibile una struttura dati che consenta agli sviluppatori di rappresentare esplicitamente un albero. Benché sia semplice implementare un costrutto corrispondente a un albero, la realizzazione di algoritmi e procedure in grado di manipolarlo risulta essere decisamente complessa. Tra le varie strutture dati a complemento della libreria standard, Scalaz offre la propria implementazione di albero: Tree[A].

Tutte le istanze di Tree[A] sono dei multi-way tree, noti anche come rose tree, ossia strutture dati ad albero con un numero variabile e illimitato di rami per nodo. Ogni nodo dell'albero è a sua volta rappresentato come un albero (il sotto-albero radicato in quel nodo), contiene un elemento di tipo A (etichetta associata al nodo) e tutti gli alberi corrispondenti ai nodi figli.

Scalaz mette inoltre a disposizione i metodi *leaf* e *node*, iniettati in tutti i tipi di dati, che consentono di creare alberi. In particolare, il primo non accetta nessun argomento e crea un albero corrispondente ad una foglia (nodo senza figli) e il secondo costruisce un nodo avente come figli i nodi dati.

```
sealed trait Tree[A] {
  def rootLabel: A
  def subForest: Stream[Tree[A]]
}
trait TreeFunctions {
  def leaf[A](root: => A): Tree[A] = node(root, Stream.empty)
  def node[A](root: => A, forest: => Stream[Tree[A]]): Tree[A] =
      new Tree[A] {
         lazy val rootLabel = root
         lazy val subForest = forest
         override def toString = "<tree>"
     }
}
object Tree extends TreeFunctions with TreeInstances {
    def apply[A](root: => A): Tree[A] = leaf(root)
}
trait TreeV[A] extends Ops[A] {
  def leaf: Tree[A] = Tree.leaf(self)
  def node(subForest: Tree[A]*): Tree[A] = Tree.node(self,
      subForest.toStream)
}

'P'.node('O'.node('L'.node('N'.leaf,'T'.leaf),'Y'.node('S'.leaf)),
   'L'.node('W'.node('C'.leaf,'R'.leaf),'A'.node('A'.leaf)))
```

CODICE 1.15: Definizione di Tree, delle sue modalità di costruzione e semplice esempio di utilizzo

Come già sottolineato, la complessità maggiore legata alla gestione dell'albero è costituita dalla realizzazione di strumenti in grado di manipolarlo. Tree[A] rappresenta un ottimo punto di partenza, ma come è possibile navigarlo? Come inserire un nodo nell'albero? Come verificare se un nodo è la radice dell'albero? Come accedere al nodo padre di un nodo dato?

Al momento Tree espone solo *rootLabel* e *subForest*, indubbiamente insufficienti per poter fare in modo semplice le operazioni appena elencate.

In quest'ambito si colloca il concetto di *zipper*, un idioma che usa l'idea di "contesto" per manipolare le posizioni in una struttura dati. Uno zipper consente di prendere una struttura dati e concentrarsi su una sua parte, rendendo più semplici e più efficienti le operazioni di accesso e modifica.

Scalaz definisce TreeLoc[A], uno zipper per Tree[A] che rappresenta un albero e una posizione al suo interno. In particolare, nel TreeLoc sono mantenuti: il nodo attualmente selezionato, i fratelli a sinistra del nodo corrente, i fratelli a destra del nodo corrente e il contesto del suo nodo padre.

Nelle istanze di Tree[A] è disponibile il metodo *loc*, che consente di generare il TreeLoc[A] corrispondente all'albero corrente. Ogni istanza di TreeLoc offre sia metodi capaci di spostare il focus all'interno dell'albero (e quindi di navigarlo) sia metodi in grado di aggiornare, inserire o eliminare degli elementi. Di seguito, sono riportate le funzionalità più significative:

- *root*, che restituisce la radice dell'albero.
- *parent*, che ritorna il padre del nodo corrente.
- *left* e *right*, che selezionano il fratello sinistro e destro del nodo corrente.
- *firstChild* e *lastChild*, che restituiscono rispettivamente il figlio più a sinistra e quello più a destra del nodo corrente.
- *getChild*, che ritorna il figlio n-esimo del nodo corrente.
- *findChild*, che seleziona il primo figlio diretto del nodo corrente in grado di soddisfare il predicato dato.
- *modifyTree*, che modifica il nodo corrente con la funzione data.
- *modifyLabel*, che modifica l'etichetta del nodo corrente applicando la funzione specificata.
- *insertDownLast*, che inserisce il nodo figlio come ultimo figlio del nodo corrente e assegna a quest'ultimo il focus.

```
sealed trait TreeLoc[A] {
  val tree: Tree[A]
  val lefts: TreeForest[A]
  val rights: TreeForest[A]
  val parents: Parents[A]
  def parent: Option[TreeLoc[A]] = ...
  def root: TreeLoc[A] = ...
  def left: Option[TreeLoc[A]] = ...
  def right: Option[TreeLoc[A]] = ...
  def firstChild: Option[TreeLoc[A]] = ...
  def lastChild: Option[TreeLoc[A]] = ...
  def getChild(n: Int): Option[TreeLoc[A]] = ...
  def findChild(p: Tree[A] => Boolean): Option[TreeLoc[A]] = ...
  def modifyTree(f: Tree[A] => Tree[A]): TreeLoc[A] = ...
  def modifyLabel(f: A => A): TreeLoc[A] = ...
  def insertDownLast(t: Tree[A]): TreeLoc[A] = ...
}
```

CODICE 1.16: Definizione di TreeLoc

1.4.3 State Monad

Oltre ad esplicitare il concetto di monade fornendo un valido supporto a chiunque ne voglia definire una, Scalaz offre delle monadi standard pensate per operare in situazioni ricorrenti, tra cui è importante ricordare:

- **Reader Monad** Ogni azione è una funzione che ha come parametro l'ambiente. Un'azione può quindi leggere quanto contenuto nell'ambiente, ma non può modificare l'ambiente visto dalle azioni successive.
- **Writer Monad** Ogni azione è una funzione che restituisce un valore da scrivere. Un'azione non può vedere quali altre azioni hanno già scritto, mentre i vari output accumulati sono disponibili al chiamante.
- **State Monad** Ogni azione accetta un valore che rappresenta il vecchio stato e restituisce un valore dello stesso tipo corrispondente al nuovo stato. Un'azione può vedere lo stato in entrata e modificarlo prima di passarlo alla successiva. La state monad porta con sé un vincolo più stringente rispetto alle precedenti: dato che lo stato deve essere passato tra le varie computazioni, queste devono essere sequenziali.

Sia per motivi di tempo sia perché le monadi reader e writer non sono state usate nella realizzazione del caso d'uso presentato, saranno approfonditi soltanto funzionamento e implementazione della state monad [10].

A differenza del concetto generale di monade, StateMonad avvolge in modo specifico le funzioni. Si tratta di funzioni particolari, che accettano uno stato e restituiscono un valore insieme ad un nuovo stato, e rappresentano quindi una computazione con stato. In Scalaz, ogni state monad è istanza di State[S, A] ed è possibile utilizzare il companion object State per inizializzarne una partendo da una funzione nella forma *s -> (a, s)*.

```
type State[S,A] = StateT[Id,S,A]
object State extends StateFunctions {
  def apply[S,A](f: S => (S,A)): State[S,A] = new StateT[Id,S,A] {
    def apply(s: S) = f(s)
  }
}
for {
  a <- State[Int,String] { s => (s+1, s"1st result is ${s+1}") }
  b <- State[Int,String] { s => (s*2, s"2nd result is ${s*2}") }
  c <- State[Int,String] { s => (s%10, s"3rd result is ${s%10}") }
} yield a + b + c
```

CODICE 1.17: Creazione e composizione di StateMonad

Nelll'esempio precedente l'oggetto State estende il trait StateFunctions, interessante perché contiene delle funzioni di supporto per operazioni che coinvolgono lo stato. Inoltre la *for comprehension* riportata fa sì che venga generata una state monad in grado di eseguire in sequenza le tre computazioni, in cui lo stato condiviso è passato da ogni computazione alla successiva.

```
trait StateFunctions {
  def constantState[S, A](a: A, s: => S): State[S, A] =
    State((_: S) => (s, a))
  def state[S, A](a: A): State[S, A] = State((_ : S, a))
  def init[S]: State[S, S] = State(s => (s, s))
  def get[S]: State[S, S] = init
  def put[S](s: S): State[S, Unit] = State(_ => (s, ()))
  def modify[S](f: S => S): State[S, Unit] = State(s => {
    (f(s), ())
  })
}
```

CODICE 1.18: Funzioni di supporto definite in StateFunctions

Si consideri però il caso in cui una o più computazioni espresse tramite StateMonad possano fallire: è oneroso delegare la gestione degli errori alla logica interna della monade, la soluzione ideale sarebbe quella di includere la gestione dei fallimenti all'interno della state monad. Per poterlo fare è necessaria l'introduzione di un nuovo concetto, quello di *monad transformer*. Si tratta di un'entità simile ad una monade che, a differenza di essa, non è autonoma ma modifica il comportamento di una monade sottostante.

Scalaz rappresenta il *monad transformer* dedicato alla StateMonad con il trait StateT[F[_], S, A]. Esso prevede che il risultato di ogni computazione sia incapsulato all'interno di un "contenitore" di tipo F e che da quest'ultimo dipendano le computazioni successive. Ad esempio, nel caso di State[Option, S, A] l'esecuzione di ogni computazione deve restituire Option[(S, A)] e, se dovesse verificarsi un fallimento, questo sarebbe propagato anche alle computazioni successive. In Scalaz State[S, A] è un "banale" type alias che fa riferimento a StateT[Id, S, A], dove Id corrisponde alla monade identità, poiché applica semplicemente la funzione incapsulata nella monade.

```
trait StateT[F[_], S, +A] { self =>
  def apply(initial: S): F[(S, A)]
  def run(initial: S): F[(S, A)] = apply(initial)
  def runZero(implicit S: Monoid[S]): F[(S, A)] = run(S.zero)
}
```

CODICE 1.19: Definizione del *monad transformer* StateT

Capitolo 2

DSL for Logic Programming

Questo secondo capitolo sarà dedicato alla presentazione del caso d'uso della libreria Scalaz realizzato in questo elaborato. Si è scelto di sfruttare alcune delle astrazioni offerte da Scalaz per implementare un DSL che imiti Prolog, in modo da fornire un supporto diretto alla programmazione logica.

Il DSL ha l'obiettivo di offrire un'interfaccia quanto più possibile semplice e simile a Prolog, ma che consenta di scrivere programmi direttamente in Scala. L'esecuzione di questi ultimi sarà basata sulla tecnica di risoluzione comunemente usata dai motori logici, ossia quella che prevede la costruzione di un SLD tree, e anch'essa sarà implementata con l'ausilio di Scalaz.

2.1 Available features

Il DSL realizzato supporta una parte di quanto offerto dal linguaggio Prolog e, di seguito, sono riportate le funzionalità che mette a disposizione:

- Numeri di tipo *Int* e *Double* possono essere utilizzati per rappresentare rispettivamente i numeri interi e decimali presenti in Prolog.
- Stringhe che iniziano con una lettera minuscola corrispondono a dei termini costanti, mentre le stringhe il cui primo carattere è costituito da una lettera maiuscola equivalgono a delle variabili.
- Mentre in fase di compilazione tutte le stringhe vengono accettate in quanto termini, a run-time non sono considerate valide le stringhe vuote e quelle che non contengono solo lettere. Quest'ultimo vincolo è stato aggiunto per colmare il gap tra un linguaggio che è tipato staticamente (Scala) e uno che non lo è (Prolog). Sono infatti evitati i casi in cui si confronta un numero (ad esempio 1) e la stringa corrispondente ("1"), che in Prolog non possono verificarsi. Inoltre, escludere i caratteri speciali consente alle stringhe di non contenere delimitatori come "," e ")".

- È possibile creare un funtore usando il metodo *apply* dell'oggetto *Struct* e specificando il nome del funtore, che deve essere una stringa non vuota, contenere solo lettere e iniziare con una lettera minuscola.

- A sua volta, un funtore offre un metodo *apply* che accetta un numero variabile di termini e restituisce un termine composto. È previsto un errore se non viene passato nessun termine quando si usa un funtore. Se si divide la definizione del funtore dalla creazione del termine composto, questa operazione è molto simile a ciò che si farebbe in Prolog.

```
Struct("functor")("a",1)

val functor = Struct("functor")
functor("a",1)
```

- In modo speculare a quanto visto per funtori e termini composti, il DSL offre l'oggetto *Predicate* con cui è possibile inizializzare dei predicati che a loro volta sono utilizzabili per la creazione di fatti.

- Sono supportate le regole, che possono essere definite a partire da un fatto chiamando il metodo :- e passando ad esso un numero variabile di fatti (previsto un errore in assenza di argomenti). Il fatto su cui viene chiamato il metodo costituisce la testa della regola, mentre gli argomenti corrispondono al suo corpo. La notazione infissa disponibile in Scala rende naturale la definizione di regole, ma è importante ricordare che si tratta di una chiamata di metodo e quindi, se il corpo della regola contiene più fatti, è necessario racchiuderli tra parentesi.

```
val s = Struct("s")
val sum = Predicate("sum")
val mul = Predicate("mul")

sum("X",0,"X")
sum("X",s("Y"),s("Z")) :- sum("X","Y","Z")
mul("X",0,0)
mul("X",s("Y"),"Z") :- (mul("X","Y","W"), sum("X","W","Z"))
```

- Gli elementi descritti finora consentono di definire un programma Prolog, che è costituito da una serie di clausole corrispondenti alla teoria (che possono essere fatti o regole) e da un insieme di fatti che rappresentano i goal. Per la loro risoluzione il DSL fornisce un oggetto *Engine*, che costituisce il motore logico condiviso nell'applicazione.

- *Engine* consente di aggiungere e resettare la teoria del programma, tramite i metodi *addTheory* e *resetTheory*, e fornisce due modalità di risoluzione alternative (le stesse offerte da tuProlog), grazie ai metodi *solve* e *solveAll*, con cui è possibile risolvere uno o più goal. Nella prima modalità la computazione procede finché non viene trovata una soluzione e, in quel momento, l'utente può scegliere se interrompere l'esecuzione o proseguire fino alla soluzione successiva. Nella seconda modalità, invece, sono esplorate tutte le possibili alternative. È inoltre possibile abilitare la stampa dell'albero di esecuzione del programma.
- Come si può facilmente intuire, ogni chiamata di *solve* o *solveAll* corrisponde all'esecuzione di un programma Prolog. Per ognuna di esse *Engine* porta a termine un procedimento di validazione dell'input, che prevede la mancata esecuzione del programma in caso di errori nei suoi elementi (variabili, termini costanti, termini composti, fatti o regole) e la restituzione di un report contenente tutti gli errori trovati.

Per utilizzare il DSL è sufficiente specificare gli statement *import prologz.dsl._*, con cui si ottengono tutti i costrutti visti in precedenza, *import prologz.dsl.ClauseImplicits._* e *import prologz.dsl.TermImplicits._*, che mettono a disposizione le conversioni implicite necessarie per la definizione di termini, fatti e regole.

2.2 Implementation details

Il codice realizzato è suddiviso in due package: *prologz.dsl*, che contiene la definizione di tutti gli elementi di Prolog supportati dal DSL e gli strumenti messi a disposizione degli utilizzatori del DSL, e *prologz.resolution*, destinato a ciò che è necessario per la risoluzione dei programmi Prolog.

2.2.1 Program validation

Per quanto riguarda i costrutti di Prolog supportati dal DSL, si è scelto di modellare i termini come case class che definiscono dei sottotipi del trait *Term*: *Atom[A]* per i termini costanti e i numeri (interi o decimali), *Variable* per le variabili e *Struct* per i termini composti. Allo stesso modo, le case class *Fact* e *Rule* corrispondono ai fatti e alle regole ed estendono il trait *Clause*. Funtori e predicati sono invece rappresentati tramite stringhe taggate, rispettivamente *String @@ Functor* e *String @@ Predicate*, poiché si tratta semplicemente di elementi che devono essere in grado di incapsulare una stringa.

Le modalità standard per la costruzione di valori di questi tipi sono però nascoste agli utilizzatori del DSL, che hanno a disposizione solo quanto visto nella sezione precedente per definire gli elementi del programma Prolog. Il motivo di questa scelta è evidente se si considerano le definizioni dei metodi *addTheory*, *solve* e *solveAll* presenti in *Engine*: essi accettano rispettivamente *PzValidation[Clause]** e *PzValidation[Fact]** (invece di *Clause** e *Fact**).

PzValidation[A] è un semplice type alias che corrisponde a *ValidationNel[String @@ InputError, A]*, dove String @@ InputError è un tipo usato per rappresentare gli errori nella definizione del programma Prolog. Tutti gli elementi presenti al suo interno, a partire dai più semplici come termini costanti e variabili, sono creati come istanze di *PzValidation[A]* (A è il tipo che rappresenta quell'elemento) e possono quindi incapsulare errori nella loro definizione.

In questo modo, sfruttando la possibilità di accumulare gli errori offerta dal costrutto *Validation* e la sintassi proposta dall'*Applicative Builder* per gestire i casi di successo, è stato realizzato il meccanismo di validazione dell'input descritto in precedenza. In caso di errori viene semplicemente restituito un report, evitando di lanciare eccezioni e di interrompere l'esecuzione.

Un altro vantaggio si ha nel caso di più programmi Prolog da eseguire: se sono presenti errori in alcuni di essi, gli altri vengono risolti normalmente.

```
implicit def fromString(name: String): PzValidation[Term] = {
  val nameVal1: PzValidation[String] =
    if(name.nonEmpty) name.successNel
    else InputError("An empty string is not valid ...").failureNel
  val nameVal2: PzValidation[String] =
    if(name.toCharArray.forall(_.isLetter)) name.successNel
    else InputError(s"String '$name' is not valid ...").failureNel
  (nameVal1 |@| nameVal2)((name, _) =>
    if(name.charAt(0).isLower) Atom(name) else Variable(name))
}

def validateProgram(theory: List[PzValidation[Clause]],
  goals: List[PzValidation[Fact]]):
  PzValidation[(List[Clause], List[Fact])] =
(theory.foldLeft(List.empty[Clause].successNel[String@@InputError])
((accumulator, element) => (accumulator |@| element)
((acc, el) => acc :+ el))
|@| goals.foldLeft(List.empty[Fact].successNel[String@@InputError])
((accumulator, element) => (accumulator |@| element)
((acc, el) => acc :+ el)))((_, _))
```

CODICE 2.1: Metodi per la creazione di *PzValidation[Term]* e per la concatenazione di *PzValidation[Clause]* e di *PzValidation[Fact]*

2.2.2 Prolog execution

L'esecuzione dei programmi Prolog è basata sulla costruzione dell'SLD tree, un albero in cui ogni nodo corrisponde ad una coppia contenente sia i goal che devono ancora essere risolti sia una sostituzione che rappresenta una soluzione valida per i goal che sono stati risolti fino a quel momento.

Nella programmazione logica, una sostituzione è definita come una mappa finita da variabili a termini e, quindi, si è scelto di rappresentarla tramite l'alias *type Substitution = Map[Variable, Term]*. Il meccanismo di risoluzione di Prolog prevede che più sostituzioni debbano poter essere unite in un'unica sostituzione avente come effetto l'esecuzione sequenziale delle varie sostituzioni considerate: dati due goal R1 e R2 risolti rispettivamente dalle sostituzioni A e B, la loro soluzione sarà rappresentata da una sostituzione AB con effetto equivalente all'applicazione di A seguita dall'applicazione di B.

La politica di combinazione delle mappe offerta da Scalaz con l'implementazione di *Monoid[Map]* non è adatta alle sostituzioni, perché le unisce semplicemente e sovrascrive i valori che compaiono più volte. Ad esempio, date due sostituzioni A={X/2,Y/X} e B={X/3}, si ottiene come risultato AB={Y/X,X/3} mentre la sostituzione equivalente è AB={X/2,Y/3}.

Perciò è stata definita una nuova implementazione di *Monoid[Substitution]*, in cui aggiungere una sostituzione ad un'altra equivale ad applicarla a tutti i termini nelle relazioni variabile/termine della prima sostituzione. In questo modo è sufficiente usare l'operatore `|+|` per unire le sostituzioni ma, affinché il procedimento funzioni, è necessario che la sostituzione inziale sia un'identità in cui ogni variabile presente nei goal è associata a sé stessa.

```
implicit object SubstitutionMonoid extends Monoid[Substitution] {
 override val zero:Substitution= Substitution()
 override def append(s1:Substitution,s2:Substitution):Substitution=
  s1.keySet.zip(s1.values.toList.substitute(s2)).toMap
}
```

CODICE 2.2: Definizione di *SubstitutionMonoid*, il metodo *substitute* consente (utilizzando un implicito) di applicare una sostituzione ad una lista di termini

Dopo aver stabilito come rappresentare e unire le sostituzioni, è stato possibile implementare il meccanismo di risoluzione. Esso prevede che tutti i programmi Prolog siano convertiti in alberi SLD, realizzati e gestiti sfruttando le strutture dati *Tree* e *TreeLoc* offerte da Scalaz. Per ogni nodo dell'albero, oltre ai goal da risolvere e alla sostituzione eseguita, è memorizzata una lista di clausole che rappresentano la parte di teoria non ancora considerata.

Inizialmente l'albero di risoluzione è costituito da un solo nodo contenente la teoria del programma, i goal e una sostituzione identità. In seguito questo nodo, quello attualmente selezionato nel *TreeLoc* dell'albero, viene passato ad una procedura ricorsiva (implementata dal metodo *navigatePrologTree* e richiamata grazie a *searchPrologTree*) che si comporta come segue:

- Se la teoria e i goal non sono vuoti, la prima clausola presente nella teoria viene rimossa dal nodo attuale perché, al termine della chiamata in corso, non dovrà più essere considerata. Si esegue quindi l'unificazione (disponibile grazie al metodo *unify*) tra questa clausola e il primo goal.

  Se ha successo, l'unificazione restituisce la sostituzione corrispondente alla soluzione e i goal rimanenti: nel caso di un fatto si tratta degli altri goal dopo l'applicazione della sostituzione, mentre nel caso di una regola è considerato anche il suo corpo. Viene quindi aggiunto un nodo all'albero (ultimo figlio del nodo corrente) contenente tutte le clausole di teoria previste dal programma (possono essere utilizzate di nuovo tutte per risolvere gli altri goal), i goal rimanenti e l'unione tra la sostituzione presente nel nodo corrente e quella ottenuta con l'unificazione. La procedura è chiamata ricorsivamente su questo nodo.

  Al contrario, se l'unificazione non ha successo, la procedura viene chiamata ricorsivamente sul nodo corrente. In questo modo, è possibile tentare di unificare il primo goal con tutte le clausole della teoria.

- Se la teoria è vuota ma devono ancora essere risolti uno o più goal ed è definito il padre del nodo corrente, è facile capire che quest'ultimo rappresenta una foglia in cui non è stata trovata nessuna soluzione. Perciò, viene eseguito automaticamente il *backtracking* e la procedura è richiamata ricorsivamente sul nodo padre del nodo corrente.

- Se non si verificano i due casi precedenti, il nodo corrente può rappresentare una foglia contenente una soluzione valida o la radice (e quindi la fine della computazione). La procedura termina restituendo in entrambi i casi il nodo corrente e offre così la possibilità di interrompere la computazione ogni volta che viene trovata una soluzione.

- Il metodo *searchPrologTree* permette di riprendere l'esplorazione dell'albero di risoluzione da qualsiasi punto, eseguendo il *backtracking* se il nodo corrente non è la radice. La sua esecuzione ripetuta consente di trovare tutte le soluzioni disponibili per un programma Prolog.

```
def searchPrologTree(theory: List[Clause],
  tree: TreeLoc[(List[Clause],List[Fact],Substitution)])
  : TreeLoc[(List[Clause],List[Fact],Substitution)] =
 if(tree.isRoot) navigatePrologTree(theory, tree)
 else navigatePrologTree(theory, tree.parent.get)
 // backtracking (leaf node with valid solution)

@scala.annotation.tailrec
private def navigatePrologTree(theory: List[Clause],
  tree: TreeLoc[(List[Clause],List[Fact],Substitution)])
  : TreeLoc[(List[Clause],List[Fact],Substitution)] =
 tree.getLabel match {
  case (clause :: otherClauses, goal :: otherGoals, subs) =>
   var currentNode: TreeLoc[(List[Clause],List[Fact],Substitution)]
    = tree.setLabel(otherClauses, goal :: otherGoals, subs)
   currentNode = clause.unify(goal, otherGoals).map(res =>
    currentNode.insertDownLast((theory, res._2, subs |+| res._1)
    .leaf)).getOrElse(currentNode)
   navigatePrologTree(theory, currentNode)
  case (_, _ :: _, _) if tree.parent.isDefined =>
   navigatePrologTree(theory, tree.parent.get)
   // automatic backtracking (leaf node without valid solution)
  case (_, goals, subs) => tree.setLabel(Nil, goals, subs)
   // valid solution (leaf node) or execution completed (root node)
 }
```

CODICE 2.3: Costruzione dell'albero di risoluzione Prolog

2.2.3 Unification

Finora l'unificazione è stata vista come un procedimento "black box", che confronta un fatto o una regola (ossia una clausola appartenente alla teoria) con un altro fatto (goal) e, in caso di successo, restituisce una sostituzione corrispondente al risultato e i goal che devono ancora essere risolti.

L'implementazione dell'unificazione è contenuta nel metodo *unify*, disponibile grazie agli impliciti (in modo simile a quanto fatto da Scalaz con la *method injection*) su qualunque clausola. In base al tipo di clausola (fatto o regola), il procedimento è differente. In entrambi i casi è molto importante il ruolo dei goal restanti da eseguire, che sono modificati in base al risultato dell'unificazione. Perciò si è scelto di implementare l'unificazione basandosi sul trasformatore della state monad, *StateT*, in cui lo stato è costituito dai goal che non sono ancora stati risolti ed è prevista l'opzionalità.

L'unificazione di due fatti prevede le seguenti operazioni:

1. Controllare se il fatto e il goal sono caratterizzati dallo stesso nome e dalla stessa arità e, in caso contrario, restituire un esito negativo.

2. Rinominare le variabili presenti nel fatto appartenente alla teoria, in modo che non vi siano né variabili comuni al goal con cui deve essere unificato né variabili comuni agli altri goal da risolvere. Questo perché se una variabile compare due volte nei goal (anche in goal differenti), deve essere associata ad uno stesso termine. Si tratta però di un vincolo che deve poter essere espresso solo in fase di definizione dei goal, e che non può derivare dall'unificazione con una clausola della teoria.

3. Unificare gli argomenti del fatto e del goal, seguendo le regole previste da Prolog per l'unificazione di termini. In caso di successo, mantenere la sostituzione ed eliminare il goal corrente da quelli da risolvere.

4. Applicare la sostituzione trovata ai goal che devono ancora essere risolti e restituire una tupla contenente sia questi sia la sostituzione. È importante notare che, grazie al comportamento *first-fail* di *StateT*, questa operazione non viene eseguita se l'unificazione ha esito negativo.

Unificare una regola con un fatto è invece un procedimento leggermente più complesso, che è caratterizzato dalle seguenti operazioni:

- Replicare i primi tre passi previsti per l'unificazione di fatti. In particolare: eseguire il controllo sulla testa della regola e sul goal, rinominare sia la testa sia il corpo della regola evitando le variabili contenute nei vari goal e unificare gli argomenti della testa della regola e del goal.

- Se l'unificazione ha successo, aggiungere ai goal da risolvere i fatti presenti nel corpo della regola. Prima è però necessario rinominare le variabili presenti nel corpo della regola ma non nella sua testa (le variabili in testa sono già state considerate), facendo in modo che non compaiano anche negli altri goal. Si evita così l'aggiunta di vincoli inesistenti che possono precludere la corretta risoluzione del programma.

- Infine, come nel punto 4 visto per l'unificazione di fatti, si applica la sostituzione trovata ai goal che devono ancora essere risolti e viene restituita una tupla contenente sia questi sia la sostituzione.

2.3 Final remarks

Il numero di funzionalità presenti in Prolog e supportate dal DSL è ancora limitato, ma ritengo che possa costituire una base valida per estensioni che supportino altre funzionalità. Ad esempio, non dovrebbe essere eccessivamente complicato aggiungere il supporto alle liste e l'operatore "'!".

Attualmente l'interazione prevista consente soltanto di visualizzare a riga di comando i risultati dei programmi risolti. Sarebbe però semplice modificare l'interfaccia presente in *Engine* e ottenere le soluzioni sotto forma di valori tipati, in modo da facilitare l'integrazione con altre applicazioni (ad esempio, con giochi che definiscono la propria logica utilizzando Prolog).

A mio parere, Scalaz si è rivelato un ottimo strumento ed ha rappresentato un valido ausilio per la realizzazione di questo progetto. Le strutture funzionali che offre hanno infatti consentito di implementare meccanismi complessi in modo ancor più naturale e semplice di quanto possibile in Scala.

Come visto in precedenza, nell'implementazione del DSL sono state fondamentali anche le caratteristiche messe a disposizione nativamente da Scala, quali ad esempio case class, pattern matching e conversioni implicite. In particolare, grazie a queste ultime è stato possibile consentire di scrivere in Scala programmi molto simili a quelli Prolog ed è stata evidente l'importanza ricoperta da questi strumenti nella realizzazione di un qualsiasi DSL.

Infine, i test realizzati con ScalaTest (e presenti nel package *test.scala.prologz*) hanno consentito di verificare il funzionamento di quanto implementato e costituiscono una valida specifica del comportamento del sistema.

Bibliografia

[1] *Scalaz*. URL: https://github.com/scalaz/scalaz.

[2] Eugene Yokota. *Learning Scalaz*. URL: http://eed3si9n.com/learning-scalaz/index.html.

[3] Sam Halliday. *Functional Programming for Mortals with Scalaz*. Leanpub, 2018. URL: https://leanpub.com/fpmortals.

[4] *Polymorphism (computer science)*. URL: https://en.wikipedia.org/wiki/Polymorphism_(computer_science).

[5] *Towards Scalaz (Part 1)*. URL: https://typelevel.org/blog/2013/10/13/towards-scalaz-1.html.

[6] *Functors, Applicatives, And Monads In Pictures*. URL: http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html.

[7] *Scalaz for Dummies*. URL: https://medium.com/bench-engineering/scalaz-for-dummies-5289f2d90352.

[8] *Accumulating More Than One Failure In A ValidationNEL*. URL: https://johnkurkowski.com/posts/accumulating-multiple-failures-in-a-validationnel/.

[9] *Tag Types*. URL: http://dcapwell.github.io/scala-tour/Tag%20Types.html.

[10] *Scalaz State Monad*. URL: https://softwarecorner.wordpress.com/2013/08/29/scalaz-state-monad/.