Azzolini Riccardo 2021-05-31

For expression e mappe associative

1 Repeated parameter

Similmente al meccanismo dei tre punti (`...`) di Java, Scala permette di indicare che l'ultimo argomento di una funzione o di un costruttore può essere ripetuto, scrivendo `*` dopo il tipo del parametro; ad esempio:

```
case class Person(name: String, isMale: Boolean, children: Person*)
```

Tale parametro prende il nome di **repeated parameter**, e permette all'utente di fornire una lista di argomenti variabile; ad esempio:

```
val lara = Person("Lara", false)
val bob = Person("Bob", true)
val julie = Person("Julie", false, lara, bob)
```

Nel codice, un repeated parameter di tipo `T` è accessibile come una sequenza di tipo `Seq[T]`.

2 For expression

Data la lista di persone

```
val persons = List(lara, bob, julie)
```

si vuole costruire la lista di tutte le coppie di nomi (*madre*, *figlio/a*) per tutte le madri contenute in `persons`. Ciò può essere fatto usando i metodi all'ordine superiore di `List` (ovvero di `Seq`):

1. si filtra la lista `persons` per escludere i maschi, che non possono essere madri;
2. si usa `map` per trasformare la sequenza di figli di una persona in una sequenza di coppie di nomi di madre e figlio/a, e ciò viene fatto per ogni persona nella lista filtrata usando `flatMap`, in modo da ottenere complessivamente una singola sequenza di coppie invece che una sequenza di sequenze.

```
persons filter (p => !p.isMale) flatMap (p =>
  p.children map (c => (p.name, c.name))
)  // res0: List[(String, String)] = List((Julie,Lara), (Julie,Bob))
```

Quest'esempio mostra che i metodi all'ordine superiore, seppur estremamente potenti, non sempre sono semplici da scrivere o da comprendere: la semantica delle operazioni effettuate qui è un po' nascosta dall'interazione non banale tra i metodi `filter`, `flatMap` e `map` coinvolti nel presente frammento di codice.

L'uso dei metodi all'ordine superiore tende a diventare più facile con l'abitudine, ma comunque Scala fornisce dello zucchero sintattico che permette di esprimere in una forma più chiara combinazioni di `map`, `flatMap` e `filter`: le for expression. Una **for expression** ha la forma generale

```
for (s) yield e
```

oppure

```
for {s} yield e
```

dove:

- *s* è una sequenza di *generatori*, *definizioni* e *filtri*, che devono essere separati da `;` (punto e virgola) nella versione della sintassi con le parentesi tonde, mentre nella versione con le parentesi graffe gli possono anche essere separati scrivendone uno per riga di testo, e in tal caso i `;` sono opzionali;
- *e* è l'espressione il cui valore è restituito da una singola iterazione della for expression.

Gli elementi che possono comparire nella sequenza *s* sono fatti in questo modo:

- Un **generatore** ha la forma *p* `<-` *e*, dove *p* è un pattern qualsiasi (in particolare può essere una variabile) ed *e* è un'espressione il cui valore è una collezione. In fase di esecuzione, *p* varia su tutti gli elementi della collezione ottenuta dalla valutazione di *e*.
- Una **definizione** ha la forma *p* `=` *e*, dove *p* è un pattern ed *e* è un'arbitraria espressione. In fase di esecuzione, il nome o i nomi (nel caso di un pattern che non è una semplice variabile) di *p* vengono legati al valore o alle parti del valore di *e* (in pratica si effettua un assegnamento strutturato, corrispondente alla sintassi `val` *p* `=` *e* che si usa fuori dalle for expression).
- Un **filtro** ha la forma `if` *f*, dove *f* è un'espressione booleana. In fase di esecuzione vengono considerate solo le combinazioni di valori dei precedenti generatori per cui la condizione *f* è vera.

Si noti che il primo elemento di *s* deve obbligatoriamente essere un generatore (non può essere una definizione o un filtro). I nomi definiti da un elemento (generatore o definizione) di *s* possono poi essere usati negli elementi successivi.

Il risultato di una for expression è una collezione che contiene i valori ottenuti dalla valutazione dell'espressione *e* su ciascuna delle combinazioni di valori generate dall'iterazione su *s*.

Ad esempio, la generazione delle coppie (*madre*, *figlio/a*) può essere riscritta come

```
for (p <- persons; if !p.isMale; c <- p.children)
  yield (p.name, c.name)
    // res1: List[(String, String)] = List((Julie,Lara), (Julie,Bob))
```

oppure, usando la sintassi con le parentesi graffe,

```
for {
  p <- persons
  if !p.isMale
  c <- p.children
} yield (p.name, c.name)
    // res2: List[(String, String)] = List((Julie,Lara), (Julie,Bob))
```

Queste due for expression (del tutto equivalenti) funzionano in questo modo:

1. generatore `p <- persons`: si considerano tutti gli elementi `p` della lista `persons`;
2. filtro `if !p.isMale`: si selezionano solo gli elementi `p` tali che `p.isMale` è `false`, riducendo così il numero di elementi a cui si applicheranno gli elementi successivi della for expression;
3. generatore `c <- p.children`: per ogni `p` su cui si sta operando (ovvero ogni `p` che ha "superato" il precedente filtro) si considerano tutti gli elementi `c` della lista dei suoi figli, `p.children`;
4. `yield (p.name, c.name)`: per ogni `p` e per ogni `c` selezionati si produce come risultato la coppia `(p.name, c.name)`.

Più in sintesi:

1. per ogni persona `p`
2. che non è di sesso maschile
3. per ogni figlio/a `c` di tale persona
4. genera la coppia dei nomi di `p` e `c`.

2.1 Altri esempi

Data ancora la lista di persone

```
val persons = List(lara, bob, julie)
```

la seguente for expression genera la lista dei nomi che iniziano con `"B"`:

```
for {
  p <- persons
  n = p.name
  if n startsWith "B"
} yield n  // res3: List[String] = List(Bob)
```

1. per ogni persona `p`
2. lega a `n` il nome di tale persona
3. se il nome `n` inizia con `"B"`
4. genera il nome `n` come risultato di un'iterazione.

Un esempio di for expression usata invece sui range è la generazione delle coppie non ordinate di interi compresi tra 1 e `n` la cui somma è un numero primo:

```
def isPrime(n: Int): Boolean =
  (2 until n) forall (d => n % d != 0)

val n = 5
for {
  i <- 1 to n
  j <- 1 to i
  if isPrime(i + j)
} yield (i, j)
  // res4: IndexedSeq[(Int, Int)] =
  //   Vector((1,1), (2,1), (3,2), (4,1), (4,3), (5,2))
```

1. per ogni `i` compreso tra 1 e `n`
2. per ogni `j` compreso tra 1 e `i` (considerando solo i `j` minori o uguali a ciascun `i` si evita di generare coppie non ordinate uguali, cioè coppie in cui cambia solo l'ordine degli elementi)
3. se la somma di `i` e `j` è un numero primo
4. genera la coppia `(i, j)`.

Infine, l'esempio del prodotto scalare visto in precedenza

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  (xs zip ys).map({case (x, y) => x * y}).sum
```

può essere riscritto mettendo al posto di `map` la seguente for expression, che illustra l'uso di un generatore con un pattern più complesso di una singola variabile, impiegato per fare l'assegnamento strutturato sulle coppie restituite da `zip`:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  (for ((x, y) <- xs zip ys) yield x * y).sum
```

(in questo caso, però, la versione con `map` è forse più leggibile).

2.2 Traduzione

Le for expression vengono tradotte in applicazioni dei metodi `map`, `flatMap` e `withFilter`,[1] dunque possono essere utilizzate non solo per le collezioni, bensì per ogni tipo che fornisce tali metodi con i prototipi corretti, come ad esempio il seguente tipo C:

```
abstract class C[A] {
  def map[B](f: A => B): C[B]
  def flatMap[B](f: A => C[B]): C[B]
  def withFilter(p: A => Boolean): C[A]
}
```

Alcuni esempi di tipi che supportano le for expression (nella libreria standard e in librerie fornite da terzi) sono liste, array, iteratori, insiemi, ma anche database, dati XML, parser, ecc.

3 Mappe associative

Una **mappa** (**associativa**) `Map[K,+V]`[2] è una struttura dati che associa chiavi di tipo `K` a valori di tipo `V`. Si osservi che `Map` è invariante sul tipo `K` e covariante sul tipo `V`.

I valori di `Map` possono essere costruiti usando il metodo factory fornito dal companion object, che riceve come argomenti un numero qualsiasi di coppie chiave-valore (anche zero, per creare una mappa vuota):

```
val romanNums = Map(("I", 1), ("V", 5), ("X", 10))
val capitals = Map(("Italy", "Rome"), ("France", "Paris"))
```

Al fine di rendere più leggibile l'uso del costruttore e di evidenziare il fatto che una mappa associativa modella sostanzialmente una funzione dalle chiavi ai valori, Scala mette a disposizione un operatore `->` come sintassi alternativa per specificare le coppie: *`a -> b`* è esattamente equivalente ad *`(a, b)`*. Con tale sintassi, gli esempi precedenti diventano:

```
val romanNums = Map("I" -> 1, "V" -> 5, "X" -> 10)
val capitals = Map("Italy" -> "Rome", "France" -> "Paris")
```

[1] Il metodo `withFilter` delle collezioni fornite dalla libreria standard di Scala (altre classi potrebbero implementarlo diversamente) è una variante *lazy* di `filter`: invece di creare la collezione filtrata in memoria, esso restituisce un oggetto che permette l'applicazione diretta di alcune operazioni come `map` e `flatMap` ai soli elementi della collezione originale che soddisfano il predicato passato a `withFilter`.

[2] `Map` è un trait definito nel package `scala.collection.immutable`, ed è disponibile senza importazione tramite un alias presente nel package `scala`.

L'operatore `->` è usato anche nella rappresentazione testuale delle mappe restituita da `toString`, dunque l'output dell'interprete corrispondente a queste definizioni è:

```
romanNums: scala.collection.immutable.Map[String,Int] =
  Map(I -> 1, V -> 5, X -> 10)
capitals: scala.collection.immutable.Map[String,String] =
  Map(Italy -> Rome, France -> Paris)
```

3.1 Operatori + e ++

`Map` fornisce vari operatori, tra cui quelli per l'aggiunta di elementi e per la concatenazione di mappe:

- *m* `+ (`*k* `->` *v*`)` aggiunge alla mappa *m* la coppia chiave-valore *k* `->` *v*, rimpiazzando il valore precedente se la chiave *k* era già presente.
- *m* `+ (`k_1 `->` v_1`,` k_2 `->` v_2`, ...)`[3] aggiunge alla mappa *m* le coppie chiave-valore specificate, rimpiazzando eventuali valori associati a chiavi già presenti.
- *m* `++` *kvs* restituisce la mappa costruita aggiungendo a *m* tutte le coppie chiave-valore contenute in *kvs*, dove *kvs* è di tipo `Iterable`, cioè può essere una mappa ma anche una `Lista` o `Vector` di coppie, ecc. Anche in questo caso eventuali valori associati a chiavi già presenti in *m* vengono rimpiazzati con i valori forniti da *kvs*.

3.2 Metodi `apply` e `get`

Per ottenere il valore associato a una chiave in una mappa si può usare il metodo `apply`, cioè applicare la mappa come una funzione passando come argomento la chiave (dato che le mappe associative modellano appunto funzioni dalle chiavi ai valori):

```
val capitals = Map("Italy" -> "Rome", "France" -> "Paris")
capitals("Italy")  // res0: String = Rome
```

Se la chiave passata come argomento non è presente nella mappa (cioè se l'argomento non è nel dominio di definizione della funzione rappresentata dalla mappa) viene sollevata una `NoSuchElementException`:

```
capitals("UK")  // java.util.NoSuchElementException: key not found: UK
```

In alternativa, per gestire senza eccezioni il caso in cui la chiave non esiste si può usare il metodo

```
def get(key: K): Option[V]
```

[3] La versione dell'operatore `+` che accetta più coppie chiave-valore è deprecata a partire da Scala 2.13.0; al suo posto si consiglia di usare `++` con una collezione di coppie, come ad esempio *m* `++ List(`k_1 `->` v_1`,` k_2 `->` v_2`, ...)`.

dove `Option[V]` è il tipo che rappresenta valori opzionali di tipo `V`, precedentemente introdotto nell'ambito degli extractor pattern. Tale metodo restituisce:

- `Some(`v`)` se nella mappa su cui è invocato la chiave `key` è associata a un valore v;
- `None` se la chiave `key` non è presente (non ha un valore associato) nella mappa.

Ad esempio:

```
capitals get "Italy"  // res1: Option[String] = Some(Rome)
capitals get "UK"     // res2: Option[String] = None
```

4 Considerazioni su `Option`

Per quanto sia già stato introdotto, il tipo `Option` merita un approfondimento. Innanzitutto, esso è definito come:

```
sealed abstract class Option[+A]
case class Some[+A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

Siccome i suoi sottotipi sono classi `case`, è possibile operare su valori di tipo `Option` tramite pattern matching; ad esempio:

```
def getCapital(country: String): String = capitals get country match {
  case Some(capital) => capital
  case None => "undefined country " + country
}

getCapital("Italy")  // res3: String = Rome
getCapital("UK")     // res4: String = undefined country UK
```

In Java, per rappresentare valori opzionali di tipo T si usa tipicamente il tipo T stesso, sfruttando il valore `null` per rappresentare il caso in cui non è presente un valore. Ciò costituisce una possibile sorgente di errore: se si invoca un metodo su un riferimento senza prima ricordarsi di controllare che esso non sia nullo, si rischia il sollevamento di una `NullPointerException`. L'uso di `Option[`T`]` risolve questo problema: il valore di tipo T contenuto all'interno di un valore `Option[`T`]` deve essere estratto esplicitamente dal programmatore (perché non esiste una conversione implicita da `Option[`T`]` a T), e nel fare ciò (tramite pattern matching o vari metodi di accesso) si decide anche come trattare il caso `None`, in cui il valore non è presente.

Dato un valore `opt: Option[A]`, alcuni metodi di classificazione e accesso disponibili su di esso sono:

- `opt.isDefined`, che restituisce `true` se e solo se il valore di `opt` è istanza di `Some[A]` (ovvero se e solo se il valore opzionale è presente);

- `opt.get`, che restituisce il valore di tipo `A` contenuto in `Some[A]` se `opt` è appunto un'istanza di `Some[A]`, mentre solleva una `NoSuchElementException` se il valore di `opt` è `None`;
- `opt getOrElse default`, che restituisce il valore di tipo `A` contenuto in `Some[A]`, oppure restituisce il valore di `default` (anch'esso di tipo `A`, o di un suo supertipo) se il valore di `opt` è `None`.

`Option[A]` può essere visto come un particolare tipo di collezione che può essere vuota o contenere un solo elemento. Perciò, anche se la classe `Option` non appartiene alla gerarchia delle collezioni Scala, essa fornisce molti dei metodi delle collection, tra cui ad esempio l'operatore di concatenazione `++` (che restituisce un `Iterable`, in particolare un'istanza di `List`, perché `Option` la concatenazione di due `Option` deve poter contenere più di un singolo valore) e metodi come `map`, `flatMap`, `filter` / `withFilter` (per cui `Option` supporta le for expression), ecc. Ad esempio:

```
capitals.get("Italy") ++ capitals.get("France")
  // res5: Iterable[String] = List(Rome, Paris)
capitals.get("Italy").map(_.length)
  // res6: Option[Int] = Some(4)
```