Azzolini Riccardo 2021-05-27

Collection nella libreria standard di Scala

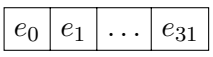
1 `Vector`

Le liste sono la struttura dati "fondamentale" nei linguaggi funzionali, ma hanno dei limiti che vanno considerati quando bisogna scegliere se usarle nelle applicazioni. Il problema principale è che esse sono strutture lineari: per accedere all'elemento in posizione n è necessario scorrere i primi n elementi della lista. Perciò, il package `scala.collection.immutable` fornisce un'altra struttura dati immutabile, la classe generica `Vector[+T]`[1] (covariante nel tipo degli elementi), che implementa sequenze di elementi con operazioni di accesso più efficienti rispetto a quelle su `List`.

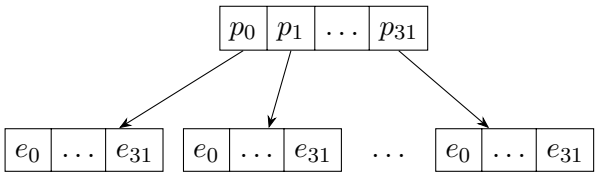
Un `Vector[T]` è rappresentato come un albero nel quale ciascun nodo può contenere al più 32 elementi di tipo `T` oppure i puntatori ad al più 32 nodi dell'albero (ma *non* può contenere un mix di elementi e puntatori). Tale struttura dati non ha un nome standard in letteratura, ma a volte è chiamata **bitmapped vector trie**, e ne esistono diverse varianti.[2]

L'organizzazione dell'albero che rappresenta un `Vector` varia in funzione del numero N di elementi che esso contiene; ad esempio:

- Se $N \leq 32$ l'albero consiste della sola radice, la quale contiene direttamente gli elementi:



- Se $32 < N \leq 32^2$ l'albero ha due livelli:

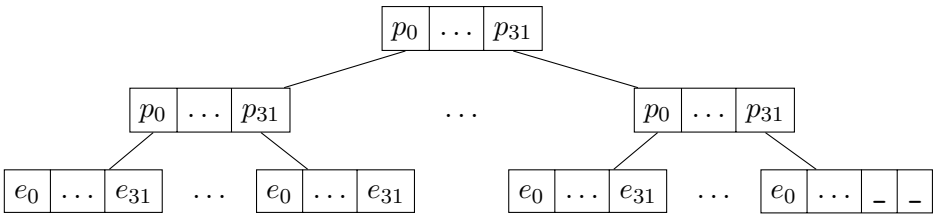


[1] Come `List`, anche `Vector` è disponibile senza bisogno di importarla, tramite un alias definito nel package `scala`.

[2] La struttura descritta qui è quella implementata nella libreria standard di Scala fino alla versione 2.13.1. A partire da Scala 2.13.2, tale struttura è stata sostituita con una variante, chiamata "radix-balanced finger tree" (anche questo non è un nome standard in letteratura), che è un po' più complicata ed efficiente (ma in termini asintotici la complessità della maggior parte delle operazioni rimane uguale alla precedente implementazione).

la radice contiene i puntatori ai massimo 32 nodi di secondo livello, ciascuno dei quali contiene 32 elementi (tranne il primo e l'ultimo nodo, che possono contenere meno elementi se l'albero non è pieno, $N < 32^2$).

- Se $32^2 < N \leq 32^3$ l'albero ha tre livelli:



la radice contiene i puntatori ai massimo 32 nodi di secondo livello, ciascuno dei quali contiene a sua volta i puntatori a 32 nodi di terzo livello, che infine contengono gli elementi (anche in questo caso, il primo e l'ultimo nodo di ciascun livello possono contenere meno di 32 puntatori o elementi).

In generale, l'altezza dell'albero (il numero di livelli) è proporzionale a $\log_{32} N$, dunque pochi livelli sono sufficienti a contenere un'enorme quantità di dati: ad esempio, un `Vector` con 6 livelli può memorizzare $32^6 = 2^{30}$ elementi.

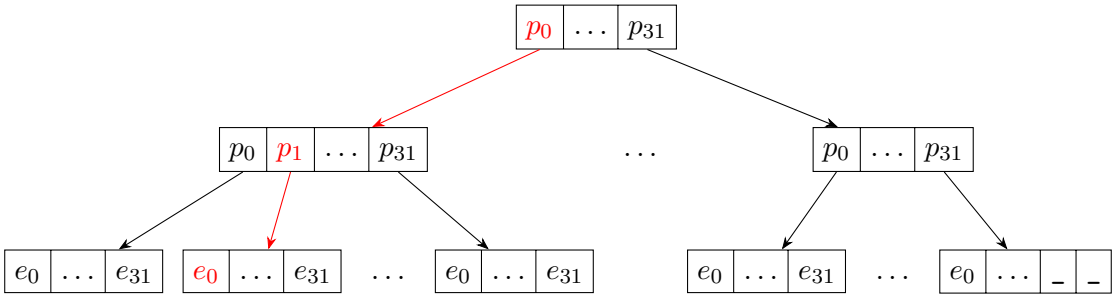
1.1 Accesso

L'accesso a un elemento di un `Vector` richiede un numero di livelli di indirettezza pari al numero di livelli dell'albero meno uno.

Ad esempio, l'accesso a un elemento di indice n in un `Vector` rappresentato da un albero a 3 livelli richiede due livelli di indirettezza, perché avviene nel seguente modo:

1. (primo livello di indirettezza) Dalla radice si raggiunge il nodo di secondo livello che è radice del sottoalbero contenente l'elemento cercato tramite il puntatore situato all'indice $n \,/\, 32^2$ della radice. Infatti, siccome ciascun sottoalbero della radice contiene 32^2 elementi, l'elemento si trova all'interno del sottoalbero di indice $n \,/\, 32^2$, e in tale albero ha l'indice $n \,\%\, 32^2$.
2. (secondo livello di indirettezza) Da questo nodo di secondo livello si raggiunge il nodo di terzo livello contenente l'elemento cercato tramite il puntatore di indice $(n \,\%\, 32^2) \,/\, 32$ (in base a un ragionamento analogo a quello del passo precedente, con la differenza che qui i sottoalberi contengono 32 elementi ciascuno).
3. Infine, in questo nodo di terzo livello si accede direttamente all'elemento di indice $(n \,\%\, 32^2) \,\%\, 32$, che è quello cercato.

Come esempio specifico, si supponga di voler accedere all'elemento in posizione $n = 32$:



1. l'indice del nodo di secondo livello è $n \;/\; 32^2 = 32 \;/\; 32^2 = 0$, che dalla radice si raggiunge tramite il puntatore p_0;
2. l'indice del nodo di terzo livello è $(n \,\%\, 32^2) \;/\; 32 = (32 \,\%\, 32^2) \;/\; 32 = 32 \;/\; 32 = 1$, che dal nodo di secondo livello si raggiunge tramite il puntatore p_1;
3. l'indice dell'elemento nel nodo di terzo livello è $(n \,\%\, 32^2) \,\%\, 32 = (32 \,\%\, 32^2) \,\%\, 32 = 32 \,\%\, 32 = 0$, cioè l'elemento a cui si vuole accedere è e_0.

Siccome il numero di operazioni necessarie per l'accesso dipende dall'altezza dell'albero, esso è in generale proporzionale a $\log_{32} N$. Ciò significa che i `Vector` hanno buone prestazioni nel caso dell'accesso per indice agli elementi (al contrario delle liste). Inoltre, essi hanno buone prestazioni anche nel caso di operazioni come *map* e *fold* (ma tali operazioni sono efficienti anche sulle liste).

L'accesso è reso veloce anche dal fatto che ogni nodo è rappresentato da una sequenza di 32 puntatori o elementi che sono memorizzati in locazioni di memoria adiacenti, e la dimensione di tale sequenza è paragonabile alla dimensione delle cache-line nei processori moderni, quindi una frazione significativa dei 32 puntatori/elementi (se non addirittura tutti, a seconda del processore) sono presenti nella medesima cache-line. Invece, ciò non vale in generale per le liste, nelle quali ciascun nodo contiene un solo elemento e non è necessariamente adiacente ad altri nodi in memoria.

1.2 Creazione e operazioni

Come la classe `List`, `Vector` ha un companion object che fornisce un costruttore al quale si può passare un numero arbitrario di argomenti per creare `Vector` di qualsiasi lunghezza; ad esempio:

```
val nums = Vector(1, 3, 5, 7)
val fruits = Vector("orange", "apple", "pineapple")
```

I `Vector` forniscono poi le medesime operazioni disponibili per le liste, tranne `::` (cons), che è rimpiazzato dagli operatori `+:` e `:+`, usati rispettivamente per l'inserimento in testa e in coda:

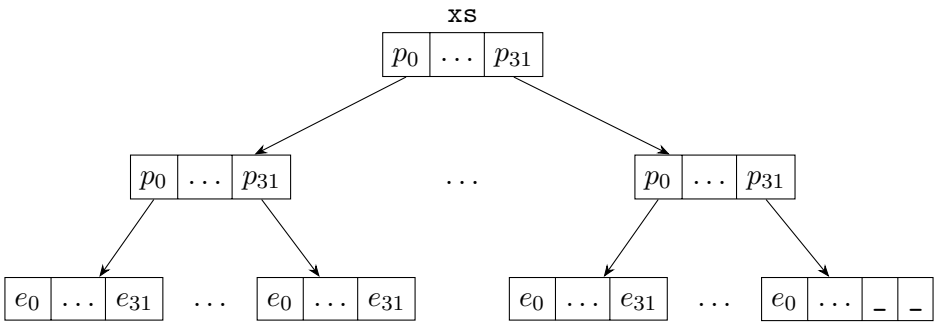
- `x +: xs` crea un nuovo `Vector` con in testa l'elemento `x`, seguito dagli elementi di `xs`;
- `xs :+ x` crea un nuovo `Vector` con in coda l'elemento `x`, preceduto dagli elementi di `xs`.

Per ricordare come si usano questi due operatori ci si può aiutare con la seguente osservazione: l'operando dal lato del carattere `+` è l'elemento da inserire, mentre l'operando dal lato di `:` è il `Vector`. Il fatto che `+:` termini con due punti mentre `:+` no serve a far sì che entrambi questi operatori vengano invocati come metodi sull'operando `Vector` e non sull'elemento da inserire.

1.2.1 Inserimento in coda

Il modo in cui un inserimento in coda `xs :+ x` viene eseguito varia a seconda che l'albero sia *pieno* oppure no.

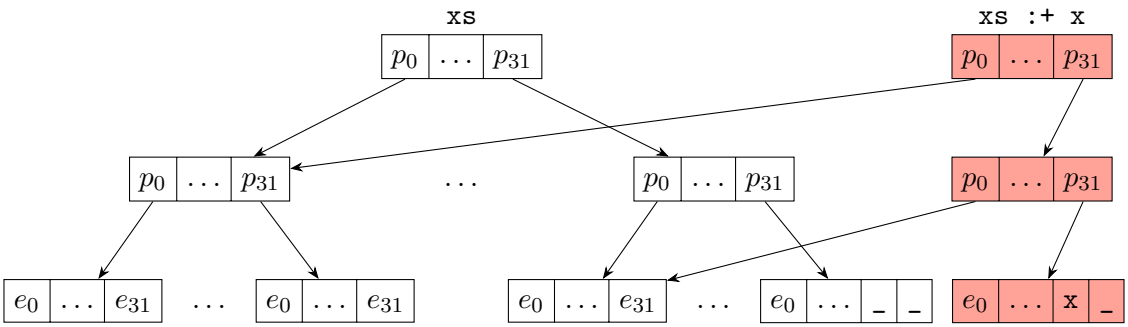
Si supponga che `xs` sia rappresentato dall'albero



il quale non è pieno, ovvero ha una o più posizioni libere (qui indicate da `_`) nell'ultimo livello. Ricordando che `Vector` è immutabile, anche se ci sono posizioni vuote l'inserimento `xs :+ x` richiede la costruzione di un nuovo albero, ma è sufficiente copiare i nodi i cui contenuti cambiano e riutilizzare gli altri. In particolare è necessario creare:

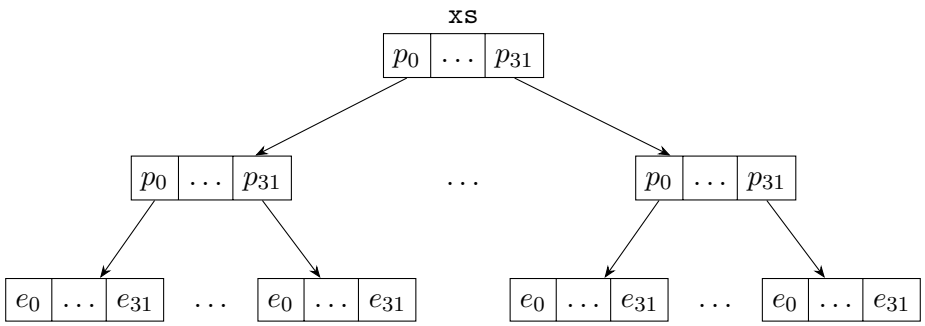
- una copia dell'ultima foglia a destra, nella quale la prima posizione vuota è riempita dall'elemento `x`;
- una copia di ciascun nodo dal padre della foglia alla radice, nella quale si aggiorna solo il puntatore al sottoalbero modificato, mentre tutti gli altri puntatori continuano a puntare ai nodi originali.

La situazione in memoria dopo l'inserimento è dunque la seguente (dove i nuovi nodi sono evidenziati in rosso):



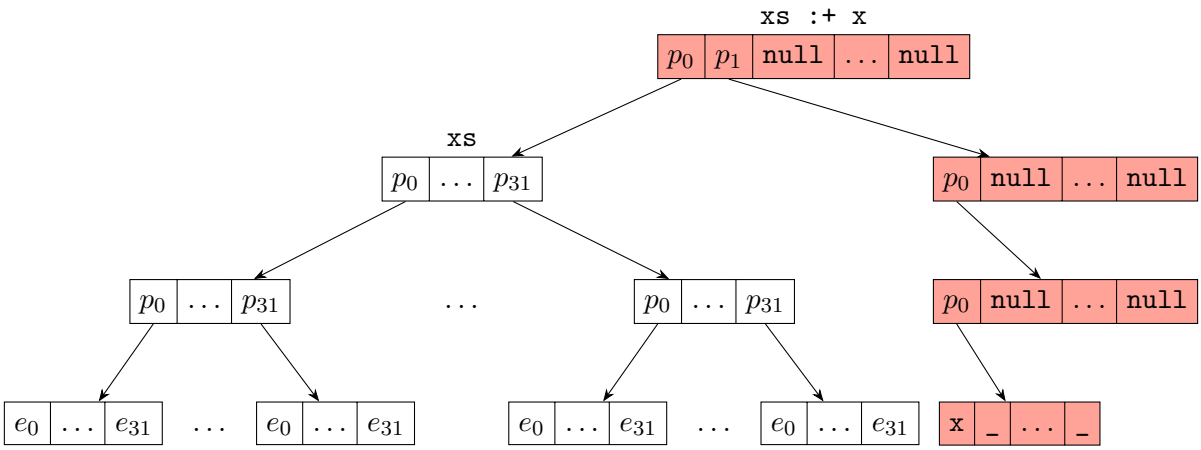
In sintesi, bisogna creare un nuovo nodo per ogni livello dell'albero.

Adesso si supponga invece che `xs` sia rappresentato dall'albero



il quale è pieno (non ha posizioni libere nelle foglie dell'ultimo livello, né posizioni libere nei nodi dei livelli superiori, le quali consentirebbero l'aggiunta di nuove foglie all'ultimo livello). In questo caso l'albero `xs :+ x` si costruisce creando una nuova radice che ha `xs` come primo sottoalbero, mentre il secondo (e ultimo) sottoalbero è costituito da un nuovo nodo per ogni livello, cioè in particolare da:

- una nuova foglia contenente solo l'elemento `x`;
- per ogni livello tra questa foglia e la nuova radice, dei nuovi nodi contenenti solo i riferimenti necessari a raggiungere la foglia.



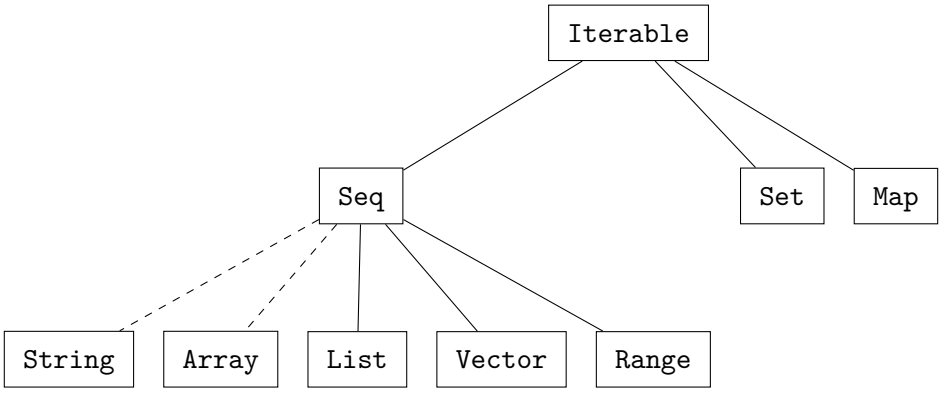
Qui il numero di nodi da creare è uguale al numero di livelli di `xs` più uno, ovvero ancora proporzionale all'altezza di `xs`, come nel caso dell'albero non pieno.

In generale, l'operazione di aggiunta in coda a un `Vector` che contiene N elementi richiede tempo proporzionale a $\log_{32} N$, decisamente inferiore al tempo lineare (proporzionale a N) richiesto per l'aggiunta in coda a una lista. Anche l'aggiunta in testa `x +: xs` è implementata allo stesso modo,[3] e ha quindi la stessa complessità, ma ciò non costituisce un vantaggio rispetto alle liste, che implementano l'inserimento in testa in tempo costante.

2 Gerarchia delle collection

Le collection fornite dalla libreria standard di Scala sono organizzate in una gerarchia, di cui in seguito sono mostrate e descritte alcune delle principali classi/trait:

[3] L'inserimento in testa può essere implementato come quello in coda perché nell'albero sono ammesse posizioni libere non solo a destra ma anche a sinistra. Di conseguenza, potrebbe essere che il primo elemento di un `Vector` si trovi in una posizione di indice maggiore di 0, ma tale indice iniziale viene gestito trasparentemente dall'implementazione di `Vector`, così che per gli utenti gli indici degli N elementi di un `Vector` vadano sempre da 0 a $N - 1$.



- Il trait `Iterable` rappresenta collezioni che è possibile scorrere elemento per elemento, fornendo appunto metodi generali per operare elemento per elemento su una collezione.
- Il trait `Seq` rappresenta i casi particolari di `Iterable` in cui è definito un ordine degli elementi.
- `Set` e `Map` sono trait che rappresentano rispettivamente i concetti astratti di insieme e mappa, dei quali sono poi fornite varie implementazioni. Questi trait estendono direttamente `Iterable`, e non `Seq`, perché negli insiemi e nelle mappe l'ordine degli elementi può non essere definito (a seconda dell'implementazione).
- `List`, `Vector` e `Range` sono alcuni tipi di collezioni ordinate, cioè di `Seq` (`List` e `Vector` sono le classi già viste, mentre `Range`, che rappresenta intervalli numerici, verrà presentata a breve).
- `Array` e `String` non appartengono tecnicamente alla gerarchia delle collection perché sono implementate dalle corrispondenti classi Java, che non sono sottotipi di `Seq`, ma per comodità si possono usare su di esse tutti i metodi di `Seq`, tramite conversioni implicite a oggetti wrapper che forniscono tali metodi.

2.1 Metodi di `Seq`

Tutti i metodi visti per le liste (ad eccezione dei costruttori) sono metodi definiti in `Seq` (se non addirittura in `Iterable`), e quindi sono disponibili per tutte le collezioni che estendono `Seq`. Alcuni altri metodi di `Seq` sono i seguenti:

- `xs exists p`, che restituisce `true` se e solo se `xs` contiene almeno un elemento x per cui il predicato `p(`x`)` è vero;
- `xs forall p`, che restituisce `true` se e solo se `p(`x`)` è vero per ogni elemento x di `xs`;
- `xs zip ys`, che restituisce la sequenza delle coppie degli elementi corrispondenti (situati nelle stesse posizioni) di `xs` e `ys`;

- `xs.unzip`, che estrae da una sequenza di coppie $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ la coppia delle sequenze $(\{x_1, \ldots, x_n\}, \{y_1, \ldots, y_n\})$, cioè svolge l'operazione inversa di `zip`;
- `xs.sum` e `xs.product`, che calcolano rispettivamente la somma e il prodotto degli elementi della sequenza, e sono applicabili solo a sequenze di elementi di tipo numerico;
- `xs.min` e `xs.max`, che individuano il minimo e il massimo valore nella sequenza, usando un parametro implicito di tipo `Ordering[T]` per determinare come ordinare gli elementi (quindi per poter applicare questi metodi a una sequenza è necessario che esista un ordine sul tipo degli elementi della sequenza).

2.2 Array e stringhe come `Seq`

Quando si invoca su un array un metodo di `Seq`, il compilatore inserisce automaticamente una conversione implicita da `Array` a `scala.collection.mutable.ArrayOps`,[4] una classe wrapper che fornisce l'implementazione dei metodi di `Seq` per gli array. Ciò permette di scrivere, ad esempio:

```
val array = Array(1, 3, 5)  // array: Array[Int] = Array(1, 3, 5)
array.head                  // res0: Int = 1
array.tail                  // res1: Array[Int] = Array(3, 5)
array map (2 * _)           // res2: Array[Int] = Array(2, 6, 10)
```

(i commenti indicano l'output dell'interprete).

Per le stringhe esiste un meccanismo analogo, dunque si può ad esempio scrivere:

```
val string = "Hello World"  // string: String = Hello World
string.head                 // res3: Char = H
string.tail                 // res4: String = ello World
string filter (_.isUpper)   // res5: String = HW
```

3 Range

Un `Range` rappresenta una sequenza di valori numerici consecutivi compresi tra ***start*** ed ***end*** e separati da un "passo di scansione" ***step***. Essi sono rappresentati da oggetti con tre campi (***start***, ***end*** e ***step***), cioè in particolare non viene memorizzata l'intera sequenza di valori.

I tipi numerici forniscono due operatori (metodi) per creare oggetti di tipo `Range`:

[4] A partire da Scala 2.13.0 la classe `ArrayOps` si trova nel package `scala.collection` invece che in `scala.collection.mutable`.

- `to`, che crea un range inclusivo, il quale include entrambi gli estremi (*`start` `to` `end`* comprende sia *`start`* che *`end`*);
- `until`, che crea un range il quale include l'estremo inferiore ma esclude quello superiore (*`start` `until` `end`* comprende *`start`* ma non *`end`*).

La classe `Range` fornisce poi un operatore (metodo) `by` che consente di impostare il passo: esso crea un nuovo range con valore iniziale e finale uguali a quelli del range su cui è invocato e con il passo specificato come argomento. È ammesso specificare un passo negativo, per ottenere un range di numeri disposti in ordine decrescente.

Si possono creare range sia di numeri interi che di numeri reali.[5] Per gli interi il passo di default è 1, ma se si desidera si può specificare un passo diverso; ad esempio:[6]

```
val r1 = 1 to 5      // r1: ...Range.Inclusive = Range(1, 2, 3, 4, 5)
val r2 = 1 until 3   // r2: ...Range = Range(1, 2)
r1 by 2              // res0: ...Range = Range(1, 3, 5)
r2 by 2              // res1: ...Range = Range(1)
6 to 1 by -2         // res2: ...Range = Range(6, 4, 2)
```

Invece, per i reali (`Float`, `Double`, ecc.) specificare il passo è obbligatorio, ovvero i range creati con `to` o `until` su tali tipi non sono utilizzabili direttamente, bensì bisogna applicare l'operatore `by` per ottenere range utilizzabili; ad esempio:[7]

```
val r3 = 1.3 to 4.3
  // r3: ...Range.Partial[...] = Range requires step
r3 by 1
  // res3: ...NumericRange[Double] = NumericRange(1.3, 2.3, 3.3, 4.3)
2.5 until 4 by 0.5
  // res4: ...NumericRange[Double] = NumericRange(2.5, 3.0, 3.5)
```

3.1 Esempio: determinare se un numero è primo

Usando il metodo `forall` su un range si può scrivere in modo compatto una funzione che determina se un numero è primo:

[5] La classe `Range` rappresenta solo i range di tipo `Int`, mentre i range di altri tipi sono rappresentati da un'altra classe simile ma generica, `NumericRange[T]`. Si noti però che i range di tipi a virgola mobile (`Float` e `Double`) sono deprecati a partire da Scala 2.12.6 e non più supportati da Scala 2.13.0 (perché vari metodi su questi range non si comportavano correttamente a causa di problemi legati intrinsecamente al funzionamento dei calcoli su `Float` e `Double`). L'unico tipo reale ancora supportato da `NumericRange` è `BigDecimal` (che rappresenta numeri a precisione arbitraria).

[6] Da Scala 2.12.0 il metodo `toString` dei range restituisce una rappresentazione simile alla sintassi usata per crearli (con gli operatori `to` o `until` ed eventualmente `by`), invece di elencare tutti gli elementi come mostrato negli esempi che seguono.

[7] Anche la rappresentazione `"Range requires step"` per i range di numeri reali con passo non specificato è stata aggiunga in Scala 2.12.0, ma qui è mostrata (insieme al resto dell'output nel formato di Scala 2.11) perché è più utile della rappresentazione usata prima, che era semplicemente quella restituita dall'implementazione di default di `toString` fornita da `Any`.

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

si costruisce il range dei possibili divisori e si verifica che il numero non sia divisibile per nessuno di essi.

Alcuni esempi di applicazione di questa funzione sono i seguenti:

```
prime(2)  // true
prime(6)  // false
prime(7)  // true
```

Si noti in particolare che nel caso `n` = `2` la funzione restituisce correttamente `true` perché il range `2 until n` = `2 until 2` è vuoto è `forall` restituisce `true` su una collezione vuota, dato che una proposizione quantificata universalmente è (vuotamente) verificata quando il dominio su cui la variabile quantificata varia è vuoto.

Equivalentemente, la funzione `prime` potrebbe essere implementata usando `exists` per verificare se esiste almeno un divisore di `n` in `2 until n` e poi negando il valore di verità così ottenuto:

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

4 Pattern matching function value

Si supponga di voler scrivere una funzione che, presi come argomenti due vettori `xs` e `ys` di `Double`,

$$\texttt{xs} = \langle x_1, \ldots, x_n \rangle \qquad \texttt{ys} = \langle y_1, \ldots, y_n \rangle$$

calcoli il loro prodotto scalare:

$$\sum_{i=1}^{n} x_i \cdot y_i$$

Il calcolo può essere scomposto in due fasi:

1. calcolare la sequenza dei prodotti degli elementi corrispondenti dei due vettori;
2. sommare questi prodotti.

Per il passo 2 si può semplicemente usare il metodo `sum` di `Seq`, mentre il passo 1 è più interessante: bisogna applicare un'operazione alle coppie di elementi corrispondenti di due collezioni. La soluzione tipica è usare `zip` per costruire la sequenza di tali coppie e poi trasformare ciascuna coppia tramite `map`. Allora, complessivamente, il calcolo del prodotto scalare può essere implementato come segue:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  (xs zip ys).map(xy => xy._1 * xy._2).sum
```

(qui i vettori sono rappresentati come `Vector`, ma in questo caso andrebbero bene anche delle `List`). Un esempio d'uso di questa funzione è:

```
val v1 = Vector(1.0, 1.5, 2.0, 2.5, 3.0)
val v2 = Vector(1.0, 2.0, 3.0, 4.0, 5.0)
scalarProduct(v1, v2)  // 35.0
```

Nell'implementazione appena mostrata la funzione anonima passata a `map` riceve come argomento una coppia e accede ai suoi elementi tramite i campi `_1` e `_2`. Per motivi di eleganza, si preferirebbe estrarre i valori degli elementi della coppia tramite pattern matching, e ciò può essere fatto mettendo un'espressione `match` come corpo della funzione anonima:

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

Siccome l'argomento `xy` è sempre una coppia, il singolo `case (x, y)` è sufficiente a gestire tutti i possibili valori dell'argomento, cioè non restano casi non trattati (i quali potrebbero portare a `MatchError` in fase di esecuzione).

Leggendo il codice di questa seconda versione di `scalarProduct` si può notare che l'espressione `match` introduce una certa ridondanza: qualunque funzione anonima con un argomento che faccia solo il pattern matching su tale argomento ha la forma x `=>` x `match {...}` (dove il nome della variabile x è irrilevante, supponendo che essa non compaia nel corpo del `match`), quindi l'unica parte veramente significativa sono i `case` scritti all'interno delle parentesi graffe. Al fine di eliminare la ridondanza, Scala fornisce una sintassi abbreviata per le espressioni di pattern matching usate come valori funzionali, chiamata **pattern matching function value**:

$$\texttt{\{case } p_1 \texttt{ => } e_1 \texttt{ ... case } p_n \texttt{ => } e_n\texttt{\}}$$

è zucchero sintattico che equivale a

$$x \texttt{ => } x \texttt{ match \{case } p_1 \texttt{ => } e_1 \texttt{ ... case } p_n \texttt{ => } e_n\texttt{\}}$$

Riscrivendolo con un pattern matching function value, il codice di `scalarProduct` diventa:

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

5 Tipo di `map` sui range

Si supponga di voler generare tutte le coppie di interi positivi $(1, j)$ per $1 \leq j \leq n$. Un modo di farlo è prima generare tutti gli interi j compresi tra 1 e n, e poi per ogni j generare la coppia $(1, j)$. In Scala, ciò si traduce nell'invocazione di `map` su un range:

```
val n = 4
(1 to n) map (j => (1, j))
```

Si osservi però quali sono il tipo e il valore del risultato di `map`:

```
res0: IndexedSeq[(Int, Int)] = Vector((1,1), (1,2), (1,3), (1,4))
```

- il tipo del risultato è `IndexedSeq[(Int, Int)]`;
- il valore è un'istanza di `Vector`.

Il metodo `map` definito in `Seq[A]` ha il prototipo

```
def map[B](f: A => B): Seq[B]
```

e una sua implementazione in genere, quando possibile, restituisce come valore una collezione dello *stesso tipo* di quella su cui `map` è invocato. Invece, il metodo `map` di `Range` ha il prototipo definito nel trait `IndexedSeq[A]` — un sottotipo di `Seq` che rappresenta i casi particolari in cui l'accesso per indice è supportato in modo efficiente, come ad esempio `Vector` e `Range` (ma non `List`). Tale prototipo è

```
def map[B](f: A => B): IndexedSeq[B]
```

il che spiega il tipo del risultato nell'esempio precedente, ma rimane da capire come mai il valore sia un'istanza di `Vector`.

In base alla convenzione di restituire una collezione dello stesso tipo di quella a cui si applica il metodo, si potrebbe pensare che `map` su `Range` restituisca un `Range` (che è sottotipo di, quindi compatibile con, `IndexedSeq`), ma ciò non è possibile perché `Range` può rappresentare solo intervalli numerici, e non è detto che il risultato di `map` su un intervallo sia ancora un intervallo (ad esempio, la sequenza di coppie restituita nell'esempio precedente non è sicuramente un intervallo numerico). Un'altra opzione da escludere è restituire direttamente un'istanza di `IndexedSeq`, che è impossibile perché `IndexedSeq` è un trait, ovvero in sostanza una classe astratta, di cui non si possono creare direttamente istanze. Allora bisogna scegliere un sottotipo concreto di `IndexedSeq` che non sia `Range`, e in generale nella libreria standard di Scala si sceglie `Vector` come implementazione "di default" di `IndexedSeq` per situazioni del genere (in cui bisogna restituire una collezione indicizzata che non può essere dello stesso tipo di quella su cui il metodo è invocato).

6 `flatten` e `flatMap`

Ora si vuole generalizzare l'esempio precedente dalla generazione delle sole coppie $(1, j)$ alla generazione di tutte le coppie di interi positivi (i, j) con $1 \leq i, j \leq n$. Un modo di procedere è il seguente: si generano tutti gli interi i compresi tra 1 e n, e per ogni i si genera la lista delle coppie $(i, 1), \ldots, (i, n)$. In altre parole, si può riutilizzare il codice precedente

```
(1 to n) map (j => (1, j))
```

sostituendo 1 con una `i` che varia da 1 a n:

```
val xs = (1 to n) map (i =>
  (1 to n) map (j => (i, j))
)
```

Tuttavia, il risultato `xs` così ottenuto

```
xs: IndexedSeq[IndexedSeq[(Int, Int)]] = Vector(
  Vector((1,1), (1,2), (1,3), (1,4)),
  Vector((2,1), (2,2), (2,3), (2,4)),
  Vector((3,1), (3,2), (3,3), (3,4)),
  Vector((4,1), (4,2), (4,3), (4,4))
)
```

non è esattamente ciò che si vuole ottenere: siccome l'invocazione "esterna" di `map` applica al range `1 to n` la funzione

```
i => (1 to n) map (j => (i, j))
```

che trasforma ciascun elemento `i` del range in un `Vector` di coppie, complessivamente si ottiene un `Vector` i cui elementi sono a loro volta `Vector` (di coppie), quando invece si vorrebbe un singolo `Vector` di coppie.

Per ottenere il risultato corretto bisogna concatenare in un'unica sequenza i vettori contenuti in `xs`. Se `xs` = `Vector(`v_1`, ..., `v_n`)`, le operazioni di concatenazione che danno il risultato desiderato possono essere scritte come

$$v_1 \texttt{ ++ (... ++ (}v_n \texttt{ ++ } \mathit{empty}\texttt{)...)}$$

che corrisponde a un'invocazione di `foldRight` su `xs` con l'operatore `++` e un qualche valore iniziale *`empty`* che è una sequenza vuota, l'elemento neutro della concatenazione. La scelta del valore da usare come elemento neutro, o meglio del tipo di tale valore, non è però immediata. Intuitivamente, sapendo che in fase di esecuzione gli elementi $v_1, \ldots, v_n$ sono istanze di `Vector`, si potrebbe pensare di usare un `Vector` vuoto,

```
xs.foldRight(Vector[(Int, Int)]())(_ ++ _)
```

ma ciò genera un errore in fase di compilazione, perché:

- il compilatore deduce il tipo del risultato di `foldRight` in base al tipo del valore iniziale, che qui è `Vector`;
- il compilatore sceglie la versione del metodo (operatore) `++` da invocare in base al tipo dell'oggetto su cui è invocato, ovvero dell'operando sinistro, che nel caso di `foldRight` (ricordando l'ordine degli argomenti dell'operatore) è un elemento della sequenza `xs`, qui di tipo (noto in fase di compilazione) `IndexedSeq` e non `Vector`;

- il metodo `++` definito da `IndexedSeq` restituisce un valore anch'esso di tipo `IndexedSeq`, il quale non è un sottotipo del tipo `Vector` che il compilatore ha dedotto come risultato di `foldRight` in base al valore iniziale.

La soluzione è indicare che il tipo in fase di compilazione del valore iniziale deve essere `IndexedSeq[(Int, Int)]` e non `Vector[(Int, Int)]`. Ci sono vari modi di fare ciò, ma uno particolarmente comodo è sfruttare il metodo factory fornito dal companion object di `IndexedSeq`. Infatti, un tipo astratto come `IndexedSeq` non può avere costruttori che consentano di crearne istanze con `new`, ma può avere un companion object con dei metodi `apply` che fungono da factory, purché tali metodi creino istanze di sottotipi concreti. Nel caso di `IndexedSeq`, il metodo factory richiama semplicemente il metodo factory di `Vector`, ma restituisce il vettore come un valore di tipo `IndexedSeq`,

```
val allPairs = xs.foldRight(IndexedSeq[(Int, Int)]())(_ ++ _)
```

così viene dedotto correttamente il tipo del risultato di `foldRight` e si ottiene finalmente la sequenza di coppie voluta:

```
allPairs: IndexedSeq[(Int, Int)] = Vector(
  (1,1), (1,2), (1,3), (1,4), (2,1), (2,2), (2,3), (2,4), (3,1), (3,2),
  (3,3), (3,4), (4,1), (4,2), (4,3), (4,4)
)
```

In realtà, non è necessario implementare manualmente la concatenazione tramite `foldRight`, perché il trait `Seq[A]` definisce un metodo `flatten` che "appiattisce" una collezione di collezioni su cui è invocato, restituendo la concatenazione delle sotto-collezioni, ovvero una singola collezione contenente, in ordine, gli elementi di tutte le sotto-collezioni. Usando `flatten` il codice diventa:

```
val xs = (1 to n) map (i =>
  (1 to n) map (j => (i, j))
)
val allPairs = xs.flatten
```

La combinazione di `map` seguito da `flatten` si usa spesso, quindi `Seq` definisce anche un metodo `flatMap` che combina tali operazioni: in generale,

$$xs \texttt{ flatMap } f = (xs \texttt{ map } f)\texttt{.flatten}$$

Allora il codice può essere scritto in modo ancora più compatto:

```
val allPairs = (1 to n) flatMap (i =>
  (1 to n) map (j => (i, j))
)
```