Azzolini Riccardo 2021-05-17

Classe di libreria `List`

1 Strutture dati immutable nella libreria standard

Le *strutture dati immutable* costituiscono uno degli elementi di base della programmazione funzionale pura. La libreria standard di Scala fornisce, all'interno del package `scala.collection.immutable`, le implementazioni di varie strutture immutable, tra cui `List`, `Vector`, `Range`, `HashMap`, `HashSet`, ecc. Adesso si inizierà a presentare la classe `List` — che è simile all'implementazione semplificata delle liste vista finora — mentre alcune delle altre strutture saranno discusse più avanti.

2 La classe `List[+T]`

La classe `scala.collection.immutable.List[+T]` implementa una *cons-list* (*immutable linked list*) che rappresenta collezioni ordinate di elementi di tipo `T`.

Così come gli array sono la struttura dati lineare "fondamentale" nei linguaggi imperativi, le liste lo sono nei linguaggi funzionali, poiché (a differenza degli array) esse sono *ricorsive* e *immutabili*. Dato che esse sono molto usate, dover ogni volta importare la classe `List` sarebbe piuttosto scomodo, perciò il package `scala` (importato automaticamente) fornisce un alias `scala.List`,

```
type List[T] = scala.collection.immutable.List[T]
```

che permette di usare `List` senza importarla.

Una lista contenente gli elementi $e_1, \ldots, e_n$ può essere creata con `List(`e_1`, ..., `e_n`)`, un costruttore (metodo `apply` del companion object di `List`) generico che accetta un numero qualsiasi di argomenti; in particolare, è ammesso specificare zero argomenti, `List()`, il che crea una lista vuota.

Le liste sono strutture dati omogenee, ovvero i cui elementi hanno tutti il medesimo tipo T. Quando si crea una lista con `List(`e_1`, ..., `e_n`)` il compilatore deduce T selezionando il più piccolo supertipo comune ai tipi di tutti gli elementi $e_1, \ldots, e_n$, o meglio il più piccolo tipo a cui sono riconducibili i tipi degli elementi $e_1, \ldots, e_n$ (considerando non solo le relazioni supertipo-sottotipo ma anche, ad esempio, le conversioni implicite tra i tipi numerici); per la lista vuota viene dedotto $T =$ `Nothing`. Ad esempio:

```
List()                                  // Tipo dedotto: List[Nothing]
List(1, 2, 3, 4, 5, 6)                  // Tipo dedotto: List[Int]
List(1, 2.5, 3L)                        // Tipo dedotto: List[Double]
List("apple", "orange", "pineapple")    // Tipo dedotto: List[String]
List(List(1, 2, 3), List(4, 5, 6))      // Tipo dedotto: List[List[Int]]
List(1, "apple", List(5, 6))            // Tipo dedotto: List[Any]
List(List(1, 2, 3), List("a", "b"))     // Tipo dedotto: List[List[Any]]
```

Si noti in particolare che il tipo dell'ultima lista è `List[List[Any]]` perché il tipo parametro di `List` è covariante — *`A <: B`* implica `List[`*`A`*`] <: List[`*`B`*`]` — quindi `List[Any]` è supertipo dei tipi dei due elementi di tale lista, che sono a loro volta liste di tipo `List[Int]` e `List[String]` rispettivamente.

3 Costruttori delle liste

I principali costruttori delle liste sono:

- `Nil`, che rappresenta la lista vuota;
- l'operatore di costruzione di liste `::`, chiamato **cons**: `x :: xs` restituisce una nuova lista che ha l'elemento `x` come testa e la lista `xs` come coda (ovvero corrisponde al metodo `xs.prepend(x)` e al costruttore `new Cons(x, xs)` che si erano definiti nell'implementazione delle liste presentata in precedenza).

In seguito sono riportati alcuni esempi di uso di questi costruttori:

```
val fruit = "apple" :: ("orange" :: ("pineapple" :: Nil))
val nums = 1 :: (2 :: (3 :: (4 :: (5 :: (6 :: Nil)))))
val empty = Nil
```

Siccome il nome dell'operatore `::` termina con i due punti, `:`, tale operatore è associativo a destra: *`A`* `::` *`B`* `::` *`C`* è equivalente ad *`A`* `::` `(`*`B`* `::` *`C`*`)`. Ciò significa che le parentesi negli esempi precedenti sono superflue, e omettendole si ottiene una maggiore leggibilità, si mette in evidenza la struttura delle liste costruite:

```
val fruit = "apple" :: "orange" :: "pineapple" :: Nil
val nums = 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: Nil
```

Inoltre, l'uso di un operatore il cui nome termina con `:` è interpretato come un'invocazione di metodo sull'operando destro invece che sul sinistro, quindi gli esempi precedenti equivalgono a:

```
val fruit = Nil.::("pineapple").::("orange").::("apple")
val nums = Nil.::(6).::(5).::(4).::(3).::(2).::(1)
```

Infatti il metodo `::` è definito sulla classe `List`, mentre se fosse invocato sull'operando sinistro dovrebbe essere definito sui tipi dei valori da inserire nelle liste.

`Nil` e (soprattutto) `::` permettono di costruire dinamicamente liste di dimensione non prefissata, tramite l'aggiunta di un elemento alla volta. Quando invece si vuole costruire una lista con un numero prefissato di elementi $e_1, \ldots, e_n$ è più comodo il costruttore `List(`e_1`, ..., `e_n`)`, che a sua volta è implementato usando `Nil` e `::`; ad esempio, `List()` equivale a `Nil` e `List(1, 2, 3)` equivale a `1 :: 2 :: 3 :: Nil`.

Un aspetto importante delle liste (come di tutte le strutture dati) sono le complessità delle varie operazioni. Una singola applicazione dell'operatore `::` richiede tempo costante, poiché esso deve semplicemente costruire un nuovo nodo della lista (si pensi all'implementazione del metodo `prepend`), e la costruzione di una lista di n elementi a partire dalla lista vuota comporta n applicazioni di `::`, quindi richiede tempo proporzionale a n (cioè lineare nel numero di elementi).

4 Operazioni sulle liste

Tutte le operazioni sulle liste possono essere espresse in termini di tre metodi fondamentali della classe `List`, che in particolare sono metodi di classificazione e accesso:

- `isEmpty`, che restituisce `true` se e solo se la lista è vuota;
- `head`, che restituisce il primo elemento della lista.
- `tail`, che restituisce la coda della lista.

`head` e `tail` sollevano un'eccezione se invocati sulla lista vuota.

4.1 Esempio: insertion sort

Come esempio di definizione di un'operazione sulle liste in base ai metodi `isEmpty`, `head` e `tail`, si vuole implementare l'algoritmo di ordinamento *insertion sort* su liste di numeri interi.

L'algoritmo opera sulla struttura ricorsiva delle liste in questo modo:

- la lista vuota è già ordinata;
- per ordinare una lista non vuota si ordina ricorsivamente la coda e poi si inserisce la testa nella posizione corretta della coda ordinata.

L'implementazione ricalca esattamente tale definizione ricorsiva:

```
def isort(xs: List[Int]): List[Int] =
  if (xs.isEmpty) Nil else insert(xs.head, isort(xs.tail))
```

Anche l'inserimento è definito in modo ricorsivo sulla struttura della lista in cui si vuole inserire un elemento:

- se la lista è vuota o inizia con una testa maggiore o uguale all'elemento da inserire, quest'ultimo viene aggiunto come nuova testa (in particolare, nel caso della lista vuota diventa l'unico elemento);
- se invece la lista non è vuota e ha una testa che è minore dell'elemento da inserire (quindi questo va inserito più avanti), si mantiene la stessa testa e si effettua l'inserimento ricorsivamente nella coda.

```
def insert(x: Int, xs: List[Int]): List[Int] =
  if (xs.isEmpty || x <= xs.head) x :: xs
  else xs.head :: insert(x, xs.tail)
```

5 Pattern sulle liste

Un altro meccanismo di accesso/decomposizione per le liste, tipicamente più usato rispetto ai metodi `head`, `tail` e `isEmpty`, sono i pattern definiti sulle liste:[1]

- `Nil`, la costante che corrisponde alla lista vuota;
- *x* `::` *xs*, che corrisponde a una lista con testa *x* e coda *xs*;
- `List(`e_1`, ..., `e_n`)`, che equivale a e_1 `::` `...` `::` e_n `::` `Nil` (in particolare, il pattern `List()` equivale a `Nil`).

Alcuni esempi di uso di questi pattern sono:

- `1 :: 2 :: xs`, che corrisponde a ogni lista che inizia con `1` e `2`, seguiti da una coda `xs` qualsiasi (si noti che `xs` può corrispondere sia alla lista vuota che a un'arbitraria lista non vuota);
- `x :: Nil` e `List(x)`, che sono tra loro equivalenti e corrispondono a tutte le liste di lunghezza 1;
- `Nil` e `List()`, entrambi corrispondenti alla lista vuota;
- `List(1 :: xs)`, corrispondente a una lista con un solo elemento che è a sua volta una lista che inizia con `1`.

Mentre `Nil` è una "semplice" costante, i pattern *x* `::` *xs* e `List(`e_1`, ..., `e_n`)` non corrispondono esattamente alle forme dei pattern specificate in precedenza, perché sono definiti sfruttando alcuni meccanismi ad-hoc offerti dal linguaggio:

- `List(`e_1`, ..., `e_n`)` è un **extractor** pattern;

[1] Qui x, xs, $e_1, \ldots, e_n$ possono essere pattern di qualsiasi tipo, non per forza variabili.

- *x* `::` *xs* è un **infix operator pattern** equivalente al costruttore `::(`*x*`,` *xs*`)` (in generale, ogni costruttore o extractor pattern con due argomenti può essere scritto in notazione infissa, mettendo il nome del costruttore/extractor in mezzo tra i due argomenti).

5.1 Extractor

I pattern possono essere definiti indipendentemente dalle classi case, tramite la definizione di un oggetto che prende il nome di **extractor object** e fornisce un metodo `unapply` (o `unapplySeq`), il quale in un certo senso *inverte* il lavoro effettuato dal metodo `apply`: In particolare, se `apply` realizza un costruttore, `unapply` dovrebbe realizzare un "decostruttore", tramite cui accedere alle informazioni usate per costruire l'oggetto su cui si sta eseguendo il pattern matching.

Un esempio di definizione e uso di un extractor object è il seguente (qui i commenti indicano alcune parti rilevanti dell'output dell'interprete):

```
object Twice {
  def apply(x: Int): Int = x * 2
  def unapply(z: Int): Option[Int] =
    if (z % 2 == 0) Some(z / 2) else None
}

val x = Twice(21)                          // x: Int = 42
x match { case Twice(n) => n }             // res0: Int = 21
val y = 13
y match { case Twice(n) => n; case _ => 0 } // res1: Int = 0
```

L'oggetto `Twice` rappresenta l'operazione di raddoppio di un numero intero, che esso fornisce tramite il metodo `apply`. Il metodo `unapply` permette l'uso di `Twice` come pattern invertendo l'operazione di raddoppio: esso deve restituire la metà del valore intero su cui si fa il pattern matching, ma solo se tale numero è effettivamente il doppio di un altro, cioè potrebbe essere stato ottenuto tramite `Twice.apply`, altrimenti deve in qualche modo indicare che il pattern non corrisponde al valore considerato.

Per poter indicare se c'è o meno corrispondenza con il pattern, il metodo `unapply` restituisce (in questo caso) un valore della classe `Option[+T]`, che rappresenta valori opzionali. Essa ha infatti due sottotipi:

- `Some[+T]`, il caso in cui il valore opzionale è presente (tale valore viene passato come argomento al costruttore `Some` e può essere estratto tramite pattern matching);
- `None`, il caso in cui il valore opzionale è assente.

Di solito `Option` si usa come meccanismo semplice ma molto comodo per segnalare se un'operazione è andata a buon fine e in caso positivo comunicare al tempo stesso il risultato dell'operazione.

Complessivamente, il meccanismo di `unapply` per l'oggetto `Twice` funziona in questo modo: quando si confronta il valore `z` del selettore con un pattern `Twice(p)` (dove `p` è un pattern qualsiasi) viene invocato il metodo `Twice.unapply(z)`, che determina se `z` corrisponde al pattern.

- Se c'è corrispondenza `unapply` restituisce `Some(z / 2)`, e il valore `z / 2` viene usato per il pattern matching con il pattern `p` specificato in `Twice(p)` (nell'esempio mostrato prima `p` è una variabile `n`, che sicuramente corrisponde al valore `z / 2` e viene legata a esso, ma in altri casi si potrebbero avere pattern che non corrispondono con i valori restituiti da `unapply`, e allora complessivamente non si avrebbe la corrispondenza con `z`).
- Se invece non c'è corrispondenza `unapply` restituisce `None`, e allora si prosegue a confrontare `z` con i successivi pattern elencati nell'espressione `match` (se ce ne sono, altrimenti viene sollevato un `MatchError`).

In generale, il tipo che il metodo `unapply` deve restituire dipende dal numero di sotto-valori che il pattern estrae, ovvero dal numero di argomenti del pattern (che sono gli altri pattern con cui confrontare i sotto-valori estratti):

- `Boolean` se il pattern non estrae alcun sotto-valore, cioè rappresenta un test che può solo essere soddisfatto o meno;
- `Option[T]` se il pattern estrae un singolo sotto-valore di tipo `T`;
- `Option[(T1, ..., Tn)]`[2] se il pattern estrae un numero prefissato `n` di sotto-valori, i cui tipi sono rispettivamente $T_1, \ldots, T_n$.

Se il numero di sotto-valori non è fissato si implementa invece il metodo `unapplySeq`, che restituisce una sequenza di valori; questo è il caso implementato nel companion object `List`.

5.2 Esempio: insertion sort con pattern matching

L'implementazione dell'algoritmo insertion sort può essere riscritta usando il pattern matching:

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match {
  case Nil => List(x)
  case head :: tail =>
    if (x <= head) x :: xs
    else head :: insert(x, tail)
```

[2] Il tipo `(T1, ..., Tn)` è una tupla. Le tuple verranno presentate più avanti.

```
}

def isort(xs: List[Int]): List[Int] = xs match {
  case Nil => Nil
  case head :: tail => insert(head, isort(tail))
}
```

Questo codice è più elegante rispetto a quello che usava i metodi di classificazione e accesso, e rende più esplicita la struttura ricorsiva dei metodi, poiché il pattern matching è molto simile (anche sintatticamente) alle definizioni per casi che si usano in matematica.

Ragionando sulla struttura ricorsiva di `insert` e `isort` si può determinare la complessità di queste funzioni nel caso peggiore:

- Per `insert` il caso peggiore è quello in cui `x` è maggiore di tutti gli elementi della lista `xs`. Se `xs` ha lunghezza n, in questo caso l'esecuzione di `insert` richiede n invocazioni ricorsive.
- L'esecuzione di `isort` su una lista di lunghezza n richiede sempre n chiamate ricorsive, una per ogni elemento della lista, e ciascuna chiamata esegue un'operazione di `insert` su una lista avente al più lunghezza n (-1, ma ciò si può trascurare nell'analisi della complessità), quindi complessivamente il tempo richiesto da `isort` nel caso peggiore è proporzionale a n^2 (quadratico nella lunghezza della lista).

6 Alcuni metodi di `List`

La classe `List` fornisce numerosi metodi che implementano varie operazioni comuni sulle liste. In seguito verranno brevemente presentati alcuni di questi metodi.

- Metodi di accesso agli elementi e alle sottoliste:
  - `xs.length` restituisce la lunghezza di `xs`.
  - `xs.last` è sostanzialmente il metodo duale di `xs.head`, che restituisce l'ultimo elemento di `xs` o solleva l'eccezione `NoSuchElementException` se `xs` è vuota.
  - `xs.init` è il metodo duale di `tail`, che restituisce la lista costituita da tutti gli elementi di `xs` tranne l'ultimo o solleva l'eccezione `UnsupportedOperationException` se `xs` è vuota.
  - `xs take n` restituisce la lista costruita dai primi `n` elementi di `xs`. Questo metodo non solleva mai eccezioni, gestendo invece i casi limite in questo modo:
    - se `n <= 0` restituisce la lista vuota;
    - se `n >= xs.length` restituisce l'intera lista `xs`.

  - `xs drop n` restituisce la lista ottenuta da `xs` togliendo i primi `n` elementi. Come `take`, `drop` non solleva mai eccezioni:
    * se `n <= 0` restituisce l'intera lista `xs`;
    * se `n >= xs.length` restituisce la lista vuota.
  - `xs(n)`, ovvero `xs apply n`, restituisce l'elemento di `xs` corrispondente all'indice `n` (cioè l'$(n+1)$-esimo elemento, perché gli indici partono da 0), o solleva l'eccezione `IndexOutOfBoundsException` se l'indice non è valido (`n < 0` o `n >= xs.length`).

- Metodi di creazione di nuove liste:
  - `xs ++ ys`, disponibile anche con il nome alfanumerico `concat` (`xs concat ys`), restituisce la concatenazione delle liste `xs` e `ys`, cioè una lista contenente gli elementi di `xs` seguiti dagli elementi di `ys`.
  - `xs.reverse` restituisce una lista contenente gli elementi di `xs` in ordine inverso.
  - `xs.updated(n, x)` restituisce una lista che contiene tutti gli stessi elementi di `xs` tranne quello all'indice `n`, che è invece sostituito con `x`. Come `xs(n)`, questo metodo solleva una `IndexOutOfBoundsException` se l'indice non è valido (`n < 0` o `n >= xs.length`).

- Metodi di ricerca di elementi:
  - `xs indexOf x` restituisce l'indice del primo elemento di `xs` che è uguale a `x` (secondo `equals`), oppure `-1` se un tale elemento non è presente in `xs`.
  - `xs contains x` restituisce `true` se e solo se `xs` contiene un elemento uguale a `x` (secondo `equals`).

Adesso, per ragionare sulla complessità di questi metodi (che è importante conoscere quando li si usa), verranno mostrate delle possibili implementazioni di alcuni di essi (per semplicità, tali implementazioni saranno sotto forma di funzioni invece che di metodi, ma a parte la sintassi dell'invocazione ciò non fa sostanzialmente alcuna differenza).

6.1 Complessità di `last`

Siccome una lista è una struttura dati lineare, per raggiungere l'ultimo elemento è necessario scorrere tutti gli elementi della lista, cioè eseguire un numero di passi lineare nella lunghezza della lista, come illustrato da questa possibile implementazione di `last`:

```
def last[T](xs: List[T]): T = xs match {
  case List() => throw new NoSuchElementException("last of empty list")
  case List(x) => x
  case _ :: ys => last(ys)
}
```

Al contrario, ottenere il primo elemento di una lista richiede tempo costante (è sufficiente accedere al campo `head`), dunque quando possibile è conveniente organizzare le liste in modo che la maggior parte degli accessi siano all'inizio piuttosto che alla fine. Se un algoritmo deve fare accessi frequenti alla fine di una lista può essere conveniente fare un `reverse` in modo da trasformarli in accessi all'inizio, ma ciò va valutato tenendo conto anche della complessità di `reverse`, che si discuterà a breve.

6.2 Complessità di ++

La concatenazione `xs ++ ys` è implementata ricorsivamente sulla struttura della lista `xs`, che deve essere "ricostruita" mettendo `ys` come coda dell'ultimo nodo (al posto di `Nil`):

```
def concat[T](xs: List[T], ys: List[T]): List[T] = xs match {
  case Nil => ys
  case z :: zs => z :: concat(zs, ys)
}
```

Di conseguenza, `concat` effettua una chiamata ricorsiva per ogni elemento di `xs`, e siccome ciascuna chiamata ricorsiva esegue solo un'operazione che richiede tempo costante (`::`) la complessità di `concat` è lineare nella lunghezza di `xs`.

6.3 Complessità di `reverse`

Il modo più "immediato" di definire `reverse` sulla struttura ricorsiva di una lista è il seguente:

```
def reverse[T](xs: List[T]): List[T] = xs match {
  case Nil => Nil
  case y :: ys => reverse(ys) ++ List(y)
}
```

- la lista vuota rimane invariata, perché non ci sono elementi di cui invertire l'ordine;
- per invertire l'ordine degli elementi di una lista non vuota si inverte ricorsivamente l'ordine della coda e si aggiunge la testa come ultimo elemento.

Su una lista `xs` di lunghezza n l'esecuzione di `reverse` richiede $n-1$ invocazioni ricorsive (qui non si conta l'invocazione iniziale, `reverse(xs)`). La prima di queste invocazioni è su una lista di lunghezza $n-1$, la seconda è su una lista di lunghezza $n-2$, e così via, fino all'ultima che è su una lista vuota, cioè di lunghezza 0. Ciascuna di queste invocazioni restituisce poi una lista della stessa lunghezza di quella passata come argomento, e tale lista viene usata come operando sinistro in un'operazione di concatenazione. Siccome il tempo richiesto da una concatenazione è lineare nella lunghezza della lista di sinistra, il tempo complessivamente richiesto da tutte le concatenazioni è proporzionale a

$$(n-1) + (n-2) + \cdots + 1 + 0 = \frac{n(n-1)}{2}$$

ovvero quadratico nella lunghezza della lista `xs`. Ciò è insoddisfacente rispetto al tempo richiesto dall'operazione `reverse` su una lista *mutable*, che è lineare nella lunghezza della lista (poiché deve solo invertire il verso dei puntatori).

Il fatto che `reverse` richieda tempo quadratico nella lunghezza della lista non è una caratteristica intrinseca delle liste immutable, bensì è dovuto all'uso dell'operatore di concatenazione. Esistono invece altre implementazioni che applicano solo un operatore `::` per ogni elemento della lista, richiedendo così tempo lineare nella lunghezza. Un esempio è la seguente implementazione tail-recursive,

```
def reverse[T](xs: List[T]): List[T] = {
  @tailrec
  def reverseIter(xs: List[T], rev: List[T]): List[T] = xs match {
    case Nil => rev
    case y :: ys => reverseIter(ys, y :: rev)
  }
  reverseIter(xs, Nil)
}
```

che in sostanza scorre la lista originale costruendo una nuova lista alla quale aggiunge iterativamente ciascun elemento per mezzo dell'operatore `::`. Siccome gli elementi sono aggiunti nell'ordine dal primo all'ultimo, ma l'operatore `::` inserisce ogni elemento in testa, l'elemento che risulta in testa alla nuova lista al termine delle chiamate ricorsive è l'ultimo elemento della lista originale, il secondo elemento della nuova lista è il penultimo dell'originale, e così via, cioè l'ordine degli elementi è invertito. Ad esempio, sulla lista `1 :: 2 :: 3 :: Nil` questa versione di `reverse` funziona come mostrato in seguito:

```
reverse(1 :: 2 :: 3 :: Nil)
    ↠ reverseIter(1 :: 2 :: 3 :: Nil,  Nil)
    ↠ reverseIter(2 :: 3 :: Nil,  1 :: Nil)
    ↠ reverseIter(3 :: Nil,  2 :: 1 :: Nil)
    ↠ reverseIter(Nil,  3 :: 2 :: 1 :: Nil)
    ↠ 3 :: 2 :: 1 :: Nil
```