Azzolini Riccardo 2021-04-29

Cons-list, tipi generici e polimorfismo

1 Cons-list

Le **cons-list**, una struttura dati fondamentale in molti linguaggi funzionali, sono **linked list immutabili** definite ricorsivamente a partire da:

- `Nil`, il caso base, che rappresenta la lista vuota;
- `Cons`, un costruttore (il suo nome è appunto un'abbreviazione di "constructor") che costruisce una lista a partire da un elemento e da un'altra lista (per la precisione, un riferimento al resto della lista), ovvero rappresenta una cella, un nodo della lista.

Le cons-list furono introdotte in Lisp, il primo linguaggio funzionale, nel quale erano la struttura dati principale, fornita direttamente a livello del linguaggio. In Scala, invece, esse sono implementate nella libreria standard, allo stesso modo di altre strutture dati che possono essere definite dall'utente; grazie ad alcune funzionalità del linguaggio (la possibilità di definire operatori, ecc.), tale implementazione rimane comunque piuttosto comoda da usare, come se fosse una struttura predefinita del linguaggio.

Le cons-list fornite dalla libreria standard di Scala possono contenere elementi di un tipo qualsiasi, ma per illustrarne l'implementazione è utile iniziare da una versione semplificata che può contenere solo numeri interi. Tale implementazione è composta dalle seguenti definizioni:

```
trait IntList { /* ... */ }
class Nil extends IntList { /* ... */ }
class Cons(val head: Int, val tail: IntList) extends IntList {
  // ...
}
```

- un trait `IntList` che specifica i metodi disponibili sulle liste (qui omessi perché si vuole prima porre l'attenzione sulla struttura ad alto livello delle definizioni);
- due sottoclassi concrete, `Nil` e `Cons`, corrispondenti ai due casi della definizione ricorsiva.

In accordo con queste definizioni, una `IntList` è una lista vuota, `new Nil`,[1] oppure una lista `new Cons(x, xs)` costituita da un elemento `x`, che prende il nome di **head** (testa), e da una lista `xs`, che assume il ruolo di **tail** (coda). Ad esempio, la lista contenente gli elementi `3`, `-1`, `7` è:

```
new Cons(3, new Cons(-1, new Cons(7, new Nil)))
```

Si noti che `Nil` potrebbe essere implementato come `object`, dato che la lista vuota è unica (tutte le sue istanze sono identiche), ma definirlo per ora come classe semplificherà la generalizzazione a liste di tipo generico che verrà fatta a breve.

2 Parametri `val`

Come già accennato in precedenza, Scala fornisce una sintassi compatta per evitare di dover introdurre esplicitamente i campi corrispondenti ai parametri di una classe. Tale sintassi è stata usata nella definizione della classe `Cons`:

```
class Cons(val head: Int, val tail: IntList) extends IntList {
  // ...
}
```

In generale, scrivere `val` di fronte a un parametro di una classe ha l'effetto di:

- definire il parametro (formale) di classe;
- definire un corrispondente campo della classe, che al momento della creazione di un oggetto viene inizializzato con il valore del parametro (attuale) di classe.

Questa funzionalità è zucchero sintattico, poiché può essere riscritta aggiungendo esplicitamente le definizioni dei campi corrispondenti ai parametri (e rinominando i parametri per evitare conflitti di nomi tra questi e i campi): ad esempio, la definizione

```
class Cons(val head: Int, val tail: IntList) extends IntList {
  // ...
}
```

può essere riscritta come

```
class Cons(_head: Int, _tail: IntList) extends IntList {
  val head = _head
  val tail = _tail
  // ...
}
```

dove `_head` e `_tail` devono essere nomi non utilizzati nel corpo della classe.

[1] Siccome la classe `Nil` non ha parametri (o meglio, siccome ha un costruttore senza parametri), nell'espressione di creazione `new Nil()` è consentito omettere le parentesi vuote.

3 Tipi generici

La definizione delle liste appena data tramite la gerarchia di `IntList` fissa a `Int` il tipo degli elementi. Nell'uso pratico questa sarebbe una forte limitazione, poiché se si volesse disporre di liste con elementi di tipi diversi bisognerebbe reimplementare tutto da capo, duplicando il codice dell'intera gerarchia di classe per costruirne altre analoghe. Fortunatamente, il linguaggio fornisce il meccanismo dei **tipi generici** (**generics**), che permette di generare la definizione astraendo sul tipo degli elementi mediante l'uso di un **tipo parametro** `T`, indicato tra parentesi quadre:

```
trait List[T] { /* ... */ }
class Nil[T] extends List[T] { /* ... */ }
class Cons[T](val head: T, val tail: List[T]) extends List[T] {
  // ...
}
```

L'identificatore `T` (scelto arbitrariamente) è un parametro formale, che al momento dell'uso viene poi istanziato specificando, ancora tra parentesi quadre dopo il nome della classe/trait, un qualsiasi tipo come parametro attuale. Ad esempio, `List[Int]` è il tipo delle liste con elementi di tipo `Int`, e una possibile lista di questo tipo (contenente gli interi `3`, `-1`, `7`) è

```
new Cons[Int](3, new Cons[Int](-1, new Cons[Int](7, new Nil[Int])))
```

mentre una lista di valori booleani (contenente due volte `false`), cioè una delle istanze del tipo `List[Boolean]`, è:

```
new Cons[Boolean](false, new Cons[Boolean](false, new Nil[Boolean]))
```

Come si può notare, specificare ogni volta i tipi parametro è piuttosto scomodo, ma nelle invocazioni dei costruttori il compilatore è spesso in grado di dedurli, quindi possono essere omessi:

```
new Cons(3, new Cons(-1, new Cons(7, new Nil)))
new Cons(false, new Cons(false, new Nil))
```

Un'osservazione importante è che quando si definisce un trait o una classe che dipende da un tipo parametro bisogna specificare come istanziare tale tipo *ogni volta* che si fa riferimento al trait/classe, anche quando lo si estende. Ad esempio, nella definizione

```
class Nil[T] extends List[T] { /* ... */ }
```

il tipo parametro `T` della classe `Nil` viene anche passato al trait `List` che essa estende. Così, al momento della costruzione, per esempio `new Nil[Int]`, viene opportunamente istanziato con il tipo `Int` (in questo caso) anche il codice del trait `List`.

Una classe (o trait) può anche dipendere da più tipi parametro. Ad esempio, la classe

```
class Pair[T, U](val x: T, val y: U)
```

rappresenta una coppia di valori di due tipi arbitrari, e potrebbe essere istanziata scrivendo `new Pair[Boolean, Int](true, 15)` oppure, equivalentemente, `new Pair(true, 15)`.

4 Metodi di `List`

Il trait `List` dichiara i metodi astratti corrispondenti alle operazioni che, nella letteratura, sono tipicamente disponibili sulle cons-list:

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}
```

Tali operazioni vanno poi implementate concretamente nelle sottoclassi.

Nella sottoclasse `Cons`

```
class Cons[T](val head: T, val tail: List[T]) extends List[T] {
  def isEmpty: Boolean = false
}
```

è necessario implementare esplicitamente solo `isEmpty` (che restituisce sempre `false` perché un'istanza di `Cons` rappresenta una lista contenente almeno un elemento), mentre i metodi `head` e `tail` sono già implementati dagli omonimi campi. Infatti, a livello concettuale non c'è distinzione tra metodi senza argomenti e campi: in sostanza, *i campi sono casi particolari di metodi, quindi possono sovrascrivere metodi o definire metodi astratti*. L'unica differenza tra campi definiti con `val` e con `def` è l'inizializzazione:

- con `val` l'espressione associata al campo viene valutata una sola volta, quando l'oggetto è inizializzato;
- con `def` l'espressione viene valutata ogni volta che si utilizza il campo.

Invece, la sottoclasse `Nil`

```
class Nil[T] extends List[T] {
  override def isEmpty: Boolean = true

  override def head: T =
    throw new NoSuchElementException("Nil.head")

  override def tail: List[T] =
    throw new NoSuchElementException("Nil.tail")
}
```

di per sé non ha campi, dunque deve implementare esplicitamente tutti i metodi. In particolare, la lista vuota non possiede né una testa né una coda, ma deve per forza implementare `head` e `tail` perché sono metodi astratti, dunque l'unica implementazione ragionevole è sollevare un'eccezione (poi, quando si usano le liste, per evitare che queste eccezioni vengano sollevate si può controllare che `isEmpty` sia `false` prima di invocare `head` e/o `tail`). Si osservi che, come già detto, è consentito usare delle clausole `throw` al posto di espressioni dei tipi `T` e `List[T]` previsti perché tali clausole hanno tipo `Nothing`, che è sottotipo di, e quindi compatibile con, ogni tipo.

5 Funzioni generiche

Oltre alle classi e ai trait, anche le funzioni e i metodi possono dipendere da tipi parametro. Ad esempio, il frammento di codice

```
def singleton[T](elem: T): List[T] = new Cons[T](elem, new Nil[T])
```

definisce una funzione generica con un tipo parametro `T` che costruisce una `List[T]` contenente il singolo elemento (di tipo `T`) specificato come argomento. Tale funzione può essere usata specificando esplicitamente come istanziare il tipo parametro:

```
singleton[Int](1)
singleton[Boolean](true)
```

Tuttavia, quando il compilatore è in grado di dedurre il tipo dell'argomento `elem` allora riesce a dedurre anche il tipo parametro, che può così essere omesso (come già visto nel caso dei costruttori):

```
singleton(1)
singleton(true)
```

6 Tipi generici e valutazione

In Scala, i tipi parametro vengono utilizzati esclusivamente dal compilatore per verificare la corretta applicazione dei metodi, dopodiché non vengono utilizzati in fase di esecuzione, ovvero non modificano il meccanismo di valutazione. Questa modalità di gestione dei tipi parametro prende il nome di **type erasure**. Ci sono invece linguaggi che adottano un modello diverso, nel quale l'informazione sui tipi parametro è preservata in fase di esecuzione.

7 Polimorfismo

In generale, il **polimorfismo** è la possibilità di presentarsi in *forme differenti*. Nei linguaggi di programmazione, questo termine si applica sostanzialmente ai tipi:

- nel caso delle funzioni significa che una funzione può essere applicata ad argomenti di tipi diversi;
- nel caso delle classi significa che un tipo può assumere come valori istanze di classi diverse.

In Scala si hanno due principali forme di polimorfismo:

- **subtyping**, tipico dei linguaggi orientati agli oggetti;
- **generics**, tipico dei linguaggi funzionali.

Il subtyping è il fatto che il tipo di una sottoclasse può essere utilizzato ovunque sia previsto il tipo della classe base. In altre parole, i valori del tipo della classe base possono assumere diverse forme, corrispondenti alle diverse sottoclassi. Ad esempio, `Nil` e `Cons` sono sottoclassi di `List`, dunque possono essere usate ogniqualvolta sia previsto un valore di tipo `List`, cioè si ha polimorfismo perché `List` può assumere due forme, `Nil` e `Cons`.

Invece, con i generics si creano istanze (forme) diverse di una funzione o di una classe/trait sulla base dei tipi parametro. Ad esempio, `List[Int]` e `List[Boolean]` sono forme diverse di `List[T]`.

Una funzione che combina entrambi questi tipi di polimorfismo è la seguente, che dati un indice e una lista restituisce l'elemento della lista corrispondente a tale indice se l'indice è valido (compreso tra 0 e la lunghezza della lista meno 1), altrimenti solleva un'opportuna eccezione:

```
def nth[T](n: Int, xs: List[T]): T =
  if (xs.isEmpty)
    throw new IndexOutOfBoundsException(n)
  else if (n == 0)
    xs.head
  else
    nth(n - 1, xs.tail)
```

Le occorrenze delle due forme di polimorfismo in questa funzione sono:

- Generics: la funzione è definita in dipendenza da un tipo parametro `T`, il che permette di applicarla a liste con tipi di elementi diversi, poiché al momento dell'uso `T` può essere istanziato con il tipo degli elementi della lista su cui si vuole operare. Ciò ha senso perché il codice necessario per estrae un elemento di una lista dato un indice è indipendente dal tipo di elementi della lista.

- Subtyping: in fase di esecuzione, il parametro attuale corrispondente al parametro formale `xs` può essere un oggetto istanza di `Cons` o di `Nil`. Si noti però che i metodi `head` e `tail` vengono chiamati solo quando esso è istanza di `Cons`, grazie al controllo eseguito su `isEmpty`.

In questo esempio la funzione `nth` ha bisogno di un tipo parametro per il tipo degli elementi della lista perché è definita al di fuori del trait `List[T]`. In alternativa, la si potrebbe definire direttamente nel trait, e allora essa riutilizzerebbe il `T` del trait stesso, senza bisogno di introdurre un nuovo tipo parametro (ma ne potrebbe introdurre altri se ne avesse bisogno per motivi diversi, come si vedrà in alcuni esempi più avanti):

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]

  def nth(n: Int, xs: List[T]): T =
    if (xs.isEmpty)
      throw new IndexOutOfBoundsException(n)
    else if (n == 0)
      xs.head
    else
      nth(n - 1, xs.tail)
}
```