Azzolini Riccardo 2021-03-25

Higher-order functions

1 Higher-order functions

Nei linguaggi funzionali, *le funzioni sono valori* a tutti gli effetti. Ciò significa che esse, come tutti gli altri valori del linguaggio, possono essere:

- memorizzate;
- passate come parametri a una funzione;
- restituite da una funzione.

Dal punto di vista dei tipi, una funzione ha un tipo che stabilisce una correlazione tra gli elementi del dominio e del codominio (in inglese *range*). In un linguaggio funzionale, questo tipo può essere usato ovunque siano ammessi gli altri tipi. Ciò fornisce meccanismi di composizione delle funzioni estremamente flessibili, che consentono la scrittura di codice più elegante, meno prolisso, e un riuso "più furbo" del codice.

Si chiama **higher-order function** (**funzione di ordine superiore** o **all'ordine superiore**) una funzione che prende una o più funzioni come parametri e/o restituisce una funzione come risultato. Al contrario, si dice **first-order function** (**funzione del primo ordine** o **al primo ordine**) una funzione che *non* è di ordine superiore, cioè i cui parametri e il cui valore restituito non sono funzioni.

Al fine di illustrare l'utilità delle funzioni di ordine superiore, e il modi in cui si definiscono e usano in Scala, è utile considerare un esempio. Si supponga di definire una funzione che calcola la sommatoria dei numeri interi compresi tra gli estremi a e b — in simboli, $\sum_{n=a}^{b} n$. Una semplice implementazione[1] è la seguente:

```
def sumInts(a: Int, b: Int): Int =
  if (a > b) 0 else a + sumInts(a + 1, b)
```

Ora, si definiscono anche delle funzioni per calcolare la sommatoria dei quadrati dei numeri tra a e b, $\sum_{n=a}^{b} n^2$,

```
def square(x: Int): Int = x * x

def sumSquares(a: Int, b: Int): Int =
  if (a > b) 0 else square(a) + sumSquares(a + 1, b)
```

[1] Per semplicità, qui e in seguito verranno mostrate implementazioni che saranno ricorsive ma non tail-recursive. Volendo, si potrebbero invece scrivere senza problemi delle implementazioni tail-recursive.

e la sommatoria dei fattoriali, $\sum_{n=a}^{b} n!$:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)

def sumFactorials(a: Int, b: Int): Int =
  if (a > b) 0 else factorial(a) + sumFactorials(a + 1, b)
```

Confrontando i corpi delle tre funzioni di somma,

```
if (a > b) 0 else           a  +       sumInts(a + 1, b)
if (a > b) 0 else    square(a) +    sumSquares(a + 1, b)
if (a > b) 0 else factorial(a) + sumFactorials(a + 1, b)
```

si osserva che esse sono molto simili, e diventano ancora più simili se si riscrive `sumInts` usando la *funzione identità* `id`:

```
def id(x: Int): Int = x

if (a > b) 0 else        id(a) +       sumInts(a + 1, b)
if (a > b) 0 else    square(a) +    sumSquares(a + 1, b)
if (a > b) 0 else factorial(a) + sumFactorials(a + 1, b)
```

l'unica differenza (a parte il nome della funzione usato per la chiamata ricorsiva) è la funzione applicata a ciascun intero nell'intervallo per calcolare il valore da aggiungere alla somma parziale — rispettivamente `id`, `square` e `factorial`. Infatti, queste funzioni calcolano casi particolari della sommatoria $\sum_{n=a}^{b} f(n)$, con appunto $f(n) = n$ (`id`), $f(n) = n^2$ (`square`) e $f(n) = n!$ (`factorial`).

Il pattern comune così individuato può essere fattorizzato definendo una funzione all'ordine superiore che prende come argomento la funzione da applicare a ogni intero compreso nell'intervallo. Il tipo del *parametro funzione* è un **tipo funzionale** (un tipo i cui valori sono funzioni), che si scrive con la notazione *`Domain => Range`* (cioè in questo caso `Int => Int`), mentre il nome del parametro segue le stesse regole che valgono per i nomi di tutti gli altri parametri formali. La definizione della funzione di ordine superiore è allora:

```
def sum(f: Int => Int, a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f, a + 1, b)
```

Adesso, le funzioni specializzate `sumInts`, `sumSquares` e `sumFactorials` possono essere riscritte come applicazioni di `sum`. Quando poi si applica `sum`, si deve specificare un primo parametro attuale di tipo funzione (`Int => Int`); come ogni parametro attuale, esso è un'espressione che genera un valore, e qui in particolare il valore generato è una funzione. La più semplice espressione che genera un valore di tipo funzione è quella costituita dal solo nome di una funzione, *senza* le parentesi che sarebbero necessarie per invocarla: ad esempio, la valutazione dell'espressione `square` produce appunto come valore la funzione `square`, che è di tipo `Int => Int`. Allora, passando in particolare

come parametri le funzioni `id`, `square` e `factorial` definite prima, si possono riscrivere `sumInts`, `sumSquares` e `sumFactorials` come invocazioni della funzione `sum`:

```
def sumInts(a: Int, b: Int) = sum(id, a, b)
def sumSquares(a: Int, b: Int) = sum(square, a, b)
def sumFactorials(a: Int, b: Int) = sum(factorial, a, b)
```

2 Funzioni anonime

Negli esempi visti finora, all'atto dell'invocazione di `sum` si è istanziato il parametro attuale di tipo funzione con una funzione precedentemente definita via `def`. Ad esempio,

```
def sumSquares(a: Int, b: Int) = sum(square, a, b)
```

fa riferimento alla funzione `square`, definita separatamente tramite

```
def square(x: Int): Int = x * x
```

In generale, risulta scomodo e prolisso dover definire ogni funzione che viene usata come parametro, soprattutto se la definizione è molto semplice (ad esempio `x * x`, nel caso di `square`). Queste ridondanze possono essere evitate grazie alle **funzioni anonime** (chiamate anche *function literal*, *lambda function*, *lambda expression* o semplicemente *lambda*[2]), che sono in sostanza dei letterali per la descrizione di valori di tipo funzione. Ad esempio, la funzione `square` può essere espressa come funzione anonima nel modo seguente:

```
(x: Int) => x * x
```

Se ci sono più argomenti, essi vengono separati dalla virgola; ad esempio:

```
(x: Int, y: Int) => x + y
```

Quando si definisce una qualunque funzione è necessario stabilire il dominio, il codominio e il metodo di calcolo (il corpo). Tuttavia, come è possibile osservare dagli esempi, la sintassi delle funzioni anonime (nella sua forma più generale) è

$$\texttt{(}x1\texttt{: }T1\texttt{, ..., }xn\texttt{: }Tn\texttt{) => }E$$

che consente di specificare solo il dominio (i tipi $T1, \ldots, Tn$ dei parametri) e il metodo di calcolo (l'espressione E).[3] Il codominio (cioè il tipo restituito) non può essere indicato esplicitamente, ma deve invece essere dedotto automaticamente. Di conseguenza, le funzioni anonime possono essere utilizzate solo quando il compilatore è in grado di inferire il codominio.

[2] Il termine "lambda" fa riferimento al λ-calcolo, poiché tale modello di calcolo, per semplicità, non ha un meccanismo di definizione, dunque le funzioni possono essere rappresentate esclusivamente in forma anonima, tramite letterali.

[3] Il simbolo `=>` è lo stesso impiegato nei tipi funzionali, ma nelle funzioni anonime esso indica, formalmente, la relazione tra i parametri formali e il metodo di calcolo della funzione.

Utilizzando le funzioni anonime, si possono riscrivere le funzioni `sumInts` e `sumSquares` come applicazioni di `sum` senza bisogno di definire separatamente le funzioni `id` e `square`; invece, per `sumFactorials` conviene mantenere la definizione separata di `factorial`, poiché essa è ricorsiva, e non si può definire (facilmente) una funzione anonima ricorsiva (dato che non si ha a disposizione un nome da usare all'interno della funzione per riferirsi alla funzione stessa al fine di chiamarla ricorsivamente).

```
def sumInts(a: Int, b: Int) = sum((x: Int) => x, a, b)
def sumSquares(a: Int, b: Int) = sum((x: Int) => x * x, a, b)
def sumFactorials(a: Int, b: Int) = sum(factorial, a, b)
```

Spesso, quando si usano le funzioni anonime, il compilatore è in grado di inferire non solo il codominio, ma anche il dominio. Allora, si possono omettere i tipi dei parametri, scrivendo ad esempio

```
(x, y) => x + y
```

invece di

```
(x: Int, y: Int) => x + y
```

Se una funzione anonima ha un solo parametro di cui non si specifica esplicitamente il tipo, è consentito omettere anche le parentesi; ad esempio,

```
(x: Int) => x * x
```

diventa

```
(x) => x * x
// oppure
x => x * x
```

Le invocazioni di `sum` nelle precedenti definizioni di `sumInts` e `sumSquares` sono esempi di casi in cui ciò è possibile, poiché il compilatore vede che le funzioni anonime vengono immediatamente usate come primo parametro di `sum`, dunque può dedurre che il dominio e il codominio di tali funzioni coincidono con quelli imposti dal tipo di tale parametro (`Int => Int`):

```
def sumInts(a: Int, b: Int) = sum(x => x, a, b)
def sumSquares(a: Int, b: Int) = sum(x => x * x, a, b)
```

2.1 Valutazione

Ogni volta che si introduce un nuovo tipo di espressione, come in questo caso le funzioni anonime, è necessario specificare come esso venga valutato nel modello di sostituzione. In questo caso, però, non è necessario definire nuovi meccanismi di valutazione, poiché una funzione anonima

$$(x1: T1, \ldots, xn: Tn) \Rightarrow E$$

può sempre essere espressa come:

```
def f(x1: T1, ..., xn: Tn) = E; f
```

1. si definisce con `def` una funzione con un nome `f` (ovvero non anonima);
2. si valuta immediatamente l'espressione `f`, che restituisce il valore della funzione appena definita (siccome la definizione e l'espressione sono separate dal punto e virgola, esse vengono valutate in sequenza e viene restituito il valore dell'espressione dopo il punto e virgola).

L'unico vincolo è che `f` sia un *nome nuovo*, cioè non precedentemente usato nel contesto in cui compare.

L'esistenza di questa riscrittura significa appunto che le funzioni anonime non implicano nuovi meccanismi nel linguaggio, ma costituiscono solo dello *zucchero sintattico* (*syntactic sugar*), una sintassi più comoda per qualcosa che può comunque essere espresso con gli altri costrutti del linguaggio.