Azzolini Riccardo 2021-03-29

Currying

1 Motivazione

Si consideri la funzione `sum` introdotta in precedenza:

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

Essa era stata impiegata per definire le funzioni specializzate `sumInts`, `sumSquares` e `sumFactorials`:

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

Ciascuna di queste ultime tre funzioni prende come argomenti gli estremi dell'intervallo e li passa alla funzione `sum` *senza modificarli*: in sostanza, si usa la funzione `sum` a tre argomenti per definire delle funzioni a due argomenti, istanziando solo il primo argomento con un valore specifico. Ci si può allora chiedere se sia possibile evitare di passare gli ultimi due parametri all'atto della definizione delle funzioni `sumInts`, ecc., in modo da rendere le definizioni più compatte e meno ridondanti.[1]

La risposta è sì, e la soluzione non richiede neanche nuove funzionalità del linguaggio: è sufficiente poter restituire le funzioni come valori dalle funzioni. L'idea è quella di definire `sum` come una funzione che prende come argomento *solo* la funzione `f: Int => Int`, e non dipende direttamente dagli estremi `a` e `b`, ma invece costruisce e restituisce un'altra funzione `(Int, Int) => Int` che riceve gli argomenti `a` e `b` ed effettua la somma con la funzione `f` prestabilita:

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

[1] Chiaramente, in questi piccoli esempi illustrativi una semplificazione del genere non dà grandi vantaggi, ma essa diventa più concretamente utile nelle situazioni maggiormente complicate che si possono incontrare in programmi di dimensioni più grandi.

Si osservi che questa funzione è all'ordine superiore sia per il parametro `f` che per il valore restituito, entrambi di tipo funzione.

Adesso, data questa versione di `sum`, le funzioni `sumInts`, ecc. possono essere definite in questo modo:

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

Infatti, quando si istanzia il primo (tecnicamente unico) argomento di `sum` si ottiene una funzione nei restanti due argomenti, che può essere associata a un nome senza bisogno di specificare esplicitamente i restanti argomenti. È importante notare che si usa la sintassi dei nomi senza argomenti, ma i tipi di questi nomi sono tipi funzionali, dunque i nomi possono essere applicati come ogni altra funzione; ad esempio:

```
sumInts(2, 3) + sumSquares(2, 3) + sumFactorials(2, 3)
```

Volendo, le funzioni restituite da `sum` potrebbero essere usate direttamente, senza associarle a dei nomi:

```
sum(x => x)(2, 3) + sum(x => x * x)(2, 3) + sum(factorial)(2, 3)
```

Questo funziona perché, considerando ad esempio `sum(factorial)(2, 3)`:

1. `sum(factorial)` applica la funzione `sum` alla funzione `factorial`, dunque restituisce la funzione di somma dei fattoriali (quella che in precedenza si era chiamata `sumFactorials`), il cui tipo è `(Int, Int) => Int`;
2. la funzione restituita viene poi immediatamente applicata agli argomenti `(2, 3)`.

In generale, *l'applicazione di funzione è associativa a sinistra*: ad esempio, l'espressione `sum(factorial)(2, 3)` corrisponde a `(sum(factorial))(2, 3)`.

2 Funzioni con liste di argomenti multiple

La versione funzione `sum` appena presentata è comoda da usare, ma la sua definizione è piuttosto ridondante:

- la definizione della funzione innestata `sumF` è una definizione temporanea, utilizzata solo per definire il valore da restituire;
- il tipo di `sumF` è esattamente il tipo restituito da `sum`.

Al fine di evitare questa ridondanza, Scala fornisce una sintassi apposita per definire le funzioni che restituiscono funzioni che hanno le due caratteristiche appena elencate. Tale sintassi sono le **liste di argomenti multiple**:

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

Questo tipo di definizione combina le liste di parametri della funzione esterna (qui `(f: Int => Int)`) e di quella innestata (qui `(a: Int, b: Int)`), seguite dal tipo restituito della funzione innestata. Anche il corpo rimane sostanzialmente uguale a quello della funzione innestata, con la sola differenza che le eventuali chiamate ricorsive devono fare riferimento non alla funzione innestata, che non ha più un nome, ma piuttosto alla funzione esterna, e quindi devono istanziare "da capo" tutti gli argomenti (qui è necessario scrivere `sum(f)(a + 1, b)` invece di `sumF(a + 1, b)`).

La funzione `sum` così definita può essere applicata esattamente come la definizione precedente, istanziando solo la prima lista di argomenti per definire funzioni specializzate[2] o istanziando subito entrambe le liste per eseguire immediatamente i calcoli:

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

sumInts(2, 3) + sumSquares(2, 3) + sumFactorials(2, 3)

sum(x => x)(2, 3) + sum(x => x * x)(2, 3) + sum(factorial)(2, 3)
```

Usando la notazione dei tipi funzionali, il tipo di `sum` è

```
(Int => Int) => ((Int, Int) => Int)
```

cioè appunto una funzione che prende come argomento una funzione `(Int => Int)` e restituisce una funzione `(Int, Int) => Int`. Siccome i tipi funzionali *associano a destra*, il tipo di `sum` può essere scritto omettendo le parentesi intorno al tipo della funzione restituita:

```
(Int => Int) => (Int, Int) => Int
```

2.1 Espansione di funzioni con liste di argomenti multiple

Le funzioni con liste di argomenti multiple, come le funzioni anonime, sono sostanzialmente zucchero sintattico, cioè possono essere riportate ai costrutti gestibili dal modello di sostituzione mediante un processo di riscrittura/espansione.

[2] Per via di una regola del linguaggio Scala, il compilatore accetta un'applicazione di una funzione che non istanzia tutte le liste di argomenti solo se tale applicazione compare in un contesto in cui è già previsto un valore di tipo funzione: ad esempio, se si sta definendo un nome è necessario indicare esplicitamente il tipo di tale nome. In alternativa, esistono alcune sintassi (basate sull'underscore, `_`) che possono essere usate per far accettare applicazioni del genere anche in contesti in cui il compilatore non è in grado di dedurre che il tipo del valore previsto è una funzione.

Una definizione di funzione con liste di argomenti multiple,

$$\texttt{def } f(args_1)...(args_n) = E$$

con $n > 1$, è equivalente a

$$\texttt{def } f(args_1)...(args_{n-1}) = \{\texttt{def } g(args_n) = E;\ g\}$$

dove g è un nome nuovo. In alternativa, la riscrittura può essere definita utilizzando una funzione anonima:

$$\texttt{def } f(args_1)...(args_{n-1}) = (args_n \texttt{ => } E)$$

ricordando che le funzioni anonime sono a loro volta zucchero sintattico, si osserva che queste due riscritture sono del tutto equivalenti, perché riscrivendo la funzione anonima usata nella seconda forma ci si riporta esattamente alla prima forma.

In pratica, la riscrittura appena mostrata elimina l'ultima lista di argomenti, trasformandola nella costruzione e restituzione esplicita di una funzione. Questo processo può essere iterato n volte, fino a ottenere una definizione di `f` senza argomenti, nella quale vengono costruite n funzioni anonime innestate:

$$\texttt{def } f(args_1)...(args_n) = E$$

equivale a

$$\texttt{def } f = (args_1 \texttt{ => } (args_2 \texttt{ => } ... \ (args_n \texttt{ => } E) \ ... \ ))$$

Le definizioni delle funzioni anonime, come i tipi funzionali, sono *associative a destra*, dunque si possono omettere le parentesi:

$$\texttt{def } f = args_1 \texttt{ => } args_2 \texttt{ => } ... \ args_n \texttt{ => } E$$

Si noti che invece, come già detto, l'applicazione delle funzioni è *associativa a sinistra*.

Lo stile di definizione delle funzioni come espressioni formate da funzioni anonime innestate, ciascuna con un solo argomento, è chiamato **currying**, dal nome del logico Haskell Brooks Curry (1900–1982), che definì formalmente questo meccanismo.