Azzolini Riccardo 2021-03-11

Ricorsione e iterazione

1 Ricorsione

Le strutture di controllo di iterazione non sono disponibili nei linguaggi funzionali puri, perché sono fortemente legate al paradigma imperativo (ad esempio, la condizione di un ciclo `while` può diventare falsa solo se cambia il valore di almeno una delle variabili che vi compaiono). Tuttavia, l'iterazione è sostanzialmente un modo "sporco" di fare ciò che in matematica viene fatto con la ricorsione. Un classico esempio di funzione ricorsiva è la funzione fattoriale (qui indicata con l'operatore postfisso !, ma si potrebbe usare equivalentemente la normale notazione delle funzioni),

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{altrimenti} \end{cases}$$

che può essere tradotta quasi direttamente in Scala:

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

(si noti che specificare il tipo restituito è obbligatorio, perché la funzione `factorial` è ricorsiva).

2 Conversione dell'iterazione in ricorsione

Nell'esempio precedente si è scritta una funzione ricorsiva a partire da una definizione matematica già ricorsiva. Adesso, si vuole mostrare che anche una tipica iterazione imperativa è riducibile a meccanismi ricorsivi.

Come esempio di algoritmo iterativo, si consideri il calcolo di $n!$ mediante la strategia suggerita dalla seguente definizione:

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

Il processo di calcolo può essere descritto usando due variabili:

- `product` contiene i prodotti parziali ottenuti a ogni fase del calcolo;

- `counter` è un contatore che controlla il processo di iterazione, e al tempo stesso fornisce la successione di numeri crescenti che vengono usati per calcolare i prodotti parziali.

Entrambe le variabili sono inizializzate a 1 (perché 1 è l'elemento neutro del prodotto e il primo numero considerato dalla definizione di fattoriale appena mostrata). Poi, a ogni iterazione, si modificano `product` e `counter` in questo modo:

```
product = product * counter    counter = counter + 1
```

Le iterazioni continuano fintantoché `counter` $\leq n$.

In Java, questa strategia è implementata dal seguente metodo:

```
public static int factorial_iter(int n) {
    int product = 1, counter = 1;
    while (counter <= n) {
        product = product * counter;
        counter = counter + 1;
    }
    return product;
}
```

Esso non può essere "riscritto" direttamente come funzione in Scala, poiché non si dispone di meccanismi di iterazione (a meno di utilizzare gli aspetti imperativi del linguaggio). Tuttavia, come già anticipato, è possibile simulare questa strategia iterativa utilizzando la ricorsione. A scopo illustrativo, la versione ricorsiva verrà prima definita in Java, e poi tradotta in Scala.

L'idea è introdurre un metodo ricorsivo che simuli le variabili mutable coinvolte nell'iterazione usando invece dei parametri formali:

- ogni chiamata ricorsiva corrisponde a un'iterazione;
- i valori passati come parametri a tale chiamata corrispondono ai valori aggiornati delle variabili;
- tutte le espressioni per il calcolo dei valori aggiornati da assegnare alle variabili vengono messe come parametri attuali della chiamata ricorsiva.

Nel caso del calcolo del fattoriale, il metodo ricorsivo deve avere ancora il parametro `n`, già presente nel metodo iterativo, ma in più anche due parametri `product` e `counter`, che simulano appunto le variabili del ciclo. L'implementazione di questo metodo è la seguente:

```
public static int factorial_rec(int n, int product, int counter) {
    if (counter <= n)
        return factorial_rec(n, product * counter, counter + 1);
    else
```

```
        return product;
}
```

- se quella che nella versione iterativa era la condizione del ciclo è ancora soddisfatta, si effettua una chiamata ricorsiva per eseguire una nuova "iterazione", passando come parametri attuali i valori aggiornati di `product` e `counter`;
- se invece la condizione del ciclo è falsa, viene restituito il valore di `product` calcolato nell'iterazione precedente, che corrisponde a $n!$.

Il metodo `factorial_rec` così definito funziona, ma non implementa la fase di inizializzazione delle variabili. Allora, si definisce un altro metodo `factorial` che, dato un singolo parametro `n`, si occupa di fornire i valori iniziali per `product` e `counter`:

```
public static int factorial(int n) {
    return factorial_rec(n, 1, 1);
}
```

Ora, quest'implementazione ricorsiva del calcolo del fattoriale può essere riscritta in Scala:

```
def factorial_rec(n: Int, product: Int, counter: Int): Int =
  if (counter <= n)
    factorial_rec(n, counter * product, counter + 1)
  else
    product

def factorial(n: Int) = factorial_rec(n, 1, 1)
```

Potrebbe sembrare che convertendo dall'iterazione alla ricorsione si stia "snaturando" il problema, ma in realtà è solo questione di abitudine: l'iterazione è una possibile implementazione della definizione matematica $n! = 1 \cdot 2 \cdots n$, ed è quella che tipicamente si preferisce in ambito imperativo, ma non è in generale la "più naturale".

Si potrebbe anche osservare che la funzione ricorsiva `factorial_rec` è apparentemente meno efficiente di `factorial_iter` dal punto di vista della memoria, perché crea sullo stack un record di attivazione per ogni iterazione, ma si vedrà più avanti che in molti contesti ciò non è vero.

3 Altro esempio: l'algoritmo di Euclide

Un altro esempio di conversione dall'iterazione alla ricorsione è l'algoritmo di Euclide per il calcolo del massimo comune divisore (MCD, o in inglese *gcd*, *greatest common divisor*), che in Java può essere implementato come segue:

```
public static int gcd(int a, int b) {
    while (b > 0) {
        int c = a % b;
        a = b;
        b = c;
    }
    return a;
}
```

Guardando un esempio di esecuzione di questo metodo,

Iterazione	(a, b)	`b > 0`	`a % b`	(a', b')
1	$(45, 25)$	`true`	20	$(25, 20)$
2	$(25, 20)$	`true`	5	$(20, 5)$
3	$(20, 5)$	`true`	0	$(5, 0)$
–	$(5, 0)$	`false`	–	–

si può osservare che, a ogni iterazione, il problema di calcolare `gcd(a, b)` viene ridotto al problema di calcolare `gcd(b, a % b)`:

Iterazione	Riduzione
1	`gcd(45, 25)` = `gcd(25, 45 % 25)` = `gcd(25, 20)`
2	`gcd(25, 20)` = `gcd(20, 25 % 20)` = `gcd(20, 5)`
3	`gcd(20, 5)` = `gcd(5, 20 % 5)` = `gcd(5, 0)`
–	`gcd(5, 0)` = `5`

A partire da quest'osservazione si può scrivere una versione ricorsiva del metodo per il calcolo dell'MCD,

```
public static int gcd_rec(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd_rec(b, a % b);
}
```

e infine tale metodo può essere riscritto in Scala:

```
def gcd(a: Int, b: Int): Int =
  if (b == 0) a else gcd(b, a % b)
```

Anche se l'implementazione ricorsiva è stata ricavata tramite un ragionamento diverso, di fatto la tecnica applicata è la stessa usata prima per il fattoriale: l'`if-else` implementa

la condizione del ciclo,[1] e il passaggio dei parametri formali `a` e `b` nelle chiamate ricorsive viene usato per simulare la modifica delle variabili. In questo caso, però, come suggerito dall'osservazione sulla riduzione del problema, non è stato necessario aggiungere nuovi parametri al metodo ricorsivo, perché il ciclo riutilizzava già i parametri formali come variabili modificate, e c'era in più solo una variabile temporanea `c`, necessaria perché il codice imperativo modifica "una variabile alla volta", mentre con le chiamate ricorsive si possono calcolare insieme i nuovi valori di tutti i parametri, senza sovrascrivere prematuramente alcuni dei valori da usare nei calcoli.

[1] Qui la condizione del ciclo è scritta in forma invertita, in modo da mettere in evidenza nel ramo `if` il caso base, ma mantenere la condizione originale `b > 0` e scambiare i rami dell'`if-else` sarebbe del tutto equivalente.