Azzolini Riccardo 2021-03-18

Tail-recursion

1 Efficienza della ricorsione

Molte strutture dati (liste concatenate, alberi, ecc.) sono definite in modo ricorsivo, dunque gli algoritmi ricorsivi risultano spesso più naturali di quelli iterativi. Tuttavia, eseguendo un numero eccessivo di chiamate ricorsive si rischia di saturare la memoria disponibile per lo stack, e ogni chiamata di funzione ha anche un certo costo in termini di tempo (per la gestione del record di attivazione), quindi nei linguaggio di programmazione "tradizionali" si tendono a preferire le versioni iterative di algoritmi naturalmente ricorsivi.

In realtà, come si vedrà a breve, ci sono delle situazioni in cui la ricorsione può essere implementata in un modo che *non* è meno efficiente dell'iterazione (se il compilatore/interprete lo supporta), e allora le due strategie possono essere considerate equivalenti.

2 Valutazione di un'applicazione di funzione

Nella pratica, i linguaggi funzionali sono implementati, come quelli imperativi, sulla macchina di von Neumann, usando il modello di esecuzione a stack, dunque le considerazioni sull'efficienza della ricorsione andrebbero fatte ragionando su tale modello. Tuttavia, il principio che permette un'implementazione efficiente della ricorsione in determinate situazioni vale anche se si ragiona a più alto livello, considerando il modello di sostituzione, che è il modello della macchina astratta sulla quale si "immagina" che i programmi funzionali vengano eseguiti. Il primo passo per studiare questo principio è formalizzare la valutazione delle applicazioni di funzioni nel modello di sostituzione.

Si consideri una generica definizione di funzione:

$$\texttt{def } f\texttt{(}x_1\texttt{: } T_1\texttt{, ..., } x_n\texttt{: } T_n\texttt{): } T \texttt{ = } B$$

La valutazione dell'applicazione di funzione $f\texttt{(}e_1\texttt{, ..., } e_n\texttt{)}$ con la strategia call-by-value avviene, intuitivamente, nel modo seguente:

1. Si valutano le espressioni $e_1, \ldots, e_n$; siano $v_1, \ldots, v_n$ i valori risultanti.
2. si rimpiazza l'applicazione di funzione $f\texttt{(}e_1\texttt{, ..., } e_n\texttt{)}$ con il corpo B, in cui i parametri formali $x_1, \ldots, x_n$ vengono sostituiti dai valori dei parametri attuali $v_1, \ldots, v_n$.

Ciò è formalizzato come un rewriting dell'intero programma:[1]

```
def f(x_1: T_1, ..., x_n: T_n): T = B
...
f(e_1, ..., e_n)
...
```
$\longrightarrow$
```
def f(x_1: T_1, ..., x_n: T_n): T = B
...
[v_1/x_1, ..., v_n/x_n]B
...
```

La notazione $[v_1/x_1, \ldots, v_n/x_n]$ indica una **sostituzione** (essa è la notazione standard per le sostituzioni nell'ambito dei linguaggi formali); in particolare, questa sostituzione sostituisce a ciascuna variabile (parametro formale) x_i il valore v_i. L'applicazione di questa sostituzione, che qui per comodità viene indicata scrivendo la sostituzione *a sinistra* dell'espressione a cui la si applica, $[v_1/x_1, \ldots, v_n/x_n]B$, ha appunto l'effetto di sostituire tutte le occorrenze di una variabile x_i in B con il valore v_i.

3 Esempi di rewriting di funzioni ricorsive

Uno degli esempi di funzioni ricorsive viste in precedenza è la funzione `gcd`:

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

Si vuole studiare la valutazione dell'applicazione `gcd(14, 21)`. Per semplicità, nello scrivere i passi di valutazione si mostrerà solo la parte del programma relativa all'applicazione della funzione, mentre in teoria il modello di sostituzione dovrebbe operare sull'intero programma, che comprende anche la definizione della funzione.

Siccome i parametri attuali presenti nell'applicazione `gcd(14, 21)` sono letterali, che per semplicità si considerano già valutati (assimilandoli ai valori astratti che essi rappresentano), la prima cosa da fare è applicare la regola di riscrittura per le applicazioni di funzione appena presentata:

```
gcd(14, 21)
    → [14/a, 21/b](if (b == 0) a else gcd(b, a % b))
```

Adesso si applica la sostituzione; ciò avviene contemporaneamente su tutte le variabili sostituite, in un unico passo di valutazione:

```
    → if (21 == 0) 14 else gcd(21, 14 % 21)
```

Ora si procede normalmente con la valutazione dell'espressione (applicando le solite regole di riscrittura nell'ordine da sinistra a destra, considerando le precedenze degli operatori):

```
    → if (false) 14 else gcd(21, 14 % 21)
    → gcd(21, 14 % 21)
```

[1] Formalmente, il modello di sostituzione opera sempre sull'intero programma, anche se ogni passo di valutazione "modifica" solo una parte del programma.

A questo punto, l'espressione è di nuovo un'applicazione della funzione `gcd`. Per valutarla, bisogna innanzitutto valutare il secondo argomento, che non è un letterale:

$\rightarrow$ `gcd(21, 14)`

Poi, si applica di nuovo la regola di rewriting di un'applicazione di funzione, e tutto il processo di valutazione svolto finora si ripete:

$\rightarrow$ `[21/a, 14/b](if (b == 0) a else gcd(b, a % b))`
$\rightarrow$ `if (14 == 0) 21 else gcd(14, 21 % 14)`
$\rightarrow$ `if (false) 21 else gcd(14, 21 % 14)`
$\rightarrow$ `gcd(14, 21 % 14)`
$\rightarrow$ `gcd(14, 7)`

Ancora una volta, ci si trova a dover fare il rewriting di un'applicazione di `gcd`. Per evitare di scrivere i passaggi ripetitivi, si indica una sequenza di riduzioni non mostrate esplicitamente con una freccia doppia, $\twoheadrightarrow$.

$\twoheadrightarrow$ `gcd(7, 14 % 7)`
$\rightarrow$ `gcd(7, 0)`

Quando il secondo parametro attuale è `0`, la condizione dell'`if-else` diventa `true`, dunque la valutazione della funzione ricorsiva termina:

$\twoheadrightarrow$ `if (0 == 0) 7 else gcd(0, 7 % 0)`
$\rightarrow$ `if (true) 7 else gcd(0, 7 % 0)`
$\rightarrow$ `7`

Un altro esempio è la funzione fattoriale che si era ricavata traducendo direttamente la definizione matematica ricorsiva di $n!$:

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

La valutazione di una sua applicazione avviene in questo modo:

`factorial(4)`
$\rightarrow$ `[4/n](if (n == 0) 1 else n * factorial(n - 1))`
$\rightarrow$ `if (4 == 0) 1 else 4 * factorial(4 - 1)`
$\rightarrow$ `if (false) 1 else 4 * factorial(4 - 1)`
$\rightarrow$ `4 * factorial(4 - 1)`
$\rightarrow$ `4 * factorial(3)`

Come nel caso di `gcd`, a un certo punto ci si trova a dover valutare di nuovo un'applicazione della stessa funzione `factorial`, la chiamata ricorsiva. In questo caso, però, c'è una differenza: la chiamata ricorsiva compare all'interno di un'espressione più complessa, mentre prima costituiva "da sola" l'intera espressione da valutare. Comunque, le regole di valutazione rimangono le stesse, quindi si procede nel solito modo:

$\rightarrow$ `4 * ([3/n](if (n == 0) 1 else n * factorial(n - 1)))`
$\rightarrow$ `4 * (if (3 == 0) 1 else 3 * factorial(3 - 1))`
$\twoheadrightarrow$ `4 * (3 * factorial(2))`

Di nuovo, arriva a dover valutare una chiamata ricorsiva, e così via, fino a quando si raggiunge al caso base della ricorsione:

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`
$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0))))`
$\twoheadrightarrow$ `4 * (3 * (2 * (1 * 1)))`

Infine, si valutano una a una le moltiplicazioni:

$\rightarrow$ `4 * (3 * (2 * 1))`
$\rightarrow$ `4 * (3 * 2)`
$\rightarrow$ `4 * 6`
$\rightarrow$ `24`

Considerando solo i passi in cui si hanno le chiamate ricorsive e la valutazione delle moltiplicazioni, si può osservare che questo processo di valutazione ha una struttura particolare:

`factorial(4)`
- $\twoheadrightarrow$ `4 * factorial(3)`
- $\twoheadrightarrow$ `4 * (3 * factorial(2))`
- $\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`
- $\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0))))`
- $\twoheadrightarrow$ `4 * (3 * (2 * (1 * 1)))`
- $\rightarrow$ `4 * (3 * (2 * 1))`
- $\rightarrow$ `4 * (3 * 2)`
- $\rightarrow$ `4 * 6`
- $\rightarrow$ `24`

In termini informali, si ha una fase di "espansione" seguita da una fase di "contrazione".

- Nella fase di espansione, il processo di valutazione costruisce una catena di operazioni che dovranno essere effettuate al termine delle chiamate ricorsive. Infatti, ogni chiamata ricorsiva introduce un operatore di moltiplicazione che non può essere immediatamente valutato, perché la valutazione di uno dei suoi due operandi (la chiamata ricorsiva) non è ancora finita.
- Solo quando si giunge al caso base della ricorsione si ottiene una moltiplicazione i cui operandi sono entrambi valutati, `1 * 1`. Allora, la fase di espansione termina e inizia la fase di contrazione, nella quale le operazioni "lasciate in sospeso" vengono effettuate.

Se invece si valuta un'applicazione della versione "iterativa" del fattoriale (quella ricavata dall'implementazione iterativa in Java),

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

def factorial2(n: Int) = fact_iter(n, 1, 1)
```

si osserva un comportamento che è molto diverso da quello di `factorial`, ed è invece analogo al comportamento di `gcd`:

```
factorial2(4)
    → [4/n](fact_iter(n, 1, 1))
    → fact_iter(4, 1, 1)
    →* if (1 <= 4) fact_iter(4, 1 * 1, 1 + 1) else 1
    →* fact_iter(4, 1, 2)
    →* if (2 <= 4) fact_iter(4, 2 * 1, 2 + 1) else 1
    →* fact_iter(4, 2, 3)
    →* if (3 <= 4) fact_iter(4, 3 * 2, 3 + 1) else 2
    →* fact_iter(4, 6, 4)
    →* if (4 <= 4) fact_iter(4, 4 * 6, 4 + 1) else 6
    →* fact_iter(4, 24, 5)
    →* if (5 <= 4) fact_iter(4, 5 * 24, 5 + 1) else 24
    →* 24
```

Omettendo anche qui i passi che non sono chiamate ricorsive, e confrontando la valutazione con quella di `factorial`, la differenza diventa particolarmente evidente:

```
factorial(4)
    ↠ 4 * factorial(3)
    ↠ 4 * (3 * factorial(2))
    ↠ 4 * (3 * (2 * factorial(1)))
    ↠ 4 * (3 * (2 * (1 * factorial(0))))
    ↠ 4 * (3 * (2 * (1 * 1)))
    → 4 * (3 * (2 * 1))
    → 4 * (3 * 2)
    → 4 * 6
    → 24
```

```
factorial2(4)
    ↠ fact_iter(4, 1, 1)
    ↠ fact_iter(4, 1, 2)
    ↠ fact_iter(4, 2, 3)
    ↠ fact_iter(4, 6, 4)
    ↠ fact_iter(4, 24, 5)
    ↠ 24
```

Nella valutazione di `factorial2`, o meglio di `fact_iter` (`factorial2` è solo una funzione "wrapper" non ricorsiva che fornisce i valori iniziali degli argomenti di `fact_iter`), la lunghezza dell'espressione rimane sostanzialmente costante, cioè non si hanno le fasi di espansione e contrazione che si presentavano nel caso di `factorial`, perché la chiamata ricorsiva è sempre "da sola" (non parte di un'espressione più complessa) quando viene valutata.

A ogni passo, i valori dei parametri attuali di `fact_iter` forniscono una descrizione completa dello stato attuale del processo di valutazione, cioè sono le uniche informazioni di cui è necessario tener traccia per proseguire con la valutazione: se il processo venisse ipoteticamente interrotto prima della fine, ad esempio dopo i passi

```
factorial2(4)
    ↠ fact_iter(4, 1, 1)
    ↠ fact_iter(4, 1, 2)
    ↠ fact_iter(4, 2, 3)
```

potrebbe essere ripreso semplicemente fornendo come argomenti a `fact_iter` i valori `4`, `2` e `3`:

```
fact_iter(4, 2, 3)
    ↠ fact_iter(4, 6, 4)
    ↠ fact_iter(4, 24, 5)
    ↠ 24
```

In pratica, anche se la forma della funzione `fact_iter` è ricorsiva, il suo processo di esecuzione è essenzialmente iterativo: a ogni passo si ripete lo stesso schema di valutazione, solo su parametri diversi. Tutto questo discorso non vale invece nel caso di `factorial`, il cui processo di valutazione utilizza delle informazioni "nascoste", non esplicitate nel

codice del programma: l'interprete tiene traccia, tramite il modello a stack, delle operazioni che non possono essere effettuate immediatamente, e dovranno invece essere eseguite al termine delle invocazioni ricorsive. Nel modello di sostituzione, tali informazioni sono rappresentate dall'espressione sempre più complessa che viene costruita nella fase di espansione; per riprendere un processo di valutazione interrotto, sarebbe necessario ricordare non solo il parametro attuale da passare a `factorial`, ma anche tutta l'espressione in cui la chiamata ricorsiva a `factorial` è inserita.

In conclusione, si può dire che `fact_iter` è di fatto una funzione iterativa, mentre `factorial` è in un certo senso "veramente" ricorsiva.

4 Tail-recursion

Il tipo di ricorsione effettuato da `fact_iter`, che equivale all'iterazione, prende il nome di **tail-recursion** (**ricorsione di coda** o **in coda**).

Definizione: Una funzione ricorsiva è **tail-recursive** se l'invocazione ricorsiva è l'*ultima azione* compiuta dalla funzione.

Ad esempio, la funzione

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

è tail-recursive, perché la chiamata ricorsiva presente nel ramo `else` è l'ultima operazione effettuata. Analogamente, anche la funzione `fact_iter` è tail-recursive. Invece, la funzione

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

non è tail-recursive: nel ramo `else`, dopo aver eseguito la chiamata ricorsiva, resta ancora da calcolare il valore del prodotto tra `n` e il risultato di tale chiamata.

Si noti che la chiamata ricorsiva deve essere l'ultima azione in *tutti i possibili percorsi di esecuzione* della funzione che comprendono tale chiamata: ad esempio, una funzione come

```
def f(x: Int): Int =
  if (x > 0) f(x - 1)
  else if (x < 0) f(x + 1) + 1
  else x
```

non è tail-recursive, perché la chiamata ricorsiva è l'ultima operazione compiuta quando il predicato `x > 0` è vero, ma *non* è l'ultima operazione quando invece `x > 0` è falso e `x < 0` è vero (dopo la chiamata ricorsiva, rimane da sommare uno al suo risultato).

4.1 Implementazione

Il processo di esecuzione di una funzione tail-recursive è essenzialmente iterativo: siccome l'unica informazione necessaria per la gestione della valutazione sono i valori correnti dei parametri attuali, il record di attivazione della funzione che effettua la chiamata ricorsiva può essere riutilizzato per la funzione che viene invocata, senza bisogno di allocare un nuovo record. In pratica, i parametri formali presenti nel record di attivazione vengono "aggiornati" tra un'iterazione e la successiva, cioè sono gestiti esattamente come le variabili mutable in un ciclo realizzato in un linguaggio imperativo.

Quando poi l'ultima chiamata ricorsiva termina, il controllo (e il valore restituito) ritorna direttamente al chiamante della funzione ricorsiva, senza bisogno di rimuovere uno a uno i record di attivazione di tutte le chiamate ricorsive.

Eseguire una funzione tail-recursive in modo iterativo comporta il risparmio del tempo e della memoria necessari a gestire i record di attivazione, ed elimina il rischio di saturare la memoria disponibile per lo stack: l'esecuzione diventa a tutti gli effetti efficiente quanto quella di una funzione iterativa "tradizionale". Tuttavia, una funzione tail-recursive non viene "spontaneamente" eseguita in modo iterativo: perché ciò avvenga, il compilatore o interprete deve individuare la tail-recursion e applicare un'ottimizzazione che la trasformi in iterazione. Tale ottimizzazione è importante, e quindi diffusa, soprattutto nei linguaggi funzionali (che hanno la ricorsione come unico meccanismo di "iterazione"), ma è possibile anche nei linguaggi imperativi (e infatti è implementata, ad esempio, dai principali compilatori C).

4.2 Tail-recursion in Scala

Il compilatore/interprete Scala individua e ottimizza automaticamente le invocazioni tail-recursive *dirette*, mentre non è in grado di ottimizzare le invocazioni tail-recursive tra funzioni *mutuamente ricorsive* (due o più funzioni diverse che si invocano a vicenda).

A volte, potrebbe capitare che il programmatore, nel modificare una funzione tail-recursive, la renda accidentalmente non tail-recursive, introducendo così tutti i problemi di efficienza legati alla ricorsione "normale". Per evitare questo errore, il linguaggio Scala fornisce l'*annotazione* `@tailrec`, che permette di informare il compilatore che una funzione deve essere tail-recursive:

```
import scala.annotation.tailrec

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

In presenza di tale annotazione, il compilatore verifica che la funzione sia effettivamente tail-recursive, e in caso contrario segnala un errore. A parte l'eventuale segnalazione di un

errore, l'annotazione `@tailrec` non ha alcun effetto sul codice generato, perché appunto l'ottimizzazione della tail-recursion è automatica, ovvero il compilatore la applica ogni volta che è possibile farlo, anche in assenza di `@tailrec`.

Le annotazioni, presenti anche in Java, forniscono in generale un meccanismo di meta-programmazione (cioè un meccanismo che opera sul programma sorgente), permettendo di dare al compilatore delle informazioni aggiuntive, ad esempio per attivare controlli o ottimizzazioni aggiuntivi, oppure per indicare ad appositi strumenti di generare automaticamente del codice, ecc. Un'annotazione è definita da una classe che ne descrive il comportamento; tale classe deve essere importata nel programma sorgente che usa l'annotazione.

4.3 Tail-call

La possibilità di riutilizzare il record di attivazione corrente non è limitata alle invocazioni ricorsive: in generale, se l'ultima azione compiuta da una funzione f è l'invocazione di una qualunque funzione g (che non necessariamente coincide con f), allora è possibile riutilizzare il record di attivazione di f per eseguire anche g. In questo caso, si parla di **tail-call**.

Il compilatore Scala ottimizza la tail-recursion, ma non la tail-call.

4.4 Quando usare la tail-recursion

L'ottimizzazione della tail-recursion permette di scrivere in modo elegante, senza alcuna perdita di efficienza, gli algoritmi ricorsivi che si prestano naturalmente a una formulazione che sia appunto tail-recursive. Ci sono però altri algoritmi che possono essere espressi facilmente solo tramite funzioni contenenti invocazioni ricorsive non in coda, e trasformarli in forma tail-recursive può complicare molto il codice (di fatto, equivale a trasformarli in forma iterativa — si pensi ad esempio a quanto si complicano gli algoritmi di visita in profondità degli alberi nel passaggio dalla versione ricorsiva a quella iterativa, che richiede l'introduzione di una struttura stack esplicita per sostituire le informazioni memorizzate implicitamente dallo stack di esecuzione).

Per questo motivo, quando si scrive codice per cui l'efficienza non è particolarmente importante, ovvero quando è consigliabile preferire l'eleganza all'efficienza, è spesso meglio usare la ricorsione non di coda (così come nei linguaggi imperativi si userebbe la ricorsione al posto dell'iterazione), soprattutto considerando che la perdita di efficienza legata alla ricorsione è molto ridotta sulle macchine moderne. L'uso della tail-recursion può allora essere limitato ai casi in cui esso non risulta troppo complicato.