

Modello di sostituzione

1 Modello di sostituzione

La valutazione delle espressioni in Scala è basata sul **modello di sostituzione** (**substitution model**), nel quale i nomi sono rimpiazzati dalle loro definizioni. Tramite una sequenza di passi, la valutazione *riduce un'espressione a un valore*, che è un **termine** che non richiede ulteriori valutazioni. Un singolo passo di valutazione può essere chiamato *valutazione, riduzione o rewriting*.

Il modello di sostituzione è formalizzato dal **λ -calcolo**, un modello computazionale Turing completo (cioè di potenza equivalente alle macchine di Turing) che è stato introdotto da Alonzo Church nel 1936, e costituisce il fondamento teorico dei linguaggi funzionali.¹

Questo modello di valutazione può essere applicato quando le espressioni non hanno **effetti collaterali**, perché non è in grado di descrivere gli aspetti mutabile di un linguaggio, ad esempio la valutazione dell'espressione `x++` di un linguaggio come C o Java.

2 Terminazione

Una sequenza di passi di valutazione è significativa se termina, altrimenti non potrebbe mai produrre un valore, ma solo “dare problemi”. Allora, è naturale chiedersi se sia sempre possibile ridurre un'espressione a un valore in un *numero finito di passi*.

La risposta è negativa: considerando ad esempio la seguente funzione ricorsiva `loop`,²

```
def loop(): Int = loop()
```

¹In genere, i linguaggi funzionali sono molto più vicini al lambda calcolo di quanto i linguaggi imperativi siano vicini al loro modello teorico, la macchina di von Neumann.

²Al di là del problema della terminazione, su questa definizione si possono fare alcune osservazioni interessanti. Per prima cosa, la sintassi con le parentesi vuote `()` definisce una funzione senza parametri, che sostanzialmente equivale a una definizione senza parentesi, `def loop: Int = loop`. Inoltre, siccome il corpo della funzione è solo un uso della funzione stessa, il tipo restituito da `loop()` dipende solo dal tipo restituito da `loop()`, dunque qualunque tipo andrebbe bene, e allora il compilatore/interprete non ha alcuna informazione per dedurre il tipo restituito: bisogna per forza specificarlo esplicitamente.

si ha che la valutazione dell'espressione `loop()` non termina, perché ogni volta l'uso della funzione viene sostituito con il corpo della funzione, ma questo è ancora esattamente un uso della stessa funzione:

$$\text{loop}() \rightarrow \text{loop}() \rightarrow \text{loop}() \rightarrow \dots$$

3 Strategie di valutazione

Nel modello di sostituzione esistono due diverse strategie di valutazione delle funzioni:

- **Call-by-value (CBV)**: gli argomenti della funzione vengono valutati prima di effettuare il rewriting dell'applicazione di funzione.
- **Call-by-name (CBN)**: *la valutazione degli argomenti viene posticipata*, cioè il rewriting dell'applicazione della funzione viene applicato senza ridurre gli argomenti.

3.1 Esempi

Date le definizioni

```
def square(x: Double) = x * x
def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
```

si consideri l'espressione `sumOfSquares(5, 2 + 2)`. I passi per la sua valutazione sono:

- secondo la strategia call-by-value:

<code>sumOfSquares(5, 2 + 2)</code>	
<code>→ sumOfSquares(5, 4)</code>	[valutazione +]
<code>→ square(5) + square(4)</code>	[rewriting sumOfSquares]
<code>→ 5 * 5 + square(4)</code>	[rewriting square]
<code>→ 25 + square(4)</code>	[valutazione *]
<code>→ 25 + 4 * 4</code>	[rewriting square]
<code>→ 25 + 16</code>	[valutazione *]
<code>→ 41</code>	[valutazione +]

- secondo la strategia call-by-name:

```

sumOfSquares(5, 2 + 2)
  → square(5) + square(2 + 2)    [rewriting sumOfSquares]
  → 5 * 5 + square(2 + 2)        [rewriting square]
  → 25 + square(2 + 2)           [valutazione *]
  → 25 + (2 + 2) * (2 + 2)       [rewriting square]
  → 25 + 4 * (2 + 2)             [valutazione +]
  → 25 + 4 * 4                   [valutazione +]
  → 25 + 16                      [valutazione *]
  → 41                           [valutazione +]

```

Da questo esempio, sembra che la strategia call-by-name sia meno efficiente, perché l'espressione $2 + 2$ viene valutata due volte, mentre con la strategia call-by-value la si valuta una volta sola. In generale, questo è vero: esiste un teorema del λ -calcolo che afferma che una valutazione call-by-name può richiedere un numero di passi esponenziale rispetto al numero di passi richiesto con la strategia call-by-value. Ci sono però dei casi in cui la valutazione call-by-name risulta invece più corta. Ad esempio, un caso estremo è una funzione che ignora totalmente uno dei suoi parametri, come la seguente:

```
def squareFirst(x: Int, y: Int): Int = x * x
```

Infatti, con la strategia call-by-name il parametro attuale corrispondente a y non viene mai valutato,

```
squareFirst(7, 2 * 4) → 7 * 7 → 49
```

mentre con la strategia call-by-value il parametro verrebbe valutato inutilmente,

```
squareFirst(7, 2 * 4) → squareFirst(7, 8) → 7 * 7 → 49
```

dunque la strategia call-by-name consente di “risparmiare” tanti più passi di valutazione quanto più è complessa l'espressione passata come parametro y .

3.2 Confronto

Si può dimostrare che entrambe le strategie riducono un'espressione al medesimo valore se:

- l'espressione è costituita da *funzioni pure* (si vedrà più avanti cosa significa questo termine, ma per ora basta sapere che le funzioni pure sono l'unico tipo di funzioni trattate in questo corso);
- entrambe le valutazioni terminano.

Ciascuna di queste strategie ha un vantaggio:

- call-by-value ha il vantaggio che ogni parametro attuale viene valutato una sola volta;
- call-by-name ha il vantaggio che un parametro attuale di una funzione non viene valutato se il corrispondente parametro formale non viene utilizzato nel corpo della funzione.

3.3 Terminazione

Esiste un teorema del λ -calcolo che afferma che, se la valutazione call-by-value di un'espressione termina, allora anche la valutazione call-by-name di tale espressione termina, e le due valutazioni producono lo stesso risultato.

Non vale invece il viceversa, cioè può essere che la valutazione di un'espressione termini con la strategia call-by-name ma non con la strategia call-by-value. Ad esempio, date le definizioni

```
def loop(): Int = loop()
def squareFirst(x: Int, y: Int): Int = x * x
```

la valutazione call-by-name dell'espressione `squareFirst(1, loop())` termina perché il secondo argomento non viene usato nel corpo di `squareFirst`,

```
squareFirst(1, loop())
  → 1 * 1                [rewriting squareFirst]
  → 1                    [valutazione *]
```

mentre la valutazione call-by-name non termina perché non si finisce mai di valutare il secondo argomento:

```
squareFirst(1, loop())
  → squareFirst(1, loop())    [rewriting loop]
  → squareFirst(1, loop())    [rewriting loop]
  ⋮
```

3.4 Strategie di valutazione di Scala

Nei linguaggi funzionali è abbastanza comune che il programmatore possa scegliere tra entrambe le strategie di valutazione, e tipicamente la strategia call-by-value è quella di default (cioè che non richiede una sintassi particolare per selezionarla). Questo è il caso di Scala, che di default utilizza appunto la strategia call-by-value, mentre applica

la strategia call-by-name a un parametro se il tipo di tale parametro è preceduto dal simbolo =>.

Si noti che la strategia può appunto essere scelta separatamente per ciascun parametro di una funzione: non è obbligatorio applicare la stessa strategia per tutti i parametri di una stessa funzione. Ad esempio, la seguente funzione `constOne` valuta il parametro `x` con la strategia call-by-name, e `y` con la strategia call-by-value:

```
def constOne(x: Int, y: => Int): Int = 1
```

Siccome nessuno dei due parametri compare nel corpo, quando si chiama `constOne` viene valutato solo il primo argomento (a causa della strategia call-by-value). Di conseguenza, ad esempio, la valutazione dell'espressione `constOne(1 + 2, loop())` termina,

```
constOne(1 + 2, loop())
  → constOne(3, loop())    [valutazione + (CBV primo argomento)]
  → 1                      [rewriting constOne]
```

mentre la valutazione di `constOne(loop(), 1 + 2)` va in loop infinito:

```
constOne(loop(), 1 + 2)
  → constOne(loop(), 1 + 2) [rewriting loop() (CBV primo argomento)]
  → constOne(loop(), 1 + 2) [rewriting loop() (CBV primo argomento)]
  ⋮
```