Azzolini Riccardo 2021-05-10

Il problema della decomposizione

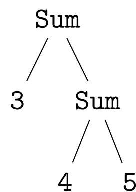
1 Esempio tipico: interprete per le espressioni aritmetiche

L'esempio che tipicamente si usa per illustrare il problema della decomposizione e le sue diverse soluzioni è l'implementazione di un interprete per le espressioni aritmetiche.

Innanzitutto, bisogna stabilire come rappresentare le espressioni, supponendo per ora che esse contengono unicamente numeri interi e l'operatore di somma $+$. Il modo più naturale per rappresentarle è tramite alberi[1] i cui nodi sono istanze di classi della seguente gerarchia:

```
trait Expr { /* ... */ }
class Number(n: Int) extends Expr { /* ... */ }
class Sum(e1: Expr, e2: Expr) extends Expr { /* ... */ }
```

Ad esempio, l'espressione $3 + (4 + 5)$ è rappresentata dall'albero



che in Scala è scritto come

```
new Sum(new Number(3), new Sum(new Number(4), new Number(5)))
```

Una volta definita la rappresentazione delle espressioni bisogna implementare le operazioni su di esse. Per operare su un'espressione è necessario visitare i nodi dell'albero, comportandosi in modo diverso a seconda del tipo di nodo. Ad esempio, per valutare un'espressione (l'operazione principale che un interprete deve svolgere) bisogna visitare in profondità l'albero e, per ogni nodo:

- se il nodo è un numero, restituire il valore del nodo;

[1] Gli alberi sono il modo più naturale di rappresentare le espressioni a prescindere dal linguaggio/paradigma di programmazione considerato: il fatto che ogni operatore abbia operandi che sono a loro volta espressioni corrisponde esattamente a una struttura ad albero, ovvero la notazione che si usa normalmente per le espressioni è sostanzialmente un formalismo lineare per rappresentare alberi.

- se il nodo è una somma, valutare ricorsivamente i suoi due operandi (corrispondenti ai sottoalberi del nodo) e restituire la somma dei due valori così ottenuti.

Il **problema della decomposizione** è proprio il problema di come implementare delle operazioni sui nodi di un albero che si comportino diversamente in dipendenza dei tipi di nodi.

2 Metodi di classificazione e di accesso

Un primo approccio per risolvere il problema della decomposizione (tipico del paradigma imperativo non orientato agli oggetti, ma usato anche in ambito OO) è fornire:

- dei **metodi di classificazione** che permettano di identificare i diversi tipi di nodi (tipicamente restituendo un valore `Boolean` che è `true` se e solo se il nodo su cui si invoca metodo è di un certo tipo);
- dei **metodi di accesso** alle informazioni contenute nei vari tipi di nodi.

Ad esempio, nel caso della gerarchia di `Expr` sono necessari:

- due metodi di classificazione,

  ```
  def isNumber: Boolean
  def isSum: Boolean
  ```

  rispettivamente per identificare i nodi `Number` e `Sum`;
- tre metodi di accesso,

  ```
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
  ```

  rispettivamente per il valore di un nodo `Number` e i due operandi di un nodo `Sum`.

Tali metodi vengono dichiarati nel trait `Expr` (in modo che siano invocabili su tutte le espressioni), e poi implementati nelle sottoclassi.

```
trait Expr {
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
}

class Number(n: Int) extends Expr {
  def isNumber: Boolean = true
```

```
  def isSum: Boolean = false
  def numValue: Int = n
  def leftOp: Expr = throw new NoSuchElementException("Number.leftOp")
  def rightOp: Expr = throw new NoSuchElementException("Number.rightOp")
}

class Sum(e1: Expr, e2: Expr) extends Expr {
  def isNumber: Boolean = false
  def isSum: Boolean = true
  def numValue: Int = throw new NoSuchElementException("Sum.numValue")
  def leftOp: Expr = e1
  def rightOp: Expr = e2
}
```

Si noti che, in generale, ciascun metodo di accesso può aver senso solo per determinati tipi di nodi, e per gli eventuali altri tipi solleva un'eccezione.

Infine si definisce il metodo di valutazione `eval`; siccome i metodi di classificazione e accesso forniscono tutte le informazioni necessarie sui vari tipi di nodi, `eval` può essere direttamente implementato concretamente nel trait `Expr`:

```
trait Expr {
  // ...

  def eval: Int =
    if (this.isNumber) this.numValue
    else if (this.isSum) this.leftOp.eval + this.rightOp.eval
    else throw
      new UnsupportedOperationException("Unknown expression " + this)
}
```

Il caso `else` finale, che solleva un'eccezione, è necessario perché in generale quando si discrimina sui tipi di una gerarchia si deve mettere in conto che la gerarchia potrebbe cambiare, in particolare potrebbe essere estesa con nuove classi, quindi il codice deve prevedere la possibilità di trovare istanze di classi che non sa come gestire.

Adesso si vuole osservare quali modifiche è necessario apportare al codice per estendere la gerarchia delle espressioni aggiungendo prodotti e variabili. Per prima cosa si implementano due nuove sottoclassi di `Expr`:

```
class Prod(e1: Expr, e2: Expr) extends Expr
class Var(x: String) extends Expr
```

In seguito si aggiungono a `Expr` due nuovi metodi di classificazione, corrispondenti ai tipi di nodi appena introdotti,

```
def isProd: Boolean
def isVar: Boolean
```

e un nuovo metodo di accesso per il nome di una variabile,

```
def name: String
```

mentre per gli operandi del prodotto possono essere riutilizzati i metodi di accesso già introdotti per gli operandi della somma (`leftOp` e `rightOp`). Complessivamente, il trait `Expr` risultante è:

```
trait Expr {
  def isNumber: Boolean
  def isSum: Boolean
  def isProd: Boolean
  def isVar: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
  def name: String

  def eval: Int = /* ... */
}
```

A questo punto bisogna:

- implementare i nuovi metodi di `Expr` (`isProd`, `isVar` e `name`) nelle classi preesistenti `Number` e `Sum`;
- implementare tutti i metodi di `Expr` nelle nuove classi `Prod` e `Var`;
- aggiungere all'implementazione del metodo `eval` la gestione dei nuovi tipi di nodi.

Come si può vedere, aggiungendo due nuovi sottotipi è aumentato notevolmente il numero di definizioni e implementazioni di metodi, che sono passate da 15 (3 tipi per 5 metodi) a 40 (5 tipi per 8 metodi). In generale, *l'aggiunta di nuovi sottotipi comporta un aumento quadratico del numero di definizioni/implementazioni dei metodi*, dunque l'impiego di questa soluzione basata sui metodi di classificazione e accesso risulta particolarmente oneroso.

3 Type test e type cast

Un secondo approccio è l'uso di meccanismi di basso livello che consentono di indagare sulla natura degli oggetti in fase di esecuzione. In Scala tali meccanismi sono forniti dai seguenti metodi della classe `Any` (invocabili quindi su qualunque oggetto):

- `def isInstanceOf[T]: Boolean`, che restituisce `true` se e solo se l'oggetto su cui si invoca il metodo è di tipo `T`, cioè effettua un *type test*, analogo all'operatore `instanceof` di Java (`x.isInstanceOf[T]` in Scala corrisponde a `x instanceof T` in Java).

- `def asInstanceOf[T]: T`, che effettua un *type cast*: tratta l'oggetto su cui il metodo è invocato come un'istanza del tipo `T`, sollevando una `ClassCastException` se l'oggetto non è di tipo `T` (`x.asInstanceOf[T]` in Scala è analogo a `(T) x` in Java). Questo metodo viene tipicamente usato per restringere il tipo di un oggetto: quando un nome (o espressione) che in fase di compilazione ha un tipo T è associato in fase di esecuzione a un oggetto di tipo $T' <: T$, l'uso di `asInstanceOf[`T'`]` permette di invocare sull'oggetto i metodi specifici di T' che magari non sono disponibili per il tipo T.

L'approccio basato su questi metodi è il seguente:

- `isInstanceOf` sostituisce i metodi di classificazione;
- si definisce ciascun metodo di accesso solo nelle classi per cui ha senso (invece che nel trait/superclasse, ovvero in tutte le classi della gerarchia) e si usa `asInstanceOf` per poterlo invocare.

Considerando per semplicità la gerarchia con solo `Number` e `Sum`, il codice è:

```
trait Expr {
  def eval: Int =
    if (this.isInstanceOf[Number])
      this.asInstanceOf[Number].numValue
    else if (this.isInstanceOf[Sum]) {
      val sum = this.asInstanceOf[Sum]
      sum.leftOp.eval + sum.rightOp.eval
    }
    else throw
      new UnsupportedOperationException("Unknown expression " + this)
}

class Number(n: Int) extends Expr {
  def numValue: Int = n
}

class Sum(e1: Expr, e2: Expr) extends Expr {
  def leftOp: Expr = e1
  def rightOp: Expr = e2
}
```

Il vantaggio di tale soluzione è che, rispetto alla soluzione con metodi di classificazione e accesso, si evita l'aumento quadratico del numero di metodi quando si aggiungono sottotipi. Invece, lo svantaggio è che si tratta di un approccio di basso livello, basato su meccanismi che se non usati attentamente possono provocare errori in fase di esecuzione (`ClassCastException`), poiché la verifica dei tipi degli oggetti e il cast vengono effettuati a runtime; nell'esempio appena mostrato ciò non accade perché ogni type cast è preceduto

da un type test che ne verifica la correttezza, ma in generale non è sempre possibile garantire l'assenza di errori a runtime.

4 Decomposizione object oriented

Un terzo approccio è sfruttare il meccanismo di overriding tipico della programmazione object oriented, che è sostanzialmente un meccanismo di classificazione internalizzato dal linguaggio, progettato in modo che la sua correttezza sia garantita dal compilatore. Secondo quest'approccio, `eval` viene dichiarato nel trait `Expr` come metodo astratto, che è poi implementato concretamente in modo specifico per ciascun tipo di nodo (sottoclasse):

```
trait Expr {
  def eval: Int
}

class Number(n: Int) extends Expr {
  def eval: Int = n
}

class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
}
```

Tale soluzione è efficace ed elegante, e facilita l'aggiunta di nuovi sottotipi (quando si definisce una nuova sottoclasse è sufficiente implementare al suo interno ciascuno dei metodi astratti, senza bisogno di modificare le altre classi della gerarchia). Tuttavia, essa ha anche dei notevoli aspetti negativi:

- Per aggiungere un nuovo metodo che dipende dalla natura dei nodi (e non può essere implementato concretamente nel trait/superclasse usando i metodi già definiti) bisogna modificare tutte le classi della gerarchia. Se da un lato è vero che almeno la correttezza di tali modifiche viene verificata dal compilatore (che controlla che i metodi astratti siano implementati in tutte le sottoclassi concrete), dall'altro il costo delle modifiche in termini di linee di codice da aggiungere può essere molto elevato, e nel caso delle classi di libreria si ha il problema che l'aggiunta di un metodo astratto obbliga tutti gli utenti della libreria a modificare le eventuali sottoclassi che hanno definito.
- Non è un approccio adatto a gestire i metodi non locali. Un metodo si dice **non locale** quando per operare esso deve conoscere la struttura dell'intero albero, ovvero quando il suo comportamento non può essere descritto a livello del singolo nodo. Un esempio è un metodo di semplificazione delle espressioni, che ad esempio

riscriva `a * b + a * c` come `a * (b + c)`. Per implementare un metodo non locale serve un meccanismo che permetta la classificazione di e l'accesso a più nodi dell'albero all'interno di una singola invocazione del metodo, mentre l'overriding tratta solo un nodo alla volta (quello su cui il metodo è invocato).

5 Decomposizione funzionale

La soluzione al problema della decomposizione tipicamente fornita dai linguaggi funzionali è una basata sui meccanismi di **pattern matching**, che verranno presentati in seguito.