Azzolini Riccardo 2021-05-13

Pattern matching

1 Decomposizione funzionale

Lo scopo dei metodi di classificazione e di accesso, e in un modo o nell'altro anche delle altre soluzioni al problema della decomposizione viste finora, è sostanzialmente quello di *invertire il processo di costruzione* degli oggetti:

- i metodi di classificazione permettono di individuare la sottoclasse che è stata utilizzata per costruire un oggetto;
- i metodi di accesso consentono di determinare gli argomenti che sono stati usati nel costruttore (in funzione dei quali, in un contesto puramente funzionale, sono definiti i valori di tutti i campi dell'oggetto).

In altre parole, tutti gli elementi necessari a classificare un oggetto e ad accedere alle informazioni che esso contiene sono fornite dalla costruzione dell'oggetto, ovvero dal valore che descrive l'oggetto nel modello di sostituzione (dato che la rappresentazione di un oggetto in tale modello corrisponde appunto all'invocazione del costruttore). Il **pattern matching** è il meccanismo tramite il quale i linguaggi funzionali consentono di utilizzare le informazioni specificate in tale valore.[1]

2 Classi case

L'implementazione del pattern matching in Scala è fortemente influenzata dal fatto che Scala sia un linguaggio orientato agli oggetti: il pattern matching agisce principalmente su un tipo particolare di classi, le classi case.

La definizione di una **classe case** (**case class**) è analoga alla definizione di una classe normale, ma è preceduta dalla parola riservata `case`. Ad esempio, le definizioni

```
trait Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
case class Prod(e1: Expr, e2: Expr) extends Expr
case class Var(name: String) extends Expr
```

[1] Nei linguaggi funzionali non orientati agli oggetti il pattern matching opera su valori che non sono oggetti, ma ha comunque lo scopo di invertire il processo di costruzione di tali valori.

realizzano la gerarchia delle espressioni aritmetiche, che è costituita dal trait `Expr` e da quattro classi case concrete che estendono il trait. Siccome il pattern matching opera sulle istanze delle classi, tipicamente si definiscono con `case` solo le classi concrete (non astratte) di una gerarchia (e `case` non è proprio ammesso nella definizione dei trait).

Come effetto della definizione di una classe case il compilatore fornisce automaticamente diversi supporti sintattici:

- i parametri di classe sono implicitamente definiti `val`, cioè memorizzati come campi;
- viene definito un companion object con un metodo factory che permette la costruzione di istanze della classe con una sintassi più agevole;
- viene generata un'implementazione "naturale" dei metodi `equals`, `hashCode` e `toString`;
- viene implementato un metodo `copy`, utile per creare copie modificate delle istanze della classe;
- si ha il supporto per il pattern matching.

Oltre che per le classi, la parola riservata `case` può essere utilizzata anche nella definizione degli `object` (in tal caso, tra i supporti sintattici appena elencati il compilatore fornisce solo quelli che hanno senso per gli oggetti singleton — ad esempio non vengono forniti né un metodo factory né un metodo `copy`, perché non si possono costruire o copiare istanze di un singleton object).

2.1 Companion object con un metodo factory

Per ogni case class il compilatore definisce implicitamente un commpanion object con un metodo factory `apply` che invoca il costruttore primario della classe e ha la stessa segnatura di tale costruttore. Tale metodo può poi essere usato (come già visto nell'implementazione di `List` presentata in precedenza) per costruire istanze della classe senza bisogno di scrivere l'operatore `new`.

Ad esempio, nel caso della gerarchia di `Expr` riportata sopra vengono implicitamente definiti i companion object

```
object Number { def apply(n: Int) = new Number(n) }
object Sum { def apply(e1: Expr, e2: Expr) = new Sum(e1, e2) }
object Prod { def apply(e1: Expr, e2: Expr) = new Prod(e1, e2) }
object Var { def apply(name: String) = new Var(name) }
```

che permettono di creare istanze delle varie classi case scrivendo, ad esempio:

- `Number(1)` invece di `new Number(1)`;

- `Sum(Number(1), Number(5))` invece di
  `new Sum(new Number(1), new Number(5));`
- ecc.

2.2 Metodi `equals`, `hashCode` e `toString`

Un oggetto istanza di una classe case costituisce un albero che ha come radice l'oggetto stesso e come sottoalberi gli argomenti del costruttore. Le implementazioni "naturali" dei metodi `equals`, `hashCode` e `toString` che il compilatore genera per una case class lavorano ricorsivamente sulla struttura di tale albero:

- `equals` confronta l'albero con un altro oggetto, restituendo `true` se e solo se l'altro oggetto è un albero con la stessa struttura e con valori uguali nelle foglie;
- `hashCode` calcola l'hash dell'albero in base agli hash dei sottoalberi;
- `toString` restituisce la stringa che descrive l'albero usando la notazione dei metodi factory.

Alcuni esempi d'uso di questi metodi sulla gerarchia di `Expr` sono i seguenti (qui i commenti indicano l'output dell'interprete, che utilizza `toString` per rappresentare i valori degli oggetti assegnati ai vari nomi):

```
val n1 = Number(1)          // n1: Number = Number(1)
val n2 = Number(5)          // n2: Number = Number(5)
val s1 = Sum(n1, n2)        // s1: Sum = Sum(Number(1),Number(5))
val s2 = Sum(n1, n2)        // s2: Sum = Sum(Number(1),Number(5))
s1 == s2                    // res0: Boolean = true
s1.hashCode == s2.hashCode // res1: Boolean = true
```

Si noti in particolare che `s1` e `s2` sono uguali secondo il metodo `equals` (invocato tramite l'operatore `==`) e hanno lo stesso hash nonostante siano due oggetti distinti (creati separatamente), perché la loro struttura e i valori nelle loro foglie (gli interi `1` e `5`) sono uguali.

2.3 Metodo `copy`

Il metodo `copy` che il compilatore aggiunge a ogni classe case permette di costruire copie eventualmente modificate dell'oggetto su cui è invocato.

Un meccanismo del linguaggio Scala che risulta particolarmente utile per questo metodo è la possibilità di istanziare i parametri di un metodo per nome anziché in modo posizionale: nel caso di `copy`, ciò permette di specificare i nuovi valori dei soli campi della classe che si vuole siano modificati nella copia; tutti gli altri campi mantengono i valori originali.

Ad esempio, date le definizioni

```
val n1 = Number(1)          // n1: Number = Number(1)
val n2 = Number(5)          // n2: Number = Number(5)
val s2 = Sum(n1, n2)        // s2: Sum = Sum(Number(1),Number(5))
```

scritte prima, l'invocazione

```
val s3 = s2.copy(e1 = n2) // val s3: Sum = Sum(Number(5),Number(5))
```

produce una copia dell'oggetto associato a `s2` nella quale il valore originale del campo `e1` (che era `Number(1)`) è sostituito dal valore di `n2` (cioè da `Number(5)`).

3 Pattern matching

In Scala il pattern matching è espresso da un'espressione introdotta dalla parola riservata `match`. Ad esempio:

```
def eval(e: Expr): Int = e match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
  // ...
}
```

La sintassi generale di un'espressione di pattern matching è

selector `match { ` *alternatives* `}`

dove:

- *selector* è il *selettore*, un'espressione sul cui valore verrà eseguito il pattern matching (tale valore dev'essere un'istanza di una case class, oppure di un altro tipo che supporta il pattern matching, come si vedrà più avanti);
- *alternatives* è una sequenza di alternative, ciascuna delle quali ha la forma

  `case` *pat* `=>` *expr*

  ovvero è introdotta dalla parola riservata `case` e associa a un **pattern** *pat* un'espressione *expr*.[2]

La semantica di tale espressione (il modo in cui viene valutata) verrà definita formalmente a breve, ma per ora può essere descritta intuitivamente come segue:

1. Viene valutata l'espressione *selector*.

[2] Se in particolare *expr* è un blocco, le parentesi graffe possono essere omesse, cioè è ammesso scrivere direttamente una sequenza di definizioni ed espressioni (una per riga oppure separate da `;`), e come al solito il valore del blocco è determinato dal valore dell'ultima espressione.

2. Il suo valore viene confrontato con ognuno dei pattern, nell'ordine in cui compaiono.
3. Il primo pattern che *corrisponde* (*matches*) al valore viene selezionato e il risultato complessivo dell'espressione di pattern matching è il risultato della valutazione dell'espressione corrispondente a tale pattern; se invece nessuno dei pattern corrisponde al valore specificato viene sollevata un'eccezione `MatchError` (quindi per evitare errori in fase di esecuzione bisogna fare attenzione a specificare pattern corrispondenti a tutti i possibili valori del selettore).

Si noti in particolare che l'ordine dei pattern è rilevante: se il valore del selettore corrisponde a più pattern, viene selezionato solo quello che compare per primo.

3.1 Forma dei pattern

I pattern sono costruiti a partire dalle seguenti componenti:

- **Costruttori** delle classi case (scritti nella sintassi dei metodi factory, senza `new`).
- **Variabili**, i cui nomi devono iniziare con una lettera *minuscola*. Ogni variabile può occorrere al più una volta in un pattern.
- **Wildcard**, indicati dal carattere `_` (underscore).
- **Costanti** e **letterali**. Una costante deve essere un nome che:
  - è definito e visibile nel punto del codice in cui scrive il pattern;
  - inizia con una lettera *maiuscola*, con l'eccezione delle costanti predefinite `true`, `false` e `null`, che però sono più propriamente dei letterali.

Alcuni esempi di pattern corrispondenti a vari valori di tipo `Expr` sono allora:

- `Number(1)`, che corrisponde solo allo specifico valore `Number(1)`, cioè non ammette variabilità;
- `Number(n)`, che corrisponde a qualunque istanza del costruttore `Number`, e all'atto della valutazione associa il valore del parametro attuale del costruttore alla variabile `n`, permettendo l'uso di tale valore nell'espressione corrispondente al pattern;
- `Number(_)`, che corrisponde anch'esso a qualunque istanza del costruttore `Number`, ma non associa ad alcun nome il valore del parametro del costruttore;
- `Number(N)`, che corrisponde a un valore istanza di `Number` avente il valore del parametro del costruttore uguale al valore della costante `N`, la quale deve essere stata definita in precedenza con `val N = ...`;

- `Sum(Number(N), _)`, che corrisponde a qualunque valore del tipo `Sum(Number(N), e)`, dove `N` è una costante (come nell'esempio precedente) ed `e` è un generico valore (che sarà di tipo `Expr`, poiché questo è il tipo del secondo parametro del costruttore `Sum`).

3.2 Valutazione di un'espressione `match`

Si consideri una generica espressione `match`:

$$e \texttt{ match \{ case } p_1 \texttt{ => } e_1 \texttt{ ... case } p_n \texttt{ => } e_n \texttt{ \}}$$

Per valutarla, si valuta innanzitutto il selettore e, dopodiché si confronta il valore di e con i pattern $p_1, \ldots, p_n$, nell'ordine in cui sono scritti:

- Se p_i è il primo pattern che corrisponde al valore del selettore e, l'intera espressione match è riscritta come l'espressione e_i nella quale i riferimenti alle variabili che compaiono nel pattern sono sostituiti con le parti corrispondenti del valore del selettore.
- Se invece nessuno dei pattern corrisponde al valore di e viene sollevata un'eccezione di tipo `MatchError`.

Le regole per determinare se un pattern **corrisponde** a un selettore sono le seguenti:

- Un *costruttore* $C(p_1, \ldots, p_n)$ corrisponde a tutti i valori del tipo C o di un suo sottotipo che sono stati costruiti passando al costruttore primario di C degli argomenti che a loro volta corrispondono ai pattern $p_1, \ldots, p_n$. Si osservi che questo meccanismo funziona anche per i sottotipi di C grazie al fatto che le sottoclassi debbano sempre specificare gli argomenti per il costruttore primario della superclasse (anche se magari indirettamente, tramite uno o più costruttori secondari).
- Una *variabile* x corrisponde a qualunque valore, e la valutazione *lega* il nome della variabile a tale valore nel corpo dell'espressione corrispondente al pattern.
- Un *wildcard* corrisponde a qualunque valore, come una variabile, ma a differenza di quest'ultima non lega alcun nome a tale valore.
- Una *costante* o un *letterale* K corrisponde ai valori che sono uguali a K secondo il metodo `equals`.

Si osservi che, siccome una variabile o un wildcard corrispondono a qualunque valore, si può mettere uno di questi come ultimo pattern in un'espressione `match` per fornire un caso di default (analogo al `default` dello `switch` in Java), il quale viene selezionato se e solo se non ci sono altri pattern più specifici corrispondenti al valore del selettore, evitando così che possa essere sollevato un `MatchError`.

3.2.1 Esempio

Data la definizione della funzione `eval` già mostrata prima,

```
def eval(e: Expr): Int = e match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
  // ...
}
```

si vuole valutare l'espressione

```
eval(Sum(Number(1), Number(2)))
```

Siccome il parametro di `eval` è passato con la strategia call-by-value, il primo passo di valutazione dovrebbe essere la valutazione del parametro attuale. Tuttavia, se per comodità si decide di semplificare la rappresentazione nel modello di sostituzione delle istanze delle classi case, usando la sintassi dei metodi factory senza l'operatore `new`, allora il parametro attuale `Sum(Number(1), Number(2))` è già completamente valutato, rappresenta il valore di un oggetto. Di conseguenza, si possono sostituire immediatamente l'applicazione della funzione con il suo corpo e i riferimenti al parametro formalo in tale corpo con il parametro attuale (qui per brevità è mostrato direttamente il risultato di entrambe queste sostituzioni):

```
↠ Sum(Number(1), Number(2)) match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
    // ...
  }
```

Adesso si applicano le regole di riscrittura del pattern matching. Dato che il selettore è già valutato, si inizia subito a confrontare il suo valore con i pattern nell'ordine in cui questi sono scritti:

1. Il primo pattern, `Number(n)`, non corrisponde al selettore perché la classe `Sum` di cui quest'ultimo è istanza non è un sottotipo della classe `Number` richiesta dal pattern.
2. Invece il secondo pattern, `Sum(e1, e2)`, corrisponde al selettore perché:
   - il costruttore del valore del selettore coincide con il costruttore `Sum` specificato nel pattern;
   - i pattern specificati per i parametri del costruttore sono variabili, che corrispondono a qualunque valore dei parametri attuali del costruttore.

A questo punto, avendo trovato il primo pattern che corrisponde al selettore, l'intera espressione `match` viene riscritta come l'espressione corrispondente a tale pattern, nella quale i riferimenti alle variabili `e1` ed `e2` sono sostituiti con i valori dei parametri attuali del costruttore a cui tali variabili corrispondono:

```
→ [Number(1)/e1, Number(2)/e2](eval(e1) + eval(e2))
→ eval(Number(1)) + eval(Number(2))
```

Ora bisogna valutare l'applicazione di `eval` che costituisce il primo operando dell'addizione. La valutazione procede sostanzialmente come prima,

```
↠ Number(1) match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
    // ...
  } + eval(Number(2))
```

ma questa volta il primo pattern che corrisponde al selettore è `Number(n)`, che lega alla variabile `n` l'argomento `1` del costruttore:

```
→ ([1/n]n) + eval(Number(2))
→ 1 + eval(Number(2))
```

Poi il processo si ripete ancora una volta per la restante applicazione di `eval`, e infine viene calcolato il risultato dell'addizione:

```
↠ 1 + (Number(2) match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
    // ...
  })
→ 1 + ([2/n]n)
→ 1 + 2
→ 3
```

4 `eval` come metodo di `Expr`

Nell'esempio precedente `eval` è stata definita come funzione che prende l'espressione da valutare come argomento, ma la si potrebbe ugualmente definire come metodo del trait `Expr`:

```
def eval: Int = this match {
  case Number(n) => n
  case Sum(e1, e2) => e1.eval + e2.eval
  // ...
}
```

Le uniche modifiche che è stato necessario apportare al codice di `eval` sono:

- fare il pattern matching su `this` invece che su un argomento;
- usare la sintassi dell'invocazione dei metodi per le chiamate ricorsive.

5 Esempio di metodo non locale

Si vuole definire una funzione `show(e: Expr): String` che restituisca la stringa che rappresenta nell'usuale notazione infissa l'espressione fornita come argomento, descrivendo in modo corretto le precedenze degli operatori con il minori numero possibile di parentesi; ad esempio:

- `show(Sum(Prod(Number(2), Var("x")), Number(5)))` deve restituire `"2 * x + 5"`;
- `show(Prod(Sum(Number(2), Var("x")), Number(5)))` deve restituire `"(2 + x) * 5"`.

Da questi due casi si può osservare che `show` è un metodo/funzione non locale (come nel caso di `eval`, definire una funzione con un argomento o un metodo di `Expr` è equivalente), il cui comportamento non può essere descritto a livello del singolo nodo dell'albero che rappresenta un'espressione. In particolare, per decidere se racchiudere tra le parentesi un operando di `Prod` bisogna conoscere il tipo di nodo corrispondente a tale operando: le parentesi servono se l'operando è una somma, ma non se è un numero, una variabile o anch'esso un prodotto. A differenza dell'overriding, il pattern matching permette di implementare facilmente tale comportamento:

```
def show(e: Expr): String = e match {
  case Number(n) => n.toString
  case Var(name) => name
  case Sum(e1, e2) => show(e1) + " + " + show(e2)
  case Prod(e1, e2) =>
    def parenthesizeIfSum(e: Expr) = e match {
      case Sum(_, _) => "(" + show(e) + ")"
      case _ => show(e)
    }
    parenthesizeIfSum(e1) + " * " + parenthesizeIfSum(e2)
}
```

6 Confronto tra decomposizione OO e funzionale

Tra la decomposizione OO e la decomposizione funzionale non è possibile individuare una soluzione che sia in generale migliore dell'altra: ciascuna ha vantaggi e svantaggi che la rendono più adatta a determinate situazioni. Indicativamente:

- La decomposizione OO è preferibile quando capita più frequentemente di aggiungere nuovi sottitipi che nuovi metodi: quando si aggiunge un nuovo tipo si implementano i metodi necessari localmente nel solo tipo aggiunto, mentre per aggiungere un nuovo metodo è necessario modificare tutta la gerarchia.
- La decomposizione funzionale è preferibile quando la gerarchia è relativamente stabile ma capita frequentemente di aggiungere nuovi metodi. Infatti, per aggiungere un metodo è sufficiente scriverlo in una singola classe (tipicamente la superclasse della gerarchia), mentre se si aggiunge un nuovo sottotipo bisogna modificare tutti i metodi che fanno pattern matching sulla gerarchia. Inoltre, la decomposizione funzionale è preferibile per i metodi non locali, che la decomposizione OO non è in grado di gestire.

Un aspetto interessante dei linguaggi funzionali a oggetti come Scala è che essi mettono a disposizione entrambe queste soluzioni, lasciando al programmatore la possibilità di scegliere volta per volta quale impiegare.

Invece, le altre due soluzioni al problema di decomposizione (metodi di classificazione e accesso e meccanismi di basso livello) non vengono tipicamente usate in linguaggi "ricchi" come Scala che forniscono sia la decomposizione OO che quella funzionale, poiché non hanno particolari vantaggi rispetto all'insieme delle soluzioni OO e funzionale.

7 Classi/trait `sealed`

La definizione di una classe o di un trait può essere preceduta dalla parola riservata `sealed`, la quale indica che tutti i sottitipi (diretti) di tale classe/trait sono definiti nello stesso file, cioè di fatto impedisce a un utente della gerarchia di aggiungere nuovi sottotipi. Ciò è utile per le gerarchie di classi case su cui si opera tramite pattern matching: se un utente aggiungesse una nuova sottoclasse introdurrebbe un caso non gestito dal pattern matching nei metodi esistenti, che quindi non funzionerebbero più correttamente. Inoltre, quando si fa il pattern matching su un'espressione di un tipo `sealed` il compilatore è in grado di emettere un'avvertenza se non vengono gestiti tutti i casi possibili, aiutando così a evitare errori (`MatchError`) in fase di esecuzione.

Ad esempio, nella definizione della gerarchia delle espressioni è opportuno dichiarare `sealed` il trait `Expr`:

```
sealed trait Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
case class Prod(e1: Expr, e2: Expr) extends Expr
case class Var(name: String) extends Expr
```

Così, l'utente non può aggiungere nuovi tipi di nodi non gestiti dalle funzioni/metodi esistenti, e se si fa il pattern matching su `Expr` senza trattare tutti i casi, come ad esempio

```
def eval(e: Expr): Int = e match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
  // Mancano i casi Prod e Var
}
```

il compilatore emette un'avvertenza:

```
match may not be exhaustive.
It would fail on the following inputs: Prod(_, _), Var(_)
```