Azzolini Riccardo 2021-05-03

Linguaggi a oggetti puri

1 Linguaggi a oggetti puri

Un **linguaggio a oggetti puro** è un linguaggio in cui *ogni valore è un oggetto*. In particolare, se il linguaggio è basato sulle classi (non tutti i linguaggi a oggetti lo sono), allora *il tipo di ogni valore è una classe*.

Concettualmente, al livello della macchina astratta, Scala è un linguaggio a oggetti puro, i cui tipi potrebbero tutti essere definiti sotto forma di classi scritte nel linguaggio stesso. Ci sono però alcuni tipi la cui implementazione come classi non è immediata, ed è quindi opportuno discutere: i *tipi base* e le *funzioni*.

Il fatto che Scala sia un linguaggio a oggetti puro è importante perché significa che il modello di sostituzione è applicabile a tutto il linguaggio (escluse le parti imperative), e questa è una condizione necessaria per poter sfruttare i meccanismi di ragionamento formali basati su tale modello, che come detto in precedenza sono uno dei principali vantaggi della programmazione funzionale pura.

2 Tipi base

Concettualmente i tipi base di Scala (`Int`, `Boolean`, ecc.) sono trattati come oggetti, istanze di classi definite nel package `scala`. Tuttavia, per motivi di efficienza, nel modello di esecuzione essi sono in realtà rappresentati utilizzando i tipi primitivi della JVM, ma ciò è trasparente al livello a cui ragiona il programmatore, che non vede differenze tra i tipi base e gli altri.

Per dimostrare che Scala è un linguaggio a oggetti puro nonostante l'uso dei tipi primitivi nell'implementazione, bisogna definire delle classi che implementino i tipi base senza fare uso dei tipi primitivi, e poi verificare che i metodi di tali classi abbiano lo stesso comportamento delle operazioni sui tipi primitivi fornite dalla JVM. Fare ciò per tutti i tipi base è piuttosto complicato e laborioso, ma l'idea generale può essere illustrata mostrando una possibile implementazione OO pura del tipo `Boolean` (scelto perché è piuttosto semplice).

2.1 Tipo `Boolean` puro

Il tipo `Boolean` di Scala corrisponde al tipo primitivo `boolean` della JVM. Una sua implementazione OO pura potrebbe essere fornita dalla seguente gerarchia di classi:

```
package idealized.scala

trait Boolean {
  def ifThenElse[T](t: => T, e: => T): T

  def &&(x: => Boolean): Boolean = ifThenElse(x, false)
  def ||(x: => Boolean): Boolean = ifThenElse(true, x)
  def unary_! : Boolean = ifThenElse(false, true)

  def ==(x: => Boolean): Boolean = ifThenElse(x, !x)
  def !=(x: => Boolean): Boolean = ifThenElse(!x, x)
}

object true extends Boolean {
  override def ifThenElse[T](t: => T, e: => T): T = t
}

object false extends Boolean {
  override def ifThenElse[T](t: => T, e: => T): T = e
}
```

- Il nome del package `idealized.scala` vuole mettere in evidenza il fatto che questa non è l'implementazione reale, ma appunto una "ideale" finalizzata al ragionamento formale.
- Il trait `Boolean` definisce le operazioni disponibili sul tipo. Qui per semplicità sono definite solo alcune delle principali operazioni, ma si potrebbero definire in modo simile anche tutte le altre.
- I singleton object `true` e `false`,[1] che estendono `Boolean`, rappresentano gli unici due valori del tipo.

In un linguaggio a oggetti puro, che tratta tutti i valori come oggetti, tutti gli operatori devono essere riscrivibili come invocazioni di metodi, compreso l'operatore condizionale `if-else`. A tale scopo, il trait `Boolean` definisce un metodo `ifThenElse`: l'espressione

`if (`*cond*`)` *thenExpr* `else` *elseExpr*

[1] I nomi `true` e `false` sono parole riservate in Scala (perché sono i letterali del tipo `Boolean` predefinito, qui usati come nomi proprio per mettere in evidenza l'equivalenza al tipo predefinito), quindi se si volesse compilare concretamente il codice sarebbe necessario rinominare questi oggetti, ad esempio mettendo le iniziali maiuscole (`True`, `False`).

è equivalente a

cond.ifThenElse(*thenExpr*, *elseExpr*)

Si noti che i due parametri di `ifThenElse`, corrispondenti alle espressioni nei rami "then" e "else" dell'`if-else`, sono passati con la strategia call-by-name, in modo che valutata solo l'espressione nel ramo corrispondente al valore della condizione, esattamente come fa l'operatore `if-else` predefinito. Inoltre, il tipo di entrambi questi parametri è il tipo parametro `T`, che è anche il tipo restituito dal metodo; se esso non viene specificato esplicitamente, il compilatore deduce il più piccolo supertipo comune ai tipi delle espressioni dei due rami, come appunto avviene per l'operatore `if-else`.

`ifThenElse` è l'unico metodo astratto di `Boolean`, e in quanto tale viene implementato concretamente negli object `true` e `false`: quando viene invocato sull'oggetto `true` restituisce il valore dell'espressione nel ramo "then" (il parametro `t`), mentre sull'oggetto `false` restituisce il valore dell'espressione nel ramo "else" (il parametro `e`). Tutti gli altri metodi di `Boolean` possono essere implementati concretamente direttamente nel trait, utilizzando solo `ifThenElse` per esprimere le regole di valutazione dei vari operatori:

- L'espressione *a* `&&` *b* deve essere riscritta come *b* se *a* vale `true` e come `false` se *a* è `false`, quindi equivale a:

  a `&&` *b* ≡ `if (`*a*`)` *b* `else false` ≡ *a*`.ifThenElse(`*b*`, false)`

- *a* `||` *b* viene riscritta come `true` se *a* vale `true` e come *b* se *a* è `false`:

  a `||` *b* ≡ `if (`*a*`) true else` *b* ≡ *a*`.ifThenElse(true,` *b*`)`

- `!`*a* deve essere riscritta come `true` se *a* è `false`, e viceversa:

  `!`*a* ≡ `if (`*a*`) false else true` ≡ *a*`.ifThenElse(false, true)`

- Per ricavare l'implementazione dell'operatore `==` si ragiona sui casi in cui due valori booleani sono uguali:
  - se *a* è `true`, allora l'espressione *a* `==` *b* è vera se e solo se anche *b* è `true`, ovvero assume lo stesso valore di verità di *b*;
  - se invece *a* vale `false`, allora l'espressione è vera se e solo se anche *b* è `false`, cioè assume il valore di verità opposto di *b*.

  Nell'esprimere queste regole è comodo riutilizzare, oltre a `ifThenElse`, l'operatore di negazione `!` precedentemente definito:

  a `==` *b* ≡ `if (`*a*`)` *b* `else !`*b* ≡ *a*`.ifThenElse(`*b*`, !`*b*`)`

- Due valori booleani sono diversi nei casi opposti a quelli in cui sono uguali, quindi si può ottenere l'implementazione dell'operatore `!=` a partire da quella di `==` scambiando i due rami di `ifThenElse`:

  $$a \texttt{ != } b \equiv \texttt{if (}a\texttt{) !}b \texttt{ else } b \equiv a\texttt{.ifThenElse(!}b\texttt{, } b\texttt{)}$$

  Una soluzione alternativa sarebbe semplicemente applicare la negazione al risultato di `==`:

  $$a \texttt{ != } b \equiv \texttt{!(}a \texttt{ == } b\texttt{)}$$

```
def !=(x: => Boolean): Boolean = !(this == x)
```

2.2 Tipi numerici puri

Come si è appena visto, fornire un'implementazione OO pura del tipo `Boolean` è abbastanza semplice, dato che esso ha solo due valori e tutte le sue operazioni possono essere ricondotte all'operatore condizionale `ifThenElse`.

Per i tipi numerici, invece, il gioco si complica. La tipica soluzione è procedere secondo lo stesso approccio assiomatico che si impiega in matematica: prima si definiscono i numeri naturali, poi a partire da essi si definiscono i numeri interi, che vengono a loro volta usati per definire i numeri razionali, e infine tramite i razionali si definiscono i numeri reali.

Ad esempio, l'assiomatizzazione di Peano dà una definizione ricorsiva dei numeri naturali: un numero naturale è zero, oppure il successore di un numero naturale. Ciò può essere espresso in Scala mediante la seguente gerarchia di classi:

```
trait Nat
object Zero extends Nat
class Succ(n: Nat) extends Nat
```

Intuitivamente, tale definizione permette di rappresentare tutti i numeri naturali:

$$\begin{aligned} 0 &\equiv \texttt{Zero} \\ 1 &\equiv \texttt{new Succ(Zero)} \\ 2 &\equiv \texttt{new Succ(new Succ(Zero))} \\ &\vdots \end{aligned}$$

Le operazioni che vengono definite sul tipo `Nat` sono poi:

```
trait Nat {
  def isZero: Boolean
  def predecessor: Nat
  def successor: Nat
  def +(that: Nat): Nat
  def -(that: Nat): Nat
}
```

- un predicato `isZero`, che come suggerisce il nome permette di determinare se un numero naturale è `Zero` oppure un'istanza di `Succ` (un numero positivo);
- `predecessor`, che restituisce il numero naturale precedente a quello su cui il metodo è invocato (e solleva un'eccezione se invocato su `Zero`, che non ha un predecessore);
- `successor`, che restituisce il numero naturale successivo a quello su cui è invocato;
- tutti i vari operatori aritmetici, di confronto, ecc. (qui per brevità sono riportati solo `+` e `-`).

3 Funzioni

In Scala anche le funzioni sono valori, quindi per dimostrare che Scala è un linguaggio orientato agli oggetti puro è necessario poter rappresentare le funzioni come oggetti.

In questo caso, a differenza di quanto fatto per i tipi base, non è necessario fornire un'implementazione OO "ideale" che sia equivalente a quella reale, perché in Scala i valori delle le funzioni sono effettivamente rappresentate da oggetti. Infatti, il tipo funzionale *`A => B`* è un'abbreviazione (fornita dal compilatore) per la classe (trait) `scala.Function1[A, B]`, che è definita sostanzialmente in questo modo:

```
package scala

trait Function1[A, B] {
  def apply(x: A): B
}
```

L'indice `1` nel nome `Function1` indica che questo trait rappresenta le funzioni con 1 argomento; per le funzioni con più argomenti il package `scala` definisce dei trait simili (attualmente fino a `Function22`):

```
trait Function2[A, B, C] {
  def apply(x: A, y: B): C
}
trait Function3[A, B, C, D] {
  def apply(x: A, y: B, z: C): D
}
// ... fino a Function22
```

Di conseguenza, in generale, una funzione è un oggetto che fornisce un metodo `apply` avente dominio e codominio corrispondenti a quelli della funzione.

Una funzione anonima con *N* argomenti viene concettualmente espansa dal compilatore nel corrispondente oggetto, che implementa il trait `FunctionN` e definisce il metodo astratto `apply` con il corpo della funzione anonima. Ad esempio, la funzione anonima

```
(x: Int) => x * x
```

viene espansa in

```
{
  class AnonFun extends Function1[Int, Int] {
    def apply(x: Int) = x * x
  }
  new AnonFun
}
```

oppure, scritto in modo più compatto utilizzando la sintassi per le *classi anonime* (che è analoga a quella di Java),

```
new Function1[Int, Int] {
  def apply(x: Int) = x * x
}
```

In realtà, per motivi di efficienza e semplicità il compilatore non effettua una riscrittura del codice sorgente, ma il codice compilato risultante è equivalente a quello per la forma riscritta (salvo eventuali ottimizzazioni).

Una volta mostrata la rappresentazione delle funzioni come oggetti, bisogna ricondurre ai meccanismi OO anche l'invocazione delle funzioni. Ciò avviene tramite il seguente meccanismo del linguaggio Scala: scrivere dopo il nome *obj* di un oggetto una coppia di parentesi tonde, eventualmente contenenti una lista di argomenti $x_1, \ldots, x_n$, equivale a invocare il metodo apply su tale oggetto con tali argomenti:

$$obj(x_1, \ldots, x_n) \equiv obj.\texttt{apply}(x_1, \ldots, x_n)$$

Questo meccanismo si applica anche alle funzioni, in quanto oggetti, quindi l'operazione di invocazione su un valore di tipo funzione viene ricondotto al meccanismo puramente OO di invocazione di un metodo (`apply`) su un oggetto. Ad esempio, il codice

```
val f = (x: Int) => x * x
f(7)
```

viene espanso in

```
val f = new Function1[Int, Int] {
  def apply(x: Int) = x * x
}
f.apply(7)
```

È importante sottolineare che la rappresentazione come oggetti e l'invocazione tramite `apply` non valgono invece per i metodi, altrimenti si genererebbe un loop infinito di espansioni, perché lo stesso metodo `apply` verrebbe espanso nella creazione di un nuovo oggetto con un metodo `apply`, che a sua volta dovrebbe essere espanso, e così via. Tuttavia, ciò non ostacola la dimostrazione del fatto che Scala è un linguaggio OO puro:

al contrario dei valori di tipo funzione, i metodi forniti da un oggetto possono essere rappresentati e invocati direttamente, senza riscritture, dato che sono uno degli elementi fondamentali dei linguaggi orientati agli oggetti.

4 `apply` e companion object

Il meccanismo di `apply` viene spesso sfruttato per definire dei costruttori "naturali" per le classi, che vengono invocati senza l'operatore `new`.

Ad esempio, data la gerarchia di classi che rappresenta le liste,

```
trait List[T]
class Cons[T](val head: T, val tail: List[T]) extends List[T]
class Nil[T] extends List[T]
```

Scala permette di definire un singleton object avente lo stesso nome della classe (trait) `List`:

```
object List
```

Siccome ora esiste un oggetto associato al nome `List`, il compilatore riscrive `List(...)` come `List.apply(...)`, quindi implementando nell'object `List` dei metodi `apply` che costruiscono istanze dell'omonimo tipo (trait) `List`

```
object List {
  def apply[T](): List[T] = new Nil

  def apply[T](x: T): List[T] = new Cons(x, new Nil)

  def apply[T](x1: T, x2: T): List[T] =
    new Cons(x1, new Cons(x2, new Nil))
}
```

si ottengono dei costruttori per tale tipo che possono essere invocati in modo più comodo rispetto all'uso diretto di `new Cons` e `new Nil`:

```
List()
List(1)
List(true, false)
```

(i costruttori definiti in quest'esempio hanno un numero fisso di argomenti, ma si vedrà più avanti che è possibile definire una versione generalizzata che accetta un numero qualsiasi di argomenti, permettendo così di costruire comodamente liste arbitrarie).

In generale, un singleton object che ha lo stesso nome di una classe (ed è definito nello stesso file della classe) è chiamato **companion object** ("oggetto compagno") di tale classe.