Azzolini Riccardo 2021-05-20

Tuple e parametri impliciti

1 Merge sort su liste di interi

Si vuole implementare il merge sort sulle liste, considerando inizialmente per semplicità solo liste di numeri interi. L'algoritmo opera in questo modo:

- se la lista è vuota o contiene esattamente un elemento, allora è ordinata;
- altrimenti, si spezza la lista in due sotto-liste aventi (circa[1]) il medesimo numero di elementi, si ordinano le due sotto-liste e si effettua il *merge* di queste ultime in un'unica lista ordinata.

```
def msort(xs: List[Int]): List[Int] = {
  val half = xs.length / 2
  if (half == 0) xs
  else {
    def merge(xs: List[Int], ys: List[Int]): List[Int] = /* ... */

    val fst = xs take half
    val snd = xs drop half
    merge(msort(fst), msort(snd))
  }
}
```

Il merge di due liste ordinate `xs` e `ys` in un'unica lista ordinata viene eseguito come segue:

- se `xs` è vuota si restituisce l'altra lista `ys`;
- altrimenti, il comportamento dipende da come è fatta `ys`:
  - se `ys` è vuota si restituisce `xs` (questo caso è simmetrico a quello in cui `xs` è vuota);
  - altrimenti si seleziona il minore tra i primi elementi delle due liste, lo si mette come testa della lista risultante e si continua il merge ricorsivamente sul resto delle liste.

[1] Quando la lista ha lunghezza dispari, una delle due sotto-liste in cui essa viene spezzata ha un elemento in più rispetto all'altra.

```scala
def merge(xs: List[Int], ys: List[Int]): List[Int] = xs match {
  case Nil => ys
  case x :: xs1 => ys match {
    case Nil => xs
    case y :: ys1 =>
      if (x <= y) x :: merge(xs1, ys)
      else y :: merge(xs, ys1)
  }
}
```

2 Tuple e assegnamenti strutturati

Per comodità, Scala fornisce una sintassi di supporto per rappresentare tuple: una **tupla** costituita dagli elementi $e_1, \ldots, e_N$, i quali possono essere di tipi differenti, è scritta come `(`e_1`, ..., `e_N`)`.

Una tupla è gestita come un unico valore, che può essere assegnato a un nome e ha il tipo `(`T_1`, ..., `T_N`)`, dove $T_1, \ldots, T_N$ sono i tipi dei suoi elementi $e_1, \ldots, e_N$; ad esempio:

```scala
val pair = ("pippo", 5)  // pair: (String, Int) = (pippo,5)
```

(il commento indica l'output dell'interprete).

È possibile fare pattern matching sugli elementi di una tupla tramite pattern del tipo `(`p_1`, ..., `p_N`)`; ad esempio:

```scala
pair match {
  case ("pippo", n) => n
  case _ => 0
}                          // res0: Int = 5
```

In generale, oltre che in un'espressione `match` un pattern può essere usato a sinistra dell'uguale in una definizione di nome con `val`. Ciò prende il nome di **assegnamento strutturato**, ed è utile ad esempio per assegnare a dei nomi gli elementi di una tupla:

```scala
val (label, len) = pair  // label: String = pippo
                         // len: Int = 5
```

2.1 Traduzione delle tuple in oggetti

Le tuple sono zucchero sintattico che il compilatore traduce nell'uso di delle normali classi `scala.Tuple`N (definite per $N = 1, \ldots, 22$), che hanno essenzialmente la seguente struttura:

```
package scala
case class TupleN[+T1, ..., +TN](_1: T1, ..., _N: TN) {
  override def toString = "(" + _1 + "," + ... + _N + ")"
}
```

La traduzione avviene in questo modo:

- un *tipo* `(`T_1`, ..., `T_N`)` è un'abbreviazione per il tipo parametrizzato

  `scala.TupleN[`T_1`, ..., `T_N`]`

- un'*espressione* `(`e_1`, ..., `e_N`)` è equivalente all'applicazione di funzione

  `scala.TupleN(`e_1`, ..., `e_N`)`

  (che è un'invocazione del metodo `apply` del companion object di `scala.TupleN`);

- un *pattern* `(`p_1`, ..., `p_N`)` equivale al constructor pattern

  `scala.TupleN(`p_1`, ..., `p_N`)`

Come si può osservare dalla struttura delle classi `scala.TupleN` (ricordando che i parametri delle classi case sono automaticamente dichiarati come campi), i campi di una tupla sono accessibili mediante i nomi `_1`, `_2`, ..., quindi ad esempio l'assegnamento strutturato

```
val (label, len) = pair
```

equivale a

```
val label = pair._1
val len = pair._2
```

ma la prima forma (l'assegnamento strutturato) è decisamente più elegante.

3 Riscrittura di merge sort con le tuple

L'implementazione dell'algoritmo merge sort può essere riscritta in modo un po' più elegante usando le tuple.

All'interno della funzione `msort` l'uso dei metodi `take` e `drop` può essere sostituito con un altro metodo della classe `List[T]`,

```
def splitAt(n: Int): (List[T], List[T])
```

Questo metodo partiziona la lista su cui è invocato alla posizione specificata dall'argomento `n`, restituendo una coppia contenente le due liste risultanti:

- la prima lista nella coppia contiene i primi `n` elementi della lista originale;
- la seconda lista contiene i restanti elementi.

In sostanza, `xs splitAt n` equivale a `(xs take n, xs drop n)` (ma non è implementato in questo modo — in particolare, scorre una sola volta invece di due i primi `n` elementi della lista `xs`). `xs splitAt n` non solleva mai eccezioni, ma piuttosto gestisce i casi limite esattamente come fanno `take` e `drop`:

- quando `n <= 0` restituisce `(Nil, xs)`;
- quando `n >= xs.length` restituisce `(xs, Nil)`.

Usando `splitAt` il codice di `msort` diventa:

```
def msort(xs: List[Int]): List[Int] = {
  val half = xs.length / 2
  if (half == 0) xs
  else {
    def merge(xs: List[Int], ys: List[Int]): List[Int] = /* ... */

    val (fst, snd) = xs splitAt half
    merge(msort(fst), msort(snd))
  }
}
```

Invece, nella funzione `merge` si può usare il pattern matching sulle tuple per fare effettivamente il matching su entrambe le liste contemporaneamente, rendendo così più simmetrici i due casi in cui una delle liste è vuota:

```
def merge(xs: List[Int], ys: List[Int]): List[Int] = (xs, ys) match {
  case (Nil, _) => ys
  case (_, Nil) => xs
  case (x :: xs1, y :: ys1) =>
    if (x <= y) x :: merge(xs1, ys)
    else y :: merge(xs, ys1)
}
```

4 Merge sort di liste generiche

Adesso si vuole generalizzare la funzione `msort` a liste di elementi di un arbitrario tipi `T` (invece che solo `Int`). Rimpiazzare semplicemente il tipo `Int` con un tipo parametro `T` non è sufficiente, perché l'implementazione di `merge` usa l'operatore di confronto `<=`, che non è definita per un generico tipo `T`.

Per risolvere questo problema ci sono varie soluzioni. La prima, basata su uno stile tipicamente funzionale, consiste nel definire `msort` come una funzione di ordine superiore

che riceve come parametri, oltre alla lista da ordinare, anche una funzione di ordinamento `lteq: (T, T) => Boolean`, la quale rappresenta l'operatore `<=` per il tipo `T`:[2]

```
def msort[T](xs: List[T])(lteq: (T, T) => Boolean): List[T] = {
  val half = xs.length / 2
  if (half == 0) xs
  else {
    def merge(xs: List[T], ys: List[T]): List[T] = (xs, ys) match {
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if (lteq(x, y)) x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
    }

    val (fst, snd) = xs splitAt half
    merge(msort(fst)(lteq), msort(snd)(lteq))
  }
}
```

Questa versione di `msort` può essere usata, ad esempio:

- su una lista di interi:

  ```
  val nums = List(1, 3, 7, 5, 9, -1)
  msort(nums)((x, y) => x <= y)
  ```

- su una lista di stringhe:[3]

  ```
  val fruits = List("pear", "orange", "pineapple", "apple")
  msort(fruits)((x, y) => x.compareTo(y) <= 0)
  ```

Come si può osservare da questi esempi, non è necessario indicare esplicitamente i tipi dei parametri delle funzioni anonime passate a `msort`, perché il compilatore è in grado di dedurli. Infatti, il compilatore deduce il tipo `T` in base al tipo della lista `xs` passata nella prima lista di argomenti, e poi nella seconda lista di argomenti si aspetta una funzione `lteq` che abbia due parametri del tipo `T` dedotto. Se le liste parametri di `msort` fossero organizzate in modo diverso la cosa invece non funzionerebbe: bisognerebbe scrivere esplicitamente i tipi dei parametri delle funzioni anonime. Il motivo è che, in generale, per cercare di dedurre un tipo parametro di una funzione il compilatore Scala si limita a considerare gli argomenti della prima lista di parametri in cui tale tipo compare, quindi:

[2] Il nome `lteq` di questo parametro è un'abbreviazione del nome inglese dell'operatore `<=`, "Less Than or EQual to".

[3] L'operatore `<=` è definito anche sulle stringhe, quindi si potrebbe scrivere anche qui direttamente `x <= y`, ma l'uso del metodo `compareTo` è mostrato a scopo illustrativo, per mettere in evidenza il fatto che è possibile specificare arbitrarie funzioni di confronto, e non solo quelle che hanno la forma `(x, y) => x <= y`.

- Con la definizione

  ```
  def msort[T](lteq: (T, T) => Boolean)(xs: List[T])
  ```

  il compilatore proverebbe a dedurre `T` in base unicamente al tipo della funzione `lteq`, dunque la deduzione riuscirebbe solo se il tipo dei parametri di `lteq` fosse conosciuto a priori. Nel caso in cui `lteq` è una funzione anonima, il tipo dei parametri sarebbe conosciuto a priori se fosse indicato esplicitamente o deducibile dal corpo della funzione (ma negli esempi precedenti non è deducibile perché nel corpo di `lteq` si invocano dei metodi, `<=` e `compareTo`, che come tutti i metodi potrebbero essere definite su tipi diversi, mentre una funzione che non è un metodo potrebbe magari avere un'unica definizione, dalla quale il compilatore riuscirebbe potenzialmente a ricavare le informazioni sui tipi dei parametri di `lteq`).

- Con la definizione

  ```
  def msort[T](xs: List[T], lteq: (T, T) => Boolean)
  ```

  il compilatore proverebbe a dedurre `T` considerando insieme i tipi di `xs` e `lteq`, dunque anche in questo caso avrebbe bisogno di conoscere a priori il tipo dei parametri di `lteq`.

5 Trait `Ordering[T]`

La soluzione appena presentata per parametrizzare `msort` rispetto a una funzione di ordinamento è semplice ed elegante, ma non impone sulla funzione passata come argomento solo il vincolo che essa sia di tipo `(T, T) => Boolean`, ma non tutte le funzioni di questo tipo sono funzioni di ordinamento (totale), e per le funzioni che non lo sono l'algoritmo non opera correttamente. Una soluzione alternativa, più orientata agli oggetti, che aiuta il programmatore a definire un ordinamento corretto, è utilizzare il trait `scala.math.Ordering[T]` (disponibile anche senza importazioni, tramite un alias nel package `scala`), che ha un ruolo analogo all'interfaccia `Comparator<T>` di Java: un'istanza di `Ordering[T]` rappresenta una strategie di ordinamento sugli elementi di tipo `T`, definendo implicitamente una relazione d'ordine $\leq$ su tali elementi.

Per implementare `Ordering[T]` è necessario definire il metodo astratto

```
def compare(x: T, y: T): Int
```

che deve restituire:

- `0` se `x == y`;
- un valore positivo se `x > y`;
- un valore negativo se `x < y`.

Usando il metodo `compare`, `Ordering[T]` definisce poi una serie di metodi concreti, tra cui:

- `def lt(x: T, y: T): Boolean`, che restituisce `true` se e solo se `x < y` (nell'ordinamento determinato dalla definizione di `compare` dall'istanza di `Ordering[T]` su cui si invoca questo metodo);
- `def lteq(x: T, y: T): Boolean`, che restituisce `true` se e solo se `x <= y`;
- `def gt(x: T, y: T): Boolean`, che restituisce `true` se e solo se `x > y`;
- `def gteq(x: T, y: T): Boolean`, che restituisce `true` se e solo se `x >= y`.

`Ordering[T]` ha anche un companion object `Ordering`, che fornisce degli oggetti predefiniti che implementano il trait, come ad esempio:

- `Ordering.Int`, un oggetto che implementa `Ordering[Int]` (per la precisione, un singleton object che estende il trait `IntOrdering`, il quale a sua volta estende `Ordering[Int]`) e definisce l'usuale ordine sui numeri interi;
- `Ordering.String`, un oggetto che implementa `Ordering[String]` (per la precisione, un singleton object che estende il trait `StringOrdering`, il quale a sua volta estende `Ordering[String]`) e definisce l'usuale ordine sulle stringhe.

Usando `Ordering[T]` al posto di una funzione `(T, T) => Boolean`, il metodo `msort` viene riscritto nel modo seguente:

```
def msort[T](xs: List[T])(ord: Ordering[T]): List[T] = {
  val half = xs.length / 2
  if (half == 0) xs
  else {
    def merge(xs: List[T], ys: List[T]): List[T] = (xs, ys) match {
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if (ord.lteq(x, y)) x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
    }

    val (fst, snd) = xs splitAt half
    merge(msort(fst)(ord), msort(snd)(ord))
  }
}
```

Per usare questa versione di `msort` sugli interi e sulle stringhe si possono passare come secondo argomento le istanze predefinite del trait `Ordering.Int` e `Ordering.String`:

```scala
val nums = List(1, 3, 7, 5, 9, -1)
msort(nums)(Ordering.Int)
val fruits = List("pear", "orange", "pineapple", "apple")
msort(fruits)(Ordering.String)
```

6 Parametri impliciti

Passare alla funzione `merge` i valori per il parametro `ord` ogni volta che la si usa è piuttosto scomodo. Per evitare di doverlo fare si può usare il meccanismo dei **parametri impliciti**: sotto opportune condizioni, il compilatore Scala può dedurre e inserire **argomenti impliciti** nelle invocazioni di funzioni/metodi e costruttori, rimpiazzando

- un'invocazione di una funzione/metodo `f(a)` con `f(a)(b)`;
- un'invocazione di un costruttore `new C(a)` con `new C(a)(b)`.

Perché il compilatore possa effettuare tale rimpiazzamento, è necessario che siano dichiarati con la parola riservata `implicit` sia il parametro non specificato della funzione/metodo/costruttore che l'identificatore utilizzato come argomento (nell'esempio `b`).

Il meccanismo dei parametri impliciti agisce sull'ultima lista di argomenti, eseguendo il processo di deduzione e rimpiazzamento di *tutti* gli argomenti appartenenti a tale lista quando questa non è specificata in un'invocazione (ma è sempre possibile scriverla esplicitamente, se si vogliono fornire manualmente tutti gli argomenti). Di conseguenza, nella definizione di una funzione/metodo o di un costruttore non è ammesso dichiarare parametri `implicit` in liste di parametri che non siano l'ultima, e non si possono "mischiare" parametri `implicit` e non all'interno di una stessa lista di parametri, tanto è vero che per dichiarare una lista di parametri impliciti si scrive la parola riservata `implicit` una volta sola, all'inizio della lista: la sintassi

$$\texttt{(implicit } x_1\texttt{: } T_1\texttt{, ..., } x_n\texttt{: } T_n\texttt{)}$$

dichiara appunto come `implicit` tutti i parametri $x_1, \ldots, x_n$ (e non solo x_1).

Nel caso della funzione `msort` si dichiara `implicit` il parametro `ord`, che allora può essere omesso sia nelle invocazioni ricorsive all'interno della funzione stessa

```scala
def msort[T](xs: List[T])(implicit ord: Ordering[T]): List[T] = {
  val half = xs.length / 2
  if (half == 0) xs
  else {
    def merge(xs: List[T], ys: List[T]): List[T] =
      // Implementazione che usa ord, come prima...

    val (fst, snd) = xs splitAt half
```

```
    merge(msort(fst), msort(snd))
  }
}
```

che nelle invocazioni della funzione dall'esterno (purché queste ultime siano su liste di elementi di tipi *T* per cui sono disponibile istanze di `Ordering[T]` dichiarate `implicit`):

```
val nums = List(1, 3, 7, 5, 9, -1)
msort(nums)
val fruits = List("pear", "orange", "pineapple", "apple")
msort(fruits)
```

6.1 Risoluzione dei parametri impliciti

Si consideri una funzione che prende un parametro implicito di tipo *T*. In presenza di un'invocazione della funzione in cui tale parametro non è esplicitamente indicato, il compilatore cerca una definizione che:

- sia dichiarata `implicit`;
- abbia un tipo compatibile con *T*;
- sia visibile nel punto della chiamata o sia definita nel companion object di *T*.

Se trova una e una sola definizione che soddisfa tali regole, il compilatore la usa come argomento per istanziare il parametro implicito, altrimenti (se non ci sono definizioni adeguate, oppure se si ha una situazione di ambiguità perché ce n'è più di una) segnala un errore.[4]

Ad esempio, nel caso delle invocazioni ricorsive di `msort` nel corpo della funzione `msort` stessa,

```
def msort[T](xs: List[T])(implicit ord: Ordering[T]): List[T] = {
  // ...
  merge(msort(fst), msort(snd))
  // ...
}
```

una definizione che soddisfa le tre condizioni è il parametro formale `ord`, il quale infatti:

- è dichiarato `implicit`;

[4] Le regole appena descritte sono in realtà semplificate rispetto a quelle che il compilatore realmente applica: in particolare, la ricerca delle definizioni `implicit` può avvenire anche in alcuni altri companion object oltre a quello di *T*, e quando esistono più definizioni adeguate ci sono dei criteri che danno la priorità a certe definizioni rispetto ad altre, risolvendo quindi automaticamente alcune situazioni di ambiguità (ma non tutte). Si veda https://docs.scala-lang.org/tutorials/FAQ/finding-implicits.html per una descrizione completa delle regole.

- ha tipo `Ordering[T]` (che è il tipo richiesto per le invocazioni ricorsive, in quanto esse sono su liste di tipo `List[T]`);
- è visibile nel punto della chiamata.

Siccome `T` è un tipo parametro, non ancora istanziato con un tipo specifico al momento della compilazione della funzione, e siccome non esistono definizioni di tipo `Ordering[T]` che possano essere applicate per un generico tipo `T` (tutte le definizioni predefinite di `Ordering` sono per tipi specifici), il parametro `ord` è l'unica definizione trovata, non c'è ambiguità, dunque il compilatore sostituisce le invocazioni `msort(fst)` e `msort(snd)` con `msort(fst)(ord)` e `msort(snd)(ord)`. In sostanza, all'interno della funzione `msort` la dichiarazione del parametro `ord` come `implicit` svolge due funzioni: permette al compilatore di cercare automaticamente un argomento con cui istanziare il parametro `ord` nelle chiamate ricorsive, e al tempo stesso fornisce l'argomento con cui istanziare tale parametro.

Invece, nelle invocazioni della funzione `msort` dall'esterno, considerando ad esempio il caso

```
val nums = List(1, 3, 7, 5, 9, -1)
msort(nums)
```

(ma il caso dell'invocazione sulle stringhe è analogo), siccome `nums` è di tipo `List[Int]` si ha `T` = `Int`, quindi il compilatore cerca un argomento implicito di tipo (compatibile con) `Ordering[Int]`. Il companion object `Ordering` di `Ordering[T]` contiene la definizione

```
implicit object Int extends IntOrdering
```

che soddisfa le tre condizioni:

- è dichiarata `implicit`;
- è di tipo compatibile con `Ordering[Int]`, perché `IntOrdering` è un trait che estende `Ordering[Int]`;
- è definita nel companion object del tipo del parametro implicito.

Allora, l'invocazione `msort(nums)` viene sostituita con `msort(nums)(Ordering.Int)`.

6.2 Esempio di definizione `implicit`

Quando si definisce un tipo sul quale esiste una relazione d'ordine (totale), come la classe `Rational` che rappresenta i numeri razionali,

```
class Rational(x: Int, y: Int) {
  // ...
  override def equals(that: Any): Boolean = /* ... */
  def <(that: Rational): Boolean = /* ... */
```

```
  // ...
}
```

è una buona idea definire un'istanza di `Ordering` per tale tipo, in modo da poterla riutilizzare ogni volta che si invoca un metodo generico che deve effettuare operazioni di confronto su valori del tipo. Ci sono vari modi di definire tale istanza; ad esempio, la si potrebbe implementare come singleton object:

```
object RationalOrdering extends Ordering[Rational] {
  def compare(x: Rational, y: Rational): Int =
    if (x == y) 0 else if (x < y) -1 else 1
}
```

Perché l'istanza appena definita sia utilizzabile come argomento implicito è necessario dichiararla `implicit`, ma la parola riservata `implicit` non è ammessa per le definizioni che sono scritte direttamente all'interno di un file: la si può usare solo per le definizioni che sono innestate in una classe/object/trait e per quelle scritte nell'interprete. La soluzione più semplice è assegnare l'oggetto `RationalOrdering` a un `implicit val` negli scope in cui deve essere disponibile come argomento implicito; ad esempio, nell'interprete:

```
implicit val ro = RationalOrdering
// ...
val rationals = List(Rational(1, 2), Rational(1, 3), Rational(1, 4))
msort(rationals)
// ...
```

Così, quando l'interprete cerca un argomento con cui istanziare implicitamente il parametro `ord` nella chiamata `msort(rationals)`, trova la definizione `ro` che soddisfa le tre regole:

- è dichiarata `implicit`;
- ha tipo compatibile con `Ordering[Rational]`;
- è visibile nel punto della chiamata.

Il vantaggio di assegnare `RationalOrdering` a `implicit val ro` invece di passarlo esplicitamente è che `ro` può essere definito una volta in un scope e poi usato implicitamente per tutte le chiamate di funzioni in tale scope che hanno bisogno di un'istanza di `Ordering[Rational]`, mentre se si passasse l'argomento esplicitamente bisognerebbe specificarlo a ogni chiamata.[5]

[5] Esiste anche un'altra soluzione, che è ancora più comoda ma si basa su una delle regole di risoluzione dei parametri impliciti che non sono state discusse prima: la ricerca di una definizione `implicit` di un tipo parametrico *`T[U]`* avviene anche nel companion object di *`U`*. Di conseguenza, se l'istanza di `Ordering[Rational]` fosse definita nel companion object di `Rational` (ad esempio come `implicit val` o `implicit object`, entrambi ammessi all'interno di un object), essa risulterebbe direttamente disponibile ovunque (come le definizioni predefinite `Ordering.Int`, ecc.), senza bisogno di ulteriori definizioni negli scope in cui serve.