Azzolini Riccardo 2021-05-24

Funzioni all'ordine superiore sulle liste

1 Metodo `map`

Si supponga di voler scrivere una funzione `scaleList` che, dati una lista di interi `xs` e un numero intero `factor`, restituisca una nuova lista ottenuta moltiplicando ogni elemento di `xs` per `factor`.

```
def scaleList(xs: List[Int], factor: Int): List[Int] = xs match {
  case Nil => Nil
  case y :: ys => (y * factor) :: scaleList(ys, factor)
}
```

Questo schema è generalizzato dalla funzione `map`, che applica un'arbitraria funzione a tutti gli elementi di una lista. Essa funzione è disponibile nella libreria standard standard di Scala come metodo della classe `List[+T]` (e di molte altre strutture dati). Concettualmente, la sua implementazione potrebbe essere la seguente:

```
sealed abstract class List[+T] {
  // ...
  def map[U](f: T => U): List[U] = this match {
    case Nil => Nil
    case x :: xs => f(x) :: xs.map(f)
  }
  // ...
}
```

(la reale implementazione evita gli svantaggi della ricorsione non in coda presente in quest'implementazione semplificata facendo uso delle parti imperative di Scala[1]). Si osservi che il tipo di `map` è parametrico nel codominio `U` della funzione `f` applicata agli elementi della lista originale, ovvero nel tipo degli elementi della nuova lista, che sono appunto generati applicando la funzione `f` e quindi appartengono al codominio di tale funzione.

Utilizzando `map` la funzione `scaleList` può essere riscritta come:

[1] Scrivere una versione tail-recursive di `map` è possibile, ma siccome l'unico modo efficiente di aggiungere un elemento a una lista immutable (per costruire la lista di risultati della funzione `f`) è l'inserimento in testa tramite `::` è necessario un qualche "trucco" per evitare che la lista risultante contenga gli elementi in ordine inverso a causa delle applicazioni iterative di `::` (la soluzione più semplice è applicare `reverse` al risultato), e ciò, pur mantenendo la stessa complessità, riduce un po' l'efficienza rispetto a un'implementazione imperativa.

```
def scaleList(xs: List[Int], factor: Int): List[Int] =
  xs map (x => x * factor)
```

2 Funzioni all'ordine superiore sulle liste

Mentre i metodi di `List[+T]` visti in precedenza sono funzioni al primo ordine, il metodo `map` appena introdotto è un esempio di funzione all'ordine superiore, in quanto prende come argomento una funzione.

In generale, le funzioni all'ordine superiore sulle liste (`map`, e altre che verranno presentate nel seguito) appartengono a tre principali famiglie:

- funzioni di **map**, che trasformano gli elementi di una lista;
- funzioni di **filtering**, che selezionano una parte degli elementi di una lista, restituendo una nuova lista costituita dai soli elementi che soddisfano un determinato criterio;
- funzioni di **reduce**, che combinano gli elementi di una lista usando un operatore.

Questi tipi di funzioni esistono non solo per le liste, ma per praticamente tutte le strutture dati.

3 Metodo `filter`

Un esempio di filtering è la seguente funzione, che restituisce la lista di tutti i valori positivi contenuti in una lista di interi:

```
def posElem(xs: List[Int]): List[Int] = xs match {
  case Nil => Nil
  case y :: ys => if (y > 0) y :: posElem(ys) else posElem(ys)
}
```

Generalizzando lo schema di `posElem` si ottiene il metodo `filter`, che è definito nella classe `List[+T]` e potrebbe essere implementato in questo modo:

```
sealed abstract class List[+T] {
  // ...
  def filter(p: T => Boolean): List[T] = this match {
    case Nil => Nil
    case x :: xs => if (p(x)) x :: xs.filter(p) else xs.filter(p)
  }
  // ...
}
```

Esso prende come argomento un predicato, cioè una funzione di tipo `T => Boolean`, e seleziona gli elementi che soddisfano tale predicato, costruendo una nuova lista contenente solo questi.

Con `filter` l'implementazione di `posElem` diventa:

```
def posElem(xs: List[Int]): List[Int] =
  xs filter (x => x > 0)
```

3.1 Altri metodi di filtering

Ci sono vari altri metodi di filtering, sostanzialmente varianti di `filter`; i principali sono:

- `xs filterNot p`, che equivale a `xs filter (x => !p(x))`, ovvero inverte l'applicazione del predicato `p`, selezionando gli elementi di `xs` che *non* lo soddisfano.
- `xs partition p`, che partiziona la lista in una coppia di sottoliste, contenenti rispettivamente gli elementi che soddisfano e che non soddisfano il predicato `p`; esso equivale a `(xs filter p, xs filterNot p)`, ma è più efficiente perché attraversa la lista `xs` una sola volta invece di due.
- `xs takeWhile p`, che restituisce il più lungo prefisso di `xs` formato solo da elementi che soddisfano `p`.
- `xs takeWhile p`, che restituisce i restanti elementi della lista dopo aver il più lungo prefisso di `xs` formato solo da elementi che soddisfano `p`, ovvero restituisce gli elementi di `xs` a partire dal primo che *non* soddisfa `p`.
- `xs span p`, che equivale a `(xs takeWhile p, xs dropWhile p)` ma attraversa una sola volta la lista `xs`.

4 Abbreviazione per la definizione di funzioni

La sintassi per la definizione di funzioni anonime tende a essere ridondante nel caso di funzioni semplici, come ad esempio `(x, y) => x + y`: il corpo `x + y` della funzione mette già in evidenza i parametri formali, che dunque si preferirebbe non dover riscrivere a sinistra di `=>`. A tale scopo, Scala mette a disposizione dello zucchero sintattico che permette di abbreviare la funzione come `_ + _`:

- ogni `_` (underscore) da sinistra verso destra rappresenta un nuovo parametro;
- il corpo della funzione anonima è delimitato dalla prima coppia di parentesi che racchiude tutti i parametri `_`, oppure è costituito dall'intera espressione se non ci sono parentesi.

Con questa sintassi abbreviata i precedenti esempi d'uso di `map` e `filter` possono essere riscritti come segue:

```
def scaleList(xs: List[Int], factor: Int): List[Int] =
  xs map (_ * factor)  // invece di (x => x * factor)
def posElem(xs: List[Int]): List[Int] =
  xs filter (_ > 0)    // invece di (x => x > 0)
```

5 Metodi di reduce

I metodi di reduce, che combinano gli elementi di una lista per mezzo di un operatore, sono un po' più complicati rispetto a quelli di map e di filtering, poiché devono gestire alcuni aspetti legati all'associatività dell'operatore applicato e al valore usato per inizializzare il processo di riduzione.

Alcuni esempi specifici di operazioni di reduce sono le seguenti funzioni `sum` e `prod`, che combinano gli elementi di una lista rispettivamente con gli operatori `+` e `*`:

```
def sum(xs: List[Int]): Int = xs match {
  case Nil => 0
  case y :: ys => y + sum(ys)
}
def prod(xs: List[Int]): Int = xs match {
  case Nil => 1
  case y :: ys => y * sum(ys)
}
```

Gli elementi sono combinati usando operatori binari, che in quanto tali richiedono due operandi per essere applicati. Allora, nel caso `Nil` delle definizioni di queste funzioni viene stabilito un valore che:

- sia restituito quando, nell'ultima chiamata ricorsiva, si opera su una lista vuota;
- funga da secondo operando quando, nella penultima chiamata ricorsiva, la lista contiene un solo elemento (usato come primo operando).

In `sum` e `prod` tale valore è l'elemento neutro dell'operatore applicato, che in quanto tale non influisce sul risultato, così `sum(xs)` restituisce correttamente la somma degli elementi della lista `xs`, e allo stesso modo `prod(xs)` restituisce il prodotto degli elementi di `xs`:

$$\begin{aligned} \texttt{sum(List(}x_1\texttt{, }x_2\texttt{, ..., }x_n\texttt{))} &= x_1 \texttt{ + } x_2 \texttt{ + ... + } x_n \texttt{ + 0} \\ &= x_1 \texttt{ + } x_2 \texttt{ + ... + } x_n \\ \texttt{prod(List(}x_1\texttt{, }x_2\texttt{, ..., }x_n\texttt{))} &= x_1 \texttt{ * } x_2 \texttt{ * ... * } x_n \texttt{ * 1} \\ &= x_1 \texttt{ * } x_2 \texttt{ * ... * } x_n \end{aligned}$$

In seguito verranno presentati i principali metodi di reduce forniti dalla classe `List[+T]`, che realizzano diverse varianti dello schema di `sum` e `product`.

5.1 Metodo `reduceLeft`

Un primo metodo di reduce di `List[+T]` è

```
def reduceLeft[U >: T](op: (U, T) => U): U
```

che applica l'operatore binario `op` a tutti gli elementi della lista su cui il metodo è invocato, procedendo da sinistra verso destra:

$$\texttt{List}(x_1, x_2, \ldots, x_n)\ \texttt{reduceLeft}\ op = (\ldots(x_1\ op\ x_2)\ op\ \ldots)\ op\ x_n$$

Come casi particolari:

- se la lista ha un unico elemento, `reduceLeft` restituisce tale elemento senza applicare alcun operatore;
- se la lista è vuota, `reduceLeft` solleva un'eccezione (poiché non c'è alcun valore da restituire).

Si osservi il tipo della funzione `op`, `(U, T) => U`: concettualmente, tale operatore combina gli elementi della lista, che sono di tipo `T`, producendo un risultato di tipo `U`. Tuttavia, siccome le applicazioni dell'operatore procedono da sinistra a destra, in ogni applicazione tranne la prima l'operando sinistro è il risultato dell'applicazione precedente, e non un elemento della lista originale, perciò il tipo del primo parametro di `op` è `U` e non `T`. Invece, nella prima applicazione dell'operatore, $x_1\ op\ x_2$, entrambi gli operandi sono elementi della lista, quindi è necessario poter passare al parametro di tipo `U` di `op` un valore di tipo `T`. Inoltre, un valore di tipo `T` deve poter essere trattato come uno di tipo `U` anche perché quando c'è un solo elemento nella lista esso viene restituito direttamente, senza "passare" dall'operatore. Per questi due motivi, `reduceLeft` impone sul tipo parametro `U` il lower bound `U >: T`, cioè richiede che il tipo `U` del risultato sia un supertipo del tipo `T` degli elementi della lista, in modo che appunto un elemento della lista possa essere passato come argomento di tipo `U` e restituito come risultato di tipo `U`.

5.1.1 Esempio: `sum` e `prod`

Il metodo `reduceLeft` può essere usato per riscrivere `sum` e `prod`, ma bisogna fare attenzione al caso in cui l'argomento `xs` è lista vuota, nel quale `sum(xs)` e `prod(x)` devono restituire l'elemento neutro mentre `xs.reduceLeft` solleva un'eccezione. Una soluzione è fare in modo che la lista su cui opera `reduceLeft` non sia mai vuota, "allungando" `xs` tramite l'aggiunta dell'elemento neutro, che così diventa il valore restituito quando `xs` è vuota e non influisce sul risultato (in quanto elemento neutro) negli altri casi:

```
def sum(xs: List[Int]): Int = (0 :: xs) reduceLeft (_ + _)
def prod(xs: List[Int]): Int = (1 :: xs) reduceLeft (_ * _)
```

5.1.2 Esempio: `Sum` di `Number`

Come altro esempio, si consideri la gerarchia delle espressioni aritmetiche,

```
sealed trait Expr {
  def eval: Int = /* ... */
}
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 { /* ... */ }
```

e si supponga di voler scrivere una funzione `sumExprOf` che, data una lista di `Number`, restituisca l'espressione `Sum` che rappresenta la somma di tali `Number`, o sollevi un'eccezione se la lista non contiene almeno due elementi (così il valore restituito sarà sempre un'espressione di tipo `Sum`, e mai un singolo `Number`, cosa che invece avverrebbe se fosse ammesso il caso di una lista con un solo elemento). `sumExprOf` può essere implementata usando `reduceLeft` (e il pattern matching per verificare che ci siano almeno due elementi):

```
def sumExprOf(nums: List[Number]): Expr = nums match {
  case _ :: _ :: _ =>  // Almeno due elementi
    nums reduceLeft ((x: Expr, y: Number) => Sum(x, y))
  case _ => throw
    new UnsupportedOperationException("list must have length >= 2")
}
```

Si noti che il tipo del risultato e del primo parametro dell'operatore è `Expr`, e non `Sum`, perché come detto prima esso deve essere compatibile sia con gli elementi della lista, qui di tipo `Number`, che con il risultato dell'operatore, il quale è un'istanza di `Sum`: `Expr` è il più piccolo supertipo comune a `Number` e `Sum`.

La funzione `sumExprOf` viene usato in questo modo:

```
val numbers = List(Number(2), Number(5), Number(7))
sumExprOf(numbers)  // Risultato: Sum(Sum(Number(2),Number(5)),Number(7))
```

5.2 Metodo `foldLeft`

Un altro metodo di reduce di `List[+T]`, simile a `reduceLeft` ma più flessibile e usato più spesso in pratica, è

```
def foldLeft[U](z: U)(op: (U, T) => U): U
```

che riceve il valore iniziale `z` come argomento invece di usare il primo elemento della lista su cui è invocato, ovvero applica l'operatore binario `op` al valore iniziale `z` e a tutti gli elementi della lista, procedendo da sinistra verso destra:

$$\texttt{List}(x_1, x_2, \ldots, x_n).\texttt{foldLeft}(z)(op) = (\ldots((z\ op\ x_1)\ op\ x_2)\ op\ \ldots)\ op\ x_n$$

Siccome il valore iniziale è passato come argomento:

- non è più necessario che il tipo `T` degli elementi della lista sia compatibile con il tipo `U` del risultato, cioè non serve più il vincolo `U >: T`, perché gli elementi della lista non vengono mai restituiti direttamente né passati come primo argomento di `op`;
- quando il metodo è invocato su una lista vuota restituisce `z` invece di sollevare un'eccezione.

5.2.1 Esempio: `sum` e `prod`

`foldLeft` permette di riscrivere le funzioni `sum` e `prod` in modo un po' più elegante rispetto a `reduceLeft`, passando l'elemento neutro come argomento invece di doverlo aggiungere all'inizio della lista:

```
def sum(xs: List[Int]): Int = xs.foldLeft(0)(_ + _)
def prod(xs: List[Int]): Int = xs.foldLeft(1)(_ * _)
```

5.2.2 Esempio: somma dei valori di delle espressioni

Un altro esempio di uso di `foldLeft` è la seguente funzione `sumOf`, che data una lista di espressioni (rappresentate dalla solita gerarchia di classi) valuta ciascuna di esse (supponendo che non ci siano variabili, la cui valutazione richiederebbe di fissarne in qualche modo i valori) e restituisce la somma dei risultati:

```
def sumOf(exprs: List[Expr]): Int =
  exprs.foldLeft(0)((x: Int, y: Expr) => x + y.eval)
```

Si osservi che qui i tipi dei due parametri della funzione `op` (indicati esplicitamente a scopo illustrativo, poiché il compilatore è in grado di dedurli in base ai tipi degli elementi della lista e del valore iniziale[2]) sono `Int` (il tipo del risultato) e `Expr` (il tipo degli

[2] Siccome il valore iniziale fornisce l'informazione sul tipo del risultato, la deduzione dei tipi tende a funzionare più spesso con `foldLeft` che con `reduceLeft`, dove invece il tipo del risultato può essere dedotto solo in base all'operatore, e se il compilatore non riesce a ricavare da quest'ultimo informazioni sufficienti assume semplicemente `U` = `T`, cioè che il tipo del risultato sia uguale a quello degli elementi della lista. Ad esempio, nel precedente esempio `sumExprOf` (paragrafo 5.1.2) è necessario indicare esplicitamente i tipi dei parametri dell'operatore passato a `reduceLeft`, oppure istanziare manualmente `U` = `Expr`, altrimenti il compilatore deduce erroneamente `U` = `T` = `Number` e poi segnala che il valore `Sum` restituito dall'operatore non è compatibile con il tipo `U` = `Number`.

elementi); ciò non sarebbe stato possibile con `reduceLeft`, poiché `Int` non è supertipo di `Expr`.

`sumOf` può essere usato in questo modo:

```
val exprList = List(Number(1), Sum(Number(3), Number(4)))
sumOf(exprList)  // Risultato: 8
```

Un'implementazione alternativa di `sumOf` è la seguente, che separa le operazioni di valutazione delle espressioni e di somma, permettendo così il riuso della funzione `sum`:

```
def sumOf(exprs: List[Expr]): Int = sum(exprs.map(_.eval))
```

5.3 `reduceRight` e `foldRight`

Altri due metodi di reduce di `List[+T]` sono

```
def reduceRight[U >: T](op: (T, U) => U): U
def foldRight[U](z: U)(op: (T, U) => U): U
```

che sono i metodi duali, in qualche modo "simmetrici", a `reduceLeft` e `foldLeft` rispettivamente, in quanto applicano l'operatore binario agli elementi della lista (e al valore iniziale, nel caso di `foldRight`) procedendo da destra verso sinistra:

$$\begin{aligned}&\texttt{List}(x_1, \ldots, x_{n-1}, x_n)\ \texttt{reduceRight}\ op\\ &\quad = x_1\ op\ (\ldots\ op\ (x_{n-1}\ op\ x_n)\ldots)\\ &\texttt{List}(x_1, \ldots, x_{n-1}, x_n)\texttt{.foldRight}(z)(op)\\ &\quad = x_1\ op\ (\ldots\ op\ (x_{n-1}\ op\ (x_n\ op\ z))\ldots)\end{aligned}$$

Si noti che i ruoli degli operandi dell'operatore, e di conseguenza i tipi dei parametri della funzione `op`, sono scambiati:

- l'operando sinistro è un elemento della lista, di tipo `T`;
- l'operando destro è il risultato della precedente applicazione dell'operatore (o il valore iniziale), di tipo `U`.

Di conseguenza, un operatore che ha un tipo compatibile con `foldLeft` / `reduceLeft` potrebbe non essere compatibile con `foldRight` / `reduceRight` (a meno di alterare la definizione dell'operatore, scambiando le posizioni dei due parametri della funzione), e viceversa.

5.4 Equivalenza tra `foldLeft` e `foldRight`

In generale `foldLeft` e `foldRight` non sono equivalenti, ma lo sono in particolare quando l'operatore specificato come argomento è *associativo* e *commutativo*, perché:

- `foldLeft` associa le applicazioni dell'operatore a sinistra, mentre `foldRight` le associa a destra, e ciò è irrilevante se l'operatore è associativo;
- `foldLeft` mette il valore iniziale come l'operando più a sinistra, mentre `foldRight` lo mette come l'operando più a destra, e ciò è irrilevante se l'operatore è anche commutativo;
- un operatore può essere commutativo e associativo solo se ha un tipo della forma `(T, T) => T` (se i tipi dei due operandi e/o del risultato fossero diversi, non si potrebbero "spostare liberamente" gli operandi e le parentesi senza introdurre errori di tipo), che è compatibile con i tipi richiesti sia da `foldLeft` che da `foldRight`.

Ad esempio, nel caso di `+` sugli `Int`:

$$\begin{aligned}&\texttt{List}(x_1, \ldots, x_n)\texttt{.foldLeft}(z)(\_ + \_)\\ &\quad = (\ldots(z + x_1) + \ldots) + x_n\\ &\quad = x_1 + (\ldots + (x_n + z)\ldots)\\ &\quad = \texttt{List}(x_1, \ldots, x_n)\texttt{.foldRight}(z)(\_ + \_)\end{aligned}$$

5.5 Implementazione dei metodi di reduce

Delle possibili implementazioni (semplificate) per i quattro metodi di reduce appena presentati sono le seguenti:[3]

```
sealed abstract class List[+T] {
  // ...
  def reduceLeft[U >: T](op: (U, T) => U): U = this match {
    case Nil =>
      throw new UnsupportedOperationException("Nil.reduceLeft")
    case x :: xs => xs.foldLeft[U](x)(op)
  }

  def foldLeft[U](z: U)(op: (U, T) => U): U = this match {
    case Nil => z
    case x :: xs => xs.foldLeft(op(z, x))(op)
  }
```

[3] `reduceRight` potrebbe essere implementato usando `foldRight`, così come `reduceLeft` è implementato usando `foldLeft`, ma ciò richiederebbe l'uso dei metodi `last` e `init` per estrarre l'ultimo elemento (da usare come valore iniziale) e il resto della lista (su cui invocare `foldRight`), dunque la lista verrebbe attraversata più volte, mentre l'implementazione mostrata qui esegue un solo attraversamento.

```
  def reduceRight[U >: T](op: (T, U) => U): U = this match {
    case Nil =>
      throw new UnsupportedOperationException("Nil.reduceRight")
    case x :: Nil => x
    case x :: xs => op(x, xs.reduceRight(op))
  }

  def foldRight[U](z: U)(op: (T, U) => U): U = this match {
    case Nil => z
    case x :: xs => op(x, xs.foldRight(z)(op))
  }
  // ...
}
```