Azzolini Riccardo 2021-04-12

Operatori, overloading e conversioni implicite

1 Introduzione

I numeri razionali modellati dalla classe `Rational` sono un concetto tanto naturale quanto i numeri interi modellati da `Int`, ma dal punto di vista dell'utilizzo c'è una caratteristica che distingue sintatticamente le due astrazioni:

- si scrive `x + y` per indicare la somma di interi;
- si scrive `x.add(y)` per indicare la somma di razionali.

Il linguaggio Scala (come molti altri linguaggi, ma non Java) mette a disposizione degli strumenti per eliminare questa differenza, che in particolare sono la notazione infissa e gli identificatori simbolici.

2 Notazione infissa

Tutti i *metodi con un unico argomento* possono essere utilizzati in **notazione infissa**. Ad esempio, nel caso della classe `Rational` si può scrivere:

- `x add y` invece di `x.add(y)`,
- `x sub y` invece di `x.sub(y)`,
- `x less y` invece di `x.less(y)`,
- ecc.

L'invocazione di un metodo in notazione infissa è zucchero sintattico, che il compilatore semplicemente riscrive in una normale invocazione con la dot-notation.

3 Classificazione degli identificatori

In Scala gli identificatori si distinguono, in base ai caratteri da cui sono composti, in due categorie: alfanumerici e simbolici.

3.1 Identificatori alfanumerici

Gli **identificatori alfanumerici** iniziano con una lettera, seguita da una sequenza di lettere e cifre, e infine, opzionalmente, un carattere `_` (underscore) seguito da degli operatori.

I caratteri `$` e `_` sono considerati lettere, ma:

- `$` non dovrebbe essere dal programmatore, in quanto viene usato dal compilatore Scala per contrassegnare i nomi di elementi del codice generati automaticamente (classi di supporto, ecc.): se lo si usasse, si potrebbero dunque creare conflitti con tali nomi.
- `_` va usato con attenzione, in quanto ha dei significati speciali nel linguaggio.

Alcuni esempi di identificatori alfanumerici sono:

```
x1    _Vector_++    counter_=
```

Si noti che il compilatore Scala interpreta `counter_=` come un unico identificatore, e non come `counter_` (anch'esso un identificatore valido) seguito da un operatore `=`; lo stesso vale per `_Vector_++`. Invece, ad esempio, `counter_2=` viene interpretato come `counter_2` seguito da `=`, perché l'operatore `=` non è "collegato" al resto dell'identificatore da un underscore: se vuole che il tutto costituisca un singolo identificatore, bisogna scrivere `counter_2_=`.

3.2 Identificatori simbolici

Gli **identificatori simbolici** iniziano con un operatore, seguito eventualmente da altri operatori. Tra tutti i simboli che sono considerati operatori, i due punti `:` hanno un significato particolare, che verrà spiegato a breve.

Alcuni esempi di identificatori simbolici sono:

```
*    +?%&    +    ++    :::    <?>    :->
```

4 Esempio: operatori su `Rational`

Usando insieme la notazione infissa e gli identificatori simbolici, si possono definire degli operatori binari infissi sui valori della classe `Rational`, analoghi a quelli disponibili sui valori `Int`. Tali operatori vengono definiti esattamente come i metodi "normali", semplicemente usando nomi che sono identificatori simbolici:

```
class Rational(x: Int, y: Int) {
  require(y > 0, "denominator must be positive")
  val num = x / abs(gcd(x, y))
  val den = y / abs(gcd(x, y))

  // ...

  def +(that: Rational): Rational = new Rational(
    this.num * that.den + that.num * this.den,
    this.den * that.den
  )

  def *(that: Rational): Rational =
    new Rational(this.num * that.num, this.den * that.den)

  def /(that: Rational): Rational =
    new Rational(this.num * that.den, this.den * that.num)

  def <(that: Rational): Boolean =
    this.num * that.den < that.den * this.num

  def max(that: Rational): Rational =
    if (this < that) that else this

  // ...
}
```

Qui la definizione del metodo `max` contiene un esempio di uso di uno degli operatori appena definiti: `this < that`. Il compilatore riscrive tale uso in notazione infissa come `this.<(that)`, una normale invocazione di un normale metodo chiamato `<`.

5 Operatori unari prefissi

Sfruttando le convenzioni appena presentate è stato possibile usare la notazione usuale per le operazioni binarie della classe `Rational`. Adesso, si vorrebbe poter usare la notazione usuale anche per l'operatore unario prefisso `-`, implementato dal seguente metodo `neg`:

```
def neg(): Rational = new Rational(-this.num, this.den)
```

Anche questa possibilità è offerta da Scala tramite un'apposita convenzione: per definire un **operatore unario prefisso**, bisogna aggiungere al nome dell'operatore il prefisso `unary_`. Ad esempio, creando un metodo il cui nome è l'identificatore alfanumerico `unary_-` si definisce l'operatore unario prefisso `-`:

```
def unary_-(): Rational = new Rational(-this.num, this.den)
```

Le parentesi della lista vuota di parametri possono essere omesse:

```
def unary_- : Rational = new Rational(-this.num, this.den)
```

Si noti che è necessario inserire uno spazio tra l'ultimo carattere (`-`) dell'identificatore e i due punti che indicano il tipo restituito (se esso non è lasciato implicito), altrimenti il compilatore Scala interpreterebbe `unary_-:` come un unico identificatore e segnalerebbe un errore di sintassi.

L'operatore unario prefisso `-` così definito può essere usato, ad esempio, per definire l'operatore di sottrazione tra valori `Rational`:

```
def -(that: Rational): Rational = this + -that
```

Come si può osservare, è ammesso definire un operatore unario prefisso e uno binario infisso identificati dagli caratteri (in questo caso `-`). Quando uno di questi due operatori compare in un'espressione, il compilatore determina se è usato in modo prefisso o infisso (guardando se a sinistra dell'operatore è presente o meno un operando), e in base a ciò riscrive l'uso dell'operatore come un'invocazione del metodo corretto. Ad esempio, l'espressione

```
new Rational(1, 2) - -new Rational(3, 2)
```

viene riscritta come

```
new Rational(1, 2).-(new Rational(3, 2).unary_-)
```

6 Precedenze e associatività

Per evitare ambiguità nell'uso degli operatori infissi è necessario stabilirne le precedenze e associatività. In Scala, esse sono determinate in base a delle convenzioni sui caratteri che compongono gli identificatori degli operatori.

6.1 Precedenza

La **precedenza** di un operatore è determinata dal suo *primo carattere*, secondo la seguente tabella:

Precedenza	Caratteri
più alta	tutti gli altri caratteri speciali
	`* / %`
	`+ -`
	`:`
	`= !`
	`< >`
	`&`
	`^`
	`|`
	tutte le lettere
più bassa	(tutti gli operatori di assegnamento)

L'eccezione alla regola del primo carattere sono gli **operatori di assegnamento**, ovvero gli operatori che terminano con `=` (ad esempio `=`, `+=`, `+?%&=`, ecc.) e che non sono uno degli operatori di confronto `<=`, `>=`, `==` e `!=`: essi hanno la precedenza minima, più bassa di tutti gli altri operatori. Comunque, tali operatori non verranno utilizzati in questo corso, poiché rientrano nella parte imperativa del linguaggio Scala.

I seguenti esempi mostrano come il compilatore applica le regole di precedenza per determinare il modo in cui riscrivere gli operatori infissi in normali invocazioni di metodi:

```
    2 + 2 * 7  ≡    2 + (2 * 7)    ≡  2.+(2.*(7))
x * x + y * y  ≡ (x * x) + (y * y) ≡  (x.*(x)).+(y.*(y))
a +++ b *** c  ≡  a +++ (b *** c)  ≡  a.+++(b.***(c))
```

6.2 Associatività

L'**associatività** di un operatore è determinata dal suo *ultimo carattere*:

- Ogni operatore che termina con il carattere `:` (due punti) è associativo a destra, e corrisponde a un metodo che viene *invocato sull'operando destro* fornendo come argomento l'operando sinistro (ad esempio, l'espressione `x ::: y` viene riscritta come `y.:::(x)`). Si vedranno più avanti alcune situazioni in cui ciò è utile.
- Gli operatori che terminano con un qualunque altro carattere sono associativi a sinistra e vengono applicati nel modo spiegato in precedenza, cioè invocando un medodo sull'operando sinistro e fornendo come argomento l'operando destro (ad esempio, `x * y` viene riscritta come `x.*(y)`).

Alcuni semplici esempi di applicazione di queste regole sono:

```
    x * y * z  ≡   (x * y) * z   ≡  (x.*(y)).*(z)
x ::: y ::: z  ≡ x ::: (y ::: z) ≡  (z.:::(y)).:::(x)
```

In generale, l'associatività viene considerata quando in un'espressione compaiono più operatori con la stessa precedenza, che possono essere più occorrenze dello stesso operatore, come negli esempi appena visti, ma possono anche essere operatori diversi:

```
  x * y / z  ≡   (x * y) / z   ≡  (x.*(y))./(z)
x *: y /: z  ≡  x *: (y /: z)  ≡  (z./:(y)).*:(x)
```

Non è però ammesso "mischiare" (senza le parentesi) operatori aventi la stessa precedenza e associatività diverse (uno a sinistra e l'altro a destra), perché l'interpretazione di un'espressione del genere sarebbe ambigua. Ad esempio, l'espressione `x * y /: z` potrebbe essere interpretata come `(x * y) /: z`, se si rispettasse l'associatività a sinistra di `*`, oppure come `x * (y /: z)`, secondo l'associatività a destra di `/:`, quindi per evitare ambiguità viene generato un errore in compilazione. Allora, se si vogliono usare insieme questi operatori è obbligatorio aggiungere le parentesi esplicite, cosa che bene fare comunque in tutte le espressioni (tranne magari quelle particolarmente semplici).

6.2.1 Ordine di valutazione

Indipendentemente dall'associatività, gli operandi di un operatore sono valutati da sinistra verso destra. Allora, in particolare, un'espressione come `x ::: y` non è tecnicamente equivalente a `y.:::(x)`, perché così `y` verrebbe valutato prima di `x`, ma piuttosto equivale a

```
{ val a = x; y.:::(a) }
```

cioè a un blocco che prima valuta l'operando sinistro `x`, e solo dopo valuta l'operando destro `y` per eseguire l'invocazione del metodo. Tuttavia, l'ordine di valutazione è importante solo se gli operandi hanno effetti collaterali, ovvero se si usano gli aspetti imperativi di Scala. Perciò, in assenza di effetti collaterali, `x ::: y` può per semplicità essere considerata di fatto equivalente a `y.:::(x)`.

7 Overloading

Attualmente la classe `Rational` non permette di effettuare operazioni tra numeri razionali e interi: ad esempio, sarebbe naturale scrivere

```
new Rational(1, 2) + 1
```

ma ciò non è possibile perché la segnatura del metodo `+` prevede un argomento di tipo `Rational`, e non `Int`.

Questo problema può essere (parzialmente) risolto sfruttando la possibilità di avere **overloading** sui metodi, cioè di definire in una classe più metodi aventi lo stesso nome ma segnature differenti. Così, si possono aggiungere a `Rational` delle versioni dei metodi per le varie operazioni che prevedano un `Int` come argomento:

```
class Rational(x: Int, y: Int) {
  require(y > 0, "denominator must be positive")
  val num = x / abs(gcd(x, y))
  val den = y / abs(gcd(x, y))

  // ...

  def +(that: Rational): Rational = new Rational(
    this.num * that.den + that.num * this.den,
    this.den * that.den
  )

  def +(i: Int): Rational =
    new Rational(this.num + i * this.den, this.den)

  def *(that: Rational): Rational =
    new Rational(this.num * that.num, this.den * that.den)

  def *(i: Int): Rational =
    new Rational(this.num * i, this.den)

  // ...
}
```

Grazie all'overloading è ora possibile scrivere, ad esempio:

```
val r = new Rational(1, 2)
r + 1
```

Come in Java, in Scala la *risoluzione dell'overloading*, cioè la selezione della segnatura del metodo da eseguire, avviene in *fase di compilazione* (compile-time), e la segnatura scelta è quella più adatta al tipo dei parametri attuali specificati nell'invocazione del metodo (per la precisione, è quella che richiede il minor numero di promozioni, ovvero di conversioni implicite di tipo). Se ci sono più segnature "ugualmente adatte", il compilatore segnala un errore di ambiguità (*ambiguous reference error*), che può ad esempio essere risolto tramite conversioni esplicite di uno o più argomenti.

Ad esempio, data la precedente espressione `r + 1`, ovvero `r.+(1)`, il compilatore sa che il tipo di `r` è `Rational`, quindi cerca tra tutti i metodi definiti o ereditati da `Rational` quello con la segnatura più adatta al tipo `Int` del parametro attuale `1`.

8 Definizione di conversioni implicite

Con l'overloading è possibile scrivere, ad esempio, `r + 1`, dove `r` è di tipo `Rational`, ma non è ancora possibile scrivere `1 + r`, perché tale espressione corrisponde all'invocazione di metodo `1.+(r)`, quindi il compilatore cerca un metodo compatibile con l'argomento `Rational` nella classe di cui il valore `1` è istanza, che è `Int`. Chiaramente, `Int` non definisce un metodo di somma con un valore `Rational` (perché `Rational` è un tipo definito dall'utente, mentre `Int` è predefinito), quindi si ha un errore di compilazione.

Una possibile soluzione è definire una **conversione implicita** da `Int` a `Rational` (eseguita, ad esempio, tramite il costruttore ausiliario a un argomento visto in precedenza):

```
implicit def intToRational(x: Int) = new Rational(x)
```

Il modificatore `implicit` indica al compilatore di provare ad applicare automaticamente questa funzione di conversione per risolvere gli errori legati ai tipi, quando ciò è possibile. Ad esempio, sapendo che

- il tipo `Int` non ha un metodo `+` con un argomento `Rational`,
- invece, un tale metodo esiste nella classe `Rational`,
- è disponibile una conversione implicita da `Int` a `Rational`,

il compilatore riscrive l'espressione `1.+(r)` come `intToRational(1).+(r)`, riuscendo così a eliminare l'errore di compilazione.

In pratica, tuttavia, questa soluzione è abbastanza scomoda, perché la definizione della conversione `intToRational` non è parte del codice della classe `Rational`, dunque va importata (separatamente da `Rational`) in ogni scope in cui è necessaria.