Azzolini Riccardo 2021-04-15

Ereditarietà

1 Classi astratte

Per illustrare gli aspetti legati all'ereditarietà nel linguaggio Scala, si vuole come esempio definire una struttura dati che rappresenti un insieme di numeri interi.

Quando si crea una struttura dati, tipicamente la si caratterizza prima in modo astratto, specificando le operazioni che possono essere effettuate su di essa, e poi si forniscono una o più implementazioni concrete (possibilmente con complessità delle operazioni diverse, ecc.) tra cui scegliere in base alle esigenze dell'applicazione.

In generale, un insieme è una "collezione" di elementi, che vengono mantenuti in modo non ordinato e senza ripetizioni. Su di esso è possibile definire svariate operazioni, ma qui per iniziare si inizia a considerarne due:

- `contains`, che verifica se un insieme contiene un determinato elemento;
- `add`, che aggiunge un elemento a un insieme, *restituendo un nuovo insieme* (e non modificando l'insieme esistente, dato che ciò non è ammesso in un contesto puramente funzionale).

Il tipo che rappresenta la struttura dati insieme di interi può allora essere modellato mediante la seguente **classe astratta**, che *dichiara ma non implementa* i metodi corrispondenti alle operazioni che caratterizzano questo tipo:

```
abstract class IntSet {
  def add(x: Int): IntSet
  def contains(x: Int): Boolean
}
```

In Scala, le classi astratte funzionano sostanzialmente come in Java:

- Una classe astratta viene dichiarata usando il modificatore `abstract`.
- Una classe astratta può contenere **metodi astratti**, che sono i metodi dichiarati senza specificarne il corpo (`= ...`). A differenza di Java, non si usa il modificatore `abstract` per contrassegnare i metodi astratti.
- Una classe astratta può anche contenere **metodi concreti**, cioè "normali" metodi non astratti, definiti specificandone il corpo.

- Una classe deve obbligatoriamente essere dichiarata astratta se contiene almeno un membro astratto, ma può essere dichiarata astratta anche se contiene solo membri concreti (o se è vuota).
- Non è possibile creare istanze di una classe astratta (con l'operatore `new`).

In questo caso, `IntSet` non ha parametri di classe, ma in generale una classe astratta può avere parametri. Siccome non è possibile istanziare una classe astratta, i suoi parametri attuali vengono specificati solo dai costruttori primari delle sottoclassi, che richiamano il costruttore (primario o ausiliario) della classe astratta.

2 Alberi binari di ricerca

Adesso, bisogna fornire un'implementazione concreta degli insiemi di interi. Nella definizione matematica di un insieme gli elementi non sono ordinati, ma quando essi vengono memorizzati si ha per forza un qualche ordine, esplicito o implicito, e bisogna fare attenzione che esso non si ripercuota sui contratti dei metodi. Tuttavia, scegliendo un'opportuna struttura dati per rappresentare l'insieme si può trarre vantaggio dall'ordine degli elementi, sfruttandolo per rendere più efficienti le operazioni. Una tale struttura dati sono gli **alberi binari di ricerca** (**BST**, **Binary Search Tree**). Prima di implementarli, è utile presentarne la definizione formale.

Innanzitutto, un **albero binario** è definito come

- o l'albero vuoto,
- oppure una tripla (N, L, R) in cui N è un *nodo* e L, R sono alberi binari — L è detto *sottoalbero sinistro* e R è detto *sottoalbero destro*.

Si noti che la definizione è ricorsiva, cioè usa essa stessa il concetto di albero binario. Come si vedrà a breve, quando si ha la definizione ricorsiva di una struttura è naturale definire le operazioni su di essa tramite funzioni ricorsive.

A partire da un albero binaro, per definire un BST è necessario fissare una proprietà invariante sull'albero, legata al concetto di ordinamento degli elementi associati ai nodi. Allora, per essere precisi, è necessario definire anche cosa si intenda con "ordinamento"[1] e con "elemento associato a un nodo".

Un **ordine totale** (o **lineare**) è una struttura $(U, \leq)$ in cui:

- U è un insieme;
- $\leq$ è una relazione binaria su U, cioè $\leq \subseteq U \times U$;

[1] Nel caso dei numeri interi, il significato di un ordinamento è intuitivo, ma dare una definizione precisa consente la generalizzazione a tipi di elementi diversi.

- $\leq$ è un **ordine lineare** su U, cioè è un **ordine parziale** su U (è una relazione riflessiva, transitiva e antisimmetrica) ed è tale che tutti gli elementi sono "confrontabili" tra di loro,
$$\forall x, y \ (x \leq y \vee y \leq x)$$

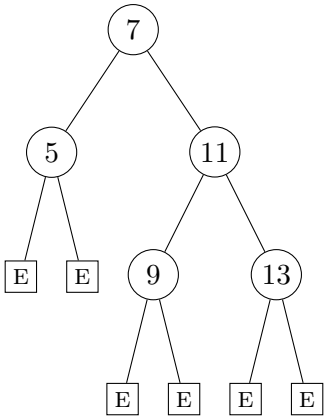
Dati $a, b \in U$, si scrive $a < b$ o $b > a$ per indicare che $a \leq b$ e $a \neq b$.

Un'**etichettatura** su un dominio D per un albero è una funzione l che associa a ogni nodo dell'albero un valore in D. Si chiama **etichettato** un albero arricchito di una funzione di etichettatura dei suoi nodi.

Dato un ordine totale $(U, \leq)$, un **BST** per un sottoinsieme $S \subseteq U$ è un albero binario etichettato in cui ogni nodo (o *vertice*) v è etichettato da un elemento $l(v) \in S$ tale che:

- per ogni nodo u nel sottoalbero sinistro di v, $l(u) < l(v)$;
- per ogni nodo u nel sottoalbero destro di v, $l(u) > l(v)$;
- per ogni elemento $a \in S$ c'è esattamente un vertice v tale che $l(v) = a$, ovvero ogni elemento di S compare una e una sola volta nell'albero.

Per ora, verranno considerati i BST sui sottoinsiemi $S \subseteq \mathbb{Z}$ dei numeri interi, basati sull'ordine totale $(\mathbb{Z}, \leq)$, dove $\leq$ è l'usuale relazione "minore o uguale" sugli interi. Un esempio di BST per il sottoinsieme $\{7, 11, 13, 9, 5\} \subseteq \mathbb{Z}$ è il seguente:



(qui E rappresenta un albero binario vuoto, il caso base della definizione ricorsiva).

3 Implementazione

Una volta data la definizione di BST, la si può usare come base per fornire un'implementazione concreta del tipo insieme di interi.

Per prima cosa, bisogna determinare come rappresentare in Scala gli alberi binari. Siccome ci sono due possibili tipi di alberi, quello vuoto e quello che consiste di un nodo (etichettato da un valore intero, in questo caso) e due sottoalberi, si definiscono due sottoclassi concrete di `IntSet`. Per la definizione di una sottoclasse si usa la parola riservata `extends`, come in Java:

```
class Empty extends IntSet {
  // ...
}

class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  // ...
}
```

Un albero vuoto non è caratterizzato da alcuna informazione, dunque la classe `Empty` che lo rappresenta non ha parametri. Invece, la classe `NonEmpty` che rappresenta un albero non vuoto ha come parametri l'etichetta `elem` del nodo e i due sottoalberi `left` e `right`. Si può osservare che esiste una corrispondenza molto precisa tra la definizione di queste classi e la definizione formale ricorsiva degli alberi binarii.

Quello realizzato fino a questo punto è solo un albero binario. Per renderlo un BST bisogna ora implementare le operazioni (i metodi astratti di `IntSet`) in modo da mantenere e sfruttare le proprietà invarianti date dalla definizione dei BST.

L'implementazione dei metodi per il caso dell'albero vuoto, la classe `Empty`, è relativamente semplice:

```
class Empty extends IntSet {
  def contains(x: Int): Boolean = false

  def add(x: Int): IntSet = new NonEmpty(x, new Empty(), new Empty())
}
```

- Un albero vuoto rappresenta appunto un insieme vuoto, che non contiene alcun elemento, dunque `contains` restituisce sempre `false`.
- L'aggiunta di un elemento `x` a un albero vuoto corrisponde a creare un nuovo albero costituito da un solo nodo, che è etichettato da `x` e ha entrambi i sottoalberi vuoti.

Il caso più interessante è invece `NonEmpty`, nel quale si evidenziano la struttura ricorsiva e le proprietà invarianti dei BST:[2]

```
class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true

  def add(x: Int): IntSet =
    if (x < elem) new NonEmpty(elem, left add x, right)
    else if (x > elem) new NonEmpty(elem, left, right add x)
    else this
}
```

- Per cercare un elemento `x` nell'albero, il metodo `contains` confronta `x` con l'etichetta del nodo:
  - se `x` è minore dell'etichetta la ricerca prosegue ricorsivamente nel sottoalbero sinistro, che per le proprietà invarianti dei BST è l'unico in cui si potrebbe trovare `x`;
  - analogamente, se `x` è maggiore dell'etichetta la ricerca prosegue nel sottoalbero destro;
  - se invece `x` non è né minore né maggiore dell'etichetta, allora può solo essere uguale, dunque `contains` restituisce `true` perché l'elemento cercato è stato trovato.

  Quando `x` non è presente nell'albero, prima o poi la ricerca ricorsiva giunge a un sottoalbero `Empty`, il cui metodo `contains` restituisce `false`.
- `add` funziona in modo simile a `contains`, facendo il confronto tra l'elemento `x` da inserire e l'etichetta del nodo:
  - se `x` è minore dell'etichetta l'inserimento prosegue ricorsivamente nel sottoalbero sinistro, e il risultato di tale inserimento costituisce il sottoalbero sinistro di un nuovo albero, che viene creato riutilizzando invece l'etichetta del nodo e il sottoalbero destro;
  - allo stesso modo, se `x` è maggiore dell'etichetta l'inserimento prosegue nel sottoalbero destro, mentre l'etichetta e il sottoalbero sinistro vengono riutilizzati.

[2] Si noti che al fine di rendere più leggibile il codice è stata usata la notazione infissa per l'invocazione (ricorsiva) dei metodi `contains` e `add`.

A un certo punto, la ricorsione raggiunge un albero `Empty`, sul quale l'operazione `add` crea un nuovo nodo, che diventa una foglia dell'albero restituito dall'inserimento. Se invece `x` è già presente nell'albero, cioè è uguale all'etichetta della radice di uno dei sottoalberi incontrati durante la ricorsione, allora l'inserimento su tale sottoalbero restituisce lo stesso sottoalbero invariato (`this`): in questo caso, `x` non viene inserito perché ogni elemento può comparire al più una volta in un insieme.

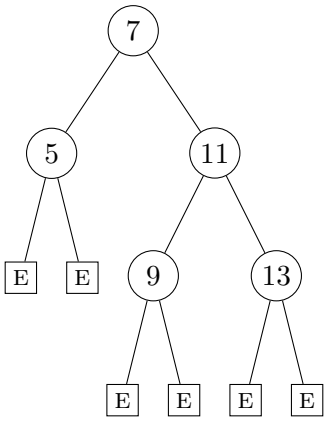
Un'osservazione di fondamentale importanza è che quest'implementazione è puramente funzionale, non ci sono mutazioni: aggiungere un elemento significa *creare un nuovo albero in cui l'elemento è aggiunto*. Si chiama **struttura dati persistente** (**persistent data structure**) una struttura dati le cui operazioni di "modifica" non alterano le strutture a cui sono applicate, ma piuttosto creano nuove strutture.

Inoltre, il nuovo albero riutilizza parte dell'albero esistente: il sottoalbero in cui l'elemento non è aggiunto. Ciò non dà problemi proprio perché non ci sono vere operazioni di modifica sull'albero, altrimenti le modifiche effettuate su una parte di un albero sarebbero visibili anche negli altri alberi che riutilizzano tale parte.

Grazie al riuso di parte dell'albero, il costo in termini di memoria dell'operazione `add` — seppur per forza maggiore rispetto a quello di una soluzione imperativa, perché si crea una nuova struttura invece di modificarne una esistente — può essere significativamente minore rispetto al costo di una copia dell'intera struttura. Comunque, è sempre necessario scegliere attentamente le strutture dati impiegate, valutando se i vantaggi dell'approccio puramente funzionale valgano il consumo aggiuntivo di memoria.

3.1 Esempio

Come esempio di inserimento che metta in evidenza il riuso dei sottoalberi, si considera un BST per il sottoinsieme $\{7, 11, 13, 9, 5\} \subseteq \mathbb{Z}$,



già mostrato in precedenza, e si aggiunge a esso l'elemento `3`.

1. L'inserimento inizia alla radice, un'istanza della classe `NonEmpty` avente `7` come valore del parametro `elem` (l'etichetta del nodo):

   ```
   new NonEmpty(7, ..., ...)
   ```

   Siccome `3 < 7`, viene valutato il corpo del primo ramo dell'`if-else` di `add`:

   ```
   if (x < elem) new NonEmpty(elem, left add x, right)
   else // ...
   ```

   Per costruire la nuova istanza di `NonEmpty` è prima necessario valutare gli argomenti del costruttore, tra cui in particolare la chiamata ricorsiva `left add x`.

2. La chiamata ricorsiva viene effettuata sull'oggetto che rappresenta il sottoalbero sinistro, la cui radice è il nodo con etichetta `5`:

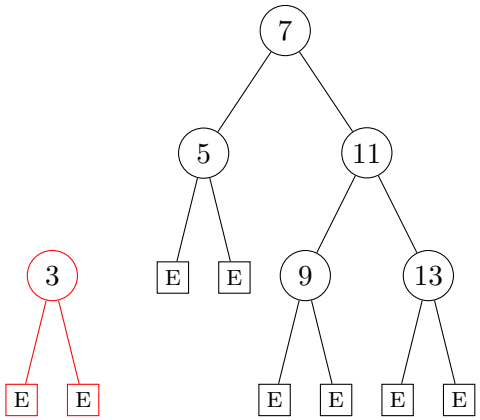
   ```
   new NonEmpty(5, new Empty(), new Empty())
   ```

   Di nuovo, si ha `3 < 5`, quindi si deve valutare la chiamata ricorsiva `left add x`.

3. Questa volta il sottoalbero sinistro è un'istanza di `Empty`, dunque viene eseguito il metodo `new Empty().add(3)`, che costituisce il caso base della ricorsione; esso restituisce una nuova istanza di `NonEmpty` con il nuovo elemento `3` come etichetta e con due sottoalberi vuoti:

   ```
   new NonEmpty(3, new Empty(), new Empty())
   ```

   Le strutture esistenti al termine di quest'ultima chiamata ricorsiva possono essere rappresentate graficamente come segue:

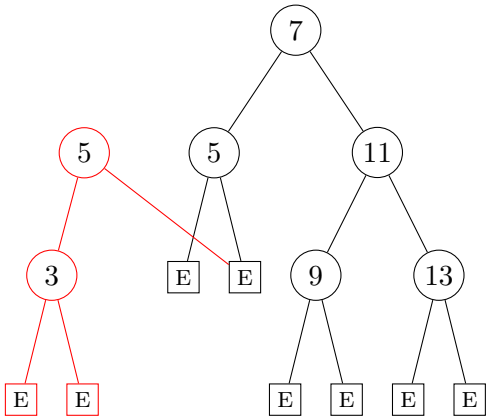
   

4. Adesso che è terminata la valutazione ricorsiva di `new Empty().add(3)` si torna al chiamante, l'invocazione del metodo `add` sul nodo etichettato `5`; qui si finisce di valutare l'espressione di creazione, che crea un nuovo nodo avente:

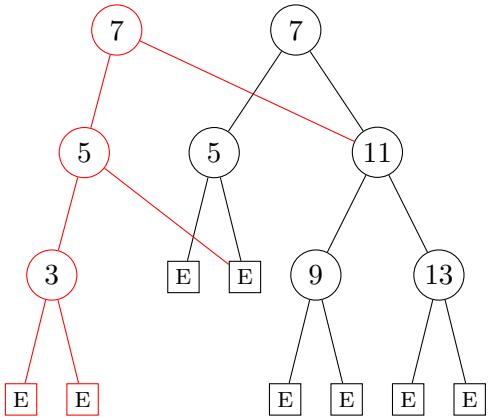
   - ancora l'etichetta `5`;
   - come sottoalbero destro la stessa istanza di `Empty` che era il sottoalbero destro del nodo `5` "originale";

- come sottoalbero sinistro il risultato di `new Empty().add(3)`, cioè il nodo `3` creato al passo precedente.

Graficamente, la situazione attuale è questa:



5. Infine si ritorna all'`add` sul nodo radice `7`. Anch'esso crea un nuovo nodo con l'etichetta e il sottoalbero destro riutilizzati dal nodo originale, e il sottoalbero sinistro risultante dall'inserimento ricorsivo:



4 Terminologia e caratteristiche dell'ereditarietà

La terminologia e il funzionamento dell'ereditarietà in Scala sono in gran parte analoghi a Java.

Considerando l'esempio dell'insieme di interi, si dice che:

- `Empty` e `NonEmpty` **estendono** la classe `IntSet`;
- `IntSet` è la **superclasse** di `Empty` e `NonEmpty`;
- `Empty` e `NonEmpty` sono **sottoclassi** di `IntSet`.

Come conseguenze dell'estensione, le sottoclassi ereditano i membri (non privati) definiti nella superclasse. Inoltre, il tipo associato a una sottoclasse *si conforma al tipo della superclasse*: un oggetto del tipo della sottoclasse può essere utilizzato ogniqualvolta sia richiesto un oggetto del tipo della superclasse. Allora, il tipo della sottoclasse è detto **sottotipo** del tipo della superclasse, e viceversa il tipo della superclasse è detto **supertipo** di quello della sottoclasse.

Ogni classe *user-defined* (definita dall'utente del linguaggio, il programmatore) estende **esplicitamente** la classe eventualmente specificata tramite `extends`, altrimenti estende **implicitamente** la classe `java.lang.Object`.

Le classi che sono superclassi **dirette** o **indirette** di una classe *C* sono chiamate **classi base** di *C*. Ad esempio, le classi base di `NonEmpty` sono `IntSet` (la superclasse diretta) e `Object` (una superclasse indiretta).