Azzolini Riccardo 2021-05-10

Raffinamenti della classe List

1 Definizione di `Nil` come `object`

Si consideri l'implementazione delle liste presentata in precedenza:

```
trait List[T] { /* ... */ }
class Cons[T](val head: T, val tail: List[T])
    extends List[T] { /* ... */ }
class Nil[T] extends List[T] { /* ... */ }
```

Al momento le liste vuote sono rappresentate da istanze della classe `Nil[T]`, ma idealmente si vorrebbe trasformare tale classe in un `object`, poiché è concettualmente corretto che tutte le liste vuote siano rappresentate dal medesimo oggetto, indipendentemente dal tipo degli elementi, dato che esse non contengono alcun elemento; rappresentarle con oggetti diversi è un'inutile ridondanza, che porta solo a uno spreco di memoria.

Un primo tentativo potrebbe essere

```
object Nil[T] extends List[T] { /* ... */ }
```

che però non viene accettato in quanto un `object` *non può dipendere da un tipo parametro*: la sua unica istanza viene automaticamente, quindi il programmatore non può specificare un tipo con cui istanziare il parametro. Allora, un `object` che estende un tipo parametrico deve istanziare i tipi parametro di quest'ultimo con dei tipi specifici. Nel caso di `Nil`, ciò significa sostanzialmente fissare il tipo di elementi della lista vuota; volendo che il valore `Nil` possa rappresentare una lista vuota per qualsiasi tipo di elementi, si sceglie il tipo `Nothing`, che è sottotipo di (compatibile con) tutti i tipi:

```
object Nil extends List[Nothing] { /* ... */ }
```

Tale scelta è coerente anche perché `Nothing` è il tipo di cui non esiste alcun valore e `Nil` è la lista che non contiene alcun elemento.

Dichiarare `Nil` come sottotipo di `List[Nothing]` non è sufficiente a rappresentare liste vuote di qualsiasi tipo: è vero che `Nothing` è compatibile con qualunque altro tipo, ma con l'attuale definizione del trait `List[T]` *non* è vero che `List[Nothing]` (il tipo di `Nil`) è compatibile con qualunque lista, perché il tipo parametro `T` è *invariante*. Ad esempio, se si prova a definire

```
val x: List[String] = Nil
```

il compilatore genera il seguente errore:

```
type mismatch;
found   : Nil.type
required: List[String]
Note: Nothing <: String (and Nil.type <: List[Nothing]),
but trait List is invariant in type T.
You may wish to define T as +T instead. (SLS 4.5)
```

Perché la cosa funzioni è necessario che `List[Nothing]` sia sottotipo di qualunque istanza di `List`, ovvero che `List` sia *covariante* nel tipo parametro `T`: bisogna scrivere `trait List[+T]` invece di `trait List[T]`.

Complessivamente, con le modifiche appena descritte, le definizioni che formano la gerarchia di classi di `List` diventano:

```
trait List[+T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}

class Cons[T](val head: T, val tail: List[T]) extends List[T] {
  def isEmpty: Boolean = false
}

object Nil extends List[Nothing] {
  override def isEmpty: Boolean = true

  override def head: Nothing =
    throw new NoSuchElementException("Nil.head")

  override def tail: Nothing =
    throw new NoSuchElementException("Nil.tail")
}
```

Un'osservazione interessante su quest'implementazione sono i tipi restituiti dai metodi `head` e `tail` di `Nil`. Siccome `Nil` estende il trait `List[T]` istanziando il parametro `T` con `Nothing`, i tipi restituiti che il trait richiede per `head` e `tail` sono rispettivamente `Nothing` (`T`) e `List[Nothing]` (`List[T]`). Per `head`, `Nothing` è il tipo più preciso possibile, dato che tale metodo solleva sempre un'eccezione. Anche `tail` solleva sempre un'eccezione, dunque il tipo più preciso è `Nothing`, che infatti è il tipo indicato nell'implementazione del metodo: ciò è consentito nonostante il tipo previsto dal trait sia `List[Nothing]` perché `Nothing <: List[Nothing]`, e in generale l'implementazione di un metodo astratto (o un override di un metodo concreto) può restituire un sottotipo del tipo previsto dal metodo astratto (o dal metodo originale che si sta ridefinendo).

2 Metodo `prepend`

Adesso si vuole aggiungere a `List` un metodo `prepend` che, dato un argomento `elem` di tipo `T`, restituisca una nuova lista ottenuta aggiungendo un nodo contenente `elem` in testa alla lista su cui il metodo è eseguito. Una prima definizione potrebbe essere:

```
trait List[+T] {
  // ...
  def prepend(elem: T): List[T] = new Cons(elem, this)
}
```

Tale definizione non è però corretta: siccome `T` è un tipo parametro *covariante*, il compilatore non permette di usarlo come argomento del metodo. Ricordando che questa regola sui tipi covarianti è stata presentata come meccanismo per evitare problemi con i metodi che modificano la struttura dati, si potrebbe pensare che in questo caso non ci siano problemi, perché `prepend` *non modifica la lista ma ne restituisce una nuova*. Invece, è corretto che il compilatore rifiuti la definizione di `prepend`, poiché essa *non soddisfa il principio di sostituzione di Liskov*.

Per verificare che `prepend` non soddisfa il principio di sostituzione di Liskov si ipotizza che la definizione del metodo sia accettata dal compilatore e si mostra che allora esistono delle operazioni eseguibili su istanze del supertipo ma non su istanze di un sottotipo. A tale scopo, si considera la solita gerarchia di `IntSet`; siccome `List[T]` è covariante in `T`, `NonEmpty <: IntSet` implica `List[NonEmpty] <: List[IntSet]`. Ora si ragiona sulle seguenti invocazioni di `prepend`:

- Se `xs: List[IntSet]` (il parametro `T` di `List` viene istanziato con `IntSet`), l'invocazione `xs.prepend(Empty)` *è corretta*: su `List[IntSet]` il metodo `prepend` richiede un argomento di tipo `T` = `IntSet`, ed `Empty` è sottotipo di (compatibile con) `IntSet`.
- Se `ys: List[NonEmpty]` (il parametro `T` viene istanziato con `NonEmpty`), l'invocazione `ys.prepend(Empty)` *non è corretta* (genera un errore di compilazione) perché `prepend` richiede un argomento di tipo `T` = `NonEmpty` ed `Empty` non è sottotipo di `NonEmpty`, quindi non può essere usato come argomento.

In sintesi, `prepend(Empty)` è un'operazione che può essere eseguita su un oggetto del supertipo (`xs: List[IntSet]`) ma non su un oggetto di un sottotipo (`ys: List[NonEmpty]`), il che invalida il principio di sostituzione di Liskov. Se si volesse che questa definizione di `prepend` sia accettata bisognerebbe rendere `List[T]` invariante in `T`, in modo che `List[NonEmpty]` non sia sottotipo di `List[IntSet]`.

Se da un lato questo ragionamento è formalmente corretto, dall'altro è intuitivamente strano e fastidioso non poter definire il metodo `prepend` sul tipo `List[+T]` covariante, considerando che `prepend` è un metodo naturale per le liste immutabili (tanto è vero che corrisponde al costruttore `Cons`). In realtà c'è un modo di definire `prepend`, perché il

problema della covarianza non è intrinsecamente legato all'operazione svolta (al contrario di quanto avviene per le operazioni di modifica di una struttura dati), bensì è dovuto al tipo che si è specificato per il metodo nella precedente definizione,

```
trait List[+T] {
  // ...
  def prepend(elem: T): List[T] = /* ... */
}
```

che obbliga il metodo a restituire una lista contenente elemento dello stesso tipo della lista "di partenza" su cui il metodo è invocato. Allora, non si può in particolare aggiungere un elemento di un supertipo di `T`, cosa che invece il principio di sostituzione di Liskov richiede che sia possibile se `T` è covariante. La soluzione è consentire appunto a `prepend` di accettare un argomento di un supertipo di `T` e restituire una lista contenente elementi di un supertipo di `T`, mediante la seguente definizione che utilizza `T` come lower bound per un altro tipo parametro `U`:

```
def prepend[U >: T](elem: U): List[U] = new Cons(elem, this)
```

Questa versione supera i controlli sulla varianza perché, oltre a quelli già detti, il compilatore ammette anche i seguenti usi per i tipi parametro covarianti o controvarianti:

- un tipo parametro *covariante* può occorrere come *lower bound* per i tipi parametro dei metodi;
- un tipo parametro *controvariante* può occorrere come *upper bound* per i tipi parametro dei metodi.

2.1 Esempi d'uso

Quando si usa quest'ultima versione (corretta) del metodo `prepend`, il compilatore deduce come tipo degli elementi della lista restituita il più piccolo supertipo comune al tipo dell'argomento di `prepend` e al tipo degli elementi della lista su cui il metodo è invocato (siccome la gerarchia dei tipi ha un'unica radice, `Any`, un supertipo comune esiste sempre, dunque `prepend` può essere usato con ogni combinazione di tipi dell'argomento e degli elementi della lista di partenza).

Ad esempio, date le definizioni

```
val set: NonEmpty = new NonEmpty(1, Empty, Empty)
val ys: List[NonEmpty] = new Cons(set, Nil)
val res = ys.prepend(Empty)
```

il tipo di `res` è `IntSet`, perché

- `prepend` viene invocato sulla lista `ys`, la quale istanzia il tipo parametro `T` di `List[T]` con `NonEmpty`,

- il parametro passato è di tipo `Empty`,

quindi il compilatore istanzia il tipo parametro `U` di `prepend` con il più piccolo tipo tale che

- `U >: T` = `NonEmpty` (per l'upper bound `U >: T` imposto nella definizione del metodo `prepend`),
- `U >: Empty` (per la compatibilità con il tipo dell'argomento),

ovvero con il più piccolo supertipo comune a `NonEmpty` e `Empty`, che è appunto `IntSet`.

Come altro esempio, considerando ancora la precedente definizione di `ys`, il tipo di

```
val res = ys.prepend("pippo")
```

è `List[AnyRef]`,[1] poiché `AnyRef` è il più piccolo supertipo comune a `NonEmpty` (il tipo degli elementi della lista `ys`) e `String` (il tipo dell'argomento).

[1] L'interprete Scala indica il tipo `List[Object]`, che è lo stesso tipo perché `AnyRef` è un alias di `Object`.