Note sul polimorfismo per inclusione (per sottotipo) e parametrico (generici)

Viviana Bono

1 Note sulla relazione di sottotipo

La relazione di *sottotipo* (detto anche *subtyping*) tra tipi corrisponde alla relazione di sottoinsieme tra insiemi (ricordiamo che i tipi sono insiemi di valori). Nei corsi di Programmazione I e II avete utilizzato questo concetto sia riferito ai tipi primitivi (per esempio, `int` è sottotipo di `double`, indicato con `int<:double`), che ai tipi oggetto. In particolare, relativamente a questi ultimi, abbiamo che ogni relazione `SC extends C` implica che il tipo indotto da `SC` (che chiamiamo `SC`, per comodità) sia sottotipo dal tipo indotto da `C` (che chiamiamo `C`, per comodità), indicato con `SC<:C`.

Alla relazione di sottotipo si affianca la regola di *sussunzione*:

$$\frac{\texttt{tipo\_1 expr} \quad \texttt{tipo\_1<:tipo\_2}}{\texttt{tipo\_2 expr}}$$

che dice: se un'espressione `expr` ha tipo `tipo_1` e `tipo_1` è sottotipo di `tipo_2`, allora `expr` ha anche tipo `tipo_2` (ovvero è polimorfa per sottotipo).

La regola di sussunzione implica il principio di *sostitutività*, ovvero un'espressione `expr` che ha tipo `tipo_1`, con `tipo_1` sottotipo di `tipo_2`, allora può essere usata anche in ogni contesto che si aspetta un'espressione di tipo `tipo_2`.

Usiamo la sostitutività, per esempio, ogni volta che scriviamo codice di questo genere:

```
int x = 5;
double y = x;
// ok, perché un int è un double con parte decimale = 0
// non sarebbe possibile il viceversa!
// (se non con cast esplicito e conseguente perdita di precisione)
```

oppure quando chiamiamo un metodo che si aspetta un `double` come parametro passandogli un `int`, o passiamo un `PuntoColorato` a un metodo che si aspetta un `Punto`, dato `PuntoColorato extends Punto`.

2 Note sui generici

2.1 Un esempio d'uso di generici

I generici sono una forma di *polimorfismo parametrico*. Nel seguito ci sono frammenti di codice da inserire in programmi di prova per poterne vedere il funzionamento.

In questo esempio si usa l'interfaccia `List` e la classe `LinkedList`, matenute per retrocompatibilità con il codice *legacy* (ovvero codice sviluppato prima di Java con i generici):

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```

Qui invece si usano l'interfaccia `List<E>` e la classe `LinkedList<E>`, ovvero le versioni con i generici:

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(0); // 2'
Integer x = myIntList.iterator().next(); // 3'
```

Si noti che i generici fanno evitare l'uso dei cast.

2.2 Qual è la semantica dei generici - hint

```
// Estratto dalle API Java

public interface List <E> {
    void add(E x);
    Iterator<E> iterator();

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Data: `myIntList = new LinkedList<Integer>();` è *come se* il compilatore creasse:

```
public interface IntegerList {
    void add(Integer x);
    Iterator<Integer> iterator();
}
```

In realtà la traduzione di una classe generica istanziata con un parametro di tipo attuale non viene tradotta così in bytecode, perché il bytecode di Java non è polimorfo. In pratica, i generici vengono *compiled away* e sostituiti con cast appositi. Infatti in Java non è possibile chiamare un metodo su una classe generica, come per esempio in `E.met()`, perché `E` nel bytecode sparirebbe e non si saprebbe con cosa sostituirla. Ci sono tuttavia linguaggi più recenti, come C#, che sono polimorfi e per i quali è stato creato un bytecode polimorfo anch'esso.

3 Generici e sottotipi

Il polimorfismo per inclusione (relazione di *sottotipo*) si può utilizzare in combinazione con i generici ed è molto utile specie con le collezioni, ma ci sono limitazioni dovute alle loro rispettive semantiche. Supponiamo di avere una libreria `Utilities.java` che contiene un metodo:

```
public static void met(List<Object> l) {...}
```

Dato che `String<:Object`, *sembrerebbe corretto* fare una chiamata del genere:

```
// main in classe Prova.java
List<String> ls = new ArrayList<String>();
Utilities.met(ls);
```

Questa chiamata però *non è corretta*, infatti il compilatore la rifiuta. Cerchiamo di capire il perché, ma, invece di usare come esempio la chiamata di un metodo, usiamo un assegnamento (perché l'applicazione del principio di sostitutività è analoga a quella per una chiamata di metodo):

```
List<String> ls = new ArrayList<String>();

List<Object> lo = ls;
// (2) --> SAREBBE LEGALE SE VALESSE:
// List<String> <: List<Object>
```

Assumiamo per assurdo che valga, allora si potrebbe fare:

```
lo.add(new Object());
// [QUI] qui uso lo (alias) IN SCRITTURA: tramite l'alias, adesso ls contiene
// anche oggetti non-String, infatti:
String s = ls.get(0);
// qui uso ls e ottengo un errore: attempts to assign an Object to a String!
```

Quindi, tramite questo controesempio, possiamo concludere che *non* vale

```
List<String> <: List<Object>
```

Infatti si può facilmente controllare che `(2)` non compila! Possiamo quindi concludere che se `FOO<:BAR` non è vero che `G<FOO><:G<BAR>`, ovvero non vale il principio di *co-varianza*. (Si può infatti vedere che l'assegnamento (2) non compila!)

Se invece facessimo il contrario, cosa capiterebbe? Vediamo:

```
List<Object> lo = new ArrayList<Object>();

List<String> ls = lo;
// (2') --> SAREBBE LEGALE SE VALESSE:
// List<Object> <: List<String>
```

Quindi si potrebbe fare:

```
ls.add(new String());
// qui uso ls (alias): tramite l'alias, adesso lo contiene
// anche oggetti non-Objects, ma...
Object s = lo.get(0);
// qui uso lo: tutto OK, leggo uno String e la assegno
// a un Object (String <: Object)!
```

Vale quindi `List<Object><:List<String>`? Sembrerebbe di sí (cioè che valga il principio di *contro-varianza*), ma un momento, se questo fosse vero allora potrei fare anche:

```
List<Object> lo = new ArrayList<Object>();

lo.add(new Object()); // qui uso lo, non un alias

List<String> ls = lo;
// (2'') --> SAREBBE LEGALE SE VALESSE:
// List<Object><:List<String>
```

Assumiamo per assurdo che lo sia, allora si potrebbe fare:

```
String s = ls.get(0);
// [QUA] qua uso ls (alias) IN LETTURA: ls è un alias a una lista di Object:
// attempts to assign an Object to a String da‘ errore!
// ottengo lo stesso errore di [QUI]: attempts to assign an Object to a String!
```

Quindi vale `List<Object><:List<String>`? No, quindi questo è un controesempio all'uso della contro-varianza. (Si può infatti vedere che l'assegnamento (2") non compila!)

La conclusione è che la co-varianza non funziona in scrittura (vedi [QUI]) e la contro-varianza in lettura (vedi [QUA]): quindi **se `FOO<:BAR`, NON è vero che `G<FOO><:G<BAR>` ma NEANCHE che `G<BAR><:G<FOO>`.** Concludendo, l'unico principio valido in generale in questo contesto è quello di *in-varianza*.

Nota. Questa sezione contiene qualche cenno di dimostrazione. La tecnica adoperata è quella della *dimostrazione per assurdo*, ovvero si parte assumendo come vera l'ipotesi da negare e da questa si costruisce un controesempio che viola altre ipotesi o fatti.

4 Tipi funzionali e sottotipi

Abbiamo visto come in Java 8 il tipo delle lambda espressioni sia solo moralmente un tipo funzionale, perché come tipo di una lambda espressione viene utilizzato un'*interfaccia funzionale*, ovvero un'interfaccia Java che contiene uno e uno solo metodo astratto (può contenere invece sia metodi statici, come già nelle versioni precedenti di Java, che metodi *di default*, introdotti essenzialmente per garantire la *backward compatibility* del codice pre-Java-8). Infatti se consideriamo per esempio la funzione identità `x -> x`, possiamo tiparla tramite un'interfaccia:

```
interface Identity<X>{
    X id(X x);
}

public class ProvaId{
  public static void main(String[] args){
    Identity<Integer> v = x -> x;
    System.out.println(v.id(3));
  }
}
```

Il metodo `id` è il *function descriptor* della lambda espressione, cioè rappresenta la sua signature, ovvero, essenzialmente, il suo tipo: prende come parametro un valore di tipo `X` e restituisce un valore dello stesso tipo. Chi ha introdotto le lambda espressioni ha preferito questa scelta, di compromesso ma omogenea rispetto alla sintassi Java, piuttosto che introdurre un nuovo costruttore di tipo. Normalmente, invece, una lambda espressione ha un tipo *funzionale*, o tipo *freccia*. In questo caso si ha: $\lambda x.x : X \to X$.

Può ora essere interessante vedere quale è la relazione di sottotipo tra tipi funzionali. Ricordiamo che il sottotipo è strettamente legato al concetto di sostitutività, quindi occorre ragionare su quando una funzione $\texttt{f}' : A \to B$ può essere usata al posto di una funzione $\texttt{f} : C \to D$ ($\texttt{f}' <: \texttt{f}$), senza causare errori di tipo. Dobbiamo quindi capire quale è la relazione tra A e C, e B e D, per avere che $A \to B <: C \to D$.

Partiamo dal codominio e assumiamo che valga $B <: D$. Per capire se questa è la relazione giusta, immaginiamo un contesto d'uso di `f`, che quindi si aspetta in uscita da f un valore $\texttt{v} : D$. Se sostituiamo `f` con $\texttt{f}'$, allora al contesto verrà restituito un valore $\texttt{v}' : B$, ma siccome abbiamo assunto $B <: D$, allora anche $\texttt{v}' : D$, quindi il contesto non osserverà mutamenti dal punto di vista dei tipi. Possiamo quindi concludere che il codominio dei tipi funzionali può variare in modo *covariante*. Per convincerci, rifacciamo **per esercizio** il ragionamento con $B =$ `int` e $D =$ `double` (ricordiamo la relazione tra i due tipi `int` <: `double`; intuitivamente, un `int` è un `double` con la parte decimale pari a zero).

Occupiamoci ora del dominio e assumiamo che valga anche per esso la co-varianza, ovvero che valga $A <: C$. Di nuovo consideriamo il contesto d'uso di `f`, il quale fornisce in input alla funzione un valore di tipo C. Se sostituiamo `f` con $\texttt{f}'$, il contesto, "cieco" rispetto al cambiamento, continuerà a fornire un valore di tipo C alla funzione $\texttt{f}'$, ma siccome abbiamo assunto che $A <: C$, $\texttt{f}'$, accettando un valore meno preciso di quello che usa ($\texttt{f}'$ si aspetta un valore di tipo A e le arriva un valore di tipo C), potrebbe generare errori a runtime. Quindi il dominio dei tipi funzionali non può variare in modo co-variante. Per convincerci, rifacciamo **per esercizio** il ragionamento con $A =$ `int` e $C =$ `double` e assumiamo che il comportamento della funzione $\texttt{f}'$ si basi sul fatto che la parte decimale del valore passato come argomento sia zero (`int` è piú preciso di `double` in questo senso). Vale invece la contro-varianza, ovvero $C <: A$ (provare **per esercizio**).

Quindi: $A \to B <: C \to D$ se e e solo se $C <: A$ (contro-varianza sul dominio) e $B <: D$ (co-varianza sul codominio).

Questo spiega perché nell'override di un metodo in Java è possibile cambiare il tipo di ritorno nella nuova versione del metodo nella sottoclasse con un tipo piú preciso (co-varianza). Per esempio, un metodo che restituisce un `Punto`, oggetto con campi `x` e `y`, può essere sostituito con una sua versione che restituisce un `PuntoColorato extends Punto`, oggetto con campi `x, y` (ereditati) e `color`, siccome ogni contesto che usava la prima versione potrà continuare a usare un `PuntoColorato` come se fosse un `Punto`. Notiamo che questo è un vero e proprio overriding, non un overloading, poiché il numero e/o il tipo dei parametri non viene modificato.

Invece, i creatori di Java hanno preferito costringere il programmatore a non cambiare i tipi dei parametri nella versione overriding (in-varianza), perché la versione co-variante sarebbe sbagliata matematicamente, e la versione contro-variante, pur essendo matematicamente corretta, potrebbe sembrare contro-intuitiva.

Aggiungiamo che, in Java, ogni cambiamento del tipo (e/o numero) dei parametri di un metodo presente nella sopraclasse e poi nella sottoclasse viene considerato *overloading* (non overriding). L'overloading non ha direttamente a che vedere con la relazione di sottotipo e le sue conseguenze, siccome introduce un metodo nuovo che ha accidentalmente il nome di un metodo già esistente (ricordiamo anche che è il compilatore a cercare di risolvere l'overloading, non il runtime).