Azzolini Riccardo 2021-05-06

Polimorfismo — Type bounds e varianza

1 Interazioni tra le forme di polimorfismo

Si è visto in precedenza che esistono due principali forme di polimorfismo: *subtyping* e *generics*. In un linguaggio che fornisce entrambe le forme, esse interagiscono. In particolare, in Scala tali interazioni riguardano due aspetti:

- i **type bounds**, cioè i vincoli che si possono porre sui tipi parametro;
- la **varianza**, che definisce le relazioni di subtyping tra tipi parametrici in base alle relazioni di subtyping tra i loro tipi parametro.

Adesso, queste interazioni verranno illustrate mediante degli esempi basati sulla gerarchia di classi `IntSet`, introdotta in precedenza:

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

2 Type bounds

Si supponga di voler definire una funzione assertiva[1] `assertAllPos` che prenda come argomento un `IntSet` e

- restituisca l'`IntSet` stesso se tutti i suoi elementi sono positivi;
- sollevi un'eccezione in caso contrario.

Intuitivamente, si potrebbe pensare di definire questa funzione con dominio e codominio `IntSet`:

```
def assertAllPos(x: IntSet): IntSet = /* ... */
```

[1] In generale, una funzione assertiva (o un metodo assertivo) è una funzione che verifica se una determinata condizione è soddisfatta, e in tal caso non fa null'altro di significativo, altrimenti solleva un'eccezione. Un esempio è `require`, che come mostrato in precedenza si usa per specificare le precondizioni.

Tale scelta è corretta, e in molte situazioni potrebbe adeguata, ma non è del tutto precisa. Infatti, il comportamento della funzione dovrebbe essere il seguente:

$$\texttt{assertAllPos(Empty)} = \texttt{Empty}$$

$$\texttt{assertAllPos(new NonEmpty(...))} = \begin{cases} \texttt{new NonEmpty(...)} & \text{se tutti positivi} \\ \texttt{throw new Exception} & \text{altrimenti} \end{cases}$$

In base a ciò, `assertAllPos` può restituire valori di diversi sottotipi di `IntSet`:

- `Empty`, se l'argomento è il valore `Empty`;
- `NonEmpty` (o `Nothing`, che è sottotipo di `NonEmpty`), se l'argomento è un'istanza di `NonEmpty`.

In sostanza, il tipo preciso del valore restituito da `assertAllPos` dipende dal tipo dell'argomento. Allora, si potrebbe provare a dare un tipo più preciso alla funzione introducendo un tipo parametro:

```
def assertAllPos[S](x: S): S = /* ... */
```

così, se la funzione viene invocata con un argomento di tipo `Empty` restituisce un valore di tipo `Empty`, mentre con un argomento di tipo `NonEmpty` restituisce un valore di tipo `NonEmpty` Adesso però il tipo della funzione è "troppo generico": il chiamante può istanziare il tipo parametro `S` in qualunque modo, ovvero può passare come argomento un valore di tipo qualsiasi, quando invece dovrebbero essere ammessi solo `IntSet` e i suoi sottotipi. La soluzione è mettere un **vincolo**, un **type bound** sul tipo parametro `S`, scrivendo:

```
def assertAllPos[S <: IntSet](x: S): S = /* ... */
```

Tale vincolo, `S <: IntSet`, indica che il tipo parametro `S` può essere istanziato solo con sottotipi di `IntSet`, compreso `IntSet` stesso (formalmente, ogni tipo è sottotipo e supertipo di se stesso) — in altre parole, `IntSet` è il più "alto" tipo della gerarchia (upper bound) con cui è ammesso istanziare `S`. Grazie a questo vincolo, dal punto di vista dei tipi la funzione `assertAllPos` si comporta finalmente come desiderato:

- Per `assertAllPos(Empty)` il compilatore deduce che `S` deve essere istanziato con il tipo `Empty`, ammesso in quanto sottotipo di `IntSet`, quindi la funzione restituisce un valore di tipo `Empty`.
- Per `assertAllPos(new NonEmpty(...))` il compilatore istanzia `S` con `NonEmpty`, un altro tipo ammesso in quanto sottitipo di `IntSet`, dunque la funzione restituisce un valore di tipo `NonEmpty`.
- Se si prova a invocare la funzione con un parametro di un tipo che non è sottotipo di `IntSet`, si ha un errore in compilazione: ad esempio, per `assertAllPos(true)` il compilatore prova a istanziare `S` con `Boolean` e genera un errore perché non è vero che `Boolean <: IntSet`.

La sintassi generale dei type bounds è la seguente:

- *S* `<:` *T* (un **upper bound**) significa che *S* deve essere sottotipo di *T*;
- *S* `>:` *T* (un **lower bound**) significa che *S* deve essere supertipo di *T* (o, equivalentemente, che *T* deve essere sottotipo di *S*);
- *S* `>:` *T1* `<:` *T2* significa che *S* deve essere supertipo di *T1* e sottotipo di *T2*, ovvero deve essere compreso nella parte di gerarchia tra *T1* e *T2*.

In tutti questi casi gli estremi sono inclusi (*S* può essere istanziato con *T* o *T1* e *T2*).

Alcuni esempi di type bounds sulla gerarchia di `IntSet` sono:

- `S >: NonEmpty`: `S` può variare su tutti i supertipi di `NonEmpty`, cioè può essere istanziato con `NonEmpty`, `IntSet`, `AnyRef` o `Any`.
- `S <: NonEmpty`: `S` può variare su tutti i sottotipi di `NonEmpty`, che sono `NonEmpty`, `Null` e `Nothing`.
- `S >: NonEmpty <: IntSet`: `S` può variare su tutti i supertipi di `NonEmpty` che sono anche sottotipi di `IntSet`, cioè solo `NonEmpty` e `IntSet` (poiché non ci sono tipi "in mezzo" tra `NonEmpty` e `IntSet` nella gerarchia).
- `S >: IntSet <: NonEmpty` genera un errore di compilazione, perché è un vincolo impossibile da soddisfare: esso indica che `S` dovrebbe poter variare sui supertipi di `IntSet` che sono anche sottotipi di `NonEmpty`, ma `IntSet` è supertipo di `NonEmpty`, quindi ogni supertipo di `IntSet` è anche supertipo (non sottotipo) di `NonEmpty` (non soddisfa la seconda parte del vincolo) e ogni sottotipo di `NonEmpty` è anche sottotipo (non supertipo) di `IntSet` (non soddisfa la prima parte del vincolo).
- `S >: IntSet <: String` è un altro vincolo che genera un errore di compilazione, perché `IntSet` è `String` appartengono a due diversi "rami" della gerarchia, quindi non esiste alcun tipo compreso tra questi due nella gerarchia.

3 Covarianza

La seconda interazione tra tipi generici e subtyping è data dalla varianza. Essa fornisce le risposte a domande come la seguente: "dato che `NonEmpty <: IntSet`, la relazione `List[NonEmpty] <: List[IntSet]` è valida?" Intuitivamente la risposta sembrerebbe essere "sì": una lista di insiemi non vuoti è un caso particolare di una lista di insiemi. I tipi parametrici per cui vale questo tipo di relazione si dicono **covarianti**, che significa "variano allo stesso modo", in quanto una relazione di sottotipo tra i tipi parametro implica una relazione di sottotipo tra i tipi parametrici: si dice che *la relazione di subtyping tra i tipi parametrici è indotta da quella esistente tra i tipi parametro.*

È importante notare che la covarianza *non* ha senso per tutti i tipi parametrici. Ciò può essere illustrato considerando il caso degli array in Java.

3.1 Covarianza degli array in Java

In Java gli array sono covarianti.[2] Ad esempio, se si riportasse in Java la gerarchia delle classi `IntSet`, `NonEmpty` e `Empty` precedentemente definite in Scala, allora `NonEmpty[]` e `Empty[]` sarebbero sottotipi di `IntSet[]`; utilizzando la notazione di Scala, si avrebbe che `NonEmpty[] <: IntSet[]` e `Empty[] <: IntSet[]`.

Ora si considerino le seguenti istruzioni:

```
1 NonEmpty[] a = new NonEmpty[] { new NonEmpty(/* ... */) };
2 IntSet[] b = a;
3 b[0] = new Empty();
4 NonEmpty s = a[0];
```

Dal punto di vista del compilatore, queste istruzioni sono corrette:

1. Alla riga 1 si crea un array `NonEmpty[]` con un elemento di tipo `NonEmpty`.
2. Alla riga 2 il riferimento all'array `NonEmpty[]` viene assegnato a una variabile di tipo `IntSet[]`, il che è consentito in quanto `NonEmpty[] <: IntSet[]`, cioè per la covarianza degli array.
3. Alla riga 3 si modifica il singolo elemento dell'array, sostituendo l'elemento di tipo `NonEmpty` con uno di tipo `Empty`. Ciò è consentito dal sistema dei tipi perché la modifica avviene tramite una variabile di tipo `IntSet[]`, dunque l'elemento inserito può essere di tipo `IntSet` o di un qualunque suo sottotipo, e appunto `Empty <: IntSet`.
4. Alla riga 4 l'elemento dell'array viene assegnato a una variabile `s` di tipo `NonEmpty`. Ciò è ammesso dal sistema dei tipi perché l'accesso all'array avviene tramite la variabile `a` di tipo `NonEmpty[]`, dunque è previsto che l'elemento estratto sia di tipo `NonEmpty`.

Tuttavia, se queste istruzioni fossero ammesse anche in fase di esecuzione provocherebbero dei seri problemi:

- L'array è costruito per memorizzare oggetti di tipo `NonEmpty`, ma alla riga 3 si inserisce un elemento di tipo `Empty`, che non è sottotipo di `NonEmpty`, dunque l'array non dovrebbe poter contenere tale elemento.
- Alla riga 4 l'elemento dell'array, che ora è di tipo `Empty`, viene assegnato a una variabile di tipo `Empty`, che non dovrebbe poter far riferimento a oggetti di tipo `Empty` in quanto questo non è sottotipo di `NonEmpty`.

[2] Gli array in Java non sono propriamente tipi generici, ma sono di fatto analoghi perché è possibile specificare arbitrariamente il tipo degli elementi, che può dunque essere assimilato a un tipo parametro, quindi ha senso parlare di covarianza sugli array.

Per evitare ciò, in fase di esecuzione Java mantiene l'informazione sui tipi base (ovvero i tipi degli elementi) degli array, e la usa per verificare la correttezza delle operazioni sugli array. In particolare, un tentativo di inserire un elemento di tipo non compatibile con il tipo base dà luogo a una `ArrayStoreException`.

Trattare come covarianti gli array e controllare a runtime la correttezza delle operazioni su di essi non è coerente con i principi di progettazione di un buon sistema dei tipi, poiché lo scopo di un sistema dei tipi è proprio quello di evitare errori di tipo in fase di esecuzione. Ci si potrebbe allora chiedere come mai i progettisti di Java abbiano preso questa decisione. La risposta è che prima dell'introduzione dei tipi generici, avvenuta con Java 5 (nel 2004, 9 anni dopo la nascita del linguaggio), l'unico strumento per scrivere codice in qualche modo generico era il subtyping. Allora, la covarianza sugli array era un compromesso che, se da un lato introduceva la possibilità di errori di tipo in esecuzione, dall'altro permetteva di definire metodi su `Object[]` e usarli su array di qualsiasi tipo. Un esempio è il metodo

```
static void sort(Object[] a)
```

definito in `java.util.Arrays`, che potrebbe essere invocato con un argomento di tipo `IntSet[]` in quanto `IntSet[] <: Object[]`. Con l'introduzione dei tipi generici la covarianza sugli array non sarebbe più necessaria, poiché i metodi su `Object[]` possono essere riscritti come metodi parametrizzati sul tipo degli elementi; ad esempio, nel caso di `sort`:

```
static <T> void sort(T[] a, Comparator<? super T> c)
```

Eliminare la covarianza avrebbe però introdotto un problema di retrocompatibilità, dunque non è stato possibile farlo.

3.2 Principio di sostituzione di Liskov

Il principio guida per determinare se un tipo generico sia covariante o meno è il **principio di sostituzione di Liskov**, il quale afferma, in termini informali, che:

> Se *`A <: B`*, tutto ciò che è possibile fare con un valore di tipo *`B`* deve essere possibile anche per ogni valore del tipo *`A`*.

Più formalmente:

> Sia $P(x)$ una proprietà dimostrabile per tutti gli oggetti x di tipo *`B`*. Se *`A <: B`* allora $P(y)$ deve essere dimostrabile per tutti gli oggetti y di tipo *`A`*.

Nel caso degli array questo principio non è soddisfatto, perché ad esempio esistono operazioni che possono essere fatte su un oggetto di tipo `IntSet[]` ma non su un oggetto di tipo `NonEmpty[]`, come l'inserimento di un elemento di tipo `Empty`. Di conseguenza, da un punto di vista teorico non dovrebbe essere vero che `NonEmpty[] <: IntSet[]`, cioè gli array non dovrebbero essere covarianti. Scala rispetta questa regola: gli array sono

rappresentati da una classe generica `Array[T]`[3] che non è covariante, perciò se riscrivesse in Scala l'esempio problematico visto in Java si otterrebbe un errore in compilazione, e non in esecuzione.

Indicativamente, i tipi *mutable* (quelli in cui è possibile modificare gli elementi) non dovrebbero essere covarianti, mentre quelli *immutable* possono essere covarianti a patto che siano verificate certe condizioni sui loro metodi.

4 Tipi di varianza

Sia `C[T]` un tipo parametrizzato, e siano `A` e `B` due tipi qualsiasi. In generale, ci sono tre possibili relazioni tra `C[A]` e `C[B]`:

- se `A <: B` implica `C[A] <: C[B]` si dice che `C` è **covariante** in `T`;
- se `A <: B` implica `C[A] >: C[B]` (si "inverte" la relazione di sottotipo) si dice che `C` è **controvariante** in `T`;
- se nessuna delle due precedenti vale, cioè se a prescindere dalla relazione tra `A` e `B` non esiste alcuna relazione di sottotipo/supertipo tra `C[A]` e `C[B]`, si dice che `C` **non è variante** in `T`, o che è **invariante** in `T`.

Si noti che la varianza riguarda un singolo tipo parametro (formale) di un tipo parametrico, cioè un tipo parametrico con più tipi parametro può avere varianza diversa su ciascun tipo parametro. Ad esempio, un tipo `C[T,U,V]` potrebbe essere covariante in `T`, controvariante in `U` e invariante in `V`, il che significa che:

- `T <: T'` implica `C[T,U,V] <: C[T',U,V]`;
- `U <: U'` implica `C[T,U,V] >: C[T,U',V]`;
- dati due tipi `V` e `V'` diversi non vale né `C[T,U,V] <: C[T,U,V']` né `C[T,U,V] >: C[T,U,V']`;
- complessivamente, `C[T,U,V] <: C[T',U',V']` se e solo se `T <: T'`, `U >: U'` e $V = V'$.

Al momento della dichiarazione di un tipo parametrico, Scala permette di indicare i tipi di varianza dei tipi parametro mediante l'annotazione dei tipi parametro stessi:

- `class C[+T]`[4] indica che `C` è covariante in `T`;
- `class C[-T]` indica che `C` è controvariante in `T`;
- `class C[T]` indica che `C` è invariante in `T`.

[3] Gli array rientrano nella parte imperativa del linguaggio, dunque non verranno presentati in questo corso.

[4] Qui la sintassi è mostrata nel caso della dichiarazione di una classe, ma la si può impiegare anche per i trait.

5 Varianza dei tipi funzionali

Un caso interessante di varianza sono i tipi funzionali. Ad esempio, si considerino due tipi funzionali

```
type A = IntSet => NonEmpty
type B = NonEmpty => IntSet
```

(questa è la sintassi che si usa in Scala per definire un *alias* di un tipo, cioè associare un tipo potenzialmente complesso a un nome più comodo). Secondo il principio di sostituzione di Liskov ha senso affermare che `A <: B`, perché l'operazione che può essere fatta con una funzione di tipo `B` è applicarla a un argomento di tipo `NonEmpty` per ottenere un risultato di tipo `IntSet`, e la stessa cosa può essere fatta con una funzione di tipo `A`, la quale infatti:

- può essere applicata a un argomento di tipo `NonEmpty` perché `NonEmpty` è sottotipo del tipo `IntSet` previsto per gli argomenti di `A`;
- restituisce un risultato di tipo `NonEmpty`, che è sottotipo di `IntSet` e quindi coerente con il tipo restituito da `B`.

Ciò significa che una funzione di tipo `A` può essere vista come un caso particolare di funzione di tipo `B`, ovvero che appunto `A <: B`. L'osservazione importante è che nel passaggio dal tipo di funzione "più particolare" a quello più generale si restringe il dominio, il tipo dell'argomento, e si allarga il codominio, il tipo del risultato.

In generale, per i tipi funzionali vale la seguente regola (che qui è presentata, per semplicità, nel caso di funzioni con un solo argomento, ma per qualunque numero di argomenti): se *A1* `>:` *A2* e *B1* `<:` *B2* allora `(`*A1* `=>` *B1*`) <: (`*A2* `=>` *B2*`)`. Detto a parole, le funzioni sono *controvarianti* nei tipi degli (uno o più) argomenti e *covarianti* nel tipo del risultato. In Scala questo fatto è codificato mediante le corrispondenti annotazioni di varianza sui tipi parametro dei trait `scala.Function`*N*; ad esempio, nel caso del trait `Function1`:

```
trait Function1[-T, +U] {
  def apply(x: T): U
}
```

6 Regole sulla varianza

Quando si definisce un tipo parametrico, per evitare errori in fase di esecuzione il compilatore deve in qualche modo verificare che la varianza dichiarata per sui tipi parametro sia corretta, cioè che non possa portare a situazioni problematiche. In particolare, esso cerca di imporre il rispetto del principio di sostituzione di Liskov tramite il controllo di alcune regole puramente sintattiche su come i tipi parametro possono essere usati nel codice del tipo parametrico.

Ad esempio, se si provasse a definire una classe `Array` covariante,

```
class Array[+T] {
  def update(i: Int, x: T) = /* ... */
}
```

il compilatore la rifiuterebbe perché al suo interno il tipo parametro covariante `T` viene usato come argomento di un metodo (qui `update`), il che può portare a situazioni problematiche.[5]

Le regole che il compilatore verifica sono in prima approssimazione le seguenti (le vere regole applicate sono in realtà più complicate):

- un tipo parametro *covariante* può occorrere solo come tipo del *risultato* di un metodo;
- un tipo parametro *covariante* può occorrere solo come tipo di un *argomento* di un metodo;
- un tipo parametro *invariante* può occorrere in *qualunque posizione*.

Considerando ad esempio il trait `Function1`,

```
trait Function1[-T, +U] {
  def apply(x: T): U
}
```

esso rispetta tali regole perché:

- il tipo controvariante `T` compare solo come argomento del metodo `apply`;
- il tipo covariante `U` compare solo come risultato del metodo `apply`.

[5] Nel caso specifico degli array l'uso del tipo covariante `T` come parametro di `update` porta ai problemi visti prima con l'esempio in Java, ma in generale, anche per tipi immutable, usando un tipo covariante come argomento di un metodo si rischiano situazioni in cui il metodo viene invocato passando un argomento di tipo sbagliato.