

prog_func

- substitution model
 - teorema del λ -calcolo
 - call-by-name (CBN)
 - call-by-value (CBV)
- operators
 - lists
- Higher order functions (HOF)
- Hierarchy
- tips and tricks

substitution model

it is based on λ -calculus. Scala normalmente utilizza la strategia di valutazione call-by-value, ma se il tipo dell'argomento di un parametro e' preceduto da `=>` applica al parametro la strategia di valutazione call-by-name.

teorema del λ -calcolo

Se la valutazione call-by-value di un'espressione termina, allora anche la valutazione call-by-name dell'espressione termina e le valutazioni producono il medesimo valore.

Il viceversa non vale.

call-by-name (CBN)

La valutazione degli argomenti viene posticipata, il rewriting dell'applicazione della funzione viene applicato senza ridurre gli argomenti.

call-by-value (CBV)

Gli argomenti della funzione vengono valutati prima di effettuare il rewriting dell'applicazione di funzione.

operators

lists

- `cons => ::`

```
val list = 1 :: 2 :: 3 :: Nil
assert(list == list.head :: list.tail)
```

Higher order functions (HOF)

function that takes as an argument another function or that returns another function. currying

```
def sum(f: Int => Int)(a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)
// sum of squares between a and b
sum(x => x * x)(2,3)
// sum of cubes between a and b
sum(x => x * x * x)(2,3)
```

Hierarchy

$S <: T$ significa S è un sottotipo di T

$S >: T$ significa S è un supertipo di T (anche T è un sottotipo di S)

$[S >: T1 <: T2]$ significa S è supertipo di $T1$ e sottotipo di $T2$

Consideriamo un tipo parametrizzato $C[T]$, e siano A e B tipi per cui

$A <: B$

in generale ci sono tre possibili relazioni fra $C[A]$ e $C[B]$:

$C[A] <: C[B]$ C è *covariante* in T

$C[A] >: C[B]$ C è *controvariante* in T

Nessuna delle due precedenti vale C non è variante (è *invariante*) in T

Scala permette di dichiarare il tipo di varianza di un tipo parametro mediante l'annotazione del tipo parametro:

`class C[+T]` C è *covariante* in T

`class C[-T]` C è *controvariante* in T

`class C[T]` C non è variante in T (è *invariante*)

Il principio di sostituzione di Liskov

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:

Il principio di sostituzione di Liskov (II)

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 .

tips and tricks

- use `require` like python `assert`
- there is only one true constructor

```
class Rational(x: Int, y: Int)
  require(y > 0, "denominator must be positive")
  val num = x / gcd(abs(x), y);
  val den = y / gcd(abs(x), y);
  def this(x: Int) = this(x,1)
```

- be careful when overloading that ends with :

```
class C(val x: Int) {
  def *(that: C): C = new C(this.x * that.x)
  def /:(that: C): C = new C(this.x / that.x)
  def @:(that: C): C = new C(this.x / that.x)
  def &:(that: C): C = new C(this.x / that.x)
  override def toString: String = "(" + x.toString + ")"
}

val uno= new C(1) // C = (1)
val due = new C(2) // C = (2)
val tre = new C(3) // C = (3)
val sei = due * tre // C = (6)
val uno2 = due /: tre // C = (1) - in quanto uguale a tre./:(due)
val zero = tre @: due // C = (0) - in quanto uguale a due.@:(tre)
```

```
val zero2 = tre &: due // C = (0) - in quanto uguale a due.&:(tre)
```