

Spezielle Kapitel Sozialer Web – Technologien - Funktionen

Seminaristischer Unterricht

Prof. Dr.-Ing. Hendrik Gärtner

Gliederung

- Wiederholung:
 - Case-Klassen
 - InsertionSort (außerhalb)
 - InsertionSort (innerhalb)
- Generische Klassen
- Higher Order Functions
 - Motivation
 - Beispiele

Definition der Methode delete

In Objekt Empty:

```
def delete(elem:Int):IntList= this
```

In Objekt Cons:

```
def delete(elem:Int):IntList = {  
    if (elem==head) tail else new Cons(head, tail.delete(elem))  
}
```

Definition der Methode deleteAll

In Objekt Empty:

```
def deleteAll(elem:Int):IntList= this
```

In Objekt Cons:

```
def deleteAll(elem:Int):IntList =  
  if (elem==head) tail.deleteAll(elem)  
  else new Cons(head, tail.deleteAll(elem))
```

Aufgabe

Implementieren Sie eine Funktion, die ein Element so in eine sortierte Liste einfügt, dass das Ergebnis ebenfalls eine sortierte Liste ist:

Klasse Cons:

```
def insertS(X:Int):IntList= X match {  
  
    case _ if (head>=X) => new Cons(X, new Cons(head,tail))  
    case _ => new Cons(head, tail.insertS(X))  
}
```

Klasse Empty:

```
def insertS(X:Int):IntList= new Cons(X,this)
```

Implementierung außerhalb

```
def insertS(elem:Int):IntList= this match{  
  
  case Empty=>new Cons(elem, Empty)  
  case Cons(head,tail) if (head>=elem) =>  
    new Cons(elem,new Cons(head,tail))  
  case Cons(head,tail) =>  
    new Cons(head, tail.insertSO(elem))  
}
```

Insertion Sort-Verfahren

```
def insertionSort:IntList= this match{  
  
  case Empty => Empty  
  case Cons(head, tail) =>  
    (tail.insertionSort).insertSO(head)  
}
```

Case Classes (3/3)

Pattern Matching:

```
def prefix (list:IntList):IntList = list match {  
  case Empty => this  
  case Cons(head, tail) => new Cons(head,this.prefix(tail))  
}
```

- Überprüfung auf vorliegenden Typ der Klasse
- Automatische Zuweisung der Eigenschaften der Klasse (hier head und tail bei Cons)
- Können im Anweisungsteil des Falls wie normale Variablen (immutable) verwendet werden

Beispiele

```
expr match {  
  case Cons(head, Empty()) => head  
  case _ => -1}
```

Matcht, wenn es sich um eine einelementige Liste handelt

```
expr match {  
  case Cons(head, Cons(4,_)) => head  
  case _ => -1}
```

Matcht, wenn die Liste mindestens zwei Elemente hat und das zweite davon die vier enthält.

```
expr match {  
  case Cons(_, Cons(head,_)) => head  
  case _ => -1}
```

Matcht, immer wenn die Liste mehr als zwei Elemente hat und gibt das zweite Element zurück.

Problemstellung

- Die entworfene Klasse ist nur für den Typ `Int` verwendbar
- Schon die Verwendung von `Doubles` würde eine neue Implementierung erfordern

Lösung 1: Ersetzen des Typs `Int` durch den Typ `Object`
(Welche Nachteile ergeben sich dadurch?)

Lösung 2: Parametrisierung der Klasse mit einem Datentyp
(wie Java Generics oder STL in C++)

Generische Version einer Liste (1/3)

```
abstract class GenList[T] {  
  def isEmpty:Boolean  
  def head:T  
  def tail:GenList[T]  
  def nth(index:Int):T  
  def contains(elem:T):Boolean  
  def insert(elem:T):GenList[T]  
  def delete(elem:T):GenList[T]  
}
```

Generische Version einer Liste (2/3)

```
case class Empty[T]() extends GenList[T]{  
  def isEmpty=true  
  
  def head:T= throw new Error("head.nil")  
  
  def tail:GenList[T]= throw new Error("tail.nil")  
  
  def contains(elem:T):Boolean=false  
  
  def nth(index:Int):T= throw new Error("IndexOutOfBounds")  
  
  def insert(X:T):GenList[T]= Cons[T](X,this)  
  
  def delete(elem:T):GenList[T]= this  
  
}
```

Generische Version einer Liste (3/3)

```
case class Cons[T] (val head:T, val tail:GenList[T]) extends GenList[T]{  
  def isEmpty=false
```

```
  
  def nth(index:Int):T= index match{  
    case 0 => head  
    case i => tail.nth(i-1)  
  }  

```

```
  def contains(elem:T):Boolean= elem match{  
    case y if (y==head) => true  
    case _ => tail.contains(elem)  
  }  

```

```
  def insert(X:T):GenList[T]= new Cons[T](X,this)
```

```
  
  def delete(elem:T):GenList[T]=  
    if (elem==head) tail else new Cons[T](head, tail.delete(elem))  
}
```

Standardbibliothek: Klasse List Verwendung (1/2)

scala.collection.**immutable**.List

Erzeugung:

```
val l:List[Integer]= List(1,2,3)
```

Wichtige Operatoren:

++ - Verknüpfung von zwei Listen

:: - Verknüpfung einer Liste mit einem Element (Bsp.: 1::List(2,3,4) entspricht List(1,2,3,4)) (Cons-Operator)

Durchlaufen iterativ:

```
def contains_iter[T](l:List[T], l:T): Boolean= {  
    var z=l  
    while (z!=Nil){  
        if (z.head==l) return true  
        z=z.tail  
    }  
    return false  
}
```

Keine Operationen zur Veränderung einzelner Elemente!

Standardbibliothek: Klasse List Verwendung (2/2)

Durchlaufen rekursiv:

```
def contains[T](l:List[T], i:T): Boolean= l match {  
  case Nil => false  
  case l::xs => true  
  case x::xs => contains(xs,i)  
}
```

```
def contains[T](l:List[T], i:T): Boolean= l.isEmpty match {  
  case true => false  
  case false if i==l.head => true  
  case _ => contains2(l.tail,i)  
}
```

Aufgabe

Schreiben Sie eine Funktion, die eine Liste mit Integerwerten übergeben bekommt und als Ergebnis eine Liste mit den Verdoppelungen der Ausgangsliste zurück gibt.

```
def double(l:List[Integer]):List[Int]
```

```
def double(l:List[Integer]):List[Int]= l match{  
  case Nil => Nil  
  case x::xs => 2*x::double(xs)  
}
```

Wie sieht eine Funktion aus, die die Werte quadriert? Gibt es da ein Muster, das vereinheitlicht werden kann?

Higher Order Functions

Higher Order Functions

- Funktionen sind in Funktionalen Programmiersprachen “first class values”
- Funktionen können wie jeder andere Wert als Parameter einer Funktion übergeben werden und können auch Ergebnis einer Funktion sein

Funktionen, die Funktionen als Parameter haben oder Funktionen zurück geben werden *Higher Order Functions* genannt

Beispiel

Verdoppelung der Zahlen einer Liste:

```
def double(l:List[Integer]):List[Int]= l match{  
  case Nil => Nil  
  case x::xs => 2*x::double(xs)  
}
```

Quadrieren der Zahlen einer Liste:

```
def square(l:List[Integer]):List[Int]= l match{  
  case Nil => Nil  
  case x::xs => x*x::double(xs)  
}
```

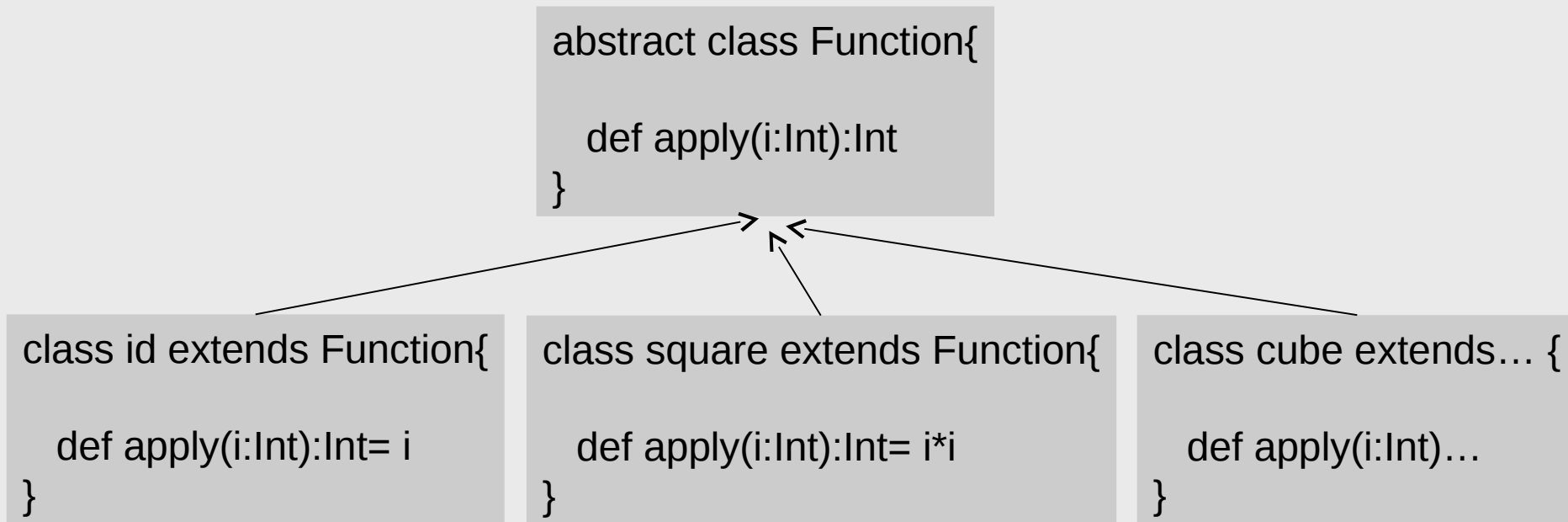
Cube bilden der Zahlen einer Liste:

```
def cube(l:List[Integer]):List[Int]= l match{  
  case Nil => Nil  
  case x::xs => x*x*x::double(xs)  
}
```

Wie kann das gemeinsame Muster genutzt werden?

Realisierung über Vererbung

- Aufteilung der Funktion in einen variablen Anteil und einen festen Anteil
 - Fest: Durchlaufen der Liste
 - Variabel: Funktion, die den aktuellen Wert quadriert, ... (Das ist eine Funktion, die ein Integer auf einen Integer abbildet)



Funktionsdefinitionen in Scala

```
abstract class Function [In,Out]{
```


```
    def apply(in:In):Out  
}
```

```
class double extends Function[Int, Int]{
```

```
    override def apply(in:Int):Int= 2*in  
}
```

```
class square extends Function[Int, Int]{
```

```
    override def apply(in:Int):Int= in*in  
}
```



Funktion f:
Int => Int

Fixer Anteil

```
def map(l:List[Int], f:Function[Int,Int]):List[Int]= l match{  
  case Nil => Nil  
  case x::xs => f.apply(x)::map(xs,f)  
}
```

Verdoppeln der Listelemente: map (l, new double)

Quadrieren der Listelemente: map (l, new square)

Schlüsselwort apply kann beim Aufruf weggelassen werden!

Übungsaufgabe

Schreiben Sie eine Funktion, die alle Elemente einer Liste aufaddiert. Verwenden Sie dazu eine Rekursion mit dem bekannten Muster.

```
def addAll(l:List[Int]):Int= l match{  
  case Nil => 0  
  case x::xs=> x+addAll(xs)  
  
}
```

Wie muss die Funktion geändert werden, wenn die Elemente miteinander multipliziert werden?

```
def multAll(l:List[Int]):Int= l match{  
  case Nil => 1  
  case x::xs=> x*multAll(xs)  
  
}
```

Wie kann das Muster für eine Funktion genutzt werden? Wie ist die Signatur der generischen Funktion?

Reduce-Funktion

```
def reduce(l:List[Int], base:Int, reduceFun:Function[(Int,Int),Int]):Int= l
match {
  case Nil => base
  case x::xs => reduceFun.apply(x,reduce(xs, base, reduceFun))
}
```

```
class add2 extends Function[(Int,Int), Int]{
  override def apply(x:(Int,Int)):Int= x._1+ x._2
}
```

```
reduce(l,0,new add2)
```


Definition der Funktion

```
abstract class Function [In,Out]{
```


```
    def apply(in:In):Out  
}
```

```
class plus extends Function[(Int,Int),Int]{
```

```
    def apply(in:(Int,Int)):Int= in._1+in._2  
}
```

```
class times extends Function[(Int,Int),Int]{
```

```
    def apply(in:(Int,Int)):Int= in._1*in._2  
}
```



Funktion f:
(Int, Int) => Int

Higher Order Functions in Scala

Map-Funktion mit Higher Order Functions

Es wird die folgende Funktion definiert:

```
def map(l:List[Int], f:Int=>Int):List[Int]= l match{  
  case Nil => Nil  
  case x::xs => f(x)::map2(xs,f)  
}
```

Damit können folgende Funktionen definiert werden:

```
def doubleFun(x:Int):Int= 2*x  
def squareFun(x:Int):Int= x*x  
def cubeFun(x:Int):Int=x*x*x
```

Wobei diese Funktionen übergeben werden:

```
map(l,doubleFun)  
map(l,squareFun)  
map(l,cubeFun)
```

Funktionstypen

- Der Typ $A \Rightarrow B$ ist einer Funktion, die ein Argument vom Typ A bekommt und ein Ergebnis vom Typ B liefert

Beispiel:

- Die Funktion `sum` bekommt als ersten Parameter eine Funktion vom Typ $\text{Int} \Rightarrow \text{Int}$ – also eine Funktion die ein Integer auf einen Integer abbildet
- Problem im Beispiel – Code wird durch die separate Funktion aufgebläht

Anonyme Funktionen

Die Übergabe von Funktionen als Parameter führt zur Erzeugung von vielen kleinen Funktionen

→ Das Definieren sollte vereinfacht werden

Vergleich mit String – dort wird keine explizite Definition benötigt:

```
def str = "abc"; println(str)
```

→ `println("abc")`

Strings existieren als *Literale*! Analog dazu sollten auch Funktionen definiert werden können - nämlich Funktionen ohne Namen also *Anonyme Funktionen*.

Syntax von Anonymen Funktionen

Beispiel: Funktion zur Berechnung eines Cubes (x^3)

$(x: \text{Int}) \Rightarrow x * x * x$

$(x: \text{Int})$ ist der Parameter der Funktion

$x*x*x$ ist die Berechnungsfunktion (der Body)

- Der Typ der Funktion kann weggelassen werden – er wird vom Compiler aus dem Kontext bestimmt
- Wenn mehr als ein Parameter in der Funktion ist, so werden diese durch ein Komma getrennt

z.B. $(x: \text{Int}, y: \text{Int}) \Rightarrow x + y$

Anonyme Funktionen sind syntaktischer Zucker

Jede Anonyme Funktion $(x_1 : T_1, \dots, x_n : T_n) \Rightarrow E$ kann ausgedrückt werden als normale Definition:

$$\text{def } f(x_1 : T_1, \dots, x_n : T_n) = E; f$$

wobei f ein name ist, der bisher noch nicht benutzt wurde.

Anonyme Funktionen bieten also keine zusätzliche Funktionalität sondern sind nur syntaktischer Zucker

Voriges Beispiel mit Anonymen Funktionen:

- `def doubleFun(l: List[Int]) = map(l, x => 2*x)`
- `def cubeFun(l: List[Int]) = map(l, x => x * x)`

Übung

Entwerfen Sie die MapReduce auf Basis der Scala Higher Order Functions.

```
def mapReduce(map:Int=>Int, reduce:(Int,Int)=>Int, zero:Int, l:List[Int]):Int=  
  case Nil => zero  
  case x::xs => reduce(map(x),mapReduce(map,reduce,zero,xs))  
}
```


Vielen Dank für

Ihre Aufmerksamkeit

Prof. Dr.-Ing. Hendrik Gärtner