Currying & Partial Application

Currying & Partielle Applikation
Motivation

- **bisher: Funktionen in Parametern erlaubt**
  geht auch in anderen Sprachen, z.B. Interfaces in Java
- **jetzt: auch Funktionen als Werte**
  erfordert andere Implementierungstechnik (Stack nicht mehr möglich)
- **Funktionswert ist wieder Funktion,**
  die auf Argumente angewendet werden kann
- **Erhält aus einer Funktion viele andere!!!!**

Currying & Partielle Applikation
Motivation

- **Wdh.:**

```
def sum(f: Int => Int, a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sum(f,a + 1, b)
```

- **Anwendung nur auf Funktion (1. Argument) liefert Signatur:**

```
def sum(f: Int => Int) : (Int, Int) => Int
```

- `def sum(f: Int => Int) (a: Int, b: Int): Int = …`
- **Implementierung: Konzepte der Nested Funktionen und Tupel (s.u.)**

Currying & Partielle Applikation
Verwende sum zur Definition neuer Funktionen

- **Definition:**

```
def sumInts : (Int,Int) => Int = sum(x => x)
def sumSquares = sum(x => x * x)
def sumPOT = sum(pot)
```

- **Aufruf:**

```
scala> sumSquares(1, 10) + sumPOT(10, 20)
unnamed0: Int = 2096513
```

- **direkter Aufruf von nested sum:**

```
sum(x => x * x)(1, 10)
```

- **direkter Aufruf von altem sum:**

```
sum(x => x * x, 1, 10)
```

Currying & Partielle Applikation
Definition

- **Definition (Currying):** aus Wikipedia

Currying (vor allem in der Linguistik auch **Schönfinkeln**) ist die Umwandlung einer Funktion mit mehreren Argumenten in eine Funktion mit einem Argument.

Obwohl das Verfahren von Moses Schönfinkel[1] erfunden und von Gottlob Frege[2] vorausgedacht wurde, wird es oft nach Haskell Brooks Curry benannt, der das Verfahren letztlich umfangreich theoretisch ausgearbeitet hat.[3]

f: Int x Int x Int -> Int => f: Int -> (Int -> (Int -> Int))

- **Definition (Partielle Applikation):**
  - Anwendung von Funktionen auf einen Teil der Argumente
  - liefert eine Funktion auf dem Rest der Argumente

Nested-Funktionen

- **Wdh.:**

```
def sum(f: Int => Int, a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sum(f,a + 1, b)
```

- **als Nested-Funktion:**

```
def sum(f: Int => Int) : (Int, Int) => Int = {
    def sumF(a: Int, b: Int): Int =
          if (a > b) 0 else f(a) + sumF(a + 1, b)
    sumF
}
```

- **Innerhalb von Block Implementierung der Hilfsfunktion** `sumF`
  - Erhält die fehlenden Parameter a und b
  - wendet äußeren Parameter f an
- **Aufruf von** `sumF`
- **Allg.: können mehrere innere Funktionen definiert werden**

Nested-Funktionen
Übung A20 als Nested-Funktion

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
    else sqrtIter(improve(guess, x), x)
  def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2
  def isGoodEnough(guess: Double, x: Double) =
    abs(square(guess) - x) < 0.001
  sqrtIter(1.0, x)    }
```

- **Überlagerung der äußeren Variablen in Parametern**
  x immer innere Variable, außer in letzter Zeile
- **Blöcke durch { } gekennzeichnet**
  - sind selbst Ausdrücke
  - Wert = Wert des Resultatsausdruck am Ende des Blocks

Tupel

- **häufig mehr als ein Rückgabewert**
- **in imperativen Sprachen dargestellt durch**
  - C: structs
  - Pascal: records
- **in OO-Sprachen dargestellt durch**
  - Klassen
- **in funktionalen Sprachen dargestellt durch**
  - Haskell: Currying `(int, int)` durch `int -> int`
  - Scala:
    - durch Case Classes für `(int, int)`

```
case class TwoInts(first: Int, second: Int)
```

    - oder durch vordefinierte Tupelschreibweise `(Int, Int)`
- **Funktion, die ganzzahlig teilt und Rest angibt**

```
def divmod(x: Int, y: Int): (Int, Int) = (x / y, x % y)
```

Currying & Partielle Applikation
Alternative Implementierungen in Scala

- **Wdh.:**

```
def sum(f: Int => Int, a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sum(f,a + 1, b)
```

- **als Nested-Funktion (s.o.)**

- **Verwendung von sum mit 2 Parameterlisten**

```
def sumCurry(f: Int => Int) (a: Int, b: Int) : Int = sum(f,a,b)
```

- **Verwendung von sum mit 1 Parameterliste und Typangabe**

```
def sumCurry(f: Int => Int): (Int,Int) => Int =
  (a,b) => sum(f,a,b)
```

- **Verwendung von sum mit 1 Parameterliste und Typangabe**

```
def sumCurry(f: Int => Int): (Int,Int) => Int = sum(f,_,_)
```

- **Verwendung eines sumCurry und Typangabe**

```
def sumPA1(f: Int => Int): (Int,Int) => Int = sumCurry(f)
```