Zusammenspiel mit parametrischer Polymorphie

- Mit GHC-Erweiterung auch sinnvolle Rolle für Constraints in Datentypdefinitionen:

```
{-# LANGUAGE ExistentialQuantification #-}

data List  =  Nil | forall a. Show a ⇒ Cons a List

list :: List
list = Cons 1 (Cons True Nil)
```

Also in Grenzen heterogene Listen, allerdings kann man mit den Elementen nichts machen, außer (hier) show auf sie anzuwenden.

Definition eigener Typklassen?

- Zunächst ein Blick auf die Definition zweier Klassen in der Standardbibliothek:

```
class Eq a where
  (==) :: a → a → Bool
  x == y  =  not (x /= y)
  (/=) :: a → a → Bool
  x /= y  =  not (x == y)

class Eq a ⇒ Ord a where
  (<), (<=), (>), (>=) :: a → a → Bool
  …
```

optionale Default-Implementierungen

- Dabei bedeutet der Eq a ⇒ Ord a <u>nicht</u>, dass jeder Eq-Typ auch ein Ord-Typ <u>ist</u>, sondern dass ein Typ nur dann zur Typklasse Ord gehören <u>kann</u>, wenn er bereits zur Typklasse Eq gehört. (Und er eben zusätzlich auch noch Operationen (<), (<=), … unterstützt, für deren Default-Implementierungen freilich die Eq-Methoden benutzt werden können.)

- Definition eigener Typklassen einfach analog (siehe Live-Beispiele).

Deskriptive Programmierung

Higher-Order

Higher-Order: Funktionen als Parameter und Resultate

- In Haskell dürfen Funktionen andere Funktionen „manipulieren“ oder „generieren“:

  - Funktionen dürfen Funktionsargumente sein.
  - Funktionen dürfen Funktionswerte sein.

- Bezeichnung für derartige Funktionen (in Anlehnung an Begrifflichkeit aus der Prädikatenlogik):

  Funktionen höherer Ordnung

- Funktionen, die „nur normale Daten“ verarbeiten und erzeugen, sind Funktionen erster Ordnung.

Curryfizierung (1)

- In Haskell werden mehrstellige Funktionen in der Regel als „mehrstufige“ Funktionale aufgefasst (und dadurch u.U. Parameterklammern gespart):

```
zwischen :: Integer → (Integer → (Integer →  Bool))
zwischen  x  y  z     | x <= y && y <= z  =  True
                      | otherwise         =  False
```

- Die Anwendung dieses Prinzips wird heute Curryfizierung genannt (nach Haskell B. Curry, der diese Technik intensiv untersucht hat; der eigentliche „Erfinder“ ist allerdings der Logiker Schönfinkel).

- Die obige Form der zwischen-Funktion wird die „currifizierte“ Form genannt, die konventionellere Form mit einem Parametertupel heißt „uncurrifiziert“.

(im Engl.: „curried“/„uncurried“)

Curryfizierung (2)

- Neben der Klammereinsparung bringt die currifizierte Notation noch den Vorteil mit sich, dass von jeder Funktion verschiedene Varianten (mit verschiedenen Stelligkeiten) automatisch mit zur Verfügung stehen.

```
zwischen :: Integer → (Integer → (Integer → Bool))
zwischen x y z  | x <= y && y <= z  = True
                | otherwise         = False
```

```
zwischen 2     :: Integer → (Integer → Bool)
zwischen 2 3   :: Integer → Bool
zwischen 2 3 4 :: Bool
```

- Jede solche partielle Applikation hat selbst alle „Rechte“ einer Funktion, darf also insbesondere selbst appliziert werden, weitergereicht werden, gespeichert werden, …

„Sections“

- Ein normalerweise zwischen seinen Argumenten geschriebener Operator kann durch Umschließung mit Klammern in eine vor die Argumente zu schreibende, currifizierte Funktion umgewandelt werden:

```
> (+) 3 4
7
```

- Eine solche „Section“ kann auch eines der Argumente in den Klammern einschließen:

```
> (/) 3 2
1.5
```

vs.

```
> (3/) 2
1.5
```

vs.

```
> (/2) 3
1.5
```

- Einige weitere Beispiele: (>3), (1+), (1/), (*2), (++ [42])

Anonyme Funktionen

- Funktionen können anonym erzeugt werden, ohne ihnen einen Namen zu geben, z.B.:

```
\x -> x + x
```

Parameter x

Funktionsrumpf

- entspricht der mathematischen Notation von „λ-Abstraktionen“, z.B.:

```
λx.(x + x)
```

- Deren Applikation wird wie normale Funktionsauswertung behandelt, z.B.:

```
> (\x → x + x) 3
6
```

Oft verwendete Higher-Order Funktionen auf Listen (1)

- Ein sehr nützliches Beispiel einer Funktion, die eine andere Funktion als Parameter akzeptiert (und sie dann auf alle Elemente einer Liste anwendet), ist die map-Funktion:

```
map f  [ ]          =   [ ]
map f  (x : xs)     =   f x : map f xs
```

Funktion als Parameter

- Zwei unterschiedliche Applikationen dieser Funktion:

```
>  map  square  [1, 2, 3]
[1, 4, 9] :: [Integer]
```

```
>  map  sqrt  [2, 3, 4]
[1.41421, 1.73205, 2.0] :: [Double]
```

- Die Funktion map ist polymorph:

```
> :t map
map ::  (a → b) →  [a] → [b]
```

Oft verwendete Higher-Order Funktionen auf Listen (2)

- Neben map gibt es noch eine Reihe weiterer wichtiger Higher-Order Funktionen für das Arbeiten mit Listen: filter, foldl, foldr, zipWith, scanl, scanr, …

- Die Funktion filter dient zum Extrahieren von Listenelementen, die eine bestimmte Boolesche Bedingung erfüllen:

```
filter  :: (a → Bool) → [a] → [a]
filter p xs  =  [ x | x ← xs, p x ]
```

„Prädikat“

```
> filter even [1, 2, 4, 5, 7, 8]
[2, 4, 8]

> filter even (filter (>3) [1, 2, 4, 5, 7, 8])
[4, 8]
```

Abkürzung für \x → x > 3
(zur Erinnerung: „Section“)

Effektiver Einsatz von Higher-Order Funktionen

- Eher Haskell-un-idiomatisch:

```
fun :: [Int] → Int
fun [ ] = 0
fun (x : xs)  | x < 20       =  5 * x – 3 + fun xs
              | otherwise    =  fun xs
```

- Besser:

```
fun :: [Int] → Int
fun  =  sum  .  map (\x → 5 * x – 3)  .  filter (< 20)
```

- Weitere für diesen Stil nützliche Funktionen: zip, splitAt, takeWhile, repeat, iterate, …

Higher-Order: etwas künstliche Beispiele

- Funktion als Parameter und Ergebnis:

```
g :: (a → a) → a → a
g  f  x  =  f  (f  x)
```

- Etwas expliziter (mit λ-Abstraktion):

```
g :: (a → a) → (a → a)
g  f  =  \x →  f  (f  x)
```

- Curryfizierung innerhalb der Sprache:

```
curry :: ((a, b) → c) → (a → b → c)
curry  f  =  \x y →  f  (x, y)
```

- Und umgekehrt:

```
uncurry :: (a → b → c) → ((a, b) → c)
uncurry  f  =  \(x, y) →  f  x y
```

Weitere Beispiele für Nutzen von Higher-Order

- Was bewirkt folgende Funktion (im Kontext von Gloss)?

```
f :: Float → [Float → Picture] → (Float → Picture)
f d fs t = pictures [ translate (i * d) 0 (a t) | (i, a) ← zip [0 ..] fs ]
```

- Und diese?

```
g :: [Float] → [Float → Picture] → (Float → Picture)
g ss fs t = pictures (map (\(s, a) → a (s * t)) (zip ss fs))
```

- Eine Aufgabe in ähnlichem Geiste auf kommendem Übungsblatt.

Weitere Beispiele für Nutzen von Higher-Order

- Erinnern wir uns an:

```
data Expr  =  Lit Int | Add Expr Expr | Mul Expr Expr

eval :: Expr → Int
eval  (Lit n)           =  n
eval  (Add e1 e2)       =  eval e1 + eval e2
eval  (Mul e1 e2)       =  eval e1 * eval e2
```

- Angenommen, wir möchten Subtraktion und Division hinzufügen.

```
...
eval  (Sub e1 e2)       =  eval e1 – eval e2
eval  (Div e1 e2)       =  eval e1 `div` eval e2
```

- Mögliches Problem: Division durch Null, daher …

Weitere Beispiele für Nutzen von Higher-Order

- Um mögliche Division durch Null abzufangen, sollten wir besser so vorgehen:

```
eval :: Expr → Maybe Int
eval  (Lit n)           =  Just n
eval  (Add e1 e2)       =  case eval e1 of
                             Nothing → Nothing
                             Just r1  → case eval e2 of
                                          Nothing → Nothing
                                          Just r2  → Just (r1 + r2)
…
```

- Aber um diese mühsamen case-Kaskaden zu vermeiden, Abstraktion der Essenz in:

```
andThen :: Maybe a → (a → Maybe b) → Maybe b
andThen m f  =  case m of  Nothing → Nothing
                           Just r   → f r
```

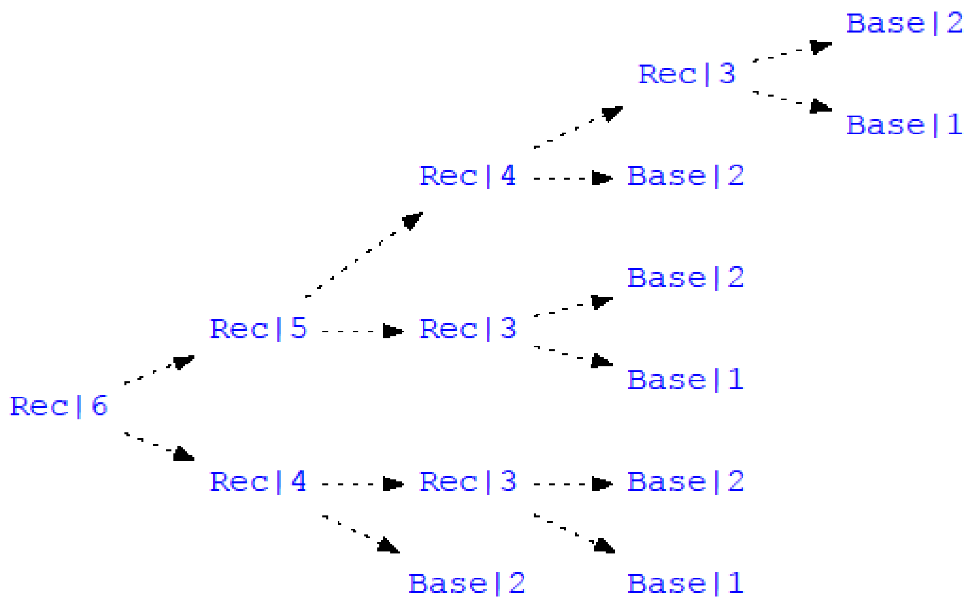
- Und dann etwa:

```
eval  (Add e1 e2)  =  eval e1 `andThen` \r1 →
                      eval e2 `andThen` \r2 → Just (r1 + r2)
```

- Betrachten wir folgendes Programm, sehr ineffizient:

```
fib :: Int → Int
fib n | n < 2 = 1
fib n         = fib (n – 2) + fib (n – 1)
```

- Die Ineffizienz liegt an folgendem „Aufrufgraph“ (für fib 6):



Higher-Order: ein etwas komplexeres Beispiel, Memoisierung (2)

- Betrachten wir folgendes Programm, sehr ineffizient:

```
fib :: Int → Int
fib n  |  n < 2  =  1
fib n            =  fib (n – 2) + fib (n – 1)
```

- Wir können Funktionsresultate „wiederverwendbar“ machen, und zwar auf sehr kanonische Weise, unabhängig von der konkreten fib-Funktion:

```
memo :: (Int → Int) → (Int → Int)
memo f  = g
          where g n    =  table !! n
                table  =  [ f n | n ←  [0 ..] ]
```

```
> let  mfib = memo fib
> mfib 30
1346269   -- nach einigen Sekunden
> mfib 30
1346269   -- „sofort“
```

Higher-Order: ein etwas komplexeres Beispiel, Memoisierung (3)

- Noch besser ist es, wenn wir Memoisierung auch innerhalb der Rekursion ausnutzen:

```
mfib = memo fib

fib :: Int → Int
fib n  |  n < 2  =  1
fib n            =  mfib (n – 2) + mfib (n – 1)
```

- Dann nämlich:

```
> fib 30
1346269   -- „sofort“
> fib 31
2178309   -- „noch sofortiger“
```

- „Aufrufgraph“ jetzt: