Vorlesung vom 22.11.2004: Funktionen Höherer Ordnung Typklassen

Inhalt

- Funktionen höherer Ordnung
  - Letzte VL: verallgemeinerte Berechnungsmuster (`map`, `filter`, ...)
  - Heute: Konstruktion neuer Funktionen aus alten
- Nützliche Techniken:
  - Anonyme Funktionen
  - Partielle Applikation
  - η-Kontraktion
- Längeres Beispiel: Erstellung eines Index
- Typklassen: Überladen von Funktionen

Funktionen als Werte

Zusammensetzen neuer Funktionen aus alten.

- Zweimal hintereinander anwenden:

```
twice :: (a-> a)-> (a-> a)
twice f x = f (f x)
```

- n-mal hintereinander anwenden:

```
iter :: Int-> (a-> a)-> a-> a
iter 0 f x = x
iter n f x | n > 0 = f (iter (n-1) f x)
           | otherwise = x
```

Funktionen als Werte

- Funktionskomposition:

```
(.) :: (b-> c) -> (a-> b)-> a-> c
(f . g) x = f (g x)
```

  - `f` nach `g`

- Funktionskomposition vorwärts:

```
(>.>) :: (a-> b)-> (b-> c)-> a-> c
(f >.> g) x = g (f x)
```

  - Nicht vordefiniert!

- Identität:

```
id :: a-> a
id x = x
```

  - Nützlicher als man denkt

Anonyme Funktionen

- Funktionen als Werte — nicht jede Funktion muß einen Namen haben.
- Beispiel (`insertsort`)

```
ins x xs = lessx ++ [x] ++ grteqx where
  (lessx, grteqx) = span less xs
  less z = z < x
```

- Besser: statt `less` anonyome Funktion

```
ins' x xs = lessx ++ [x] ++ grteqx where
  (lessx, grteqx) = span (\z-> z < x) xs
```

  - `\x-> E` ≡ `f where f x= E`
  - Auch pattern matching möglich:

```
map (\(a,b)-> a+b) [(1,4), (3,2)]
```

Beispiel: Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl `p` alle Vielfachen heraussieben
  - Dazu: filtern mit `\n-> n ‘mod‘ p /= 0`

```
sieve :: [Integer]-> [Integer]
sieve []     = []
sieve (p:ps) =
  p:(sieve (filter (\n-> n ‘mod‘ p /= 0) ps))
```

- Primzahlen im Intervall `[1..n]`

```
primes :: Integer-> [Integer]
primes n = sieve [2..n]
```

  - NB: Mit 2 anfangen!

Zeigen.

η-Kontraktion

- Beispiel: nützliche vordefinierte Funktionen:
  - Disjunktion/Konjunktion von Prädikaten über Listen

```
all, any :: (a-> Bool)-> [a]-> Bool
any p      = or  . map p
all p      = and . map p
```

- Da fehlt doch was?!

```
any p = or . map p   ≡   any p x = (or . map p) x
```

- η-Kontraktion:
  - Allgemein: `\x-> E x` $\equiv$ `E`
  - Bei Funktionsdefinition: `f x = E x` $\Leftrightarrow$ `f = E`

Partielle Applikation

- Funktionen können partiell angewandt werden:

```
double :: String-> String
double = concat . map (replicate 2)
```

  - Zur Erinnerung: `replicate :: Int-> a-> [a]`

Die Kürzungsregel bei Funktionsapplikation

Bei Anwendung der Funktion

$$\texttt{f :: t}_1 \texttt{ -> t}_2 \texttt{ -> ... -> t}_n \texttt{-> t}$$

auf k Argumente mit $k \leq n$

$$\texttt{e}_1 \texttt{ :: t}_1\texttt{, e}_2 \texttt{ :: t}_2\texttt{, ..., e}_k \texttt{ :: t}_k$$

werden die Typen der Argumente gekürzt:

$$\texttt{f :: }\not{\texttt{t}}_1 \texttt{ -> }\not{\texttt{t}}_2 \texttt{ -> ... -> }\not{\texttt{t}}_k \texttt{ -> t}_{k+1} \texttt{ -> ...-> t}_n \texttt{ -> t}$$
$$\texttt{f e}_1 \texttt{ ...e}_k \texttt{ :: t}_{k+1} \texttt{ -> ...-> t}_n \texttt{ -> t}$$

Partielle Anwendung von Operatoren

```
elem :: Int-> [Int]-> Bool
elem x = any (== x)
```

- `(== x)` Sektion des Operators == (entspricht `\e-> e == x`)
- Auch möglich: `(x ==)` (entspricht `\e-> x == e`)
- Vergleiche `(3 <)` und `(< 3)`

Gewürzte Tupel: Curry

- Unterschied zwischen

  `f :: a-> b-> c` und `f :: (a, b)-> c` ?

  - Links partielle Anwendung möglich.
  - Ansonsten äquivalent.
- Konversion:
  - Rechts nach links:

```
curry   :: ((a, b)-> c)-> a-> b-> c
curry f a b = f (a, b)
```

  - Links nach rechts:

```
uncurry :: (a-> b-> c)-> (a, b)-> c
uncurry f (a, b) = f a b
```

  - Es gilt: `curry . uncurry` $=$ `id`, `uncurry . curry` $=$ `id`.

Beispiel: Der Index

- Problem:
  - Gegeben ein Text

    ```
    brösel fasel\nbrösel brösel\nfasel brösel blubb
    ```

  - Zu erstellen ein Index: für jedes Wort Liste der Zeilen, in der es auftritt

    ```
    brösel [1, 2, 3]        blubb [3]        fasel [1, 3]
    ```

- Spezifikation der Lösung

```
type Doc = String
type Word= String
makeIndex :: Doc-> [([Int], Word)]
```

Zerlegung des Problems in einzelne Schritte

Ergebnistyp

1. Text in Zeilen aufspalten: `[Line]`
   (mit `type Line= String`)
2. Jede Zeile mit ihrer Nummer versehen: `[(Int, Line)]`
3. Zeilen in Worte spalten (Zeilennummer beibehalten):
   `[(Int, Word)]`
4. Liste alphabetisch nach Worten sortieren: `[(Int, Word)]`
5. Gleiche Worte in unerschiedlichen Zeilen zusammenfassen:
   `[([Int], Word)]`
6. Alle Worte mit weniger als vier Buchstaben entfernen:
   `[([Int], Word)]`

Erste Implementierung:

```
type Line = String
makeIndex =
  lines        >.> -- Doc -> [Line]
  numLines     >.> --     -> [(Int,Line)]
  allNumWords >.> --     -> [(Int,Word)]
  sortLs       >.> --     -> [(Int,Word)]
  makeLists    >.> --     -> [([Int],Word)]
  amalgamate   >.> --     -> [([Int],Word)]
  shorten          --     -> [([Int],Word)]
```

Implementierung von Schritt 1–2

- In Zeilen zerlegen: `lines :: String-> [String]` (vordefiniert)
- Jede Zeile mit ihrer Nummer versehen:

```
numLines :: [Line]-> [(Int, Line)]
numLines lines = zip [1.. length lines] lines
```

- Jede Zeile in Worte zerlegen:

  Pro Zeile: `words:: String-> [String]` (vordefiniert)

  - Berücksichtigt nur Leerzeichen.
  - Vorher alle Satzzeichen durch Leerzeichen ersetzen.

Implementierung von Schritt 3

- Zusammengenommen:

```
splitWords :: Line-> [Word]
splitWords = words . map (\c-> if isPunct c then ' '
                                else c) where
   isPunct :: Char-> Bool
   isPunct c = c `elem` ";:.,\'\"!?(){}-\\[]"
```

- Auf alle Zeilen anwenden, Ergebnisliste flachklopfen.

```
allNumWords :: [(Int, Line)]-> [(Int, Word)]
allNumWords = concat . map oneLine where
   oneLine :: (Int, Line)-> [(Int, Word)]
   oneLine (num, line) = map (\w-> (num, w))
                             (splitWords line)
```

Implementation von Schritt 4

- Liste alphabetisch nach Worten sortieren:
  - Ordnungsrelation definieren:

```
ordWord :: (Int, Word)-> (Int, Word)-> Bool
ordWord (n1, w1) (n2, w2) =
  w1 < w2 || (w1 == w2 && n1 <= n2)
```

  - Sortieren mit generischer Sortierfunktion `qsortBy`

```
sortLs :: [(Int, Word)]-> [(Int, Word)]
sortLs = qsortBy ordWord
```

Implementation von Schritt 5

- Gleiche Worte in unterschiedlichen Zeilen zusammenfassen:
  - Erster Schritt: Jede Zeile zu (einelementiger) Liste von Zeilen.

```
makeLists :: [(Int, Word)]-> [([Int], Word)]
makeLists = map (\ (l, w)-> ([l], w))
```

- Zweiter Schritt: Gleiche Worte zusammenfassen.
  - Nach Sortierung sind gleiche Worte hintereinander!

```
amalgamate :: [([Int], Word)]-> [([Int],Word)]
amalgamate []  = []
amalgamate [p] = [p]
amalgamate ((l1, w1):(l2, w2):rest)
  | w1 == w2  = amalgamate ((l1++ l2, w1):rest)
  | otherwise = (l1, w1):amalgamate ((l2, w2):rest)
```

Implementation von Schritt 6 — Test

- Alle Worte mit weniger als vier Buchstaben entfernen:

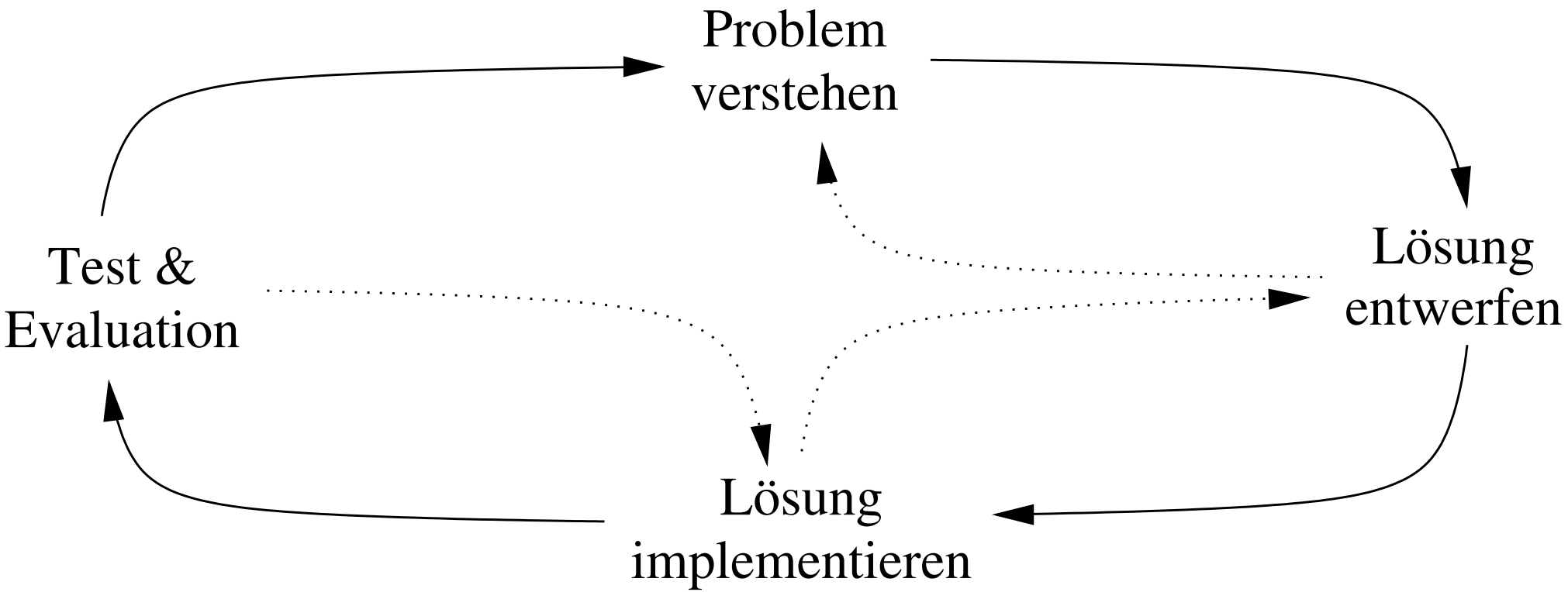
```
shorten :: [([Int],Word)] -> [([Int],Word)]
shorten = filter (\ (_, wd)-> length wd >= 4)
```

  - Alternative Definition:

```
shorten = filter ((>= 4) . length . snd)
```

- Testen.

Der Programmentwicklungszyklus



Typklassen

- Allgemeinerer Typ für `elem`:

  ```
  elem :: a-> [a]-> Bool
  ```

  zu allgemein wegen `c ==`

  - `(==)` kann nicht für alle Typen definiert werden:
  - Gleichheit auf Funktionen (z.B. `(==) :: (Int-> Int)-> (Int-> Int)-> Bool`) nicht entscheidbar.

- Lösung: `Typklassen`

  ```
  elem :: Eq a=> a-> [a]-> Bool
  elem c = any (c ==)
  ```

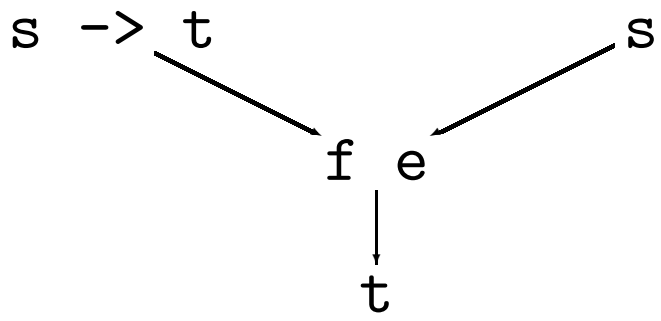
  - Für `a` kann jeder Typ eingesetzt werden, für den `(==)` definiert ist.
  - `Eq a` ist eine Klasseneinschränkung (*class constraint*)

Standard-Typklassen

- `Eq a` für `== :: a-> a-> Bool` (Gleichheit)
- `Ord a` für `<= :: a-> a-> Bool` (Ordnung)
  - Alle Basisdatentypen
  - Listen, Tupel
  - Nicht für Funktionen
- `Show a` für `show :: a-> String`
  - Alle Basisdatentypen
  - Listen, Tupel
  - Nicht für Funktionen
- `Read a` für `read :: String-> a`
  - Siehe `Show`

Typüberprüfung

- Ausdrücke in Haskell: Anwendung von Funktionen
- Deshalb Kern der Typüberprüfung: Funktionsanwendung



- Einfach solange Typen monomorph
  - d.h. keine freien Typvariablen
  - Was passiert bei polymorphen Ausdrücken?

Polymorphe Typüberprüfung

- Bei polymorphen Funktionen: Unifikation.
- Beispiel:

```
f(x,y)  =  (x,  ['a'        ..        y])
                          a-> a-> [a]
                  Char
                                    Char
                             [Char]
                a
                     (a, [Char])
f :: (a, Char)-> (a, [Char])
```

- Zweites Beispiel:

```
g(m, zs)  =   m    +    length    zs

                          [b]-> Int

                                    [b]

                               Int

                    Int

                         Int

g :: (Int, [b])-> Int
```

- Drittes Beispiel:

```
h = g . f
```

  - `(.) ::(y-> z)-> (x-> y)-> x-> z`
  - `g :: (Int, [b])-> Int`
  - `f :: (a, Char)-> (a, [Char])`

- Hier Unifikation von `(a, [Char])` und `(Int, [b])` zu

```
(Int, [Char]
```

- Damit

```
h :: (Int, [Char])-> Int
```

Typunifikation

- Allgemeinste Instanz zweier Typausdrücke `s` und `t`
  - Kann undefiniert sein.
- Berechnung rekursiv:
  - Wenn beides Listen, Berechnung der Instanz der Listenelememente;
  - Wenn beides Tupel, Berechnung der Instanzen der Tupel;
  - Wenn eines Typvariable, zu anderem Ausdruck instanziieren;
    - ▷ Dabei Überprüfung auf Diskjunktheit!
  - Wenn beide unterschiedlich, undefiniert;
  - Dabei Vereinigung der Typeinschränkungen
- Anschaulich: Schnittmenge der Instanzen.

Zusammenfassung

- Funktionen als Werte
- Anonyme Funktionen: `\x-> E`
- η-Kontraktion: `f x = E x` $\Rightarrow$ `f` $=$ `E`
- Partielle Applikation und Kürzungsregel
- Indexbeispiel:
  - Dekomposition in Teilfunktionen
  - Gesamtlösung durch Hintereinanderschalten der Teillösungen
- Typklassen erlauben überladene Funktionen.
- Typüberprüfung