Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 26.11.08: Funktionaler Entwurf & Standarddatentypen

Christoph Lüth

WS 08/09

Fahrplan

- Teil I: Grundlagen
  - Rekursion als Berechnungsmodell
  - Rekursive Datentypen, rekursive Funktionen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung
  - Funktionaler Entwurf, Standarddatentypen
- Teil II: Abstraktion
- Teil III: Beispiele, Anwendungen, Ausblicke

Inhalt

- Funktionaler Entwurf und Entwicklung
  - Spezifikation
  - Programmentwurf
  - Implementierung
  - Testen
- Beispiele
- Standarddatentypen: `Maybe`, Bäume

Funktionaler Entwurf und Entwicklung

1. Spezifikation:
   - Definitionsbereich (Eingabewerte)
   - Wertebereich (Ausgabewerte)
   - Anforderungen definieren
   - Anforderungen als Eigenschaften formulieren

   ⇝ Signatur
2. Programmentwurf:
   - Wie kann das Problem in Teilprobleme zerlegt werden?
   - Wie können Teillösungen zusammengesetzt werden?
   - Gibt es ein ähnliches (gelöstes) Problem?

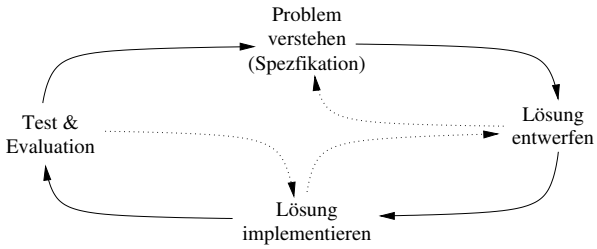
   ⇝ Erster Entwurf

Funktionaler Entwurf und Entwicklung

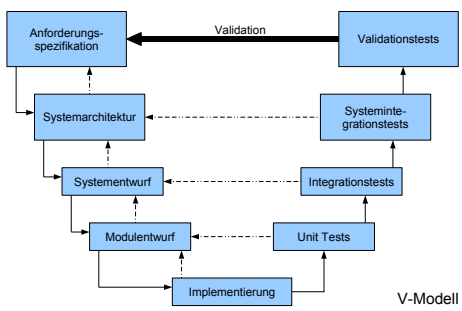
3. Implementierung:
   - Effizienz
   - Wie würde man Korrektheitheit zeigen?
   - Termination
   - Gibt es hilfreiche Büchereifunktionen
   - Refaktorierung: mögliche Verallgemeinerungen, shared code

   ⇝ Lauffähige Implementierung
4. Test:
   - Black-box Test: Testdaten aus der Spezifikation
   - White-box Test: Testdaten aus der Implementierung
   - Testdaten: hohe Abdeckung, Randfälle beachten.
   - `quickcheck`: automatische Testdatenerzeugung

Der Programmentwicklungszyklus im kleinen



Vorgehensmodelle im Großen



V-Modell

1. Beispiel: größter gemeinsame Teiler

- Definitionsbereich: `Int Int`
- Wertebereich: `Int`
- Spezifikation:
  - Teiler: $a \mid b \iff \exists n.a \cdot n = b$
  - Gemeinsamer Teiler: $\text{is_cd}(x, y, z) \iff z \mid x \wedge z \mid y$
  - Grenzen: $\gcd(x, y) \leq x, \gcd(x, y) \leq y$ damit $\gcd(x, y) \leq \min(x, y)$
  - größter gemeinsamer Teiler: $\forall i.\, \gcd(x, y) < i \leq \min(x, y) \Longrightarrow \neg \text{cd}(x, y, i)$

ggT: Spezifikation

- Signatur

```
gcd :: Int-> Int-> Int
```

- Eigenschaften (ausführbare Spezifikationen) formulieren
  - Problem: Existenzquantor — besser: $a \mid b \Longleftrightarrow b \mod a = 0$

```
divides :: Int-> Int-> Bool
divides a b = mod b a == 0
```

  - Gemeinsamer Teiler:

```
is_cd :: Int-> Int-> Int-> Bool
is_cd x y a = divides a x && divides a y
```

  - Größter gemeinsamer Teiler:

```
no_larger :: Int-> Int-> Int-> Bool
no_larger x y g = all (\i-> not (is_cd x y i)) [g .. min x y]
```

ggT: Analyse

- Reduktion auf kleineres Teilproblem: $a \mid b \Longleftrightarrow \exists n.a \cdot n = b$
- Fallunterscheidung:
  - $n = 1$ dann $a = b$
  - $n = m + 1$, dann $a(m + 1) = am + a = b$, also $am = b - a \Longleftrightarrow a \mid b - a$
- Damit Abbruchbedingung: beide Argumente gleich
- Reduktion: $a < b \rightsquigarrow \gcd(a, b) = \gcd(a, b - a)$
- Besser: $a < b \rightsquigarrow \gcd(a, b) = \gcd(b - a, a)$
- Implementierung:

```
gcd a b
  | a == b    = a
  | a < b     = gcd (b- a) a
  | otherwise = gcd b a
```

Kritik der Lösung

- Terminiert nicht bei negativen Zahlen oder 0.

```
gcd2 :: Int-> Int-> Int
gcd2 a b = gcd' (abs a) (abs b) where
  gcd' a b | a == 0 && b == 0 = error "gcd 0 0 undefined"
           | a == b || b == 0 = a
           | a < b            = gcd' (b- a) a
           | otherwise        = gcd' b a
```

- Ineffizient — es gilt auch $\gcd(a, b) = \gcd(b, a \mod b)$ (Euklid'scher Algorithmus)
- Es gibt eine Büchereifunktion.

2. Beispiel: das n-Königinnen-Problem

- Problem: n Königinnen auf $n \times n$-Schachbrett sicher plazieren
- Spezifikation:
  - Position der Königinnen

```
type Pos = (Int, Int)
```

  - Eingabe: Anzahl Königinnen, Rückgabe: Positionen

```
queens :: Int-> [[Pos]]
```

n-Königinnen: Spezifikation

- Sicher gdw. kein gegenseitiges Schlagen.
- Diagonalen: $x - y = c$, $x + y = c'$
- $(x, y) \sim (p, q) \Longleftrightarrow \quad x \neq p \wedge y \neq q \wedge x - y \neq p - q \wedge x + y \neq p + q$
- Spezifikation:
  - Alle Lösungen sind auf dem Feld,
  - alle Lösungen haben n Positionen,
  - und sind gegenseitig sicher:

$$\forall Q \in \text{queens}(n).\, \forall (x, y) \in Q.\, 1 \leq x \leq n \wedge 1 \leq y \leq n$$
$$\forall Q \in \text{queens}(n).\, |Q| = n$$
$$\forall Q \in \text{queens}(n).\, \forall p_1, p_2 \in Q.\, p_1 = p_2 \vee p_1 \sim p_2$$

n-Königinnen: Eigenschaften

- Eigenschaften (ausführbare Spezifikation):

```
inRange :: Int-> Pos-> Bool
inRange n (x, y) = 1 <= x && x <= n && 1 <= y && y <= n

enough :: Int-> [Pos]-> Bool
enough n q = length q == n

isSafe :: Pos-> Pos-> Bool
isSafe (x, y) (p, q) =
  x /= p && y /= q && x- y /= p- q && x+ y /= p+ q

allSafe :: [Pos]-> Bool
allSafe q =
  all (\p-> all (\r-> (p == r || isSafe p r)) q) q

isSolution :: Int-> [[Pos]]-> Bool
isSolution n q = all (all (inRange n)) q && all (en
```

- Schon fast eine Lösung, aber kombinatorische Explosion

n-Königinnen: Rekursive Formulierung

- Rekursive Formulierung:
  - Keine Königin— kein Problem.
  - Lösung für n Königinnen: Lösung für $n - 1$ Königinnen, n-te Königin so stellen, dass sie keine andere bedroht.
- Vereinfachung: n-te Königin muß in n-ter Spalte plaziert werden.
  - Limitiert kombinatorische Explosion

n-Königinnen: Hauptfunktion

- Hauptfunktion:
  - Sei `p` neue Zeile
  - `cand p` bisherige Teillösungen, die mit `(n, p)` sicher sind
  - `put p q` fügt neue Position `p` zu Teillösung `q` hinzu

```
queens num = putqueens num where
  putqueens :: Int-> [[Pos]]
  putqueens n =
    if n == 0 then [[]]
    else let cand p   = filter (\q-> safe q (n, p))
                               (putqueens (n-1))
             put  p q = q ++ [(n, p)]
         in  concatMap (\p-> map (put p) (cand p))
                       [1.. num]
```

  - Rekursion über Anzahl der Königinnen
  - Daher Termination

Das n-Königinnen-Problem

- Sichere neue Position: durch keine andere bedroht

```
safe :: [Pos]-> Pos-> Bool
safe others p = all (not . threatens p) others
```

- Gegenseitige Bedrohung:
  - Bedrohung wenn in gleicher Zeile, Spalte, oder Diagonale.

```
threatens :: Pos-> Pos-> Bool
threatens (i, j) (m, n) =
  (j== n) || (i+j == m+n) || (i-j == m-n)
```

  - Test auf gleicher Spalte `i==m` unnötig.

Das n-Königinnen-Problem: Testen

- Testdaten (manuell):
  - `queens 0`, `queens 1`, `queens 2`, `queens 3`, `queens 4`
- Test (automatisiert):
  - `all (\n-> is\_solution n (queens n)) [1.. 8]`

3. 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)]
```

  - Keine Leereinträge
  - Alle Wörter im Index müssen im Text in der angegebenen Zeile auftreten

Der Index: Eigenschaften

- Keine Leereinträge

```
notEmpty :: [([Int], Word)] -> Bool
notEmpty idx = all (\ (l, w)-> not (null l)) idx
```

- Alle Wörter im Index im Text in der angegebenen Zeile
  - NB. Index erster Zeile ist 1.

```
occursInLine :: Word-> Int-> Doc-> Bool
occursInLine w l txt = isInfixOf w (lines txt !! (l-1))
```

- Eigenschaften, zusammengefasst:

```
prop_notempty :: String-> Bool
prop_notempty doc = notEmpty (makeIndex doc)

prop_occurs :: String-> Bool
prop_occurs doc =
  all (\ (ls, w)-> all (\l-> occursInLine w l doc) ls)
      (makeIndex doc)
```

Zerlegung des Problems: erste Näherung

- Text in Zeilen zerteilen
- Zeilen in Wörter zerteilen
- Jedes Wort mit Zeilennummer versehen
- Gleiche Worte zusammenfassen
- Sortieren

Zerlegung des Problems: zweite Näherung

Ergebnistyp

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

Erste Implementierung:

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

Implementierung von Schritt 1–2

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

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

- Jede Zeile in Wörter zerlegen:
  - Pro Zeile `words:: String-> [String]`
  - 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 = concatMap oneLine where
   oneLine :: (Int, Line)-> [(Int, Word)]
   oneLine (num, line) = map (\w-> (num, w))
                             (splitWords line)
```

Einschub: Ordnungen

- Generische Sortierfunktion
  - Ordnung als Parameter

```
qsortBy :: (a-> a-> Bool)-> [a]-> [a]
qsortBy ord [] = []
qsortBy ord (x:xs) =
  qsortBy ord (filter (\y-> ord y x) xs) ++ [x] ++
  qsortBy ord (filter (\y-> not (ord y x)) xs)
```

- Vordefiniert (aber andere Signatur):

```
sortBy ::(a-> a-> Ordering)-> [a]-> [a]
```

Implementation von Schritt 4

- Liste alphabetisch nach Wörtern 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 Wörter 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 Wörter zusammenfassen.
  - Nach Sortierung sind gleiche Wörter 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 Wörter mit weniger als vier Buchstaben entfernen:

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

  - Alternative Definition:

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

  - Testfälle:
    - `makeIndex ""`
    - `makeIndex "a b a"`
    - `makeIndex "abcdef abcde"`
    - `makeIndex "a eins zwei\nzwei\nzwei,eins"`

Standarddatentypen

- Listen `[a]`
- Paare `(a,b)`
- Lifting `Maybe a`
- Bäume

Modellierung von Fehlern: `Maybe a`

- Typ `a` plus Fehlerelement
  - Im Prelude vordefiniert.

```
data Maybe a = Just a | Nothing
```

- `Nothing` modelliert Fehlerfall:

```
find :: (a-> Bool)-> [a]-> Maybe a
find p []     = Nothing
find p (x:xs) = if p x then Just x
                else find p xs
```

Funktionen auf `Maybe a`

- Anwendung von Funktion mit Default-Wert für Fehler (vordefiniert):

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe d f Nothing  = d
maybe d f (Just x) = f x
```

- Liften von Funktionen ohne Fehlerbehandlung:
  - Fehler bleiben erhalten.

```
fmap :: (a-> b)-> Maybe a-> Maybe b
fmap f Nothing = Nothing
fmap f (Just x)= Just (f x)
```

Binäre Bäume

Ein binärer Baum ist

- Entweder leer,
- oder ein Knoten mit genau zwei Unterbäumen.

Knoten tragen eine Markierung.

```
data Tree a = Null
            | Node (Tree a) a (Tree a)
```

- Andere Möglichkeit: Unterschiedliche Markierungen Blätter und Knoten

```
data Tree' a b = Null'
               | Leaf' b
               | Node' (Tree' a b) a (Tree' a b)
```

Funktionen auf Bäumen

- Test auf Enthaltensein:

```
member :: Eq a=> Tree a-> a-> Bool
member Null _ = False
member (Node l a r) b =
  a == b || (member l b) || (member r b)
```

- Höhe:

```
height :: Tree a-> Int
height Null = 0
height (Node l a r) = max (height l) (height r) + 1
```

Funktionen auf Bäumen

Primitive Rekursion auf Bäumen:

- Rekursionsanfang
- Rekursionsschritt:
  - Label des Knotens,
  - Zwei Rückgabewerte für linken und rechten Unterbaum.

```
foldT :: (a-> b-> b-> b)-> b-> Tree a-> b
foldT f e Null = e
foldT f e (Node l a r) = f a (foldT f e l) (foldT f e r)
```

Funktionen auf Bäumen

- Damit Elementtest:

```
member' :: Eq a=> Tree a-> a-> Bool
member' t x =
   foldT (\e b1 b2-> e == x || b1 || b2) False t
```

- Höhe:

```
height' :: Tree a-> Int
height' = foldT (\ _ h1 h2-> 1+ max h1 h2) 0
```

Funktionen auf Bäumen

- Traversion: preorder, inorder, postorder

```
preorder  :: Tree a-> [a]
inorder   :: Tree a-> [a]
postorder :: Tree a-> [a]
preorder   = foldT (\x t1 t2-> [x]++ t1++ t2) []
inorder    = foldT (\x t1 t2-> t1++ [x]++ t2) []
postorder  = foldT (\x t1 t2-> t1++ t2++ [x]) []
```

- Äquivalente Definition ohne `foldT`:

```
preorder' Null          = []
preorder' (Node l a r)  = [a] ++preorder' l ++preorder' r
```

- Wie würde man geordnete Bäume implementieren?

Zusammenfassung

- Funktionaler Entwurf:
  - Entwurf: Signatur, Eigenschaften
  - Implementierung: Zerlegung, Reduktion, Komposition
  - Testen: Testdaten, `quickcheck`
  - Ggf. wiederholen
- Standarddatentypen: `Maybe a`, Bäume
- Nächste Woche: abstrakte Datentypen