Skriptteil zur Vorlesung:
Proinformatik - Funktionale Programmierung

Dr. Marco Block-Berlitz

24.Juli 2009

Zip

Die Elemente zweier Listen lassen mit dem Reisverschluss-Prinzip elementeweise zu Tupeln zusammenfügen. Das erste mit dem ersten usw. Sollten die Elemente einer Liste erschöpft sein, so ist die neu erzeugte Tupeliste fertig.

```
zippe :: [t] -> [u] -> [(t,u)]
zippe (x:xs) (y:ys) = (x,y):zippe xs ys
zippe  _      _     = []
```

Testen wir das mit drei Beispielen, bei denen die Listenlängen gleich und unterschiedlich sind

```
Hugs> zippe "abcde" [1,2,3,4,5]
[('a',1),('b',2),('c',3),('d',4),('e',5)]

Hugs> zippe "abcdefghi" [1,2,3,4,5]
[('a',1),('b',2),('c',3),('d',4),('e',5)]

Hugs> zippe "abcde" [1,2]
[('a',1),('b',2)]
```

Unzip

Wir können den Reißverschluss auch wieder öffnen die zwei Elemente in separate Listen zerlegen:

```
unzippe :: [(a,b)] -> ([a],[b])
unzippe []          = ([], [])
unzippe ((a,b):xs) = (c,d)
   where
      c = [a] ++ fst (unzippe xs)
      d = [b] ++ snd (unzippe xs)
```

Schauen wir uns dazu ein Beispiel an:

```
Hugs> unzippe [(1,'a'),(2,'b')]
([1,2],"ab")
```

Aber Achtung, `unzippe` ist nicht die Umkehrfunktion von `zip`, da bei der Reißverschluss-Funktion `zip` Daten aus einer der beiden Listen verloren gehen können, siehe:

```
Hugs> unzippe (zippe [1,2,3,4,5] "abc")
([1,2,3],"abc")
```

Zunächst haben wir die Listen `[1,2,3,4,5]` und `['a','b','c']` zusammengefügt und erhielten `[(1,'a'),(2,'b'),(3,'c')]`. Durch die Zerlegung der Tupelliste erhalten wir aber die Listen `[1,2,3]` und `['a','b','c']`.

Funktionskompositionen

Wenn wir die Resultate einer Funktion als Eingabe in eine neue Funktion verwenden möchten, können wir das so machen

$$f(...(f(f(f(x)))...)$$

Alternativ können wir eine kürzere Schreibweise mit dem Punkt-Operator verwenden. Es gilt

$$(f.g)\,x = f(g(x)).$$

So läßt sich der Ausdruck $a(b(c(d(e(f(g(h(i(j(k(x)))))))))))$ übersichtlich verkürzen zu $(a.b.c.d.e.f.g.h.i.j.k)\,x$.

Der Operator `(.)` läßt sich in Haskell wie folgt definieren:

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

Zunächst dient `x` mit dem Datentyp `a` als Eingabe für die Funktion `g` mit `a->b` und liefert als Ergebnis einen Wert vom Typ `b`. Dieser wird in eine Funktion f mit der Signatur `b->c` gegeben und wir erhalten `c`.

Betrachten wir das Beispiel unzippe. Dort haben wir mit dem Aufruf

```
Hugs> unzippe (zippe [1,2,3,4,5] "abc")
```

ein zufriedenstellendes Ergebnis erhalten.

An dieser Stelle ist ist es interessant, über das Ergebnis der Funktionskomposition des folgenden Aufrufes einmal nachzudenken:

```
Hugs> (unzippe.zippe) [1,2,3,4,5] "abc"
```

Wir würden das gleiche Ergebnis erwarten, aber Hugs liefert eine Fehlermeldung. Das Problem dabei ist, dass die Funktionskomposition einelementige Eingaben erwartet.

Wenn wir eine Funktionskomposition `(...)` für zwei Eingaben selber definieren, z.B. so:

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

dann läßt sich auch die Funktion `unzippe` mit `zippe` kombinieren

```
Hugs> (unzippe...zippe) [1,2,3,4,5] "abc"
([1,2,3],"abc")
```

Eq, Ord, Enum, Show

Wir wollen die vier Jahreszeiten als neuen Datentyp definieren.

```
data Saison = Fruehling | Sommer | Herbst | Winter
```

Jetzt lassen sich s1 und s2 von Typ Saison definieren

```
s1, s2 :: Saison
s1     = Fruehling
s2     = Sommer
```

Ausführung

```
Hugs> s1
Hugs> ERROR - Cannot find "show" function for:
Hugs> *** Expression : s1
Hugs> *** Of type    : Saison
```

Ein Grund für den Fehler ist, dass Haskell nicht weiß, wie er den neuen Datentyp auf der Konsole ausgeben soll. Es gibt eine bereits definierte Klasse `Show`. Alle Instanzen dieser Klasse besitzen eine Funktion `show`, die eine Zeichenkette auf dem Bildschirm ausgibt. So sind beispielsweise die Klassen `Int`, `Float`, usw. ebenfalls Instanzen der Klasse `Show`.

```
Hugs> s1 == s2
Hugs> ERROR - Cannot infer instance
Hugs> *** Instance   : Eq Saison
Hugs> *** Expression : s1 == s2
```

Auch auf Gleichheit können wir nicht testen, da für Saison keine Gleichheit definiert wurde. Schauen wir uns zunächst die vier wichtigsten Klassen an und lösen anschließend die Problematik:

`Eq` für Gleichheit, Nicht-Gleichheit

`Ord` für Ordnung

`Enum` für das Aufzählen

`Show` zum Ausgeben

Wir wollen unseren neuen Datentyp `Saison` zu diesen vier Klassen hinzufügen, dazu erweitern wir die Definition um das Schlüsselwort `deriving`

```
data Saison = Fruehling | Sommer | Herbst | Winter
           deriving(Eq, Ord, Enum, Show)
```

Jetzt erhalten wir für `s1` und `s2` auf der Konsole auch eine Ausgabe und können auf Gleichheit prüfen

```
Hugs> s1
Hugs> Fruehling

Hugs> s2
Hugs> Sommer

Hugs> s1==s2
Hugs> False
```

Des Weiteren erlaubt uns die Klasse `Enum` eine Aufzählung der Parameter

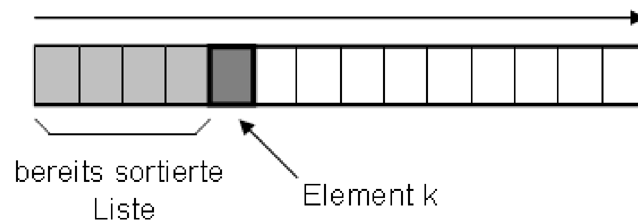
```
Hugs> [Fruehling .. Herbst]
Hugs> [Fruehling,Sommer,Herbst]
```

Für Größenrelationen ist die Klasse `Ord` zuständig. Wir können jetzt auch die Jahreszeiten der Größe nach sortieren

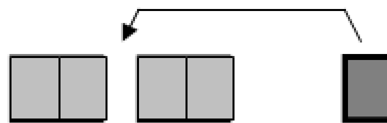
```
Hugs> Sommer > Herbst
Hugs> False
```

InsertionSort

Angenommen, die Liste liegt bereits aufsteigend sortiert vor, das Einordnen eines neuen Elements $x[k]$ ist dann einfach. Es wird solange von links beginnend überprüft, ob das aktuelle Element kleiner oder gleich dem Einzufügenden ist.



Sollte die korrekte Position i_k identifiziert worden sein, für die gilt, dass $x[i_k] > x[k]$ und $i_k < k$, so wird das Element $x[k]$ an die Position i_k eingefügt und alle Elemente von $x[i_k]$ bis $x[k-1]$ um einen Index erhöht.



Damit erhöht sich die Anzahl der bereits sortierten Elemente auf k. Genau das macht die Funktion `einfuegen`, Einfügen in eine aufsteigend sortierte Liste

```
einfuegen :: Ord a => a -> [a] -> [a]
einfuegen x [] = [x]
einfuegen x (y:ys)
          | x<=y      = x:y:ys
          | otherwise = y:einfuegen x ys
```

Die Sortierung mit InsertionSort, also Einfügen in den sortierten Rest", ist jetzt auch genau so in Haskell anzugeben

```
isort :: Ord a => [a] -> [a]
isort []     = []
isort (x:xs) = einfuegen x (isort xs)
```

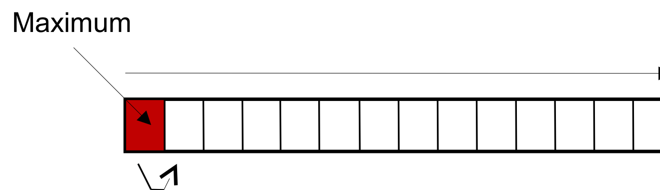
Damit ist der Algorithmus fertig.

BubbleSort

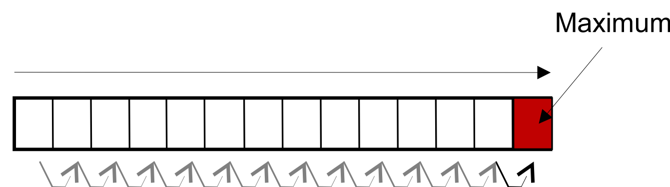
Der BubbleSort-Algorithmus in Pseudocode

```
for (i=1 to n-1)
   for (j=0 to n-i-1)
      if (x[j] > x[j+1]) vertausche x[j] und x[j+1]
```

Wenn wir uns die zweite Schleife für `i=1` anschauen, wird schnell klar, was der Algorithmus macht. Angenommen, in der vorliegenden Liste steht das Maximum der Elemente am Anfang und es gibt nur eines.



Es wird überprüft, ob das erste Element größer als das zweite ist. Da das Maximum sicherlich größer ist, als das zweite Element, werden diese beiden Elemente getauscht. Am Ende der Schleife, wenn das Element $x[n-2]$ mit $x[n-1]$ verglichen wird, landet das Maximum ganz hinten und die Schleife ist fertig.



Sollte das Maximum irgendwo in der Mitte liegen, wird es auch von dort nach hinten getragen. Die Analogie der Luftblasen, die im Wasser nach oben blubbern, verleiht dem Algorithmus seinen Namen.

```haskell
bubble :: Ord a => [a] -> [a]
bubble []       = []
bubble [x]      = [x]
bubble (x:y:xs)
       | x>y        = y : bubble (x:xs)
       | otherwise = x : bubble (y:xs)
```

Nach der Ausführung der `bubble`-Funktion auf eine Liste `xs` ist gewährleistet, dass das größte Element an der letzten Stelle steht. Das können wir uns zu Nutze machen, um eine doppelte for-Schleife in Haskell durch Rekursion über die Größe der Liste zu realisieren. Jetzt kann BubbleSort mit Hilfe der Bubble-Funktion definiert werden.

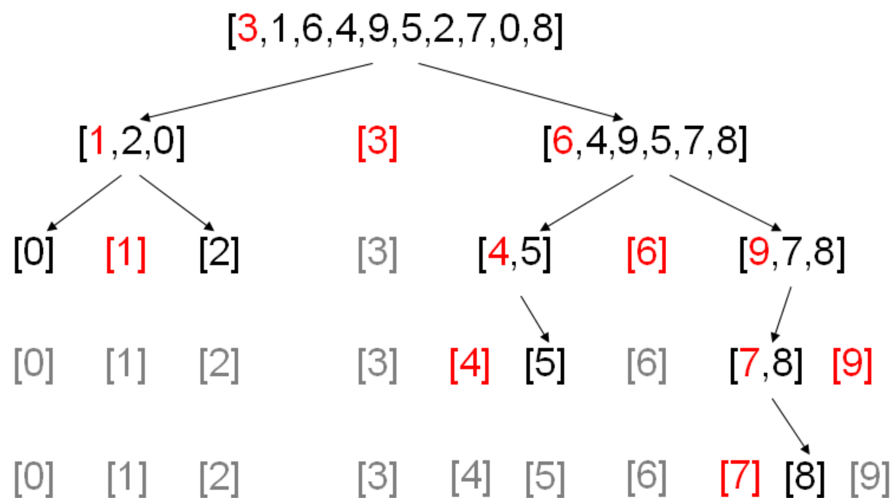
```haskell
bsort :: Ord a  => [a] -> [a]
bsort  []       = []
bsort (x:xs)    = bsort (init ys) ++ [last ys]
       where ys   = bubble (x:xs)
```

Das Prinzip, sortiere die Liste der Länge n durch die Funktion bubble, läßt das größte Element am Ende der Liste stehen. Die Position dieses Elements ist bereits richtig, also sortieren wir analog die übrige Liste ohne das letzte Element, usw.

QuickSort

Der QuickSort-Algorithmus ist ein typisches Beispiel für Algorithmen mit dem Teile-und-Herrsche-Prinzip. In jedem Schritt, beginnend mit der gesamten Liste, wird ein Element ausgewählt (Pivotelement) und die Liste zweigeteilt. In der linken Liste sind die Elemente (ohne das Pivotelement) enthalten, die die kleiner oder gleich dem Pivotelement sind und in der rechten die größeren. Das Pivotelement verbleibt dabei in der Mitte.

Schauen wir uns dazu ein Beispiel an, in dem immer das erste Element als Pixotelement ausgewählt wird



Nach der Aufspaltung und lokalen Sortierung in größere und kleiner Elemente, müssen die resultierenden einelementigen Listen nur noch zusammengefügt werden. Die Liste ist dann aufsteigend sortiert.

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort [y | y<-xs, y<=x]++[x]++
               qsort [y | y<-xs, y> x]
```

MergeSort

merge

```
merge [] ys = ys
merge xs []    = xs
merge (x:xs) (y:ys)
    | x <= y      = x:merge xs (y:ys)
    | otherwise   = y:merge (x:xs) ys
```

Mergesort

```
msort :: Ord a => [a] -> [a]
msort xs
    | length xs<2 = xs
    | otherwise   = merge (msort erst) (msort zweit)
```

```
    where
        erst       = take haelfte xs
        zweit      = drop haelfte xs
        haelfte    = div (length xs) 2
```

Modularisierung und Schnittstellen

Um erfolgreich größere Projekte zu realisieren, ist die Modularisierung ein wichtiges Werkzeug, um Programmcode übersichtlich zu gestalten und die Interaktion der Methoden untereinander transparent zu halten. Aus Gründen der Wiederverwendbarkeit, Wartung und Fehlerlokalisierung ist es ebenfalls ratsam, Programme in Teile zu zerlegen und durch diese Modularisierung das Abstraktionsniveau zu erhöhen. Auch gerade wenn mehrere Leute an einem Projekt arbeiten, können so Aufgaben in Teilbereiche zerlegt und gemeinsam bewältigt werden.

Ein wichtiges Konzept für große Softwareprojekte stellt die Definition von Schnittstellen (Interfaces) dar. Heutzutage sind keine großen Projekte ohne klare Schnittstellendefinitionen vorstellbar. Haskell bietet in diesem Zusammenhang die Definition von `modules` an.

Definition von Modulen

Ein Modul pro Datei. Als Konvention gilt, dass der Name eines Moduls identisch mit dem Dateinamen ist. Legen wir dazu die Datei `A.hs` an.

```
1 module A (funktion1,       -- String
2           funktion2        -- String
3           ) where
4
5 funktion1 :: String
6 funktion1 = "Modul A - Funktion1"
7
8 funktion2 :: String
9 funktion2 = "Modul A - Funktion2"
```

Wir können nach dem Laden des Moduls `A` die beiden Funktionen verwenden und erhalten eine Ausgabe.

```
Hugs> funktion1
Hugs> "Modul A - Funktion1"
```

Sichtbarkeit von Funktionen

Angenommen, es gibt ein weiteres Modul `B`, mit folgender Beschreibung

```
1 module B (funktion1,       -- String
2           ) where
3
4 funktion1 :: String
5 funktion1 = "Modul B - Funktion1"
```

Modul `A` und `B` stellen beide eine Funktion mit dem Namen `funktion1` zur Verfügung. Wir wollen ein weiteres Modul C schreiben, dass die beiden Module `A` und `B` verwendet und ebenfalls eine zusätzliche Funktion `f1` bereitstellt

```
1 module C (f1) where
2
3 import A (funktion1, funktion2)
4 import B (funktion1)
5
6 f1 :: String
7 f1 = "Modul C - F1"
```

Durch `import` geben wir an, welche der zur Verfügung stehenden Funktionen wir aus dem Modul verwenden wollen. An dieser Stelle sollten auch immer nur die benötigten Funktionen angegeben werden. Nachdem wir das Modul `C` in Haskell geladen haben, können wir versuchen auf die verschiedenen Funktionen zuzugreifen

```
Hugs> f1
Hugs> "Modul C - Funktion1"
Hugs> funktion1
Hugs> "Modul B - Funktion1"
```

Was geschieht eigentlich, wenn wir auf Funktionen der anderen Module zugreifen? Versuchen wir zunächst `funktion2` und anschließend `funktion1`.

```
Hugs> funktion2
Hugs> "Modul A - Funktion2"
Hugs> funktion1
Hugs> "Modul B - Funktion1"
```

Die Funktion auf Modul `B` wurde ausgeführt. Die Reihenfolge der Importierung ist an dieser Stelle wichtig. Hätten wir die `import`-Zeilen 3 und 4 vertauscht, würde die Funktion aus Modul `A` ausgeführt werden. Um aber explizit anzugeben, aus welchem Modul die Funktion verwendet werden soll, können wir das über die Punktnotation `Modul.Funktion` vornehmen

```
Hugs> A.funktion1
Hugs> "Modul A - Funktion1"
```

Literatur

[1] O'Neill M.E.: „*The Genuine Sieve of Eratosthenes*", unpublished draft, siehe: *http://www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf*

[2] Block M.: „*Java Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*", Springer-Verlag 2007

[3] Dankmeier D.: „*Grundkurs Codierung: Verschlüsselung, Kompression, Fehlerbeseitigung.*", 3.Auflage, Vieweg-Verlag, 2006

[4] Schulz R.-H.: „*Codierungstheorie: Eine Einführung*", 2.Auflage, Vieweg+Teubner, 2003

[5] Schöning U.: „*Algorithmik*", ISBN-13: 978-3827410924, Spektrum Akademischer Verlag, 2001

[6] Pope B.: „*A tour of the Haskell Prelude*", unpublished (*http://www.cs.mu.oz.au/~bjpop/*), 2001

[7] Hudak P., Peterson J., Fasel J.: „*A gentle introduction to Haskell Version 98*", unpublished (*http://www.haskell.org/tutorial/*), 2000

[8] Cormen T.H., Leiserson C.E., Rivest R.L.: „*Introduction to Algorithms*", MIT-Press, 2000

[9] Gibbons J., Jones G.: „*The Under-Appreciated Unfold*", Proceedings of the third ACM SIGPLAN international conference on Functional programming, pp. 273-279, United States, 1998

[10] Haskell-Onlinereport 98: *http://haskell.org/onlinereport/index.html*

[11] Data.Char: *http://www.haskell.org/ghc/docs/6.4.1/html/libraries/base/Data-Char.html*

[12] Webseite der Helium-IDE: *http://www.cs.uu.nl/wiki/Helium*

[13] Wikibook zur Datenkompression: *http://de.wikibooks.org/wiki/Datenkompression*

[14] Haskell-Funktionen: *http://www.zvon.org/other/haskell/Outputglobal/index.html*

[15] Webseite des Euler-Projekts: *http://projecteuler.net/*

[16] Webseite Haskellprojekte: *http://hackage.haskell.org/packages/archive/pkg-list.html*

[17] Projektwebseite Frag: *http://haskell.org/haskellwiki/Frag*

[18] Projektwebseite Monadius: *http://www.geocities.jp/takascience/haskell/monadius_en.html*

[19] Haskell-Suchmaschine Hoogle: *http://www.haskell.org/hoogle/*

[20] Wikipedia: *http://www.wikipedia.com*