Fachhochschule
Nordwestschweiz

Functional Programming

Higher Order Functions

Daniel Kröni
University of Applied Sciences Northwestern Switzerland

Learning Targets

You recognize the most common patterns of recursion.

You can replace recursive definitions by parameterized higher order functions.

Recursion is the 'goto' of functional programming.

Eric Meijer

Content

- List transformations
- List removals
- List aggregations

Worksheet: List Transformations

- **Square a list of ints**

```
squares :: [Int] -> [Int]
squares []     = []
squares (i:is) = i^2 : squares is
```

- **Extract email addresses**

```
data Student = Student { email :: String, grade :: Float }

emails:: [Student] -> [String]
emails []     = []
emails (s:ss) = email s : emails ss
```

Worksheet: List Transformations

- **Square a list of ints**

```
transform :: [Int] -> [Int]
transform []     = []
transform (i:is) = i^2 : transform is
```

- **Extract email addresses**

```
data Student = Student { email :: String, grade :: Float }

transform :: [Student] -> [String]
transform []     = []
transform (s:ss) = email s : transform ss
```

Worksheet: List Transformations

- **Square a list of ints**

```
transform :: [Int] -> [Int]
transform []     = []
transform (a:as) = a^2 : transform as
```

- **Extract email addresses**

```
data Student = Student { email :: String, grade :: Float }

transform :: [Student] -> [String]
transform []     = []
transform (a:as) = email a : transform as
```

Worksheet: List Transformations

- **Square a list of ints**

```
transform :: [Int] -> [Int]
transform []     = []
transform (a:as) = a^2 : transform as
```

- **Extract email addresses**

```
data Student = Student { email :: String, grade :: Float }

transform :: [Student] -> [String]
transform []     = []
transform (a:as) = email a : transform as
```

Worksheet: List Transformations

- **Square a list of ints**

```
transform :: [Int] -> [Int]
transform []     = []
transform (a:as) = f a : transform as
```

- **Extract email addresses**

```
data Student = Student { email :: String, grade :: Float }

transform :: [Student] -> [String]
transform []     = []
transform (a:as) = f a : transform as
```

Worksheet: List Transformations

- **Square a list of ints**

```
transform :: (Int -> Int) -> [Int] -> [Int]
transform _ []     = []
transform f (a:as) = f a : transform f as
```

- **Extract email addresses**

```
data Student = Student { email :: String, grade :: Float }

transform :: (Student -> String) -> [Student] -> [String]
transform _ []     = []
transform f (a:as) = f a : transform f as
```

Worksheet: List Transformations

- **Square a list of ints**

```
transform :: (a -> b) -> [a] -> [b]
transform _ []     = []
transform f (a:as) = f a : transform f as
```

- **Extract email addresses**

```
data Student = Student { email :: String, grade :: Float }

transform :: (a -> b) -> [a] -> [b]
transform _ []     = []
transform f (a:as) = f a : transform f as
```

Map

- **Implementation**

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []                    -- (map.0)
map f (x:xs) = f x : map f xs        -- (map.1)
```

- **Evaluation**

```
map f (a:(b:(c:[])))
~> f a : map f (b:(c:[]))            -- by (map.1)
~> f a : f b : map f (c:[])          -- by (map.1)
~> f a : f b : f c : map f []        -- by (map.1)
~> f a : f b : f c : []              -- by (map.0)
```

- **Properties**
  - The type of the list may change
  - The length of the list does not change

Worksheet: List removals

- **even numbers**

```
evens :: [Int] -> [Int]
evens [] = []
evens (i:is) | even i    = i : evens is
             | otherwise = evens is
```

- **'good' students**

```
data Student = Student { email :: String, grade :: Float }

goodS :: [Student] -> [Student]
goodS [] = []
goodS (s:ss) | grade s > 5  = s : goodS ss
             | otherwise    = goodS ss
```

Filter

- **Implementation**

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []                                       -- (fil.0)
filter f (x:xs) | f x         = x : filter f xs        -- (fil.1)
                | otherwise =     filter f xs          -- (fil.2)
```

- **Example**

```
filter even (2:(3:(4:[])))
~> 2 : filter even (3:(4:[]))     -- by (fil.1)
~> 2 : filter even (4:[])         -- by (fil.2)
~> 2 : 4 : filter even []         -- by (fil.1)
~> 2 : 4 : []                     -- by (fil.0)
```

- **Properties**
  - The type of the list does not change
  - The length of the list may change

Worksheet: List Aggregations

- **Examples**

```
sum []          = 0
sum (x:xs)      = x + sum xs
```

```
product []      = 1
product (x:xs) = x * product xs
```

```
or []           = False
or (x:xs)       = x || or xs
```

```
and []          = True
and (x:xs)      = x && and xs
```

List Aggregations

- **Examples**

```
sum []          = 0
sum (x:xs)      = x + sum xs
```

```
and []          = True
and (x:xs)      = x && and xs
```

- **Common pattern of recursion**

```
aggregate []            = z
aggregate (x:xs)        = x `op` aggregate xs
```

- **Abstracting over z and op**

```
aggregate :: (a -> a -> a) -> a -> [a] -> a
aggregate _  z []       = z
aggregate op z (x:xs)   = x `op` (aggregate op z xs)
```

List Aggregations

- **Counting spaces**

```
countSpace :: [Char] -> Int
countSpace []     = 0
countSpace (c:cs) = (if isSpace c then 1 else 0)
                    + countSpace cs
```

- **Sum of the prices of items in a basket**

```
data Item = Item { desc :: String, price :: Float }

total :: [Item] -> Float
total []     = 0
total (i:is) = price i + total is
```

Fold right

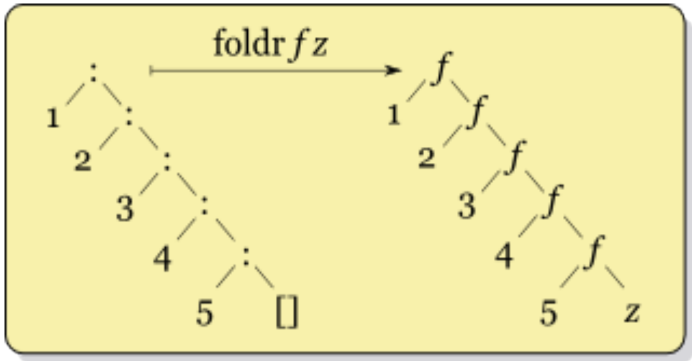
- **Implementation**

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

  – If the list is empty the result is the initial value z else
  – apply f to the first element and the result of folding the rest

```
foldr f z [1,2,3,4,5]
=
f 1 (f 2 (f 3 (f 4 (f 5 z))))
```

```
foldr ⊙ z [1,2,3,4,5]
=
1 ⊙ (2 ⊙ (3 ⊙ (4 ⊙ (5 ⊙ z))))
```



http://www.haskell.org/haskellwiki/Fold

Fold left

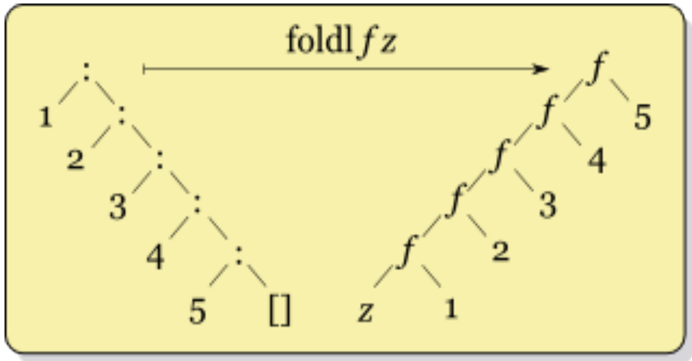
- **Implementation**

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ z []      = z
foldl f z (x:xs) = foldl f (f z x) xs
```

  - If the list is empty the result is the initial value z else
  - apply f to the initial value and first element and use the result as the new initial value for folding the rest

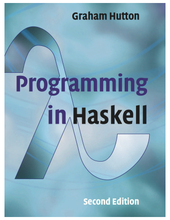
```
foldl f z [1,2,3,4,5]
=
f (f (f (f (f z 1) 2) 3) 4) 5
```

```
foldl ⊙ z [1,2,3,4,5]
=
((((z ⊙ 1) ⊙ 2) ⊙ 3) ⊙ 4) ⊙ 5
```

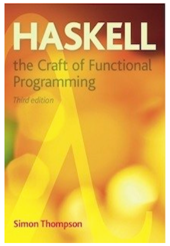


http://www.haskell.org/haskellwiki/Fold

Further Reading



Chapter 7



Chapter 11

Chapter 5

http://learnyouahaskell.com/higher-order-functions