Programming Paradigms

Summer Term 2017

8th Lecture

Prof. Janis Voigtländer
University of Duisburg-Essen

Higher-Order: somewhat artificial examples

- Function as parameter and result:

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

- Somewhat more explicit (with λ-abstraction):

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

- Currying inside the language:

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

- And conversely:

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

Commonly used higher-order functions on lists (1)

- A very useful example function that takes another function as parameter, and then applies it to all elements of a list, is the map-function:

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

function as parameter

- Two different applications of this function:

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

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

- The function map is polymorphic:

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

Commonly used higher-order functions on lists (2)

- Beside map, there are several further important higher-order functions for working with lists: filter, foldl, foldr, zipWith, scanl, scanr, …

- The function filter lets us select list elements that satisfy a certain Boolean condition:

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

“predicate”

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

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

abbreviation for \x → x > 3
(reminder: “section”)

Effective use of higher-order functions

- Rather un-idiomatic Haskell:

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

- Better:

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

- Further functions useful for this style: zip, splitAt, takeWhile, repeat, iterate, …

Further examples for using higher-order functions

- What does the following function achieve (in the context of Gloss)?

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

- And this one?

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

- Something of similar spirit is part of the exercises as a bonus task.

Further examples for using higher-order functions

- Recall:

```
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
```

- Let’s assume we want to add subtraction and division.

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

- Possible problem: division by zero, hence …

Further examples for using higher-order functions

- To take care of possible division by zero, we could proceed as follows:

```
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)
…
```

- But to avoid these tedious case-cascades, abstraction of the essence into:

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

- And then, e.g.:

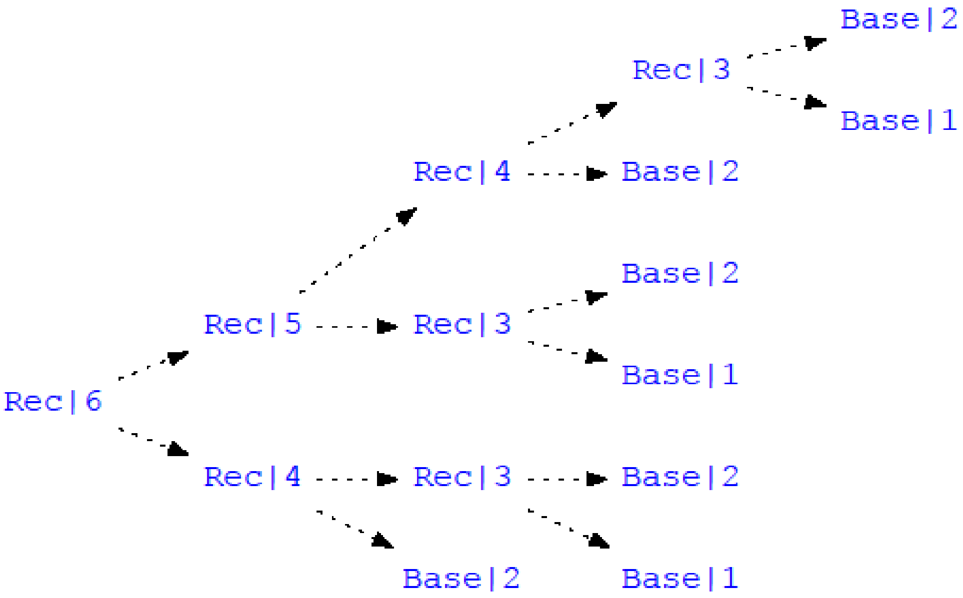
```
eval  (Add e1 e2)  =  eval e1 `andThen` \r1 →
                      eval e2 `andThen` \r2 → Just (r1 + r2)
```

Higher-Order: a somewhat more complex example, memoization (1)

- Let’s consider the following program, which is very inefficient:

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

- The inefficiency is due to the structure of the “call graph” (here for fib 6):



Higher-Order: a somewhat more complex example, memoization (2)

- Let’s consider the following program, which is very inefficient:

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

- We can make function results “reusable”, in a very canonical way, independently of the concrete fib-function:

```
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   -- after a few seconds
> mfib 30
1346269   -- “immediately”
```

Higher-Order: a somewhat more complex example, memoization (3)

- It is even better to exploit memoization also inside the recursion:

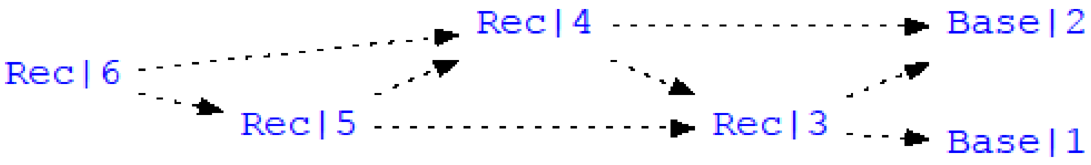
```
mfib = memo fib

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

- Since then:

```
> fib 30
1346269   -- “immediately”
> fib 31
2178309   -- “even faster”
```

- “Call graph” now:



Structural recursion on lists as a higher-order function

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

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

- The list functions for summing or multiplying list elements use the same recursion pattern, which can be realized with the help of a standard function for “folding” binary operators over lists:

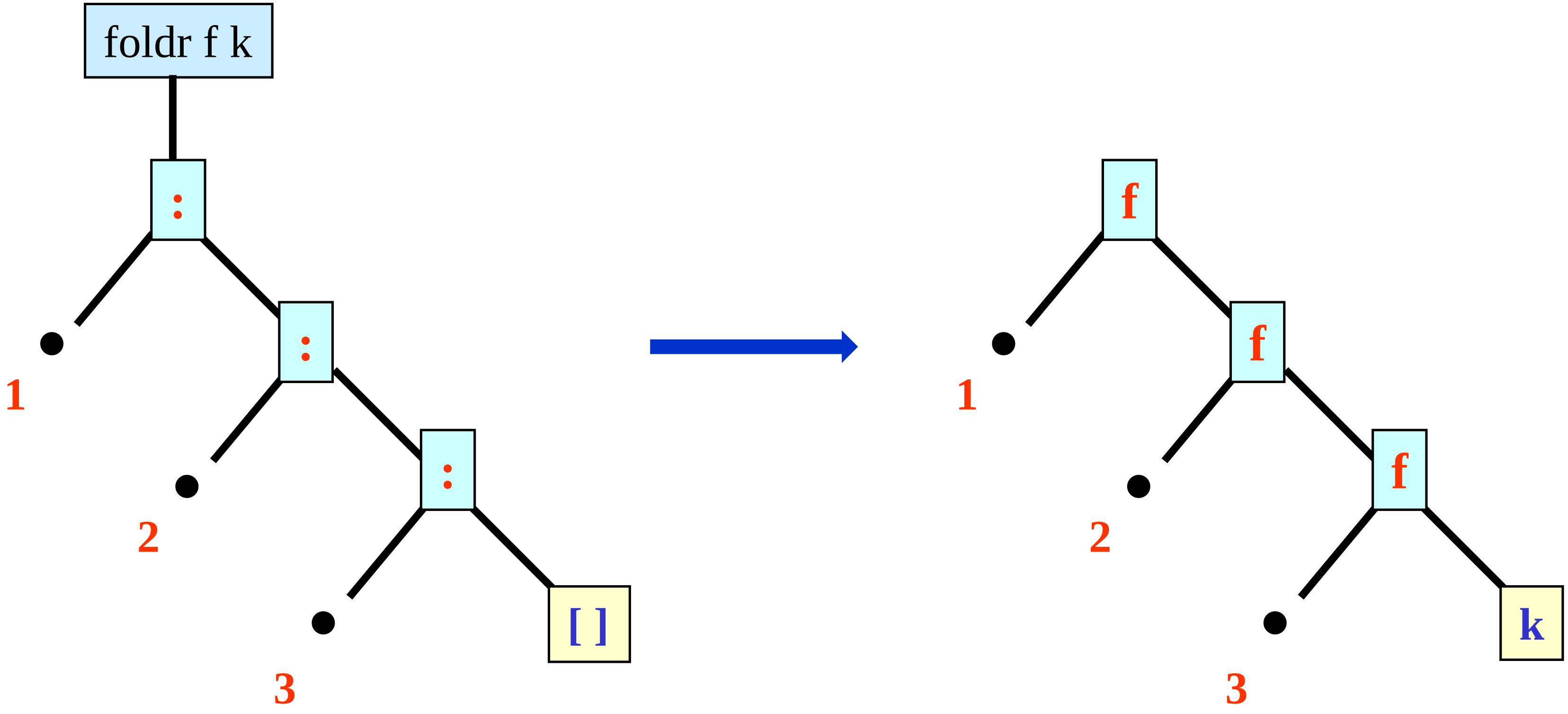
```
foldr  :: (a → b → b) → b → [a] → b
foldr  f  k  [ ]       =  k
foldr  f  k  (x : xs)  =  f  x  (foldr  f  k  xs)
```

(..r for “right”;
there is also a foldl)

- For example, definitions of sum and prod as applications of foldr:

```
sum, prod :: [Int] → Int
sum   =  foldr  (+)  0
prod  =  foldr  (*)  1
```

Visualization of foldr



Further examples for using foldr

- By using foldr, there are predefined logical junctors that operate on implementiert, lists of Boolean values:

```
and, or  :: [Bool] →  Bool
and  =  foldr (&&) True
or  =  foldr (| |) False
```

- "Quantors" over lists are realized as generalizations of these junctors via composition:

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

```
e.g.:   all (<100)  [ x^2 | x ← [1 .. 19] ]
```

General strategy for using foldr

- When can a function be expressed using foldr?

- Whenever it is possible to bring it into the following form:

```
g [ ]        =  k
g (x : xs)  =  f  x (g xs)
```

for any k and f

- Then:

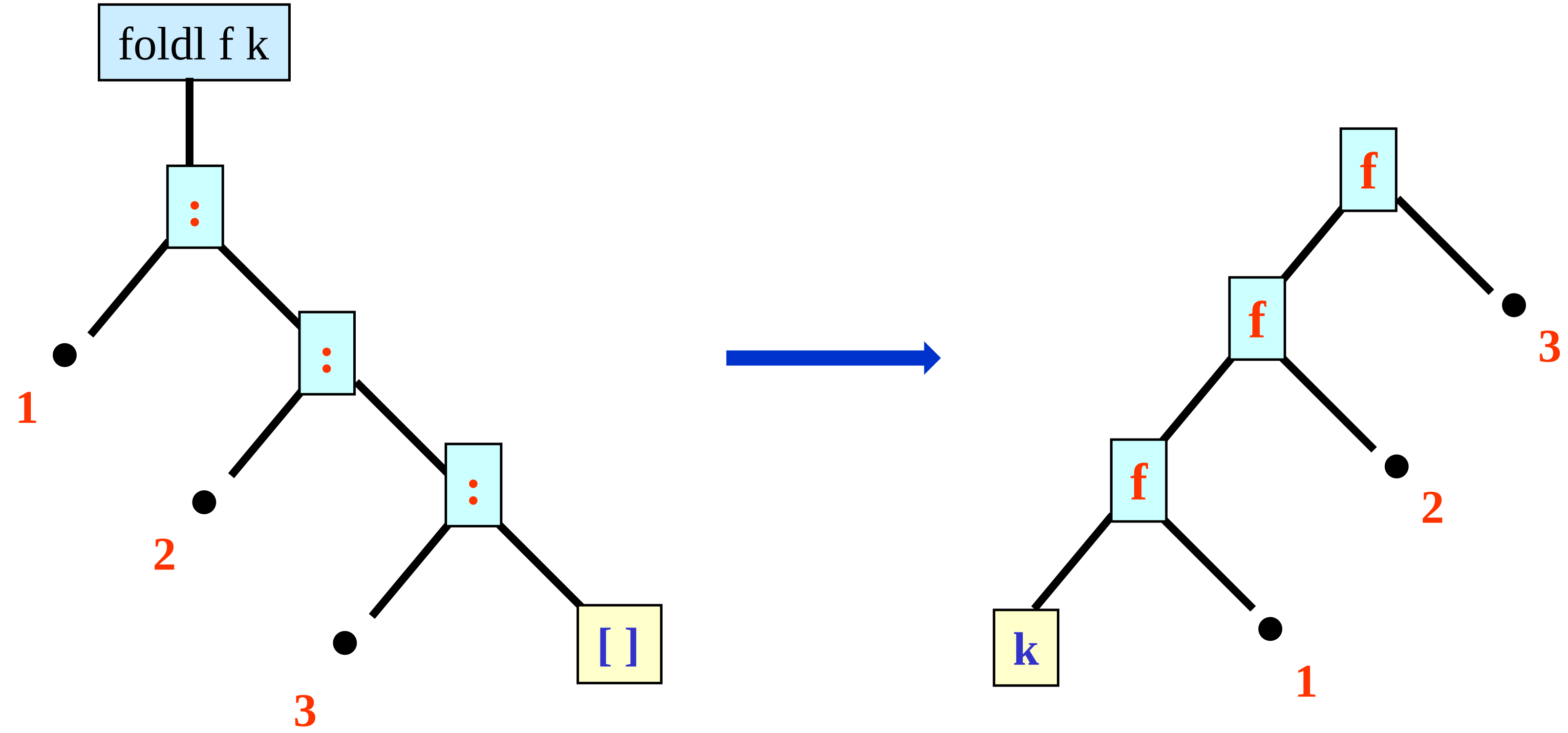
```
g  =  foldr  f  k
```

- This gives a simple (and complete) characterization of structural recursion on lists!

A left-leaning variant of foldr

Beside foldr, there is:

```
foldl  :: (b → a → b) → b → [a] → b
foldl  f  k  [ ]        =  k
foldl  f  k  (x : xs)  =  foldl  f  (f  k  x)  xs
```



Variations on foldl and foldr

- Returns also all the intermediate results of foldl:

```
scanl :: (b → a → b) → b → [a] → [b]
scanl f  k xs = k : case xs of
                     [ ]      → [ ]
                     x : xs' → scanl f  (f  k  x)  xs'
```

- For example:

```
> scanl (+) 0 [1 .. 5]
[0, 1, 3, 6, 10, 15]
```

- In a certain sense dual to foldr:

```
unfoldr :: (b → Maybe (a, b)) → b → [a]
unfoldr f b = case f b of
                Nothing    → [ ]
                Just (a, b') → a : unfoldr f b'
```