Concepts of Higher Programming Languages
Higher-Order Functions

Jonathan Thaler

Department of Computer Science

FH Vorarlberg
University of Applied Sciences

Introduction

Definition

A function is called **higher-order** if it takes a function as an argument or returns a function as a result.

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

`twice` is higher-order because it takes a **function** as its **first argument**.

Why Are They Useful?

- **Common programming idioms** can be encoded as functions within the language itself.
- **Domain specific languages** can be defined as collections of higher-order functions.
- **Encapsulation using partial function application** can be used to hide implementation details.
- **Algebraic properties** of higher-order functions can be used to reason about programs.

The Map Function

Definition

The higher-order library function called `map` applies a function to every element of a list.

```
map :: (a -> b) -> [a] -> [b]
?
?

> map (+1) [1,3,5,7]
[2,4,6,8]
```

The Map Function

Definition

The higher-order library function called `map` applies a function to every element of a list.

```haskell
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

> map (+1) [1,3,5,7]
[2,4,6,8]
```

The Filter Function

Definition

The higher-order library function `filter` selects every element from a list that satisfies a predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
?
?
?
?`

> filter even [1..10]
[2,4,6,8,10]
```

The Filter Function

Definition

The higher-order library function `filter` selects every element from a list that satisfies a predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
   | p x        = x : filter p xs
   | otherwise = filter p xs

> filter even [1..10]
[2,4,6,8,10]
```

Folding

A number of functions on lists can be defined using the following simple pattern of recursion:

Definition

```
f []     = v
f (x:xs) = x ⊕ f xs
```

`f` maps the **empty list** to some value `v`, and any **non-empty list** to some function $\oplus$ applied to its head and `f` of its tail.

Folding

v=0 ⊕=+

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

v=1 ⊕=*

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

v=True ⊕=&&

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

The Foldr Function

The higher-order library function foldr (**fold right**) encapsulates this simple pattern of recursion, with the function $\oplus$ and the value v as arguments.

```haskell
sum     = foldr (+) 0

product = foldr (*) 1

or      = foldr (||) False

and     = foldr (&&) True
```

The Foldr Function

`foldr` itself can be defined using recursion:

Definition

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

It is best to think of `foldr` non-recursively, as simultaneously replacing each `(:)` in a list by a given function, and `[]` by a given value.

The Foldr Function

Evaluating `sum` - Replace each `(:)` by `(+)` and `[]` by `0`

```
  sum [1,2,3]
=
  foldr (+) 0 (1:(2:(3:[])))
=
  (+) 1 (foldr (+) 0 (2:(3:[])))
=
  (+) 1 ((+) 2 (foldr (+) 0 (3:[])))
=
  (+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))
=
  (+) 1 ((+) 2 ((+) 3 0))
=
  (+) 1 ((+) 2 3)
=
  (+) 1 5
=
  6
```

The Foldr Function

Evaluating `product` - Replace each `(:)` by `(*)` and `[]` by 1.

```
  product [2,3,4]
=
  foldr (*) 1 (2:(3:(4:[])))
=
  (*) 2 (foldr (*) 1 (3:(4:[])))
=
  (*) 2 ((*) 3 (foldr (*) 1 (4:[])))
=
  (*) 2 ((*) 3 ((*) 4 (foldr (*) 1 [])))
=
  (*) 2 ((*) 3 ((*) 4 1))
=
  (*) 2 ((*) 3 4)
=
  (*) 2 12
=
  24
```

Other Foldr Examples

Even though `foldr` encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.

Recall the `length` function:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Other Foldr Examples

Evaluating `length`

```
  length [1,2,3]
=
  length (1:(2:(3:[])))
=
  1+(1+(1+0))
=
  3
```

Replace each `(:)` by `\_ n -> 1+n` and `[]` by `0`.

Definition

```
length = foldr (\_ n -> 1+n) 0
```

Other Foldr Examples

Recall the reverse function:

```haskell
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```
  reverse [1,2,3]
=
  reverse (1:(2:(3:[])))
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

Replace each `(:)` by `\x xs -> xs ++ [x]` and `[]` by `[]`.

Other Foldr Examples

We have:

Definition

```
reverse = foldr (\x xs -> xs ++ [x]) []
```

Finally, we note that the append function `(++)` has a particularly compact definition using `foldr`:

```
(++ ys) = foldr (:) ys
```

Replace each `(:)` by `(:)` and `[]` by `ys`.

The Foldl Function

`foldr` folds from the **right**. If folding from the **left** is required (e.g. operator is not *commutative*), use `foldl`:

Definition

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

`foldr` and `foldl` have nearly identical signatures, except in the folding function they take as argument:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldl :: (b -> a -> b) -> b -> [a] -> b
```

The Foldl Function

Evaluating sum with foldl

```
  sum [1,2,3]
=
  foldl (+) 0 (1:(2:(3:[])))
=
  foldl (+) ((+) 0 1)) (2:(3:[]))
=
  foldl (+) ((+) ((+) 0 1) 2) (3:[])
=
  foldl (+) ((+) ((+) ((+) 0 1) 2) 3) []
=
  (+) ((+) ((+) 0 1) 2) 3
=
  (+) ((+) 1 2) 3
=
  (+) 3 3
=
  6
```

The Foldl Function

Evaluating product with foldl

```
  product [2,3,4]
=
  foldl (*) 1 (2:(3:(3:[])))
=
  foldl (*) ((*) 1 2) 3:(4:[]))
=
  foldl (*) ((*) ((*) 1 2) 3) (4:[])
=
  foldl (*) ((*) ((*) ((*) 1 2) 3) 4) []
=
  (*) ((*) ((*) 1 2) 3) 4
=
  (*) ((*) 2 3) 4
=
  (*) 6 4
=
  24
```

The Foldl Function

Definition

`foldl` can be understood to **accumulate** the final value step-by-step by

1. Mapping the **empty list** to the **accumulator** value `v`.
2. Mapping any **non-empty list** to the result of **recursively processing the tail** using a **new accumulator** value obtained by applying an operator $\oplus$ to the current value and the head of the list.

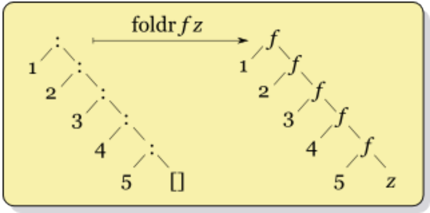
This can be expressed in the following pattern of recursion:

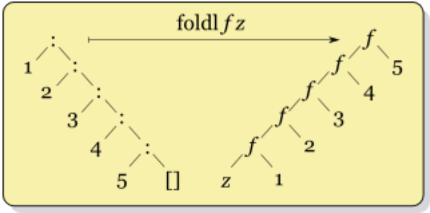
Definition

```
f []     = v
f (x:xs) = f (v ⊕ x) xs
```

Foldr and Foldl

The recursion patterns of `foldr` and `foldl` can be visualised as the following structural transformation:





Foldr and Foldl

Evaluating `sum` with `foldl`

```
  sum [1,2,3]
=
  foldl (+) 0 (1:(2:(3:[])))
=
  foldl (+) ((+) 0 1)) (2:(3:[]))
=
  foldl (+) ((+) ((+) 0 1) 2) (3:[])
=
  foldl (+) ((+) ((+) ((+) 0 1) 2) 3) []
=
  (+) ((+) ((+) 0 1) 2) 3
=
  (+) ((+) 1 2) 3
=
  (+) 3 3
=
  6
```

Evaluating `sum` with `foldr`

```
  sum [1,2,3]
=
  foldr (+) 0 (1:(2:(3:[])))
=
  (+) 1 (foldr (+) 0 (2:(3:[])))
=
  (+) 1 ((+) 2 (foldr (+) 0 (3:[])))
=
  (+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))
=
  (+) 1 ((+) 2 ((+) 3 0))
=
  (+) 1 ((+) 2 3)
=
  (+) 1 5
=
  6
```

Foldr vs. Foldl

- `foldr` is most commonly the right fold to use, in particular when **transforming lists** (or other foldables) into lists with related elements in the **same order**.
- The folding function of `foldr` can **short-circuit**, that is, terminate early by yielding a result which does not depend on the value of the accumulating parameter. **Left folds can never short-circuit**.
- **foldl** is **not** able to deal with **inifinite lists**, but **foldr** is. We will have a look at this in *Chapter 8: Lazy Evaluation*.
- There is a **strict** version of `foldl` called `foldl'`, which often has better run time and memory behaviour than `foldr`. Again, we will have a look at this in *Chapter 8: Lazy Evaluation*.
- For an in-depth discussion of folds see `https://wiki.haskell.org/Fold`

Foldr and Foldl

Short-Circuiting

```
> foldl (\sum x -> if x == 2 then x else x + sum) 0 [1..]
Killed

> foldr (\x sum -> if x == 2 then x else x + sum) 0 [1..]
3
```

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

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

Why Is Fold Useful?

- Some recursive functions on lists, such as `sum`, are **simpler** to define using `fold`.
- Properties of functions defined using `fold` can be proved using **algebraic properties** of `fold`, such as fusion and the banana split rule.
- Advanced program **optimisations** can be simpler if `fold` is used in place of explicit recursion.

Other Library Functions

The library function `(.)` returns the composition of two functions as a single function.

Definition

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

Example

```
odd :: Int -> Bool
odd = not . even
```

Other Library Functions

The library function `(.)` returns the composition of two functions as a single function.

Definition

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

Example

```
odd :: Int -> Bool
odd = not . even
```

Other Library Functions

The library function `all` decides if every element of a list satisfies a given predicate.

Definition

```
all :: (a -> Bool) -> [a] -> Bool
all p xs = and (map p xs)
```

Example

```
> all even [2,4,6,8,10]

True
```

Other Library Functions

Dually, the library function `any` decides if at least one element of a list satisfies a predicate.

Definition

```
any :: (a -> Bool) -> [a] -> Bool
any p xs = or (map p xs)
```

Example

```
> any (== ' ') "abc def"

True
```

Other Library Functions

The library function `takeWhile` selects elements from a list while a predicate holds of all the elements.

Definition

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile ?
takeWhile ?
```

Example

```
> takeWhile (/= ' ') "abc def"

"abc"
```

Other Library Functions

The library function `takeWhile` selects elements from a list while a predicate holds of all the elements.

Definition

```haskell
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
   | p x       = x : takeWhile p xs
   | otherwise = []
```

Example

```haskell
> takeWhile (/= ' ') "abc def"

"abc"
```

Other Library Functions

Dually, the function `dropWhile` removes elements while a predicate holds of all the elements.

Definition

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile ?
dropWhile ?
```

Example

```
> dropWhile (== ' ' ) "      abc"

"abc"
```

Other Library Functions

Dually, the function `dropWhile` removes elements while a predicate holds of all the elements.

Definition

```haskell
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs)
   | p x         = dropWhile p xs
   | otherwise = x:xs
```

Example

```
> dropWhile (== ' ' ) "     abc"

"abc"
```

Other Library Functions

The library function `uncurry` takes a curried function and constructs an uncurried version of this function.

Definition

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

```
addCurried :: Int -> Int -> Int
addCurried x y = x+y

> addCurried 1 2
3
> uncurry addCurried (1,2)
3
> :t uncurry addCurried :: (Int,Int) -> Int
```

Other Library Functions

The library function `uncurry` takes a curried function and constructs an uncurried version of this function.

Definition

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

```
addCurried :: Int -> Int -> Int
addCurried x y = x+y

> addCurried 1 2
3
> uncurry addCurried (1,2)
3
> :t uncurry addCurried :: (Int,Int) -> Int
```

Other Library Functions

Dually, the library function `curry` takes an uncurried function and constructs a curried version of this function.

Definition

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

```
addUncurried :: (Int,Int) -> Int
addUncurried (x,y) = x+y

> addUncurried (1,2)
3
> curry addUncurried 1 2
3
> :t curry addUncurried :: Int -> Int -> Int
```

Other Library Functions

Dually, the library function `curry` takes an uncurried function and constructs a curried version of this function.

Definition

```haskell
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f = \x y -> f (x, y)
```

```haskell
addUncurried :: (Int,Int) -> Int
addUncurried (x,y) = x+y

> addUncurried (1,2)
3
> curry addUncurried 1 2
3
> :t curry addUncurried :: Int -> Int -> Int
```

Other Library Functions

Identity of `curry`

```
curry . uncurry == id

> :t curry . uncurry
curry . uncurry :: (a -> b -> c) -> a -> b -> c
```

Identity of `uncurry`

```
uncurry . curry  == id

> :t curry . uncurry
uncurry . curry :: ((a, b) -> c) -> (a,b) -> c
```