Session 9: Folds continued and λ-calculus

COMP2221: Functional programming

Laura Morgenstern

laura.morgenstern@durham.ac.uk

Recap

- Introduced lazy evaluation
- Saw how expression graphs are evaluated with innermost and outermost strategy
- Contrasted pros and cons of lazy and eager evaluation
- Introduced the idea of folds

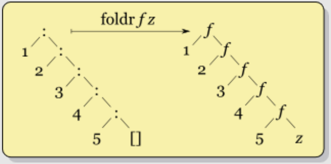
Folds: (yet another) family of higher order functions

Folds

- *folds* process a data structure in some order and build a return value
- Haskell provides a number of these in the standard prelude, with more available in the `Data.List` module

`foldr`: right associative fold

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

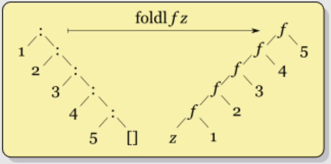


Folds

- *folds* process a data structure in some order and build a return value
- Haskell provides a number of these in the standard prelude, with more available in the `Data.List` module

`foldl`: left associative fold

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



How to think about this

- `foldr` and `foldl` are recursive
- However, often easier to think of them *non-recursively*

foldr

Replace `(:)` by the given function, and `[]` by given value.

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

foldl

Same idea, but associating to the left

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

Purpose of folds

- Capture many linear recursive patterns in a clean way
- Can have efficient library implementation ⇒ can apply program optimisations
- Actually apply to all `Foldable` types, not just lists
- e.g. `foldr`'s type is actually

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

- So we can write code for lists and (say) trees identically

Folds are general

- Many library functions on lists are written *using folds*

```
product = foldr (*) 1
sum = foldr (+) 0
maximum = foldr1 max
```

- Practical sheet 4 asks you to define some others

Which to choose?

foldr

- Generally `foldr` is the right choice
- Works even for infinite lists
- Note `foldr (:) [] == id`
- Can terminate early

foldl

- Can't terminate early
- Doesn't work on infinite lists
- Usually best to use *strict* version:

```haskell
import Data.List
foldl' -- note trailing '
```

- Aside: it is probably a historical accident that `foldl` is not strict (see `http://www.well-typed.com/blog/90/`)

$\Rightarrow$ Caution: `foldr` and `foldl` lead to different result if `f` not commutative

Foldable data structures

- `Foldable` type class: if we can *combine* an `a` and a `b` to produce a new `b`, then, given a start value and a container of `a`s we can reduce it to a `b`

```
class Foldable f where
  -- minimal definition requires this
  foldr :: (a -> b -> b) -> b -> f a -> b

data List a = Nil | Cons a (List a)
  deriving (Eq, Show)

instance Foldable List where
    foldr :: (a -> b -> b) -> b -> List a -> b
    foldr _ z Nil            = z
    foldr binop z (Cons a tail) = a `binop` (foldr binop z tail)
```

λ-calculus

Introduction

- Simplest known turing-complete programming language
- Inspired functional programming languages
- Calculus: set of rules to transform things
- λ-calculus: set of rules to transform expressions of the following form
  - v (Variables; lower case letters)
  - (MN) (Application of M to N)
  - $(\lambda v.M)$ (Abstraction aka function with parameter v and body M)
  - with M and N being expressions of the same form
- Functions take exactly one argument

Examples

Valid λ-expressions:

- $x \rightarrow$ a variable
- $(\lambda x.x) \rightarrow$ the identity function
- $((\lambda x.x)a) \rightarrow$ the identity function applied to value a
- $(\lambda x.(\lambda y.(xy))) \rightarrow$ nested function, i.e. currying
- $(((\lambda x.(\lambda y.(xy)))a)b) \rightarrow$ nested function applied to values a and b

$\rightarrow$ application associates to the left $\rightarrow$ abstraction associates to the right

Transformation rules

α-Conversion

α-conversion allows to resolve name conflicts by renaming parameters via $(\lambda x.M[x]) \to (\lambda y.M[y])$.

β-Reduction

β-reduction allows to substitute the argument of an abstraction with the value of an application $((\lambda x.M[x])N) \to (M[x := N])$.

Summary

- Saw implementation of `foldr` and `foldl`
- Introduced and used type class *Foldable* to capture computational pattern *reduction*
- Introduced syntax of λ-calculus
- Saw how abstraction, application and reduction work in λ-calculus