Session 9: Folds continued and monads

COMP2221: Functional programming

Laura Morgenstern

laura.morgenstern@durham.ac.uk

Recap

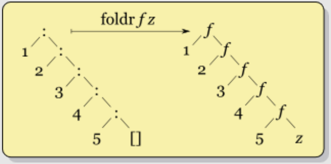
- Introduced lazy evaluation
- Saw how expression graphs are evaluated with innermost and outermost strategy
- Contrasted pros and cons of lazy and eager evaluation
- Introduced the idea of folds

Folds

- *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)
```

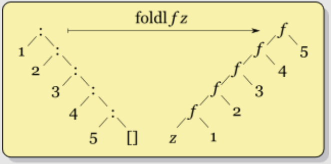


Folds

- *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 recursively defined
- However, easier to think about there semantics *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 $\Rightarrow$ can apply program optimisations
- Actually apply to all `Foldable` types, not just lists
- e.g. `foldr`'s type is actually

```haskell
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*

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

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:

  ```
  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/`)

⇒ 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)
```

Monads

Monad

- In category theory, a monad is a functor with additional structure
- In Haskell, can consider it as an abstract datatype for actions (do notation as syntactic sugar for writing monadic expressions)
- Monads can be used to structure and compose computations
- Essentially, a standard programming interface for data and control structures

Monad type class

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    return :: a -> m a
```

- `return`:
  - wrap a value in a context - resulting in the monadic value `m a`
  - Note: not like return in imperative programming languages - does not end function execution
- Bind operator `>>=`:
  - Compose two actions, passing any value produced by the first as an argument to the second
  - Definition contains instance-dependent implementation of additional actions
- Bind operator `>>` without value passing

Example for composition with bind: function chaining

```haskell
stringToNum :: String -> IO Int
stringToNum s = return (read s)

inc :: Int -> IO Int
inc x = return (x + 1)

main :: IO ()
main = getLine >>= stringToNum >>= inc >>= print
```

Equivalently, with `do` notation as syntactic sugar:

```haskell
main :: IO ()
main = do
    input <- getLine
    num <- stringToNum input
    result <- inc num
    print result
```

Monad laws

- Monads have to fullfill monad laws to behave properly
- Left identity: `return a >>= f <=> f a`

→ Wrapping a value in a context and binding it to a function is the same as applying the function to the extracted value

- Right identity: `m >>= return <=> m`

→ Taking a monadic value and binding it to return leaves the monadic value unchanged

- Associativity:

```
 (m  >>=  (\x -> g x))  >>=  (\y -> h y)
<=>
m  >>=  (\x -> g x    >>=  (\y -> h y))
```

Maybe monad

- `Maybe` monad represents computations which can fail by not returning a value

```haskell
data Maybe a = Just a | Nothing

instance Monad Maybe where
    Nothing >>= f = Nothing
    Just x >>= f  = f x
    return x = Just x
```

Monads in Haskell

- `Maybe`: provides context to model failure
- `IO`: represents IO actions `(>>=) :: IO a -> (a -> IO b) -> IO b`, e.g., to allow waiting for user input; `getLine` does not return a `String` but is rather an IO action which resolves as a string on evaluation.
- `List`: binding means joining together a set of calculations for each value in the list - `(>>=) :: [a] -> (a -> [b]) -> [b]`
- And many others: `Either`, `MonadState`, `MonadReader`, `MonadWriter`, ...