Session 8: Lazy evaluation and folds

COMP2221: Functional programming

Laura Morgenstern

laura.morgenstern@durham.ac.uk

Recap

- Introduced higher order functions, saw examples `map`, `filter`, `any`, . . .
- `Functor` as a type class for mappable containers
- *Functor laws*
  - `fmap id == id`
  - `fmap (f . g) == fmap f . fmap g`
- Discussed purpose of type class instances for custom data types

Recap: Correctness of `listMap`

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

instance Functor List where
  fmap _ Nil = Nil
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

To show `fmap id == id`, need to show `fmap id (Cons x xs) == Cons x xs` for any x, xs.

```haskell
-- Induction hypothesis
fmap id xs = xs
-- Base case
-- apply definition
fmap id Nil = Nil
-- Inductive case
fmap id (Cons x xs) = Cons (id x) (fmap id xs)
== Cons x (fmap id xs)
== Cons x xs -- Done!
```

Exercise: check whether the second law holds

Lazy evaluation

How does this work?

Fibonacci sequence

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

```haskell
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
Prelude> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

How long?

```python
def slow_function(a):
    ... # 5 minute computation

def compute(a, b):
    if a == 0:
        return 1
    else:
        return b

compute(0, slow_function(0))
compute(1, slow_function(1))
```

```haskell
slow_function :: Int -> Int
-- 5 minute computation
slow_function a = ...

compute :: Int -> Int -> Int
compute a b | a == 0    = 1
            | otherwise = b

compute 0 (slow_function 0)
compute 1 (slow_function 1)
```

Lazy evaluation AKA I'll get it when you need it

- Not only is Haskell a pure *functional* language
- It is also evaluated *lazily*
- Hence, we can work with infinite data structures
- . . . and defer computation until such time as it's strictly necessary

Definition (Lazy evaluation)

Expressions are not evaluated when they are bound to variables. Instead, their evaluation is *deferred* until their result is needed by other computations.

Evaluation strategies

- Haskell's basic method of computation is *application* of functions to arguments
- Even here, though we already have some freedom

Example

```
inc :: Int -> Int
inc n = n + 1

inc (2*3)
```

Two options for the evaluation order

```
inc (2*3)
= inc 6 -- applying *
= 6 + 1 -- applying inc
= 7 -- applying +
```

```
inc (2*3)
= (2*3) + 1 -- applying inc
= 6 + 1 -- applying *
= 7 -- applying +
```

- As long as all the expression evaluations *terminate*, the order we choose to do things doesn't matter.

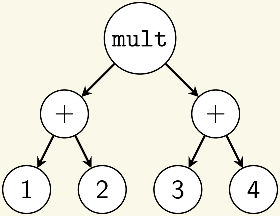
Evaluation strategies II

- We can represent a function call and its arguments in Haskell as a graph
- Nodes in the graph are either *terminal* or *compound*. The latter are called *reducible expressions* or *redexes*.

Example

```haskell
mult :: (Int, Int) -> Int
mult (x, y) = x*y

mult (1+2, 3+4)
```

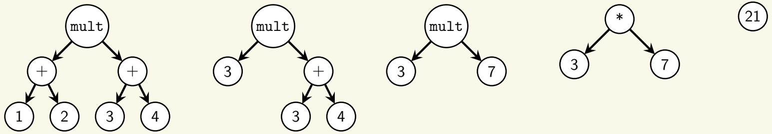


- `1`, `2`, `3`, and `4` are terminal (not reducible) expressions
- `(+)` and `mult` are reducible expressions.

Innermost evaluation

- Evaluate "bottom up"
- First evaluate redexes that only contain terminal or *irreducible* expressions, then repeat
- Need to specify evaluation order at leaves. Typically: "left to right"

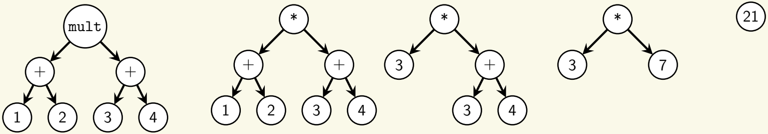
Example



Outermost evaluation

- Evaluate "top down"
- First evaluate redexes that are outermost, then repeat
- Again, need an evaluation order for children, typically choose "left to right".

Example



Termination

- For *finite* expressions, both innermost and outermost evaluation terminate.
- Not so for infinite expressions

Example

```
inf :: Integer
inf = 1 + inf
fst :: (a, b) -> a
fst (x, _) = x
Prelude> fst (0, inf)
```

- Innermost evaluation will fail to terminate here, whereas outermost evaluation produces a result.

Termination II

Innermost evaluation: never terminates

```haskell
inf :: Integer
inf = 1 + inf
fst :: (a, b) -> a
fst (x, _) = x
Prelude> fst (0, inf)
Prelude> fst (0, 1 + inf) -- applying inf
Prelude> fst (0, 1 + 1 + inf) -- applying inf
...
```

Outermost evaluation: terminates in one step

```haskell
inf :: Integer
inf = 1 + inf
fst :: (a, b) -> a
fst (x, _) = x
Prelude> fst (0, inf)
0  -- applying fst
```

Call by name or value?

Call by value

- Also called *eager evaluation*
- Innermost evaluation
- Arguments to functions are always fully evaluated before the function is applied
- Each argument is evaluated exactly once
- Evaluation strategy for most imperative languages

Call by name

- Also called *lazy evaluation*
- Outermost evaluation
- Functions are applied *before* their arguments are evaluated
- Each argument may be evaluated more than once
- Evaluation strategy in Haskell (and others)

Avoiding inefficiences: sharing

- Straightforward implementation of call-by-name can lead to inefficiency in the number of times an argument is evaluated

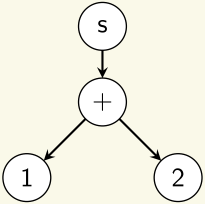
Example

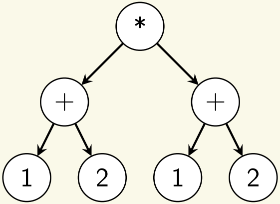
```
square :: Int -> Int
square n = n * n
Prelude> square (1+2)
== (1 + 2) * (1 + 2) -- applying square
== 3 * (1 + 2) -- applying +
== 3 * 3 -- applying +
== 9
```

- To avoid this, Haskell implements *sharing* of arguments.
- We can think of this as rewriting the evaluation tree into a graph.

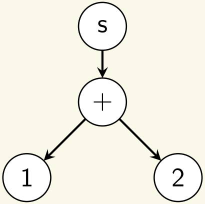
Avoiding inefficiences: sharing

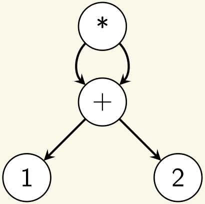
Without sharing





With sharing





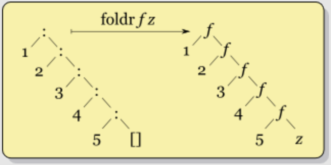
Folds: (yet another) family of higher order functions

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)
```

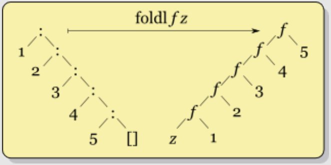


Folds

- *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:[])))
= (((1 + 2) + 3) + 0)
= 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

```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
```

- 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:

```
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 operator `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` (foldList binop z tail)
```

Summary

- Introduced the concept of lazy evaluation
- Saw implementation of `foldr` and `foldl`
- Introduced and used type class *Foldable* to capture computational pattern *reduction*