Session 8: Lazy evaluation

COMP2221: Functional programming

Lawrence Mitchell*

*lawrence.mitchell@durham.ac.uk

Recap

- Saw `type` and `data` declarations
- Discussed difference between sum and product types
- Saw some more on type classes
- `Functor` as a type class for mappable containers
- *Functor laws*
  - `fmap id == id`
  - `fmap (f . g) == fmap f . fmap g`
  - How to prove this for a datatype (inductively, or by exhaustive enumeration [see also exercises]).
- Discussed why one might want to implement type class instances for our data types

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))
```

Thunk slow(..)

True or slow(..)

```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 ask

- 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

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

inc (2*3)
```

Two options for the evaluation order

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

```haskell
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.

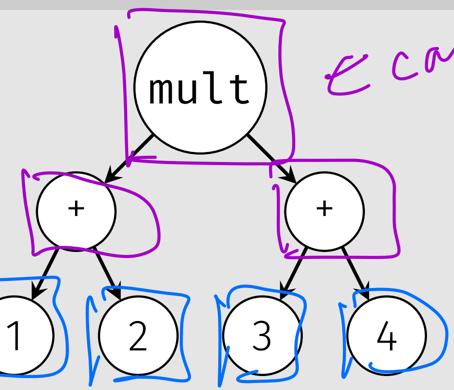
Evaluation strategies II

- 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)
```

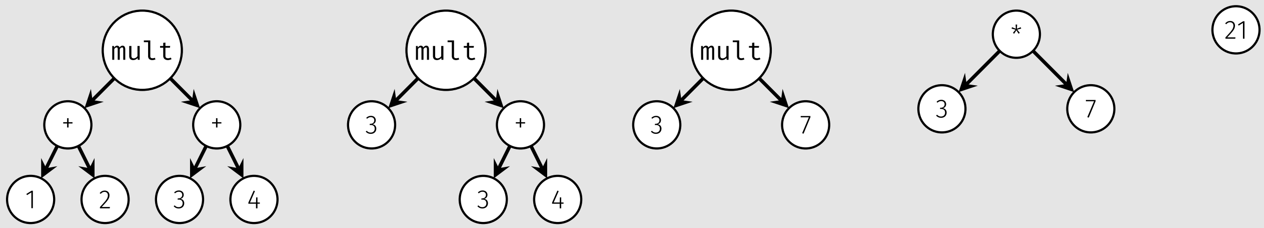


- 1, 2, 3, and 4 are terminal (not reducible) expressions
- (+) and `mult` are reducible expressions.

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"

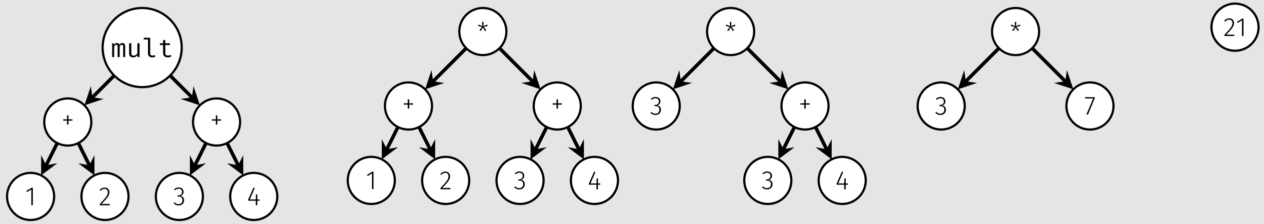
Example



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

```
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

```
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
```

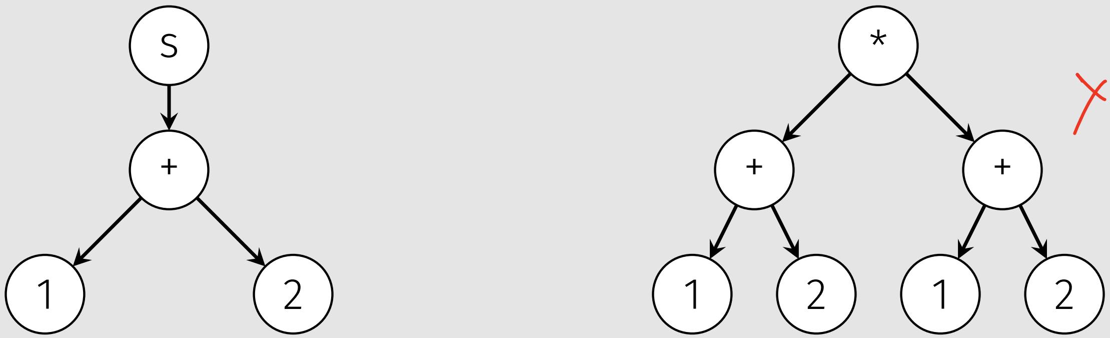
inexpensive! this is expensive :(

let x = (1+2) in square x.

- 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

Building block summary

- Prerequisites: none
- Content
    - Saw some examples of lazily-evaluated (and infinite) expressions in Haskell
    - Introduced different evaluation strategies for expression graphs: innermost and outermost
    - Defined “call-by-name” and “call-by-value” models of evaluation
    - Discussed termination of the evaluation of expressions
    - Saw how Haskell uses “call-by-value” along with argument sharing (treating the expression tree as a graph)
- Expected learning outcomes
    - student can *describe* difference between call-by-name and call-by-value evaluation schemes.
    - student can *explain* how Haskell uses argument sharing to avoid inefficiency when implementing call-by-value.
- Self-study
    - None

Controlling evaluation order

When should I stop evaluating?

How does Haskell evaluate an expression graph?

Definition (Normal form)

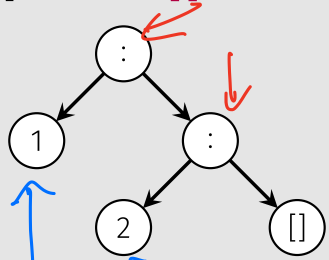
The expression graph contains no redexes, is *finite*, and is *acyclic*.

Data constructors are not reducible, so although they “look” like functions, there is no reduction rule

Example

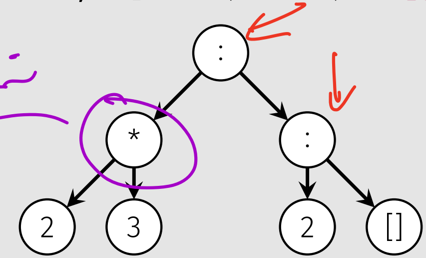
In normal form

```
[1, 2] == 1:2:[]
```



Not in normal form

```
[2 * 3, 2] == (2 * 3):2:[]
```



[1, 1, ...]

NOT in normal form

→ cycle.

How does Haskell evaluate an expression graph? II

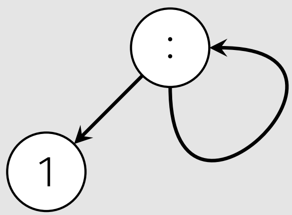
Definition (Weak head normal form (WHNF))

The expression graph is in normal form, *or* the topmost node in a the expression graph is a constructor.

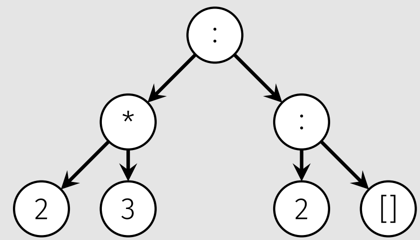
This allows for cycles.

Example

```
ones = 1 : ones
```




```
[2 * 3, 2] == (2 * 3):2:[]
```



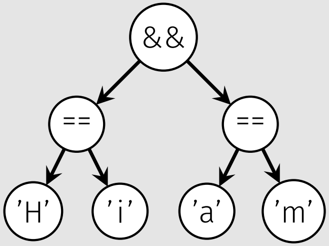
Just (2+4) is in WHNF but not normal form.

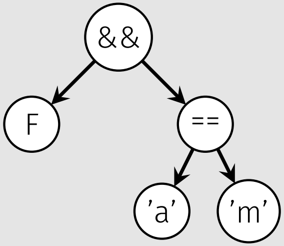
Evaluation rule

- Apply *reduction rules* (functions) *outermost first*
- Evaluate children "left to right"
- Stop when the expression graph is in WHNF
- Function definitions introduce new *reduction rules*

Example

```
('H' == 'i') && ('a' == 'm')
```





Right hand (second) argument is never evaluated. In this way, we get "short circuit" evaluation for free for *all* functions.

Lazy evaluation in strict languages

- All (probably!) languages have one place where they do something akin to lazy evaluation

Boolean expressions

```
#include <stdlib.h>
int blowup(int arg)
{
    abort();
}
int main(int argc, char **argv)
{
    return (argc < 10) || blowup();
}
```

- Boolean expressions do *short circuit* evaluation
- Avoids evaluating unnecessary expressions
- But not possible when assigning to variables.

Lazy evaluation in strict languages II

- Python generators are lazily evaluated

Infinite generator of integers

```
import itertools
def integers():
    i = 0
    while True:
        yield i # yield control to caller
        i = i+1

for p in itertools.takewhile(lambda x: x < 5, integers()):
    print(p)
0
1
2
3
4
```

- Somewhat painful to work with when combining them

Strict functions

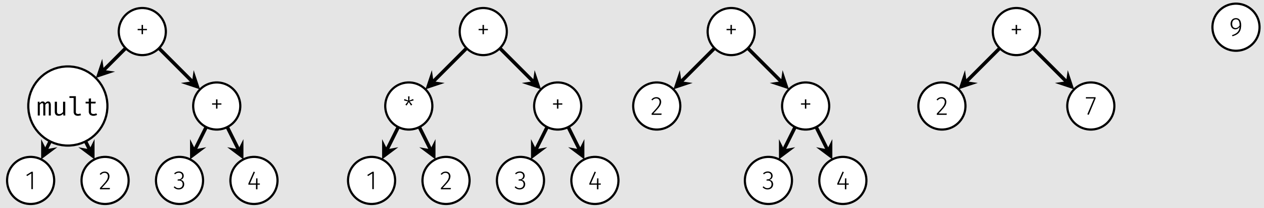
Definition (Strict function)

A function which requires its arguments to be evaluated before being applied.

Even when using outermost evaluation.

- Some functions in Haskell are strict (normally when working with numeric types)

Example



Strict functions: saving space

- Haskell uses lazy evaluation by default
- It also provides a mechanism for *strict* function application, using the operator (`$!`)

```
($!) :: (a -> b) -> a -> b
f $! x -- evaluate x then apply f
```

f x apply f then eval x.

- When using (`$!`), the evaluation of the argument is forced *until* it is in weak head normal form.

Example

```
square $! (1 + 2)
== square $! 3 -- applying +
== square 3 -- applying $!
== 3 * 3 -- applying square
== 9 -- applying *
```

- This allows us to write functions that evaluate as if we had call-by-value semantics, rather than the default call-by-name

Pick up about here next time.

Strict functions: saving space II

- Lazy evaluation can require a large amount of space to generate the expression graph

```
sumwith :: Int -> [Int] -> Int
sumwith v [] = v
sumwith v (x:xs) = sumwith (v+x) xs
Prelude> sumwith 0 [1, 2, 3]
== sumwith (0+1) [2, 3]
== sumwith ((0+1)+2) [3]
== sumwith (((0+1)+2)+3) []
== (((0+1)+2)+3)
== ((1+2)+3)
== (3+3)
== 6
```

a big expression.

- This formulation generates an expression graph of size $\mathcal{O}(n)$ in the length of the input list
- In contrast, strict evaluation always evaluates the summation immediately, using constant space.

Saving space III

- This kind of strict evaluation *can* be useful
- `sumwith` is "just" a tail recursive left fold

```haskell
sumwith = foldl (+) 0
```

- For a strict version, which will use less space, we can use `foldl'`

```haskell
import Data.Foldable
sumwith' = foldl' (+) 0
```

- This can have reasonable time saving for large expressions

Example

```
Prelude> foldl (+) 0 [1..10^7]
2 secs
Prelude> foldl' (+) 0 [1..10^7]
0.25 secs
```

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

Building block summary

- Prerequisites: none
- Content
  - Introduced the evaluation rules for Haskell expressions
  - Defined terms *normal form* and *weak head normal form*
  - Saw some examples of “lazy” evaluation in strict languages
  - Saw how to define strict functions in Haskell using `($!)`
  - Saw an example where strict evaluation can improve runtime (but note this is not a silver bullet)
- Expected learning outcomes
  - student can *explain* Haskell’s evaluation rules for expressions
  - student can provide an *example* of “lazy evaluation” in strict languages
  - student can *write* strict functions in Haskell
- Self-study
  - None