

Functional Programming (H)

Lecture 1: Introduction

Simon Fowler & Jeremy Singer

Semester 1, 2024/2025



University
of Glasgow

Administrivia

- **Course co-ordinators**
 - Simon Fowler simon.fowler@glasgow.ac.uk, SAWB 510c
 - Jeremy Singer jeremy.singer@glasgow.ac.uk, S104 Lilybank Gardens
- **Office Hours:**
 - By e-mail appointment (or drop by)
 - The course will meet three times a week
 - **Lecture Monday, 2-3PM** in 42 Bute Gardens, LT 1115
 - **Lecture Thursday, 12-1PM** in Graham Kerr Building 224 (Main LT)
 - **Labs Friday, 9-noon** in Boyd Orr 720
 - Materials will be posted on Moodle before the lecture / lab

Teams

- You should have been added to the FP Teams channel
 - The link is also on the Moodle page
- This will be our **main method of communication** with you, so please monitor it
- Please use Teams to ask questions or discuss with your peers
 - We have already created channels for technical support and the coursework
 - We're happy to answer DMs / emails, but consider posting (non-sensitive) questions publicly so your peers can learn too

Assessment

- **Written Exam, Semester 2 Diet (60%)**
 - Likely to be lab-based & open-book
- **Moodle Quizzes (5%)**
 - Weekly, due Fridays at 23:59, this week onwards
- **Class Test (10%)**
 - End of W4, administered by Moodle
- **Programming Assignment (25%)**
 - Released W8, with overview lecture

Changes from Last Year

- Last year we made significant changes to the course
 - MOOC no longer used, introduced labs, exercises for every week, rewrote lectures...
- This worked well! So this year, changes are more minor
 - Reworked some of the 'Monads' section of the course in response to student feedback
 - New class test and programming assignment

Moodle

- Course resources will be on **Moodle** (let us know if you're not enrolled!)
- Each week:
 - Lecture slides
 - Lecture recordings
 - Exercise sheet
- Also: coursework handouts and solutions collected through Moodle

Weekly Reflections

- New this year: each week will have an optional ‘weekly reflection’ form on Moodle
- Typical week
 - What did you enjoy most about this week?
 - What did you find most difficult?
 - How did you find the quiz and the lab sheet?
 - Is there something that we could have explained better, or done differently?
 - ...and another silly question, to keep things interesting
- These will be anonymous but will help us refine future weeks

Course Content

- Three main sections (each 3 weeks)
 - **Haskell Fundamentals**
 - Expressions, reduction, functions, equations, datatypes, polymorphism, evaluation strategies, property-based testing
 - **Monads and Side-effects**
 - Purity, IO, do-notation, example monads, monadic parser combinators, monad laws
 - **Advanced Concepts**
 - Monad transformers, functors & applicatives, the lambda calculus & equational reasoning
- Week 8 will be used to introduce the coursework

Intended Learning Outcomes

1. Write simple programs involving elementary Haskell techniques
2. Define new algebraic data types and use recursion to define functions that traverse recursive types, and use common higher-order functions such as map, fold, and filter;
3. Demonstrate understanding of how to express data structures and function interfaces using types, and how to infer types;
4. Understand parametric polymorphism, and ad-hoc polymorphism through typeclasses;
5. Demonstrate understanding of the differences between strict and lazy evaluation, and the tradeoffs of the two approaches;
6. Demonstrate understanding of how to structure programs using monads, how to use the most common standard monads (including IO, Maybe, and State), and how to use a monad transformer;
7. Understand and use common functional abstractions such as functors and applicative functors;
8. Develop substantial functional software applications including external libraries;
9. Use formal methods and property-based testing to reason about the correctness of functional programs;
10. Construct, adapt, and analyse code using standard Haskell tools such as Stack and HUnit

Labs

- Every week is accompanied by an **exercise sheet** that will be released after the Monday lecture
- **Labs** every Friday morning
 - 9-10, 10-11, or 11-noon in Boyd Orr 720 (see MyCampus for your group)
 - Optional but recommended
 - (In practice, feel free to come to any slot / multiple slots – but those scheduled in that group have priority)
- Opportunity to do previous week's exercise sheet and get help from course staff
- Later in the course: opportunity to get help with the coursework

Quizzes

- Every week, there will be **Moodle quiz questions** on the previous week's content
 - Due at 23:59 on Friday
 - Worth 5% of your overall mark
- **Do not submit Good Cause claims** for missed quizzes, e-mail us

Programming Assignment

- In W8 we will release a **programming assignment**, worth 25% of your module grade
- More substantial piece of code, incorporating many concepts you have learnt in the course
- In the past we've done Turtle Graphics & Connect 4, this year should be similarly entertaining
- There will be a lecture introducing the coursework in W8

...hopefully you will enjoy it!

Functional Programming?

```
addOne x = x + 1  
  
map addOne [1,2,3]  
> [2,3,4]
```

```
def addOne(x):  
    return x + 1  
  
xs = []  
for x in range(1,3):  
    xs.append(addOne(x))  
return xs
```

- Functional programming languages: **functions are first-class**
- Imperative languages (e.g., Java, Python) describe a **sequence of steps** to compute a result
- Functional languages describe how to **reduce an expression to a value**
- Often very concise code, much easier to reason about

Functional Programming: Concise!

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    quicksort [y | y <- xs, y <= x] ++
    [x] ++
    quicksort [y | y <- xs, y > x]
```

All concepts above (type classes, lists, polymorphism, list comprehensions, pattern matching, recursion) will be explained during the course

Functional Programming goes Mainstream

- When I started functional programming in 2011, it was still pretty obscure
- Nowadays, FP is everywhere!
 - Java Streams, C++ Lambdas, JavaScript / TypeScript...
- Massive industry interest



Jane Street

ORACLE

 Meta

Bloomberg®



Haskell

Haskell (/ˈhæskəl/) is a general-purpose, statically-typed, purely functional programming language with type inference and lazy evaluation.

(Wikipedia entry for Haskell)

Functional language with separation between pure and side-effecting code

Usable for a wide variety of programming tasks

Type system catches errors before a program is run

Computations performed on-demand

Types can be deduced from program code; do not need to be specified explicitly





- Mature, industry-grade functional language
- Has directly inspired other languages (e.g., Idris, Agda, Elm...)
- **Opinionated:** You will learn a lot by understanding purity, laziness, static typing
- (Historical ties to UofG: see MOOC for more details)

Resources (Useful, not required)

- *Programming in Haskell*. Graham Hutton, 2016. Cambridge University Press.
- *Learn you a Haskell for Great Good!* Miran Lipovaca, <http://learnyouahaskell.com/>
- *Thinking Functionally in Haskell*. Richard Bird, 2014. Cambridge University Press.
- Links to eBooks on the Moodle page

Expressions and Statements

Expressions vs. Statements

Statement:

An instruction / computation step

Doesn't return anything

Expression:

A term in the language that eventually reduces to a **value** (e.g., a string, integer, ...)

Can be contained within a statement

```
def addOne(x):  
    return x + 1  
  
xs = []  
for x in range(1,3):  
    xs.append(addOne(x))  
return xs
```

**In a functional language,
everything is an expression!**

Conditional Statements as Expressions

if 2 < 3 **then**

“two is less than three”

else

“three is less than two”

Important: Since a Haskell conditional is an expression, it must have an **else** clause

Statements as Expressions

```
let msg =  
  if 2 < 3 then  
    "two is less than three"  
  else  
    "three is less than two"  
in  
  "Result: " ++ msg
```

Types

- Each expression has a **type**, which classify values

5 :: Int

True :: Bool

False :: Bool

(1, "Hello") :: (Int, String)

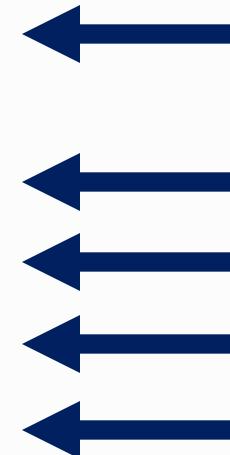
- Types can rule out nonsensical expressions (e.g., 5 + True)
- Haskell can **infer** a type for most expressions, but it is good practice to add in a type signature for top-level functions

Reduction

Evaluation in Imperative vs. Functional Languages

- Imperative language:
Program Counter + Call Stack + State
- We record our **current position** in the program
- Statements can alter that position
- Variable assignments alter some **store**

```
def addOne(x):  
    return x + 1  
  
xs = []  
for x in range(1,3):  
    xs.append(addOne(x))  
return xs
```



Compare with the functional version...

map addOne [1,2,3]

→ [addOne 1, addOne 2, addOne 3]

→ [2, addOne 2, addOne 3]

→ [2, 3, addOne 3]

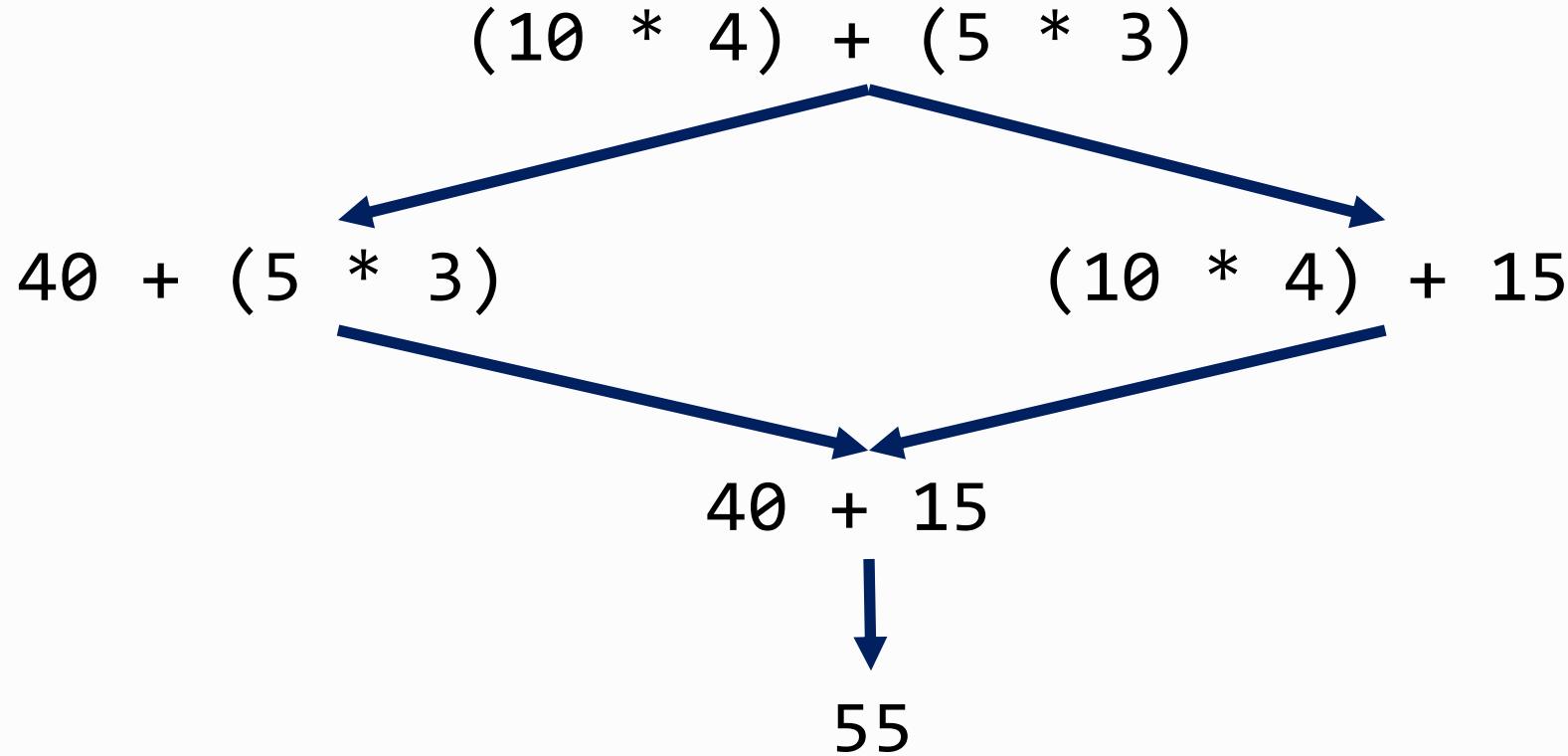
→ [2, 3, 4]

Reduction

$$\begin{aligned}1 + (2 + (3 + 4)) \\ \rightarrow 1 + (2 + 7) \\ \rightarrow 1 + 9 \\ \rightarrow 10\end{aligned}$$

Functional reduction: reduce complex
expressions to a **value**

Reduction



Church-Rosser property: reductions can be performed in **any order**

Reduction

- **Key point:** Reduction takes an **expression** and eventually produces a **value**
- **Church-Rosser:** evaluation can be in **any order**
 - In the context of abstract rewriting systems, reduction is **confluent**
 - Useful for **equational reasoning** and **functional parallelism**
- Treated in (much) more depth in Theory of Computation (and Programming Languages, this year)

Wrapping up

- Today
 - Course Details
 - Lectures, labs, coursework...
 - Dipping our toes in the water
 - Expressions vs. statements, Types
 - Reduction & Church Rosser
- Next time
 - Functions, lists, and polymorphism
- Over to you
 - Please make sure you're on Teams & Moodle
 - If you're using a laptop, please install Haskell (details on Moodle)

Questions?

Functional Programming (H)

Lecture 2: Functions & Lists

Simon Fowler & Jeremy Singer

Semester 1, 2024/2025



University
of Glasgow

Today

- Last time
 - Course intro
 - Expressions & reduction
- Lecture today will introduce core concepts behind FP and Haskell
 - Tuples
 - Functions and Equations
 - Lists and Sequences
 - Haskell building blocks: conditionals, bindings, case statements
- Please ask questions!

Tuples

- A **tuple** is an ordered sequence of expressions, with a known length

(1, 2, “hello”) :: (Int, Int, String)

(1.0, 1) :: (Float, Int)

() :: ()

- Very useful, e.g., for returning multiple values from functions

Deconstructing Tuples

- We can deconstruct a pair (a tuple of length 2) by using the `fst` and `snd` functions

`fst (1, "hello") → 1`

`snd (1, "hello") → "hello"`

- We can also deconstruct by **pattern matching** (more on this later in the lecture & in Jeremy's lectures next week)

`let (x, y) = (1, "hello") in x → 1`

Functions & Equations

Functions

```
\name -> “Hello, ” ++ name  
      \n -> n + 5
```

- A **function** maps a value to another value
- Function definitions have **parameters**, which act as placeholders for arguments when the function is applied
- In Haskell, a function is **first-class**: we can let-bind it, return it from another function, pass it as an argument...
- Functions without a name (like the above) are known as **anonymous**
- The backslash should be read “lambda” after the Greek letter λ

Functions with Multiple Parameters

- A function maps a single argument to a result
- What about functions with multiple arguments?
- We could use a tuple...

$$\lambda(n1, n2) \rightarrow n1 + n2$$

- A little annoying: have to construct and deconstruct a tuple whenever calling the function
- Can we do better?

Currying

- In practice, functions with multiple arguments are defined by **nesting** multiple functions

$$\lambda n_1 \rightarrow (\lambda n_2 \rightarrow n_1 + n_2)$$

Int -> Int

$$\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$$

- *"If you give me an Int, I will give you a function from Int -> Int"*

Equations

- An **equation** gives meaning to a name

`favouriteNumber = 5`

`add = \x -> \y -> x + y`

- Equations can be used to **define functions** more concisely

`add x y = x + y`

`min x y = if x < y then x else y`

- The left-hand-side **cannot contain any computations**

`(x + 1) = 5` 

Function Equations

- A **function** takes some arguments, performs some computation, and returns a **result**
- Haskell has many useful functions defined in the **Prelude**

```
min :: Int -> Int -> Int  
min x y = if x < y then x else y
```

Left hand side of the equation:
function name, and arguments

Type signature: take two integers,
produce an integer

Right hand side of the equation:
implementation

Function Application

```
addOne x = x + 1
```

```
addOne 5
```

```
addOne 5 * 2
```

```
addOne (5 * 2)
```

- Expressions can contain function calls (like addOne)
- In Haskell (and most other functional languages) you do **not** need to parenthesise arguments to functions
- Function application binds tightest:
 $f x * 2$ means $(f x) * 2$

Partial Application

- An advantage of currying is **partial application**

`add x y = x + y`

`addFive = add 5`

- This allows us to specialize a function without needing to rewrite or duplicate the logic

Function Composition

We can **compose** two functions using the function composition operator

`(.) :: (b -> c) -> (a -> b) -> (a -> c)`

`f . g = \x -> f(g(x))`

Function composition is used to write code in **point-free** style, which tries to avoid introducing variable names where possible

`show . add10`

vs.

`\x -> show (add10 x)`

Aside: Parentheses

$$\begin{aligned}1 + 2 * 3 \\= \\1 + (2 * 3) \\= \\(1 + (2 * 3))\end{aligned}$$

$$\begin{array}{r} \cancel{1} * -1 \\ 1 * (-1) \end{array}$$

- Parentheses can keep things neat and avoid ambiguity
- They are needed around things like negative numbers (to disambiguate between subtraction)
- As a matter of style, use only where necessary

Equations are **not** assignments

- In an imperative language, we might write something like
 $x := x + 1$, or $x++$
 - This **modifies** the value referred to by variable x
- Consider $x = x + 1$ in Haskell
 - This is **not** the same as $x := x + 1$ or $x++$
 - Tries to define x as the successor of x !
 - Haskell will try to calculate it, though...
- Reassignment is **not permitted**
 - It doesn't make sense for x to be defined as **both** 5 and 6

Never destroy old values, just compute new ones!

Parametric Polymorphism (Briefly)

- Polymorphism: “Many forms” in Greek

$\lambda x \rightarrow \lambda y \rightarrow x :: \text{Int} \rightarrow \text{Bool} \rightarrow \text{Int}$

$\lambda x \rightarrow \lambda y \rightarrow x :: \text{String} \rightarrow \text{Int} \rightarrow \text{String}$

$\lambda x \rightarrow \lambda y \rightarrow x :: \text{Float} \rightarrow \text{Bool} \rightarrow \text{Float}$

Same function, many different types!

Type Variables

- Polymorphic functions: have **type variables** to stand for types

$$\lambda x \rightarrow \lambda y \rightarrow x :: a \rightarrow b \rightarrow a$$
$$\lambda x \rightarrow x :: a \rightarrow a$$

- Can also have type variables in types (e.g., tuples and lists)

$$\text{reverse} :: [a] \rightarrow [a]$$
$$\lambda (x, y) \rightarrow (y, x) :: (a, b) \rightarrow (b, a)$$

Lists & Sequences

Lists

- A list is an **ordered sequence of values of the same type**
`[1,2,3] :: [Int]`
`["Hello", "FP", "Students"] :: [String]`
- Haskell supports a concise notation for creating ordered lists
 - `[1..10]`
 - `['a'..'z']`
 - `[1..]`
- Note that these are constructed **lazily**
 - You can construct an infinite list and only compute what you need
 - More on this later in the course

List Comprehensions

- Remember set builder notation from Algorithmic Foundations 2?

$$doubleEvens = \{ 2x \mid x \in \mathbb{Z}^+, x \bmod 2 = 0 \}$$

- **List comprehension** notation allows the same construction:

```
doubleEvens = [ 2 * x | x <- [1..], x `mod` 2 == 0 ]
```

Body Generators Condition

The diagram illustrates the structure of a list comprehension. It shows the expression `doubleEvens = [2 * x | x <- [1..], x `mod` 2 == 0]` with three parts highlighted by yellow boxes and labeled below with arrows: "Body" points to `2 * x`, "Generators" points to `x <- [1..]`, and "Condition" points to `x `mod` 2 == 0`.

Accessing Lists

`head :: [a] -> a`

`tail :: [a] -> [a]`

`(!!) :: [a] -> Int -> a`

- Note: Only use these if you know what you are doing (e.g., definitely know the number of elements in the list)
 - List indexing, head, tail **all potentially undefined**
 - Often better to use **pattern matching** (described later)

Zipping Lists

$$\text{zip} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \begin{bmatrix} \text{“One”} \\ \text{“Two”} \\ \text{“Three”} \\ \text{“Four”} \\ \text{“Five”} \end{bmatrix} = \begin{bmatrix} (1, \text{“One”}) \\ (2, \text{“Two”}) \\ (3, \text{“Three”}) \\ (4, \text{“Four”}) \\ (5, \text{“Five”}) \end{bmatrix}$$

`zip :: [a] -> [b] -> [(a, b)]`

Zipping Lists

$$\text{zip} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \begin{bmatrix} \text{“One”} \\ \text{“Two”} \\ \text{“Three”} \end{bmatrix} = \begin{bmatrix} (1, \text{“One”}) \\ (2, \text{“Two”}) \\ (3, \text{“Three”}) \end{bmatrix}$$

Zipped list: as long as the **shortest** of the two lists

Question: Is this the only / best design?

Aside: Dependent Types (Non-examinable)

This is an occasion where our type signature is not specific enough to let us know the function's behaviour.

Dependent type systems allow values within types, so we can be much more specific.

```
zipDep :: Vect n a -> Vect n b -> Vect n (a, b)
```

`Vect n a`: List of length n containing values of type a

Haskell has some support for dependent types (but we won't use it in this course)

Zipping with a Custom Function

zipWith (*)

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \quad \begin{bmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 6 \\ 8 \\ 10 \end{bmatrix}$$

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

Question: When would you use a tuple, and when would you use a list?

- Use a **tuple** when:
 - You know the number of values you are storing
 - The types of the values you are storing are different
- Use a **list** when:
 - You don't necessarily know the number of elements in advance
 - You have an ordered sequence of values of the same type
 - You want to operate uniformly over your data
- Another Question: which is the better type signature?
 - `pythagTriples :: Int -> [(Int, Int, Int)]`
 - `pythagTriples :: Int -> [[Int]]`

Haskell Building Blocks

So, how do we write programs?

For a quadratic formula:

$$ax^2 + bx + c = 0$$

Calculate x using:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
def roots(a,b,c):
    det2 = b * b - 4 * a * c
    det = sqrt(det2)
    rootp = (-b + det)/ a / 2
    rootm = (-b - det)/ a / 2
    return [rootm,rootp]
```

Let bindings

```
roots a b c =  
let  
    det2 = b * b - 4 * a * c  
    det = sqrt det2  
    rootp = (-b + det)/ a / 2  
    rootm = (-b - det)/ a / 2
```

```
in  
[rootm, rootp]
```

Can use previously-bound patterns

Body has access to all variables bound in let clause

Equivalently...

```
roots a b c =
  let det2 = b * b - 4 * a * c in
  let det = sqrt det2 in
  let rootp = (-b + det)/ a / 2 in
  let rootm = (-b - det)/ a / 2 in
  [rootm, rootp]
```

Note: each **let** needs a **continuation** (the rest of the computation).
This is because it is not a statement!

Where bindings

```
roots a b c = [rootm, rootp]
```

where

```
det2 = b * b - 4 * a * c
```

```
det = sqrt det2
```

```
rootp = (-b + det)/ a / 2
```

```
rootm = (-b - det)/ a / 2
```

- Equivalent to **let** bindings, but bindings go *after* the function body
- Often a matter of taste whether to use **where** or **let**

Case expressions

In languages like C / Java, we have a **switch / case** statement:

```
switch (num) {  
    case 1: return "one"  
    case 2: return "two"  
    case 3: return "three"  
    default:  
        return "something else"  
}
```

Guards

- Sometimes we want to do different things based our input value
- We could do this via if-else chains...

```
gradeFromGPA :: Int -> String
gradeFromGPA gpa =
  if gpa >= 18
  then "A" else
  if gpa >= 15
  then "B" else
  if gpa >= 12
  then "C" else
  "below C"
```

- In practice, Haskell's *guard* notation is often cleaner: each is 'guarded' by a Boolean predicate
- 'otherwise' is just a synonym for True

```
gradeFromGPA :: Int -> String
gradeFromGPA gpa
| gpa >= 18 = "A"
| gpa >= 15 = "B"
| gpa >= 12 = "C"
| otherwise = "below C"
```

Wrapping Up

- **Today**
 - Fundamental concepts: expressions, types, lists, evaluation, function application, equations, parametric polymorphism, guards
- **Next up**
 - Next time, we'll learn algebraic data types, pattern matching, and recursion (very important!)
- **Over to you**
 - Say hi on the Teams channel
 - Have a go at the exercise sheet: help available in lab on Friday (9-noon) if you need it
 - Install GHC if you haven't already, and please do the weekly reflections
- **Questions?**

Functional Programming (H)

Lecture 3: Recursion and Algebraic Datatypes

Simon Fowler & Jeremy Singer

Semester 1, 2024/2025
v2 7th October 2024

Join at menti.com | use code 1794 1041



University
of Glasgow

Today

- Last time
 - Whistle-stop tour of Haskell features: tuples, functions, lists and list comprehensions, Haskell building blocks
- Today
 - Recursion (+ mutual recursion / tail recursion)
 - Algebraic datatypes: definition, pattern matching
 - (Probably the most important lecture in the “fundamentals” part of the course!)
- **By popular demand, quizzes + reflections are now open until Monday at 11:59AM**

W2 Reflections

- Thank you for doing the W2 reflections! I read them all. Some themes:
- Many of you enjoyed the challenge of learning a new paradigm and enjoyed the lectures / lab --- great!
- Syntax / function composition caused the most headaches.
 - Syntax: it gets better over time 😊
 - Function composition: think of it as making a new function based on two existing functions
- In terms of suggestions:
 - Lecture recordings up earlier: these are actually uploaded pretty much straight after each lecture (look for Echo360 on Moodle)
 - Case expressions / guards a bit rushed (will recap today)
 - Save live coding examples and put on Moodle – good idea!

W2 Reflections: Dad Jokes

(Some of the ones that are safe to put on slides...)

- What do you call a man from Glasgow who's lost his dog? Douglas!
- What two crows always stick together? Velcrows!
- What do you call a Scottish man halfway through his door? Hamish!
- I asked my dog how his day was. He said it was ruff.
- Why do seagulls fly over the sea? If they flew over the bay, they would be bagels.
- Why did the man fall down the well? He couldn't see that well...

Mentimeter

- I have been asked whether I can have a Slido-style way of asking questions
 - I think this is a good idea!
 - Also feel free to still ask questions in person though.
- I'm trying out Mentimeter as a way of doing this, as well as playing around with some of the other features...
 - This is my first time doing this, so please bear with me 😊

Join at menti.com | use code 1794 1041



Case expressions

In languages like C / Java, we have a **switch / case** statement:

```
switch (num) {  
    case 1: return “one”  
    case 2: return “two”  
    case 3: return “three”  
    default:  
        return “something else”  
}
```

Case expressions

Haskell has something similar:

```
case num of
  1 -> "one"
  2 -> "two"
  3 -> "three"
  _ -> "something else"
```

```
case list of
  [] -> "empty"
  [x] -> "one element"
  _ -> "more elements"
```

The idea is to allow pattern matching *within* a definition

Guards

- Sometimes we want to do different things based our input value
- We could do this via if-else chains...

```
gradeFromGPA :: Int -> String
gradeFromGPA gpa =
  if gpa >= 18
  then "A" else
  if gpa >= 15
  then "B" else
  if gpa >= 12
  then "C" else
  "below C"
```

- In practice, Haskell's *guard* notation is often cleaner: each is 'guarded' by a Boolean predicate
- 'otherwise' is just a synonym for True

```
gradeFromGPA :: Int -> String
gradeFromGPA gpa
| gpa >= 18 = "A"
| gpa >= 15 = "B"
| gpa >= 12 = "C"
| otherwise = "below C"
```

Recursion

to understand recursion,
you first have to
understand recursion



How are lists defined?

So far, we have considered lists to be **primitive data types**

`[1, 2, 3] :: [Int]`

In fact, lists are **inductively-defined data structures**

`[] :: [a]`

`(:) :: a -> [a] -> [a]`

So, `[1,2,3]` is actually **syntactic sugar** for `1 : (2 : (3 : []))`

This allows us to write **recursive functions** over lists

Recursion vs. Iteration

Python

```
def listSum(xs):  
    res = 0  
    for x in xs:  
        res = res + x  
    return res
```

Haskell

```
listSum :: [Int] -> Int  
listSum [] = 0  
listSum (x:xs) =  
    x + (listSum xs)
```

We pattern match on the different ways of constructing a list

- In **imperative** languages, we typically rely on **iteration** when working with collections of values
 - This is because we often perform **statements** using data in the collections
- In **functional** languages, **recursion** is our basic building block
 - This is because we often want to use the data to **construct a new result expression**

Designing Recursive Functions

Base case: sum of elements in an empty list is 0

```
listSum :: [Int] -> Int
listSum [] = 0
listSum (x:xs) =
  x + (listSum xs)
```



Recursive case: add current element to sum of the rest of the list



- Think about the **structure** you are defining recursion on
 - Integers? Lists?
- Write a **base case**: you will want (at least pure!) functions to terminate
- Write a **recursive or inductive** case which calls the same function with an argument which **converges to the base case**

Name a recursive function!

28 responses



greatest common divisor
post order tree traversal
binary search a word
heapsort bubblesort
 fibonacci
factorial hugh quicksort
 bogosort map russian dolls
 bob another word in order tree
 backtrack
 merge sort
pre order tree traversal

Example: Factorial

```
fac :: Int -> Int
fac 0 = 1
fac n =
  if n < 0 then
    error "bad argument"
  else
    n * (fac (n - 1))
```

- We can recurse on integers, too – just need to ensure we converge to a base case
- Need negativity check to ensure we do not “overshoot” base case

Example: Fibonacci

```
fib :: Int -> Int
fib 0 = 0
fib n | n < 3 = 1
      | otherwise =
        (fib (n - 1)) + (fib (n - 2))
```

- Fibonacci sequence: $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$
- 0,1,1,2,3,5,8,13,21,34,55, ...
- Direct implementation very inefficient...

Example: Better Fibonacci

```
betterFib :: Int -> Int
betterFib 0 = 0
betterFib goal =
    if goal < 2 then 1
    else betterFib' 0 1 2
where
    betterFib' prev1 prev2 idx =
        if idx == goal then prev1 + prev2
        else
            betterFib' prev2 (prev1 + prev2) (idx + 1)
```

- Idea: calculate running sums as we go
- Base case: return sum of previous two running sums when we work up to goal

Tail Recursion

```
listSum :: [Int] -> Int
listSum xs = listSum' xs 0
where
    listSum' [] i = i
    listSum' (x:xs) i =
        listSum' xs (i+x)
```

- What about *stack overflow*?
- Worry not: Haskell uses **tail call optimisation**
- All **tail calls** (where a call is the last part of an expression) can be implemented using **constant stack space**
- Note: our naïve Fibonacci example is **not** tail-recursive, but our “better” Fibonacci is

Mutual Recursion

```
tick :: Int -> String
tick 0 = ""
tick n =
    "tick " ++ (tock (n-1))

tock :: Int -> String
tock 0 = ""
tock n =
    "tock " ++ (tick (n-1))
```

- Sometimes, we want to write **mutually recursive** functions, which call each other
- Haskell allows us to do that, since all other definitions are in scope
- In other functional languages, sometimes you need to mark these explicitly

Algebraic Datatypes

Defining Data Types

Data keyword: states that we are defining a new type

```
data Season =  
    Spring | Summer | Autumn | Winter
```

Name of the new type

Data constructors: different ways of creating values of this type

- This is a *sum type*, with alternative values (see enum in C or Java)
- So far we have mainly used built-in types
 - Int, String, Bool, [a]...
- It is often useful to define our own data types
- All of Spring, Summer, Autumn, Winter have type Season

Defining Data Types

Data keyword: states that we are defining a new type

Name of the new type

Data constructors: to create values of the type

```
data UofGGrade = Grade Char Int
```

- This is a *product type*, with a combination of values
- We could restrict to legal grades by
 - enumerating allowed letters A .. H
 - or adding validation functions

Richer Data Types with Content

```
data Suit =  
    Hearts | Diamonds | Clubs | Spades
```

```
data Card = King Suit | Queen Suit | Jack  
Suit | Ace Suit | Number Suit Int
```

- Each data constructor can also have some **associated data**
- Good for modelling, but need some behaviours (type classes, next week!)

Pattern Matching

```
showCard :: Card -> String  
showCard (King _) = "K"  
showCard (Queen _) = "Q"  
showCard (Number _ n) = (show n)
```

pattern match
Card values to
convert to
String

- When working with a data type, we often need to **pattern match** to see which data constructor was used to construct the value
- Notice the need for (brackets) around compound values
- If pattern matching is incomplete, GHC gives a warning but not an error
 - There will be an error at runtime if we encounter an unmatchable pattern
- Underscore `_` means “don’t care” - match anything. We can also bind a value to a name in the pattern match - like `n` in the final line of the `showCard` function above
- Pattern matching can be done using **equations** (as here) or a **case** expression

Recursive Data Types: Binary Trees

```
data Tree =  
    Leaf | Node Int Tree Tree
```

- Aka "inductively defined" data type
 - If you have done Algorithmic Foundations 2, you may be familiar with recursive definitions
- This is a binary tree with an `Int` payload value at each non-leaf node

Parameterised Data Types: Binary Trees

```
data BinaryTree a =  
    Leaf | Node a (BinaryTree a) (BinaryTree a)
```

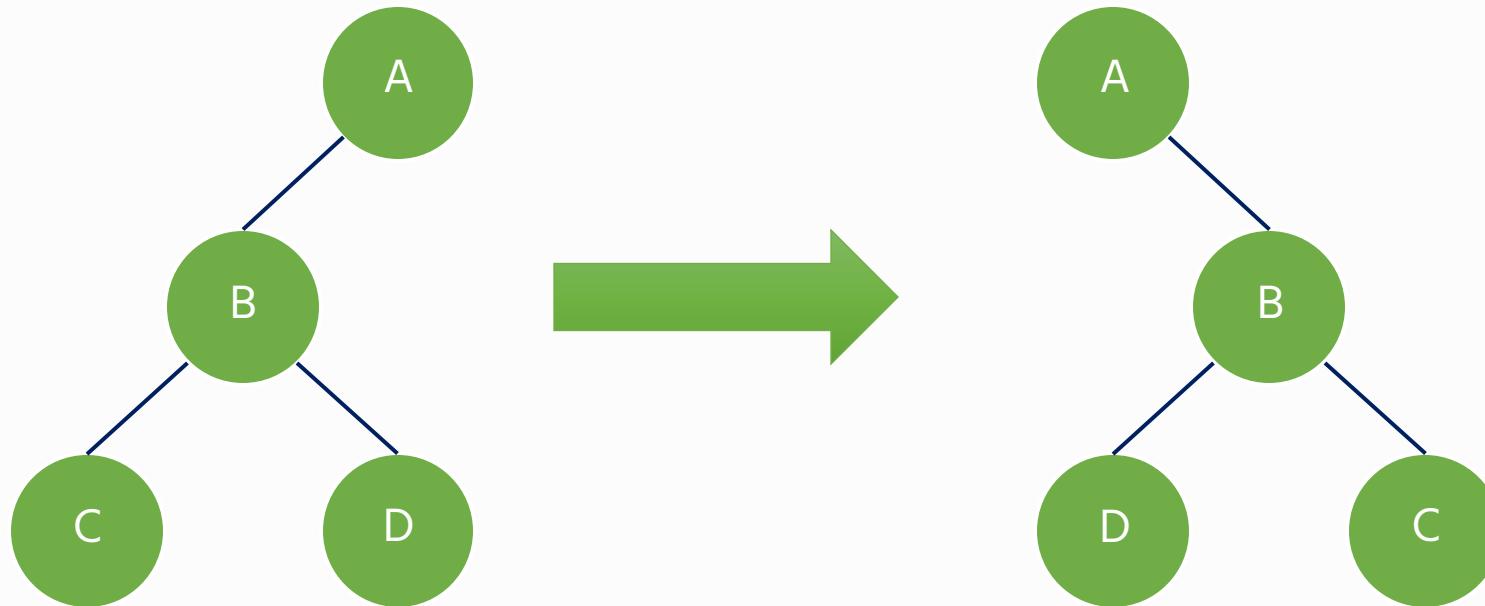
- We can **parameterise** a data type by putting a type variable on the left-hand side of the data definition
- We have already seen an example of this: the list type [a] can contain any type of values
- In the above example, we have a tree which can contain any type of value

```
data Tree =
    Leaf | Node Int Tree Tree

treeSum :: Tree -> Int
-- ^ recursively traverse binary
--   tree and compute sum

treeSum Leaf = 0
treeSum (Node x left right) =
    x + treeSum left + treeSum right
```

Example: Inverting a Binary Tree



```
invert :: BinaryTree a -> BinaryTree a
invert Leaf = Leaf
invert (Node x t1 t2) =
  Node x (invert t2) (invert t1)
```

Challenge

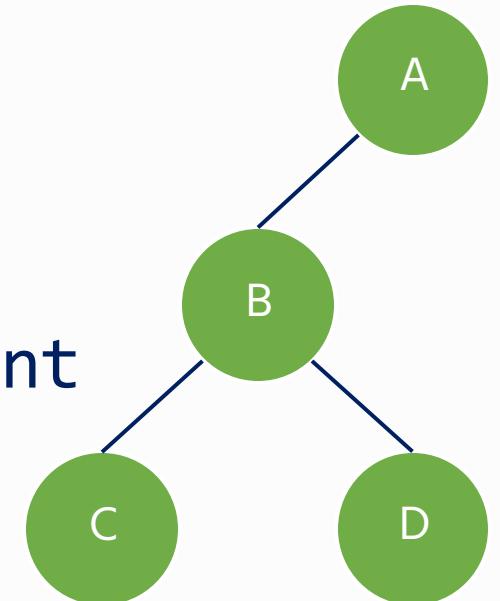
- Write a `treeDepth` function...

`treeDepth :: Tree -> Int`

...which works out maximum depth of tree from root to furthest leaf. For example, the depth of the tree on the right is 3.

- Follow the same pattern as `treeSum` above
- Can use helper function `max :: Int -> Int -> Int`

```
treeSum :: Tree -> Int
-- ^ recursively traverse binary
--   tree and compute sum
treeSum Leaf = 0
treeSum (Node x left right) =
  x + treeSum left + treeSum right
```



```
treeDepth :: Tree -> Int
-- ^ recursively traverse binary
--   tree and compute depth
treeDepth Leaf = 0
treeDepth (Node x left right) =
  1 + max (treeDepth left) (treeDepth right)
```

example of
parametric
polymorphism,
a is type variable

The Maybe Type

```
data Maybe a = Just a | Nothing  
  
safeHead :: [a] -> Maybe a  
safeHead []      = Nothing  
safeHead (x : _) = Just x
```

- What about potentially-failing computations?
- The Maybe a type says that either there is some data (Just a), or there isn't (Nothing)
- We must pattern match to find out whether the computation failed so that we can handle all eventualities
- Like Option in Rust, Scala, Java

Maybe Emoji

```
emote :: String -> Maybe String
```

```
emote "happy" = Just "😊"
```

```
emote "sad" = Just "😢"
```

```
emote "realised FP quizzes were due a week early on Moodle"  
      = Just "呜"
```

```
emote _ = Nothing
```

Answers to Menti Questions

- You said || DOESN'T do short circuit OR in Haskell, can you work with binary at all in haskell?
 - You can certainly work with binary – see <https://hackage.haskell.org/package/binary>
 - “|” on its own doesn’t do short-circuit OR, but “||” does:
 $(||) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$
True || _ = True
False || _ x = x
- Does ‘cons’ (:) stand for something?
 - I looked this up – apparently it stands for “**construct**”. It came from Lisp originally. For this course you just need to know that it’s the way that you make a new list from a ‘head’ value and the remainder of a list, e.g.,
 $1 : (2 : (3 : [])) = [1, 2, 3]$.

Answers to Menti Questions (2)

- What is deriving (Show)?
 - I'll come onto this when we talk about typeclasses next week – but it essentially allows GHCi to print a data type out at the console. If I hadn't put "deriving (Show)" at the end of the Season definition I'd have got an error:

```
ghci> data Season = Spring | Summer | Autumn | Winter
ghci> Spring

<interactive>:2:1: error:
 * No instance for (Show Season) arising from a use of `print'
```
- Does the sequence of definition matter? Do we need to define base case before inductive case?
 - No: the order of data constructors in a definition doesn't matter. However, when you're doing pattern matching, each equation will be tried in order.

Answers to Menti Questions (3)

- Are lists in Haskell just like linked lists in general?
 - Essentially! The difference is that in other languages you'd need to work with the pointers directly (e.g., in C you'd need a pointer to the next cell). Haskell allows us to be more concise in our definition and pattern matching makes working with lists quite elegant.
- Is the prime notation ('') you've used a convention for if you are calling the function recursively within the function?
 - / What do the apostrophes mean?
 - Using apostrophes is legal in Haskell variable names (e.g., `listSum'`). It's weird the first time you see it! It's typically used when you're creating a related function (e.g., using a different definition, or used as a helper function).

Answers to Menti Questions (4)

- Is the type name like a class and the data constructor like the constructor in a Java class?
 - Almost, but it would be difficult to model it using constructors alone in Java I think. It's more like an inheritance hierarchy. For seasons you'd have:

```
abstract class Season { }
public class Spring extends Season { public Spring() {} }
public class Summer extends Season { public Summer() {} }
public class Autumn extends Season { public Autumn() {} }
public class Winter extends Season { public Winter() {} }
```
- Can we have an infinitely running program in Haskell - how would we implement this without loops?
 - Yes, certainly, but it wouldn't make sense for a **pure** function to run infinitely. We'll talk about this more when we talk about side effects later on in the course, but we can always write an infinitely-running program using recursion. There are also functions like 'forever' that will run a function forever.

Answers to Menti Questions (5)

- What exactly are " side effects" and how do you define them?
 - A “side effect” is something that isn’t pure computation: for example, printing to the console, receiving from a network socket, mutating some memory. At the moment we’re concentrating on pure computations but Section 2 will talk about side effects in much more depth.
- Why do we bother using cons if the result is just a list anyway?
 - Indeed [1,2,3] and 1: (2 : (3:[])) mean the same. But the former is just a shorthand for the latter – by treating lists as being built up piece by piece from the empty list, we can deconstruct them when we write recursive functions (i.e., by having [] and (x:xs) cases). We also might want to create a new list programmatically too.

Answers to Menti Questions (6)

- Is tail recursion just when the recursive call is the last part of the expression?
 - Mostly, yes – the key point is that the function doesn't need the result of the recursive call. So...
 - `listSum (x:xs) = x + (listSum xs)`
 - Even though `listSum xs` **looks** like it's the last part of the expression, the result is needed by `+` (you can see this by writing the addition as `add x (listSum xs)`).
 - In contrast, the result isn't needed by the `+` operator in the tail-recursive version
 - `listSum' (x:xs) i = listSum' xs (i+x)`

Wrapping Up

- **Today**
 - Recursion and Algebraic Data Types
 - (A very important lecture ☺)
- **Next up on Thursday**
 - Higher-order functions and property-based testing
- **Questions?**

Functional Programming (H)

Lecture 4: Higher-Order Functions & Property-Based Testing

Simon Fowler & Jeremy Singer

Semester 1, 2024/2025
v2 12th October 2024



<https://sli.do>, code 2834 621



University
of Glasgow

Today

- Last time:
 - Recursion and algebraic data types
- Today we will:
 - Recognise some common functional **design patterns** involving higher-order functions
 - Use the **QuickCheck** library for testing basic properties of functions

Haskell libraries

- There are thousands ... see <https://hackage.haskell.org>
- Varying in quality and state of maintenance
 - Just because something is on Hackage, doesn't mean it is any good 😊
- Today we will explore **QuickCheck**
- Package management
 - You can browse for packages at Hackage online
 - You can search for packages with **cabal** on the terminal
 - You can install packages with **cabal** (globally) or **stack** (per-project)

Dependencies for this week

- `cabal update`
- `cabal install --lib QuickCheck`
- This should download and install appropriate libraries / dependencies on your local Haskell system (system-wide)
- Check it works -- in GHCi write:
`import Test.QuickCheck`

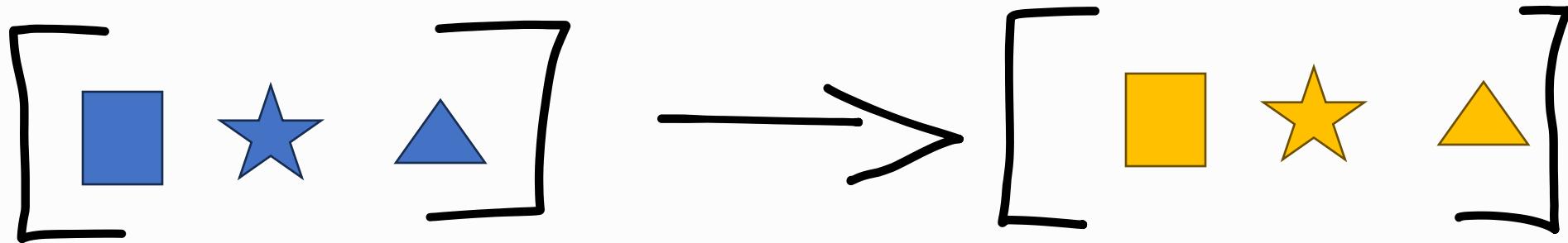
Higher-Order Functions

Higher-order functions

- A higher-order function is a function which **takes another function as an argument**
- You may have seen several of these already:
 - `map :: (a -> b) -> [a] -> [b]`
 - `filter :: (a -> Bool) -> [a] -> [a]`
 - `twice :: (a -> a) -> a -> a`
- It is good practice to use HOFs where applicable rather than hand-roll recursive functions yourself
 - Concentrate on **application logic** rather than recursive boilerplate
- Today we will look at some common HOFs on lists

Map

map
makeYellow



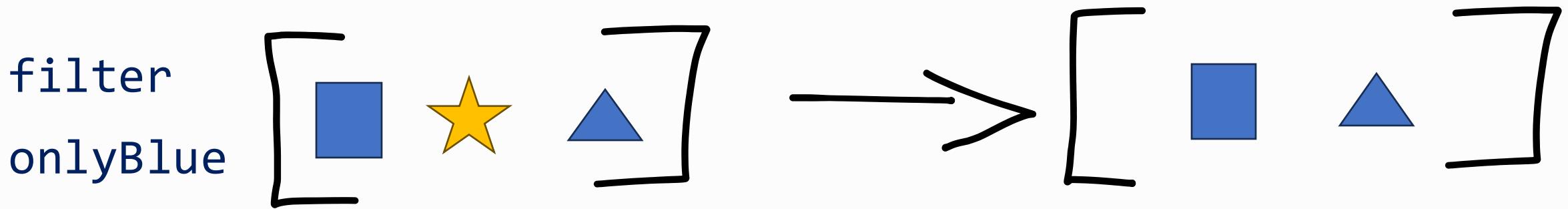
- A **map** applies a function to every element in a list
- Defined recursively by **building up a new list** with function applied to each element
- **Map fusion:** $\text{map } f \ (\text{map } g \ xs) \Leftrightarrow \text{map } (f \ . \ g) \ xs$

Map

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

- A map applies a function to every element in a list
- Defined recursively by **building up a new list** with function applied to each element
- **Map fusion:** $\text{map } f \ (\text{map } g \ xs) \Leftrightarrow \text{map } (f \ . \ g) \ xs$

Filter



- **Filter** retains every element which satisfies a **predicate**
- Only prepend the element if the predicate evaluates to True

Filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x : xs) =
  if (f x) then (x : filter f xs)
  else filter f xs
```

- **Filter** retains every element which satisfies a **predicate**
- Only prepend the element if the predicate evaluates to **True**

Remember list comprehensions?

`map f xs` *is equivalent to* `[f x | x <- xs]`

`filter p xs` *is equivalent to* `[x | x <- xs, p x]`

Question: Can you define list comprehensions in terms of filter and map?

You can for simple list comprehensions, for example doubling every even number...

```
[ 2 * x | x <- [0..], x `mod` 2 == 0]
= map ((* 2)
        (filter (\x -> x `mod` 2 == 0) [0..]))
```

Question: Can you define list comprehensions in terms of filter and map?

However, this doesn't scale to when we have multiple generators, like with Pythagorean triples:

```
[ (x, y, z) | x <- [0..], y <- [0..], z <- [0..],  
           x^2 + y^2 == z^2]
```

For this we need some way of **flattening** intermediate lists

Folds

- A **fold** is a way of reducing a list into a single value
- **Idea:** We have a function which takes an element of the list, and an **accumulator** value, producing a new accumulator
- We have two types of fold, depending on the desired **associativity**
 - `foldl :: (b -> a -> b) -> b -> [a] -> b`
 - `foldr :: (a -> b -> b) -> b -> [a] -> b`

Left Fold

```
foldl (+) 0 [1, 2, 3, 4]
           ↓
(((0 + 1) + 2) + 3) + 4
           ↓
10
```

Since foldl is **left-associative**:

- Brackets “bunched” at the left
- List traversed in order from left-to-right

Try in GHCI to see it ...

```
f = \x y -> "(f " ++ x ++ " " ++ (show y) ++ ")"
```

```
foldl f "0" [1..5]
```

```
"(f (f (f (f (f 0 1) 2) 3) 4) 5)"
```

Right Fold

```
foldr (+) 0 [1, 2, 3, 4]
           ↓
1 + (2 + (3 + (4 + 0)))
           ↓
10
```

Since foldr is **right-associative**:

- Brackets “bunched” at the right
- List traversed in reverse from right-to-left

Try in GHCI to see it...

```
f = \x y -> "( f " ++ (show x) ++ " " ++ y ++ ")"
```

```
foldr f "0" [1..5]
```

```
"( f 1 ( f 2 ( f 3 ( f 4 ( f 5 0))))))"
```

Folds, side-by-side

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

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

Main difference:

- Recursive call outermost in left fold
- User-supplied function outermost in right fold

Sli.do Break

Property-based Testing with QuickCheck

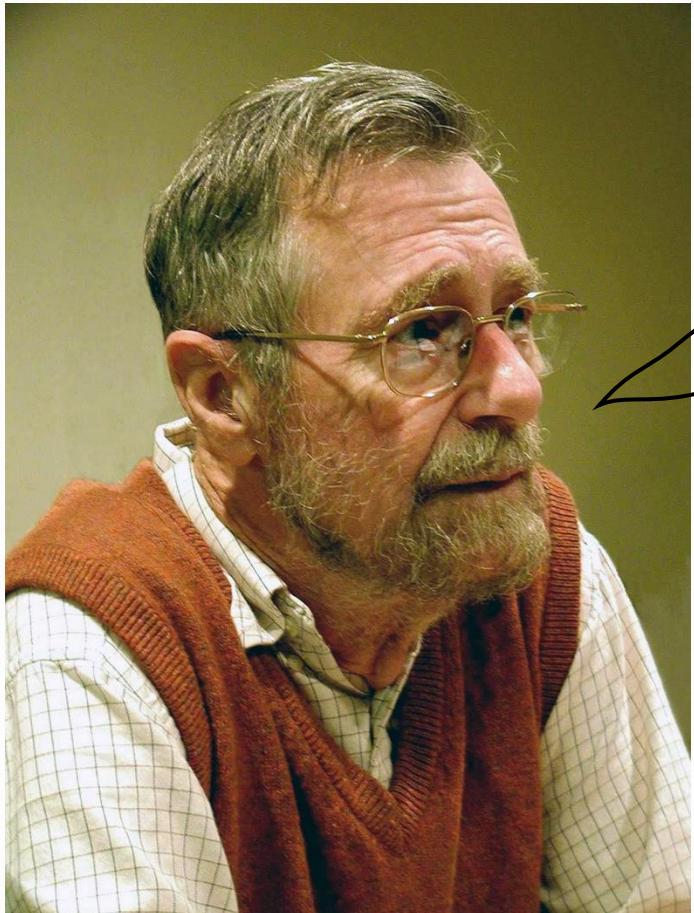
Property-Based Testing



- The Haskell **QuickCheck** library is the original property-based testing framework, invented by John Hughes
- Has been reimplemented in many other languages
- Commercialised by QuviQ

Property-based testing?

- You have a **program**
- You specify some **properties** that should be true for that program
 - code up as Boolean predicates
- You run the **QuickCheck** tool with your boolean predicates and it generates lots of random (type-correct) input data and checks the properties hold true



Program *testing* can
be used to show the
presence of *bugs*,
but never to show
their absence!

Typical example - length of a list

- Property in English:
 - 'a list containing n elements has length n'
- `let prop_len = \n -> (length [1..n] == n)`
- properties must return a Bool value
- `quickCheck prop_len`

Fixing up property for length of a list

- Problem: Our property is incorrect!
- Better property ...
 - ‘a list containing n elements has length n , for non-negative integer values n ’
- `let prop_len = \n -> (if n>=0 then length [1..n] == n else True)`
- use `verboseCheck` to see all the test cases (with their randomized input values)

QuickCheck behaviour

- It looks for the simplest case it can find that violates a property, then reports this value
- In general a lot of research in property-based testing has concentrated on shrinking counterexamples into a minimal case

Another example

- Check list is sorted in (strictly) ascending order:

```
isAscending :: [Int] -> Bool  
isAscending [] = True  
isAscending [x] = True  
isAscending (x:y:xs) = (x < y) && isAscending xs
```

isAscending properties

- Can we think of some properties to check for this function?

Any sublist of the natural numbers beginning at 1 should be ascending

```
prop_asc1 = \n -> isAscending (take n [1..])
```

Lists containing repeats of a single value are not ascending

```
prop_asc2 = \n -> if n < 2 then True else  
not (isAscending (take n (repeat 1)))
```

Adding the length as an element to the end of an ascending list makes it not ascending

```
prop_asc3 = \n -> if n<=0 then True else  
not (isAscending ([1..n]++[n]))
```

It fails! ... because we have an incorrect version of isAscending

- Below is what we *actually* need to implement! (which passes our tests)

```
isAscending :: [Int] -> Bool
isAscending [] = True
isAscending [x] = True
isAscending (x:y:xs) = (x < y) && isAscending (y:xs)
```

Reflection - different kinds of testing

- *Unit* testing - e.g. JUnit - specify a scenario with precise data and make assertions on the results
- *Fuzz* testing - e.g. AFL - generate randomized data for opaque system
- *Property-based* testing - e.g. QuickCheck - generate random inputs for specified properties

Sli.do Break

Sli.do Questions (1)

- Class test?
 - Please see Jeremy's post on Teams. Review lecture next Thursday.
- What's the name of the transformation $(\lambda x \rightarrow f x) \leftrightarrow f$?
 - This is known as eta-reduction (non-examinable)
- What would be 'quicker': filter/map or doing it using a list comprehension?
 - Good question! I don't know. My hunch is that it would be similar. Why not do an experiment? You can see how after we talk about IO in week 5.

Sli.do Questions (2)

- What if I want a list with 2 or more entries followed by list xs (x,y,xs)? How would I define that?

You can certainly nest the cons operator both when constructing a list and pattern matching on it. For example, see the following in GHCI:

```
ghci> let x:y:zs = 1:2:3:4:[]
```

```
ghci> x
```

```
1
```

```
ghci> y
```

```
2
```

```
ghci> zs
```

```
[3,4]
```

Sli.do Questions (3)

- What's the difference between the two folds if they compute same value?
 - The difference is due to associativity: that is, where the brackets are bunched. Foldl bunches the brackets to the left; foldr bunches the brackets to the right.
 - The two folds return the same value in our example only because + is an **associative operator**, meaning that $(x + y) + z = x + (y + z)$.
 - Trying this out on a non-associative operator like subtraction gives us different answers:
 - `foldl (-) 0 [1,2,3] = (((0 - 1) - 2) - 3) = -6`
 - `foldr (-) 0 [1,2,3] = (1 - (2 - (3 - 0))) = 2`

Sli.do Questions (4)

- Why do folds take an initial argument as well as the list? What is the advantage of that?
 - We can't actually define a fold without an initial accumulator. Have a look at the type signature for foldr again:
 - `foldr :: (a -> b -> b) -> b -> [a] -> b`
 - The function that we apply to each argument needs an accumulator value (in this case, something of type b), so we need to give it something to start with.
 - The initial value will, though, usually be something fairly sensible (e.g., an empty string, an empty list, 0...)

Sli.do Questions (5)

- In list sequencing [x..y] must it be the case that $x < y$?
 - Yes: typing [10..0] into GHCI will get you the list [].
 - You can get a descending list if you put the second element in, though:
 - $[10,9,..0] = [10,9,8,7,6,5,4,3,2,1,0]$
 - To learn more, check out description the enumFromTo function here:
<https://hackage.haskell.org/package/base-4.20.0.1/docs/GHC-Enum.html>
- With the prop_len are we just generating a list of consecutive nums then making sure it's the same len as the biggest num?
 - `let prop_len = \n -> (if n>=0 then length [1..n] == n else True)`
 - Yes: that's precisely it.

Next steps

- Try out **QuickCheck** for yourself
- Install on your local Haskell distribution
- Run through Friday's exercise sheet in the lab
- Do the quiz
- Next up: polymorphism and evaluation strategies
- No quiz next week, but 10% class test on the Friday

FP Lecture 5: Evaluation Strategies and Polymorphism

Simon Fowler & Jeremy Singer



<https://sli.do>, code 8728 242

Semester 1, 2024/2025
v2 14th October 2024



University
of Glasgow

Class Test

- Please see Jeremy's announcement on Teams
 - If you are not on Teams it is **crucial** that you join it ☺
- Key points:
 - Class test on Friday, worth 10% of overall grade.
 - Asynchronous, delivered by Moodle quiz.
 - Open between 9AM (earliest start) and 4:30PM (latest finish).
 - 1 hour time limit.
 - Anything covered up to and including this lecture is assessable.
- This Thursday's lecture will be a **revision / recap** lecture in advance of the class test.
- There is still a lab sheet for Friday (please do it ☺) but **no quiz**.

W3 Reflections (1)

- Of the answers I received, people really enjoyed the **lab** and learning **higher-order functions** (and rewiring themselves to think in this way).
- A few people said the **pacing was quite fast** at times. This is my first time doing the entire first half of the course, so it's easier for me to see this year that maybe splitting L1-3 into 4 lectures might be beneficial – we'll look at this for next year.
- A few people asked for **more exercises on ADTs / recursion**. I've dug up a lab from a couple of years ago and posted it (along with solutions) to the Moodle for more practice.

W3 Reflections (2)

- One common comment was about QuickCheck and writing good QuickCheck properties.
 - The key idea is to write a function that takes some random input data and calls the function you wish to test, returning a Boolean value.
 - QuickCheck will then spam you with a load of random input data and verify that your property returns True.
- For example, to test a sort function, you want to make sure:
 - That the sort doesn't change the list length:
`prop_sameLen = \xs -> length (sort xs) == length xs`
 - That the sort actually does sort the elements:
`prop_sort = \xs -> isAscendingEq (sort xs)`
 - That applying the sort twice doesn't mess up the order
`prop_idemp = \xs -> isAscendingEq (sort (sort xs))`

Superpowers (1)

- Possess my cat to see what he gets up to when he leaves the house
- Permanently revoke social media access for any celebrity of my choice
- Ability to manipulate probability. The most obvious one would be to go to the casino...
- Teleportation
- Have the complete knowledge and IQ of one person chosen by me for 24 hours
- I'd use super strength to build a really big flywheel, connect it to the power grid, then spin it a lot. Climate change solved 😎

Superpowers (2)

- Obviously I would take Batman's super power, his credit card.
- Superspeed, pretty much gives you teleportation and strength and the ability to slow time
- Large bags of unmarked cash appear at my doorstep
- See the future so I could see what the answers for the exam will be (and the lottery numbers)
- I'd pick instant learning. It'd be awesome to master any language or tool in an instant. If I mastered language, I could communicate with all living things.
- The power to Shapeshift. I've always wondered what echolocation is like



...and my favourite
Fly up the 11 floors

Today

- Last week
 - Recursion & algebraic data types
 - Higher-order functions
 - QuickCheck
- Lecture today will concentrate on evaluation strategies and polymorphism
 - Lazy vs. eager evaluation
 - Parametric polymorphism (in more detail)
 - Ad-hoc polymorphism (via typeclasses)
- Please ask questions!

Evaluation Strategies

Evaluation Strategies

- Expressions are evaluated as a program runs
- The *order* of expression evaluation depends on both the language semantics and the implementation pragmatics
- **Eager evaluation** - also known as strict evaluation or call by value
- **Lazy evaluation** - also known as non-strict evaluation or call by need

Evaluation Strategies

`fib :: Int -> Int`

Naïve Fibonacci: expensive to evaluate

`useFirst :: (a -> c) -> a -> b -> c`

Function that only uses first argument

`useFirst id 42 (fib 1000)?`

- In Python, this would take a very long time to evaluate
- In Haskell, it would terminate immediately

What are the benefits of lazy evaluation?

- We can compute with **large data structures**, including potentially **infinite lists**
- We can compute with **expensive functions**, which are only evaluated when we need
- We can compute with dangerous values, such as **undefined** or **bottom** - these will only cause problems if they are evaluated
 - **undefined** raises a runtime exception when evaluated, but has a generic type so can be used in any expression context
 - **let bottom = bottom** is an infinite recursively defined type whose evaluation will loop for ever (again, it has a generic type)

What are the drawbacks of lazy evaluation?

- Difficult to combine with **exception handling** since the order of evaluation is unclear
- Sometimes more difficult to predict: e.g., lazy IO
- Runtime **memory pressure** as an evaluation graph builds up, full of unevaluated expressions (thunks) - evaluation is only triggered when a value is 'needed' - e.g. when output via IO

Have you encountered lazy evaluation before?

- Yes, in C operators **&&** and **||**
- Yes, in Python 3 **range()** function
- However, in general, most languages (imperative, OO and functional) operate with **eager evaluation** strategies; in that sense Haskell is unusual

Some interesting infinite lists

- `[1..]`
 - the list of all non-negative integers
- `let ones = 1:ones`
 - an infinite list of 1 values
- `primes = sieve [2 ..]`
 where `sieve (x:xs) = x:(sieve [x' | x' <- xs, x' `mod` x /= 0])`
 - the infinite list of primes (via sieve of Eratosthenes)
- `fibs = 1:1:(zipWith (+) fibs (tail fibs))`
 - the infinite list of Fibonacci numbers

Useful functions for infinite lists

- **take :: Int -> [a] -> [a]**
 - selects the first n elements from a list
- **repeat :: a -> [a]**
 - produces an infinite list of repeated values
- **cycle :: [a] -> [a]**
 - produces an infinite list from a finite list by appending the list to itself infinitely
- **iterate :: (a -> a) -> a -> [a]**
 - produces an infinite list of function applications to an initial value
[x, f x, f (f x), ...]

Polymorphism (in more depth)

Polymorphism

- Polymorphism means “many forms” in Greek
- It’s the idea that code (in our case, functions) can operate over values from a variety of different types
- This enables code reuse, good software engineering
 - Also, sometimes help us to reason about program behaviour

Different kinds of polymorphism

- **Parametric** polymorphism: Functions operate on the **shape** of the arguments rather than with the data, so can operate on many types
 - Uses **type variables**
- **Ad-hoc** polymorphism: like method overriding in Java. Same function name but different implementations for different types
 - Achieved in Haskell using **typeclasses**

Parametric Polymorphism (in more depth)

Parametric polymorphism: recap

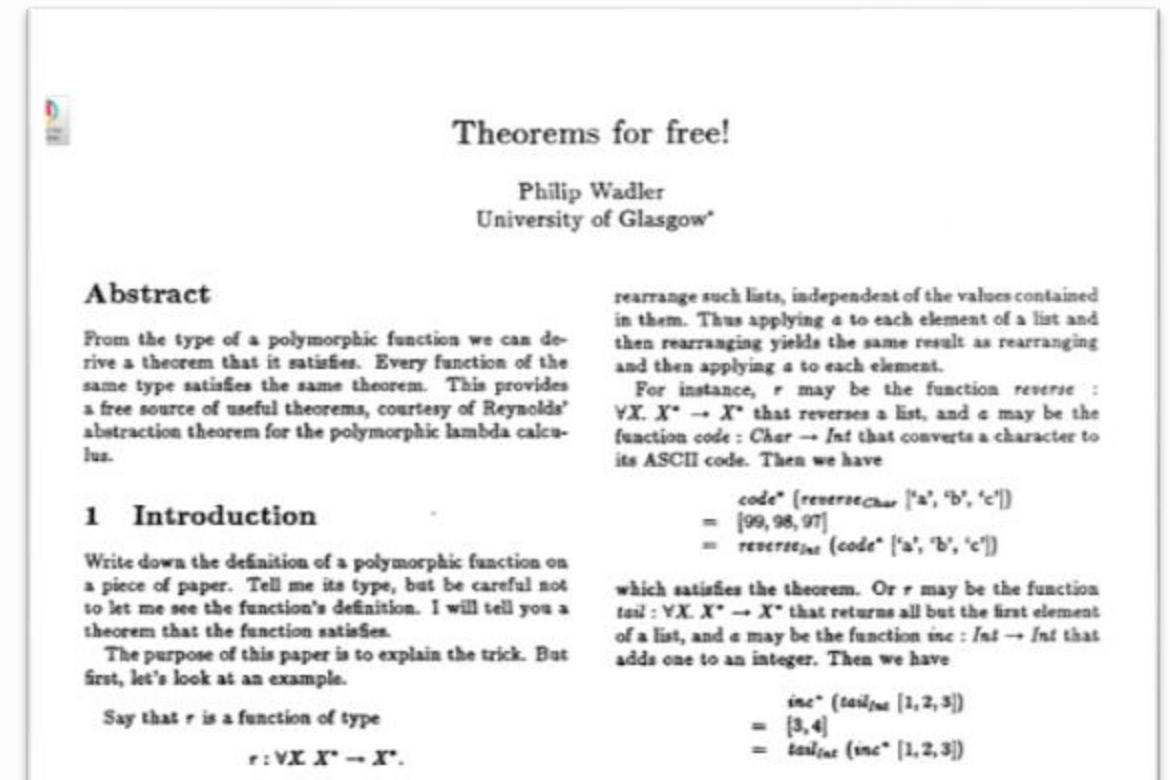
- Some classic examples:
 - `id :: a -> a`
 - `const :: a -> b -> a`
 - `map :: (a -> b) -> [a] -> [b]`
 - `filter :: (a -> Bool) -> [a] -> [a]`
 - ...
- **a** and **b** are type variables. These will be specialized to concrete types when the map is called on concrete values
- The behaviour of `map` does not depend in any way on the types of the list elements – it just applies the function to each element
- Other examples? `length`, `filter`, `folds`

How many functions have type $a \rightarrow b \rightarrow a$?

- Precisely one:

```
const :: a -> b -> a
const x _ = x
```

- This is but one consequence of parametricity
- See “Theorems for free!” for more details



Theorems for free!
Philip Wadler
University of Glasgow*

Abstract
From the type of a polymorphic function we can derive a theorem that it satisfies. Every function of the same type satisfies the same theorem. This provides a free source of useful theorems, courtesy of Reynolds' abstraction theorem for the polymorphic lambda calculus.

1 Introduction
Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies.
The purpose of this paper is to explain the trick. But first, let's look at an example.
Say that r is a function of type
 $r : \forall X. X^* \rightarrow X^*$.

rearrange such lists, independent of the values contained in them. Thus applying a to each element of a list and then rearranging yields the same result as rearranging and then applying a to each element.
For instance, r may be the function $\text{reverse} : \forall X. X^* \rightarrow X^*$ that reverses a list, and a may be the function $\text{code} : \text{Char} \rightarrow \text{Int}$ that converts a character to its ASCII code. Then we have

$$\begin{aligned} &\text{code}^* (\text{reverse}_{\text{char}} ['a', 'b', 'c']) \\ &= [99, 98, 97] \\ &= \text{reverse}_{\text{int}} (\text{code}^* ['a', 'b', 'c']) \end{aligned}$$

which satisfies the theorem. Or r may be the function $\text{tail} : \forall X. X^* \rightarrow X^*$ that returns all but the first element of a list, and a may be the function $\text{inc} : \text{Int} \rightarrow \text{Int}$ that adds one to an integer. Then we have

$$\begin{aligned} &\text{inc}^* (\text{tail}_{\text{int}} [1, 2, 3]) \\ &= [3, 4] \\ &= \text{tail}_{\text{int}} (\text{inc}^* [1, 2, 3]) \end{aligned}$$

We can define our own polymorphic functions

- Duplicate and put in a list

```
dapial :: a -> [a]  
dapial x = [x,x]
```

- Extract element from triple

```
takeLast :: (a,b,c) -> c  
takeLast (x,y,z) = z
```

- Note: Both only **manipulate** (rather than **use**) the variables

Polymorphic data types

- We can also use type variables within data type declarations:

```
data Tree a =  
    Leaf a  
    | Node (Tree a) (Tree a)
```

- Tree can contain values of any type within it (although once it is specialised, we have fixed the concrete type)

Node (Leaf 5) (Leaf 10) :: Tree Int

Node (Node (Leaf "Hello") (Leaf ("World")))
(Leaf "!") :: Tree String

Functions for polymorphic container types

```
treeMap :: (a -> b) -> Tree a -> Tree b
```

```
treeMap f (Leaf x) = Leaf (f x)
```

```
treeMap f (Node t1 t2) = Node q1 q2
```

where

```
q1 = treeMap f t1
```

```
q2 = treeMap f t2
```

(looking ahead: this is an instance of a **functor** for trees)

Ad-hoc Polymorphism via Typeclasses

Ad-Hoc Polymorphism

- Method or operator overloading in Java is a form of **ad-hoc polymorphism**

```
public float add(float f1, float f2) { ... }  
public int add(int i1, int i2) { ... }
```

- Ad-hoc is not meant to be a bad thing – just that, unlike parametric polymorphism, it is not a core part of the type system
- It is useful in allowing us to perform similar operations (e.g., addition, converting to a string...) on different types where each operation acts potentially differently on each type

How to make *ad-hoc* polymorphism less *ad hoc*

Philip Wadler and Stephen Blott
University of Glasgow*

Abstract

This paper presents *type classes*, a new approach to *ad-hoc* polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the “eqtype variables” of Standard ML. Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal introduction to type classes, and defines them formally by means of type inference rules.

1 Introduction

Strachey chose the adjectives *ad-hoc* and *parametric* to distinguish two varieties of *polymorphism* [Str67].

Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type. A typical example is overloaded multiplication: the same symbol may be used to denote multiplication of integers (as in $3*3$)

ML [HMM86, Mil87], Miranda¹[Tur85], and other languages. On the other hand, there is no widely accepted approach to *ad-hoc* polymorphism, and so its name is doubly appropriate.

This paper presents *type classes*, which extend the Hindley/Milner type system to include certain kinds of overloading, and thus bring together the two sorts of polymorphism that Strachey separated.

The type system presented here is a generalisation of the Hindley/Milner type system. As in that system, type declarations can be inferred, so explicit type declarations for functions are not required. During the inference process, it is possible to translate a program using type classes to an equivalent program that does not use overloading. The translated programs are typable in the (ungeneralised) Hindley/Milner type system.

The body of this paper gives an informal introduction to type classes and the translation rules, while an appendix gives formal rules for typing and translation, in the form of inference rules (as in [DM82]). The translation rules provide a semantics for type classes. They also provide one possible implementation technique: if desired, the new system could be

class Show a where
show :: a -> String

- In Haskell, ad-hoc polymorphism is achieved using typeclasses
- A typeclass specifies a list of operations to be defined for a type
- In a sense, typeclasses bring parametric and ad-hoc polymorphism closer together
 - Since we can say “we don’t care what a particular type a is, only that it supports certain operations”

Typeclasses

For type a to be part of the Show typeclass, we must implement a function show (which takes a and returns a String)

```
class Show a where  
    show :: a -> String
```

- Only values with a **show** function can be represented as **Strings**
 - But we can use the **Show** typeclass to indicate that a value has a **show** function defined

```
show :: (Show a) => a -> String
```

Typeclass constraints go on the left hand side
of the big arrow

How do we show that a type belongs to a typeclass?

- We can specify that an algebraic data type belongs to a typeclass in two different ways
 - **instance** declaration
 - Here, we specify the implementations of each function explicitly
 - **deriving** clause
 - This will implement the typeclass using default behaviour

```
data Insect = Spider | Centipede | Ant
```

Instances

```
class Eq a where  
  (==) :: a -> a -> Bool
```

```
class Show a where  
  show :: a -> String
```

We can show that a type belongs to a typeclass by writing **instance declarations** that specify the implementations for each function specification in the class

```
instance Eq Insect where  
  Spider == Spider = True  
  Centipede == Centipede = True  
  Ant == Ant = True  
  _ == _ = False
```

```
instance Show Insect where  
  show Spider = "Spider"  
  show Centipede = "Centipede"  
  show Ant = "Ant"
```

Deriving

```
data Insect = Spider | Centipede | Ant  
deriving (Show, Eq)
```

The **deriving** clause provides us with default implementations for the appropriate functions, here **show** and **(==)** and **(/=)**

```
Ant == Ant  
Spider /= Ant  
show Centipede → “Centipede”
```

This is possible because we can derive a string by using the data constructor, and declare two insects equal if they use the same data constructor

- If the data constructors had associated data, in order to support deriving, their types would need to support Show/Eq too

Question: When would we use deriving vs. writing an instance?

- Deriving:
 - When you want “default” behaviour / behaviour is as you would expect
 - When the behaviour is already specified for any associated data
 - When the typeclass supports deriving
- Instance:
 - When the typeclass does not support deriving (e.g., a custom typeclass)
 - When you want behaviour that’s different to the default (e.g., you want Show to pretty-print an abstract syntax tree)

The Read typeclass

- **read** allows us to convert **String** values into other values (c.f. **input** in Python or **scanf** in C)

```
data Insect = Spider | Centipede | Ant  
    deriving (Read, Show, Eq)  
(read "Centipede") :: Insect
```

Note that we must give an explicit type annotation!

Example: The Leggy Class

```
class Leggy a where  
    numLegs :: a -> Int
```

a is the type variable that is a placeholder for the concrete type that will be an instance of the typeclass

We use **a** in the type specifications of the functions

Now we can add instances to the Typeclass

```
instance Leggy Insect where
    numLegs Spider = 8
    numLegs Ant = 6
    numLegs Centipede = 100
```

Note:

- Put the typeclass name before the instance name read “Insect is an instance of the Leggy typeclass”
- We have to provide definitions for all functions specified in the typeclass

Now we can use the typeclass

```
describe :: (Show a, Leggy a) => a -> String  
describe x = (show x) ++ "s have " ++  
             (show $ numLegs x) ++ " legs"
```

Could this not just have the type:

```
describe :: Insect -> String
```

Yes - but our code is much more extensible - we can describe any instance of **Leggy** !

Four Legs Good, Two Legs Bad/Better

```
data Mammal = Human | Dog | Cat | Pig  
deriving (Show, Eq)
```

```
instance Leggy Mammal where  
    numLegs Human = 2  
    numLegs _ = 4
```

-- and now we can describe Mammal values too! So, *ad-hoc polymorphism enables type-safe code reuse*

Towards a nursery rhyme...

```
class Noisy a where  
    mkNoise :: a -> String
```

```
instance Noisy Mammal where  
    mkNoise Dog = "woof"  
    mkNoise Cat = "meow"  
    mkNoise Pig = "oink"  
    mkNoise _ = "ow"
```

Nursery Rhyme!

Live coding / on Moodle

```
song :: (Show a, Noisy a) => [a] -> String  
song animals = concatMap verse animals
```

```
putStrLn $ song [Pig,Dog,Cat]
```

Useful References about Typeclasses

- <http://learnyouahaskell.com/making-our-own-types-and-typeclasses> - this article is long but worth reading
- Hutton, Chapter 8 (Declaring Types and Classes)

Answers to sli.do questions (Administrative)

- Will the class test include this week's material?
 - Yes, everything up to and including this lecture is examinable.
- When will the assignment be released?
 - Week 8, see the roadmap.
- Where is this week's quiz?
 - There isn't one, only the class test. That will be visible on Friday.
- Can we only do the class test in the lab?
 - No – you can do it from wherever. In fact we would ask that you don't use the lab session to do the class test.
- Are there any sample class tests?
 - Yes, have a look at "Previous exam papers" on Moodle.
- How soon after the lecture will the recording be released?
 - In general when Echo360 decides to release it (I'm not sure why there is a delay) – but I'm trying to remember to release it manually shortly after each lecture!

Answers to sli.do questions (Technical, 1)

- Is there a way to "mark as eager" evaluation in Haskell in the same way you can "mark as lazy" in the other languages mentioned?
 - Yes! You can use "strictness annotations" on arguments – see this week's lab sheet.
 - For example...

```
constStrict :: a -> b -> a
constStrict x !y x = x
```

would ensure the second argument is evaluated even if it is not used
- Can you elaborate on "thunk". Is it a term within haskell or more generally?
 - A thunk is an un-evaluated expression / computation. In a language like Haskell if we had

```
const (1 + 2) (3 + 4)
```

here both $(1 + 2)$ and $(3 + 4)$ are both thunks as they are not evaluated until required.
 - In a language such as OCaml with strict evaluation, you can still make a computation into a thunk by 'hiding' it behind a lambda, like $\lambda x \rightarrow \text{fib } 1000$. Then you would 'force' the thunk by applying it to a unit value:
 $(\lambda x \rightarrow \text{fib } 1000) ()$

Answers to sli.do questions (Technical, 2)

- All these examples include ignoring the second argument, but can you ignore the first? Will it just jump over the first parameter if it is not needed?
 - Yes, absolutely. There's nothing special about the definition of the const function:

```
const :: a -> b -> a
const x y = x
```
 - We could also write a function to return the second argument:

```
const2 :: a -> b -> b
const2 x y = y
```
 - The const2 function would similarly not evaluate its first argument since it is unused

Answers to sli.do questions (Technical, 3)

- Why bother with anonymous functions instead of giving it a name?
 - Often it's good / necessary to give a function a name. Anonymous functions can be very useful, though, when you want to write an inline definition (e.g., as an argument to a HOF). For example:
 - `evens = filter (\x -> x `mod` 2 == 0) [0..]`
 - `addOneToList xs = map (\x -> x + 1) xs`
- Do unevaluated expressions remain in memory until forced?
 - Until forced, or the garbage collector determines that they are not needed. Functional GC is an active research area – Jeremy knows a lot more about this than me 😊

Answers to sli.do questions (Technical, 4)

- Why can't you give [5,10] in a func of type $a \rightarrow [a]$? Can't I just do something like = $xs:10$
 - The issue here is that we know that 10 is a **specific value** (of type Int) whereas a function of type $a \rightarrow [a]$ must work on **all types**.
 - Suppose we try and compile the following:
`addTenToList :: a -> [a]`
`addTenToList xs = xs ++ [10]`
(Note we need `++` rather than `:` as we're putting something on the **end** of the list)
 - Here is the error we get:
Couldn't match expected type 'a' with actual type 'Int'
'a' is a rigid type variable bound by
the type signature for:
`addTenToList :: forall a. a -> [a]`

Answers to sli.do questions (Technical, 5)

- Do functors relate to polynomial time reductions? I think of how we'd try to convert a clique problem to sat in algs and it feels like a similar concept
 - Interesting question! I don't think so, though. A functor allows you in essence to apply a function to the contents of a data structure, but you're not allowed to change its structure. For example, map will apply a function over a list but cannot change the structure of the list.
 - From the (admittedly little) I remember about polynomial time reductions, they're often quite a global transformation that will require you to change the structure of the representation (even if you keep the meaning the same).

Answers to sli.do questions (Technical, 6)

- I thought Haskell didn't support OOP. The classes/typeclasses etc. seem similar to objects/inheritance etc.
 - Formally speaking in PL theory an object is a record that can refer to itself with a 'self' field (see "A Theory of Objects" by Abadi & Cardelli), and a class is a template for creating these objects.
 - In terms of how Java envisages objects, an object has a set of (mutable) fields with accessibility annotations (public / private) along with *methods* that are functions with access to the object's methods.
 - In Haskell, I wouldn't say typeclasses (introduced by the 'class' keyword) really have much to do with OOP. Instead they say that we can define an operation (e.g., equality or serialisation) on a set of types that implement that typeclass. They are a bit closer to Java's interfaces.

Answers to sli.do questions (7)

- What does \$ do?
 - The dollar operator is a shorthand that sometimes means we can write fewer parentheses. For example we might want to write
 - `show (id 5)`
 - Instead we can use dollar notation to avoid the brackets:
 - `show $ id 5`

Answers to sli.do questions (8)

- Is `show :: (Show a) => a -> String` just another way of writing '**class Show a where show :: a -> String**'?
 - Good question. You can't write a function directly with that type signature, but once you've written the typeclass out then you've declared the 'show' function.
 - For example, try the 'Leggy' example; in GHCi you will get:
`ghci> :t numLegs`
`numLegs :: Leggy a => a -> Int`
 - You give the function implementation **for a particular type** using 'instance' declarations.
- Why would you take an argument and not use it? (as with `const`)
 - It's quite useful in point-free style, when you're using a higher-order function but you don't need all of the arguments that you're given.

Wrapping Up

- **Today**
 - Evaluation strategies: lazy vs. eager
 - Polymorphism: parametric & ad-hoc (via typeclasses)
- **Next up**
 - Revision lecture ahead of the class test
 - Class test this Friday (details will be on Teams / Moodle)
- **Over to you**
 - Try defining some of your own typeclasses
 - Have a go at the exercise sheet (available tomorrow)
- **Questions?**

Functional Programming (H)

Section 1 Recap

Simon Fowler & Jeremy Singer

Semester 1, 2024/25



<https://sli.do>, code 3399 105



University
of Glasgow

Today

- Last time
 - Evaluation strategies
 - Parametric polymorphism
 - Ad-hoc polymorphism via typeclasses
- Today's lecture is for revision, questions, and consolidation
 - Recap of all material so far
 - Interleaved with previous class test questions and solutions
- Please ask questions!

Class Test

- Please see Jeremy's announcement on Teams
 - If you are not on Teams it is **crucial** that you join it ☺
- Key points:
 - Class test on Friday, worth 10% of overall grade.
 - Asynchronous, delivered by Moodle quiz.
 - Open between 9AM (earliest start) and 4:30PM (latest finish).
 - 1 hour time limit.
 - Anything covered up to last lecture is assessable.
- There is still a lab sheet for Friday (please do it ☺) but **no quiz**.

Expressions and Reduction

Expressions vs. Statements

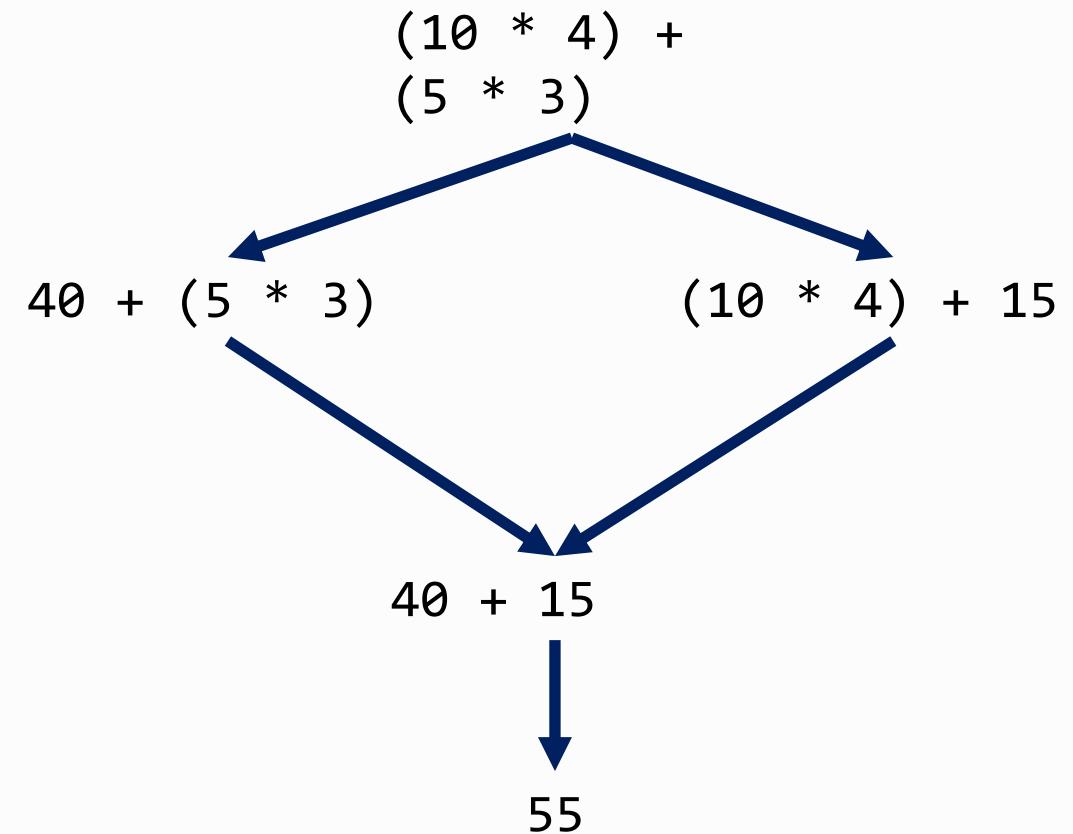
- In an **imperative** language (e.g., Java), a statement is an **instruction**
 - Computation driven by evaluating statements to manipulate program counter / control flow
- An **expression** is a (potentially complex) term in the language (e.g., $5 + 5$) that will eventually evaluate to a **value** (e.g., 10)

In a functional language, **everything is an expression**, and this is the main thing that sets FP apart from imperative languages

Reduction

Computation happens by **reducing** an expression towards a value

Given that we don't have side-effects, reduction satisfies the **Church-Rosser** property: no matter how we choose to reduce the expression, it will always reduce to the same value



Example Question

Is the following code legal Haskell?

```
if (if 5 < 10 then True else False) then  
    “Branch 1”  
else  
    “Branch 2”
```

- No: since **if** is a statement, it cannot be used inside another **if**
- No: both **if-then-else** blocks must have the same type
- Yes: this is OK since the inner **if** has type **Bool**

Example Question

Is the following code legal Haskell?

```
if (if 5 < 10 then True else False) then  
    “Branch 1”  
else  
    “Branch 2”
```

- No: since **if** is a statement, it cannot be used inside another **if**
- No: both **if-then-else** blocks must have the same type
- Yes: this is OK since the inner if has type Bool

Functions and Currying

Functions

```
\x -> x + 5
```

```
addFive :: Int -> Int  
addFive x = x + 5
```

- Functions take parameters and produce a result
- Haskell functions are **first-class** and can be let-bound, passed around, returned, etc.
- Functions can be **anonymous** (lambdas) or **named** using an equation

Currying and Partial Application

$$\begin{array}{c} \backslash n_1 \rightarrow (\backslash n_2 \rightarrow n_1 + n_2) \\ \text{Int} \rightarrow \text{Int} \\ \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \end{array}$$

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

addFive :: Int -> Int
addFive = add 5
```

- Strictly speaking, a function takes only one parameter
- We can get multi-argument functions in two ways:
 - Tuple up all arguments
 - Use **currying**, where a function returns another function
- Currying has the advantage of allowing **partial application**, where we can specialize a function to a given argument

Example Question

- Assume a function `ordinal :: Int -> String`, where
`ordinal 1 = "first"`
`ordinal 2 = "second"`, etc.
- What is the type of the function:
 - `greetMonarch title firstname num =
 "Hello, " ++ title ++ " " ++ firstname ++
 " the " ++ (ordinal num)`
- `greetMonarch :: String -> Int -> String`
- `greetMonarch :: String -> String -> Int -> String`
- `greetMonarch :: a -> a -> Int -> a`
- `greetMonarch :: String -> String -> String -> String`

Example Question

- Assume a function `ordinal :: Int -> String`, where
`ordinal 1 = "first"`
`ordinal 2 = "second"`, etc.
- What is the type of the function:
 - `greetMonarch title firstname num =
 "Hello, " ++ title ++ " " ++ firstname ++
 " the " ++ (ordinal num)`
- `greetMonarch :: String -> Int -> String`
- **greetMonarch :: String -> String -> Int -> String**
- `greetMonarch :: a -> a -> Int -> a`
- `greetMonarch :: String -> String -> String -> String`

Lists and List Comprehensions

Lists

```
[1,2,3] :: [Int]  
["Hello", "COMPSCI4021"] :: [String]
```

```
doubleEvens =  
  [ 2 * x | x <- [1..], x `mod` 2 == 0 ]
```

The diagram illustrates the structure of the list comprehension `[2 * x | x <- [1..], x `mod` 2 == 0]`. It is divided into three main parts: the **Body** (`2 * x`), the **Generators** (`x <- [1..]`), and the **Predicate** (`x `mod` 2 == 0`). Each part is highlighted with a yellow box, and arrows point from these boxes to their respective labels below.

- A list is an **ordered sequence** of values of the **same type**
- We can use **sequence** notation to create ordered lists
 - These can potentially be infinite
- We can also create lists using **list comprehensions**

Deconstructing Lists

```
head :: [a] -> a
tail :: [a] -> [a]
(!!) :: [a] -> Int -> a
```

```
headDef :: [a] -> a -> a
headDef [] def    = def
headDef (x:xs) _ = x
```

- We can deconstruct a list using accessor functions
 - These are a bit of a code smell, as they may fail
- Alternatively, we can deconstruct using pattern matching
 - (x:xs) – x is the head of the list, xs is the tail
 - Generally better as it encourages you to cover all cases

Algebraic Data Types, Pattern Matching, and Recursion

Algebraic Data Types

```
data Colour = Red | Black  
  
data Suit =  
    Hearts | Diamonds  
    Clubs   | Spades  
  
data Card =  
    King Suit  
    Queen Suit  
    Jack Suit  
    Ace Suit  
    Number Suit Int
```

- Algebraic data types: different ways of constructing a type
 - Hearts :: Suit, Diamonds :: Suit, etc.
- When writing a function on an ADT, need to **pattern match**:

```
getColour :: Suit -> Colour  
getColour Hearts     = Red  
getColour Diamonds   = Red  
getColour Clubs      = Black  
getColour Spades     = Black
```

Writing Recursive Functions

Base case: sum of elements in an empty list is 0

```
listSum :: [Int] -> Int
listSum [] = 0
listSum (x:xs) =
  x + (listSum xs)
```

Recursive case: add current element to sum of the rest of the list

- Think about the **structure** you are defining recursion on
 - Integers? Lists?
- Write a **base case**: you will want (at least pure!) functions to terminate
- Write a **recursive or inductive** case which calls the same function with an argument which **converges to the base case**

Higher-Order Functions

Higher-Order Functions?

```
map    :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
twice  :: (a -> a) -> a -> a
```

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

```
map (\x -> x * 2) [1,2,3]
→ [2,4,6]
```

- A **higher-order function** is a function which **takes another function as an argument**
- It is good practice to use HOFs where applicable rather than hand-roll recursive functions yourself
 - Concentrate on **application logic** rather than recursive boilerplate
- Often convenient to supply a HOF with an anonymous function

Common HOFs

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

Applies a function to each element of a list

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

Only keeps elements of a list that satisfy a predicate

`twice :: (a -> a) -> a -> a`

Applies a function twice

Folds

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

```
foldl (+) 0 [1, 2, 3, 4]  
↓  
(((0 + 1) + 2) + 3) + 4  
↓  
10
```

```
foldr (+) 0 [1, 2, 3, 4]  
↓  
(1 + (2 + (3 + (4 + 0))))  
↓  
10
```

- A **fold** is a way of reducing a list into a single value
- **Idea:** We have a function which takes an element of the list, and an **accumulator** value, producing a new accumulator
- We have two types of fold, depending on the desired **associativity**

Property-Based Testing

Property-Based Testing / QuickCheck

Check that a list [1..n] has length n

```
prop_len :: Int -> Bool
prop_len n =
  if n >= 0 then
    length [1..n] == n
  else True
```

- We often want to see whether a function satisfies some property
- Property-based testing (as implemented in QuickCheck) will generate many random instances and check whether a property holds
- If not, it will provide a counterexample

Question

- Would Property-Based Testing still work in a language without referential transparency?
 - No: PBT requires all functions to be pure
 - Yes: the PBT technique is independent of referential transparency (but side effects might make it more difficult to reason about why a test passes or fails)

Question

- Would Property-Based Testing still work in an impure language?
 - No: PBT requires all functions to be pure
 - Yes: the PBT technique is independent of purity (but side effects might make it more difficult to reason about why a test passes or fails)
- QuickCheck is also implemented for Erlang, which is impure. We would typically want our properties to be pure, though.

Evaluation Strategies

Eager vs. Lazy Evaluation

```
take 5 [1..] →  
[1,2,3,4,5]
```

```
cycle [1,2] →  
[1,2,1,2,1,2,...]
```

- Haskell uses **lazy evaluation**: this means that computations only happen as they are needed
- This is in contrast to **eager evaluation**, where computations are evaluated earlier (e.g., when passed as an argument to a function)
- We can use this to, for example, create infinite lists

Example Question

- Suppose we have a function `blowup x = blowup (x + 1)`. What would be the result of evaluating `const 5 (blowup 1)`?
- The program would immediately evaluate to return 5
- The program would return the infinite list `[5,6,7,...]`
- The program would not terminate

Example Question

- Suppose we have a function `blowup x = blowup (x + 1)`. What would be the result of evaluating `const 5 (blowup 1)`?
- **The program would immediately evaluate to return 5**
- The program would return the infinite list `[5,6,7,...]`
- The program would not terminate

Polymorphism

Parametric Polymorphism

```
id    :: a -> a  
const :: a -> b -> a  
map   :: (a -> b) -> [a] -> [b]  
filter :: (a -> Bool) -> [a] -> [a]
```

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

- Some functions only use their arguments structurally
 - Applying a supplied function, or reversing a list
- Therefore they can have **many different types**
- We can therefore give them very general types using **type variables**
- The types sometimes mean there is only one implementation, due to **parametricity**

Ad-hoc Polymorphism via Typeclasses

```
class Show a where
    show :: a -> String

instance Show Suit where
    show Hearts      = "Hearts"
    show Diamonds   = "Diamonds"
    show Spades     = "Spades"
    show Clubs      = "Clubs"

show :: (Show a) => a -> String
```

- Java supports ad-hoc polymorphism using function overloading
 - For example, we could define an open function on a socket, or stdout, or a file
- Haskell supports ad-hoc polymorphism using typeclasses
 - A specification of the functions that an instance must implement

Code Questions

Code Questions

- Some of the class test may consist of free-text code questions, and will be marked qualitatively
- Feel free to use functions from the standard library (e.g., the Prelude, and Data.List or Data.Char)

Example Code Question (Class Test 2022)

Tally mark counting is a unary number system.

Numbers 1 (I) to 4 (IIII) are represented by sequences of adjacent vertical bars. Then number 5 (IIII\) is represented by 4 vertical bars and a single diagonal bar.

Larger numbers are grouped into clusters of fives, with the remainder on the right hand side - for instance 13 is: IIII\ IIII\ III

Your task is to write a function `tally :: Int -> Maybe String` which computes the tally mark representation of an arbitrary `Int` value.

- Note there is no tally mark representation for negative numbers.
- The tally mark representation for 0 is the empty string.
- Your result should include spaces between clusters of five tally marks - as shown above for 13.
- Ideally there should be no trailing space at the end of the string.

Example Code Question (Class Test 2022)

(Live coding)

Sli.do Questions (Administrative)

- Are we expected to write code that compiles?
 - No – although it should get the key ideas across.
- How can I get a job in functional programming?
 - Many places (e.g., Jane Street, Bloomberg, Standard Chartered, Meta, Tarides, Tweag) hire FP-specific developers.
 - Other companies might have more general roles that use FP languages (e.g. Scala – I know ITV do, also many banks)
- Will lab tutors still be at the lab tomorrow to help with the lab sheet?
 - Yes, labs are on as normal. Please don't do the class test in the lab.
- How long is the quiz and how difficult is it?
 - Class test is an hour. “How difficult” is subjective ☺ But it will be fair. Jeremy is handling the quiz and I haven’t seen the questions.

Sli.do Questions (Technical, 1)

- So currying is almost always used when a function has multiple parameters?
 - Yes – almost all Haskell multi-argument functions are curried
- When should you use an anonymous lambda function, and when should you use a parameter name? My IDE always tries to correct my anonymous functions.
 - Use an anonymous function when you don't need to give it a name (for example, as an argument to a HOF). If you want to refer to it multiple times, give it a name.
 - In general, don't worry about your IDE's hints – they all assume you are an experienced Haskell developer rather than someone who is learning.

Sli.do Questions (Technical, 2)

- What is the difference in pattern matching where you have a base case and another, and recursive functions?
 - You can use pattern matching without recursion (see for example where we case split on a Suit).
 - You can also use recursion without pattern matching (e.g., when implementing a factorial function). You'd still need to test that you'd reached the base case, though (e.g., using a conditional).
- When defining an algebraic data type, is it possible to use a type but like constrained? For example Ints between 2 and 9 instead of just any Int?
 - Excellent question! In vanilla Haskell, no. But you *can* do this using more advanced type systems, for example **refinement types** (see LiquidHaskell) and **dependent types** (see Idris / Agda).

Sli.do Questions (Technical, 3)

- Is Haskell Maybe similar to Java's Optional?
 - Yes, very similar.
- In the greetMonarch example question you mentioned that '++' is a list concatenation operator. Does that mean this works the same as ':?'
 - Not quite.
 - The 'cons' operator prepends a **single value** to the top of an existing list.
`(:) :: a -> [a] -> [a]`
 - The append operator **concatenates two lists** together
`(++) :: [a] -> [a] -> [a]`

Sli.do Questions (Technical, 4)

- I think I missed the explanation on what the difference between the \$ and . composition operators. Do they do the same thing?
 - '\$' is function application, and '.' is function composition.
 - Take an 'inc' function as an example:

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

- If we want to apply the function twice, we can write
`inc (inc x)`

The '\$' notation allows us to avoid writing the brackets:

```
inc $ inc x
```

- Function composition allows us to **create a new function** that increments twice:

```
let incTwice = inc . inc in  
incTwice 5
```

Sli.do Questions (Technical, 5)

- For the common HoFs, I know some like map are loaded in with the prelude, can we overload these with our own implementations by simply making a function called map?
 - If you were to implement your own with the same name, you'd probably get an ambiguity error (unfortunately).
 - Instead, either give it a different name (e.g., 'myMap'), or explicitly tell GHC not to load the Prelude (https://wiki.haskell.org/No_import_of_Prelude)
- When doing let bindings from lecture 2, can vars within the binding be accessed completely separately from the equation? E.g. could we use det2 outside the body
 - No. If you have `let x = M in N`, for some expressions M and N, 'x' is only bound in N, and is free elsewhere (so can't be accessed outside of the scope).

Sli.do Questions (Technical, 6)

- Is it invalid (i.e., the compiler will error) if my recursive statement doesn't explicitly contain fewer elements than the current call of the function?
 - Not in Haskell. You can quite straightforwardly write nonterminating functions (e.g., $f\ x = f\ x$)
 - Other functional languages like Agda include a **termination checker** that does require that pure functions are terminating.
 - This is because Agda can be used for theorem proving – and nontermination allows you to prove arbitrary things.
 - (PL-geek-ing a bit here, there are some interesting papers about termination checks and its consequences. Check out "Turing Completeness, Totally Free" by Conor McBride, and various works on sized types by Andreas Abel if you're interested)

Wrapping Up

- Well done on completing Section 1 of the course!
- Good luck for the class test tomorrow. There is no quiz this week (but please still do the exercise sheet – it's slightly shorter).
- Next up is an exciting part of the course...
How do Haskell programs interact with the real world, and how can we manage side effects?

Functional Programming (H)

Lecture 7: Introduction to IO

Simon Fowler & Jeremy Singer



<https://sli.do>, code 4098 384

Semester 1, 2024/2025
21st October 2024



University
of Glasgow

Today

- Last time:
 - Recap lecture for Section 1 of the course
 - Class test
 - Parametric polymorphism / evaluation strategies
- Today:
 - (Finally) – how do we write Haskell programs that interact with the outside world?
 - How do we integrate pure and impure code?
 - The IO type and do-notation

Overall Course Structure

1. Haskell Fundamentals
2. Monads <- **you are here** (starting this section today)
(Note: Jeremy will start lecturing next week – but I'll be back)
3. Haskell in the Large
4. Advanced Concepts

Programming Assignment

- At the beginning of Week 8 we will release the **programming assignment**, worth 25% of the course grade
- This will be a larger program (last year was Connect 4) – this year's will be similar
- Released: Monday 11th November 2024, deadline: Friday 6th December 2024
- The lecture on the 11th November will introduce the coursework in more depth
- Feel free to work on this in the labs; we are happy to discuss it

Class Test

- Thanks to everyone for completing the class test on Friday
- It's worth **10%** of the final course grade
 - If you think you did well – great, marks in the bag!
 - If you had trouble – don't worry, you can make up the marks elsewhere (and it hopefully highlighted where you need more practice)
- If you missed it, submit a Good Cause claim if appropriate
- Jeremy will be going through and marking these
 - You should hopefully have the results back within the usual timeframe (3 weeks from submission)

W4 Reflections

- Overall the feeling was “OK I guess” on the class test – several people thought they did OK, several struggled with not knowing the precise format
- **If you are struggling please get in touch with me: I am happy to meet**
- Type classes are still causing a few issues: I have put a video out going into a bit more detail and can distribute a bit more reading material.
- People seemed to enjoy the lectures (esp. revision lecture). This is good to know since I was considering axing that one next year!

Interesting Facts* (1)

***Not fact-checked!**

- People are born with only 2 fears. The fear of falling and loud noises.
- Most sharks don't know how to file taxes
- Plato was a nick name meaning "Broad Shouldered" likely because he was muscular
- Fishing crates in Terraria are actually locked to the difficulty you fished them in. If you stockpile a bunch of crates & defeat wall of flesh, thus putting you into hardmode, your crates will still drop pre-hardmode loot. Shimmer updates their loot drops.
- Ants can survive in the microwave cos they can see microwave radiation like we can see light.

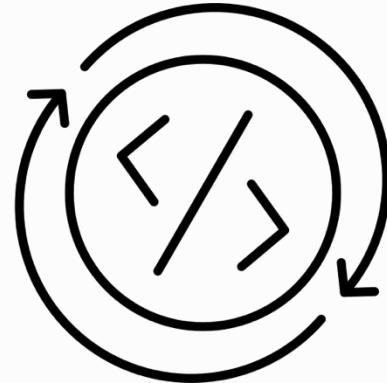
Interesting Facts* (2)

- The inventor of the frisbee was cremated and had their ashes made into a frisbee.
- Humanity is doomed by overpopulation, climate change, resource depletion, and uncontrolled technology, leading to ecosystem collapse, uninhabitable environments, and societal breakdown.

***Not fact-checked!**

Purity

- So far, all the Haskell we have seen has been *pure code*
- But what does **functional purity** mean?



Stateless: the function always evaluates to the same result, given the same arguments



Side-effect free: the function does nothing apart from simple expression evaluation - no interaction with the 'outside world'



(ideally) **Total:** defined for all inputs

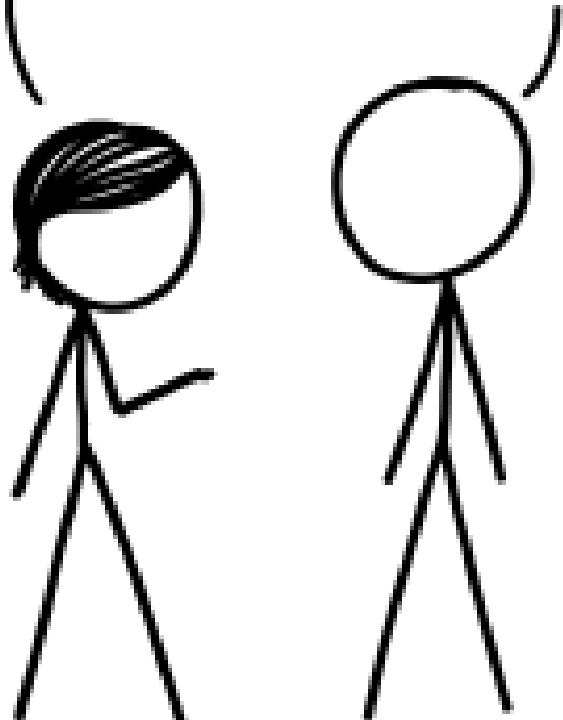
What's the problem with purity?

- Our code needs to do I/O
 - to get input from peripheral devices
 - to send output to the user
 - to read/write files
 - ...any others?
- But these are externally visible operations - side-effecting code
 - How do we manage this in a pure functional language like Haskell?
 - How do we integrate pure and impure code?

<https://sli.do>, code 4098 384

CODE WRITTEN IN HASKELL
IS GUARANTEED TO HAVE
NO SIDE EFFECTS.

...BECAUSE NO ONE
WILL EVER RUN IT?



<https://xkcd.com/1312/>

Randall Munroe

CC BY-NC 2.5 Deed

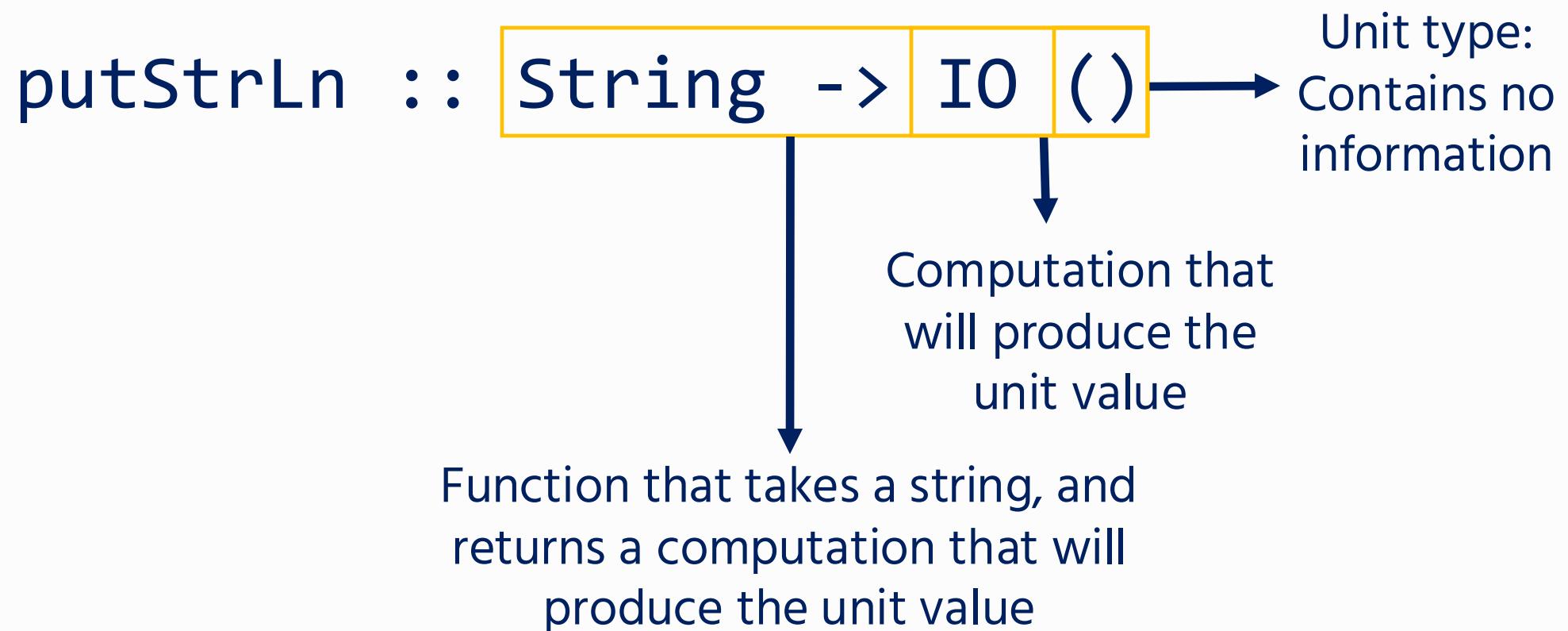
What if we did the naïve thing?

$$\begin{array}{c} (\text{read getLine :: Int}) - (\text{read getLine :: Int}) \\ \xleftarrow{\quad} \qquad \qquad \qquad \xrightarrow{\quad} \\ 5 - (\text{read getLine :: Int}) \qquad \qquad \qquad (\text{read getLine :: Int}) - 5 \\ \downarrow \qquad \qquad \qquad \downarrow \\ 5 - 10 \qquad \qquad \qquad 10 - 5 \\ \downarrow \qquad \qquad \qquad \downarrow \\ -5 \qquad \qquad \qquad 5 \end{array}$$

- We lose Church-Rosser
- We also lose **referential transparency** (the property that we can replace a piece of code with the value it produces)
- It also wreaks havoc with lazy evaluation
- ...and therefore we lose all our reasoning power

IO Types

- Key idea: Each side-effecting operation is marked with a *type constructor*, IO



Mixing pure functions and IO computations

reverse getLine 

```
reverse :: String -> String  
getLine :: IO String
```

A String is not the same as an IO String!

A String is **data**.

An IO String is a **computation which produces a String**.

Side-effecting computations are **always** marked in their types.

Mixing pure functions and IO computations

```
getAndPrintReverse :: IO  
getAndPrintReverse = do  
    str <- getLine  
    let revStr = reverse str  
    putStrLn revStr
```

Print reversed string to console

Mark that we're putting together a computation

Within a 'do' block, the <- operator allows us to give a name (str) to the result of an IO operation. Here str has type String

Bind result of (pure) reverse function to revStr using let (you don't need 'in' in a do block)

IO computations that return values

```
getAndReverse :: IO String  
getAndReverse = do  
    str <- getLine  
    let revStr = reverse str  
    return revStr
```

Here we are defining an IO computation that returns a String

Note that we have to use the **return** function
`return :: a -> IO a -- roughly`
(since revStr is of type String, and we need an IO String)

The Bigger Picture

- Every Haskell program has an entry point, `main`

```
main :: IO ()  
main = do  
    line <- getLine  
    putStrLn (makeUpper line)
```

```
makeUpper :: String -> String
```

- `main` function is evaluated when program is run
 - It can then make use of other functions with `IO` type
- GHC runs everything as an `IO` computation – which is why we can print things out
- Good practice: keep as much of the program as **pure as possible**
 - Pure functions are much easier to test and reason about

Escaping IO?

- There is no* way to “project” a value out of an IO computation
 - While you might want a function with type `IO String -> String`, this doesn’t make sense: we would run into the same issues with Church-Rosser as before
- Instead, think of IO as though you are using do-notation to build a bigger computation by stringing together smaller IO computations, with `main` as your entry point

*I don't like lying to you.

If you import `System.IO.Unsafe`, there is a function

```
unsafePerformIO :: IO a -> a
```

This is not idiomatic Haskell and should only be used when you **really** know what you're doing. **You won't need it in this course.**



Sli.do Break

Trace Debugging

- That said, it is sometimes useful to do ‘print debugging’ where we wish to print some program state to the console
- We can do this using the **trace** function

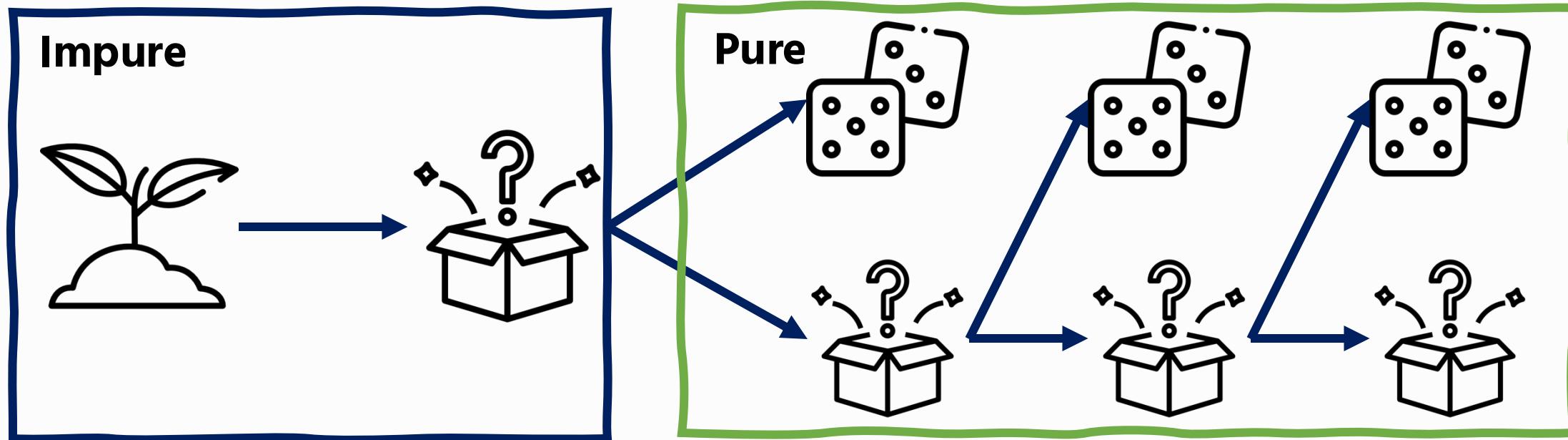
```
import Debug.Trace  
trace :: String -> a -> a  
      trace "returning 42" 42
```

- This uses `unsafePerformIO` under the hood, and should **only** be used for debugging

Beyond Console IO

- IO encompasses a large number of impure operations beyond just reading / writing to a console
- Getting current time:
 - `getCurrentTime :: IO UTCTime`
- File operations:
 - `readFile :: FilePath -> IO String`
 - `writeFile :: FilePath -> String -> IO ()`
- Also sockets, graphics, printing, spawning processes...
- Anyone think of any others?

Pseudo-Random Number Generation



- Random number generation might **seem** to be an impure operation
- In fact, a pseudo-random number generator generates a random value, and a new generator
- Only **seeding** the PRNG is impure, generation is pure

Reference Cells

- With IO we can make use of **mutable reference cells** that store some data, and its contents can be changed

`newIORef :: a -> IO (IORef a)`

Creates a new IO reference with an initial value of type a

`readIORef :: IORef a -> IO a`

Reads the content of an IORef

`writeIORef :: IORef a -> a -> IO ()`

Writes to an IORef

Sli.do Questions (1)

- Multiple choice in class test?
 - This was my (Simon)'s bad for suggesting it might contain MCQs – I'd extrapolated from previous Moodle quizzes / class tests. Nevertheless hopefully people still found it a fair assessment (this was the sense I got from the reflections).
- That fact about the fears seems made up
 - I am not fact checking these ☺ Added a disclaimer.
- Can Slide 10 be fixed as the text doesn't make sense without animations?
 - Yes – fixed (although I had to look up the meaning of "tgt")
- Can Monads be explained in detail?
 - Certainly! This is the subject of Thursday's lecture – and next week.

Sli.do Questions (2)

- Reference cells were a little rushed
 - They're not particularly important, but I'm happy to go over them again at the start of Thursday's lecture.
 - In short, though, a reference cell is like a type-safe pointer.
We can create an `IORef#Int` that is a reference to an integer (for example).
We can then read the current value of that reference, or update its contents to a new integer.
- How can you have reference cells in Haskell -- I thought everything was immutable?
 - So the `IORef` *itself* is still immutable – it's a pointer. But its *contents* are mutable using the `readIORef` and `writeIORef` operators. Since working with `IORefs` is impure, all creation/read/write operations must be in `IO` computations.
- So reference cell = mutable reference cell, there's no such thing as immutable reference cell so immutable data just has no reference cell?
 - Yes. Data by default is immutable so doesn't need a reference cell. We use a reference cell when we want (controlled) mutability. Whenever I say "mutable reference cell" I mean "reference cell that contains mutable data".

Sli.do Questions (3)

- What's the difference between `putStrLn`, `show`, and `print`?
 - `putStrLn :: String -> IO ()` – this is an IO computation that prints the given string and a newline character to the console
 - `show :: Show a => a -> String` – this is a pure function that, for a member of the `Show` typeclass, returns its string representation
 - `print :: Show a => a -> IO ()` – this is an IO computation that (using the `show` instance) prints the string representation of the given value to the console
- Are `IO ()` and `IO a` the same?
 - No. `IO ()` is the type of a computation that produces the unit value (but may perform some side effects).
 - `IO a` itself is a strange type: the only way something could have type `IO a` is if it did not return (e.g. by erroring or looping forever).
 - However, you may be thinking of the return function `return :: a -> IO a`. The `return` function takes a value of type `a`, and gives back a trivial IO computation that produces that result.

Sli.do Questions (4)

- Why is it `getLine` and then `putStrLn` and not `putStrLnLine`? It annoys me.
 - Good point! (Not my choice, but sensible critique).
 - That said this is a great teachable moment about the difference between '`let`' and '`<-`' in do-blocks. Let's instead go with the convention that we want `putStrLn` and `getLn`...

```
getAndPrintReverse :: IO ()  
getAndPrintReverse = do  
    let getLn = getLine  
    str <- getLn  
    let revStr = reverse str  
    putStrLn revStr
```

Note that
`let getLn = getLine`
is renaming the `getLine` computation
rather than running it and binding the
result. We can then use the `<-` operator
to invoke our new `getLn` computation.

Sli.do Questions (5)

- Could you use the `<-` operator to "bind" the result of ``reverse str`` on ``revStr`` instead of the keyword ``let``?
 - No – it's strictly "let" is for pure computations, "`<-`" is for running an IO computation and naming its result
- So to keep program as pure as possible you use the minimum number of IO operations?
 - It doesn't matter so much the number of IO operations you use. What you want to do is write as many pure functions as you can and not unnecessarily 'pollute' otherwise pure functions with IO.
 - It's better to do the IO in 'main' and then pass the results of user interactions as parameters, for example.

Sli.do Questions (6)

- Can we use do and the <- operator to split Just from a the same way we can split IO under a do operation?
 - Indeed we can! IO is what's known as a monad, and we can use do-notation and <- for all monads. We'll see this on Thursday.
- What do you mean by “escaping” from IO - do you mean converting it to string and returning string without the IO?
 - I meant taking an IO computation (e.g. `getLine :: IO String`) and using it in a pure function (using some imaginary function with type `IO String -> String` – which would allow us to ‘pretend’ that an IO computation is actually pure).

Sli.do Questions (7)

- On slide 17, `revStr` has type `String` and we only return `revStr` so in the type signature why do we write `IO String` instead of just `String`?
 - This is a great question. Remember that when we use a 'do' block we are describing how to build up a computation. That computation (since it uses IO operations) must be an IO computation, so the final type must be `IO String` (hence why we need to use `return`).
- What do you mean, `return` is 'pure'?
 - 'return' creates a 'trivial' computation that returns the given value (without performing any side effects).
 - My point was that 'return' is a bad name as it confuses people who are familiar with 'return' in Java and C. So you can also use the function called 'pure' in Haskell instead of 'return' (indeed this is what's done in Idris).
- So IO has no referential transparency?
 - Indeed – IO computations are not referentially transparent.

Sli.do Questions (8)

- What is an “IOU”?
 - Typically a note acknowledging debt. But I think you may have misheard me say “IO Unit”, which is the type `IO ()` that describes an IO computation that returns the unit value `()`
 - So the unit type is kind of like futures?
 - No. The unit type on its own is just an empty record. In a pure function it’s not much use, but an IO computation of type `IO ()` typically indicates that the computation doesn’t return a result, but performs a side effect (like `putStrLn`: we print to the console, but don’t return a sensible result).
 - The main comparison to `IO ()` in imperative languages is a void method in Java. But since Haskell is expression based, we need to return *something*, so we return `()`.
 - On the other hand you can think of a future as a placeholder value. You make a request, and get back a `Future a`. You can then block waiting for the result.
 - There are implementations of futures in Haskell (see e.g. <https://hackage.haskell.org/package/futures-0.1/docs/Futures.html>, and the Concurrent Haskell MVar structure: <https://hackage.haskell.org/package/base-4.20.0.1/docs/Control-Concurrent-MVar.html#t:MVar>).
- Since futures make use of concurrency, they are necessarily IO computations.

Sli.do Questions (9)

- In strict evaluation it's always left one evaluated first then right, but in lazy evaluation you never know the order?
E.g. in `(read getLine :: Int) - (read getLine :: Int)`
 - It depends on the language semantics. The main thing with strict evaluation is that all arguments are evaluated prior to making a function call – the order is implementation-dependent (for example I believe OCaml has a right-to-left evaluation order).
 - With lazy evaluation, the arguments are only evaluated on-demand. If there are multiple possible reduction paths then (assuming pure computations) you can evaluate them in any order and get the same result.
 - The point with this example is that sometimes ordering really matters when doing side-effecting computations, so we need to separate (at the type level) pure and impure computations.

Sli.do Questions (10)

- Functor vs map?
 - Jeremy will go into this in more detail later in the course – so he will explain it much more there.
 - But a ‘functor’ is a generalisation of the ‘map’ function we have seen for lists: it allows us to apply a function to the contents of a data structure without changing its structure. For example...

```
map      :: (a -> b) -> [a]      -> [b]
treeMap :: (a -> b) -> Tree a -> Tree b
```
 - We are able to generalise this using the Functor typeclass, which we separately introduce for trees and lists, then we can use the fmap function:
 - `fmap :: Functor f => (a -> b) -> f a -> f b`

Sli.do Questions (11)

- So 'type class' is the only class in Haskell - there is no other 'class'?
 - Yes. The object-oriented notion of a 'class' is not present in Haskell (since Haskell doesn't have objects). Think of them as separate things.
 - A class in Java is a template for creating objects (strictly speaking).
 - A typeclass in Haskell is more like a Java interface. It specifies a list of operations that a data type must implement in order to be a member of a class (e.g. to be a member of the Leggy typeclass, a type a must implement `numLegs :: a -> Int`).
- Parametric polymorphism vs ad-hoc polymorphism? So parametric = fn can accept val of any type, adhoc = fn behaves differently depending on each different type?
 - Yes, that's exactly it. Parametric polymorphism means that we can't make any assumptions about the precise types, but then the function works for any type. For example:
`rotate :: (a,b,c) -> (c, a, b)` is parametrically polymorphic because we just shift around the elements of a tuple without doing any concrete operations on them.
 - Then as you say, ad-hoc polymorphism allows us to implement a function with the same name differently on each type.

Sli.do Questions (12)

- So is it that all adhoc polymorphic functions must be parametric polymorphic (so it can accept different types in the first place) but not the other way around?
 - Mostly correct. The idea is that we write a type signature, say numLegs...
 - numLegs :: Insect -> Int
 - Then we think – could this be useful for other data types? And we can generalise it as a typeclass (like our Leggy typeclass). And then the type of numLegs becomes...
 - numLegs :: Leggy a => a -> Int
 - Which can be read: numLegs is defined for any 'a', *as long as* that 'a' is a member of the Leggy typeclass. So unlike parametric polymorphism we're allowed to make *some* assumptions about the type we have.

Sli.do Questions (13)

- IO is like a generator then?
 - I would say a generator is more specific to lists / arrays, whereas an IO computation is a description of a side-effecting operation.
- Why don't you need 'in' in `main` when you use `let`?
 - When we're using 'do' notation we can drop the 'in' (it's syntactic sugar). We'll go more into specifics about do-notation on Thursday.
- Is do not a while loop if in `main`?
 - No, please don't confuse 'do'-notation and the imperative 'do-while' looping construct; there are no similarities between the two.
- Why does `getLine` assume input is a string? What if you want to perform `int` operations? Do you need to convert?
 - Yes – you need to convert manually. Otherwise Haskell would be guessing the meaning of your input. It's better to return it as a string and leave you to be explicit. The lab sheet has a question on this.

Sli.do Questions (14)

- How did you make random number generator generate the same set of values each time? Did it not have a seed at all in this case?
 - There were two times that I made the RNG give the same set of values. The first was when I explicitly seeded it with '123'.
 - The second was when I seeded it with the number of seconds since the Unix epoch. Invoking the function twice in the same second would give the same seed and generate the same numbers.
- When you say only seeding is impure is it because just the seeding dynamically chooses the probability distribution?
 - The main thing is that we need some entropy (randomness) to pick the initial seed. This can be things like the current time, mouse position, contents of memory – but picking any of these things is impure.

Sli.do Questions (15)

- In Haskell aren't functions defined as a chain of single-argument functions by default? How is monad type class special when it comes to 'chaining'?
 - So the chaining you are thinking of is currying. Here we're building up an n-argument function by nesting n single-argument functions. The chaining here is based on having access to all parameters.
 - Monadic 'chaining' is different: we're building up a description of a side-effecting computation. The 'chaining' allows us to use the results of previous side-effecting computations in the remainder.
 - This should hopefully become clear when I give the concrete typeclass definitions next time. The main thing about this lecture was to build up the intuition.

Sli.do Questions (16)

- Type constructor is different from type signature?
 - Yes. A type signature tells us what type a function has, e.g.
`add :: Int -> Int -> Int`
 - A type constructor is something like Maybe, which takes other types as parameters (allowing us to have Maybe Int, Maybe Bool, etc.)
- Would '`x <- [1..]`' break lazy evaluation?
 - This has a nuanced answer and should become a little clearer next lecture. In essence we couldn't write `x <- [1..]` in an IO computation, since `[1..]` doesn't have an IO type. We can build a "list" computation though, which you'll see next time. Lazy evaluation still works as expected.

Sli.do Questions (17)

- Can you explain the trace function a little more?
 - `trace :: String -> a -> a`
 - Here “String” is the message to print to the console. The “a” parameter is what we want to return.
So `trace “returning 42” 42` will print “returning 42” to the console, and return 42.
 - Here is an example for tracing a factorial function.

```
fact :: Int -> Int
fact 0 = trace (show 1) 1
fact n = trace ((show n) ++ " * ") (n * fact (n-1))
```

```
ghci> fact 5
5 *
4 *
3 *
2 *
1 *
1
120
```

Sli.do Questions (18, phew!)

- How does it wreak havoc with lazy evaluation? With strict evaluation it still accepts user input twice?
 - This is a good question. “Wreak havoc” probably was a bit hyperbolic, but it does mean that we need to be careful with IO.
 - The point is – laziness means that we only evaluate an expression when it is needed (and indeed this means we can have *multiple reduction paths*, which in turn causes problems with Church-Rosser).
 - do-notation and IO computations ensure that the IO operations have a strict sequencing, which gives us some predictability about when they are evaluated.

Wrapping Up

- **Today**
 - The need for impurity
 - The IO type
 - do-notation
- **Next up**
 - Building up other types of computation: for example, Maybe computations, nondeterministic computations
 - The 'M' word...
- **Questions?**

I'M A FUNCTIONAL PROGRAMMER

WHAT'S A FUNCTIONAL PROGRAMMER?

do, code 4098 384



IT MEANS
HE IS AFRAID
OF SIDE EFFECTS

I'M NOT
AFRAID OF
SIDE EFFECTS



AHH MUTATING GLOBAL STATE

STOP IT PATRICK,
YOU'RE SCARING HIM!



Functional Programming (H) Lecture 8: Introduction to Monads

Simon Fowler & Jeremy Singer

Semester 1, 2024/2025



<https://sli.do>, code 9308 697



University
of Glasgow

Today

- Last time
 - Introduction to impurity and IO in Haskell
- Lecture today will look at various patterns of computation
 - Potentially failing computations (Maybe)
 - Nondeterministic computations (List)
 - Another look at IO
- From there we will introduce **monads**, a general way of building up computations
 - The Monad typeclass
 - do-notation
- Please ask questions!

Recap: IO Computations

```
getAndPrintReverse :: IO  
getAndPrintReverse = do  
    str <- getLine  
    let revStr = reverse str  
    putStrLn revStr
```

Print reversed string to console

Mark that we're putting together a computation

Within a 'do' block, the <- operator allows us to give a name (str) to the result of an IO operation. Here str has type String

Bind result of (pure) reverse function to revStr using let (you don't need 'in' in a do block)

Potentially-Failing Computations

Remember the Maybe type?

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

- The `Maybe a` data type is like a ‘type-safe null’, and is used to define potentially failing computations.
- For example, both `Just 5` and `Nothing` can have type `Maybe Int` – although only `Just 5` actually contains an `Int`.
- Whenever we want to use a value of type `Maybe a`, we’ll need to case split to see whether it contains a value or not.

Managing Multiple Maybes

The Maybe type allows us to return `Nothing` if a computation fails

```
safeDiv :: Int -> Int -> Maybe Int  
safeDiv x 0 = Nothing  
safeDiv x y = Just (x `div` y)
```

Suppose we want to use the `safeDiv` function twice, and add the results.
How do we deal with multiple values with a Maybe type?

Managing Multiple Maybes

```
divAndAdd :: Int -> Int -> Int -> Int -> Maybe Int
divAndAdd x y d1 d2 =
  case (safeDiv x d1, safeDiv y d2) of
    (Just x', Just y') -> Just (x' + y')
    (_, _)                 -> Nothing
```

`divAndAdd`: Divide numbers x by d_1 , and y by d_2 , and add the results together

We could case split on both results of `safeDiv`: what are the advantages and disadvantages here?

This is horrible to write!

- We are having to write a lot of boilerplate to check whether a value is present or not
- Worse, this style may lead to deeply-nested case statements, making code very difficult to read, write, and maintain
- Is there a way that we can write `divAndAdd` as if each **safeDiv computation succeeds**, and have it automatically evaluate to Nothing if any subcomputation fails?

Can we do better?

- Remember that we have **higher-order functions**
 - ...and using backticks we can use any function as an infix operator
- How about we write a function that takes a `Maybe a`, and a function that we can pass the unwrapped value to if it exists? If it doesn't, we'll return `Nothing` overall.

The potentially failing computation

The function to run with the `Just` value, if it exists

The final result

```
withJust :: Maybe a -> (a -> Maybe b) -> Maybe b
withJust Nothing _ = Nothing
withJust (Just x) f = f x
```

Using our withJust function

```
withJust :: Maybe a -> (a -> Maybe b) -> Maybe b  
withJust Nothing _ = Nothing  
withJust (Just x) f = f x
```

```
withJust (safeDiv 10 5) (\x ->  
  withJust (safeDiv 20 10) (\y ->  
    Just (x + y)))
```

Using our withJust function

```
withJust :: Maybe a -> (a -> Maybe b) -> Maybe b  
withJust Nothing _ = Nothing  
withJust (Just x) f = f x
```

The potentially failing computation

Our withJust function, that tries to unwrap but returns Nothing otherwise

```
safeDiv 10 5 `withJust` (\x ->  
safeDiv 20 10 `withJust` (\y ->  
Just (x + y)))
```

In the continuation function, x has type Int
(as we've already unwrapped it in withJust)

Using our withJust function

```
withJust :: Maybe a -> (a -> Maybe b) -> Maybe b  
withJust Nothing _ = Nothing  
withJust (Just x) f = f x
```

```
safeDiv 10 5 `withJust` (\x ->  
safeDiv 20 2 `withJust` (\y ->  
Just (x + y)))
```

```
Just 2 `withJust` (\x ->  
safeDiv 20 2 `withJust` (\y ->  
Just (x + y)))
```

```
safeDiv 20 2 `withJust` (\y ->  
Just (2 + y))
```

```
Just 10 `withJust` (\y ->  
Just (2 + y))
```

```
Just (2 + 10)
```

```
Just 12
```

Using our withJust function

```
withJust :: Maybe a -> (a -> Maybe b) -> Maybe b  
withJust Nothing _ = Nothing  
withJust (Just x) f = f x
```

```
safeDiv 10 5 `withJust` (\x ->  
safeDiv 20 0 `withJust` (\y ->  
Just (x + y)))
```



```
Just 2 `withJust` (\x ->  
safeDiv 20 0 `withJust` (\y ->  
Just (x + y)))
```



```
safeDiv 20 0 `withJust` (\y ->  
Just (2 + y))
```



```
Nothing `withJust` (\y ->  
Just (2 + y))
```



```
Nothing
```

Nondeterministic Computations

Nondeterministic Computations

- Sometimes we might want to have a computation that returns **multiple possible results**: we call this a **nondeterministic computation**.
- For example, a coin toss can either return heads or tails. How about if we wanted to build a list of the possible outcomes of two tosses?

```
coinToss :: [CoinToss]
coinToss = [Heads, Tails]
```

We want...

```
[(Heads,Heads), (Heads,Tails), (Tails,Heads), (Tails,Tails)]
```

Nondeterministic Computations

- How do we implement this?
- We want to draw the first element from the first list, and use that result for the remaining computation...
 - Then draw the first element of the second list, and return the pair
 - Then draw the second element of the second list, and return the pair
 - And repeat for the second element of the first list
- Can we use a similar approach as with Maybe?

```
withEach :: [a] -> (a -> [b]) -> [b]
withEach []      _ = []
withEach (x:xs) f = f x ++ (withEach xs f)
```

Nondeterministic Computations

```
withEach :: [a] -> (a -> [b]) -> [b]
withEach []     _ = []
withEach (x:xs) f = f x ++ (withEach xs f)
```

(equivalently)

```
withEach :: [a] -> (a -> [b]) -> [b]
withEach xs f = concat (map f xs)
```

Coin tosses using withEach

```
withEach :: [a] -> (a -> [b]) -> [b]
withEach [] _ = []
withEach (x:xs) f = f x ++ (withEach xs f)
```

```
twoCoinTosses :: [(CoinToss, CoinToss)]
twoCoinTosses =
  coinToss `withEach` (\x ->
  coinToss `withEach` (\y ->
    [(x, y)]))
```

```
[(Heads, Heads)]
  ++
[(Heads, Tails)]
  ++
[(Tails, Heads)]
  ++
[(Tails, Tails)]
```

```
=
[(Heads,Heads), (Heads,Tails),
 (Tails,Heads), (Tails,Tails)]
```

We can of course also use a list comprehension!

```
twoCoinTosses :: [(CoinToss, CoinToss)]  
twoCoinTosses = [ (x, y) | x <- coinToss,  
                    y <- coinToss ]
```

Note that this requires **multiple generators**, which we **can't express** with just map and filter.

In fact, list comprehensions are syntactic sugar, implemented using (an analogue of) `withEach`

Another Look at IO

Thinking again about IO

```
getAndPrintReverse :: IO ()  
getAndPrintReverse = do  
    str <- getLine  
    let revStr = reverse str  
    putStrLn revStr
```

- So far we've used do-notation to build up IO computations
- If you think about it, though, we are describing a computation that:
 - Runs an IO computation
 - Uses its result in the rest of the computation

Writing IO computations without do

```
withIOResult :: IO a -> (a -> IO b) -> IO b
```

```
getAndPrintReverse :: IO ()  
getAndPrintReverse =  
    getLine `withIOResult` (\str ->  
        let revStr = reverse str in  
        putStrLn revStr)
```

- The implementation of `withIOResult` maps to a **primitive** and is handled by the runtime system
- It runs the IO computation, and applies the given function to the result
- Nevertheless it allows us to write our `getAndPrintReverse` function **without** do-notation

Monads

Can you see a pattern?

`withJust :: Maybe a -> (a -> Maybe b) -> Maybe b`

`withEach :: [a] -> (a -> [b]) -> [b]`

`withIOResult :: IO a -> (a -> IO b) -> IO b`

In each case, we have:

- A computation producing a result (be it potentially-failing, nondeterministic, or side-effecting)
- A function that builds a bigger computation from the result
- The overall result being the bigger computation

Monads

- Here we are, finally! The M-Word!
- We can **generalise** the pattern we have seen so far:
 - Maybe, List, and IO are all instances of a more general pattern known as a **monad**
- To be a monad, a data type needs two things*:
 - A way of constructing a trivial computation
 - $a \rightarrow \text{Maybe } a$ (we can use Just)
 - $a \rightarrow [a]$ (we can use the singleton list constructor)
 - $a \rightarrow \text{IO } a$ (it's a library function, but it creates a pure computation without side effects)
 - And a way of building a larger computation from the results of a previous one

*plus they need to satisfy some laws – more on this later

The Monad Typeclass

Every monad must also be an applicative functor (not important for now – we will talk about applicatives later)

Return: “Injects” a pure value into the monad. For example, return 5 = Just 5 in the Maybe monad

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

>> (pronounced “sequence”): runs but ignores first computation, returns result of second

Derivable from definition of (=>)

$m1 \ (.) \ m2 = m1 \ =>= \ \lambda _ \ -> m2$

>> (pronounced “bind”): allows us to build up a computation.

Takes a computation of type $m\ a$, and a function $a \ -> m\ b$ to build a new computation $m\ b$, and returns $m\ b$.

What is a monad?

- You may have heard many definitions of monads
 - “A monoid in the category of endofunctors”
 - “Like a burrito / spacesuit”
- There is nothing magical about them ☺

A monad is just a data type that implements the Monad typeclass, and satisfies the Monad laws.

(Jeremy will go into detail on the monad laws, but they just ensure that our instance satisfies some sensible properties.)

Example: The Maybe Monad

- The Maybe Monad is determined: it will stop at Nothing!
- Idea: Can sequence many computations, but if any return Nothing then the whole computation returns Nothing
- We are just implementing `>>=` the same as our `withJust` function!

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

Managing Multiple Maybes Monadically

```
divAndAdd' :: Int -> Int -> Int -> Maybe Int  
divAndAdd' x y d1 d2 = safeDiv x d1 >>= (\res1 ->  
    safeDiv y d2 >>= (\res2 ->  
        return (res1 + res2)))
```

Here we use the **bind** operator `>>=` and provide a function where we assume each safe division has succeeded

We can then use **return** to create a `Maybe Int` from `res1 + res2`

If any sub-computation fails, then the whole result will be `Nothing`

(Now do you understand the Haskell logo?)



do-Notation

do-notation

- We introduced IO using do-notation to get across the intuition of building up IO computations
- Since IO is a monad, we can also write IO computations using explicit `>>=` notation (without needing `do`)

```
greetReverse :: IO ()  
greetReverse = do  
    name <- getLine  
    let reverseName =  
        reverse name  
    putStrLn "hello"  
    putStrLn reverseName
```



```
greetReverse :: IO ()  
greetReverse =  
    getLine >>= \name ->  
    let reverseName =  
        reverse name in  
    putStrLn "hello" >>  
    putStrLn reverseName
```

Rules of do-Notation

do-Notation is syntactic sugar

Widely used & very useful --- but sometimes quicker / more concise to write the monadic expression directly

do-Notation	Monadic notation
<code>do x <- M N</code>	<code>M >>= \x -> N</code>
<code>do M N</code>	<code>M >> N</code>
<code>do let x = M N</code>	<code>let x = M in N</code>

do-notation for other monads

- Do notation works for **any** data type that is a member of the monad typeclass (not just for IO!)
- We can rewrite our Maybes example from earlier using do notation:

```
divAndAdd' :: Int -> Int -> Int -> Maybe Int
divAndAdd' x y d1 d2 = do res1 <- safeDiv x d1
                           res2 <- safeDiv y d2
                           return (res1 + res2)
```

Monad typeclass hierarchy

- Recently, the Monad typeclass changed so that each Monad must also be an instance of two other typeclasses called Functor and Applicative
 - We will describe these in more detail later in the course
- There is another proposal to remove `return` from the Monad typeclass, and rely on:
`pure :: (Applicative f) => a -> f a`
from Applicative
- This makes sense mathematically, but is terrible pedagogically ☺
- Explicit definitions of `return` now trigger a warning in recent versions of GHC
- For the purposes of this course, use the definitions given in the lectures

Wrapping Up

- **Today**
 - The Monad typeclass
 - The Maybe and List monads
 - do-notation
- **This is my last “core” lecture!**
 - I will be still be here, though, to introduce + support the coursework, and will deliver a research lecture at the end
 - It’s been very fun teaching you all, thanks for the great questions + reflections
- **Next up**
 - Some more interesting monads! Reader / Writer / State, and more fun with Jeremy
- **Over to you**
 - Have a go at the exercise sheet / come to the lab (this one is very important ☺)

Functional Programming (H) – Lecture 9 – Mon 28 Oct

More Monads

Jeremy.Singer@glasgow.ac.uk

Today we are going on a deep dive with the monad typeclass. Our *learning objectives* are:

- Gain increasing familiarity with monad concepts and functions
- Recognize a number of common library monads
- Become confident at using appropriate monads in your Haskell programs

Introduction

So, let's revisit the `Monad` typeclass. Recall it has two characteristic methods, i.e. `return` and `bind (>>=)`. The `return` function “puts something into” the monad — i.e. boxes it up in the structure. The `(>>=)` function injects a monad value into arbitrary function `f` as a parameter, with the added constraint that `f` returns a value in the same monad — i.e. `(>>=)` wangles the types so functions can take monadic values as parameters.

Maybe Monad

We will motivate these (somewhat abstract) concepts with the now-familiar `Maybe` datatype, which is monadic.

Consider a function `allButLast :: [Char] -> Maybe [Char]`. This function will return the first $(n-1)$ characters of a string with (n) characters, wrapped up as a `Just` value. If the input string is empty, then the function evaluates to `Nothing`.

```
allButLast [] = Nothing
allButLast [x] = Just []
allButLast (x:xs) =
  let rest = allButLast xs
  in case rest of
    Nothing -> Just [x]
    (Just xs') -> Just (x:xs')
```

Now let's think about what happens when we repeatedly apply this function to a string input value ... eventually we will run out of characters so we will bottom out at the `Nothing` value:

```
take 10 $ iterate (\x -> x >>= allButLast) (Just "abcdef")
```

Can you see what happens when we try to bind a function with a `Nothing` value? The result (thanks to the definition of `(>>=)`) is always `Nothing`.

Exercise for the reader: Rework the above example with the `Either` datatype.

Identity Monad

Next we will look at the `Identity` monad, which might seem a little pointless but it's a useful base case.

We need to `import Control.Monad.Identity`, which might require some library configuration in `ghci` ... perhaps `:set -package mtl` on the interactive prompt.

We can put something into the `Identity` monad with the `return` function:

```
(return 42) :: Identity Int
```

and we can apply a function to `Identity` monadic values with `(>>=)`:

```
((return 42) :: Identity Int) >>= (\x -> return (x+1))
```

We extract a value out of an `Identity` computation by *running* the monad, using the `runIdentity :: Identity a -> a` function.

```
runIdentity $ ((return 42) :: Identity Int) >>= (\x -> return (x+1))
```

I suppose the equivalent case of *running* the `IO` monad is the invocation of the `main` function at top-level in Haskell.

Exercise for the reader: Can you define the `return` and `(>>=)` functions for `Identity`. They are completely trivial!

List monad

We can think of using a monad as being like putting a value into a structure, *elevating* or *lifting* the value into the monad. What structures do we already know in Haskell? Well, the list is probably the simplest ... and guess what? Yes, list is a monad.

List `return` simply puts value into a singleton list (i.e. syntactically, just put square brackets around it).

```
listreturn :: a -> [a]
listreturn x = [x]
```

List `bind` takes each element out of the list, applies a function to that element, giving a list result, then concatenates all the results into a single, new results list. The key point (which confuses some people) is that all the results are glued together into a single list, rather than being separate sublists ... like they might be with a `map` function call.

```
listbind :: [a] -> (a->[b]) -> [b]
listbind xs f = concat $ map f xs
```

Here is a simple example of list bind:

```
ghci> let f x = [x,x]
ghci> f 2
[2,2]
```

```
ghci> [1,2,3] >>= f  
[1,1,2,2,3,3]
```

Here is another example:

```
ghci> let f = \x -> (show x) ++ " mississippi... "  
ghci> [1,2,3] >>= f  
"1 mississippi... 2 mississippi... 3 mississippi... "
```

So we see that list bind is reminiscent of a `join` in Python, or a `flatMap` in Java/Scala.

Reader, Writer and State Monads

Warning: now we are accelerating to rocket speed for the second half of this lecture.

So far we have looked at fairly straightforward monads, many of which you have seen before. We are going to look at three useful library Monads. For each one, we will look at their API and a typical use case. These examples *might* be helpful for your coursework, coming up in a few weeks.

These three monads involve an *environment*, which we pass around, threading it from function context to function context. Typical use cases for each monad are as follows:

- Reader - shared environmental configuration
- Writer - logging operations
- State - computing a large value recursively

Reader

The Reader monad is also known as the *environment* monad. It's useful for reading fixed values from a shared state environment, and for passing this shared state between multiple function calls in a sequence.

Here is a trivial example:

```
import Control.Monad.Reader  
-- ^^^ may need some GHCi hackery ...  
  
hi = do  
    name <- ask  
    return ("hello " ++ name)  
  
bye = do  
    name <- ask  
    return ("goodbye " ++ name)  
  
conversation = do  
    start <- hi  
    end <- bye  
    return (start ++ " ... " ++ end)
```

```
main = do
    putStrLn $ runReader conversation "jeremy"
```

The Reader structure has two parameters, one of which is the environment (here a string) and the other is the result of the computation. So the Reader datatype is parameterized on two type variables: Reader environment result

The `runReader :: Reader r a -> r -> a` function takes a Reader expression to execute, along with initial environment, and extracts the final value from it.

Writer

The Writer monad builds up a growing sequence of state (think about logging a sequence of operations). Officially, this state is a *monoid*, but we will only consider String values for today.

The `tell` function appends something to the log.

```
import Control.Monad.Writer

addOne :: Int -> Writer String Int
addOne x = do
    tell ("incremented " ++ (show x) ++ ",")
    return (x+1)

double :: Int -> Writer String Int
double x = do
    tell ("doubled " ++ (show x) ++ ",")
    return (x*2)

compute =
    tell ("starting with 0,") >> addOne 0 >>= double >>= double >>= addOne

main = do
    print $ runWriter (compute)
```

State

Finally, the State monad. This is very similar to the Reader and Writer monads, in that it allows us to pass around a shared state, but also to update it.

We can get the current state, or put a new value into the state.

Let's do this with the fibonacci function ...

```
import Control.Monad.State

fibState :: State (Integer, Integer, Integer) Integer
fibState = do
    (x1,x2,n) <- get
```

```
if n == 0
then return x1
else do
  put (x2, x1 + x2, n - 1)
  fibState

main = do
  print $ runState fibState (0,1,100)
```

Homework and Further Reading

- Can you redefine another interesting recursive function using the State monad? Why is it more efficient?
- Read about these utility monads at <https://learnyouahaskell.com/for-a-few-monads-more> or <https://www.adit.io/posts/2013-06-10-three-useful-monads.html>
- Here is a more complex piece about the State monad: <https://hackage.haskell.org/package/state-monad-a-bit-of-currying-goes-a-long-way>
- Did Simon warn you about monad analogies? Look up the *monads as burritos* or *monads as spacesuits* explanations online. Do these make sense?

Finally, don't worry if we went way too fast today. We will consolidate this material on Thursday and in the lab on Friday morning.

Functional Programming (H) – Lecture 10 – Thu 31 Oct

More Monads (2)

Jeremy.Singer@glasgow.ac.uk

Today we are continuing our deep dive with the monad typeclass. Our *learning objectives* are:

- Gain increasing familiarity with monad concepts and functions
- Recognize a number of common library typeclasses
- Understand the relationships between library typeclasses including `SemiGroup`, `Monoid` and `Foldable`

Introduction

We are going to start in a fairly abstract, mathematical mood today. This is abstract algebra, probably more suited to a Maths course instead of Computing Science. If you want a deeper formal understanding, check out *category theory* online – but that's beyond the scope of this course.

Semigroups

Think about a set of elements S with an associative binary operation \oplus . In Haskell-speak, \oplus has type $S \rightarrow S \rightarrow S$, i.e. it takes two parameter values from S and returns a single result value in S . The only constraint imposed on \oplus is *associativity*, i.e. $(x \oplus y) \oplus z = x \oplus (y \oplus z)$.

This abstract mathematical structure, (S, \oplus) is known as a *semigroup*. There is a corresponding Haskell typeclass, called `Data.Semigroup` with the characteristic function being the infix operator `(<>) :: a -> a -> a`.

The Haskell typeclass definition is as follows:

```
class Semigroup m where
  (<>) :: m -> m -> m
```

Note that Haskell is *not* expressive enough to specify the associativity constraint ... we have to be aware of this as programmers.

Examples of semigroups include:

- the set of positive integers with addition.
- the set of all finite strings over a fixed alphabet with string concatenation
- 2×2 square nonnegative matrices with matrix multiplication

Monoids

A *monoid* is a set S with an associative binary operation \oplus and an identity element e . The identity element must satisfy the constraint that: $\forall a \in S. e \oplus a = a$ and $a \oplus e = a$

In effect, a monoid is a semigroup with an identity element. The identity element is unique, within the monoid.

The Haskell typeclass definition is as follows:

```
class Semigroup m => Monoid m where
    mempty :: m

    -- defining mappend is unnecessary, it copies from Semigroup
    mappend :: m -> m -> m
    mappend = (<>)

    -- defining mconcat is optional, since it has the following default:
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

All instances of `Monoid` must satisfy the following laws:

```
-- left and right identity
x <>> mempty == x
mempty <>> x == x

-- Associativity
(x <>> y) <>> z == x <>> (y <>> z)
```

Again, we can't express these constraints in the typeclass definition — it's up to the Haskell developer to ensure the laws are respected by `Monoid` instances.

List as a Monoid

The most straightforward example of a monoid instance is the list datatype.

```
instance Semigroup [a] where
    (++) = (++)

instance Monoid [a] where
    mempty = []
```

The list concatenation operation is `mappend`, and the empty list is the identity element. So we can construct lists using a new syntax now ... `[1] <>> [2] <>> [3,4,5] <>> mempty` etc.

Sum as a Monoid

It's easy to see why lists are monoids - the `mempty` element and the `mappend` operation are obvious. How about integers? If the `mappend` operation is integer addition, then the `mempty` element is 0.

Let's express this in Haskell:

```
newtype Sum n = Sum n

instance Num n => Semigroup (Sum n) where
  Sum x <>> Sum y = Sum (x + y)

instance Num n => Monoid (Sum n) where
  mempty = Sum 0
```

(Actually, there is a library `Sum` datatype defined in `Data.Semigroup` already.) Now we can construct integer values using monoid syntax:

```
Sum 0 <>> Sum 1 <>> Sum 100
```

Writers use Monoids

Do you remember, in the previous lecture, we introduced the `Writer` typeclass? We explained how a `Writer` instance accumulates an append-only log of data. Last time, the append-only log was a `String`. However now we understand than any `Monoid` instance would be suitable for an append-only log. Let's revisit `Writer` and build up a `Sum` accumulator value in the log, counting how many operations we perform.

```
-- our writer state accumulates the number of
-- arithmetic operations performed
-- (i.e. subtractions and multiplications)

fact :: Integer -> Writer (Sum Integer) Integer
fact 0 = return 1
fact n = do
  let n' = n-1
  tell $ Sum 1
  m <- fact n'
  let r = n*m
  tell $ Sum 1
  return r
```

Foldables use Monoids

Earlier in the course, we saw the use of `foldl` and `foldr` to *reduce* lists of values to single summary values. For example, consider `foldr (+) 0 [1..10]`.

Now we know about monoids, we recognize that if we have an identity element `mempty`, and an associative combining function `mappend`, then we can easily fold over an arbitrary collection of monoid values. If we import `Data.Foldable`, then we can use `fold :: (Foldable t, Monoid m) -> t m -> m` and `foldMap :: (Foldable t, Monoid m) -> (a -> m) -> t a -> m`, which, when given a Monoid `m`, these functions know how to aggregate values using `mappend`.

```

fold [[1,2], [3,4], [5]]
-- evaluates to [1,2,3,4,5]

fold (Sum (Sum 3))
-- evaluates to (Sum 3)

foldMap Sum [1..5]

```

Foldable Trees

Recall the binary Tree container data structure we looked at, earlier in the course. We can make this type an instance of the Foldable typeclass, simply by providing a definition of the `foldMap` function (or, alternatively and equivalently, a definition of the `foldr` function).

Notice that a `Leaf` value will map onto an `mempty` value, and a non-empty `Node` value is recursively combined with `mappend` operations.

```

import Data.Semigroup
import Data.Foldable

data Tree a = Leaf
            | Node a (Tree a) (Tree a)
deriving (Show, Eq)

instance Foldable Tree where
    foldMap :: Monoid m => (a -> m) -> Tree a -> m
    foldMap _ Leaf = mempty
    foldMap f (Node x left right) =
        f x <> foldMap f left <> foldMap f right

```

Homework and Further Reading

- Think about tree order traversal. How would you modify the `foldMap` definition above to do *post-order* traversal?
- Can you define the `foldr` function for this `Tree` data type? You can then remove the `foldMap` definition ... only one of the two is required for a minimal instantiation of `Foldable`.
- Could you define an instance of `Foldable` for another data type you have encountered so far in Haskell?
- Read more about `Foldable` at <https://en.wikibooks.org/wiki/Haskell/Foldable>
- Read more about `Monoid` at <https://caiorss.github.io/Functional-Programming/haskell/monoids.html> or <https://wiki.haskell.org/Monoid>

Finally, a word of warning that these concepts are important. If you feel like you are getting confused, please come to the lab on Friday morning to do the exercises and chat with the friendly tutors.

Functional Programming (H) – Lecture 11 – Mon 4 Nov

Parser Combinators

Jeremy.Singer@glasgow.ac.uk

Today we will explore an example utility library in Haskell, which is an example of a monadic type. Our *learning objectives* are:

- learn how *parser combinators* can be used for incremental construction of a text parser
- gain familiarity at interacting with a conventional Haskell library
- explore a practical use of monads for a real-world problem, i.e. parsing

What is Parsing?

A *parser* is a program that recognizes sentences from a grammar. Sometimes a parser may be hand-written but often it is synthesized from a higher-level grammar description by a parser generator tool (antlr, yacc, etc). The parser is generally *table*-driven or *rules*-driven.

In Haskell, there is an alternative approach involving the use of *parser combinators*, whereby a parser can be constructed incrementally based on combining functions.

Informally, a *combinator* is a higher-order function that combines ‘things’ to create more complex ‘things’ of the same type. For example, we have already looked at list combinators like `map` and `filter`. A *parser combinator* is a higher-order function that combines two simpler parsers (sub-parsers) to create a compound parser.

The Parsec Library

`parsec` is a Haskell parser combinator library. You can install it on your system with cabal:

```
cabal install --lib parsec
```

or provision an interactive interpreter with stack:

```
stack ghci --package parsec
```

Note that Haskell library management is highly complex. You will need `parsec` installed for the lab on Friday, so it’s worth trying to complete this step ahead of time, if you can. Alternatively come to the lab on Friday to get technical support from our team of tutors.

You can verify that your `parsec` installation is working properly in `ghci`:

```
ghci> import Text.Parsec
ghci> parse anyChar "input" "a"
Right 'a'
```

My First Parser

This simplest of parsers recognizes (and accepts) String values containing the single character 'c'.

```
firstParser = (char 'c') :: Parsec String st Char
```

The three type parameters for the Parsec type are, respectively:

- String - type of the input data stream to be parsed
- st - type for user state (unused in this trivial example)
- Char - type of the value recognized by this parser

Let's see how this parser behaves:

```
parseTest firstParser "c"  
parseTest firstParser "cat"  
parseTest firstParser "dog"
```

We notice that our `firstParser` will accept any String that begins with the 'c' character.

There are several different ways to run a parser. Above we have shown the `parseTest` function, which is fine for interactive input. In larger programs, we might use the `parse` or `runParser` functions.

```
parse firstParser "foo" "hello"  
runParser firstParser () "foo" "cat"
```

Detour: the Either datatype

Do you remember `Maybe`? Of course you do! It's our favourite example type so far. `Maybe` has a data constructor, i.e. `Just` and a default 'empty' value, i.e. `Nothing` which represents an error value. Sometimes, we want the error value to contain richer information, perhaps specifying some detail about the problem. This is where the `Either` datatype is useful. It has *two* data constructors: `Right` (for correct output) and `Left` (for error output). The definition of `Either` is:

```
data Either a b = Left a | Right b
```

We use `Either` in parsing, where `Left` values include information about the parse error and its context.

As with `Maybe`, you can work with `Either` values in a monadic style.

More Complex Parsing

We can recognize an entire String, as a sequence of characters.

```
dogParser = string "dog" :: Parsec String st String
```

This parser recognizes a string of characters (and returns the entire String value, unlike a simple sequencing of `char` parser calls).

Now we can start combining simpler sub-parsers into more complex parsers using the *alternative* operator `<|>`. (Really, this is a monadic function allowing us to choose between alternative evaluations.)

```
catParser = string "cat"
petParser = dogParser <|> catParser
parseTest petParser "dog"
parseTest petParser "cat"
parseTest petParser "cog"
```

Parsers as Monads

Since Parser values are monads, we can sequence their operation using monadic do notation in Haskell. For example:

```
animalParser :: Parsec String st String
animalParser = do
    animal <- dogParser
    char '/'
    country <- count 2 anyChar
    let bark =
        case country of
            "UK" -> "woof"
            "FR" -> "waouh"
            "GR" -> "gav"
    return bark
```

Exploring the Parsec Library

Haskell libraries are well-documented online. For instance, the Parsec library docs are at <https://hackage.haskell.org/package/parsec>. There are lots of online code examples, starting here: [https://en.wikipedia.org/wiki/Parsec_\(parser\)](https://en.wikipedia.org/wiki/Parsec_(parser)) More detailed coverage is available here: <https://jsdw.me/posts/haskell-parsec-basics/>

Building a Parse Tree for Arithmetic Expressions

The main use for parsers is to process a sequence of tokens (lexemes) and build a parse tree data structure, representing abstract syntactic structure and relationships.

Let's consider a toy example. Suppose we want to process simple integer arithmetic expressions, along with bracketed sub-expressions. Sentences in our language would look like:

`(1+2) * 3`

We could define simple Haskell algebraic datatypes to encapsulate such expressions:

```

data BinOp = Add | Sub | Mul | Div deriving (Show,Eq)

data ArithExpr =
    Compound ArithExpr BinOp ArithExpr
  | Value Int
  deriving Show

```

and now we can construct a series of miniature parsers to process different input character sequences and construct the appropriate Haskell in-memory data structures to represent the abstract syntax.

```

numberParser :: Parsec String st ArithExpr
numberParser = do
  digs <- many1 digit
  let num = read digs
  return $ Value num

```

Note that `digit` is a built-in primitive matching any single digit character; `many1` is a parser combinator matches one or more of the subparser's accepted input, similar to the `+` in Unix regular expressions. The standard Prelude `read` function converts the `String` value to an `Integer`. We return an `ArithExpr` which is constructed as a `Value` value, storing the number we have parsed from the input.

```

operatorParser :: Parsec String st BinOp
operatorParser = do
  op <- (oneOf "+-*/")
  return (selectOp op)
  where selectOp '+' = Add
        selectOp '-' = Sub
        selectOp '*' = Mul
        selectOp '/' = Div

```

Here we want to parse a single operator character and construct the appropriate `BinOp` value. The `oneOf` parser will match one character from a list of characters. This is similar to the Unix square brackets `[]` regex syntax.

```

expressionParser :: Parsec String st ArithExpr
expressionParser = (between (char '(') (char ')')) binaryExpressionParser <|>
  numberParser

binaryExpressionParser :: Parsec String st ArithExpr
binaryExpressionParser = do
  e1 <- expressionParser
  op <- operatorParser
  e2 <- expressionParser
  return (Compound e1 op e2)

```

Here we are building up compound expressions from their constituent elements, using the simpler parsers defined above. Note the `between` parser combinator for parsing expressions enclosed in brackets, and the `<|>` alternative combinator.

We can run this parser with code like this:

```
parseTest expressionParser "(1+1)"  
-- or  
parse expressionParser "error" "(1+(2*3))"
```

Once we have a parse tree style data structure, we can process this to evaluate the expression or perhaps to rewrite it in some way.

The Haskell code for the above parsing example is available on github:
<https://gist.github.com/jeremysinger/ca2a5e9a2671c856f3fcf5ebe57dd3d>

Alternative typeclass

Sometimes we want to try a computation, then if it fails, try something else. The Alternative typeclass helps us here ... in particular, the `<|>` operator, which intuitively means: 'try the left hand action or, if it fails, try the right hand action'.

```
class Applicative f => Alternative f where  
    empty :: f a  
    (<|>) :: f a -> f a -> f a
```

(Minor reassurance: don't worry about `Applicative` for now ... pretend it says `Monad` instead ... we will fill in the details later in the course.)

Further Reading

- Library API documentation for Parsec: <https://hackage.haskell.org/package/parsec>
- A tutorial guide to Parsec, with great details on the broad range of Parsec library functions for parsing https://jakewheat.github.io/intro_to_parsing/
- A walk-through coding example of Parsec use, building an XML parser <https://www.futurelearn.com/info/courses/functional-programming-haskell/0/steps/27222>
- Discussion of Alternative typeclass <https://typeclasses.com/alternative>

Functional Programming (H) – Lecture 12 – Thu 7 Nov

Monads and Monad Laws

Jeremy.Singer@glasgow.ac.uk

Today we will continue our deep dive into monads, looking at some helpful Haskell syntax and introducing the monad laws. Our *learning objectives* are:

- use record syntax for defining custom types
- learn how monads are constrained in their specification
- understand and memorize the three monad laws
- appreciate by examples and analogy how the monad laws shape the definition of monads

Defining Custom Data Types

We can model a *student* with a name, address, and level in a few ways in Haskell.

- As a *tuple*
 - type Student = (String, String, Int)
- As a *data type*
 - data Student = Student String String Int

However neither of the above approaches is ideal. It is difficult to know which *String* is a name and which is an address. We need to *pattern match* to deconstruct and get a relevant field. All fields must be specified when updating the data structure instance.

In Haskell, *records* allow us to have *named fields*.

```
data Student = MkStudent { name :: String, address :: String,
level :: Int }
```

We can construct a record as normal, or by explicitly specifying fields:

```
s = MkStudent "jeremy" "glasgow" 1
s' = MkStudent { name = "simon", address = "glasgow", level = 6 }
```

We can project a record by using its field name, effectively the field name becomes a function:

```
-- implicitly ... name :: Student -> String
putStrLn(name s)
```

We can also *update* a record without needing to specify all fields.

```
let s2 = s { level = 2 }
```

A newtype declaration is a special form of a data declaration where there is only one data constructor. It can be used like a *type alias*, while ensuring that types are treated separately in the type system.

```
newtype Variable = MkVariable String
```

We will encounter plenty of newtypes as we look at the various monad instances.

What is a Monad?

As we have already seen, `Monad` is a type class in Haskell, which provides a systematic way of sequencing operations. Monads enable values to be put into contexts, enforced by the type system. We have considered contexts such as lists, `Maybes`, `IO`, and `Reader/Writer`. Effectively, `Monad` is a design pattern — a recurring solution to a common problem in functional programming.

Laws

What is a law? It's a principle that governs or constrains things. In theoretical computing terms, laws constrain behaviours and relationships between entities.

In a previous lecture, Simon introduced laws for the `map` function:

- identity law
 - `map id == id`
- composition
 - `(map f).(map g) == map (f.g)`

And we also looked at three laws for the `Monoid` typeclass:

- left identity law
 - `mempty <> x == x`
- right identity law
 - `x <> mempty == x`
- associativity law
 - `(x <> y) <> z == x <> (y <> z)`

Remember that we stressed the point that Haskell cannot enforce these laws in the language or type system directly. Instead, developers have to manually ensure the laws are respected by their code ... or we might use a theorem prover.

Monad Laws

There are three monad laws, which constrain the behaviour of monads.

First Monad Law (Left Identity)

- `(return x) >>= f == (f x)`

If we think about the types of `return :: Monad m => a -> m a` and `(>>=) :: Monad m => m a -> (a -> m b) -> m b` and then fit in the types of `x :: a` and `f :: Monad m => a -> m b` then everything makes sense.

In natural language terms, we put `x` into the monad, then bind strips it out of the monad, feeds it to function `f` and the result `f x` is evaluated.

Second Monad Law (Right Identity)

- $(y \gg= \text{return}) == y$

Here, `y` is a monadic context containing a value `x`, and the bind operation feeds value `x` to `return`, which *re-wraps* `x` into the monadic context, to recover `y`.

Third Monad Law (Associativity)

- $(y \gg= f) \gg= g == y \gg= (\lambda x \rightarrow (f x \gg= g))$

On the left hand side, the first bind feeds `y` to `f` to get a monadic result, then the second bind feeds this interim result to `g` to get the final result.

On the right hand side, we do the innermost bind in the brackets first ... so we bind the result of `f x` to `g`, where `x` is the parameter from the intermediate lambda abstraction we have to construct to get the types correct.

In natural language, the third monad law says that when we have a chain of monadic function applications with `>>=`, it doesn't matter how they're nested.

Monad Laws for Maybe

Let's consider the monad laws and the `Maybe` monad. Suppose we defined `return` for `Maybe` as follows:

```
return :: a -> Maybe a
return _ = Nothing
```

then would the two identity laws hold? No, of course they wouldn't!

Instead, we define it as:

```
return :: a -> Maybe a
return x = Just x
```

Now both identity laws do hold.

Further Reading

- Think about the List monad. Can you show that the default `return` and `(>>=)` operators respect the monad laws. Can you think of an alternative bind operator for lists? Does this respect the third monad law?
- A poem about monad analogies <https://8thlight.com/insights/the-community-guide-to-monads>

- A philosophical guide to monads
<https://tomas.net/academic/papers/monads/monads-programming.pdf>

Functional Programming (H) The Haskell Ecosystem & Introduction to Assessed Exercise

Simon Fowler & Jeremy Singer

Semester 1, 2024/25



Join at [sli.do 379 9172](https://sli.do/3799172)



University
of Glasgow

VIA VERITAS VITA

Today

- Last time (with Jeremy): The Monad Laws
- Today:
 - Haskell in the Large: The Haskell Ecosystem
 - Introduction to the Assessed Exercise (Solitaire)
- Please ask questions!

This Week

- This week is designed to be a coursework / catchup week
- There will be:
 - No lecture on Thursday 14th November
 - No lab sheet
 - No quiz
- There will be a lab on Friday; you can use this to get started on the assessed exercise. I will be there (and will try to get to as many of the labs as I can between now and the handin date).

The Haskell Ecosystem

Haskell in the Large

- So far, you have mainly written shorter Haskell programs
 - Algorithm implementations, recursive functions, etc.
- But Haskell is a **general-purpose** programming language: it can do much more!
- In particular, used heavily in industry (particularly in financial sector)

GHC: The Glasgow Haskell Compiler

- Haskell is the language, GHC is the most popular compiler
- GHC supports the Haskell 2010 language definition, along with (many, many) extensions
- Under active development
- Extensive support for parallelism (Phil Trinder worked heavily on this)

The screenshot shows the official website for The Glasgow Haskell Compiler. The header features the "GHC" logo and the tagline "The Glasgow Haskell Compiler". A sidebar on the left contains links for "About GHC" (Home, License, Documentation, Blog, Download, Report a security issue, Report a bug, Developers Wiki), "About Haskell" (Haskell.org, Haskell Language Wiki, Haskell 2010 Report, Haskell Mailing Lists), and "Links" (Hackage). The main content area is titled "Latest News" and lists three recent releases: "7 November 2022" (GHC 9.2.5 Released! [download]), "3 November 2022" (GHC 9.4.3 Released! [download]), and "22 August 2022" (GHC 9.4.2 Released! [download]). Below the news is a section titled "What is GHC?" which provides an overview of the compiler's features and capabilities, including support for Haskell 2010, concurrency, parallelism, and various optimization and performance features. It also mentions the availability of platforms like Windows, Mac, Linux, and Unix, as well as LLVM support and the interactive environment.

GHC The Glasgow Haskell Compiler

About GHC

- Home
- License
- Documentation
- Blog
- Download
- Report a security issue
- Report a bug
- Developers Wiki

About Haskell

- Haskell.org
- Haskell Language Wiki
- Haskell 2010 Report
- Haskell Mailing Lists

Links

- Hackage

Latest News

7 November 2022
GHC 9.2.5 Released! [[download](#)]

3 November 2022
GHC 9.4.3 Released! [[download](#)]

22 August 2022
GHC 9.4.2 Released! [[download](#)]

What is GHC?

GHC is a state-of-the-art, open source, compiler and interactive environment for the functional language [Haskell](#). Highlights:

- GHC supports the entire [Haskell 2010 language](#) plus a wide variety of [extensions](#).
- GHC has particularly good support for [concurrency](#) and [parallelism](#), including support for [Software Transactional Memory \(STM\)](#).
- GHC generates fast code, particularly for concurrent programs. Take a look at GHC's performance on [The Computer Language Benchmarks Game](#).
- GHC works on several [platforms](#) including Windows, Mac, Linux, most varieties of Unix, and several different processor architectures. There are detailed [instructions](#) for porting GHC to a new platform.
- GHC has extensive [optimisation](#) capabilities, including inter-module optimisation.
- GHC compiles Haskell code either directly to native code or using [LLVM](#) as a back-end. GHC can also generate C code as an intermediate target for porting to new platforms. The [interactive environment](#) compiles Haskell to bytecode, and supports execution of mixed bytecode/compiled programs.
- [Profiling](#) is supported, both by time/allocation and various kinds of heap profiling.
- GHC comes with several [libraries](#), and thousands more are available on [Hackage](#).

GHC is heavily dependent on its users and [contributors](#). Please come and join the [mailing lists](#) and send us your comments, suggestions, bug reports and contributions!

Hackage

- Haskell has many, many libraries
- These are contained on Hackage (<https://hackage.haskell.org>)
- Also hosts detailed documentation
- (Just because a package is on Hackage doesn't mean it is any good ☺)

Cabal

- Cabal is Haskell's package manager
 - Used for installing, updating, packaging, packages
 - Handles dependencies, etc.
 - Also used for packaging projects for distribution
- Issue: Packages are installed system-wide
 - Conflicts across versions, "cabal hell"
- Often, used under-the-hood by...

Stack

- Stack: Haskell toolchain and build system
 - Key goal: **reproducible builds**
 - Isolated GHC instances
 - Isolated per-project sandboxes (so versions can be different across project)
 - **Consistent sets of Haskell packages, called snapshots:** known to work together, hosted on **Stackage**
- Simple commands (`stack build`, `stack install x`, `stack run`, ...)
- Used for the coursework

Hoogλe

- Search engine for Haskell functions
 - By type
 - By name
- Can also do approximate lookups of type signatures
- <https://hoogλe.haskell.org>

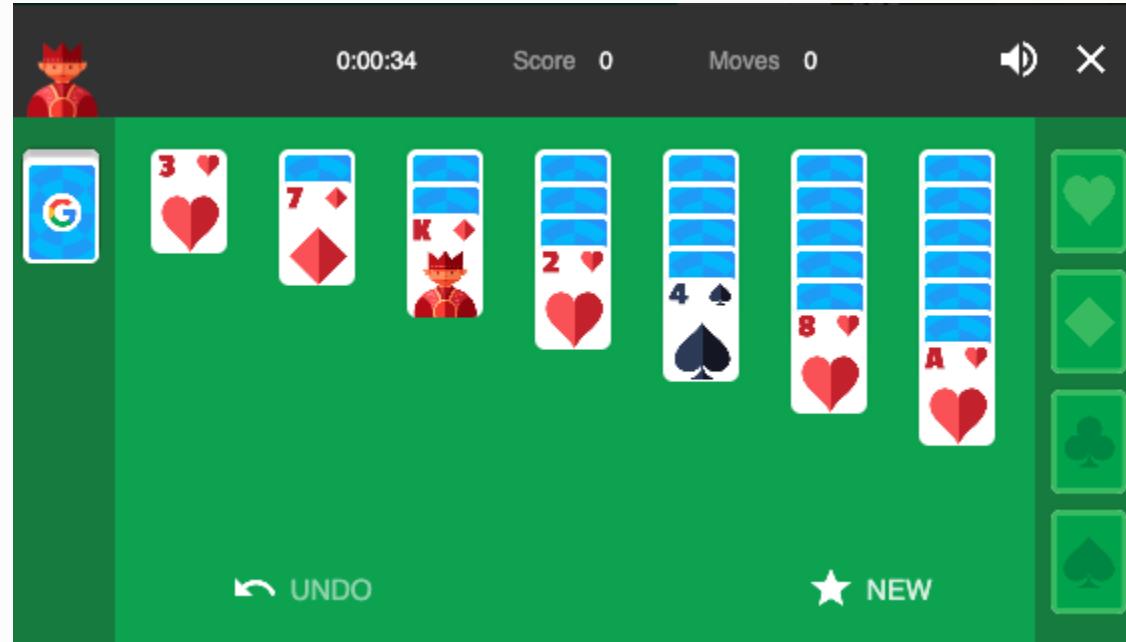
The screenshot shows the Hoogλe search interface. At the top, there's a search bar containing the type signature `[String] -> String`, a dropdown menu set to `stackage`, and a `Search` button. The page title is **Hoogλe**. On the left, there's a sidebar titled **Packages** with a list of Haskell packages, each with a minus sign and a plus sign icon. The packages listed are: `is:exact`, `base`, `ghc`, `hedgehog`, `base-compat`, `base-prelude`, `rio`, `numeric-prelude`, `ihaskell`, `clash-prelude`, `dimensional`, `ghc-lib-parser`, `llvm-hs-pure`, `rebase`, `mixed-types-num`, `xmonad-contrib`, `Cabal-syntax`, `numhask`, `stack`, and `LambdaHack`.

The main content area displays search results for three type signatures:

- `:: [String] -> String`**: This section includes the `unlines` function from the `base` package. The documentation states: "Appends a \n character to each input string, then concatenates the results. Equivalent to foldMap (s -> s ++ "\n")."
- `unwords :: [String] -> String`**: This section includes the `unwords` function from the `base` package. The documentation states: "An inverse operation to words. It joins words with separating spaces."
- `escapeArgs :: [String] -> String`**: This section includes the `escapeArgs` function from the `base` package. The documentation states: "Given a list of strings, concatenate them into a single string with escaping of certain characters, and the addition of a newline between each string. The escaping is done by adding a single backslash character before any whitespace, single quote, double quote, or backslash character, so this
- `renderStack :: [String] -> String`**: This section includes the `renderStack` function from the `base` package.

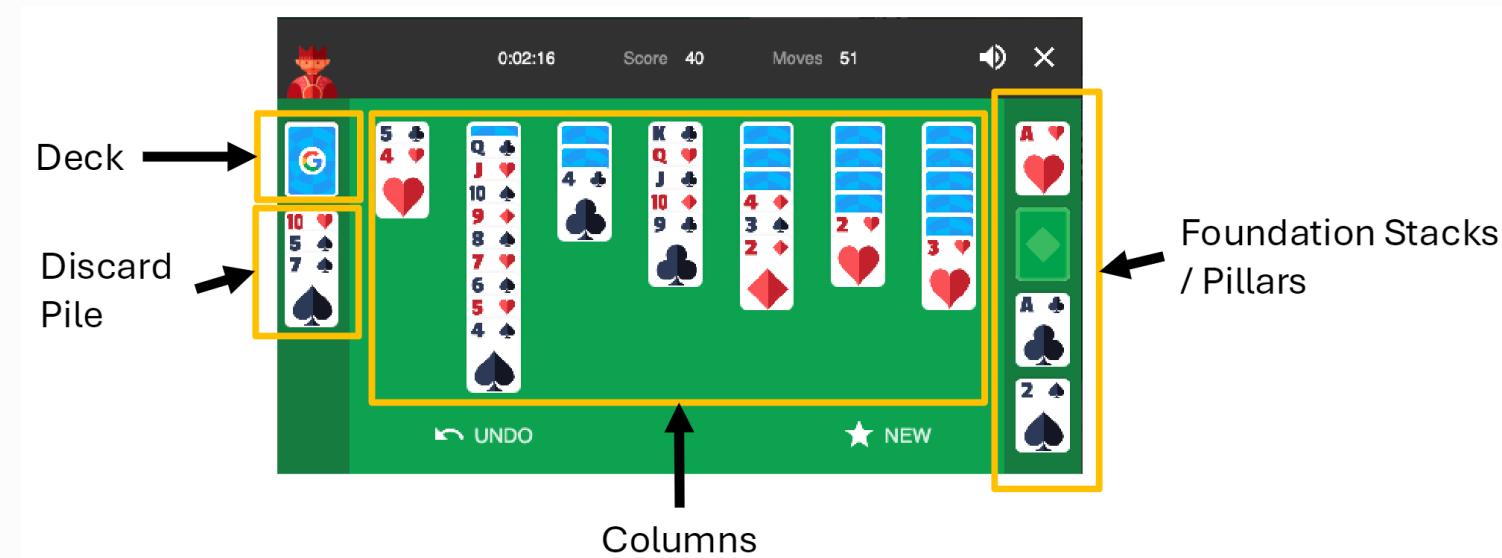
Introduction to the Assessed Exercise

Solitaire



- **Solitaire:** A single-player card game
- Cards initially set up in 7 columns: 1 face-up card in first, 1 face-down and 1 face-up card in the second, and so on

Solitaire



- The **deck** is a face-down stack of cards. Drawing a card from the deck adds it to the **discard pile**
- All visible cards must be arranged with **alternating colours** in **descending order**
- There is a **pillar** or **foundation stack** for each suit; these must be built up in **ascending order** (so Ace, then Two, all the way up to King).
- The goal of the game is to move all cards to the pillars

Your Tasks

Deck size: 7
Discard: ♣5, ♣6
Pillars:
Spades: ♣2
Clubs: <empty>
Hearts: ♥2
Diamonds: ♦2

[0]	[1]	[2]	[3]	[4]	[5]	[6]
♦K	♥K	???	???	???	???	???
♣Q	♠Q	???	???	???	♦9	???
♥J	♦J	♠K	♦6	???		???
♠10		♥Q	♣5	♥6		♦5
♥9		♣J	♥4			
♣8		♦10				
♥7		♣9				
♣6						
♥5						
♣4						
♦3						
♣2						

- Your goal is to use Haskell to write a text-based version of Solitaire!
- Deck Creation and Shuffling
 - We've given you some basic data structures, but you'll begin by implementing deck creation and a shuffling algorithm
- Game Logic
 - Board setup & win detection
 - Show instance
 - Helper functions
 - Implementing each command

Board Representation

Deck size: 7
 Discard: ♦5, ♦6

Pillars:
 Spades: ♦2
 Clubs: <empty>
 Hearts: ♥2
 Diamonds: ♦2

[0]	[1]	[2]	[3]	[4]	[5]	[6]
♦K	♥K	???	???	???	???	???
♦0	♦Q	???	???	???	♦9	???
♥J	♦J	♦K	♦6	???	???	
♦10		♥Q	♦5	♥6	♦5	
♥9		♦J	♥4			
♦8		♦10				
♥7		♦9				
♦6						
♥5						
♦4						
♦3						
♦2						

(a) Example board

```
MkBoard {
    boardDeck = [ ♦4, ♦8, ♦7, ♥8, ♥3, ♦8, ♦J],
    boardDiscard = [ ♦6, ♦5],
    boardPillars = MkPillars {
        spades = Just 2,
        clubs = Nothing,
        hearts = Just 2,
        diamonds = Just 2
    },
    boardColumns = [
        [(♦2,True), (♦3,True), (♦4,True),
         (♥5,True), (♦6,True), (♥7,True),
         (♦8,True), (♥9,True), (♦10,True),
         (♥J,True), (♦Q,True), (♦K,True)],
        [(♦J,True), (♦Q,True), (♥K,True)],
        [(♦9,True), (♦10,True), (♦J,True),
         (♥Q,True), (♦K,True), (♦3,False),
         (♦A,False)],
        [(♦4,True), (♦5,True), (♦6,True),
         (♥10,False), (♦Q,False)],
        [(♥6,True), (♦7,False), (♦10,False),
         (♦K,False)],
        [(♦9,True), (♦3,False)],
        [(♦5,True), (♦7,False), (♦4,False),
         (♦9,False)]
    ]
}
```

(b) Board representation

- Board representation is given for you
- 4-field record:
 - Deck: list containing shuffled deck
 - Discard pile: list containing discarded cards
 - Pillars: Nested record, Maybe Value for each suit
 - Columns: Nested list of length 7, containing (Card, Bool) pairs where the Bool indicates card visibility
- All columns are stored in most-recent-first order – please pay attention to this! (It makes things easier)

Commands

Long command	Short command	Command data constructor	Description
draw	d	Draw	Draws a card from the deck
move n from s_1 to s_2	mv $n\ s_1\ s_2$	Move Count StackIndex StackIndex	Moves n cards from stack s_1 to stack s_2
movest $s_1\ s_2$	—	MoveStack FromStack ToStack	Moves the visible contents of stack s_1 to stack s_2
movefd s	—	MoveFromDiscard StackIndex	Moves the card from the top of the discard stack to stack s
movetp discard	movetp d	MoveToPillar FromDiscard	Moves a card from the discard pile to the pillars
movetp s	—	MoveToPillar (FromStack StackIndex)	Moves a card from stack s to the pillars
movefp $st\ s$	—	MoveFromPillar Suit StackIndex	Moves the card at the top of the pillar for suit st to the top of stack s (e.g., movefp clubs 2)
solve	—	Solve	Attempts to repeatedly move the top of each stack to the pillars, until no more cards can be moved.

Skeleton Code

- We have provided some skeleton code for you
 - Feel free to add in helper functions or use library functions that do not require additional dependencies
 - Please do not edit existing data types or function type signatures
- Main files:
 - app/Main.hs is the entry point and contains the user input loop.
 - src/CommandParser.hs contains the parser for user commands
 - src/Error.hs contains code for reporting and displaying errors
 - src/Deck.hs contains the data structures for representing cards and the deck
 - src/Game.hs contains the game logic
- You will only need to edit Deck.hs and Game.hs

Working on the Lab Machines

- You are welcome to work on either your own machine or the lab machine
- There have been significant technical issues with Stack on the lab machines this year, unfortunately
- For this reason we have provided a Virtual Machine image, downloadable from Moodle. You can use this with VirtualBox on the lab machines; Stack and GHC are pre-installed, and I have successfully tested the coursework on it

Practicalities

- **Due:** 6th December 2024, 16:30
- **Submission:** Submit Deck.hs and Game.hs via Moodle; see the handout
- **We expect your code to compile:** mark cap of 15/66 applies otherwise
- **Best-effort technical support:** We / the tutors will try to help with setup issues, but please try to ask these early on. If requests are nontrivial / come in too late, we will ask you to use the VM on the lab machines
- **Support:**
 - Please ask questions in the Coursework channel on Teams (I will also post updates there). You can also DM/email, but Teams is preferable so that your peers can also learn.
 - Feel free to ask questions during the labs; tutors will be able to help (within reason) with both the coursework and the labs
 - If you are spending a long time on something, ask for help

Wrapping Up

- **Today**
 - A whistle-stop tour of the Haskell ecosystem
 - An introduction to the assessed exercise
- **Next week**
 - Jeremy will talk about error handling & monad transformers
- **Over to you**
 - Get started on the coursework! Remember: help is always available
- **Questions?**

Functional Programming (H) – Lecture 14 – Mon 18 Nov

Error Handling

Jeremy.Singer@glasgow.ac.uk

Today we will examine how to handle runtime errors in Haskell programs. Our *learning objectives* are:

- revisit various data types that indicate success or failure of Haskell computation
- identify advantages and disadvantages of various Haskell patterns for handling runtime errors
- explore practical case studies of real-world scenarios that deal with runtime errors

What is an Error?

A *error* is a fault in the specification, implementation or operation of software, which causes an incorrect or unexpected result, leading to unanticipated behaviour.

Many of the errors we have encountered so far in our Haskell coding have been *static* errors, such as type errors. These are highlighted by the GHC compiler or interpreter when the code is parsed.

However sometimes errors occur when the program executes; we refer to these errors as *runtime exceptions*. Examples might include:

- arithmetic exceptions, such as integer division by zero
- taking head or tail of an empty list
- I/O exceptions when dealing with files or network access
- incomplete pattern matching in a function evaluation

Immediate Termination

The simplest way to deal with a runtime error is to terminate the program immediately. This is achieved in Haskell by calling the function:

```
error :: String -> a
```

which has a polymorphic return type that matches any expression context. The `error` function takes a `String` parameter that specifies the human-readable error message which gets printed to the standard error stream.

When the program is interpreted in GHCI, the error is reported and a stack trace is provided.

This behaviour is equivalent to the `exit()` function in C or Python.

Indicate Error Conditions with Wrapper Types

Software engineering principles like *defensive programming* and *graceful failure* can be followed in Haskell programming. This involves actively anticipating problems and handling errors within the program logic.

We have already encountered how to indicate error values using wrapper types like `Maybe` or `Either`. The `Maybe` type uses `Nothing` to indicate the absence of a value, i.e. a safe null value, and `Just x` for a normal value `x`.

One drawback of using the `Maybe` data type to deal with errors is that there is a lack of error information in `Nothing` – we have no specific details about what caused the error.

`Either` is a richer type which supports error information:

```
data Either a b = Left a | Right b
```

Here, `Left a` indicates an error value, with the `a` value (often `String`) holding error information. `Right b` indicates a proper value, equivalent to `Just b` in the `Maybe` type.

In the same way that `Maybe` is monadic, so it can take part in a bind call sequence or a `do` block – so is `Either`.

When we looked at parsing, we noted that parser error values are wrapped as `Left` values, when we invoke the parser with the `runParser` function.

```
runParser :: Parsec String st a -- our parser
           -> String      -- state (ignore)
           -> String      -- filename
           -> String      -- string to parse
           -> Either ParseError a -- result

catParser = string "cat"
runParser animalParser "" "" "cat" -- result: Right "cat"
runParser animalParser "" "" "dog" -- result: Left ParseError
```

Runtime Exception Handling

Like in Java, there are different types of exceptions that occur for different sorts of problems. There are specific circumstances in which each type of exception can be raised.

There are plenty of exception handling facilities (actually functions) in the Haskell `Control.Exception` module. We look at two design patterns in particular:

1. the `catch` function, and
2. the `try` function.

Catching Exceptions

The `catch` function specifies what to do if an expression evaluation raises an error. We provide a handler with a type-compatible alternative expression to evaluate. Below are three examples:

```
catch :: Exception e => IO a -> (e -> IO a) -> IO a

-- example 1
(randomIO :: IO Bool) >>= \x -> (if x then putStrLn "hello" else error
"oops!") `catch` (\(e :: ErrorCall) -> putStrLn "exception")

-- example 2
((readFile "foo.txt") >>= putStrLn) `catch` (\e -> (putStrLn $ "unable to
open file: " ++ (show (e :: IOException)))))

-- example 3
getContentFrom :: URL -> IO (Maybe String)
getContentFrom url =
    catch (do { str <- scrapeURL url scrapeBody; return $ str })
        (\err -> do {putStrLn $ show (err :: IOException); return Nothing})

-- full text for example 3 at:
-- https://gist.github.com/jeremysinger/cb387f12619b5320c7b8eefe7c96a4a9
```

Note it is necessary to annotate the exception names with their types explicitly, to make type inference work properly.

Try Clauses for Exceptions

The `try` function embeds the result in an `Either e b` -- where `Left e` is an `Exception` and `Right b` is a normal result. Subsequently, we can pattern-match on this `Either` value. Effectively this allows us to wrap runtime errors neatly as an `Either` result type.

```
try :: Exception e => IO a -> IO (Either e a)

do
    x <- try ((readFile "foo.txt") >>= putStrLn)
    case x of
        Left e -> putStrLn $ "you've got problems: " ++ (show (e :: IOException))
        Right r -> return r
```

Note that unlike in Java, Python and other mainstream languages, exception handling is based purely on function calls rather than requiring built-in language keywords.

Further Reading

To find out more about error handling, consult some of the documentation below. Note this level of detail is non-examinable.

- https://wiki.haskell.org/Handling_errors_in_Haskell has some pragmatic advice about errors, mostly similar to the material in this lecture
- <https://book.realworldhaskell.org/read/error-handling.html> provides lots of detail (more than you need) but might be interesting to Haskell enthusiasts
- <http://learnyouahaskell.com/input-and-output> gives helpful info about I/O exceptions and how to handle them

Functional Programming (H) – Lecture 15 – Thu 21 Nov

Monad Transformers

Jeremy.Singer@glasgow.ac.uk

Today we will examine the advanced concept of *monad transformers* in Haskell.

Our *learning objectives* are:

- appreciate the motivation for monad transformers and their use to combine individual monads into compound stacks of monads in Haskell programs
- gain familiarity with the standard Haskell patterns for a monad transformer stack

What is a Monad?

Recall from earlier lectures that a *monad* is a typeclass in Haskell, representing a computational structure or context. A monad combines computations into an ordered sequence with the *bind* operator ($>>=$). Monads enable containment of side-effects in the type system, i.e. the IO monad.

What is a Monad Transformer?

This is a technique for *composing* two monads into a single combined monad that has the joint behaviour of both individual monads. Monad transformers are a *monad composition technique*.

There is a Haskell typeclass `MonadTrans` with a characteristic function `lift`:

```
class MonadTrans t where
    lift :: (Monad m) => m a -> t m a
```

The `lift` function promotes a base monad function into the combined monad.

Motivating Example

Do you remember we had a pattern matching triangle, before we knew how to use the monadic bind operator ($>>=$)? We are going to see something similar here, when we try to use `Maybe` values in the context of the `IO` monad.

The library function we will use for this example is

```
findExecutable :: String -> IO (Maybe FilePath)
```

with information at <https://hackage.haskell.org/package/directory-1.3.8.1/docs/System-Directory.html#v:findExecutable>

The `findExecutable` function searches through the system PATH to find an executable file matching the string parameter value.

```
import System.Directory

findAllExes = do
    a <- findExecutable "git"
    case a of
        Nothing -> return Nothing
        Just pathA -> do
            b <- findExecutable "ghc"
            case b of
                Nothing -> return Nothing
                Just pathB -> do
                    c <- findExecutable "clang"
                    case c of
                        Nothing -> return Nothing
                        Just pathC -> return $ Just (pathA, pathB, pathC)
```

The key problem is that we are trying to deal with `Maybe` values while we are with the `IO` monad. The return type of `findExecutable` is `IO (Maybe FilePath)`, and strictly speaking this isn't a monad in itself - it's a monad-within-a-monad.

Example with Transformers

Let's think about combining the `IO` monad and the `Maybe` monad. We will let `IO` be our *base* monad — this is generally the case in monad transformer stacks. So we use the `MaybeT` monad transformer. In this way, we provide the `IO` monad with the characteristics of the `Maybe` monad.

Here is the example program, which looks for three executable files and returns a triple of their paths:

```
import Control.Monad.Trans.Maybe
import System.Directory

findAllExes :: MaybeT IO (FilePath, FilePath, FilePath)
findAllExes = do
    a <- MaybeT $ findExecutable "git"
    b <- MaybeT $ findExecutable "ghc"
    c <- MaybeT $ findExecutable "clang"
    return (a, b, c)

runMaybeT findAllExes
```

`findExecutable` must be in the `IO` monad, because it is touching file storage, looking for exe files. The `Maybe` is necessary in case a particular exe file is not found in the PATH. If any exe is not found, we return `Nothing`, otherwise we return a `Just` triple value, which is simultaneously in the `Maybe` monad *and* in the `IO` monad.

The `MaybeT` constructor puts an `IO` `Maybe` value into the monad transformer, and the `runMaybeT` function extracts the `IO` value from the monad transformer.

Similarly, the `lift` function will run an `IO` action in the `MaybeT` context. See the example below for more details, where we use `getLine` and `putStrLn` inside the `MaybeT IO` monad.

Note the use of the `guard` monadic function here. If the condition fails, `Nothing` is returned. See <https://learnyouahaskell.com/a-fistful-of-monads> for more details about this. The `guard` call succeeds or fails, in the monadic context, based on the value of the input boolean.

```
import Control.Monad
import Control.Monad.Trans
import Control.Monad.Trans.Maybe

isValid :: String -> Bool
isValid v = '!' `elem` v

maybeExcite :: MaybeT IO String
maybeExcite = do
    v <- lift getLine
    guard $ isValid v
    return v

doExcite :: MaybeT IO ()
doExcite = do
    lift $ putStrLn "say something exciting!"
    exciting <- maybeExcite
    lift $ putStrLn ("this is very exciting! " ++ exciting)
```

Revisiting a Lie

Do you remember when we looked at the `Reader`, `Writer` and `State` monads in an earlier lecture? I told you I was lying about the types ... well, now we understand the reason for this. Actually, `Reader` is a `ReaderT` monad transformer, with the `Identity` monad at the base of the transformer stack. Similarly, `Writer` is a `WriterT` with an `Identity`. See for example <https://hackage.haskell.org/package/transformers-0.6.1.2/docs/Control-Monad-Trans-Writer-Lazy.html#g:2> for documentation about this.

Implementation of a Transformer

This is beyond the scope of our coverage for today, but you might like to look at https://en.wikibooks.org/wiki/Haskell/Monad_transformers to see how the `MaybeT` monad transformer is implemented.

Further Reading

To find out more about monad transformers, consult some of the documentation below. Note this level of detail is *non-examinable*.

- <https://www.fpcomplete.com/haskell/library transformers/> has example code for using monad transformers
- <https://mmhaskell.com/monads/transformers> presents some more nice example code for monad transformers
- <https://www.youtube.com/watch?v=UPb83JiGiY> is a friendly video about monad transformers

Functional Programming (H) – Lecture 16 – Mon 25 Nov

Functors & Applicatives

Jeremy.Singer@glasgow.ac.uk

After our excursion into advanced concepts last week, we are returning to very familiar ideas this week, but examining them in a new light. Today we will consider the related concepts of *functors* and *applicatives* in Haskell.

Our *learning objectives* are:

- construct functor instances for user-defined datatypes like binary trees
- recognize appropriate contexts for using the functor and/or applicative typeclasses
- understand the relationships between functors, applicatives and monads in Haskell

Back to map

Do you remember the `map` function? It was the first *higher-order* function we encountered. The `map` function takes a function `f` and maps it over a list:

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

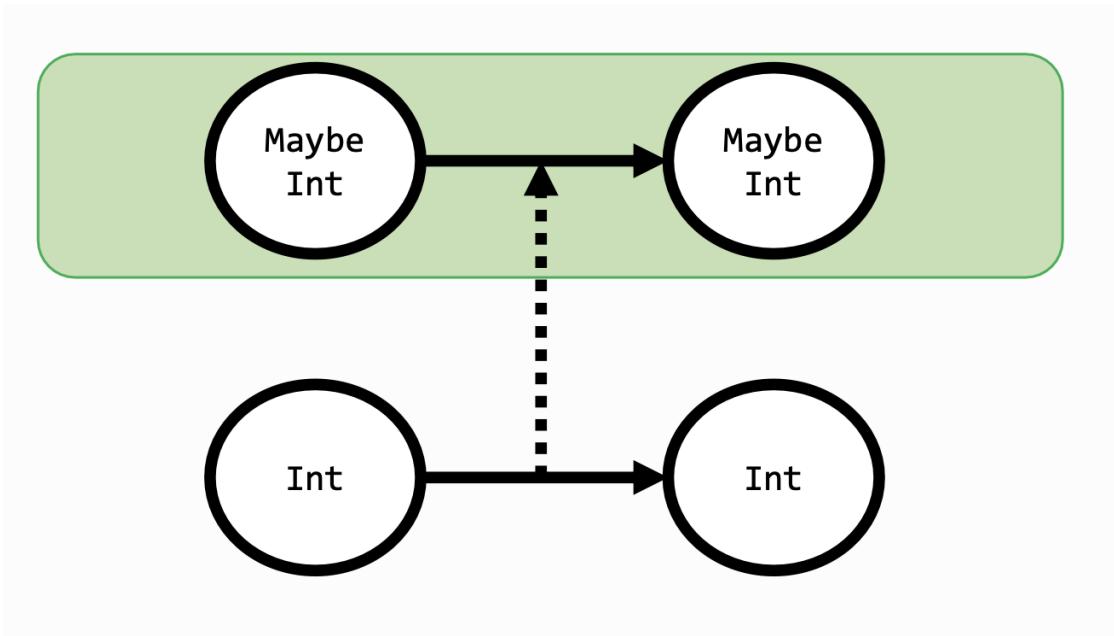
The `map` function preserves the *structure* of the list, but transforms each individual element in the list according to the `f` function.

So, we realise that the length of the mapped list will be the same as the length of the original input list, and each transformed element in the mapped list occupies the corresponding position as its untransformed element in the original list.

Mapping Maybe values

The `map` function allows us to operate on elements in a list context; it would be nice to do the same to values in other contexts. For instance, if we have an increment function `\x->x+1`, how could we apply this function to a value `Just 41`?

Ideally, we would like to *lift* our increment function into the `Maybe` context. See the diagram below:



Lifting a function into the Maybe context

In fact, there is a function to do this, it's called `fmap`.

```
fmap (\x->x+1) (Just 41)
-- > evaluates to (Just 42)
```

Note the `fmap` can transform the type of the contained element, as well ...

```
fmap (length) (Just "Glasgow University")
-- > evaluates to (Just 18)
```

The reason why we can map over `Maybe` values is because `Maybe` is an instance of the `Functor` typeclass in Haskell.

What is a Functor?

The `Functor` typeclass is used for types that can be mapped over. A functor is a container or context (e.g., `Maybe`) that allows us to apply a function to its contents.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Conventionally, Haskell syntax also allows us to use an infix operator for `fmap`:

```
(<$>) :: (a -> b) -> f a -> f b
```

Here are some examples of `fmap` in action:

```
fmap (+1) (Just 3)
fmap (*2) [1,2,3]
fmap (fmap Data.Char.toUpper) getLine
```

all of which could be rewritten using the equivalent infix syntax:

```
(+1) <$> (Just 3)
(*2) <$> [1,2,3]
(Data.Char.toUpper <$>) <$> getLine
```

To summarise, an instance of a **Functor** is a `map`-able type.

Functor instances

We can make any container type into an instance of the **Functor** typeclass by providing a `fmap` definition. For example, the Haskell Prelude defines:

```
instance Functor Maybe where
  fmap f (Nothing) = Nothing
  fmap f (Just x) = Just (f x)
```

Consider our binary tree custom datatype used in the labs:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
deriving (Show, Eq)
```

We can provide an `fmap` definition for `Tree` as follows:

```
instance Functor Tree where
  fmap f Leaf = Leaf
  fmap f (Node x t1 t2) =
    Node (f x) (fmap f t1) (fmap f t2)
```

Functor Laws

The **Functor** typeclass is used for types that can be mapped over. The `fmap` function modifies the *elements* in the structure, but preserves the structure itself.

To guarantee this property of functors, instances of the **Functor** typeclass are expected to follow these rules:

1. *identity*: `fmap id == id`
2. *composition*: `fmap (f.g) == (fmap f).(fmap g)`

Like the monad laws we saw in Lecture 12, Haskell is unable to encode or check these laws directly. It is the responsibility of the developer (or a theorem prover) to check the laws hold for a **Functor** instance.

So, for example, here is an example of a *bad* functor that doesn't obey the laws:

```
instance Functor [] where
  fmap f [] = []
  fmap f (x:xs) = (f x) : (fmap f xs)
```

You can see that the structure of the list is altered - each element is duplicated. Hence both the identity law and the composition law will be broken by this definition of `fmap`.

Motivation for Applicative Functors

Suppose we have a function (like `(+1)`) wrapped up inside a context (like a `Maybe`). How do we apply this function to values inside other `Maybe` contexts? The answer is the `Applicative` typeclass, particularly the `<*>` or *apply over* function.

```
(<*>) :: Applicative f => f (a->b) -> f a -> f b
```

Look at the following code fragments:

```
(Just (+1)) <*> (Just 2)  
-- > evaluates to (Just 3)
```

```
[(*3)] <*> [1..4]  
-- > evaluates to [3,6,9,12]
```

Applicative Functors

The `Applicative` typeclass is defined as follows:

```
class Functor f => Applicative f where  
pure :: a -> f a  
(<*>) :: f (a -> b) -> f a -> f b
```

Here `pure` puts a value (often a function) into the applicative context and `(<*>)` applies a wrapped function to a wrapped value. Consider this example:

```
pure (+) <*> (Just 41) <*> (Just 1)  
-- > evaluates to (Just 42)
```

Note that `pure` exactly corresponds with monadic `return` function.

All `Applicative` instances are also `Functor` instances. This is because `fmap` may be defined directly in terms of `pure` and `(<*>)`, i.e.

```
fmap f x = (pure f) <*> x
```

Maybe is an Applicative

The built-in definition of `Applicative` for `Maybe` is as follows:

```
instance Applicative Maybe where  
pure = Just  
(<*>) Nothing _ = Nothing  
(<*>) _ Nothing = Nothing  
(<*>) (Just f) (Just x) = Just (f x)
```

It's possible to 'lift' two-operand functions into `Applicative` structures, e.g. using the utility `liftA2` function. try:

```
import Control.Applicative  
liftA2 (+) (Just 1) (Just 2)  
liftA2 (+) [1,2,3] [4,5,6]
```

Note that `liftA2` can be defined in terms of `pure` and `<*>` —can you see how?

In summary, Applicative functors take the concept of regular functors one step further, since Applicatives enable function application to occur *within* the container context.

Applicative Laws

These are similar to the laws for functors and monads. The definitions of the applicative laws are beyond the scope of this course, and therefore non-examinable. See https://en.wikibooks.org/wiki/Haskell/Applicative_functors for full details.

Relationship between Functors, Applicatives and Monads

Look at the characteristic functions for these three typeclasses:

```
(<$>) :: Functor x => (a -> b) -> x a -> x b  
(<*>) :: Applicative x => x (a -> b) -> x a -> x b  
(>>=) :: Monad x -> x a -> (a -> x b) -> x b
```

Can you observe the similarities between these three functions, in terms of how they apply functions on values in contexts? Admittedly, to make it neater, we should probably consider the *reverse bind* function (`=<<`) which flips the order of the parameters:

```
(=<<) :: Monad x => (a -> x b) -> x a -> x b
```

The `fmap` function lifts a function into a context and applies it to elements in that context. The `<$>` function applies a function in a context to a value in the same context. The *bind* function applies a function that expects a value outside the context but returns a result in the context, to a value *already* in the context.

If you understand the above sentence, you are well on the way to Haskell enlightenment. If you don't understand it, think hard about why `(Just (Just (Just (Just "foo"))))` is not the same as `(Just "foo")` and maybe read some of the articles listed below...

Further Reading

To find out more about functors and applicatives, consult some of the documentation below. Note this level of detail is *non-examinable*.

- https://www.adit.io/posts/2013-04-17-functors_applicatives_and_monads_in_pictures.html This is the best ever graphical introduction to functors and applicatives, which I **highly recommend** reading.
- <https://learnyouahaskell.com/functors-applicative-functors-and-monoids> is not as good as the intro article above, but it comes a close second.

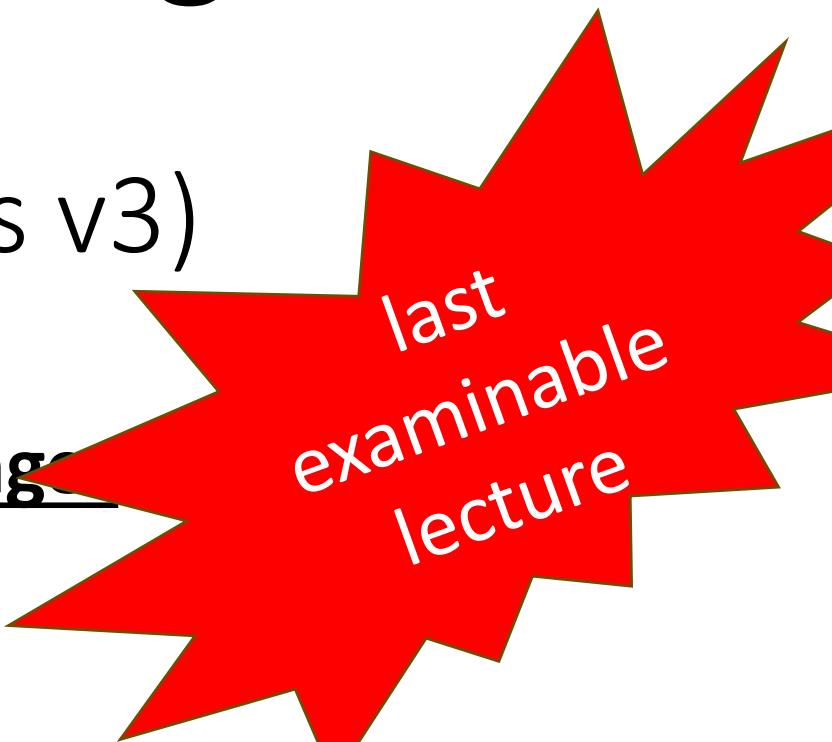
- <https://www.jerf.org/iri/post/2958/> is a very long read — save this for a rainy day.
It makes sense, but it is in-depth.

Functional Programming (H)

Lecture 17: Lambda Calculus & Equational Reasoning

Thu 28 Nov 2024 (slides v3)

Simon Fowler and Jeremy Sing

A large, stylized red starburst or speech bubble shape with a yellow outline and a white center. It is positioned in the bottom right corner of the slide.

last
examinable
lecture

News Update

- Next week - non-examinable (but very exciting) lectures
 - Coursework deadline - due Friday 6 December
 - Lab tomorrow - focus on Solitaire coursework
-
- Today - (FP)² - Functional Programming Fashion Parade - hats and jumpers
 - Evasys online questionnaires (?)

Today's learning objectives

- appreciate the differences between **C-like** and **Haskell-like** approaches to program evaluation
- recognize the *benefits* of **referential transparency**
- construct and calculate simple expressions in the **untyped lambda calculus**
- understand the motivation for **static analysis & reasoning** about programs in Haskell
- perform simple **structural induction** for functions that operate on lists

Referential Transparency

What is referential transparency?

- An expression is said to be *referentially transparent* if that expression can be replaced with its corresponding value without modifying the program behaviour.
- i.e. expression *value* is always independent of expression *context*
- cf. pure functions in Python / Java / ... / Haskell!
- expressions always evaluate the same way

non-referential transparency

- e.g. side-effecting code in C ...

```
int f(int x) {  
    static int secret = 0;  
    secret++;  
    return x+secret;  
}
```

hidden
preserved
state across
function calls

```
printf("%d\n", f(1) + f(1));  
// compare with ...  
int tmp = f(1);  
printf("%d\n", tmp + tmp);
```

prints 5

prints 4

Haskell example of Referentially Transparent Code

- `let f x = x+1`
- `let tmp = (f 1) in print $ tmp + tmp`
- `cf`
- `print $ (f 1) + (f 1)`

- these evaluate in precisely the same way!
- If we wanted to preserve state across calls to `f`, we would need to use the State monad.

Using State in Haskell

```
import Control.Monad.State
import Control.Monad.IO.Class (liftIO)

f :: Int -> StateT Int IO ()
f x = do
    secret <- get
    let result = x+secret+1
    liftIO $ putStrLn (show result)
    put (secret+1)
```

hidden preserved state across function calls

liftIO to do IO action within State transformer stack

```
main :: IO()
main = do
    ((), state') <- runStateT (f 1) 0
    ((), state'') <- runStateT (f 1) state'
    return ()
```

why is Referential Transparency good?

- good for **reasoning** about programs
- whether manually, or automated ... whether informally or formally
- good for **parallelism** - no side-effects / inter-thread dependencies
- evaluation is simply ***term rewriting***
 - cf evaluation of lambda terms

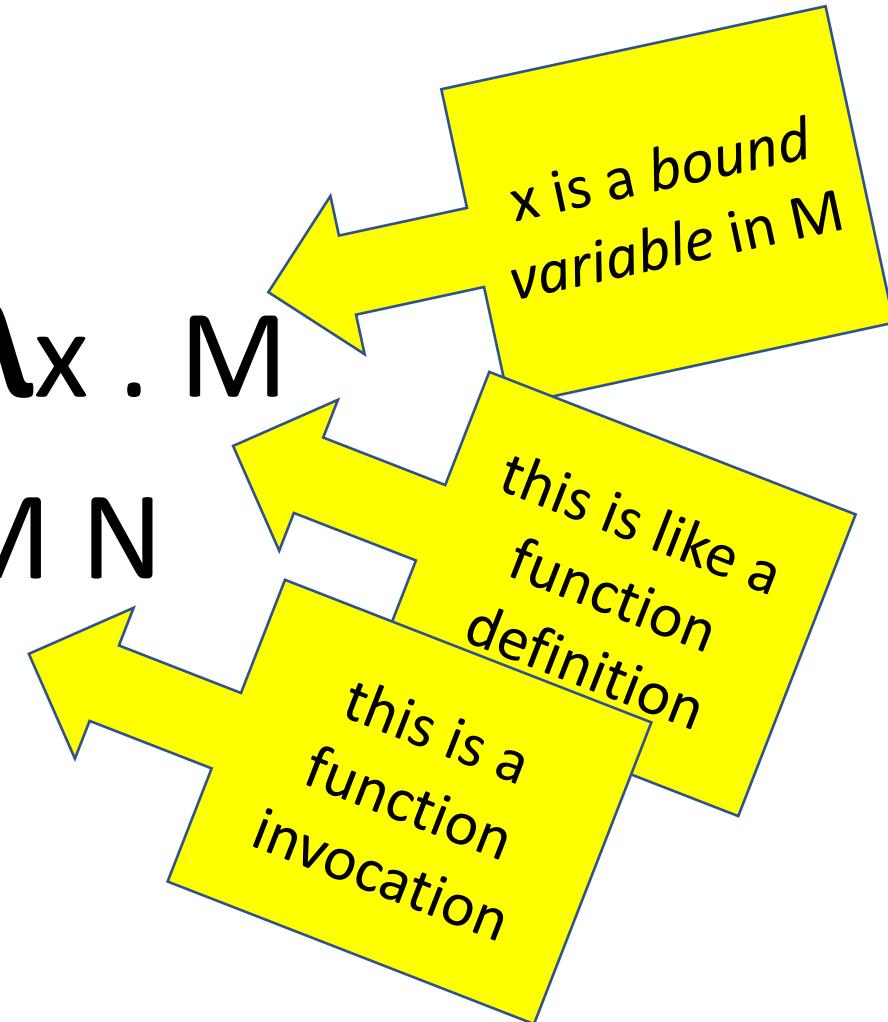
Lambda Calculus

Untyped Lambda Calculus

- invented by Alonzo Church in 1930s
- a theoretical model for computation
 - universal computation can be represented in lambda calculus!
- equivalent to Turing machines
 - Church-Turing thesis
- basis of functional programming
- evaluation involves term rewriting

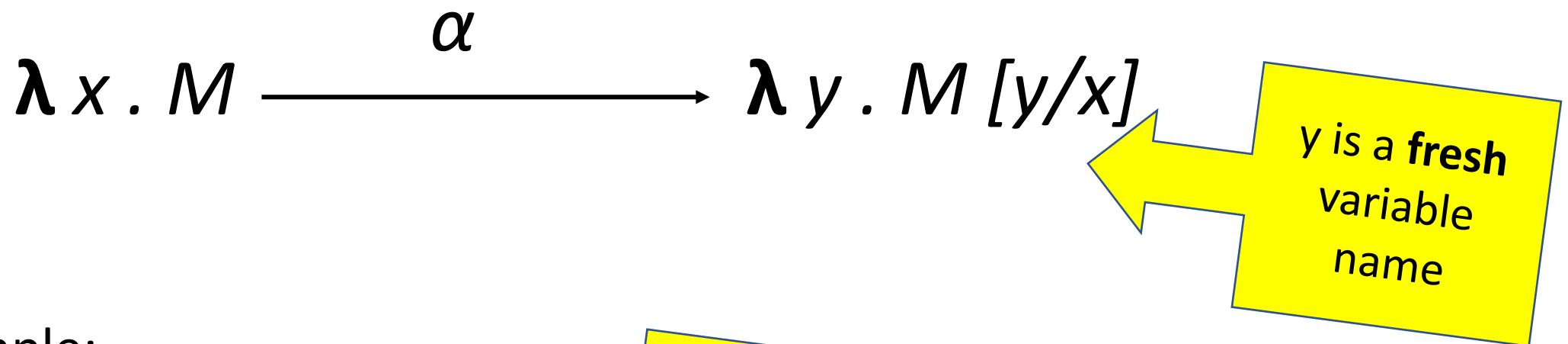
Components (terms) of Lambda Calculus

- **variables:** like x, y
- **function abstraction:** $\lambda x . M$
- **function application:** $M N$



Rewrite Rules for Lambda Calculus (1)

- *α conversion is bound-variable renaming*



- example:

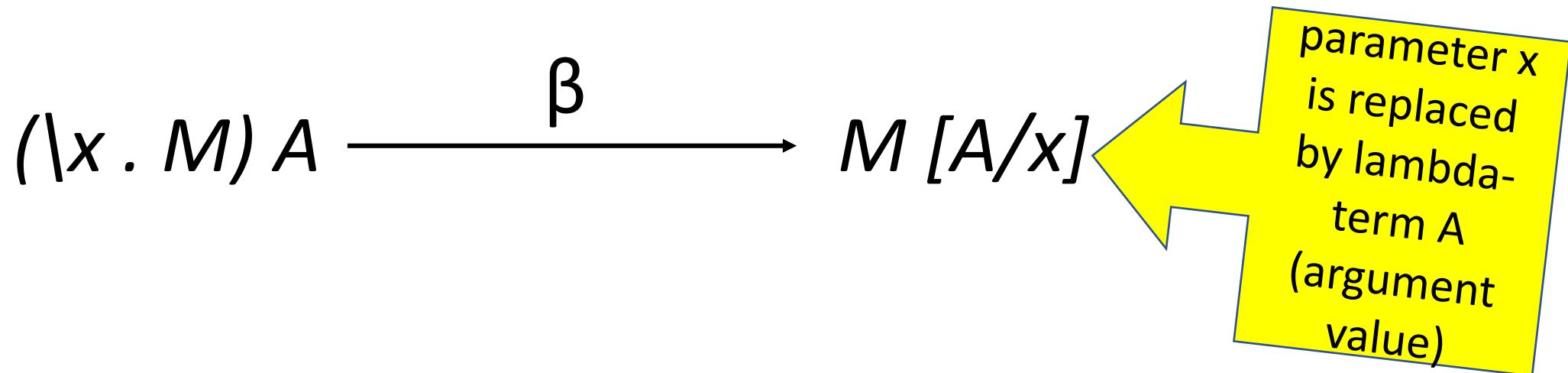
$$\lambda x . x \xrightarrow{\alpha} \lambda z . z$$

identity function is the same, no matter what name we give to its parameter

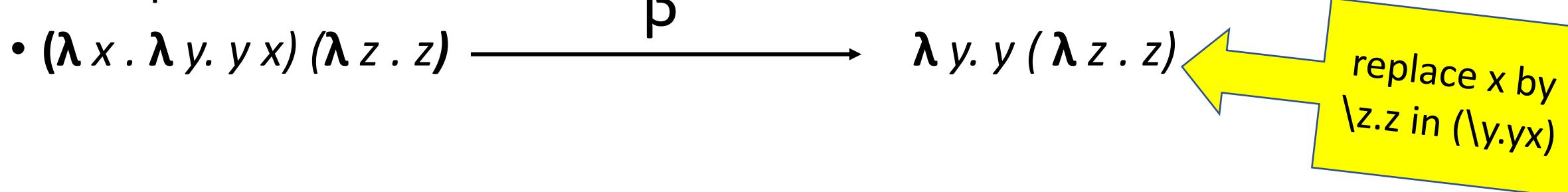
The diagram illustrates the α conversion rule with an example. It shows the transformation of the identity function $\lambda x . x$ into $\lambda z . z$. A horizontal arrow labeled α points from the left to the right. A yellow callout box contains the text "identity function is the same, no matter what name we give to its parameter".

Rewrite Rules for Lambda Calculus (2)

- β reduction is function call evaluation



- example:



Computability

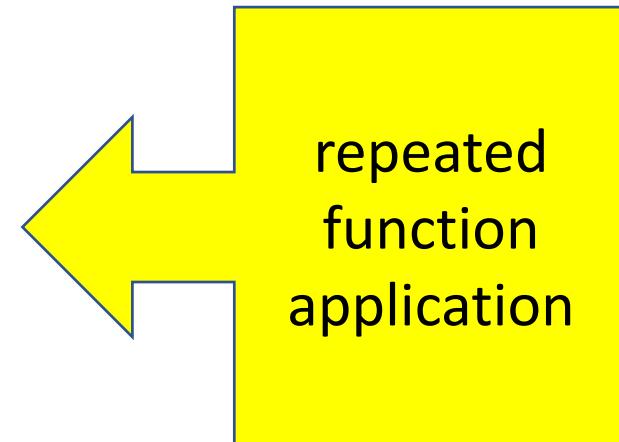
Any computable function can be represented in Lambda calculus

Church numerals - encoding of numbers:

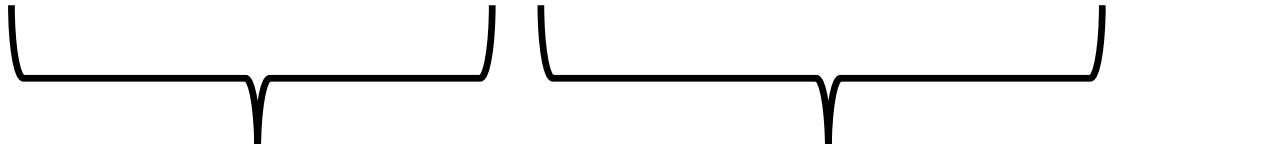
A Church numeral is a function that takes two parameters:

$\lambda f \rightarrow \lambda x \rightarrow \dots$

0: $\lambda f \rightarrow \lambda x \rightarrow x$
1: $\lambda f \rightarrow \lambda x \rightarrow f x$
2: $\lambda f \rightarrow \lambda x \rightarrow f(f x)$
n: $\lambda f \rightarrow \lambda x \rightarrow f^n x$ -- n applications of f
-- (iterate f x) !! n



Adding Church numerals

- add - take two Church numerals M and N and use M as the ‘zero’ parameter for N
 - $\text{add} = \lambda M N f x \rightarrow N f (M f x)$
 - this will evaluate to:
 - $f (f (f (\dots (f (\dots (f x))))))$


N applications of f M applications of f
- N applications of f, applied to M applications of f, applied to x

Decode Church numerals in Haskell

- **unchurch** :: $(a \rightarrow a) \rightarrow a \rightarrow a \rightarrow \text{Int}$
- **unchurch** $n = n (+1) 0$

Note about lambda calculus

- It's more of a theoretical representation, not actually used for computation in 'the real world'
- It shows the smallest 'programming language' that we can build to represent any computable function - such a programming language only needs function abstraction and function application
- Of course, although the programming language is trivially simple, the corresponding programs will be very verbose.

Extra (non-examinable) info about lambda calculus

- https://www.youtube.com/watch?v=eis11j_iGMs - Intro to Lambda calculus from Computerphile
- <https://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf> - Textbook on Lambda calculus

Equational Reasoning

program properties

- previously we used *QuickCheck* to do property based testing
- this was **dynamic testing** - use random input values to test specified invariants hold, for a fixed number of tests.
- Now - we want to perform **static verification** - prove properties of functions generally, that hold for all inputs ...

A simple example

- Given a function definition:

example :: Int -> Int -> Int -> Int

example $x \ y \ z = x * (y + z)$

- Prove that $\text{example } a \ b \ c == \text{example } a \ c \ b$

$\text{example } a \ b \ c$

$= a * (b + c)$ { example function def }

$= a * (c + b)$ { symmetry of + }

$= \text{example } a \ c \ b$ { example function def }

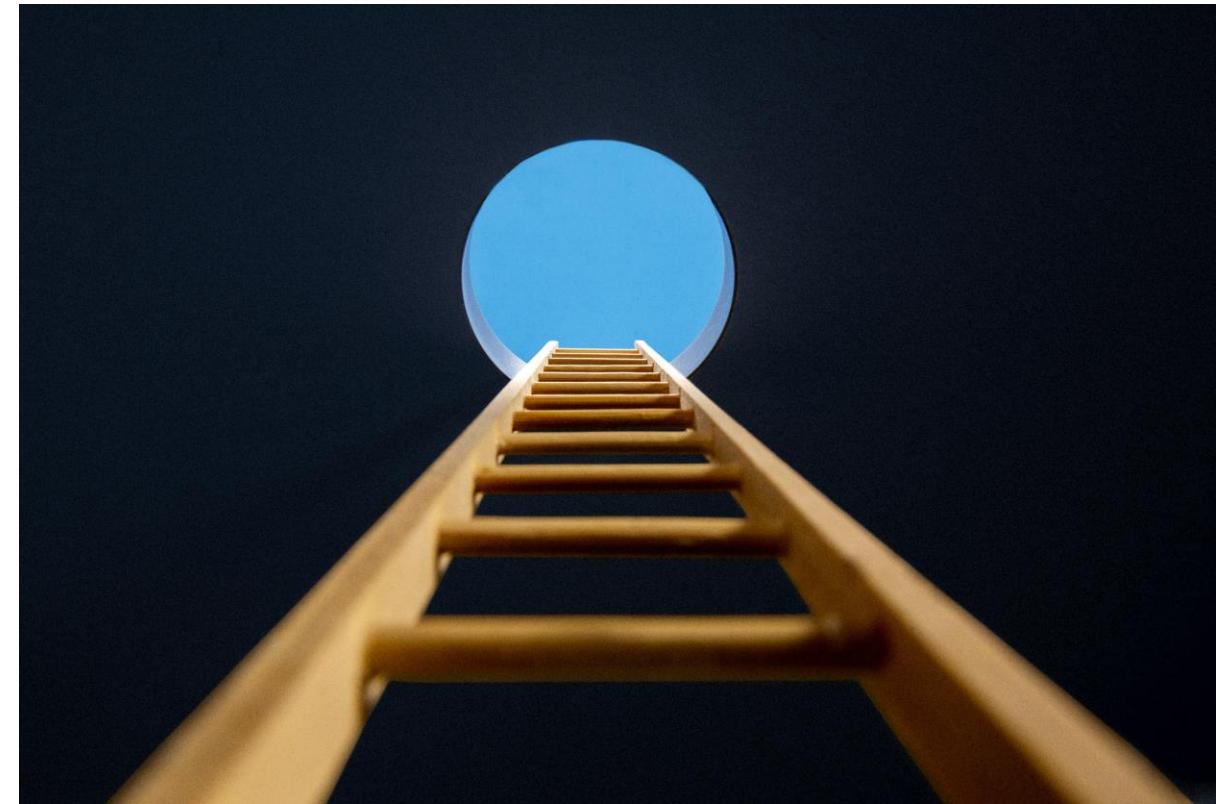
↑
rewrites

↑
justification for rewrites

only using function
definition and simple
properties of arithmetic

proofs by induction

- do you remember these from Algorithmic Foundations 2 course?
- or from high school maths?



structural induction

- given a property $p(x)$ where x is a list
- prove $p([])$
 - i.e. property holds for empty list (**base case**)
- prove that $p(\mathbf{xs}) \rightarrow p(\mathbf{x:xs})$
 - i.e. if property holds for list xs of length n , then we can show it holds for list $(x:xs)$ of length $(n+1)$ (**inductive step**)
- then by structural induction, p holds for arbitrary lists

Some rewrites we might need to make ...

-- *library definition of (++)*

`[] ++ ys = ys` -- base case

`(x:xs) ++ ys = x:(xs++ys)` -- recursive case

-- *library definition of length*

`length [] = 0`

`length (x:xs) = 1 + length xs`

example of structural induction

- We want to prove that, for all lists xs and ys :
 - $\text{length } (xs \text{ ++ } ys) = (\text{length } xs) + (\text{length } ys)$
 - base case, xs is $[]$

$$\begin{aligned} \text{length } ([] \text{ ++ } ys) &= (\text{length } []) + (\text{length } ys) \\ \text{definition of append} &\quad \text{definition of length} \\ \text{length } (ys) &= 0 + (\text{length } ys) \\ &\quad \text{arithmetic} \\ \text{length } ys &= \text{length } ys \end{aligned}$$

true

inductive step

- assume for list xs of length n ...

- $\text{length } (xs \text{ ++ } ys) = (\text{length } xs) + (\text{length } ys)$

- now, given list $(x:xs)$ of length $(n+1)$

$$\begin{aligned} \text{length } ((x:xs) \text{ ++ } ys) &= (\text{length } (x:xs)) + (\text{length } ys) \\ \text{\{definition of length\}} &\quad \text{\{definition of length\}} \\ 1 + \text{length } (xs \text{ ++ } ys) &= (1 + \text{length } xs) + (\text{length } ys) \\ \text{\{inductive hypothesis\}} &\quad \text{\{associativity of (+)\}} \\ 1 + ((\text{length } xs) + (\text{length } ys)) &= 1 + ((\text{length } xs) + (\text{length } ys)) \end{aligned}$$

true

A second example of structural induction

- We want to prove that, for all lists xs

$\text{length } \text{xs} \geq 0$

i.e. length of arbitrary list xs is non-negative

base case, xs is []

length [] == 0 { definition of length function}

0 >= 0 BASE CASE is proven

inductive step

- assume for list xs
 - $\text{length}(xs) \geq 0$
- Now need to prove that for list $x:xs$
 - $\text{length}(x:xs) \geq 0$

$\text{length}(x:xs) == 1 + \text{length}(xs)$ {definition of length}

We are assuming $\text{length}(xs) \geq 0$, so

$1 + \text{length}(xs) \geq 0$ {laws of arithmetic}

INDUCTIVE STEP IS PROVEN

Examples to try at home

- list append function (`++`) is associative
- Can also do structural induction over other recursive data types e.g. binary trees
- Further info online:
<https://www.youtube.com/watch?v=cQ1iziWpt8Q>

Takeaways

- **Lambda calculus** is a powerful formalism to model computing
- '*All you need is lambda*'
- **Equational reasoning** enables us to prove program properties
- Next week - non-examinable (but very exciting) lectures