Session 10: Summary

COMP2221: Functional programming

Laura Morgenstern

laura.morgenstern@durham.ac.uk

Recap

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

Revision: evaluation of foldr and foldl

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

foldr

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

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

foldl

Same idea, but associating to the left

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

Summary

Content overview

- Intro to functional programming paradigm
- Types I: built-in types, type checking
- Functions I: currying and λ-expressions
- Lists: pattern matching, comprehensions
- Types II: polymorphism, algebraic data types, type classes
- Recursion: structure, classification, and complexity
- Functions II: higher-order functions
- Evaluation strategies: lazy vs. eager
- λ-calculus: syntax and reduction rules

Functional programming paradigm

- A programming *paradigm* where the building block of computation is the *application of functions* to arguments.
- Functional programs specify a data-flow to describe *what* computations should proceed
- Algebraic programming style dominated by function application and composition

$\Rightarrow$ a functional language is one that *supports* and *encourages* programming in this style.

Data types

Type: collection of values

Haskell built-in types

- Int, Integer, Char, String, ...
- Lists `[1,2,3]`
- Tuples `(1,2,3)`

Haskell custom data types

- `type` keyword for synonyms
- `data` keyword for new algebraic types

Polymorphism

- Polymorphism: functions that are defined generically for many types.
- Types of polymorphism: parametric, ad-hoc, subtype polymorphism
  - Type variables: `length :: [a] -> Int` "a" is a type variable, length is generic over the type of the list.
  - Haskell uses *parametric polymorphism* "generic functions"
- Constraining polymorphic functions: type classes
  - `(+) :: Num a => a -> a -> a` "+ works on any type a as long as that type is numeric"
  - Relevant type classes: `Num` "numeric", `Eq` "equality", `Ord` "ordered"

⇒ Include class constraints in type definitions when appropriate

Lists

- Pattern matching: can match literal values but also match a list pattern, and bind the values

```
sumTwo :: Num a => [a] -> a
sumTwo (x:y:_) = x + y
```

- List comprehensions: constrcut new lists based on generator and guard expressions

```
[ x | x <- [1..5], even x]
```

Recursion

Recursion: a function that calls itself until it reaches a base case.

Definition (Tail recursion)

A function is *tail recursive* if the *last result of a recursive call* is the result of the function itself.

Definition (Linear recursion)

The recursive call contains only a *single* self reference.

Definition (Multiple recursion)

The recursive call contains *multiple* self references.

Definition (Direct recursion)

The function calls *itself* recursively.

Definition (Mutual/indirect recursion)

Multiple functions call *each other* recursively.

Functions

- Saw nameless or anonymous functions (λ-expressions), and syntax
- Formalises idea of functions defined using currying

```haskell
add x y = x + y
-- Equivalently
add = \x -> (\y -> x + y)
```

Definition (Higher order function)

A function that does at least one of

- take one or more functions as arguments
- returns a function as its result

- Due to currying, every function of more than one argument is higher-order in Haskell

Generalizing computational patterns

- Saw `Functor` for mappable types and `Foldable` for foldable types
- Instances must obey some equational laws

Functor laws

"Mapping behaves as expected"

```haskell
-- Distributes over composition
fmap (f . g) xs == fmap f (fmap g xs)
-- Preserves identity
fmap id xs == id xs
```

Evaluation strategies

- Lazy evaluation
  - Infinite data structures are fine, as long as we don't try and look at all of them
- Call by name (lazy) vs. call by value (eager) $\rightarrow$ contrast with imperative languages
- Think about expression as a graph of computations: multiple different evaluation orders possible

λ-calculus

- λ-calculus: set of rules to transform expressions of the following form
  - v (Variables; lower case letters)
  - (MN) (Application of M to N)
  - $(\lambda v.M)$ (Abstraction aka function with parameter v and body M)
  - with M and N being expressions of the same form
- α-conversion: solving name conflicts by renaming variables
- β-reduction: reducing expressions by applying functions to arguments

Summer exam

- Open book, tests mainly comprehension, application and synthesis
- Format: coding-based + conceptual questions

⇒ Practice programming in Haskell

⇒ Think about functional paradigms, look for them elsewhere. Has your mindset changed?

Relevant past paper questions

2022 Q1 (not (d)) and Q2

2021 Q1 (not (e))

2020 Q1 and Q2

2019 Q2 (the only Haskell question)

2018 Q1 (b–e, g) (not (a), (f))

Thank you!