Session 10: Summary

COMP2221: Functional programming

Lawrence Mitchell*

*lawrence.mitchell@durham.ac.uk

Summary

- Mathematically structured programming
- Types types types
- Forces us to think hard about interfaces/API boundaries

Concepts

- Imperative vs. functional
- Lazy vs. eager
- Referential transparency → result of pure language.
- Higher order programming and data types

Data types

Builtin

- Lists `[1,2,3]`
- Tuples `(1,2,3)`
- ...

Our own

```haskell
-- Polymorphic
-- New data type "Maybe",
-- with constructors "Nothing" and "Just"
-- Can contain a value of any type
data Maybe a = Nothing | Just a
-- Inductive
data List a = Empty | Cons a (List a)
```

Interfaces, constrained polymorphism

Constrained polymorphism

Ord, Eq

```
-- Type classes, describe an interface that
-- a type should satisfy
class Num a where
   (+) :: a -> a -> a
   ...
-- Instances implement the interface
instance Num Int where
   (+) = plusInt

-- Sum operates on any type "a" as long
-- as it satisfies Num interface
sum :: Num a => [a] -> a
```

More polymorphism

- Polymorphism: functions that are defined generically for many types.
  - 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”

$\Rightarrow$ Include class constraints in type definitions when appropriate

Principled type classes

- Saw `Functor` for mappable types and `Foldable` for foldable types
- Termed “Principled” type classes since implementations must obey some equational laws

Functor laws

“Mapping behaves as expected”

```
f :: (a -> b)
g :: (c -> d)
-- Distributes over composition
fmap (f . g) xs == fmap f (fmap g xs)
-- Preserves identity
fmap id xs == id xs
```

Should be able to show this correctness for simple definitions (e.g. see exercise 6 solutions)

Lazy evaluation

- Lazy evaluation
    - Infinite data structures are fine, as long as we don't try and look at all of them
- Call by name vs. Call by value (contrast with strict languages)
- Evaluation strategies and reducible expressions
- Think about expression as a graph of computations: multiple different orders possible
- What are Haskell's evaluation rules: normal form and weak head normal form
- Apply reduction rules (functions) until expression is in WHNF
- How to write strict function application with (`$!`)

Data types and API definitions

- Functional approach leads to hard to misuse APIs
- For example: compile-time correctness of protocol exchanges
- Concept: “make illegal states unrepresentable”
- Approach is gaining favour across the board, for safer APIs in critical software
- e.g. Rust and “type state” programming → especially in cryptography.

Summer exam

- Open book, tests application and synthesis (less focus on recall/knowledge)
- Format: coding-based questions + (short) "essay" covering broad-brush concepts

$\Rightarrow$ Practice programming in Haskell

$\Rightarrow$ Think about functional paradigms, look for them elsewhere. Has your mindset changed?

Relevant past paper questions

Sample	All questions
2021	All questions
2020	Q1 and Q2
2019	Q2 (the single Haskell question)
2018	Q1 (c–e, g) (not (a), (b), (f))
2017	Q1 and Q2. These are mostly programming questions that should be doable if you have looked at the practicals
2016	Q1 (a, c, e, g, h), Q2 (a, b, d, e)

Today

Definition

recursion *noun*

see: recursion.

Thank you!

Fin