Session 10: Summary

COMP2221: Functional programming

Lawrence Mitchell*
March 21, 2019

*lawrence.mitchell@durham.ac.uk
### Exam assesses

- **knowledge and comprehension**: how do things work in Haskell, why do they work, ...
- **application**: what does some code do; can you write code to solve problem X...
- **evaluation**: what are the concepts; what properties does some solution have...

### Remarks

- Practice via problem sheets (will cover programming knowledge)
- Types are important: *always write types in code*
- Theory, methodology, concepts from lectures are also relevant
- Please use exact terminology (definitions)
Session 1

- Functional languages, definition of side effects
- Difference between imperative and functional programming styles
- Why programming languages at all?
- Idea of abstract machine models
- Compilers serve to map from one paradigm (e.g. functional) to another (e.g. execution on CPU)
- First examples of Haskell
- Naming requirements: functions must start with lowercase letter
- Layout rule: whitespace alignment
- Comments
• First look at types
• Why use types? Correctness, documentation
• Typing in Haskell
• Defining types
  \[ e :: T \quad \text{-- e is of type } T \]
  \[ \text{not} :: \text{Bool} \rightarrow \text{Bool} \quad \text{-- Function type} \]
• Builtin types \texttt{Bool, Char, String, Int, Integer, ...}
• Lists: sequence of values of same type:
  \[ [1, 2, 3] :: [\text{Int}] \]
• Tuples: sequence of values of (different) types:
  \[ (\text{'a'}, 1) :: (\text{Char, Int}) \]
• Function types
  • currying: take arguments “one at a time”
  • association of \(-\rightarrow\) to the right, and function application to the left:
    \[ \text{mult} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \equiv \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})) \]
    \[ \text{mult } x \, y \, z \, \equiv \, (((\text{mult } x) \, y) \, z) \]
• Advice: even with type inference, *always write types for functions*

• Infix calling convention for binary operators:

\[
1 + 2 == (+) 1 2 \\
elem 1 xs == 1 `elem` xs
\]

• Defining functions

• Conditional expressions:

```haskell
if expr then true_expr else false_expr
```

• Guarded equations

```haskell
abs :: Int -> Int
abs n | n >= 0 = n  \\
| otherwise = -n
```

• Pattern matching

```haskell
not :: Bool -> Bool
not False = True  \\
not True = False
```

Patterns matched *in order* from top to bottom. Wildcard matches with `_`

• Pattern matching lists in session 4.
• Polymorphism: functions that are defined generically for many types.
  • Type variables: \texttt{length :: [a] -> Int} “a” is a type variable, length is generic over the type of the list.
  • Haskell uses \textit{parametric polymorphism} “generic functions”
• Constraining polymorphic functions: type classes
  • \texttt{(+) :: Num a => a -> a -> a} “+ works on any type a as long as that type is numeric”
  • Relevant type classes: \texttt{Num} “numeric”, \texttt{Eq} “equality”, \texttt{Ord} “ordered”
  \Rightarrow Include class constraints in type definitions when appropriate
• Generic programming in other languages (contrast with Java)
Session 4

- λ-expressions: “anonymous” functions
- Formalises the idea of currying
- Lists
  - List construction syntax \([1, 2, 3] == 1 : (2 : (3 : [])))\)
  - Linked list ⇒ traversing list or getting elements is \(O(n)\)
  - Brief interlude on \(O\) notation
- Pattern matching lists: use list constructor syntax
  ```haskell
  scan :: Num a => [a] -> a
  scan [] = []
  scan [x] = [x]
  scan (x:y:xs) = x : scan (x+y:xs)
  ```
- Binds variables in pattern to values: can’t repeat names!
- List comprehensions: similar to set builder notation in maths
  ```haskell
  pairs = [(x, y) | x <- [1..10], y <- [1..x], even y]
  ```
- Functionally similar to nested for loops
• Recursion
  • Idea: only solve simple problems, reducing more complicated ones to simpler ones
  • Step-by-step writing recursive functions (example with `drop`)
  • Classification of recursive functions: linear, multiple, direct, mutual/indirect. Tail recursion: a special case
  • “Complexity” of recursive functions: how many times do they call themselves. Linear: $O(n)$ calls on data of size $n$.

• Higher-order functions
  • A function which `takes a function as an argument`; or `returns a function as its result`
  • Core method of composition in Haskell (especially with currying)
  • Some examples: `map`, `filter`, `(.)`
  • Folds: `foldr`, `foldl`
Session 6

- Building new data types: `type` for synonyms; `data` for more complicated things
- Syntax: new type names must start with capital letter
- Data declarations introduce a new type and new `constructors`
  ```haskell
  -- New type "IsTrue"; New constructors Yes, No, Perhaps
  data IsTrue = Yes | No | Perhaps
  ```
- We can do pattern matching on the constructors
- Constructors can take parameters
- They can be polymorphic
  ```haskell
  data Maybe a = Nothing | Just a
  ```
- They can refer to themselves
  ```haskell
  data List a = Nil | Cons a (List a)
  ```
- Product vs. Sum types
- Pros and cons of Haskell’s “algebraic data types” and normal OO classes
- More on type classes: useful for writing generic code
• 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 ($!)
• Input and output
  • IO is a side-effectful action
    ⇒ does not immediately fit the pure functional paradigm
  • Hide it behind a special “action” type `IO a`
  • Conceptually IO destroys the universe and creates a new one

• `do` notation for executing actions and binding their results to variables

• Why we can’t treat IO with normal functions: referential transparency and impurity

• Actions as promises for a future value of a given type.

• A small example program (try it out!)
• Functional programming in the “real world”
• Material not examinable
<table>
<thead>
<tr>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>recursion <em>noun</em></td>
</tr>
<tr>
<td>see: recursion.</td>
</tr>
</tbody>
</table>
By its nature, cannot be exhaustive.

Past papers a good guide. Broadly they cover these types of questions:

- Can you write (short) Haskell functions and can you understand what (short) Haskell functions do? Type annotations, class constraints, pattern matching, guard expressions, conditionals.
- Can you use list-based functions from the standard library? `head`, `tail`, `length`, `map`, comprehensions, ...
- Can you explain/define key terms? Classes of recursion, types of polymorphism, currying, side effects, higher order functions, ...
- Can you explain/describe differences in different programming paradigms? Functional/imperative, pure/impure (side effects/side effect free), compiled/interpreted, lazy/strict, ...
Fin