Session 4: λ-expressions, list patterns, and comprehensions

COMP2221: Functional programming

Lawrence Mitchell[*]

[*]lawrence.mitchell@durham.ac.uk

Recap

- Saw how Haskell implements *polymorphism* through generic functions

```haskell
-- length operates on a list of any type a
-- and returns an Int
length :: [a] -> Int
```

- Saw how overloading works with *class constraints* and *type classes*

```haskell
-- sort sorts any list of things of type a,
-- as long as that type is orderable
sort :: Ord a => [a] -> [a]
```

- $\Rightarrow$ will go over this again when we see user data types.
- Recapitulated *currying* and the idea of functionals (functions that return other functions).
- Saw special syntax for calling binary functions

```haskell
foo :: [a] -> [a]
foo a b = ...
-- These two forms are equivalent
-- 1. foo x y
-- 2. x `foo` y
```

Lambda expressions

Nameless functions

- As well as giving functions names, we can also construct them *without* names using *lambda expressions*

```haskell
-- The nameless function that takes
-- a number x and returns x + x
\x -> x + x
```

- Use of λ for nameless functions comes from *lambda calculus*, which is a theory of functions.
- There is a whole formal system on reasoning about computation using λ calculus (developed by Alonzo Church in the 1930s) $\Rightarrow$ a different course
- It is also a way of formalising the idea of *lazy evaluation* (on which more later)

Use cases for unnamed functions I

- Formalises idea of functions defined using currying

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

- The latter form emphasises the idea that `add` is a function of one variable that returns a function
- Also useful when returning a function as a result

```haskell
const :: a -> b -> a
const x _ = x
-- Or, perhaps more naturally
const x = \_ -> x
```

"`const` eats an `a` and returns a function which eats a `b` and always returns the same `a`."

Use cases for unnamed functions II

- What good is a function which always returns the same value?
- Often when using *higher-order* functions, we need a base case that always returns the same value.

  ```
  length' :: [a] -> Int
  length' xs = sum (map (const 1) xs)
  ```

  “The length of a list can be obtained by summing the result of calling `const 1` on every item in the list”
- We will see some more of this when we look at *higher order* functions.

Use cases for unnamed functions III

- Also useful where the function is only used once

```haskell
-- Generate the first n positive odd numbers
odds :: Int -> [Int]
odds n = map f [0..n-1]
  where
    f x = x*2 + 1
```

- Can be simplified (removing the `where` clause)

```haskell
odds :: Int -> [Int]
odds n = map (\x -> x*2 + 1) [0..n-1]
```

Translating between the two forms

- It is always possible to translate between named functions and arguments, and the approach using λ expressions of one argument
- Just move the arguments to the right hand side and put it inside a λ, repeat with remainder until you're done.

```haskell
f a b c = ...
-- Move formal arguments to right hand side with a lambda
f = \a b c -> ...
-- move remaining arguments into new lambdas
f = \a -> (\b -> (\c -> ...))
```

- Which option fits more naturally is often a style choice
- *Pattern matching* is supported in the argument list in exactly the same way as normal functions

```haskell
head = \(x:_) -> x
```

- I sometimes find it easier to think about *composing* functions or currying by explicitly writing λ expressions

Building block summary

- Prerequisites: none
- Content
  - Introduce the idea of anonymous, or nameless functions
  - Saw syntax for these λ expressions
  - And how they can formalise (or make it easier to read) curried functions:

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

- Expected learning outcomes
  - student *knows* about anonymous functions
  - student can *use* λ expressions when defining functions
  - student can *translate* between λ expressions and “normal” function syntax.
  - student can describe connection between λ expressions and currying.
- Self-study
  - Write the `curry` and `uncurry` functions with a λ.

```haskell
curry :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c
```

Lists: patterns matching

Representation of lists

- Every non-empty list is created by repeated use of the `(:)` operator "*cons*truct" that adds an element to the start of a list

```
[1, 2, 3, 4] = 1 : (2 : (3 : (4 : [])))
```

- This is a representation of a *linked list*
- Operations on lists such as indexing, or computing the length must therefore *traverse* the list.

⇒ Operations such `reverse`, `length`, `(!!)` are linear in the length of the list.

- Getting the `head` and `tail` is constant time, as is `(:)` itself.

Pattern matching on lists

- lists can be used for pattern matching in function definitions

```
startsWithA :: [Char] -> Bool
startsWithA ['a', _, _] = True
startsWithA _ = False
```

- Matches 3-element lists and checks if the first entry is the character `'a'`.

Careful

Use patterns in the equations defining a function. Not in the type of the function.

Pattern matches in the equations don't change the *type* of the function. They just say how it should act on particular expressions.

Pattern matching on lists

- How match `'a'` and not care how long the list is?
- Can't use literal list syntax. Instead, use list constructor syntax for matching.

```
startsWithA :: [Char] -> Bool
startsWithA ('a':_) = True
startsWithA _ = False
```

- `('a':_)` matches any list of length *at least* 1 whose first entry is `'a'`.
- The *wildcard* match `_` matches anything else.
- This works to match multiple entries too:

```
startsWithAB :: [Char] -> Bool
startsWithAB ('a':'b':_) = True
startsWithAB _ = False
```

Binding variables in pattern matching

- As well as matching literal values, we can also match a (list) pattern, and bind the values.

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

- Match lists of length *at least* two and sum their first two entries

Example

```
sumTwo [1, 2, 3, 4]
-- introduces the bindings
x = 1
y = 2
_ = [3, 4]
```

- Reminder: can't repeat variable names in bindings (exception _)

```
-- Not allowed
sumThree (a:a:b:_) = a + a + b
-- Allowed
second (_:a:_) = a
```

What types of pattern can I match on?

- Patterns are constructed in the same way that we would construct the arguments to the function

```haskell
(&&) :: Bool -> Bool -> Bool
True && True = True
False && _ = False
-- Used as:
a && b
```

```haskell
head :: [a] -> a
head (x:_) = x
-- Used as:
head [1, 2, 3] == head (1:[2, 3])
```

- This is a general rule in constructing pattern matches “If I were to call the function, what structure do I want to match?”
- Caveat: can only match “data constructors”

```haskell
-- Not allowed
last :: [a] -> a
last (xs ++ [x]) = x
```

Lists: comprehensions

List comprehensions I: syntax

- In maths, we often use *comprehensions* to construct new *sets* from old ones

$$\{2, 4\} = \{x \mid x \in \{1..5\}, x \bmod 2 = 0\}$$

"The set of all integers x between 1 and 5 such that x is even."

- Haskell supports similar notation for constructing lists.

```
Prelude> [x | x <- [1..5], x `mod` 2 == 0]
[2, 4]
```

"The list of all integers x where x is drawn from `[1..5]` and x is even"

- `x <- [1..5]` is called a *generator*
- Compare Python comprehensions

```
[x for x in range(1, 6) if (x % 2) == 0]
```

List comprehensions II: generators

- Comprehensions can contain multiple generators, separated by commas

```
Prelude> [(x, y) | x <- [1,2,3], y <- [4, 5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

- Variables in the later generator change faster: analogous to nested loops

```
l = []
for x in [1, 2, 3]:
  for y in [4, 5]:
    l.append((x, y))

# analogously
[(x, y) for x in [1, 2, 3] for y in [4, 5]]
```

- Later generators can reference variables from earlier generators

```
Prelude> [(x, y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

“All pairs (x, y) such that $x, y \in \{1, 2, 3\}$ and $y \geq x$”

List comprehensions III: guards

- As well as binding variables to values with generators, we can restrict the values using *guards*
- A guard can be any function that returns a `Bool`
- Guards and generators can be freely interspersed, but guards can only refer to variables to their left

```
Prelude> [(x, y) | x <- [1..3], even x, y <- [x..3]]
[(2, 2), (2, 3)]
Prelude> [(x, y) | x <- [1..3], y <- [x..3], even x, even y]
[(2, 2)]
Prelude> [(x, y) | x <- [1..3], even x, even y, y <- [x..3]]
error: Variable not in scope: y :: Integer
```

Some examples

- Produce a list of all factors of some positive integer

```haskell
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]
```

- For example

```haskell
> factors 10
[1, 2, 5, 10]
```

- Now we can determine if a number is prime

```haskell
prime :: Int -> Bool
prime n = factors n == [1, n]
```

- And use it to (very expensively) enumerate primes below a limit

```haskell
primes :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]
```

Building block summary

- Prerequisites: none
- Content
  - Saw how the literal list syntax translates into construction with `(:)`
  - Discussed *implementation* and therefore complexity of common list operations
  - Made connection to pattern matching of lists
  - Introduced list comprehensions as analogous to set notation
  - Saw how nested comprehensions and guards work
- Expected learning outcomes
  - student *knows* how lists are implemented in Haskell
  - student can *use* pattern matching on list expressions to define functions
  - student can *use* list comprehensions to generate new lists
- Self-study
  - None