COMP304 Tutorial 2
22/07/2016

1 Pattern Matching

Pattern-matching lets the exact definition of a function change depending on the form of the arguments passed into it. We've already seen pattern-matching with the `fact` function.

```haskell
fact :: Int -> Int
fact 0 = 1
fact n = n * fact(n-1)
```

The patterns being matched on the left-hand side of the "=" sign are 0 and n. When Haskell sees an expression like `fact 2` it tries to match it against each definition of `fact`, top-to-bottom. This means the order in which you define patterns is important. If we switched the order around:

```haskell
fact n = n * fact(n-1)
fact 0 = 1
```

We'd get infinite recursion when trying to evaluate `fact 2`. This is because Haskell is able to match any `Int` to the first pattern `fact n`. Thus it never reaches the base case. It is doing something like this:

```
fact 2
→ 2 * fact(2-1)
→ 2 * fact(1)
→ 2 * (1 * fact(1-1))
→ 2 * (1 * fact(0))
→ 2 * (1 * (0 * (fact (0-1))))
→ 2 * (1 * (0 * (fact (-1))))
```

And so on. `GHCi` can sometimes warn us about situations like this. If you try to load the bad version of `fact.hs` it will tell you:

```
*Main> :l fact.hs
[1 of 1] Compiling Main ( fact.hs, interpreted )

fact.hs:2:1: Warning:
    Pattern match(es) are overlapped
    In an equation for ‘fact’: fact 0 = ...
```

If your patterns don't cover every possible case that could apply, Haskell won't complain. If a function can't be matched at runtime an exception is thrown. Here's a function which returns `True` if you pass it the empty list. If you try to pass it a non-empty list, `GHCi` will complain that your patterns are non-exhaustive (they don't cover every possible form of list).

```
isTheEmptyList :: [a] -> Bool
isTheEmptyList [] = True
```

```
*Main> isTheEmptyList []
True
*Main> isTheEmptyList [1,2,3]
*** Exception: fact.hs:3:1-19: Non-exhaustive patterns in function emptyList
```

If you run `GHCi` from the command-line like so: `ghci -fwarn- incomplete-patterns` it will tell you specifically what patterns are missing.

2 Guards

Here's the `fact` function defined with guards.

```
fact n
   | n > 0 = n * fact(n-1)
   | otherwise = 1
```

This version with guards is essentially the same as the following:

```
fact n = if n > 0 then n * fact(n-1) else 1
```

Guards end up being much more readable than using `if`...`then`...`else`.... Unlike pattern-matching, guards can test whether any arbitrary property of the input holds. However, pattern-matching can be used to de-structure the input and pull them apart. Here's an example using both to determine if every `Int` in a list is even.

```
isEven :: Int -> Bool
isEven n = rem n 2 == 0 -- even if remainder on dividing by 2 is 0

listIsEven :: [Int] -> Bool
listIsEven [] = True
listIsEven (x:xs) -- pattern-matching on the list
   | isEven x = listIsEven xs -- check if current x is even, recurse on rest of the list
   | otherwise = False
```

In mathematics we usually write arithmetic operations like `rem` as infix, rather than prefix. You can do this in Haskell with any function by using the backtick/grave character.

```
isEven :: Int -> Bool
isEven n = n ‘rem‘ 2 == 0
```

`n ‘rem‘ 2` is the same as `rem n 2`. Whether you choose to write some functions as infix is a stylistic choice. Note that ‘ is not an apostrophe; it's the grave/backtick character.

3 Polymorphism

Polymorphism is the ability for a single definition to automatically adapt and apply to multiple types. It helps facilitate code re-use and modularity.

Note the exact meaning of polymorphism is different across languages and paradigms. While the general idea is the same, it's realised in slightly different ways. In Java, polymorphism is achieved by method overloading and inheritance. In Haskell, it is achieved by ad-hoc polymorphism and parametric polymorphism.

Parametric polymorphism is the ability for the same function to operate on multiple types. For example, the identity function is defined on any type `a`. Depending on whether we pass e.g. an `Int` or a `Bool`, the `id` function may be acting like an `Int` → `Int` function or like a `Bool` → `Bool` function. Parametric polymorphism is a little bit like the use of generics in Java.

```haskell
id :: a -> a
id x = x
```

Ad-hoc polymorphism is the ability for functions to do different things depending on the types of the values it is being applied to. We've seen an example of this already:

```haskell
max :: Ord a => a -> a -> a
max x y
   | y > x = y
   | otherwise = x
```

The `max` function operates on arguments of type `a`, where `a` is in the `Ord` type-class. `Int` and `Char` are both in `Ord`. The function for comparing two `Ints` is different to the function for comparing two `Chars`, so `max 1 2` and `max 'a' 'c'` do different things when evaluated. But we don't have to account for it in the definition of `max`.
In general, if you see type-class constraints in the signature of a function, then ad-hoc polymorphism is at play.

4 Tail Recursion

In imperative languages each recursive call creates a new stack-frame to store variables local to that recursive call. If you make n recursive calls then your recursive function will be making $\mathcal{O}(n)$ stack frames. Typical linear recursion therefore has a space and time complexity of $\mathcal{O}(n)$.

If the last expression in a recursive function is returning the value of the recursive call, it is called tail recursive. Since a tail recursive function doesn't need to do anything except return after the recursive call, its local variables don't need to be stored anymore. The only stack frame that needs to be remembered is the most recent one. Tail recursive calls can be optimised to have a constant $\mathcal{O}(1)$ time complexity.

Many compilers (such as Haskell's `GHC`) will automatically optimise tail-recursive functions to be $\mathcal{O}(1)$ in space complexity. In an imperative language, tail recursion can be converted into an equivalent loop with $\mathcal{O}(1)$ space complexity.

Haskell doesn't *really* have a stack like imperative languages (think lazy evaluation), but the same principle applies. The following example is not tail recursive, because in the case for `myLen(x : xs)`, you must add 1 after computing `myLenxs`.

```haskell
-- Get the length of a list.
myLen :: [a] -> Int
myLen [] = 0
myLen (x:xs) = 1 + (myLen xs)
```

If we perform substitution on an example like `myLen`$[1, 1, 1, 1]$ we'll see the length of the lines grows proportional to the length of the list, i.e. the space complexity is $\mathcal{O}(n)$.

```
myLen [1,1,1,1]
→ 1 + (myLen [1,1,1])
→ 1 + (1 + (myLen [1,1]))
→ 1 + (1 + (1 + (myLen [1])))
→ 1 + (1 + (1 + (1 + (myLen []))))
→ 1 + (1 + (1 + (1 + 0)))
→ 1 + (1 + (1 + 1))
→ 1 + (1 + 2)
→ 1 + 3
→ 4
```

You can often rewrite functions to be tail-recursive by introducing extra parameters (often called accumulators) and helper functions. To make `myLen` tail-recursive we'll add an argument (called `acc`) which keeps track of how many times we've seen `goal` so far. When we get to the end of the list we return `acc`.

```haskell
-- Get the length of a list.
myLen :: [a] -> Int
myLen xs = myLen' 0 xs

-- Helper function to make it tail recursive.
myLen' acc [] = acc
myLen' acc (x:xs) = myLen' (acc+1) xs
```

Doing substitution on this version reveals it has $\mathcal{O}(1)$ space complexity, because each line is the same length. This optimised form is sometimes called "iteration".

```
myLen [1,1,1,1]
→ myLen' 0 [1,1,1,1]
→ myLen' 1 [1,1,1]
→ myLen' 2 [1,1]
→ myLen' 3 [1]
→ myLen' 4 []
→ 4
```

Final note: we could have just made `Count` take extra arguments instead of defining an extra `Count'` function, but it's nicer to have a helper because the person using your function doesn't have to know about the implementation details.

Final final note: there's another concept called "guarded recursion" which is more relevant to Haskell because of lazy evaluation. We may talk about it later, or you can find explanations online.

5 Tuples

Tuples can be used to group multiple types together to form a new type. This is useful for passing multiple values around together or returning multiple values.

In a language like Python, tuples are essentially immutable lists; this is not the case in Haskell. For example, tuples can be composed out of different types: (`3`, `True`) is a valid tuple but [`3`, `True`] is not a valid list. Here's a function which operates on tuples, using pattern-matching:

```
myMax :: Ord a => (a,a) -> a
myMax (x,y) = max x y
```

Take care not to confuse `max x y` and `max` (`x`, `y`). The first is trying to apply `max` to two arguments `x` and `y`; the second is trying to apply `max` to one argument, which is the tuple (`x`, `y`).

Haskell won't accept two functions with the same name but different signatures. The following is an error:

```
myMax :: Ord a => a -> a -> a
myMax x y
   | y > x = y
   | otherwise = x

myMax :: Ord a => (a,a) -> a
myMax (x,y) = myMax x y
```

`GHCi` will complain that you have two functions called `myMax` with different signatures.

```
:Prelude> :l mymax
fact.hs:7:1:
    Duplicate type signatures for ‘mymax’
    at fact.hs:2:1-5
       fact.hs:7:1-5

fact.hs:8:1:
    Multiple declarations of ‘mymax’
    Declared at: fact.hs:3:1
                 fact.hs:8:1
Failed, modules loaded: none.
```