CA320 - Computability & Complexity

Introduction to Haskell

Dr.David Sinclair

These are just notes! Programming requires practice.

- These notes are just to give you an overview of the Haskell programming language. The real work will be in the lab sessions.
- Attending and actively participating in the lab sessions is **vital**.
- Programming is part knowledge, part design and part skill. The *design* element required experience (which comes through **practice**). The *skill* element requires **practice**.
- Programming in Haskell, or any other functional language, requires you to think differently than programming in an imperative language.

Styles of Programming Languages

There are two basic styles of programming languages.

Imperative — In *imperative programming* you describe how to do something, usually as a series of sequential operations (that modify the state of the program) and conditional jumps (based on the state of the program). Imperative programming is all about *state*, and these state changes can be persistent and as a result have *side-effects*.

Declarative — In *declarative programming* you describe "what to do", not "how to do". Declarative languages fall into 2 broad categories, *logic programming* in which you describe the properties of the solution and then the programming language searches for the solution; and *functional languages* in which you describe the solution as the evaluation of mathematical functions that avoids state and side-effects.

Functional Languages and Referential Transparency

- A functional language program is constructed using *expressions*. Each *expression*, which can be a function call, has a *meaning* which evaluates to a *value*.
- There is no sequencing.
- There is no concept of state in the sense of variables and the values assigned to them.
- As a result of this there are no side-effects! An expression, or a function, simply evaluates to a value. If you replace an expression, or a function, by another expression, of function, that evaluates to the same values given the same arguments, it will have no effect on the rest of the program. This is called *referential transparency*.

Advantages of functional languages

Why use a language that does not have variables, assignments and sequencing?

- Concepts such as variables and sequencing only really make sense if you focus on the hardware of a computer. By focusing at this level we are really modeling a particular style of a *solution*. Functional programming is more abstract and focuses on modeling the *problem*.
- Functional programs are easier to reason about.
- Functional programming promotes good programming practices. Even if you never use function programming again, your programming in imperative languages will improve.

Disadvantages of functional languages

Functional languages are not perfect!

They are particularly difficult to use:

- when you require input and output (because these require side-effects); and
- when the program needs to run continuously and/or requires interaction with the user.

Because functional programs are at a level of abstraction above the hardware, it can be hard to reason about the time and space complexity of functional programs.

Haskell

The Haskell programming language was designed in 1998 by Peyton-Jones *et al* and is named after Haskell B. Curry who pioneered the λ-calculus, the mathematical theory on which functional programming is based.

Haskell is:

- a *purely functional language* with no side-effects and referential transparency
- a *lazy* programming language, in that it does not evaluate an expression unless it really has to; and
- a *statically typed* so that at compile time we know the type of each thing and has a *type inference* system that works out the type of a thing if it hasn't been explicitly specified.

Types

Everything in Haskell has a type!

There are *basic types*. Some examples are:

Keyword	type	value
`Int`	integer	12
`Float`	floating-point number	-1.23
`Char`	character	`'A'`
`Bool`	boolean	`True`

There are *compound types*. Some examples are:

Description	type	example
tuple	`(Int, Float)`	`(2,12.75)`
list	`[Int]`	[1,3,5,7,9]
string	`String`	`‘‘Hello there"`
function	`Int -> Int`	`square`

Functions and Function Types

Here is a quick example.

```
rectangleArea x y = x * y
```

A function has a *name* and a set of parameters, each separated by a space. How the function is evaluated is defined by what follows the = symbol.

This only tells part of the story of the `rectangleArea` function.

- A function always evaluates to a value which has a type.
- Each of the parameters of a function have a type.

```
rectangleArea ::  Float -> Float -> Float
rectangleArea x y = x * y
```

The parameters and return type are separated by `->` in the function declaration. The return type is the last item and the parameter types are the previous items.

if . . . then . . . else

Haskell has an `if` . . . `then` . . . `else` expression.

```
doubleSmallNumber x = if x > 100
                        then x
                        else x*2
```

Because it is an expression we can use `if` . . . `then` . . . `else` inside another expression.

```
incDoubleSmallNumber x = (if x > 100 then x else x*2) + 1
```

Lists

Lists are a very useful data structure. They are a *homogeneous* data structure where each element of the list has the same type.

```
let numbers = [1,3,5,7,9,11,13,15]
```

creates a list called `numbers`.

```
let joined = [1,3,5,7,9] ++ [2,4,6,8,10]
```

The ++ operator *concatenates* two lists, in this case generating the list `[1,3,5,7,9,2,4,6,8,10]` called `joined`.

Strings are lists of characters so,
`"Hello " ++ "World"`

evaluates to `"Hello World"`.

Lists (2)

The *cons* operator, `:`, adds an element to the start of a list.

```
let mylist = 1:[5,4,3,2]
```

evaluates to a list called `mylist` containing `[1,5,4,3,2]`.

```
1:2:3:[]
```

evaluates to the list `[1,2,3]`.

`[]` is the empty list.
So `3:[]` evaluates to `[3]`.
`2:[3]` evaluates to `[2,3]`.
`1:[2,3]` evaluates to `[1,2,3]`.
In fact, `[1,2,3]` is syntactic sugar for `1:2:3:[]`.

Lists (3)

To access a specific element of a list we use the `!!` operator. The `!!` operator, like the ++ operator, is an infix operator. Before the `!!` operator we have the list we are accessing and after the `!!` operator we have the index of the element we are accessing (indices start at 0).

```
[[1,2,3],[2,4,6,8],[1,3,5,7,9]] !!  2
```

results in the list `[1,3,5,7,9]` since this list is at index 2 of the lists of lists.

How would we get the value 7 from `[[1,2,3],[2,4,6,8],[1,3,5,7,9]]`?

Lists can be compared lexicographically using the <, <=, > and >= operators. The first elements are compared and if they are equal the second elements are compared, and so on.

Some useful List functions

`head [2,4,6,8]` evaluates to 2.

`tail [2,4,6,8]` evaluates to `[4,6,8]`.

`last [2,4,6,8]` evaluates to 8.

`init [2,4,6,8]` evaluates to `[2,4,6]`.

`length [2,4,6,8]` returns the size of the list, which in this case evaluates to 4.

`null [2,4,6,8]` tests whether or not a list is empty, which in this case evaluates to `False`.

`reverse [2,4,6,8]` reverses a list, which in this case evaluates to `[8,6,4,2]`.

`take 2 [2,4,6,8]` extracts the specified number of elements from the start of a list, which in this case evaluates to `[2,4]`.

Some useful List functions (2)

`drop 3 [2,4,6,8]` removes the specified number of elements from the start of a list, which in this case evaluates to `[8]`.

`maximun [2,8,4,9,6]` returns the largest element of a list, which in this case evaluates to 9.

`miniumum [2,8,4,9,6]` returns the smallest element of a list, which in this case evaluates to 2.

`sum [2,8,4,9,6]` returns the sum of all the elements of a list, which in this case evaluates to 29.

`product [2,8,4,9,6]` returns the sum of all the elements of a list, which in this case evaluates to 384.

`elem 4 [2,8,4,6]` tests if the specified element is contained in a list, which in this case evaluates to `True`. This can also be written as an infix operator, i.e. `4 ‘elem‘ [2,8,4,6]`.

Lists and Ranges

Rather than having to write down all the elements of a list, we can use *ranges* if there is a regular interval between the elements.

`[1,2..10]` evaluates to the list `[1,2,3,4,5,6,7,8,9,10]`.

`[2,4..20]` evaluates to the list `[2,4,6,8,10,12,14,16,18,20]`.

Ranges also work with characters.

`['a'..'z']` evaluates to the string `"abcdefghijklmnopqrstuvwxyz"`.

Ranges can be used to generate infinite lists, e.g.,

```
[1,2..]
[12,24..]
```

List Comprehension

When we defined sets we used a technique called *set comprehension*, e.g. $\{2x | x \in \mathcal{N}, x \leq 10\}$.
The expression before the pipe operator, (|), is called the output function. In this case the output function is a function of x. The expressions after the pipe are the predicates that the variables of the output function must satisfy.

We can do something very similar for lists and this is called *list comprehension*.

```
[ 2*x | x <-[1..10]]
```

```
[ x*y | x <- [5,10,15], y <- [4..6], x*y < 50]
```
evaluates to `[20,25,30,40]`

What does this function do?

```
mysteryFn :: String -> String
mysteryFn x = [y | y <- x, y `elem` ['a'..'z']]
```


Tuples

Lists are homogeneous collections of data. If we want to collect together data of *different types* we need to use *tuples*.

A *tuple* has its own type and that type depends on its size, the types of its components and the order in which they occur.

The tuples `(5,"hello")` and `("world", 1)` have different types since the first tuple has the type `(Int, String)` whereas the second tuple has the type `(String, Int)`.

A *pair* is the smallest tuple and pairs have 2 functions, `fst` and `snd` for extracting data from them.

`fst (5,"hello")` evaluates to 5 and `snd(5,"hello")` evaluates to `"hello"`.

Extracting data from larger tuples, such as triple, 4-tuples, 5-tuples, etc., requires pattern matching.

Tuples (2)

For example, we could use a tuple to represent a triangle. Each element of the tuple is the length of one of the sides. Let's limit ourselves to triangles with integer side lengths.

We could generate a list of triangles whose perimeter was at most 24 as follows.

```
[(a,b,c) | a <-[1..24], b<-[1..24], c<-[1..24], a+b+c
<= 24]
```

This generates a lot of triangles most of which are duplicates since the triangles `(10,12,2)`, `(12,10,2)`, `(2,12,10)` and many more are effectively the same triangle. We can remove the "duplicates" by letting `a` be the longest side, `b` the second longest and `c` the shortest side.

```
[(a,b,c) | a <-[1..24], b<-[1..a], c<-[1..b], a+b+c
<= 24]
```


Tuples (3)

Still a lot of triangles, but how many of these are right-angle triangles?

```
length [(a,b,c) | a <-[1..24], b<-[1..24],
c<-[1..24], a+b+c <= 24, a^2 == b^2 + c^2]
```

Typeclasses

You can find out the type of a item including functions by using the `:t` command.

`:t head` evaluates to `head::[a]->a`

which says that `head` is a function that takes an array of things of type `a` and evaluates to a thing of type `a`.

`:t (==)` evaluates to `(==)::(Eq a)=>a->a->Bool`

What does the `(Eq a)` before the => signify?

`(Eq a)` is an example of a *class constraint*. In this case, the == function takes two items of the same type and this type must belong to the `Eq` class. The `Eq` class includes all standard Haskell types except IO and functions.

Typeclasses (2)

Some other type classes are:

Ord This is the *Ordered* class. Functions using `Ord` are >=, <=, >, <, `compare`, `max` and `min`. Most Haskell standard types belong to `Ord` except function types and some abstract types.

Show Types belonging to this class can be displayed as strings. Again most Haskell standard types belong to `Show` except function types.

Read Contains all the type that can be taken in as a string and converted into a type. Again all standard types are included in `Read`, but sometimes some additional information is required if the intended type is unclear. `read "2"` returns an ambiguous type error, so `read "2"::Int` is required.

Num This is the *Numeric* class and includes all types that act as numbers.

Integral This class includes only whole numbers.

Pattern Matching and Recursion

Pattern marching and *recursion* are very common programming constructs in functional programming. *Recursion* is where a a function definition calls itself when performing an evaluation. To prevent an infinite loop, a recursive function must have a *base case* and the parameters of each invocation of the recursive function should move closer to the *base case*.

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n-1)
```

- The 2nd line is the *base case*.
- Lines 2 and 3 demonstrate pattern matching. If the parameter to the `factorial` function is 0 the 2nd line is evaluated. If the parameter is not 0 the 3rd line is evaluated. The parameter is matched to `n` and the `factorial` function is evaluated with `n-1`.

Pattern Matching and Recursion (2)

We can write our own version of the standard list`length` function.

```
listLength :: (Integral b) => [a] -> b
listLength [] = 0
listLength (_:xs) = 1 + listLength xs
```

- The *base case* is the empty list.
- If the list we are measuring is not the empty list we try the next pattern `_:xs`.
  - The list is matched against `_:xs`, which is a list with something at its head (start) followed by a tail.
  - The tail of the list is matched with `xs`.
  - The head of the list is matched with `_`. We use `_` when we don't care what the value we are matching against is.
  - It is **very important** to make sure that the patterns you specify match against all possible patterns.

as Patterns

To refer to the original complete item and not just the elements of the pattern, we can use *as patterns*. An *as pattern* consists of the name of the complete item followed by an `@` and then the pattern.

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++
                        " is " ++ [x]
```

We need to be careful with the amount of whitespace we use at the start of a line in Haskell as Haskell is whitespace sensitive.

- The first non-whitespace after a `do`, `let` or `where` defines the start of a *layout block* and its column number is rememberd.
- If a new line has the same amount of whitespace as the start of the layout block it is a new line.
- If a new line has more whitespace, it is a line continuation.
- If a new line has less whitespace, it is a new line.

Guards

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "Have a steak!"
    | weight / height ^ 2 <= 25.0 = "Supposedly normal!"
    | weight / height ^ 2 <= 30.0 = "How about a walk?"
    | otherwise                   = "Skip that snack"
```

- *Guards* follow a function name and parameters. They are boolean expressions that follow a pipe (|) symbol.
- It is good practice to ensure the pipes are aligned.
- When a guard evaluates to `True`, the function body is evaluated.
- When a guard evaluates to `False`, the next guard is checked.
- `otherwise` is defined as `True` and acts as a “catch-all”.

Where

The `where` keyword allows you to share *name bindings* within a function.

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "Have a steak!"
    | bmi <= normal = "Supposedly normal!"
    | bmi <= comfortable = "How about a walk?"
    | otherwise = "Skip that snack"
    where bmi = weight / height ^ 2
                (skinny, normal, comfortable)
                       = (18.5, 25.0, 30.0)
```

- Note that you can use pattern matching inside `where` blocks.

Where (2)

Here is another example of the `where` keyword and pattern matching.

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

`where` bindings can also be nested.

- It is common when defining a function to define some helper function in its where clause and then to give those functions helper functions as well, each with its own where clause.

Let

`let` bindings are far more local than `where` bindings. `let` bindings have the form:

`let` *bindings* `in` *expressions*

For example:

```
cylinderArea :: (RealFloat a) => a -> a -> a
cylinderArea r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r^2
    in  sideArea + 2 * topArea
```

`let` bindings are expressions in themselves, so we can write expressions such as:

```
4 * (let a = 9 in a + 1) + 2
```


Case

case expressions are like case statements from imperative languages but with pattern matching added.

```
headOfList :: [a] -> a
headOfList xs = case xs of []     -> error "Empty list!"
                           (x:_) -> x
```

But couldn't we have written `headOfList` as:

```
headOfList :: [a] -> a
headOfList [] = error "Empty list!"
headOfList (x:_) = x
```

Yes, they are interchangeable because pattern matching in function definitions is just syntactic sugar for case expressions.

More Recursion

Let's implement our own versions of a few standard list functions

```
takeList :: (Num i, Ord i) => i -> [a] -> [a]
takeList n _
       | n <= 0    = []
takeList _ []      = []
takeList n (x:xs) = x : takeList (n-1) xs
```

Note that there are 2 base cases, `n <= 0` and the empty list.

```
reverseList :: [a] -> [a]
reverseList [] = []
reverseList (x:xs) = reverseList xs ++ [x]
```

While this works it is not the most efficient solution as concatenating strings takes time.

More Recursion (2)

A common trick with recursive functions is to use an *accumulator*.

```
reverseListAcc :: [a] -> [a] -> [a]
reverseListAcc [] x = x
reverseListAcc (x:xs) y = reverseListAcc xs (x:y)

reverseList2 :: [a] -> [a]
reverseList2 x = reverseListAcc x []
```

Quicksort

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```


Higher Order Functions

Why do Haskell function type signatures have structures like `a->a->b->c` and not `a,a,b->c`?

Because Haskell functions actually only take **one** parameter!

Consider the `max` function.

```
max :: (Ord a) => a -> a -> a
```

When `max 4 5` is evaluated it creates a new function that takes one parameter and evaluates to `4` if the new parameter is less than `4` or the new parameter if it is greater than `4`.

So `max 4 5` and `(max 4) 5` are equivalent and a `<space>` between 2 items is a *function application*.

`max` is an example of a *curried function*.

Higher Order Functions (2)

Why use *curried functions*?

Because when you supply a curried function with less parameters than required you get back a *partially applied function* which is like a new function that used as a parameter in another function.

```
comparehi = compare 100
comparelo = compare 20

checkThreshlods :: (Num a) => a -> (a->Ordering) ->
                                 (a-> Ordering) -> String
checkThresholds x hi lo
    | high = "too high"
    | low = "too low"
    | otherwise = "OK"
   where (high, low) = (hi x == LT, lo x == GT)

compareThresholds 80 comparehi comparelo
```


Haskell

Higher Order Functions (3)

`map` takes a function and applies it to every element of a list.

Some examples:

- `map (>3) [1,8,2,4]` evaluates to `[False,True,False,True]`
- `map fst [(3,2),(1,4),(2,5)]` evaluates to `[3,1,2]`
- `map (map (^2)) [(3,2),(1,4),(2,5)]` evaluates to `[(9,4),(1,16),(4,25)]`

`filter` takes a predicate and a list and evaluates to a list of the elements of the original list that satisfy the predicate.

- `filter (>3) [1,8,2,4,5,1,6]` evaluates to `[8,4,5,6]`
- `let notNull x = not (null x) in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]` evaluates to `[[1,2,3],[3,4,5],[2,2]`.

Higher Order Functions (4)

Find the sum of all odd squares that are smaller than 10,000.

- `sum` is a function that sums the elements of a list.
- `takeWhile` is a function that removes elements from that start of a list while a predicate is true.

```
sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

Things to note:

- An infinite data structure.
- Functions as parameters.
- Partially applied functions.

Lambda Functions

These are anonymous functions that are mainly used as a parameter to a higher-order function.

```
chain :: (Integral a) => a -> [a]    -- Collatz chains
chain 1 = [1]
chain n
    | even n =  n:(chain (n `div` 2))
    | odd n  =  n:(chain (n*3 + 1))

numLongChains :: Int
numLongChains = length (filter (\xs -> length xs > 15)
                               (map chain [1..100]))
```

Lambda functions are expressions and hence can be passed to functions.

Input/Output

Input/Output (I/O) can be tricky in a purely functional environment since there are no side-effects and no changes of state. For the contents of a screen to change there has to be some state change.

Haskell provides a mechanism for I/O that isolates the state changes from the rest of the Haskell program, maintaining the benefits of a purely functional programming language.

```
main = do
         putStrLn "What's your name?"
         name <- getLine
         putStrLn ("Hi " ++ name")
```

- I/O is performed by *IO actions*.

Input/Output (2)

- `putStrLn`'s type is `String->IO()`, i.e. it takes a `String` and evaluates to an *IO action* with a result type of `( ), empty tuple`.
- `getLine`'s type is `IO String`, i.e. it evaluates to an *IO action* with a result type of `String`.
- The `do` keyword creates a block of *IO actions* that are "executed" in sequence and act as one composite *IO action*.
- The syntax `name <- getLine` binds the result of `getLine` to `name`.

Input/Output (3)

A common construct is to print out a list of IO actions.

```
main = do
    rs <- sequence [getLine, getLine, getLine]
    sequence $ map print rs
```

- $ is the function application operator. `x $ y` applies function `x` to the results of `y`

`mapM` maps the function over the list and then sequences it. `mapM_` does the same, only it throws away the result.

```
main = do
    rs <- sequence [getLine, getLine, getLine]
    mapM_ print rs
```


Input/Output (4)

Other useful I/O functions are:

- `putStr` write a string to the terminal but doesn't terminate it with a newline.
- `getChar` which reads a character from the terminal.
- `getContents` which reads everything from the terminal until an end-of-file character.
  - The key aspect of `getContents` is that it is a *lazy* function. It doesn't read everything from the terminal and store it somewhere. Instead it return but not actually read anything from the terminal **until you really need it**!

File Input/Output

File I/O is done by *IO Handles*.

```
import System.IO

main = do
    handle <- openFile "file.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

- `openFile` opens a file in either `ReadMode`, `WriteMode`, `AppendMode` or `ReadWriteMode`. `openFile` returns an `IO Handle` which can be bound to an identifier.
- `hGetContents` is line `getContents` except it takes a handle.
- `hClose` takes a file handle and closes the file.
- `hPutStr`, `hPutStrLn`, `hGetLine`, `hGetChar` are like their counterparts, but they take an additional parameter, an `IO Handle`.