Functional Programming and Haskell

Tim Dawborn

University of Sydney, Australia
School of Information Technologies
What are Programming Paradigms?

A **programming paradigm** is a fundamental style of programming regarding how solutions to problems are to be formulated in a programming language.

- **Declarative Programming** – describes what you want, rather than how to do it
  - SQL, XSLT

- **Imperative Programming** – computation is a set of statements which change the programs overall state
  - Fortran, Pascal, C

- **Logic Programming** – use of mathematical logic for computer programming
  - Prolog, Datalog, Gödel

- **Object Oriented Programming** – models “real-world” objects and their interactions to design programs
  - Java, C#, O’Caml
Functional Programming

- Treats computation as a series of mathematical functions
- Avoids states and mutable data
- Foundation concepts include:
  - pure functions
  - λ-calculus
- e.g. APL, Scheme, Lisp, Haskell
Pure Functions

- Have no **side effects**

- Retain **referential transparency**
  - thus pure functions are thread safe by definition

Example pure functions:
- sin(x)
- length("hello world")
- encrypt(msg)

Example impure functions:
- random()
- global x = 4; void doit(){ x = 42; }
Lambda Calculus ($\lambda$–calculus)

- $\lambda$–calculus is a formal system used within computer science and formal logic theory
- Can be used to define what a computable function is
- Concerned with entities called $\lambda$–terms
  - $v$ variable name
  - $\lambda v.E1$ abstraction (anonymous function)
  - $(E1 \ E2)$ $\lambda$–terms
- Checking for equivalence between two $\lambda$–calculus expressions is undecidable
- Easy to think of as an anonymous pure function (implies referential transparency)
- e.g. square = $\lambda y. y \times y$
Lambda Calculus ($\lambda$–calculus)

What if we have a list of numbers $L$ and we wanted to create a new list $L' = [ n | n^3 + 3n + 3 \ \forall n \in L]$? (in Python)

```python
def doit(items):
    r = []
    for i in items:
        r.append(i**3 + 3*i + 3)
    return r
doit([1,2,3,4,5])
```

```python
def doit(n):
    return n**3 + 3*n + 3
map(doit, [1,2,3,4,5])
```

```python
map(lambda x: x**3 + 3*x + 3, [1,2,3,4,5])
```
Introduction to Haskell

- Functional, interpreted language
- Lazy call-by-name evaluation — does not evaluate the function calls; knows the relation and will evaluate when needed

```
List makeList() {
    List m = new List();
    m.value = 2;
    m.next = makeList();
    return m;
}
```

```
makeList = 2 : makeList
```
Introduction to Haskell

- All functions are pure (no side effects)
- Does not support destructive updates: the code
  
  ```
  int x = 5;
  x = x + 1;
  ```

  has no equivalent in Haskell
- Easy to read (and understand ?) code as a result

  ```
  qsort :: (Ord a) => [a] -> [a]
  qsort [] = []
  qsort (x:xs) = qsort (filter (< x) xs)
                  ++ [x]
                  ++ qsort (filter (>= x) xs)
  ```
Syntax

- No parenthesis’s used for invoking functions
- Arbitrary precision integers

> 3 * 5 + 1
16

> 2 ^ 50
1125899906842624

> sqrt 2
1.4142135623730951

> 5 == 4.5 - (-0.5)
True
Lists \[ ]

- Homogeneous
- **Cons** and **concatenation** operators are "\:" and "++" respectively

> 0 : [1, 2, 3, 4, 5]  
[0, 1, 2, 3, 4, 5]

- This displayed pre-formed list syntax as shown above is in-fact syntactic sugar – the compiler breaks this down into individual elements cons’d together with a concluding empty list. i.e.

> 0 : 1 : 2 : 3 : 4 : 5 : []  
[0, 1, 2, 3, 4, 5]

- String’s are lists’ of Char’s
- Some useful functions for lists include head, tail, length, and null
**Tuples ( )**

- Heterogeneous $k$-tuple’s, $(k \in \mathbb{N} > 0)$

- Special inbuilt functions for pairs (2-tuple)
  - \texttt{fst} — gets 1st item in a pair
  - \texttt{snd} — gets 2nd item in a pair

Using a combination of standard tuple operations, how do you get out the character from $((1, 'a'), "foo")$?

\[
\texttt{snd (fst ((1, 'a'), "foo"))}
\]
Four key standard list processing functions: map, filter, foldl, and foldr

```haskell
> map Char.toUpper "Hello World"
"HELLO WORLD"

> map Char.isUpper "Hello World"
[True, False, False, False, False, False, True, False, False, False, False]

> filter Char.isLower "Hello World"
"elloorld"
```
Folding is a key concept used a lot in functional programming.

Syntax is: `foldr` function `init_val` list

Replaces occurrences of `cons` with the function parameter, and replaces the empty list constructor (end of the cons sequence) with the initial value.

Associativity is crucial in folding: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$?
- `foldr` is right associative
- `foldl` is (surprise surprise) left associative
List Processing – Folding

> foldr (−) 1 [4, 8, 5]
  ---> 4 − (foldr (−) 1 8:5:[])
  ---> 4 − (8 − (foldr (−) 1 5:[]))
  ---> 4 − (8 − (5 − (foldr (−) 1 [])))
  ---> 4 − (8 − (5 − (1)))
0

> foldl (−) 1 [4, 8, 5]
  ---> (foldl (−) 1 4:8:[]) − 5
  ---> (((foldl (−) 1 4:[]) − 8) − 5
  ---> ((((foldl (−) 1 []) − 4) − 8) − 5
  ---> (((1) − 4) − 8) − 5
−16
How can I compute the number of lowercase letters in a String?

```
length (filter Char.isLower "aBCde")
```

```
(length . filter Char.isLower) "aBCde"
```

Given that “max” compares two numbers and returns the max, write a function using folding that returns the max item in a list of integers, or zero if the list is empty

```
foldr max 0 [5,10,2,8,1]
```
Implement the built-in “filter” function

```haskell
my_filter :: (a -> Bool) -> [a] -> [a]
my_filter _ [] = []
my_filter f (x:xs) =
  if f x
    then x : (my_filter f xs)
    else my_filter f xs
```

Implement the built-in “map” function

```haskell
my_map :: (a -> b) -> [a] -> [b]
my_map _ [] = []
my_map f (x:xs) = (f x) : my_map f xs
```
Code Snippets

Implementation of a prime sieve:

```haskell
primes = sieve [2..] where
  sieve (p:xs) =
    p : sieve (filter (\x -> x `mod` p > 0) xs)

take 20 primes
```

Implementation of a `uniq` function for a list:

```haskell
uniq :: (Eq a) => [a] -> [a]
uniq [] = []
uniq (x:xs) = x : (uniq . filter (/=x)) xs
```
Code Snippets

Implementation of the built-in “foldr” function

\[
\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\]
\[
f \text{foldr} _ s \ [\] = s
\]
\[
f \text{foldr} f s (x:\text{xs}) = f x \ (\text{foldr} f s \ \text{xs})
\]

Non-optimised implementation of quicksort

\[
\text{qsort} :: (\text{Ord } a) \Rightarrow [a] \rightarrow [a]
\]
\[
\text{qsort} \ [\] = [\]
\]
\[
\text{qsort} \ (x:\text{xs}) = \text{qsort} \ (\text{filter } (< x) \ \text{xs})
\]
\[
\qquad + [x]
\]
\[
\qquad + \text{qsort} \ (\text{filter } (> = x) \ \text{xs})
\]
Three different ways to implement “factorial”: two recursive and one iterative

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

```
fact :: (Integral a) => a -> a
fact n | n <= 0    = 1
        | otherwise = n * fact (n-1)
```

```
fact :: (Integral a) => a -> a
fact n = foldr (*) 1 [1..n]
```
Implement the built-in “zip” function

```haskell
zip :: [a] -> [b] -> [(a, b)]
zip [] _ = []
zip _ [] = []
zip (y:xy) (z:xz) = (y,z) : zip xy xz
```

Does the following implementation also work?

```haskell
zip :: [a] -> [b] -> [(a, b)]
zip ay@(y:xy) az@(z:xz) =
  | null ay || null az = []
  | otherwise = (y,z) : zip xy xz
```
Conclusion

- This was a very basic and brief intro into Functional Programming in Haskell

- For a good introduction into Haskell, have a go at “Yet Another Haskell Tutorial (YAHT)”

- COMP3109 – Programming Languages and Paradigms
References