Introduction to Functional Programming

Outline

1. What is FP?
2. History
3. Haskell basics
4. More Haskell features
5. The outside world
6. Advantages of Haskell and FP

What is functional programming?

FP is an approach to programming which is older than OO but which is becoming increasingly popular. There are many functional languages, ranging from ancient to cutting edge.

- Lisp family: Common Lisp, Scheme,
- ML family: OCaml, F#
- Purely functional: Clean, Haskell,
- Erlang (Ericsson, telecoms).

What is functional programming?

- The basic unit of code is the function, and always has a return value.
- Referential transparency. No side effects and functions always give the same value for the same input.
- No destructive assignment, no shared state (so no loops!).
- Functions can be passed as arguments and returned as values – higher order.

What is functional programming?

Functional languages tend to be:

- Very concise with minimal syntax.
- Focused on recursion rather than iteration.
- Declarative: focused on what should be done rather than how something should be done.
- More amenable than imperative languages to algebraic reasoning.

History

Alonzo Church and the pure λ-calculus.

The λ-calculus was originally a framework in which to explore what is now known as computability theory.

What is computable in theoretical computer science?

Very simple and powerful system of functional abstraction and application. Formally equivalent to Turing machines.

History

Alonzo Church and the pure λ-calculus.

Based on the idea of an anonymous function, e.g.

$$\lambda x.E$$

In the above term x is the argument to the function and the term E is the body.

Reducing the term means applying the function by supplying a value for x, e.g.

$$(\lambda x.E)\ y$$

History

Alonzo Church and the pure λ-calculus.

Note: any term can be a function, so

$$\lambda x.(\lambda y.x)$$

is a valid function. What does it return?

History

Alonzo Church and the pure λ-calculus.

Note: any term can be a function, so

$$\lambda x.(\lambda y.x)$$

is a valid function. What does it return?

Given a term x, it returns the function that takes any argument y, and returns x.

History

Alonzo Church and the pure λ-calculus.

Example:

$$\begin{aligned} \lambda x.(\lambda y.x)\ 42\ 9 &= \\ (\lambda y.\ 42)\ 9 &= 42 \end{aligned}$$

History

Alonzo Church and the pure λ-calculus.

Note: an expression can be an application, so

$$\lambda x.(x\ y)$$

is a valid λ-function. What does it return?

History

Alonzo Church and the pure λ-calculus.

Note: an expression can be an application, so

$$\lambda x.(x\ y)$$

is a valid λ-function. What does it return?

Given a name x as argument, it returns the application $x\ y$.

History

Alonzo Church and the pure λ-calculus.

Example:

$$\begin{aligned} \lambda x.(x\ y)(\lambda z.z) &= \\ \lambda z.z\ y &= z\ y \end{aligned}$$

History

Alonzo Church and the pure λ-calculus.

Some useful functions:

$\lambda x.x$	identity
$\lambda x.(x\ x)$	self-application
$\lambda x.\lambda y.(x\ y)$	function application

History

Simply-typed λ-calculus.

In the pure λ-calculus any term can be applied to another, but self-application will never terminate.

$(\lambda x.(x\ x))\ \lambda x.(x\ x)$

$(\lambda(\lambda x.(x\ x)).(x\ x))$

$(\lambda x.(x\ x))\lambda x.(x\ x)$

$\ldots$

History

Simply-typed λ-calculus.

The answer is types, and the simply-typed λ-calculus.

In the simply-typed λ-calculus every term has a type.

If x has type A and e has type B, then the lambda-term $\lambda x.e$ has the type $A \to B$, written

$$\frac{x : A \quad e : B}{(\lambda x.e) : A \to B}$$

History

Simply-typed λ-calculus.

If f has type $A \rightarrow B$ and x has type A, then $f\ x$ has the type B, written

$$\frac{f : A \rightarrow B \quad x : A}{f\ x : B}$$

History

Simply-typed λ-calculus.

Functional programming is very closely related to the simply-typed λ-calculus.

Abstraction and application work in the same way.

There are no concepts of global variables or loops, but these are not required.

Haskell basics

We are using the purely functional language Haskell.

Haskell has an industrial-strength, cross-platform compiler that produces fast code, and a very powerful and modern type system which has inspired changes to Python, Java, C#.

Haskell basics

The basic types

Int, Bool, Char, String, Tuples

```
5 :: Int                    'Y' :: Char

True :: Bool                False :: Bool

"Hello" :: String           asciiCode :: Char -> Int

(42, "foo") :: (Int, String)
```

Haskell basics

Function definition, lambdas, if .. then .. else

Haskell is whitespace-sensitive! Indent carefully.

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

square ::  Int -> Int
square x = x*x

square' ::  Int -> Int
square' x = (\y -> y*y) x
```

Haskell basics

Function definition, lambdas, if .. then .. else

```
lt :: Int -> Int -> Bool
lt x y = if x < y then True else False

fst :: (a, b) -> a
fst (x, y) = x

not :: Bool -> Bool
not p = if p then False else True
```

Haskell basics

Function definition, lambdas, if .. then .. else

```
not ::  Bool -> Bool
```

What does `not` do? There are only three things it might do:

- return it's argument unchanged,
- return a constant value,
- negate its argument.

It cannot write a file, access a database or pick a random value. This makes it possible for us to reason about our code more easily.

Haskell basics

Lists

Lists are very widely used, with special syntactic support.

- Literal lists: `[1, 2, 3]`
- Ranges: `[1..99]`
- Adding to a list: `'A': ['b', 'c']`
- Joining lists: `[1, 2, 3] ++ [4, 5, 6]`
- List comprehensions: `[(x, x) | x <- [0..]]`

A `String` is a list of `Char`s.

Haskell basics
Lists

Pattern matching on lists.

```
null ::  [a] -> Bool
null [] = True
null (x:xs) = False
```

Haskell basics

Let vs where

Two ways to introduce a local variable.

```
f ::  ([Int], [Int])
f = let x = [1..9] in (x, x)

g ::  ([Int], [Int])
g = (x, x)
  where x = [1..9]
```

Haskell basics

The head and tail of a list

```
head ::  [a] -> a
head [] = error "empty list"
head (x:xs) = x

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

```
> head ‘‘Why not.’’
’W’
> tail [1..5]
[2, 3, 4, 5]
```

Haskell basics

Summing a list of Ints

sum is our first function that calls itself recursively. In an imperative language we might use a loop.

```
sum ::  [a] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```

Haskell basics

Summing a list of Ints

Evaluating `sum`.

`sum [5, 6, 7]`	
`5 + (sum [6, 7])`	Definition of `sum`
`5 + 6 + (sum [7])`	Definition of `sum`
`5 + 6 + 7 + sum []`	Definition of `sum`
`5 + 6 + 7 + 0`	Definition of `sum`
`18`	Definition of `(+)`

Haskell basics

Applying a function to every element of a list

```
map ::  (a -> Bool) -> [a] -> [a]
map f [] = []
map f (x:xs) = f x :  map f xs
```

```
> map (\x -> x < 3) [1..5]
[True, True, False, False, False]
```

Haskell basics

Filtering a list of Ints

```
isAsciiLower :  Char -> Bool

filter ::  (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x then x :  (filter p xs)
                  else filter p xs
```

```
> filter isAsciiLower "How Do You Do."
"owoouo"
```

Haskell basics

User defined types

```
data Bool       = True | False

data Colour     = Red | Blue | Green | RGB (Int, Int, Int)

data Customer   = Customer {id :: Int,
                  name :: String,
                  address :: [String] }
```

Haskell basics

User defined types

```
data BinTree a = Leaf a | Node a (BinTree a) (BinTree a)

tree ::  BinTree Char
tree = Node 'a' (Node 'b' (Leaf 'c') (Leaf 'd')) (Leaf 'e')

size ::  BinTree a -> Int
size (Leaf _) = 1
size (Node _ t1 t2) = 1 + (size t1) + (size t2)
```

Haskell basics

User defined types

Compare the Haskell definition of BinTree with one in Java.

Functional Programming

Functional programming is powerful and fun!

Try the Haskell exercises.

References

The Haskell wikibook: http://en.wikibooks.org/wiki/Haskell
Real World Haskell (O'Reilly book):
http://www.realworldhaskell.org

Day 2

More Haskell features.

More Haskell features

Haskell is lazy!

Haskell's semantics are call by need. Terms will not be evaluated until they are needed.

Laziness makes it easy to create infinite data structures, also known as streams, or codata.

```
ones ::  [Int]
ones = 1 :  ones
```

```
ints ::  [Int]
ints = [1..]
```

More Haskell features

More handy types

Maybe is the type of computations which might fail.

```
data Maybe a = Just a | Nothing

safeHead ::  [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x

applyWithDefault ::  (a -> b) -> a -> [a] -> b
applyWithDefault f def xs = case safeHead xs of
                Nothing -> f def
                Just x -> f x
```

More Haskell features

Typeclasses: Show

Typeclasses provide support for generic programming and a form of overloading. Similar to but not the same as Java interfaces.

```
class Show a where
       show ::  a -> String

instance Show Colour where
       show Red = "Red"
       ...

data Colour = Red | Green | Blue deriving Show

show-hi ::  Show a => a -> String
show-hi c = "Hi, " ++ show c
```

```
> show-hi 5
"Hi, 5"
> show-hi True
"Hi, True"
```

More Haskell features

Typeclasses: Eq

```
class Eq a where
       (==) ::  a -> a -> Bool
       (/=) ::  a -> a -> Bool

data Colour = Red | Green | Blue deriving (Show, Eq)

elem ::  Eq a => a -> [a] -> Bool
elem x [] = False
elem x (y:ys) = if x == y then True else elem x ys
```

```
> elem '.'  "Why not."
True
> elem Red [Blue, Blue, Red, Green]
True
```

More Haskell features

Typeclasses: Ord

```
class Eq a => Ord a where
       compare ::  a -> a -> Ordering
       (<) ::  a -> a -> Bool
       ...

data Colour = Red | Green | Blue deriving (Show, Eq,
Ord)
```

More Haskell features

Typeclasses: Ord

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = (quicksort lt) ++ [x] ++
(quicksort gt)
        where lt = filter (\y -> y <= x) xs
              gt = filter (\y -> y > x) xs
```

```
quicksort "Why not."
> "  .Whnoty"
quicksort [Green, Blue, Red, Green, Red]
> [Red, Red, Green, Green, Blue]
```

More Haskell features

Sections and partial application

In reality, all Haskell functions take one argument. Functions with a signature `(a -> b -> c)` really take one argument of type `a` and return a function with the signature `b -> c`. If elem has the type `a -> [a] -> Bool`, what is the type of (elem 'c')?

More Haskell features

Sections and partial application

In reality, all Haskell functions take one argument. Functions with a signature `(a -> b -> c)` really take one argument of type `a` and return a function with the signature `b -> c`. If elem has the type `a -> [a] -> Bool`, what is the type of (elem 'c')?

```
> :t (elem 'c')
(elem 'c') :  [Char] -> Bool
```

(Note that this is no longer polymorphic.)

More Haskell features

Sections and partial application

Here's another example of a function which has been partially applied. `(+1)` is the function which takes an `Int` and adds one to it.

```
> map (+1) [1, 2, 3]
[2, 3, 4]
```

More Haskell features

Sections and partial application

And another: `(dropWhile isSpace)` is the function which takes `String`s and removes leading spaces.

```
> map (dropWhile isSpace) [" 2", "f", " i"]
["2", "f", "i"]
```

More Haskell features

Pointsfree style

Writing code in a pointsfree style means leaving the arguments out of the definition. The simplest case is to omit the last argument:

```
toUpper ::  Char -> Char

upperString ::  String -> String
upperString s = map toUpper s
```

More Haskell features

Pointsfree style

Writing code in a pointsfree style means leaving the arguments out of the definition. The simplest case is to omit the last argument:

```
toUpper :: Char -> Char

upperString :: String -> String
upperString s = map toUpper s
```

is the same as

```
upperString :: String -> String
upperString = map toUpper
```

(Think about the types of map and toUpper)

More Haskell features

Functional composition

Functional composition means to combine two or more functions into a single function. This is powerful way to reuse code:

```
(.)  ::  (b -> c) -> (a -> b) -> a -> c
(.)  f g x = g (f x)
```

More Haskell features

Functional composition

Functional composition means to combine two or more functions into a single function. This is powerful way to reuse code:

```
(.)  ::  (b -> c) -> (a -> b) -> a -> c
(.)  f g x = g (f x)
```

```
> ((take 3) .  drop 1) [1 ..  7]
[2, 3, 4]
```

```
capCount = length .  filter (isUpper .  head) .
words
```

```
> capCount "Hello there, Everybody!"
2
```

More Haskell features

Functional patterns : folds

We have been seeing lots of examples of primitive recursion. FP is all about abstractions, so can we abstract this? We need to provide a case for the empty list (the base case) and a case for lists with elements (the inductive case).

More Haskell features

Functional patterns : folds

We have been seeing lots of examples of primitive recursion. FP is all about abstractions, so can we abstract this? We need to provide a case for the empty list (the base case) and a case for lists with elements (the inductive case).

```
foldr ::  (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Tip: `(foldr f b)` takes a list `(x:y:z:[])` and replaces `(:)` with `f` and `[]` with `b`.

More Haskell features

Familiar functions as folds

Folds reduce a list down to a single value.

```
sum' ::  [a] -> Int
sum' = foldr (+) 0
```

More Haskell features

Familiar functions as folds

Folds reduce a list down to a single value.

```
sum' ::  [a] -> Int
sum' = foldr (+) 0
```

```
sum' [1, 2, 3]
```

More Haskell features

Familiar functions as folds

Folds reduce a list down to a single value.

```
sum' ::  [a] -> Int
sum' = foldr (+) 0
```

```
sum' [1, 2, 3]
foldr (+) 0 [1, 2, 3]
```
Definition of `sum'`

More Haskell features

Familiar functions as folds

Folds reduce a list down to a single value.

```
sum' ::  [a] -> Int
sum' = foldr (+) 0
```

`sum' [1, 2, 3]`	
`foldr (+) 0 [1, 2, 3]`	Definition of `sum'`
`(+) 1 (foldr (+) 0 [2, 3])`	Inductive case of `foldr`

More Haskell features

Familiar functions as folds

Folds reduce a list down to a single value.

```
sum' ::  [a] -> Int
sum' = foldr (+) 0
```

`sum' [1, 2, 3]`	
`foldr (+) 0 [1, 2, 3]`	Definition of `sum'`
`(+) 1 (foldr (+) 0 [2, 3])`	Inductive case of `foldr`
`(+) 1 ((+) 2 (foldr (+) 0 [3]))`	Inductive case of `foldr`

More Haskell features

Familiar functions as folds

Folds reduce a list down to a single value.

```
sum' ::  [a] -> Int
sum' = foldr (+) 0
```

`sum' [1, 2, 3]`	
`foldr (+) 0 [1, 2, 3]`	Definition of `sum'`
`(+) 1 (foldr (+) 0 [2, 3])`	Inductive case of `foldr`
`(+) 1 ((+) 2 (foldr (+) 0 [3]))`	Inductive case of `foldr`
`(+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))`	Inductive case of `foldr`

More Haskell features

Familiar functions as folds

Folds reduce a list down to a single value.

```
sum' ::  [a] -> Int
sum' = foldr (+) 0
```

`sum' [1, 2, 3]`	
`foldr (+) 0 [1, 2, 3]`	Definition of `sum'`
`(+) 1 (foldr (+) 0 [2, 3])`	Inductive case of `foldr`
`(+) 1 ((+) 2 (foldr (+) 0 [3]))`	Inductive case of `foldr`
`(+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))`	Inductive case of `foldr`
`(+) 1 ((+) 2 ((+) 3 0))`	Base case of `foldr`

More Haskell features

Familiar functions as folds

Folds reduce a list down to a single value.

```
sum' ::  [a] -> Int
sum' = foldr (+) 0
```

`sum' [1, 2, 3]`	
`foldr (+) 0 [1, 2, 3]`	Definition of `sum'`
`(+) 1 (foldr (+) 0 [2, 3])`	Inductive case of `foldr`
`(+) 1 ((+) 2 (foldr (+) 0 [3]))`	Inductive case of `foldr`
`(+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))`	Inductive case of `foldr`
`(+) 1 ((+) 2 ((+) 3 0))`	Base case of `foldr`
6	Definition of $(+)$

More Haskell features

Familiar functions as folds

An amazing number of functions can be expressed as folds, and it's a good idea to do so:

- They make for elegant code,
- They have predictable behaviour,
- We only need to learn how they work once,
- Others reading our code will readily understand them.

The same goes for the other higher order functions we've seen, map, filter etc.

More Haskell features

Familiar functions as folds

Some of the functions we've already seen, redefined as folds:

```
concat ::  [[a]] -> [a]
concat = foldr (++) []

length ::  [a] -> Int
length = foldr (const (1+)) 0

map ::  (a -> b) -> [a] -> [b]
map f = foldr ((:)  .  f x) []

filter ::  (a -> Bool) -> [a] -> [a]
filter p = foldr (\x ys -> if p x then x :  ys else
ys) []
```

The outside world

Problem: purity!

All the functions we have seen so far are pure. That means they have no side-effects and we can be sure what will happen when we call them.

Input/Output (IO) is inherently impure. Most languages don't worry about this but seperating pure from impure code is a good discipline.

The outside world

Monads and IO

A program that interacts with the outside world:

```
under30 ::  String -> String
under30 s = let i = (read s) ::  Int in
        if i < 30 then "Under 30" else "Over 30"

main ::  IO ()
main = do putStrLn "Enter your age"
          x <- getContents
          putStrLn (under30 x)
```

Advantages of Haskell and FP

Why would I use this?

Productivity benefits:

- Higher level.
- Sophisticated abstractions are easy to create and understand.
- Fewer lines of code.
- Typechecking helps with debugging.
- Very well suited for tasks involving parallelism and concurrency.

Advantages of Haskell and FP

Parallelism and concurrency

Haskell is ready for many cores, with minimum fuss for the programmer.

Some of the things that make this easy are referential transparency and a sophisticated compiler.

If two functions have no side effects and no shared data, the compiler can choose to spark threads on separate cores where possible and allow the main process to combine the results when they are available.

Advantages of Haskell and FP

A parallel program

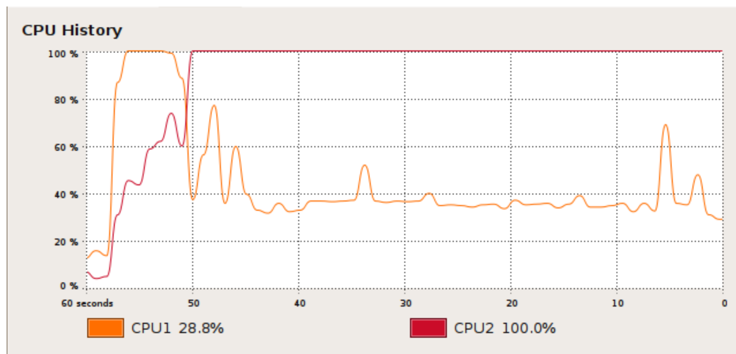
This number-crunching program uses a lot of CPU time. par and pseq instruct the compiler to utilise multiple cores if possible. This is all the effort we need to make!

```
import Control.Parallel

main = a `par` b `par` c `pseq` print (a + b + c)
              where a = ack 3 10
                    b = fac 42
                    c = fib 34

fac 0 = 1
fac n = n * fac (n-1)

ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))

fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Advantages of Haskell and FP

A parallel program

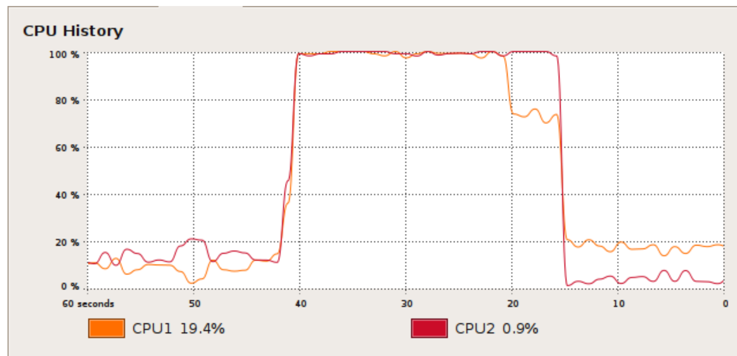
Running with no parallelism (runs for about one minute).



Advantages of Haskell and FP

A parallel program

Running the same program after telling the compiler to use two cores (runs for about 25s).



Functional Programming

Functional programming is powerful and fun!

Try the Haskell exercises.

References

The Haskell wikibook: http://en.wikibooks.org/wiki/Haskell
Real World Haskell (O'Reilly book):
http://www.realworldhaskell.org