Functional Programming and FOSS

Les Kitchen
Department of Computing and Information Systems
University of Melbourne
Linux Users of Victoria
luv-fp-foss@po.ljk.id.au

2 November, 2016

Agenda

The Four Paradigms

Imperative	Object-Oriented
Logic	Functional

Imperative Fortran, 1953/1957 Algol, 1958/1960	**Object-Oriented** Simula, 1967
Logic Planner, 1969 Prolog, 1972	**Functional** Lisp, 1958/1962

Functional Programming

Functional Programming

Programming with “first class” functions

- functions as function arguments
- functions as function results
- functions as data
- “higher order” functions
- abstraction of data and control

FP variations

- pure versus impure
- static versus dynamic typing
- lazy versus eager evaluation
- lexical versus dynamic binding

FP “extremes”

- Traditional Lisp
  - impure, dynamic typing, eager evaluation, dynamic binding
- Haskell
  - pure, static typing, “lazy” evaluation, lexical binding

Sketch history of FP

60 years in the making, or longer...

- Alonzo Church, λ-calculus, 1930s
- John McCarthy, Lisp, 1950s
- various Lisp-like languages
  - Scheme (1970s), Common Lisp (1980s), Clojure (2000s)
- statically typed FP languages
  - ML, SML, Miranda, OCaml, Haskell
- also: Erlang, Scala

Why FP?

- Better for big systems
  - type-safety
  - correctness
  - controlled interaction
  - expressive power
- Better for parallelism, multi-core, cache

mapfl in Haskell

```
mapfl _ [] = []
mapfl f (x:xs) = f x : mapfl f xs
```

mapfl in Scheme

```
(define (mapfl fun lst)
  (if (pair? lst)
      (cons
        (fun (car lst))
        (mapfl fun (cdr lst)))
      '() ))
```

Scala

- Developed by Martin Odersky from 2001
- “Multi-paradigm”, but strong support for FP
- Java-like syntax
- JVM

Clojure

- Developed by Rich Hickey, 2007
- Lisp-like syntax
- JVM
  - Clojurescript $\longrightarrow$ Javascript
- Strong emphasis on
  - immutability
  - concurrency
  - persistent data structures

Haskell

- Started by committee, 1987/1990
  - Simon Peyton-Jones, Phil Wadler, John Hughes, …
- Pure — "referential transparency"
- Strong statically typed
- Non-strict ("lazy") evaluation
- GHC compiler: multi-target + Javascript
- Hugs, Yhc: byte-code

Worth mentioning

- Common Lisp
- Scheme (Guile, Racket)
- OCaml
- Clean
- Mercury
- Erlang
- F#

FP at work

- Haskell
  - Darcs distributed version-control system
  - Xmonad window manager
  - Facebook anti-spam framework
  - GHC
- Clojure
  - Australia Post, Silverpond, Thoughtworks, Zendesk, Walmart Labs, eBay, Facebook
- Scala
  - REA Group
- Mercury
  - YesLogic

FP community in Melbourne

All on Meetup:

- Melbourne Functional User Group, MFUG
- Melbourne Haskell User Group, MHUG
- clj-melb (Clojure)
- Melbourne Scala User Group, MSUG

Recent Compose Melbourne FP event, videos available.

- `http://www.composeconference.org/`
- videos: search for "compose melbourne" on Youtube

FP and FOSS

- Haskell
  - GHC — BSD 3-clause
  - Hugs — BSD
  - Haskell Platform — BSD
- Clojure — Eclipse Public Licence
- Scala — BSD 3-clause
- Scheme — Guile, LGPL; Racket, LGPL
- Mercury — GPL/LGPL
- many others

Characteristics of Haskell

- concise, clean notation
- strong static typing
- pattern matching
- currying
- lazy evaluation
- “no side-effects”
- monads & IO
- compiled — good performance achievable

Haskell's type system

- strong static typing
- type inference
- polymorphism
- abstract datatypes
- typeclasses

Lazy evaluation

- evaluation “by need”
- not unlike Unix pipelines
- allows “infinite” data structures, `[1..]`
- strict evaluation possible

Control structures by lazy evaluation

If-Then-Else as an ordinary function:

```
ite :: Bool -> a -> a -> a
ite True t _ = t
ite False _ e = e
```

Maybe monad and failure

```
mb_sqrt x
  | x >= 0 = Just (sqrt x)
  | otherwise = Nothing

mb_rec x
  | x == 0 = Nothing
  | otherwise = Just (1/x)

mb_inc x = Just (x+1)

mb_isr x =
  mb_rec x >>= mb_sqrt >>= mb_inc
```

IO monad and IO actions

```
main = do
  putStrLn "What is your name?"
  name <- getLine
  print ("Hello " ++ name)
```

Lambda expressions & currying

```
threeone x = 3 * x + 1
threeone = \x -> 3 * x + 1
tom = map (\x -> 3 * x + 1)
```

Tail-call optimization

Straight recursive version:

```
rfac 0 = 1
rfac n = n * rfac (n-1)
```

Tail-recursive version:

```
tfac n = tfac' n 1
tfac' 0 p = p
tfac' n p = tfac' (n-1) (n*p)
```

Haskell downsides

- Lazy evaluation:
  - run-time overhead
  - hard to predict resource usage
  - (used to make debugging difficult)
  - solution: judicious use of strictness and compiler optimizations
- Sometimes confusing error messages
- Monads don't compose well

Summary

- FP context and history
- FP advantages and characteristics
- FP FOSS implementations

Work in progress:

`https://github.com/LJKitchen/ljk-luv-fp-foss`

Produced using the LaTeX Beamer package, along with other free-software programs.