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

Here is the material used in my talk, along with additional commentary.

Notes

An introduction to functional programming, historical, philosophical, and practical, linking up with FOSS, Free and Open-Source Software.

1 The Four Programming Paradigms

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

Notes

To see how functional programming (FP) fits into the overall scheme of programming languages, I'll introduce the concept of *The Four Programming Paradigms*, which you might already have encountered. Under this concept, there are four main ways of programming:

Imperative Programming A program is thought of as a sequence of instructions to the computer to do things, typically to change (mutate)

the data stored in variables by assignment (or to do input and output). Hence, *Imperative*. Typically, the data is organized into data structures, like records and arrays, and the instructions are organized into what are variously called subroutines, functions, or procedures (with some mechanism for passing arguments). Hence, this is sometimes also referred to as *Procedural Programming*. There are also control structures, like conditionals and loops, to execute instructions selectively or repetitively (*iteration*).

Object-Oriented Programming A running program is thought of as a collection of *objects*, which bundle data and procedures (usually called *methods*). The program runs by the objects sending messages to each other to invoke methods. Aside from this object-oriented organizing principle, an object-oriented program works much like an imperative program, by mutating the state of objects, with conditionals and loops.

Logic Programming A program is thought of as collection of *facts* and inference *rules* in some formal logic notation. The program runs essentially by (multi-step) inferences from the facts using the rules, to produce new (useful) facts.

Functional Programming A program is thought of as a collection of *functions* (sometimes confusingly called *procedures*). The program runs by functions invoking functions, often recursively. The emphasis is on the values computed by the functions, not on any side-effects of mutating variables. In some functional programming languages such side-effects are expressly forbidden (these are called *pure*); in others (*impure*), side-effects are allowed, but are generally discouraged. Characteristic of functional programming languages is that functions are "first class", that is, they can be treated much as data in imperative languages: they can be passed as arguments to functions, computed and returned as results from functions, and stored into data-structures. This leads to very powerful programming techniques. Also characteristic of functional programming languages is the use of recursion instead of iteration. Even when they provide iterative constructs, they are usually just syntactic sugar for some underlying recursive schema.

As you can see from the various programming languages mentioned in the slide, these Four Paradigms go back to the early, sometimes very early, days of modern computing. The years mentioned are only indicative. A single year is typically the year in which the language was first released, with a working compiler or interpreter. Where there are two years, like 1958/1962

for Lisp, the first year is when there was a substantial proposal or design for the language, and the second is the year of first release of a working compiler or interpreter.

Obviously, there are no sharp boundaries between these paradigms, and the languages that claim them. It's largely a matter of emphasis. Even a low-level, notionally imperative language like C provides some limited capabilities for doing functional programming. The first implementation of what we would call logic programming was done as an implementation of Planner embedded inside Lisp, an impure functional language which also supports imperative constructs. The pure functional language Haskell provides constructs which permit programming in an imperative style where appropriate (although these constructs are ultimately defined in pure functional terms). There's interest in adapting "advanced" functional techniques like monads to languages like Javascript. These cross-over examples can be multiplied almost endlessly.

Even though there is a lot of cross-over, The Four Paradigms provide a useful frame of reference, and most programming languages can be seen as fitting into one or other of these paradigms (with varying strictness), or of providing capabilities from the paradigms.

As mentioned, imperative and object-oriented programming typically use iteration (loops), while functional and logic programming typically use recursion. While Algol'60 is notionally an imperative language, I point out as something of an historical anecdote that in the Algol'60 report (the defining document for the language), iteration by the `while` loop is actually defined in terms of recursion. In a way, this is not surprising, since one of the co-authors of the Algol'60 report was John McCarthy, who was also creator of the functional language Lisp.

Notes

Functional programming and logic programming are often collectively referred to as *declarative programming*, since the emphasis is on declaring the functional or logical relationships in the program, and not so much on the step-by-step execution of the program through changing state.

2 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

Notes

Stuff.

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

3 History of FP

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

Notes

Stuff.

4 Why FP?

Why FP?

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

5 Simple examples

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)))
      '() ))
```

6 Quick sample tour of FP Zoo

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

7 FP and FOSS

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

8 Haskell

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.