An introduction to functional programming

Rolfe Bozier

02-Sep-2014

This work is licensed under a Creative Commons Attribution 4.0 International License.

Agenda

- What is functional programming?
- Why write software this way?
- What are some functional languages?
- Some code examples

This work is licensed under a Creative Commons Attribution 4.0 International License.

What is functional programming?

- Writing software in the same way that one uses functions in a mathematical proof or derivation
  - origins back to the 1930s
  - Lambda Calculus - how can you manipulate or compose functions [y = f(x)] to get some kind of desired result
    - e.g. a proof, quicker solution, generalization etc
- General philosophy:
  - Describe what needs to be computed, not the how to do it
  - State-less programming

This work is licensed under a Creative Commons Attribution 4.0 International License.

Some features of functional languages

- Pure functions
  - functions take parameters, return a value only - no side-effects
  - no read/write to memory (state), no I/O, no communications outside
  - result is completely determined by the parameters
  - you always get the same result for a set of parameters
- Functions are first-class entities
  - Functions are data
  - You can pass them to and return them from functions
  - Partial function application, supply some of the parameters (but not all) – and get a new function
  - You can compose functions to get new ones
  - Closures - functions with a bit of data context
- Recursion is preferred over iteration
  - Inductive algorithm definitions fit naturally
  - Tail recursion is used elimination to avoid running out of stack space

This work is licensed under a Creative Commons Attribution 4.0 International License.

Some features of functional languages

- Lazy evaluation
  - Only compute what you need to get the result
  - For example, how many elements are in this list?
    - [3.45, 22+x, 17/0, 0, -12345*3]
- Immutable variables and values
  - Variables are placeholders for values or functions
  - Operating on a value always produces a new result
- No “hidden” state information
  - Or at least, minimise / control it
  - Programs with *no* state are not very interesting…
- Pattern matching
  - Like a switch, but on steroids
- List comprehensions
  - List generators
  - apply functions to lists (e.g. map, fold, reduce, ...)
- Complex type systems, e.g.
  - Variants: a type composed of several other types
  - Type classes: a set of types matching some specific criteria, e.g. A == B -> Bool

This work is licensed under a Creative Commons Attribution 4.0 International License.

Why write functional programs?

- Lack/control of side-effects seems like an obvious benefit:
  - code elimination, reordering
  - thread safety
  - caching (memoization) is trivial
  - easy parallelization...?
- Seems easier to reason about correctness
- Curry-Howard Isomorphism
  - proof systems based on logical/mathematical rules are equivalent to computational models in lambda calculus
  - "a proof is a program, the formula it proves is a type for the program"
  - there is a lot of software that does program proving/verification written in functional languages: Coq, Coccinelle, Frama-C, CIL
- Functional languages were expected to deliver automatic parallelisation of code "for free“ - but this didn't happen
  - immutable values tend to mean a lot more churn on the GC
  - declarative style tends to hide the implementation, this makes it difficult to determine the right level of granularity
  - performance tends to be limited by cache/memory access – this seems to be better handled using careful data mutation

This work is licensed under a Creative Commons Attribution 4.0 International License.

Some functional languages

- There is no strict definition, so the spectrum ranges from “pure” functional languages through multi-paradigm ones, to languages with a bit of functional behaviour tacked on
- The following languages are relatively popular; they are ordered from “more functional” to “less functional”…
- Haskell:
  - pure functional language, committee-designed, academic
  - functions are pure, so no side-effects
  - but side-effects are needed so there is a mechanism to log them and "crystallise" them after the function has returned
  - lazy evaluation, immutable values etc.
  - compiled to executable code
  - strong typing, static typing

This work is licensed under a Creative Commons Attribution 4.0 International License.

Some functional languages

- Clojure
  - modern descendent of Lisp, invented by one person
  - immutable variables, lazy sequences
  - concurrent
  - compiles to JVM, CLR
  - dynamic typing
- Scala
  - multi-paradigm
  - immutable values, lazy evaluation
  - concurrent - uses Java concurrent APIs
  - compiles to JVM
  - strongly typing, static typing, type inference
- Erlang
  - multi-paradigm language, designed by Ericsson
  - immutable values (by default), eager evaluation (by default)
  - concurrent - message passing with no shared state
  - high reliability
  - eager evaluation
  - compiles to bytecode

This work is licensed under a Creative Commons Attribution 4.0 International License.

Some functional languages

- OCaml
  - descendent of ML, designed by INRIA, multi-paradigm
  - immutable values (by default), eager evaluation (by default)
  - strong typing, static typing, type inference
  - emphasis on performance
  - concurrent - sort of, uses time-slicing
  - compiles to executable or bytecode
- F#
  - inherited from others including OCaml, designed by Microsoft
  - multi-paradigm with emphasis on functional aspects
  - compiles to CLR
  - strong typing, static typing, type inference
- Others with functional aspects
  - JavaScript
  - Perl (list comprehensions, anon functions)
  - Python (generators, list comprehension, anon functions, functools module)
  - Ruby

This work is licensed under a Creative Commons Attribution 4.0 International License.

Some examples (in Ocaml)

```
open Printf

(* recursive function to return the nth Fibonacci number *)
let rec fib n =
    match n with
    | (0|1) -> n
    | _     -> fib(n-2) + fib(n-1)

(* partial function application *)
let pr_d = printf "%d\n"

(* print the result of the function *)
pr_d (fib 5)
```

This work is licensed under a Creative Commons Attribution 4.0 International License.

Some examples

```
(* create a new type which is either nothing or an integer *)
type opt_int = None | Some of int

(* what value were we given *)
let ft x =
    match x with
    | None -> "none"
    | Some 0 -> "nothing"
    | Some _ -> "something"

printf "%s\n" (ft None)
printf "%s\n" (ft (Some 0))
printf "%s\n" (ft (Some 34))

.... Gives:

          none
          nothing
          something
```

This work is licensed under a Creative Commons Attribution 4.0 International License.

Some examples

```
(* a function to return the last 2 objects in a list *)
let rec get_last_two x =
    match x with
    | [] -> None
    | [_] -> None
    | [a; b] -> Some (a, b)
    | _ :: tail -> get_last_two tail

get_last_two [1; 2; 3; 4; 5; 6; 7]

.... gives: Some (6, 7)

get_last_two [”a”; ”b”; ”c”; ”d”]

.... gives: Some (”c”, ”d”)
```

This work is licensed under a Creative Commons Attribution 4.0 International License.

Some examples

```
(* function to square an integer *)
let square x = x * x

(* function to calculate the factorial of a number *)
let rec fact x =
    if x <= 1 then 1 else x * fact (x - 1)

(* function to compose two functions *)
let compose f g =
    fun x -> f (g x)

(* create a square of factorial function *)
let square_of_fact = compose square fact

square_of_fact 5

.... Gives:

          14400
```

This work is licensed under a Creative Commons Attribution 4.0 International License.