Why Functional Programming Matters - Summary

Binbin Weng (binbinw2)

In this article, the author introduced us the two key things that functional programming has, which make functional programming very powerful compared with its conventional counterparts.

Functional Programming gets its name because functions can perform as arguments for other functions.

Functional programming has the following specific characteristics and advantages:

1. In functional programs, once a variable is given a value, it cannot be changed. The great advantage brought by this is that it eliminates a major source of bugs, makes the order of execution irrelevant and relieves the programmer of the burden of prescribing the control flow.
2. Functional programs are more concise than its conventional counterpart, which makes the programmers more productive. (Conventional programs consist of 90% assignment statements, but functional programs doesn't have assignment statements.)

But these characteristics and advantages cannot convince programmers of the power of functional programming. We need explain these characteristics and advantages more deeply to get the sense why function programming so different from conventional programming.

As we all know the success of programming is modularity. The advantages are:

1. small modules are easy to code
2. modules can be reused
3. modules can be tested independently

Modular design allows programmers to divide problems into subproblems, then solve the subproblems, and finally combine the solutions together. But how to combine the solutions effects greatly on the how a problem is modularized.

Functional programming provides two kinds of glues which make it so different from conventional programming:

1. higher-order functions: to glue functions together easily

Functional languages treat functions as first-class values which means that like other value, a function can be passed as a parameter and returned as a result. This provides a flexible way to compose functions. Functions that take other functions as parameters or that return functions as results are called higher-order functions. https://zhuanlan.zhihu.com/p/46448517

```
--data type of a list
```

```
data Listof a = Nil | Cons a (Listof a)
  deriving (Show)

-- the sum function is to calculate the sum of a list
-- can be modularized by gluing together a general recursive
-- pattern and "+"
sum' Nil = 0
sum' (Cons n (list)) = n + sum' list

-- foldr is a higher-order function because it takes f, another function,
-- as its parameter
(foldr f x) Nil = x
(foldr f x) (Cons a l) = f a ((foldr f x) l)
-- then sum can be rewritten as
sum'' = foldr (+) 0
-- foldr can be reused to create other functions on lists
product' = foldr (*) 1

-- double all elements in a list
doubleall = foldr' doubleandcons Nil
doubleandcons n list = (Cons (2*n) list)
-- split doubleandcons down
double' n = 2*n
fandcons f el list = Cons (f el) list
map' f = foldr (Cons . f) Nil
doubleall' = map' double'
```

The above gives examples of construction functions on lists using higher-order functions. Some functions can be broken down to simple ones, and some can be written as general ones which can be reused again and again.

With the high-order functions and its simple arguments, programmers can write many similar functions on lists easily.

Functional programming allows functions to be expressed as combination of higher-order functions and simple functions, which is *not* allowed in conventional programming. Once a new datatype is defined, higher-order functions (such as `foldr` and `map` above) should be defined, which will make the manipulation of the datatype easy.

2. Lazy evaluation: to glue programs together easily

If `f` and `g` are two programs, and `h = (g . f)` is a program that is composed of `g` and `f`, different from conventional programming, when we run `h` , `g` and `f` are run together in strict synchronization. `f` is started only when `g` tried to read inputs from `f`, and as long as it delivers outputs to `g`, it will terminate. In addition, if `g` terminates before it reads all `f`'s outputs, `f` will also terminate. This benefits for that `f` can even be a nonterminating program.

The lazy evaluation runs `f` as little as possible, which makes it practical to modularize a program as a generator that constructs a large number or even infinite number of possible answers, and a selector that chooses the appropriate one. In this case, `f` can be the generator, and `g` can be the selector.

Examples:

```
-------------- Newton-Raphson Square Roots---------------------
-- Use the formula ai+1 = (ai+n/ai)/2 to estimate the square root of n.
next n x = (x + n/x)/2
repeat' f a = Cons a (repeat' f (f a))
-- repeat' (next n) a0 will be a function with infinite outputs
-- but it doesn't matter, it is just a potential
-- any number of approximations can be computed if required
-- The following within is taking a tolerance and a list of approximations
within eps (Cons a (Cons b rest)) =
    if abs(a-b) <= eps then b
    else within eps (Cons b rest)
-- final Newton-Raphson Square Root function
sqrt' a0 eps n = within eps (repeat' (next n) a0)
-- in this case, (repeat (next n) a0) is the generator,
-- (within eps) is the selector

------------ Differentiate a function at a point ------------------
-- Given a real-valued function f of one real argument, and a point x,
-- estimate the slope of the function f at the point x.
easydiff f x h = (f (h + x) - f x)/h
differentiate h0 f x = map' (easydiff f x) (repeat' halve h0)
halve x = x/2
newdiff eps h0 f x = within eps (differentiate h0 f x)
-- in this case, (differentiate h0 f x) is the generator,
-- (within eps) is the selector

--------------- Numerical Integration -----------------------
-- Given a real-valued function f of one real argument, and two points,
-- a and b, estimated the area under the curve that f describes between
-- the points
easyintegrate f a b = (f a + f b) * (b - a)/2
-- works when a and b are close enough
-- better method:
integrate f a b = Cons (easyintegrate f a b)
                       (map' addpair (zip2 (integrate f a mid)
                                           (integrate f mid b)))
                  where mid = (a + b)/2
zip2 (Cons a s) (Cons b t) = Cons (a, b) (zip2 s t)
addpair (Cons (a,b) rest) = Cons (a+b) (addpair rest)
newintegrate eps f a b = within eps (integrate f a b)
-- in this case, (integrate f a b) is the generator,
-- (within eps) is the selector
```

These examples give us much sense that even though we may not know what the result is, but we still can create a function that gives us a large number or even infinite number of potential answers and create a function to select the best answer with a standard. As long as the answer reaches our standard, no more calculation is needed.

An Example from Artificial Intelligence:

```
-- alpha-beta heuristic: an algorithm works for estimating
-- how good a position a game-player is in
-- what moves can be made from a position (in one move)
-- here moves is not defined, because it depends on how the rules of the game
is
moves :: position -> listof position
-- reptree here is a higher-order function analogous to repeat' above
reptree f a = Node a (map' (reptree f) (f a))
-- a gametree is a tree with current position p, what the following steps
will be
gametree p = reptree moves p
-- static is a function to calculation the score of a position, which is not
-- defined with the same reason as moves. But we can assume the larger, the
better.
static :: position -> number
-- thus we can convert the game tree of positions to a tree of scores
-- the following functions are based on the assumption that
-- when it is the computer's turn, it will move to the position with the
highest score
-- when it is the opponent's turn, the opponent will move to the position
with
-- its highest score, but the computer's lowest score
maximize (Node n Nil) = n -- base case
maximize (Node n sub) = max (map minimize sub)
minimize (Node n Nil) = n
minimize (Node n sub) = min (map maximize sub)

-- given a position and return its value
evaluate = maximize . maptree static . gametree
-- But in reality, it's unrealistic to evaluate the whole of the game tree
-- even it is a finite tree
-- function to cut off nodes which are over n steps from current position
prune 0 (Node a x) = Node a Nil
prune (n + 1) (Node a x) = Node a (map (prune n) x)
-- look 5 steps ahead
newevaluate = maximize . maptree static . prune 5 . gametree
```

Higher-order functions including `reptree` and `maptree` allows us to construct and manipulate game trees with ease, Lazy evaluation permits us to modularize evaluate in this way. `gametree` is a potentially infinite tree, if we don't have lazy evaluation, the program would never terminate and the whole program cannot be written in this way.

Conclusion

Modularity is the key to successful programming. Functional programming provides two kinds of glues, higher-order functions and lazy evaluation. With these glues, smaller, simpler and more general modules can be reused widely, and make the programs easier to write, because it removes some restrictions and gives programmers more possibilities.

The link for the article, *Why Functional Programming Matters*, is
https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf