Why Functional Programming Matters

Ted Neward

Neward & Associates

http://www.tedneward.com | ted@tedneward.com

History

In the beginning, God created objects....

History

Functional programming is a different way of thinking about modularizing applications

And, admittedly at times it is a different way of thinking that runs entirely contradictory to the way that object-oriented programmers think

Functional Programming

What's it mean, exactly?

Concepts

Functional languages

- functional as in mathematics' notion of function
  - **for every x, there is a corresponding value y**
  - **this implies no side effects**
- not imperative statements, but expressions
  - **"x = x+1" is not increment... it's impossible**
  - **this implies expressions can be substituted**
  - **... or executed independently (parallellism)**
- spectrum of "functional-ness", known as purity
  - **"pure" functional languages allow for no side effects**
  - **"impure" functional languages allow for side effects**

Concepts

What's wrong with imperative statements?

- dependences on mutable state
- compiler out-of-order rewrites
- difficulties reasoning about the code
- concurrency planning/programming

Concepts

Some basic functional concepts

- functions as first-class values
- currying, partial-application of functions
- strongly-typed, type-inferenced
- immutable values
- recursion
- expressions-not-statements
- tuples, lists
- pattern-matching
- laziness/deferred execution

Concepts

Functions as first-class values

- think about common operations--if we could vary the actual operation itself as a parameter, how much work could be saved?

  example: you need to iterate through a collection to...
  ... and each time you write it as a "for" loop, you're violating DRY

- this enables the use of functions as "higher-order" functions

  - **"take this function, and execute inside your context"**
  - **similar in some ways to a callback, but with clearer semantics**
  - **in a lot of ways, this is Inversion-of-Control all over again**

Concepts

Higher-order functions

```
let numbers = [1, 2, 3, 4, 5]
let squares = numbers.map((num) -> num * num);
  // squares = [1, 4, 9, 16, 25]
```

Concepts

Partial application

- providing some of the parameters (but not all) to a function and capturing that as a function to be called

Currying

- it turns out (thank you Alonzo Church!) that all functions can be reduced to functions taking one parameter and either yielding a result or another function
- this permits easy "pipelining" and composition of functions in a highly reusable manner (at the micro level)

Concepts

Partial application

```
let add x y = x + y

let five = add 2 3  // = 5
let addBy2 = add 2  // = (anonymous fn) = 2 + y
let six = addBy2 addBy2 2 // = 6
```

Concepts

Function composition

- In functional languages, then, we achieve reuse through the composition/combination of functional parts into larger functions
- By doing so, we "build up" larger more complex functions
- When combining several in a row using currying, this is also called "pipelining"

Concepts

Strongly-typed

the dynamic language community will have you believe that it's better to write unit tests by hand than to have a system that can do common-sense checking for you

Type-inferenced

why do I have to be explicit to the language, when it can figure out what I'm trying to do and when?

Concepts

Immutable values

once bound, a binding remains constant throughout its lifetime, and thus offers no opportunity for confusion

Recursion

immutable values doesn't mean no state changes

instead, hold state on the stack, not in mutable memory

Concepts

Expressions-not-statements

- this is an outgrowth of the functions-as-first-class-citizens idea: if functions yield values, what is the practical difference between a keyword and a function?
- even C++ tried to make user-defined elements look and feel like built-in constructs and vice versa
- if we're really good about this, developers can create new "language" primitives and nobody will know the difference

Concepts

Tuples, lists

- "bundles" of data in different directions
- tuples give developers a "lightweight" object that needn't be named or otherwise formalized

Concepts

Pattern-matching

- switch/case is to pattern-matching as my kid's soccer team is to Arsenal or Manchester United
- pattern-matching also encourages "destructuring" of data when necessary/desired

Concepts

Laziness

- object-oriented laziness has nothing on functional laziness
- don't compute anything until absolutely necessary (but make sure to maintain the dependency graph so everything is there when needed)
- laziness is highly encouraged/permissible in pure FP
- just to be fair, laziness is highly desirable inside the process, not so across processes unless carefully managed

Concepts

Sequences

- lots of things can be seen as sequences

  characters in a string
  fields in a record
  records in a database
  files in a directory
  algorithmic calculations (factorial, fibonacci, ...)
  lines in a file

- sequences and collections have a deep relationship

  in many ways, this is the gateway to FP ideas/concepts

Application

- Continuations
  - instead of wiring steps together explicitly, do it implicitly by passing in the next "thing" to do as a function
- Concurrency
  - instead of locking explicitly, allow the underlying language library or runtime to manage the physical details of the parallelization, or (better yet) avoid the need entirely
- Abstractions
  - Parsing, for example, is made easier because the functional approach better matches what parsers do
  - How many tortured object designs must we build before we acknowledge that objects don't fit everything we build?

Summary

Functional programming is not going to replace object-orientation, but supplement it

- objects didn't replace procedural programming, but built on top of it and incorporated it
- most new FP languages are functional/object hybrids, not pure FP languages

Functional programming represents a new tool in your toolbox, not wholesale rejection of prior art