Foundations of Functional Programming

May 6, 2014

Goals

What I hope you will get from this presentation:

- A look into functional programming's origins
- A language-neutral introduction to important FP concepts
- An idea of which problems are good candidates for FP
- Curiosity about some ideas which underpin FP
- A pragmatic plan for introducing FP

Not goals ☺

- *"Functional programming will save the world!"*
- *"Functional programming is better than …"*

Who am I

- Brian Hartin
  - brian.hartin@gmail.com
  - twitter: @bjhartin
- Developer at Iowa Student Loan Liquidity Corporation
- Program mostly in Groovy, Java, Scala and Ruby

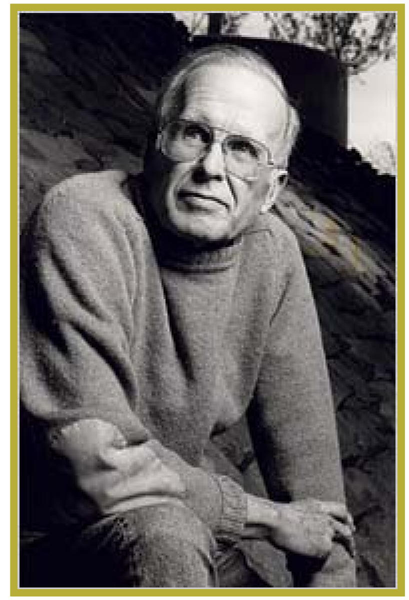
A Leaky Abstraction?

- Most popular programming languages are loose representations of the computer architecture
  - Includes imperative languages such as C, C++, Java, etc.
  - Building blocks represent CPU and memory activities

variables	int i;	memory locations
control statements	if / while / for	CPU 'test' and 'jump' instructions
assignment statements	x = 10; a = b;	fetching, storing instructions
expressions	3 + (4 * x)	fetching and arithmetic instructions

Von Neumann Languages

- So designated in 1977 Turing-award paper: *“Can Programming be Liberated from its Von Neumann Style?”*
  - Inspired a lot of research into FP
- These languages focus on *telling the computer to do something*.
  - We must think carefully about time and state, i.e. *sequencing*, *sharing* and *changing*
  - Programs divided into *statements* and *expressions*
  - Different from *thinking about computation*
- Programs don’t have many useful mathematical properties.
  - Difficult to formally reason about them, or prove things



John Backus, IBM

Von Neumann Languages, cont.

- Backus proposed a language in which:
  - Computations are mathematical functions
  - Everything is an expression (no statements)
  - Shared state is prohibited
  - Mutable state is prohibited
  - Programs had useful mathematical properties
  - provably correct
  - combined via composition
  - automatic program simplification/reduction

Complexity

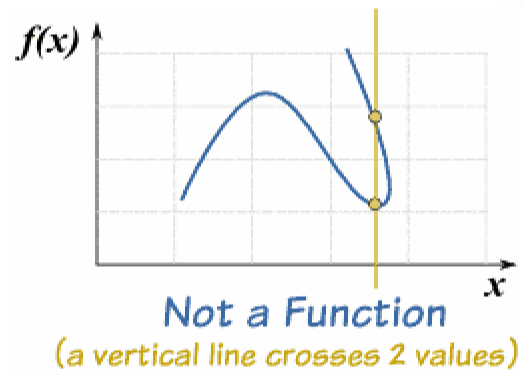
- Consider the following code

```
// Need example here which shows:

-  We can’t fully comprehend what this code is doing without
   seeing more of the program
-  We can’t safely change the order of a computation
-  We can’t parallelize it safely
-  We’re not sure where the values are coming from
-  Spring context mess
```

Functional Languages

- Strive to allow most computations to be defined as *mathematical* functions
  - deterministic
  - idempotent
  - side-effect free



- Ideals (not necessarily guaranteed)
  - Programs composed of *referentially transparent* expressions
  - Functions are *pure* and *first class*
  - Some functions are *higher order*
  - Values are immutable
  - Logic expressed declaratively (often recursively)
  - Reduction or elimination of *state,* especially global or shared
  - Reduction in the *number* of concepts, increase in their *power*
  - Hand sequencing replaced by *composition* (dependency graph)

Functional Languages, cont.

"In functional programming, programs are executed by evaluating *expressions*, in contrast with imperative programming where programs are composed of *statements* which change global *state* when executed. "

-- http://www.haskell.org/haskellwiki/Functional_programming

Some Additional FP Idioms

- List comprehensions / maps, filters, folds
- Pattern matching
- Lazy evaluation
- Partial application and currying
- Composition
- Range types

Non-Pure Functional Programming

- Some languages support some FP features, but not all
  - They may allow shared state, mutability and side effects
  - They support features such as higher-order functions, list comprehensions, etc.
  - Ruby, Groovy, Scala and Java 8 all fall into this category
- Can write nearly pure functional code in any of these languages
- Or, you can simply leverage certain features as beneficial
- Part of being an effective functional programmer is understanding what benefits come from purity, and making smart compromises

Expressions vs. Statements

- Functional languages emphasize *expressions* over *statements*

Expressions:

- Express (return) a value — 1 + 2 // of type int
- Consist of values, operators and other expressions — 1 + x + foo(1)
- Can be used where a value of the same type is expected — bar(int x, int y) / bar(3, **(x + foo(1))**)
- Can be composed into more complex expressions — f(g(), h(1, “a”), j())

Expressions vs. Statements

Statements

- Do not return a value — `Thread.sleep(1000)`
- Are illegal in some places — `return (int y = 2)`
- Exist *solely* for the purpose of their side effect — `System.out.println("Hi")`
- Typically change some global state of the program or the system — `void save()`
- Cannot be substituted for values — `foo((int y = 2), 2)`

Referential Transparency

- An expression is *referentially transparent* if, once computed, its value can safely be substituted for the expression.

```
// Trivial case
int x = 1 + 2;  // is 3
…
// could replace (1 + 2) with 3
int y = 3 * 5 – (1 + 2);

// Non-trivial case
int amount = order.total() * getDiscount() // is $34.00
…
// Can we safely replace the second term with $34.00?
int amount2 = dailyTotal() + (order.total() * getDiscount())
```

- Cannot safely replace occurrences of an expression if its computation has side effects or hidden inputs

Referential Transparency, cont.

- In mathematics, all functions have this property
- Any referentially transparent expression is *deterministic*
- Any referentially transparent expression is *idempotent*
- Allows humans and compilers to reason more effectively about program behavior
- Enables features such as
  - Memoizing
  - Parallelization
  - Lazy Evaluation
  - Common sub-expression elimination (simplification)
- *How to guarantee referential transparency?*

Immutability

- A value cannot be changed, once assigned
  - Instead, new values are created
- Required for referential transparency – mutable types make expressions *referentially opaque*
  - Impossible / very hard to rule out side effects
  - **Be aware of this if you 'sprinkle' functional programming into OO**
- Allows deconstruction of objects

Immutability, cont.

- Some problems go away entirely if values (objects) are immutable
- A good strategy is to push mutability to the boundary of a module
    - An API that accepts mutable types only at entry/exit points
    - Allows advantages of pure FP within the module
    - Good for integrating with legacy code, non-FP libraries and frameworks

FP Big Idea ™: Making entire <u>categories</u> of errors impossible to express

Declarative Programming

- The opposite of imperative programming
- Instead of telling the computer/language *what to do*, we tell it what we *want*
- Easier to reason about *correctness*, perhaps harder to reason about *performance*
  - SQL is an example – the DB plans the execution

First Class Functions

- Functions can be treated as values
- Can be given names like other values, x = …

Scala

```
val doubleIt = (x: Int) => x * 2
```

Groovy

```
def doubleIt = {int x -> x * 2}
```

Java 8

```
Function<Integer, Integer> doubleIt = (x) -> {return x * 2;};
```

Higher Order Functions

- Can be passed as parameters
- Can be returned

Scala – Higher Order Functions

```
def isEven(n: Int) = {n % 2 == 0}

// Accepts a function
def checkNumber(n: Int, check: Int => Boolean) = {check(n)}

// Returns a function
def multiplyBy(m: Int) = (n:Int) => {n * m}

checkNumber(2, isEven)    // true
multiplyBy(5)(6)          // 30
```

Higher Order Functions, cont.

- A form of inversion of control
- Great for when you want to ‘wrap’ behavior *around* other behavior, e.g.
  - Closing files or database connections

Scala

```
def doWithWriter(filename: String, f: PrintWriter => Any) = {
  val writer = new PrintWriter(new File(filename))
  try {
    f(writer)    //  f doesn’t open or close the file
  } finally {
    writer.close
  }
}
```

- Also useful for other cross-cutting concerns like transactions, benchmarking, logging, etc.

Higher Order Functions, cont.

- Goes hand-in-hand with first-class functions
- Valuable in processing collections (as we will see…)
- Many OO patterns overlap with higher order functions
  - Strategy
  - Visitor
  - Command
  - Anonymous inner classes
  - Template method
  - Iterator

FP Big Idea ™: Simplicity through unification: statements vs. expressions, functions vs. values

Anonymous Functions (Lambda Expressions)

- Functions can be defined and called without a name

Scala

```
checkNumber(2, (number:Int) => {number % 2 != 0})
checkNumber(2, {_ % 2 != 0})    // Type infer. allows this
```

Groovy

```
checkNumber(2, {n -> n % 2 != 0})
checkNumber(2, {it % 2 != 0})    // Can use ‘it’ when one arg
```

- Often used for ad-hoc, simple computations, which don’t deserve a name
  - Similar to some anonymous classes (e.g., comparators)

Closures

- Lambdas are often (lexical) *closures*
- Functions that retain bindings to their enclosing scope
- Technically, anonymous functions don't have to be closures
  - But usually are

Groovy

```
def callWith3(Closure f) {f(3)}

def closureTest() {
  int x = 5
  callWith3({y -> x + y})       // 8, because x was bound to 5
}
```

- Note that x is out of scope in callWith3
- Closures + mutable types = possibility of surprising side effects

List Comprehensions

- A way of performing arbitrary operations on list-like structures
  - Largely replaces for and while loops
  - Comes from ‘set-builder’ notation in mathematics:

$$S = \{ 2 \cdot x \mid x \in \mathbb{N},\ x \leq 10 \}$$

  - Encourages us to think more about the results, not the steps
- Common to languages with functional programming support
- Built on top of higher-order functions
- A core set of list comprehensions are common to most FP languages

Filters & Maps

- Filter selects items by a predicate function

Scala

```
List(1, 2, 3, 4, 5).filter({_ % 2 == 0})  // List(2, 4)
```

Groovy

```
[1, 2, 3, 4, 5].findAll({it % 2 == 0})   // [2, 4]
```

- Map transforms items with a transformation function

Scala

```
List(1, 2, 3, 4, 5).map({_ * 2})        // List(2, 4, 6, 8, 10)
```

Groovy

```
[“a”, “b”].collect({it.toUpperCase()})  // [“A”, “B”]
```

Folds (Reductions)

- Reduces a list to a single item
- Common uses: totaling, averaging or other aggregation

Scala

```
List(1, 2, 3, 4, 5).foldLeft(0)((acc, n) => {acc + n}))  // 15
```

Groovy

```
[1, 2, 3, 4, 5].inject(0, {acc, n -> acc + n})           // 15
```

- The function argument takes an accumulator and the item
- Accepts an initial value for the accumulator

Others

- Many others

grouping	partitioning	map + filter
removing duplicates	flattening nested lists	concatenating
set arithmetic	sorting	transposing

- Can dramatically simplify code

FP Big Idea ™: Expressing what we want, not the steps to do it

Pattern Matching

- Allows us to specify behavior by matching values to patterns
- In simple case, similar to **if/else** or **switch**

Scala

```
val result = myValue match {
  case 1 => “one”
  case _ => “other”
}
```

- Patterns can be very concise

Scala

```
val result = myValue match {
  case s: String if s.startsWith(“f”) => “starts with f”
  case s: String => “other string, ” + s
  case i: Int => “an int”
  case _ => “other”
}
```

Pattern Matching, cont.

- More powerful with immutable types (Scala case classes)

Scala

```
case class Order(cust: Customer, lines: [OrderLine])
case class Customer(email: String, addr: Address,
    totalOrders: Int)
case class Address(line1: String, line2: String, city: String,
    state: String, zip: String)

val shippingChargeAdjustment = order match {
  case hawaiiOrder: Order(Customer(_, Address(_, _, _, “HI”,
_)), _, _) => 1.5
  case bigOrder: Order(_, l1:l2:l3:l4:l5:ls, _) => 0.9
  case goodCustomerOrder: Order(c, _, _) if c.totalOrders > 10
=> 0.8
  case _ => 1.0  // Normal order
}
```

Pattern Matching, cont.

Kinds of patterns

Type	Example
By value	case s : "my string"
By type	case s: String
By deconstruction	case c: Customer(_, Address(_, "Des Moines", _, _))
With alternatives	case 0 | 1 | 2
By deconstruction (lists)	case List() => // None case x :: List() => // One case x :: xs => // More than one
By pattern guard	case o if o.totalAmount > 3
With multiple bindings	case c: Customer(fname, lname, _, _, _, _) => lname + "," + fname
By exhaustion	case _ => // No other cases matched

Partial Application

- A function may be *partially applied* by providing only *some* of its arguments
- This *fixes* those arguments, producing a new function

Scala

```
def addEm(x: Int, y: Int) = {x + y}
def addFive = addEm(5, _:Int)          // Fixes the first arg
addFive(3)                             // 8
```

- Haskell is beautiful in this regard. Any function may be partially applied with no special syntax

Haskell

```
add x y = x + y                        // Function body
addFive = add 5
addFive 3                              // 8
```

Lazy Evaluation

The ‘Killer Feature’ of Func. Prog. (IMHO)

- Allows us to think about *computations,* instead of computers
- Gives us a way to write programs composed of subprograms
- We can more easily understand a part of a program without understanding the whole (**example**)
- Part of a historic trend toward a higher level of abstraction as hardware capabilities increase
- Can leverage some of these ideas in non-functional languages (**example**)

The Killer Feature, cont.

- Can more easily parallelize computations (**example**)
- Can memoize computations (**example**)
- Compilers can perform more kinds of optimizations
  - Reordering computations
  - Pulling things out of loops
  - Tail call optimization (appropriate here ???)
- Can use lazy evaluation
- Can deal with infinite ranges

Good Candidates for FP

Complexity

- Example: Shared state as complexity
- Example: Mutable state as complexity
- Example: Non-determinism as complexity
- Example: Side-effects as complexity
- Example: Sequencing as complexity

Java 8 Support for Func. Prog.

- Lambda expressions supported through *functional interfaces*
  - An interface with one abstract method, e.g. *Runnable* or *Comparator*
- lambdas are instances of an Object subtypes
  - Not necessarily well-behaved objects
- Are type compatible with functional interfaces if the signatures match
- *Method references* capture existing methods as lambdas
- Not *pure* functions – can be done with *void* methods, and with side effects

Java 8, cont.

- List comprehensions / operations supported via *streams*
  - Sequences which may or may not be finite
- Functional interfaces define the *type* of higher order function accepted
- Operations defined as intermediate (lazy) and terminal (eager)

Operation / example	Functional Interface / method	Evaluation style
filter	Predicate.test	lazy
map	Function.apply	lazy
sorted	Comparator.compare	lazy
reduce (fold)	BiFunction.apply	eager
forEach	Consumer.accept	eager

Applying Functional Concepts in Non-Functional Languages

- Can apply some of these concepts in non-functional languages
  - Immutability
  - Pure functions
    - Determinism
    - No side-effects
    - Idempotency

Haskell

- An purely-functional programming language
  - Introduced in 1987
  - All objects immutable
  - All functions pure
    - Models side effects (IO, etc.) as return values
- Almost the reference implementation of functional programming
- Very good way to learn functional programming fundamentals
- Often, ‘if it compiles, it works’

Haskell Curry
1900 – 1982
American
Mathematician
and Logician

Haskell, cont.

Haskell

```
isBig n = n > 100
f = isBig
f(101)                    -- First class functions         true

                          -- List comprehensions and
filter isBig [1, 3, 105] -- higher order functions and    [105]
map (\n -> n/2) [2, 4]    -- anonymous functions           [1, 2]

take m ys = case (m,ys) of        -- Pattern matching
            (0,_)    ->  []
            (_,[])   ->  []
            (n,x:xs) ->  x : take (n-1) xs

add x y = x + y                   -- All functions are curried
addFive = add 5                   -- and can be partially applied
addFive 3                         -- 8
```

Haskell, cont.

- Great, free online IDE: fpcomplete.com
  - Includes 'School of Haskell' tutorials
- Incredible free online book: learnyouahaskell.com
  - Strong introduction to FP concepts, including advanced ones
  - Even if learning Scala/Groovy/other, this is a great resource

Related, Interesting Topics

- Non-Von Neumann computer architectures
- Lambda calculus
- Applying functional thinking outside your program
- Logic programming

References

1. "Can Programming be Liberated from its Von Neumann Style?" (1977 Turing award paper/lecture by John Backus)
   - http://www.thocp.net/biographies/papers/backus_turingaward_lecture.pdf
2. "Functional Thinking" series by Neal Ford
   - http://nealford.com/functionalthinking.html
3. "Referential Transparency" – Wikipedia
   - http://en.wikipedia.org/wiki/Referential_transparency_(computer_science)
4. Learn You a Haskell for Great Good! (Miran Lipovača)
   - http://learnyouahaskell.com
5. Java 8 Lambda FAQ (Marice Naftalin)
   - http://www.lambdafaq.org
6. An Introduction to Functional Languages by Ken Sipe (two parts)
   - http://java.dzone.com/articles/introduction-functional
   - http://java.dzone.com/articles/introduction-functional-0

Thank you!