Foundations of Functional Programming

Brian Hartin
twitter @bjhartin
brian.hartin@gmail.com
CIJUG May 6 2014

Goals

What I hope you will get from this presentation:

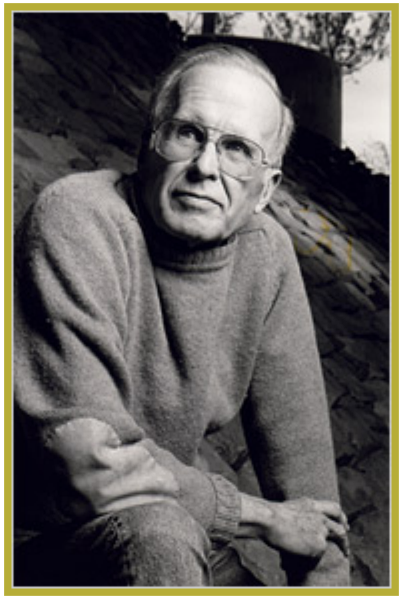
- A look into functional programming's origins
- A language-neutral introduction to important FP concepts
- Pragmatic ideas for introducing FP

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

Von Neumann Languages

- So designated in Turing-award paper: *"Can Programming be Liberated from its Von Neumann Style?"*
  - Inspired a lot of research into FP
- These languages mirror the Von Neumann architecture of the computer
  - Reflect CPU, memory and bus operations
- Programs don't have many useful mathematical properties.
  - Difficult to formally reason about them, or prove things



John Backus, IBM

A Leaky Abstraction?

- 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

- Imperative - “do this, then do that”
- Allows fine control over performance

Complexity

- Backus claimed that this leaky abstraction introduces complexity
- Instead of *defining* computations, we are forced to think about how to *get a computer to perform them*
  - Thinking about *sequence* of computations, instead of *dependency*
  - Thinking about *side effects*
    - The division between *statements* and *expressions*
  - Dealing with *non-determinism*
  - Carefully managing shared, mutable state

Our Example Project

- We'll examine these kinds of complexity in a simple example:
  - Service / domain layer classes for e-commerce site
  - Order processing tasks
    - Receive a file of orders, represented as JSON
    - Read the file
    - Parse the orders
    - Validate them
    - Save the valid ones
    - Return a summary
  - Very high tech - you must all now sign a non-disclosure agreement ☺

State as Complexity

```
Float computeBill(Order order) {
    Float shippingCharge = order.computeShippingCharge();
    Float discount = OrderService.computeDiscount(order);
    order.save();
    discount = OrderService.computeDiscount(order);
    return order.total() * discount + shippingCharge;
}
```

- Can we swap the shipping charge and discount computations?
  - Can we do them in parallel?
- What effects does this method have on the system as a whole?
  - Does it change other objects in memory?
- If we were trying to reproduce a problem, what state must be established prior to this method?
- Can the discount be cached, and the second computation discarded?

Complexity, cont.

- Without reading much further into the code...
  - We don't know what all of their inputs
  - We don't know what all of the side-effects are
  - We don't know if they will return the same value if executed twice with the same inputs
  - We don't know what data they share
- In short, these methods are not well behaved *functions*

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
    - could be combined via composition
    - automatic program simplification/reduction
- This inspired a lot of research into FP

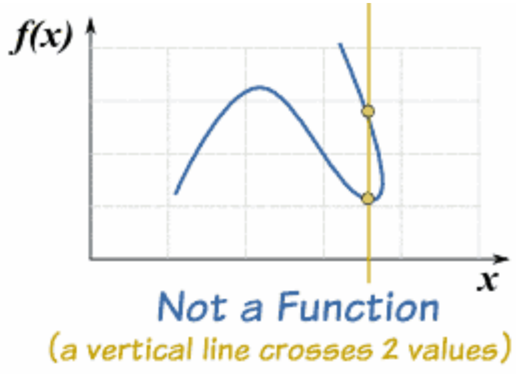
Functional Languages

"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

Functional Languages

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



- A key benefit is simplicity
  - Shared state, mutability and sequencing add complexity
  - Can be thought of as optimization techniques
    - Like many optimizations, they make the logic less clear

Pure Functional Languages

- 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)
- ‘Pure’ is an abstraction
  - Leaks a bit if you think of memory/cpu usage as mutable shared state

Non-Pure Functional Languages

- 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

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
- 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**

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

- Goes hand-in-hand with first-class functions
- Can be passed as parameters
- Can be returned

Scala – Higher Order Functions

```scala
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.

- 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

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})     // Alternate syntax
```

Groovy

```
checkNumber(2, {n -> n % 2 != 0})
checkNumber(2, {it % 2 != 0})     // Alternate syntax
```

- 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*
- Retain bindings to variables in their defining scope
- Technically, anonymous functions and closures are distinct

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

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
```

Currying

- Method of partial application
- A function of n arguments can be modeled as a function of 1 argument, that returns a function of n-1, and so on.
- Example: **f(a, b) = a + b**
  - Can be considered as f(a)(b)
    - f(a) defined as {b -> {a + b}}
      - This is a closure, in which a is **bound** and b is **free**
    - f(1)(2) breaks down as:
      - f(1) = {b -> {1 + b}}
      - {b -> {1 + b}}(2) = 1 + 2

List Comprehensions

- A way of generating lists or list-like structures
  - Largely replaces for loops
  - Comes from ‘set-builder’ notation in mathematics:

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

  - Builds from an *input set*, a *predicate* and an *output* function
  - Encourages us to think more about the results, not the steps

Scala

```
doubledEvens = for(n <- (1 to 10) if n % 2 == 0) yield n*2
```

Groovy

```
// Not exactly the same, but still concise
doubleEvens = (1..10).findAll({it % 2 == 0}).collect({it*2})
```

Lazy Evaluation

- Delays the evaluation of an expression until it is needed
- Directly related to referential transparency
- Allows for:
  - Infinite ranges, e.g. [1..] (Haskell)
  - Calculations can be delayed until needed
    - Avoided if never needed
- Most languages default to *eager* evaluation, are lazy only for certain types or when asked
- Some languages default to lazy evaluation (Haskell)

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

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 Examples

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 structural types	case c: {def quack(): String} => c.quack()
By exhaustion	case _ => … // No other cases matched

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::ls, _) => 0.9
  case goodCustomerOrder: Order(c, _, _) if c.totalOrders > 10
=> 0.8
  case _ => 1.0  // Normal order
}
```

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

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
- Good free online book: learnyouahaskell.com
  - Strong introduction to FP concepts, including advanced ones
  - Even if learning Scala/Groovy/other, this is a good resource

Applying Functional Concepts in Non-Functional Languages

- Can apply some of these concepts in non-functional languages
- Language features exist for
  - Immutability
  - Reduction of shared state
  - Referential transparency / pure functions
    - Determinism
    - No side-effects
    - Idempotency
- Consider shared state as an optimization technique
  - Old adage - ‘make it correct, make it clear, make it fast’ - how does this affect our view of shared state?

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!

Slides and code available at:

https://github.com/bjhartin/FoundationsOfFunctionalProgramming