A (PERSONAL) INTRODUCTION TO FUNCTIONAL PROGRAMMING

Juan Manuel Gimeno Illa

WHAT IS FUNCTIONAL PROGRAMMING?

> *Functional programming (FP) is based on a simple premise with far reaching implications: we construct our programs using only **pure functions** — in other words, functions that have **no side effects**.*
>
> *Functional Programming in Scala*

WHAT ARE SIDE EFFECTS?

> *A function has a side effect if it does* ***something other*** *than simply returning a result*

- Modifying a variable
- Modifying a data structure in place
- Setting a field on an object
- Throwing an exception or halting with an error
- Printing to the console or reading user input
- Reading from or writing to a file
- Drawing on the screen

IMPERATIVE VS. FUNCTIONAL

IMPERATIVE

- Turing Machine
- Statement oriented
- Programming with places
- Simulation in time

FUNCTIONAL

- Lambda-calculus
- Expression oriented
- Programming with values
- Dataflow in space

REFERENTIAL TRANSPARENCY

> *An expression e is referentially transparent if, for all programs p, all occurrences of e in p can be replaced by the result of evaluating e without affecting the meaning of p.*

> *A function f is pure if the expression f(x) is referentially transparent for all referentially transparent x.*

(Functional Programming in Scala)

REFERENTIALLY TRANSPARENT

```
scala> val x = List(1, 2, 3)
x: List[Int] = List(1, 2, 3)
scala> val y = x.reverse
y: List[Int] = List(3, 2, 1)
scala> val r1 = y.toString
r1: String = List(3, 2, 1)
scala> val r2 = y.toString
r2: String = List(3, 2, 1)
```

Suppose we replace all occurrences of the term y with the expression referenced by y (its definition)

```
scala> val x = List(1, 2, 3)
x: List[Int] = List(1, 2, 3)
scala> val r1 = x.reverse.toString
r1: String = List(3, 2, 1)
scala> val r2 = x.reverse.toString
r2: String = List(3, 2, 1)
```

So the y can be referentially transparent (and reverse can be pure)

NOT REFERENTIALLY TRANSPARENT

```
scala> val x = new StringBuilder("Hello")
x: StringBuilder = Hello
scala> val y = x.append(", World")
y: StringBuilder = Hello, World
scala> val r1 = y.toString
r1: String = Hello, World
scala> val r2 = y.toString
r2: String = Hello, World
```

But now, if we replace all occurrences of the term y by its definition

```
scala> val x = new StringBuilder("Hello")
x: StringBuilder = Hello
scala> val r1 = x.append(", World").toString
r1: String = Hello, World
scala> val r2 = x.append(", World").toString
r2: String = Hello, World, World
```

So this shows that y is not referentially transparent (and append isn't pure)

BENEFITS

- Easier to reason about (substitution model works)
- The effects of a function are expressed in its signature
- Expressions are "context-free"
- Greater modularity & composability
  - No matter when I evaluate I get the same result (parallelizability)
  - No matter where I evaluate I get the same result (modularity)

FUNCTIONAL PROGRAMMING LANGUAGES

We can do functional programming in any language, but there are languages which promote (or even force) a functional programming style:

- Clojure / ClojureScript
- Scala
- Haskell
- Idris

CLOJURE / CLOJURESCRIPT

```clojure
(def start 458)
(def end 14)

(defn init-state [n] {n #{[]}})

(defn add-step [step paths]
  (set (map #(conj % step) paths)))

(defn step [[n ps]]
  [{(* 2 n)      (add-step :double ps)}
   {(quot n 10) (add-step :drop   ps)}])

(defn next-state [state]
  (apply merge-with union (mapcat step state)))

(defn final-state? [state] (get state end))

(def final-state
  (->> start
       init-state
       (iterate next-state)
       (drop-while (complement final-state?))
       first))
```

CLOJURE / CLOJURESCRIPT

- Lisp !!!
- JVM & JS Engines
- Interoperable with Java/JavaScript
- Immutable data structures
- Concurrency semantics for references
- core.async: channels and goroutines
- React.js: reagent, om, om.next, ...
- Datomic: the database as a value
- But no static typing

SCALA

```scala
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}

val intAddition: Monoid[Int] = new Monoid[Int] {
  override def op(a1: Int, a2: Int) = a1 + a2
  override def zero = 0
}

def monoidLaws[A](m: Monoid[A], gen: Gen[A]): Prop =
  forAll(gen)(a => m.op(a, m.zero) == a) &&
  forAll(gen)(a => m.op(m.zero, a) == a) &&
  forAll(gen ** gen ** gen){ case ((a, b), c)
    => m.op(m.op(a, b), c) == m.op(a, m.op(b, c)) }

def foldMap[A, B](as: List[A], m: Monoid[B])(f: A => B): B =
  as.foldLeft(m.zero)((b, a) => m.op(b, f(a)))
```

SCALA

- Object Functional (a better Java)
- Powerful type system
- Advanced libraries: Scalaz & Cats
- Advanced frameworks: Akka & Play
- Haskell-ish but practical
- But clumsy syntax
- Scala 2 -> Scala 3

HASKELL

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

instance Monad Maybe where
  -- return       :: a -> Maybe a
  return x        =  Just x
  -- (>>=)        :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing  >>= _ =  Nothing
  (Just x) >>= f =  f x

instance Monad [] where
  -- return :: a -> [a]
  return x  =  [x]
  -- (>>=)  :: [a] -> (a -> [b]) -> [b]
  xs >>= f  =  concat (map f xs)

sequence            :: Monad m => [m a] -> m [a]
sequence []         =  return []
sequence (mx:mxs)   =  do x  <- mx
                          xs <- sequence mxs
                          return (x:xs)
```

HASKELL

- Pure functional language
- Curried Functions
- Completely lazy (call by need)
- Pattern matching
- Typeclassopedia: Functors, Applicatives, Monads, ...
- The IO Monad !!!
- Very interesting typesystem

LIQUID HASKELL

```haskell
{-@ type IncrList a = [a]<{\xi xj -> xi <= xj}> @-}

split :: [a] -> ([a], [a])
split (x:y:zs) = (x:xs, y:ys)
  where
    (xs, ys)   = split zs
split xs       = (xs, [])

{-@ merge :: (Ord a) => IncrList a
                     -> IncrList a
                     -> IncrList a
  @-}

merge xs []          = xs
merge [] ys          = ys
merge (x:xs) (y:ys)
  | x <= y           = x : merge xs (y:ys)
  | otherwise        = y : merge (x:xs) ys

{-@ mergeSort :: (Ord a) => [a] -> IncrList a @-}

mergeSort []  = []
mergeSort [x] = [x]
mergeSort xs  = merge (mergeSort ys) (mergeSort zs)
  where
    (ys, zs)  = split xs
```

LIQUID HASKELL

- Refinement types: HaskellTypes + Predicates
- Uses a SMT (Solver Module Theories)
- So more invariants can be checked at compile time
  - Guarantee totality
  - Vector access bounds
  - Avoid infinite loops
  - ...

IDRIS

```
import Data.Vect

insert : Ord elem =>
  (x : elem) -> (xs_sorted : Vect k elem) -> Vect (S k) elem
insert x [] = [x]
insert x (y :: xs) = case x < y of
                          False => y :: insert x xs
                          True => x :: y :: xs

ins_sort : Ord elem => Vect n elem -> Vect n elem
ins_sort [] = []
ins_sort (x :: xs) = let xs_sorted = ins_sort xs in
                         insert x xs_sorted
```

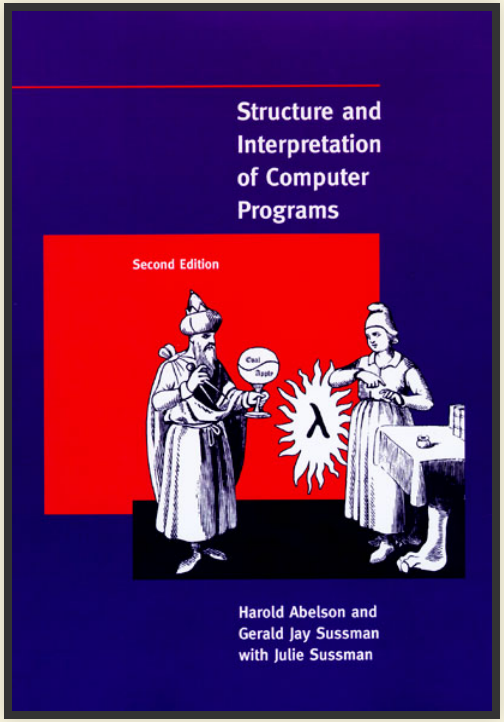
IDRIS

- Similar to Haskell but strictly evaluated
- Dependent types !!!
- Idris 2 on the making

SUGGESTIONS FOR FURTHER STUDY

- Functional programming principles
- (A/Some) functional programming language(s)
- Functional Data Structures
- Some Category Theory concepts
- A little bit about type theory
- Lots & lots of videos of presentations to watch
- Lots & lots of code to read
- ...

SUGGESTED BOOKS

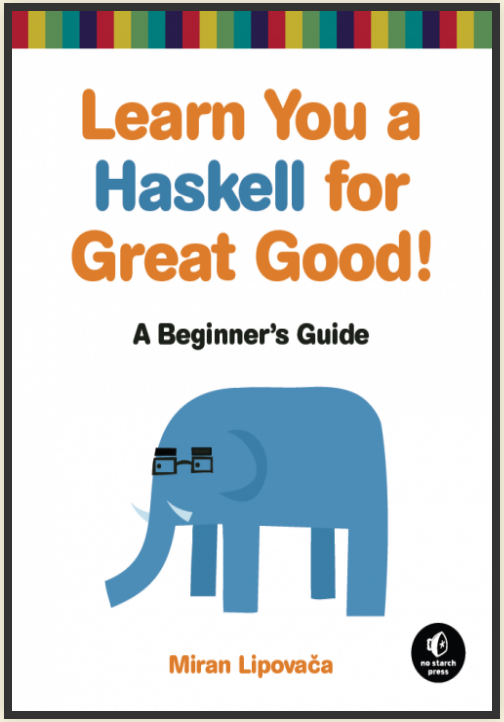


Structure and Interpretation of Computer Programs

SUGGESTED BOOKS

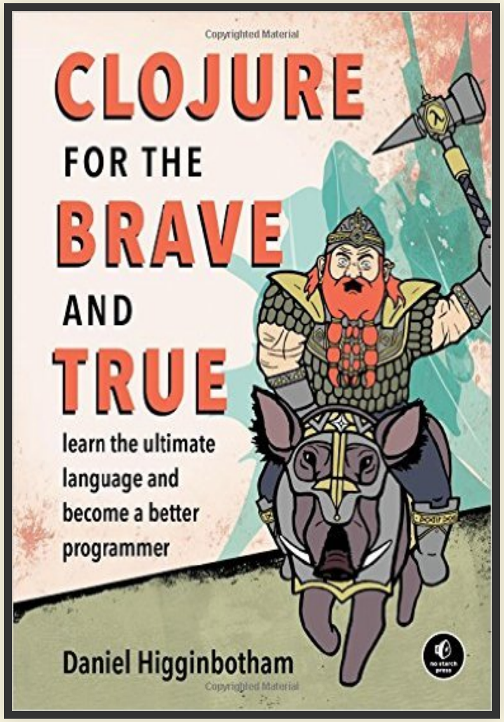
Functional Programming in Scala

SUGGESTED BOOKS



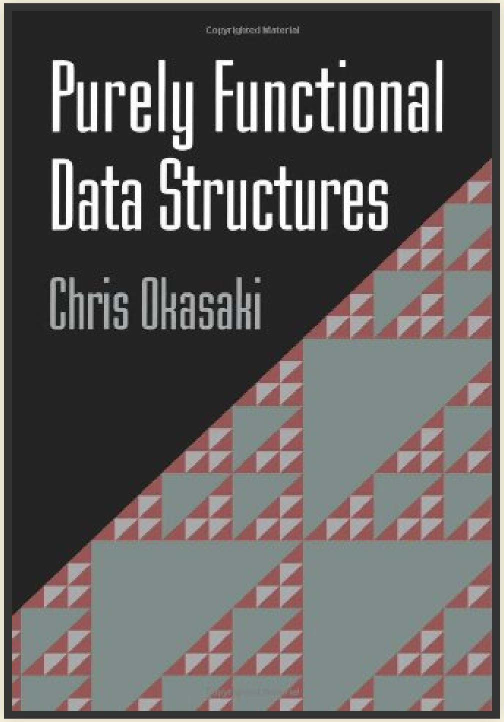
Learn You a Haskell for Great Good!

SUGGESTED BOOKS



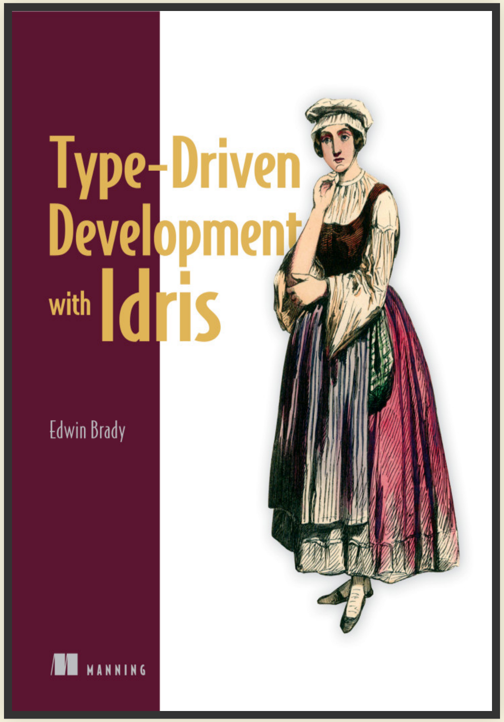
Clojure for the Brave and True

SUGGESTED BOOKS



Purely Functional Data Structures (PhD Thesis)

SUGGESTED BOOKS



Type-Driven Development with Idris

… AND THE BEST IS YET TO COME !!!

THANKS !!!

OPEN DISCUSSION