Six (give or take) Easy Pieces on Functional Programming

Adam Keys, developer at large and expert typist
Developer Day Austin
http://therealadam.com

Saturday, February 6, 2010

Hi, I'm Adam Keys. I'm a developer at large, expert typist and a bit of a language lawyer. Over the past couple years, I've become increasingly interested in functional programming.

What is FP?

* Programs composed of function
* Functions are first-class things
* Verb-oriented

What is FP? λ **Hype!!!!!11!!**

Saturday, February 6, 2010

* make your system faster
* make your code run on multiple cores
* make your code robust
* make your code easy to understand

What is FP? λ **You’re already doing it.**

Saturday, February 6, 2010

* jQuery is a monad?
* Higher-order functions
* Small, simple methods

Pattern matching

Saturday, February 6, 2010

* Regexes break text into useful chunks
* Pattern matching breaks function calls, messages and data into useful chunks

Pattern matching λ **Specialize, kill conditionals**

Saturday, February 6, 2010

* Decompose a problem into its case and write a function for each one
* Using pattern matching pushes all sorts of messy code down into the compiler
* Why debug conditionals when you can do it in the language?

```
fib(0) -> 1;
fib(1) -> 1;
fib(N)
  when N > 0 ->
    fib(N - 1) + fib(N - 2).
```


We’ve specialized the easy cases on the first two lines. The last bit handles the common case. Note the use of a pattern guard.

Pattern matching λ **Tail recursion**

Saturday, February 6, 2010

* For many kinds of state, you can model transitions via some form of recursion
* But that means you need a way to mitigate quickly growing call stacks
* For some forms of recursion, you can optimize the recursion away into a loop

```
fib2(N) ->
     fib_tr(N, 0, 1).

fib_tr(0, Result, _) -> Result;
fib_tr(Iter, Result, Next)
  when Iter > 0 ->
    fib_tr(Iter - 1, Next, Result + Next).
```


* We call into a tail recursive version
* We pass state along via `Iter` and `Next`

```
% Eshell V5.7.4  (abort with ^G)
% 1> c(fib).
% {ok,fib}
% 2> fib:fib2(4).
% 3
% 3> fib:fib2(100).
% 354224848179261915075
% 4> fib:fib2(1000).
%
43466557686937456435688527675040625802564660517371780402481729089536
55541794905189040387984007925516929592259308032263477520968962323987
33224711616429964409065331879382989696499285160037044761377951668492
28875
% 5> fib:fib2(10000).
%
33644764876431783266621612005107543310302148460680063906564769974680
08144216666236815559551363373402558206533268083615937373479048386526
82630408924630564318873545443695598274916066020998841839338646527313
00088830269235673613135117579297437854413752130520504347701602264758
31890652789085515436615958298727968298751063120057542878345321551510
38708182989697916131278562650331954871402142875326981879620469360978
79900350962302291026368131493195275630227837628441540360584402572114
33496118002309120828704608892396232883546150577658327125254609359112
82039252853934346209042452489294039017062338889910858410651831733604
37470737908552631764325733993712871937587746897479926305837065742830
16163740896917842637862421283525811282051637029808933209990570792006
```

Saturday, February 6, 2010

* `erl` launches eshell, the Erlang read-evaluate-print loop (REPL)
* We compile our module and then run some examples

Types

Saturday, February 6, 2010

* If you’ve only used C, C++, Java, or C#, you’ve had a taste of type systems
* But when you add pure functions into the mix, things get a lot more interesting

Types λ **Type classes**

* In OOP binds state and behavior, but Haskell isn't OO
* In FP, behavior is king and state is a peculiar admission
* Haskell lets you specify types of values and the operations you can perform
* It's a little like interfaces in OOP

```haskell
class Addressable a where
    address :: a -> String

data Residence = Residence String String String String
                 deriving (Show, Read, Eq)

instance Addressable Residence where
    address (Residence street city state zip) = join parts
        where parts = ["(Residence)", street, city, state, zip]
```


* I want a class of things I can put a mailing label on
* a is a type variable
* Residence is implicitly in `Show`, `Read`, and `Eq` and explicitly `Addressable`
* Note the pattern match in `address` and the type inference

```haskell
data Business = Business {
      attention :: String,
      street :: String,
      city :: String,
      state :: String,
      -- So as not to conflict with Prelude.zip
      bZip :: String
    }
                deriving (Show, Read, Eq)

instance Addressable Business where
    address b = join parts
      where parts = ["(Business)", attention b, street b, city b,
state b, bZip b]
```


* `Business` is a record-style type with getters. Setters aren’t possible.
* `address` looks similar, but we’re using functions to grab values out of the record

```haskell
mail :: Addressable a => String -> a -> IO ()
mail msg addr = putStrLn message
    where message = join parts
          parts = ["Mailing message: ", msg, "to: ", address addr]

-- A more-familiar looking helper
join :: [String] -> String
join = intercalate "\n" -- NB: In Haskell, we can "curry" and omit
the last parameter
```


* `mail` can operate on any instance of Addressable
* `IO ()` means this has side-effects and must run inside the IO monad
* `join` is a helper that does what you probably think it would
* `intercalate` is an example or partial application

```haskell
main = do
  mail msg r
  where r = Residence "123 Main" "Dallas" "Texas" "75201"
        msg = "Hello, from Developer Day Austin!"

-- *Main> :load "addresses.hs"
-- [1 of 1] Compiling Main             ( addresses.hs, interpreted )
-- Ok, modules loaded: Main.
-- *Main> :main
-- Mailing message:
-- Hello, from Developer Day Austin!
-- to:
-- (Residence)
-- 123 Main
-- Dallas
-- Texas
-- 75201
```

Saturday, February 6, 2010

* We run this in GHCI, the Haskell REPL included in GHC
* `:load` and `:main` are special constructs

Types λ **Algebraic data types**

Saturday, February 6, 2010

* It's not OOP, but you can still model the relationship between types
* This comes in really handy when you're pattern matching

```
data Login = User String | Login String String | Token String
             deriving (Show)

login :: Login -> Bool
login (User s) = True
    where validLogins = ["bishop", "whistler", "mother", "carl"]
login (Login u p) = (u, p) `elem` validLogins
    where validLogins = [("adam", "letmein"),
                         ("alice", "sekrit"),
                         ("bob", "foo")]
login (Token t) = t `elem` validTokens
    where validTokens = ["0xdeadbeef",
                         "myvoiceismypasswordverifyme"]
```


* The type `Login` can be any of these three things
* We use pattern matching to handle each kind of login

```haskell
main = do
  args <- getArgs
  let method = case head args of
                 "user" -> User (args !! 1)
                 "login" -> Login (args !! 1) (args !! 2)
                 "token" -> Token (args !! 1)
  case login method of
    True -> putStrLn "You are authorized"
    False -> putStrLn "Access denied!"
```


* `do` and `<-` are sugar over monads
* `let` is another way to define a function
* `case` is a way to do pattern matching outside of function definitions
* `!!` is the list subscript function

Types λ **Functors, monads, arrows**

* In Haskell, you end up building your own control abstractions
* They’re all about different ways to work through encapsulation and coupling
* Some of them have really weird names and really bad explanations
* Fun to turn your brain inside out with

Multi-core

* In the future, computers will be overflowing with cores!
* Current languages make it really hard to deal with this, by virtue of exposing developers to locks or by exposing themselves to internal locks

Multi-core λ **Concurrency for async tasks**

Saturday, February 6, 2010

* We want one set of abstractions for running processes on multiple cores for IO-bound workloads
* Actors, queues, processes, message passing

```erlang
%%% The Computer Language Benchmarks Game
%%% http://shootout.alioth.debian.org/
%%% Contributed by Jiri Isa

-module(threadring).
-export([main/1, roundtrip/2]).

-define(RING, 503).

start(Token) ->
   H = lists:foldl(
      fun(Id, Pid) -> spawn(threadring, roundtrip, [Id, Pid]) end,
      self(),
      lists:seq(?RING, 2, -1)),
   H ! Token,
   roundtrip(1, H).

roundtrip(Id, Pid) ->
   receive
      1 ->
         io:fwrite("~b~n", [Id]),
         erlang:halt();
      Token ->
         Pid ! Token - 1,
         roundtrip(Id, Pid)
   end.

main([Arg]) ->
   Token = list_to_integer(Arg),
   start(Token).
```


`spawn` is the Erlang function that creates a new process. Its parameters indicate the module and function to call, and the array of arguments to pass the function.

```erlang
%%% The Computer Language Benchmarks Game
%%% http://shootout.alioth.debian.org/
%%% Contributed by Jiri Isa

-module(threadring).
-export([main/1, roundtrip/2]).

-define(RING, 503).

start(Token) ->
   H = lists:foldl(
      fun(Id, Pid) -> spawn(threadring, roundtrip, [Id, Pid]) end,
      self(),
      lists:seq(?RING, 2, -1)),
   H ! Token,
   roundtrip(1, H).

roundtrip(Id, Pid) ->
   receive
      1 ->
         io:fwrite("~b~n", [Id]),
         erlang:halt();
      Token ->
         Pid ! Token - 1,
         roundtrip(Id, Pid)
   end.

main([Arg]) ->
   Token = list_to_integer(Arg),
   start(Token).
```


Here we start the counting process. We tell the first process in the ring to start counting by sending it a message using `!` with the token as the argument. Then we call roundtrip, which will end up halting the parent process.

```erlang
%%% The Computer Language Benchmarks Game
%%% http://shootout.alioth.debian.org/
%%% Contributed by Jiri Isa

-module(threadring).
-export([main/1, roundtrip/2]).

-define(RING, 503).

start(Token) ->
   H = lists:foldl(
      fun(Id, Pid) -> spawn(threadring, roundtrip, [Id, Pid]) end,
      self(),
      lists:seq(?RING, 2, -1)),
   H ! Token,
   roundtrip(1, H).

roundtrip(Id, Pid) ->
   receive
      1 ->
         io:fwrite("~b~n", [Id]),
         erlang:halt();
      Token ->
         Pid ! Token - 1,
         roundtrip(Id, Pid)
   end.

main([Arg]) ->
   Token = list_to_integer(Arg),
   start(Token).
```


`receive` is the Erlang function that handles receiving messages from other processes. The interesting part is the Token pattern, which will decrement the counter and call down the thread ring. The roundtrip call at the end is the tail recursion that keeps this process going.

Multi-core λ **Parallelism for computation**

* We want another set of abstractions for running CPU-bound workloads
* STM, refs

```clojure
(defn run [nvecs nitems nthreads niters]
  (let [vec-refs (vec (map (comp ref vec)
                           (partition nitems (range (* nvecs nitems)))))
        swap #(let [v1 (rand-int nvecs)
                    v2 (rand-int nvecs)
                    i1 (rand-int nitems)
                    i2 (rand-int nitems)]
                (dosync
                 (let [temp (nth @(vec-refs v1) i1)]
                   (alter (vec-refs v1) assoc i1 (nth @(vec-refs v2) i2))
                   (alter (vec-refs v2) assoc i2 temp))))
        report #(do
                 (prn (map deref vec-refs))
                 (println "Distinct:"
                          (count (distinct (apply concat (map deref vec-refs))))))]
    (report)
    (dorun (apply pcalls (repeat nthreads #(dotimes [_ niters] (swap)))))
    (report)))

;; Taken from http://clojure.org/refs
```


To start off, we're going to put a bunch of `ref`s into a vector. We create the data inside each vector using `ref`, which gives us a mutable value.

```clojure
(defn run [nvecs nitems nthreads niters]
  (let [vec-refs (vec (map (comp ref vec)
                           (partition nitems (range (* nvecs nitems)))))
        swap #(let [v1 (rand-int nvecs)
                    v2 (rand-int nvecs)
                    i1 (rand-int nitems)
                    i2 (rand-int nitems)]
                (dosync
                 (let [temp (nth @(vec-refs v1) i1)]
                   (alter (vec-refs v1) assoc i1 (nth @(vec-refs v2) i2))
                   (alter (vec-refs v2) assoc i2 temp))))
        report #(do
                 (prn (map deref vec-refs))
                 (println "Distinct:"
                          (count (distinct (apply concat (map deref vec-refs))))))]
    (report)
    (dorun (apply pcalls (repeat nthreads #(dotimes [_ niters] (swap)))))
    (report)))

;; Taken from http://clojure.org/refs
```


Here, we're going to use Clojure's STM machinery to transactionally modify our vectors. `dosync` enters a transaction and `alter` modifies ref values.

```clojure
(defn run [nvecs nitems nthreads niters]
  (let [vec-refs (vec (map (comp ref vec)
                           (partition nitems (range (* nvecs nitems)))))
        swap #(let [v1 (rand-int nvecs)
                    v2 (rand-int nvecs)
                    i1 (rand-int nitems)
                    i2 (rand-int nitems)]
                (dosync
                 (let [temp (nth @(vec-refs v1) i1)]
                   (alter (vec-refs v1) assoc i1 (nth @(vec-refs v2) i2))
                   (alter (vec-refs v2) assoc i2 temp))))
        report #(do
                 (prn (map deref vec-refs))
                 (println "Distinct:"
                          (count (distinct (apply concat (map deref vec-refs))))))]
    (report)
    (dorun (apply pcalls (repeat nthreads #(dotimes [_ niters] (swap)))))
    (report)))

;; Taken from http://clojure.org/refs
```


Finally, we'll print out our modified vectors. `deref` is how we read values out of a mutable `ref`. The Clojure STM will make sure we get a consistent value out of each `ref`.

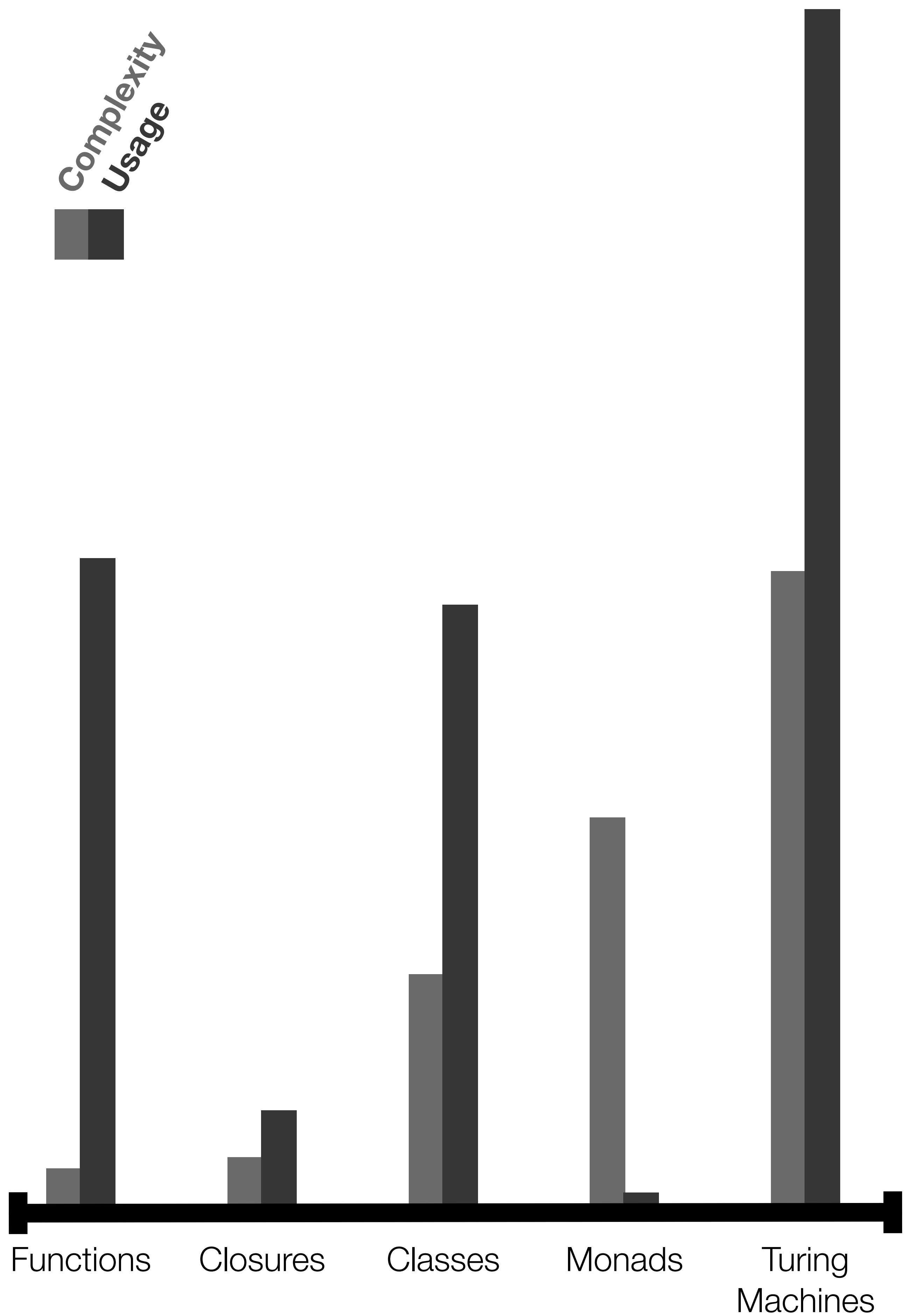
Orders of Magnitude

Saturday, February 6, 2010

In preparing this, I got to thinking about scale and how one thing compares to the next in terms of size and transition cost.

Orders of Magnitude λ **Organization**

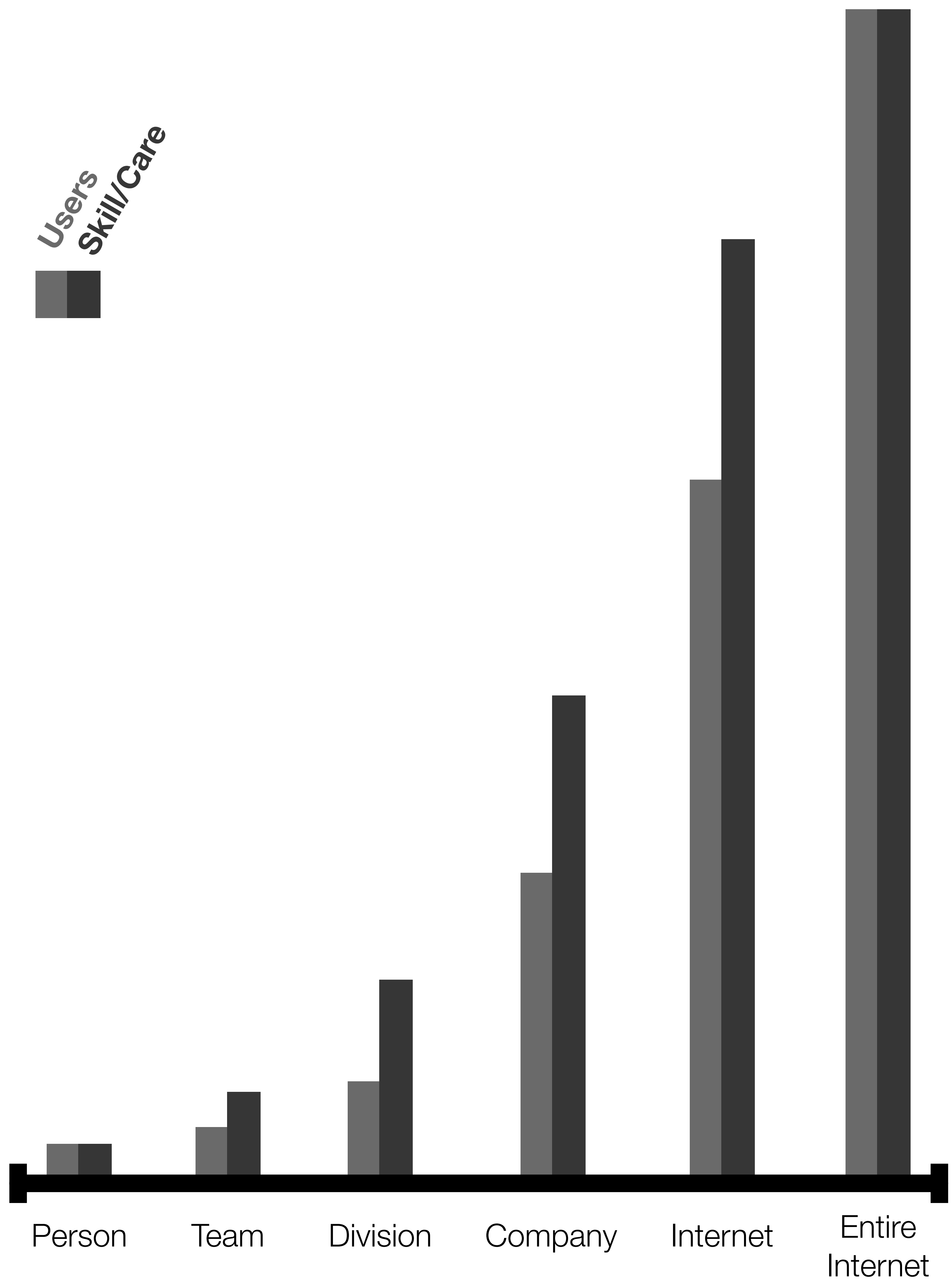
* What do we use to organize our code?
* Functions, Closures, Classes, Monads, Turing Machines



* The things on the left are limited and can’t do much, but are easy to reason about
* The things on the right are more general, can do more, and require more theory
* The middle is fertile ground for exploration in programming

Orders of Magnitude λ **Scale**

* Person, team, group, division, company, internet, INTERNET

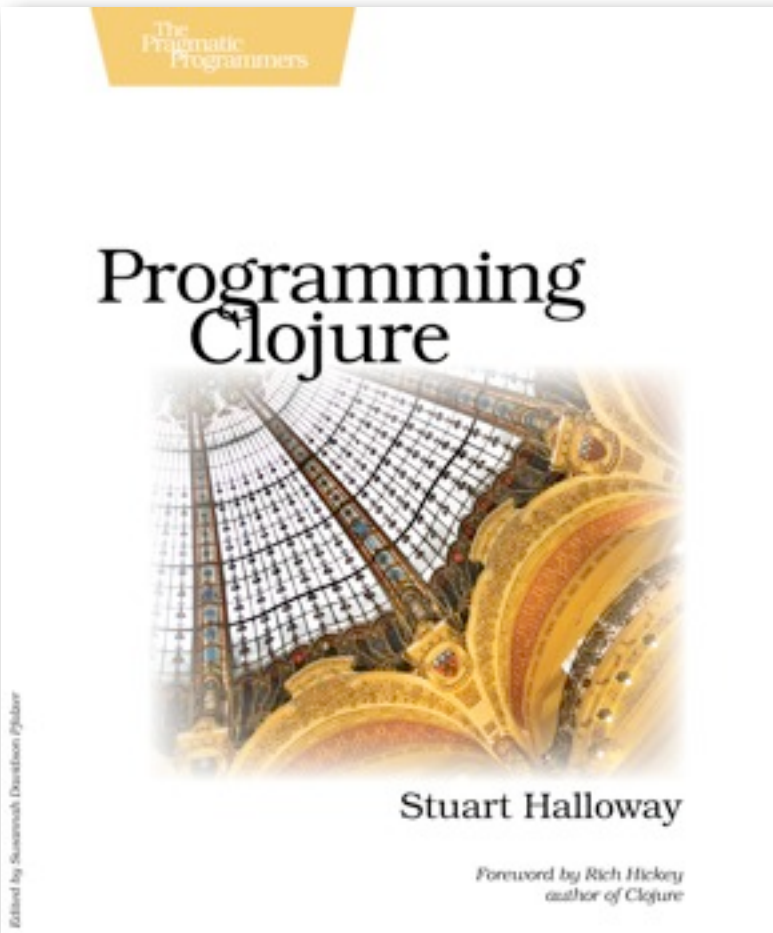


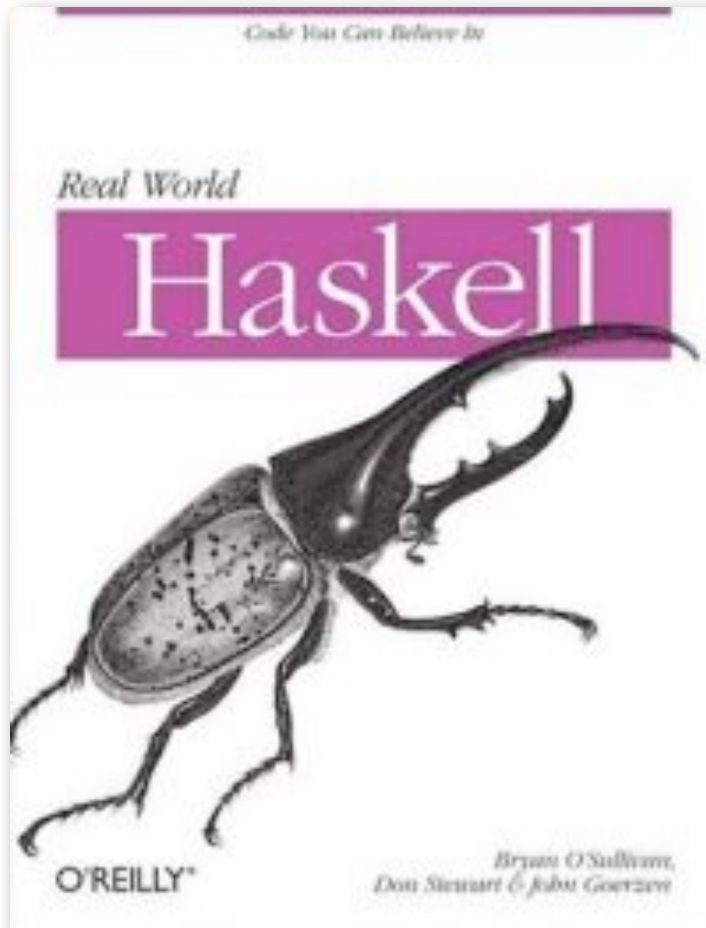
* The things on the left can be written with anything with little care
* The things on the right require specially suited tech applied with great care
* Perhaps FP is a way to more gracefully move along this gradient?

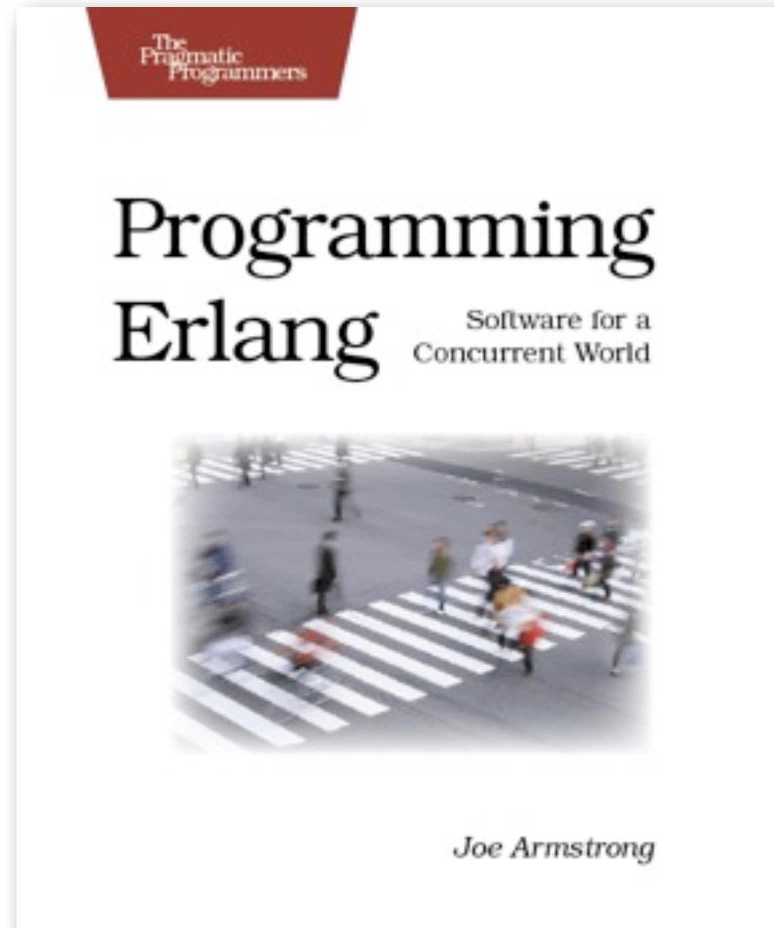
When should I use FP?

Saturday, February 6, 2010

* When other languages fail you
* When it makes you happy







Further reading, for your enjoyment

Ask me about:
- encoding time
- monads and composition
- laziness, purity, and state
- dachshunds

If you see me around and are curious about topics I didn't have to cover, ask me about...

Thanks!

Saturday, February 6, 2010

I'll post these slides and code on my site at therealadam.com later this week. Thanks!