WHY FP?

Brisbane Functional Programming Group - 2013-07-23

WHAT MAKES GOOD CODE?

IT NEEDS TO BE DRY

Logic shouldn't be repeated in a system.

Create abstractions for common patterns.

High level abstractions usually take logic as parameters.

DRY CODE, OO STYLE

Behaviour is abstracted over with objects:

- Decorators
- Visitors
- SimpleBeanFactoryAwareAspectInstanceFactory !!!

Even with mixins, these compose awkwardly.

IT NEEDS TO BE REASONED ABOUT

Reading ≠ Comprehension

Reasoning about code requires knowing and thinking about all dependencies.

Coupling should minimised and explicit.

Coupling to time is most difficult to reason about.

If you need to open up a debugger to comprehend things, you've already lost (IMHO).

GOOD CODE, GROSSLY TRIVIALISED

Built from composable abstractions so we don't repeat ourselves.

Coupling of these abstractions should be minimal and explicit so they can be understood.

WHAT IS FUNCTIONAL PROGRAMMING?

PROGRAMMING WITH FUNCTIONS! DUH!

Break things up into lots of small reusable functions.

Compose those functions together into bigger functions.

Pass functions to other (higher order) functions.

SOUNDS EASY?

Not quite!

Because you are passing functions to functions, you have less control over execution order.

It makes weird mutation bugs even weirder and difficult to figure out!

Coding style needs to be changed to not rely on execution order.

Pure / Referencially Transparent Functions

Functions that can be replaced with it's value without changing program behaviour.

`1 + 1` is always `2`.

Functions that mutate structures in place but don't leak the mutable instance to the caller are pure too!

These functions are the building blocks of safe FP abstractions.

"TYPE ASSISTED REFERENTIAL TRANSPARENCY"

"Pure vs Impure Langs/FP" is a false dichotomy.

Side effects are always useful, regardless of the paradigm.

Haskell has side effects, just explicitly segregates pure from impure functions with types.

Conversely, purity is still a concern in "impure" languages, it is just implicit.

Favour immutability

Effective Java, written 12 years ago, recommends this.

Variables are a large cause of bugs, regardless of PL.

If something changes, anything that reads it is coupled to time / execution order.

In FP, we want to feed values through pipelines of functions without needing to build copying into our abstractions.

Immutability - Language Comparison

All refs and collections are immutable by default.

Haskell: Mutable Refs and STM in IO code only

Clojure: STM only and 'Transient' data structures.

Scala: Variables anywhere. Also has STM.

OCaml: Unrestricted Refs and STM.

Lazy evaluation

Lazy evaluation will only execute a computation when a value is needed.

Can describe a computation without caring about it is needed without wasting CPU cycles.

Wonderful for abstractions relating to memoization or infinite structures.

Code cannot be coupled to time given no guarantees to execution order.

Lazy vs Eager: Language Comparison

Haskell is lazy by default.

Clojure is an eager language, but has lazy collections & some lazy library functions.

Scala & OCaml are eager but have laziness language constructs.

Statically vs Dynamically Typed

Has nothing to do with FP! Completely subjective.

Some people that think that dealing with compiler errors is too hard.

Some people that think that not having types to help reason about and verify code is too hard.

You can do FP on either, but be wary that I'm biased to static types (Haskell,Scala,OCaml)

SUMMARY

FP is about weakening parts of our code so we can write stronger abstractions around it.

By decoupling our code from execution order, we can do lots of wonderful things.

FUNCTIONAL ABSTRACTIONS

COLLECTION LIBRARIES

Functional collection libraries are not novel.

map, flatMap, filter, reduce/fold, ...

Perl,Ruby,Groovy,Python have been doing similar things for a long time.

Nested mutable data structures cause big headaches.

COLLECTION EXAMPLE

```
val xacts = List(Xact("Buy Pizza", -20,"food"),...)

//Get me the list of accounts -> List[Xact] in descending spend order
xacts.filter(
  _.amount < 0 //Only count negative transactions (expenses)
).groupBy(
  _.account  //Group each xact by its account
).toSeq.sortBy(
  _._2.map(_.amount).sum //Sort by the sum of all xacts.
)
```

PARALLEL COLLECTION EXAMPLE

```
val xacts = List(Xact("Buy Pizza", -20,"food"),...)

//Get me the list of accounts -> List[Xact] in descending spend order
xacts.par.filter(
  _.amount < 0 //Only count negative transactions (expenses)
).groupBy(
  _.account  //Group each xact by its account
).toSeq.sortBy(
  _._2.map(_.amount).sum //Sort by the sum of all xacts.
)
```

Parallelisation remarks

Operations must be pure to make this work.

No distribution. Middle ground between single core and MapReduce-style parallelism.

Scoobi (in Scala) gets pretty close to this api while distributing computation over hadoop.

No FP language can parallelise automatically. Not even haskell.

Explicit computations

Because we have the power to abstract over computation, we can write our own 'computation types' that describe and constrain what we're doing.

Allows us to be precise, while FP abstractions help keep it expressive.

Option / Maybe

I use this heavily in my scala code.

```
def findUserByName(users:Seq[Users],name:String):Option[User]={
 users.find(_username==name)
}

//Convert the optional user to an email address
findByUserName(users,"dkolera1").map(_name).getOrElse("Unknown")

//They even compose. This will return a full option if both are present
for{
 creator<-findUserByName(users,creatorUsername)
 owner  <-findUserByName(users,ownerUsername)
}yield(creator,owner)
```

DISJUNCTION / EITHER

I use this heavily in my scala code.

```
def findUserByName(users:Seq[Users],name:String):Error\/User={
 users.find(_username==name).match{
  caseNone  =>-\/(Error(s"User$namenotfound"))
  caseSome(u)=>\/-(u)
 }
}

//Convertthepotentiallyfounduserto.aname
findByUserName(users,"bkolera1").map(_name).getOrElse("Unknown")

//Thiswillreturna\/-(String,String)ifbothareright,elsethe
//firsterrormessagewillbereturnedontheleft
for{
 creator<-findUserByName(users,creatorUsername)
 owner  <-findUserByName(users,ownerUsername)
}yield(creator.name,owner.name)
```

Are you crazy? Why not just use nulls and exceptions?

Because these are explicit. Caller knows what to expect from me.

No loss of semantics (except not jumping the stack!) and are actually more expressive.

It is also really cool that caller can treat errors exactly like no value (or use the error if they please)!

TAKING THIS EVEN FURTHER!

Can describe lots of computations that have properties like:

- Reader: A computation that requires some configuration before producing a value.
- Writer: A computation logs some state between steps.
- State: A computation that can mutate state (purely) between steps.
- A mixture of any of the above mentioned things.

EXPLICIT BUT EXPRESSIVE PURITY

Data structures that keep stateful computations pure and also

I find this very exciting and beautiful!

Sure, they aren't going to work for 100% of problems but they are a terrific first step to attempt to keep things pure and composable.

PARSER COMBINATORS

```
import Text.Parsec
import Control.Applicative hiding ((<|>))

number         = many1 digit
positiveNumber = char '+' *> number
negativeNumber = (:) <$> char '-' <*> number

integer = stringToInteger <$> (positiveNumber <|> negativeNumber <|> number)
  where stringToInteger = read :: String -> Integer
```

Each bit is ~ String -> Either Error (A,String)

Glued together with special combinators (many1, <|>,etc.).

Parser Combinators - Continued

We'd not be able to break these parsers up if they had internal state.

These compose better and more explictly than appending regex strings.

Primitives have error handling in them, each knows how to describe what it was looking for and couldn't find.

Something special: <$>, *> and <*> aren't even part of parsec.

TYPECLASSES: OPEN POLYMORPHISM

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)

findThing :: (Eq a) => a -> [a] -> Maybe a
findThing a = find (a ==)
```

We don't need to know what a is. The eq instance can be implemented elsewhere.

Abstraction is completely open, yet safe. Unlike java's equality method!

THESE THINGS COMPOSE TOO!

An example from argonaut, an awesome scala JSON parsing library:

```
case class Account(id: Int, name: String)
case class Person(name: String, age: Int, accounts: List[Account])

implicit def AccountCodecJson =
 casecodec2(Account.apply, Account.unapply)("id", "name")

implicit def PersonCodecJson =
 casecodec3(Person.apply, Person.unapply)("name", "age", "accounts")

val people  = List(Person("ben","26",List(Account(1,"account1"))),...)
val prettyStr = people.asJson.spaces2
val personOpt = prettyPrinted.decodeOption[List[Person]]
```

(FP IS SO AWESOME IT EVEN HAS DECOUPLED, COMPOSABLE POLYMORPHISM. ;))

FUTURES

```
//Use dispatch to call a REST API and parse its XML
def temperature(loc:Location): Future[Double] = ???

def hottest(locs:Location*) = {
  val locFutures = locs.map(loc => temperature(loc).map(_ -> loc))
  val futureLocs = Future.sequence(locFutures)
  futureLocs.map(_.maxBy(_._1))
}
```

We use this a lot for glue APIs that need to hit lots of services.

Calls can happen in parallel reducing latency and are also non blocking.

SOFTWARE TRANSACTIONAL MEMORY

```
(def account1 (ref 100))
(def account2 (ref 0))

(defn transfer [amount from to]
  (dosync
    (alter from - amount) ;alter from => (- @from amount)
    (alter to + amount))) ;alter to => (+ @to amount)

;=> @account1 -> 100
;=> @account2 -> 0
;=> (transfer 100 account1 account2) -> 100
;=> @account1 -> 0
;=> @account2 -> 100
```

GHC Optimisations

Because haskell is completely lazy and knows which functions are pure, it can make some very cool improvements to your code.

Things like fold fusion, common expression substitution, inlining, etc.

Generally have really good performance without trying, except when you have space leaks!

CONCLUSIONS

WE ARE ALREADY KINDA USING FP

Collection libraries

Ruby uses loan patterns, etc.

Lack of purity inhibits our ability to fully realise FP abstractions.

FP IS ACTUALLY NOT EASY TO LEARN!

It forces us to be careful about how we architect our code.

Doing the things that we need to do in FP requires us to change the structures and abstractions that we use.

This change in thought process is painful, yet wonderful and irreversible.

IT ISN'T JUST ABOUT CONCURRENCY!

... or parallelism

... or testing

... or running away from OO

... or a general feeling of smugness and superiority ;)

It is bigger and more profound than anything like that.

IT ISN'T ABOUT REMOVING SIDE EFFECTS!

Side effects are very important!

We all have files, databases and sockets to read and write from.

We're accepting the fact that functions that have side effects don't compose too well.

THE ABSTRACTIONS ARE KEY!

We make as much of our logic referentially transparent to make it easy to reason about.

Decoupling our code from time makes our code much easier to reuse and build abstractions over that would otherwise be unimaginable.

We strive to use pure code to tame the uncertainty of the outside world (like with STM) and make it as safe and easy to reason about as is possible.

IT ISN'T REALLY ABOUT TYPES!

FP doesn't mean coding with rigid types everywhere.

You can have your programmers make there own assertions and reasoning about safeness of code.

If you're smart enough to do that, go for it!

I'm stupid, I make mistakes all of the time. I prefer to let the computer help.

WHERE TO GO FROM HERE?

BOOKS!

- Haskell: Learn you a Haskel For Great Good
- Haskell: Real World Haskell
- Scala: Functional Programming in Scala
- Clojure: The Joy Of Clojure

WRITE CODE!

Slow relevant practice is the only way.

Use FP for any reports, personal automations or microservices.

ASK QUESTIONS!

Lots of very knowledgeable people on our mailing list all to willing to help.

Lang specific mailing lists too.

Also freenode.org irc channels #haskell, #scala, #fp-in-scala, #bfpg

TEACH OTHERS WHAT YOU LEARN!

Teaching is the best form of learning, with pretty much everything.

Commit yourself to learning something and plan to do a BFPG talk! (We can help and mentor)

THANKS FOR LISTENING!

twitter: @benkolera

email: ben.kolera@gmail.com

Slides: whyfp.benkolera.com