Free Monad + Free Applicative === Free Workshop

While you are waiting, please download the internet:

brew install sbt

git clone https://github.com/lukestephenson/monad-coproduct-workshop.git

sbt compile

WIFI

- Network: RMIT-University
- User: x80305
- Password: rmit.1234
- While you are waiting, please download the internet:
- brew install sbt
- git clone https://github.com/lukestephenson/monad-coproduct-workshop.git
- sbt compile
- starting point branch: free-applicative
- solution branch: free-applicative-solution

Me

- Lead Developer at REA
- Interest in Scala / Reactive / FP
- Wanted to learn about Free Applicative

Today

- 15 minutes - Slides
- 1.5 hours - Time to work on problem
- 15 minutes - Summary / discussion / feedback

Monad vs Applicative

- Applicative (cats):
- Applicative[A].map(f: A => B): Applicative[B]
- Monad adds:
- Monad[A].flatMap(f: A => Monad[B]): Monad[B]

Monad

- M[B] represents a computation with result B
- Monad[A].flatMap(f: A => **Monad[B]**): Monad[B]
- If we can represent a Computation as M[B], then flatMap allows us to compose Computations.
- The final result can be affected by a chained computation

Applicative

- Applicative[A].map(f: A => **B**): Applicative[B]
- Has all the data required to evaluate
- Allows for some static analysis

Free (recap)

- Check the brilliant cats docs http://typelevel.org/cats/tut/freemonad.html

- DSL for application effects (with manipulation)

```
sealed trait AppAction[A]

case class LoadUser(userId: String) extends
AppAction[User]

case class GetFollowers(userId: String)
extends AppAction[List[User]]
```

Free (recap)

- Lift the DSL for application effects into Free

```
val script: Free[AppAction, User] =
Free.liftF(LoadUser(“luke”))
```

```
script.flatMap(user => …)
```

Free - Why

- Separate Intention from Implementation
- Easily swap out implementations
- Extract side effects from business logic
- Simple testing of business logic

Free Applicative

- DSL for for application side affects, without the ability to chain / manipulate

- Can be exactly the same DSL used with Free

Use the simplest abstraction possible

- If two operations are independent, why are we expressing them using a monadic composition

```
for {
    user1 <- loadUser(“luke”)
    user2 <- loadUser(“kenbot”)
} yield doSomething(user1, user2)
```

Use the simplest abstraction possible

- De-sugared it is not so pretty

```
loadUser(“luke”).flatMap { user1 =>
  loadUser(“kenbot”).flatMap {user2 =>
    doSomething(user1, user2)
  }
}
```

Use the simplest abstraction possible

- Wrong! (The stack overflow answer)
- Coupled to behaviour of Future

```
val user1Future = loadUser(“luke”)
val user2Future = loadUser(“kenbot”)
for {
   user1 <- user1Future
   user2 <- user2Future
} yield doSomething(user1, user2)
```

Use the simplest abstraction possible

- Combine independent operations

```
loadUser(“luke”) |@| loadUser(“kenbot”) {
  case (user1, user2) =>
    doSomething(user1, user2)
}

Apply.map2(loadUser(“luke”),
loadUser(“kenbot”))(doSomething)
```

Coproduct

- Allows for composing DSLs used by Free (or FreeApplicative)
- type MyApp[A] = Coproduct[DSL1, DSL2, A]
- Chain together NaturalTransformations (aka Interpreters)
- val interpreter: MyApp ~> Id = DSL1Interpreter or DSL2Interpreter

Inject

- With a 1 to 1 match between the DSL and free, simply:
- Free.liftF(LoadUser(“luke”))
- With CoProduct the DSL no longer maps directly to Free
- The DSL could potentially be used in many programs!

Inject

```
class ConfigActions[F[_]](implicit I: Inject[ConfigAction, F]) {
  def getConfig(key: String) = Free.inject[ConfigAction, F](GetConfig(key))
}
```

- Lifting a DSL into a partially applied Free “F”
- “F” will be defined later implicitly

```
val S = implicitly[ConfigActions[AppAction]]
```

Workshop

- Take a (inefficient) solution expressed only with Monadic composition
- Introduce Free Applicative where monadic composition is not required
- (I have more ideas if that is not enough)

Conclusion

- Is Free Applicative a tool you want to add to your application

- Is abstracting away concurrency a good thing?