An Intuitive Guide to Combining Free Monad and Free Applicative

Who Am I?

- I'm Cameron!
  - Linkedin - https://www.linkedin.com/in/cameron-joannidis/
  - Twitter - @camjo89
- Consult across a range of areas and have built many big data and machine learning systems
- Specialise in several areas
  - Big Data / Data Engineering
  - Machine Learning / Data Science
  - Scala / Functional Programming

Agenda

- Abstracting effects from domain definitions
- Abstracting effects with Final Tagless or Free Structures
- Combining Free and FreeAp

Before we start

- I am not advocating that you should be using Free Monad and/or Free Applicative over final tagless. Every team and project is different and both approaches are sufficiently powerful for most requirements
- I'm hoping to introduce you more generally to some more interesting things you can do with Free Structures
- Source Code: https://gist.github.com/camjo/d2dd391b5a44b55d407f041477341242

A Basic Domain

```scala
case class User(name: String, age: Int)

trait UserRepository {
  def createUser(name: String, age: Int): User
  def getUserById(id: String): Option[User]
  def listUsers(): List[User]
}
```

A Basic Implementation

```
class MyUserRepository() extends UserRepository {
  def createUser(name: String, age: Int): User = ???
  def getUserById(id: String): Option[User] = ???
  def listUsers(): List[User] = ???
}
```

Capturing Failure

```scala
trait UserRepository {
  def createUser(name: String, age: Int): Either[Throwable, User]
  def getUserById(id: String): Either[Throwable, Option[User]]
  def listUsers(): Either[Throwable, List[User]]
}
```

Capturing Async Effects

```scala
trait UserRepository {
  def createUser(name: String, age: Int): Task[Either[Throwable, User]]
  def getUserById(id: String): Task[Either[Throwable, Option[User]]]
  def listUsers(): Task[Either[Throwable, List[User]]]
}
```

Combining Multiple Effects

```scala
trait UserRepository {
  def createUser(name: String, age: Int): EitherT[Task, Throwable, User]
  def getUserById(id: String): EitherT[Task, Throwable, Option[User]]
  def listUsers(): EitherT[Task, Throwable, List[User]]
}
```

What Happened To Our Domain?

```scala
trait UserRepository {
    def createUser(name: String, age: Int): User
    def getUserById(id: String): Option[User]
    def listUsers(): List[User]
}

trait UserRepository {
    def createUser(name: String, age: Int): EitherT[Task, Throwable, User]
    def getUserById(id: String): EitherT[Task, Throwable, Option[User]]
    def listUsers(): EitherT[Task, Throwable, List[User]]
}
```

Implementation Effects Domain

```
EitherT[Task, Throwable, User]
```

Problems

- Tightly couples implementation details to domain
- Can not test domain logic in isolation of implementation

Final Tagless

```scala
trait UserRepository[F[_]] {
  def createUser(name: String, age: Int): F[User]
  def getUserById(id: String): F[Option[User]]
  def listUsers(): F[List[User]]
}
```

Final Tagless Implementation

```scala
trait FutureUserRepository extends UserRepository[Task] {
  def createUser(name: String, age: Int): Task[User] = ???
  def getUserById(id: String): Task[Option[User]] = ???
  def listUsers(): Task[List[User]] = ???
}
```

Free (Monad) / FreeAp (Applicative)

- Encode domain as ADT
- Lift domain into Free structure
- Create interpreters to interpret domain instructions with some effect
- Write logic as a Free program
- Interpret Free program to produce output effect

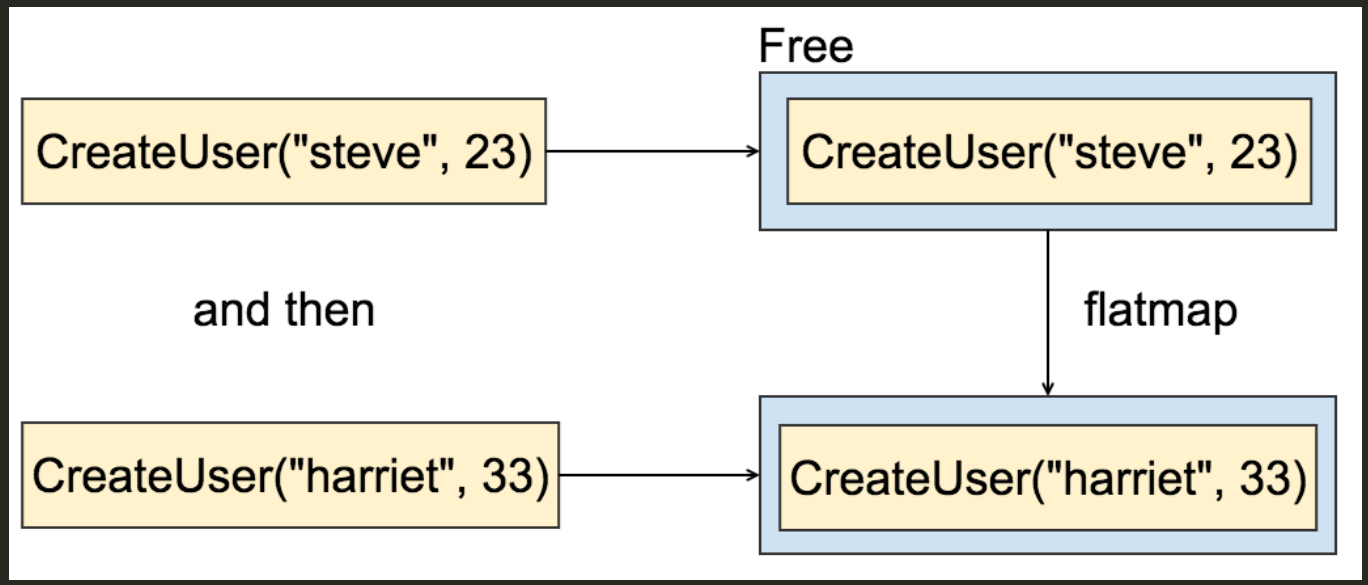
Our Domain As An ADT

```scala
sealed trait UserOperation[T]
case class CreateUser(name: String, age: Int) extends UserOperation[User]
case class GetUserById(id: String) extends UserOperation[Option[User]]
case class ListUsers() extends UserOperation[List[User]]
```

Our Domain As An ADT

```scala
sealed trait F[T]
case class CreateUser(name: String, age: Int) extends F[User]
case class GetUserById(id: String) extends F[Option[User]]
case class ListUsers() extends F[List[User]]
```

Lifting Operations Into Free Monad



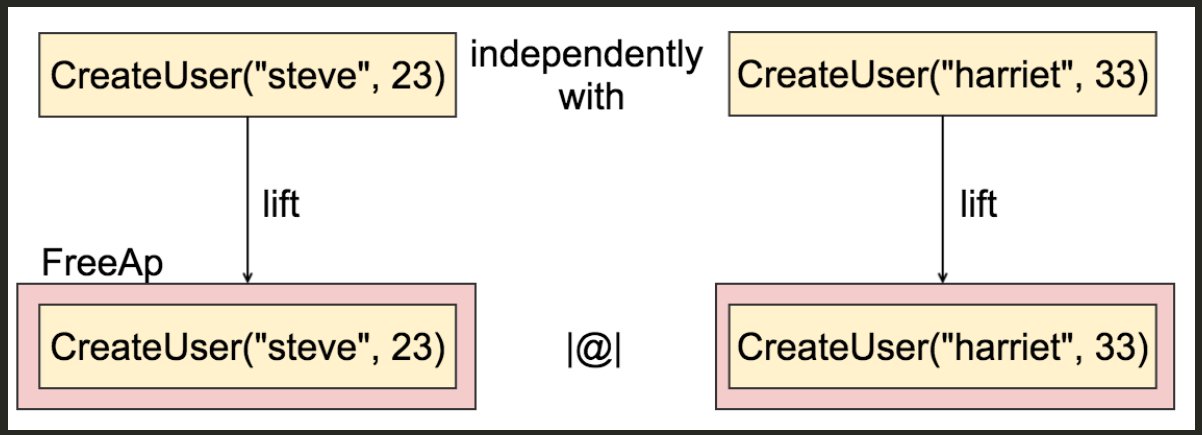
Lifting Our Domain Into Free

```scala
object UserRepository {
  def createUser(name: String, age: Int): Free[UserOperation, User] =
    Free.liftF(CreateUser(name, age))

  def getUserById(id: String): Free[UserOperation, Option[User]] =
      Free.liftF(GetUserById(id: String))

  def listUsers(): Free[UserOperation, List[User]] =
      Free.liftF(ListUsers())
}
```

Lifting Operations Into FreeAp



Lifting Out Domain Into FreeAp

```scala
object UserRepository {
  def createUser(name: String, age: Int): FreeAp[UserOperation, User] =
    FreeAp.lift(CreateUser(name, age))

  def getUserById(id: String): FreeAp[UserOperation, Option[User]] =
      FreeAp.lift(GetUserById(id: String))

  def listUsers(): FreeAp[UserOperation, List[User]] =
      FreeAp.lift(ListUsers())
}
```

A Little Trick

- Lets us defer the choice of using a monad effect type or an applicative effect type until we actually use it
- Can be extended to any Free structure in theory

```scala
case class ExecStrategy[F[_], A](fa: F[A]) {
  val seq: Free[F, A] = Free.liftF(fa)
  val par: FreeAp[F, A] = FreeAp.lift(fa)
}
```

A real example

- Combine multiple domains
- Combine Free (sequential) and FreeAp (parallel)

Two (Simple) Domains

```scala
case class User(name: String, age: Int)

sealed trait UserOperation[T]
case class CreateUser(name: String, age: Int) extends UserOperation[User]

sealed trait AnalyticsOperation[T]
case class AnalyseUser(user: User) extends AnalyticsOperation[Int]
```

Lift Domain

- Handling the combination of multiple domains by making the domain functor generic as F

```
case class UserRepo[F[_]](implicit ev: Inject[UserOperation, F]) {
  def createUser(name: String, age: Int): ExecStrategy[F, User] =
    ExecStrategy[F, User](ev.inj(CreateUser(name, age)))
}
object UserRepo {
  implicit def toUserRepo[F[_]](implicit ev: Inject[UserOperation, F]): UserRepo[F] =
    UserRepo[F]
}
```

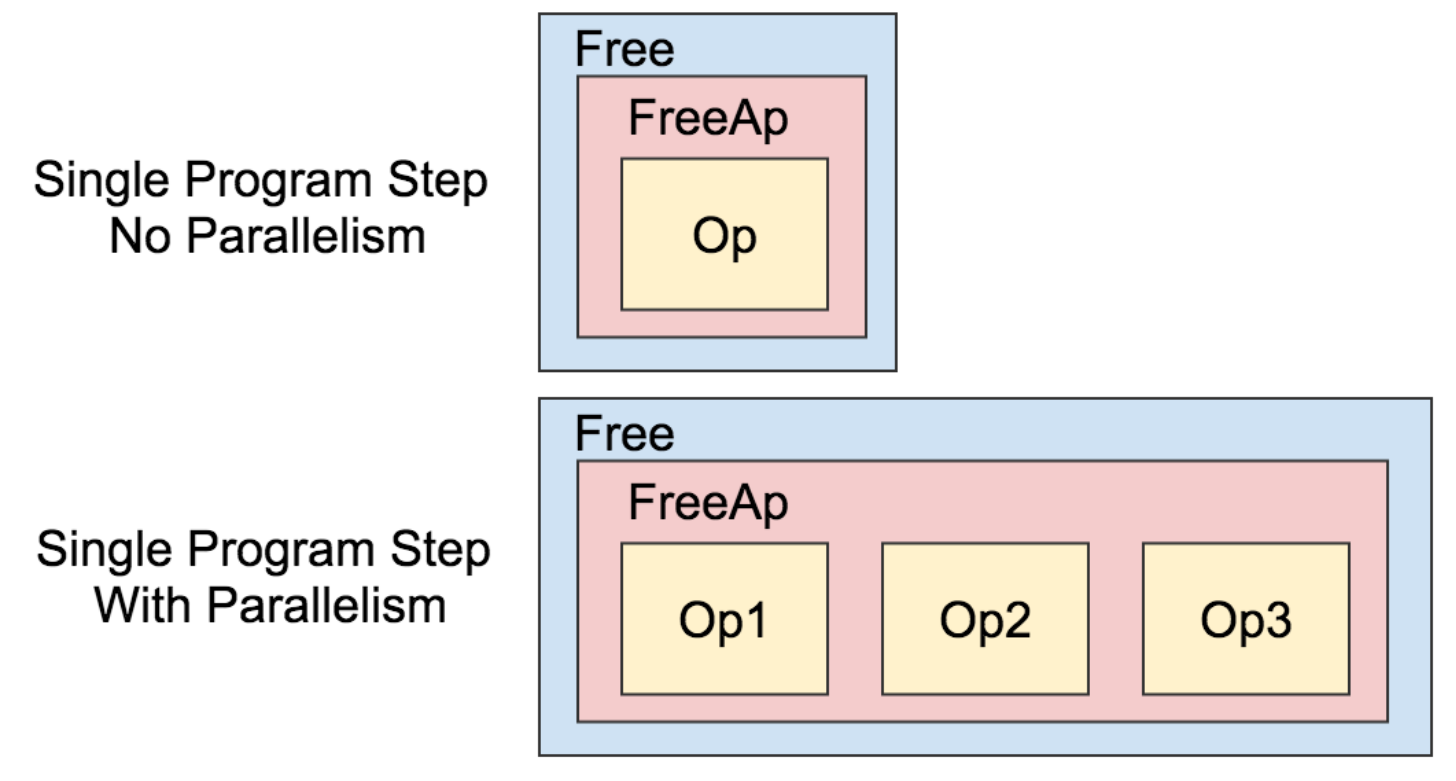
Two Main Choices

- Sequential program of parallel steps
- Parallel program of sequential steps

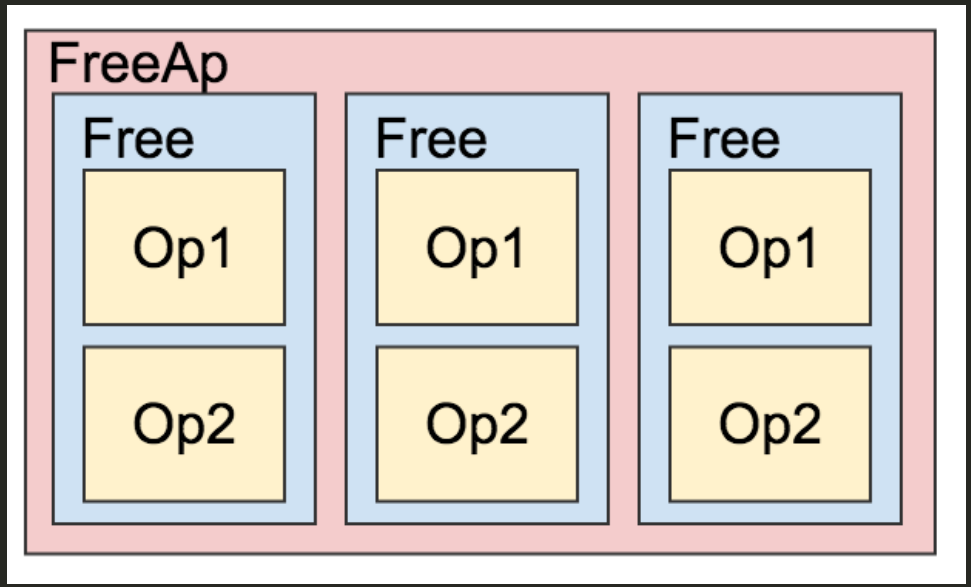
```
type Program[F[_], A] = Free[FreeAp[F, ?], A]
type Program[F[_], A] = FreeAp[Free[F, ?], A]
```

Sequential Program Of Parallel Steps

This is the one we will use for the example

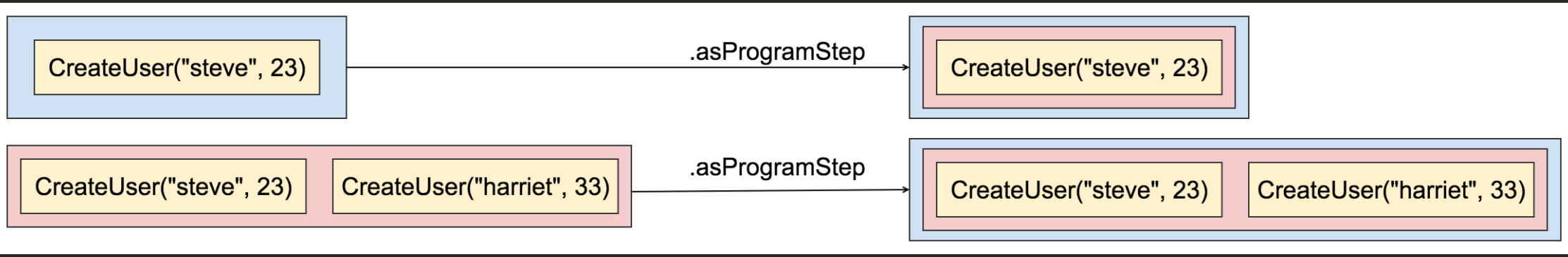


Parallel Program Of Sequential Steps



Lifting Operations Into Program

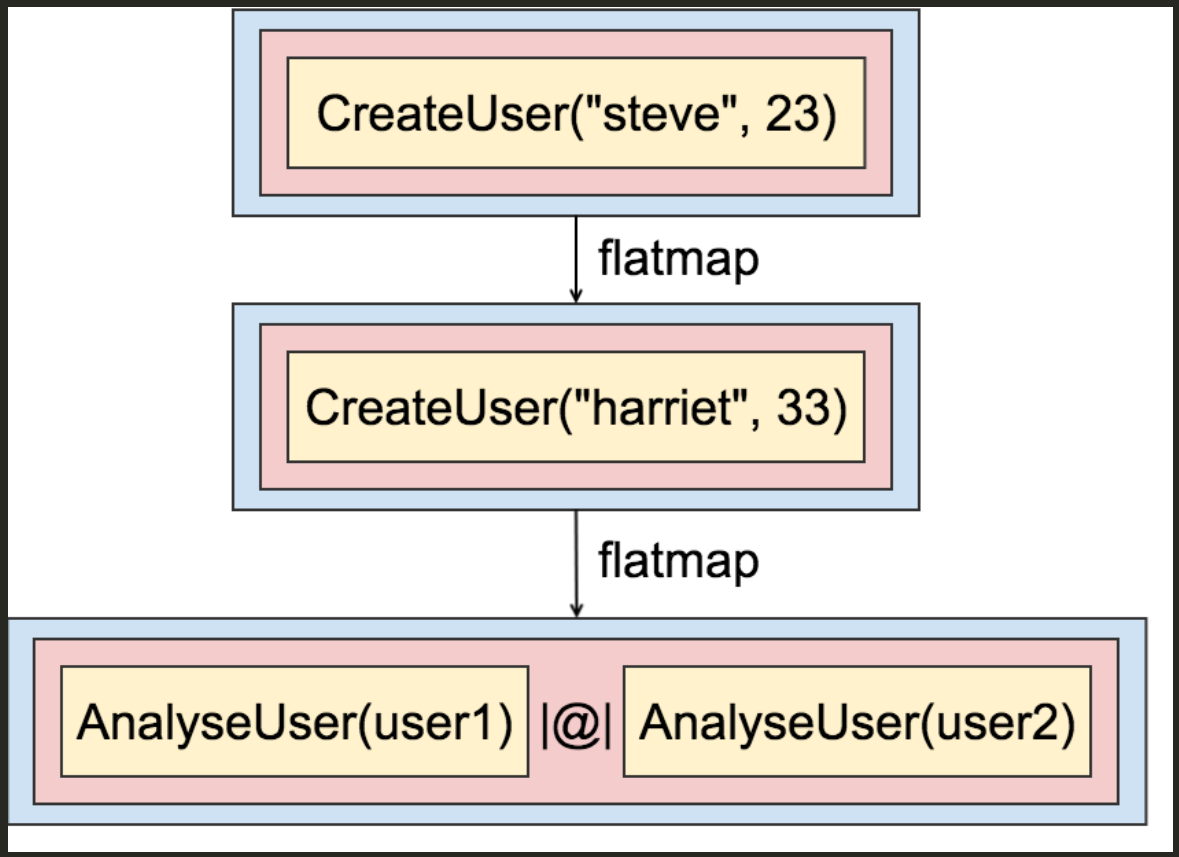
- Free needs to wrap Operation with FreeAp
- FreeAp steps need to be wrapped in Free



Writing Our Program

```
def program[F[_]](implicit userRepo: UserRepo[F],
                  analyticsRepo: AnalyticsRepo[F]): Program[F, Int] = {
  for {
    user1 <- userRepo.createUser("steve", 23).seq.asProgramStep
    user2 <- userRepo.createUser("harriet", 33).seq.asProgramStep
    sumOfAnalytics <- (
        analyticsRepo.analyseUser(user1).par |@|
        analyticsRepo.analyseUser(user2).par
    )((a, b) => a + b).asProgramStep
  } yield sumOfAnalytics
}
```

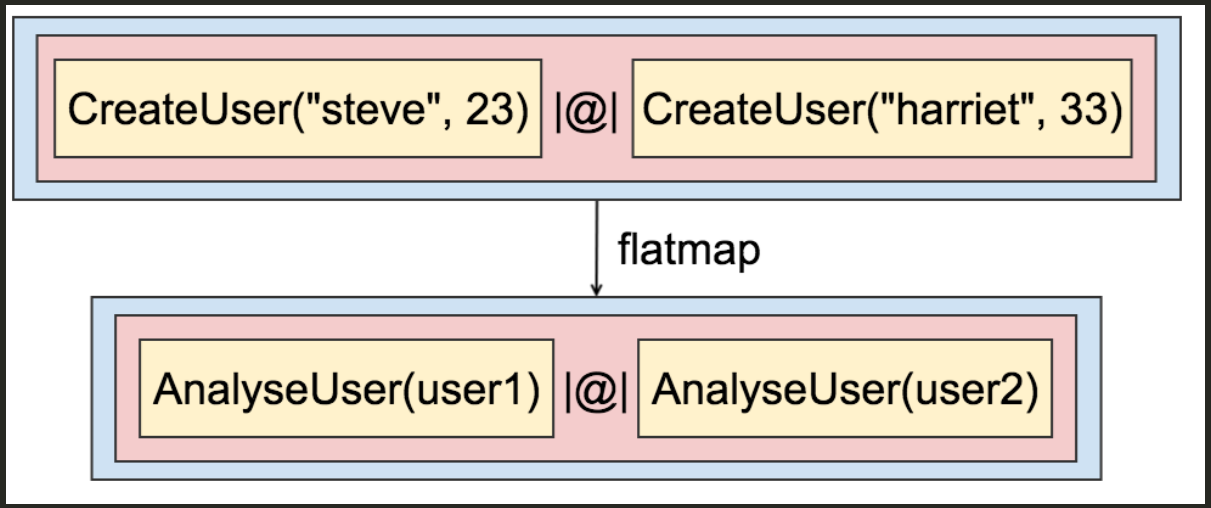
Program Structure



Writing Our Program (Another Way)

```scala
def program2[F[_]](implicit userRepo: UserRepo[F],
                   analyticsRepo: AnalyticsRepo[F]): Program[F, Int] = {
  for {
    users <- (
        userRepo.createUser("steve", 23).par |@|
        userRepo.createUser("harriet", 33).par
    )((u1, u2) => (u1, u2)).asProgramStep
    (user1, user2) = users
    sumOfAnalytics <- (
        analyticsRepo.analyseUser(user1).par |@|
        analyticsRepo.analyseUser(user2).par
    )((a, b) => a + b).asProgramStep
  } yield sumOfAnalytics
}
```

Program structure



Our Interpreter

We make it slow so that we can easily observe the difference in runtime between the sequential and parallel versions

```
object SlowUserInterpreter extends (UserOperation ~> Task) {
  override def apply[A](fa: UserOperation[A]): Task[A] = fa match {
    case CreateUser(name, age) =>
      Task {
        println(s"Creating user $name")
        Thread.sleep(5000)
        println(s"Finished creating user $name")
        User(name, age)
      }
  }
}
```

Our Interpreter

We make it slow so that we can easily observe the difference in runtime between the sequential and parallel versions

```
object SlowAnalyticsInterpreter extends (AnalyticsOperation ~> Task) {
  override def apply[A](fa: AnalyticsOperation[A]): Task[A] = fa match {
    case AnalyseUser(user) =>
      Task {
        println(s"Analysing user $user")
        Thread.sleep(2000)
        println(s"Finished analysing user $user")
        Random.nextInt(50)
      }
  }
}
```

Combining Our Domains

- Define our operations as a common instruction set
- Combine out interpreters to read this instruction set

```
type ProgramInstructions[A] = Coproduct[UserOperation, AnalyticsOperation, A]
val programInterpreter: ProgramInstructions ~> Task =
    SlowUserInterpreter or SlowAnalyticsInterpreter
```

Interpreting Our Program

- Our Free Monads operation type was FreeAp[F, ?]
- Provide a way to turn that into our output effect G

```
case class ParallelInterpreter[G[_]](f: ProgramInstructions ~> G)
                                    (implicit ev: Applicative[G])
    extends (FreeAp[ProgramInstructions, ?] ~> G) {
        override def apply[A](fa: FreeAp[ProgramInstructions, A]): G[A] = fa.foldMap(f)
    }
```

Running the program

```
program[ProgramInstructions]
  .foldMap(ParallelInterpreter(programInterpreter))
  .unsafePerformSync
```

Program Results

```
// Creating user steve
// Finished creating user steve
// Creating user harriet
// Finished creating user harriet
// Analysing user User(harriet,33)
// Analysing user User(steve,23)
// Finished analysing user User(harriet,33)
// Finished analysing user User(steve,23)
```

Things to note

- Target effect must have an applicative instance
- Default Task applicative instance runs sequentially left to right (i.e. not in parallel - since it is also a monad)
- Need to explicitly import parallel applicative instance or create your own in scope (See github code example for how)

How does this look all together?

- Defining your domain ADTs
- Lifting your ADTs into Free Structures
- Writing your program in terms of Free Structures
- Writing Interpreters for each domain ADT
- Run program

Freestyle - Removing the boilerplate

- Freestyle (frees.io) uses macros to remove most of the boilerplate
- Defaults to using the Free[FreeAp[F, ?], A] approach that we used
- Several integrations and library abstractions out of the box
- Out of the box optimisations to reduce the overhead of using many domain algebras

What we've covered

- Why we abstract effects from domain
- Abstracting effects with Final Tagless or Free Structures
- Combining Free and FreeAp to introduce parallelism into our domain
- Freestyle to remove boilerplate

Thank You

Questions?