Composing an App with Free Monads (using Cats)

https://github.com/hermannhueck/free-monad-app

Abstract

In this talk I will explain what Free Monads are and how to use them (using the Cats implementation).

After having shown the basics I build a small app by composing several Free Monads to a small program.

I discuss the pros and cons of this technique.

Finally I will demonstrate how to avoid some boilerplate with the *FreeK* library.

Agenda

1. Free Monads - what they are.
2. Free Monad Recipe
3. Extending your DSL
4. *FunctionK* and Natural Transformation
5. More Interpreters
6. Partial Lifting with *Free.inject*
7. Composing two DSLs and two Interpreters
8. Composing three DSLs and three Interpreters
9. Routing the workflow through DSLs
10. Pros & Cons
11. The *FreeK* library
12. Resources

1. Free Monad - What is it?

Free Monad - What is it?

A free monad is a construction which allows you to build a monad from any ADT with a type parameter. Like other monads, it is a pure way to represent and manipulate computations.

In particular, free monads provide a practical way to:

- represent stateful computations as data, and run them
- run recursive computations in a stack-safe way
- build an embedded DSL (domain-specific language)
- retarget a computation to another interpreter using natural transformations

(https://typelevel.org/cats/datatypes/freemonad.html)

2. Free Monad Recipe

See: *app1.MyApp*

Recipe, how to proceed ...

1. Study your topic/domain/use case. Which operations do you need?
2. Create an ADT (algebraic data type) for the operations (computions as data)
3. "Lift" your ADT into the Free Monad, i.e. use the Free Monad to implement a smart constructor (lowercased function) for each element of your ADT. This is your DSL.
4. Write one or more interpreters for your DSL (using natural transformation)
5. Build a program using the DSL (typically a for comprehension). The program is not executable.
6. Execute the program with one of your interpreters.

Step 1: Study your topic / domain / use case.

Which operations do you need?

In this very simple exmaple we want to interact with a user at a terminal. We want to print an output string to the terminal or get a string of user input from the terminal.

Operations:

- PRINT LINE: displays an output string to the user and returns nothing
- GET LINE: returns an input string entered by the user

Step 2: Create an ADT for the operations

```
// Algebra as an ADT - one type param for the return type
trait Inout[A]
final case class Println(out: String) extends Inout[Unit]
case object Getline extends Inout[String]
`
```

Note: The trait's type parameter represents the return type of an operation.
The *Println* operation returns nothing, hence it extends *Inout[Unit]*.
The *Getline* operation returns a String, hence it extends *Inout[String]*.

The ADT is just **computations as data**. You cannot use them for program execution, you cannot invoke them.

Step 3: "Lift" your ADT into the Free Monad

```
// DSL
def println(out: String): Free[Inout, Unit] = Free.liftF(Println(out))
def getline: Free[Inout, String] = Free.liftF(Getline)
```

Implement a smart constructor (lowercased function) for each element of your ADT. This is your DSL.

Later we will also use *Free.inject* instead of *Free.liftF*.

Step 4: Write an interpreter for the DSL

```
// interpreter
object ConsoleInterpreter extends (Inout ~> Id) {

  override def apply[A](fa: Inout[A]): Id[A] = fa match {
    case Println(out) =>
      println(out)
      () : Id[Unit]
    case Getline =>
      val in = scala.io.StdIn.readLine()
      in : Id[String]
  }
}
```

Every interpreter is/extends a natural transformation from your DSL monad (Inout) to a target monad (Id). The squiggly arrow is a shortcut for the natural transformation *FunctionK*. (More on that later in this presentation)

An Interpreter must implement the apply method and is typically a pattern match over the case classes of the ADT.

You can write more than one interpreter for the same ADT.

Step 5: Build a program using the DSL

```
// program definition (does nothing)
def prog: Free[Inout, (String, Int)] = for {
  _ <- println("What's your name?")
  name <- getline
  _ <- println("What's your age?")
  age <- getline
  _ <- println(s"Hello $name! Your age is $age!")
} yield (name, age.toInt)
```

The program is not executable.

The program can be written with map and flatMap, but is typically a for comprehension written in your DSL.

The programs return type is a *Free[MyADT, RESULT]*.

Step 6: Execute the program

```
// Execute program with ConsoleInterpreter
val result: Id[(String, Int)] = prog.foldMap(ConsoleInterpreter)

println(s"result = $result")
`
```

Use *Free.foldMap(...)* to execute the program with a specific intrpreter.

Note: *Free.foldMap* internally uses a technique called Tranpolining.
Trampolining makes the Free Monads stack-safe. No *StackOverflowError*!

3. Extending your DSL

See: *app2.MyApp*

Extending your DSL

Write a small function as a for comprehension or with map/flatMap.

```
def println(out: String): Free[Inout, Unit] = Free.liftF(Println(out))
def getline: Free[Inout, String] = Free.liftF(Getline)

def ask(prompt: String): Free[Inout, String] = for {
  _ <- println(prompt)
  input <- getline
} yield input

def ask2(prompt: String): Free[Inout, String] = // same with flatMap
  println(prompt).flatMap(_ => getline)
```

This allows you to simplify programs written in this DSL.

```
def prog: Free[Inout, (String, Int)] = for {
  name <- ask("What's your name?")
  age <- ask("What's your age?")
  _ <- println(s"Hello $name! Your age is $age!")
} yield (name, age.toInt)
```

4. *FunctionK* and Natural Transformation

FunctionK

See also: https://typelevel.org/cats/datatypes/functionk.html

Function1[-A, +B] takes an *A* and returns a *B*.
Shortcut: *A => B*

```
trait Function1[-A, +B] {

  def apply(a: A): B
}
```

FunctionK[F[_], G[_]] takes an *F[A]* and returns a *G[A]*.
Shortcut: *F ~> G*

```
trait FunctionK[F[_], G[_]] {

  def apply[A](fa: F[A]): G[A]
}
```

FunctionK

Function1: A --> [A => B] --> B

"hello" --> _.length --> 5

A *Function1* changes a value.

FunctionK: F[A] --> [F ~> G] --> G[B]

List(1, 2) --> _.headOption --> Some(1)

A *FunctionK* changes the context.

Natural Transformation

If the contexts F and G are Functors, the context conversion is called

Natural Transformation or **Functor Transformation**.

FunctionK

In analogy to *Function1* *FunctionK* also provides methods *compose* and *andThen*. It also provides a method 'or' which allows to compose two *FunctionK*s, i.e. two interpreters. (We will use them later.)

```
trait FunctionK[F[_], G[_]] { self =>

  // Applies this functor transformation from `F` to `G`
  def apply[A](fa: F[A]): G[A]

  // Composes two instances of FunctionK into a new FunctionK with this
  // transformation applied last.
  def compose[E[_]](f: FunctionK[E, F]): FunctionK[E, G] = ???

  // Composes two instances of FunctionK into a new FunctionK with this
  // transformation applied first.
  def andThen[H[_]](f: FunctionK[G, H]): FunctionK[F, H] = f.compose(self)

  // Composes two instances of FunctionK into a new FunctionK that transforms
  // a [[cats.data.EitherK]] to a single functor.
  // This transformation will be used to transform left `F` values while
  // `h` will be used to transform right `H` values.
  def or[H[_]](h: FunctionK[H, G]): FunctionK[EitherK[F, H, ?], G] = ???
}
```

See: Usage of *FunctionK.or*

5. More Interpreters

See: *app3.MyApp*

More Interpreters

We can provide several interpreters for the same ADT / DSL.

We can execute a programm written in a DSL with different interpreters for that DSL.

More Interpreters

```scala
object ConsoleInterpreter extends (Inout ~> Id) {
  override def apply[A](fa: Inout[A]): Id[A] = ???
}
object AsyncInterpreter extends (Inout ~> Future) {
  override def apply[A](fa: Inout[A]): Future[A] = ???
}
class TestInterpreter(inputs: ListBuffer[String],
                      outputs: ListBuffer[String]) extends (Inout ~> Id) {
  override def apply[A](fa: Inout[A]): Id[A] = ???
}


def prog: Free[Inout, (String, Int)] = for {
  name <- ask("What's your name?")
  age <- ask("What's your age?")
  _ <- println(s"Hello $name! Your age is $age!")
} yield (name, age.toInt)

val result: Id[(String, Int)] = prog.foldMap(ConsoleInterpreter)

val futureResult: Future[(String, Int)] = prog.foldMap(AsyncInterpreter)

val testResult: Id[(String, Int)] =
                      prog.foldMap(new TestInterpreter(inputs, outputs))
```

6. Partial Lifting with *Free.inject*

See: *app3a.MyApp*

Free.inject instead of *Free.liftF* (1/4)

DSL lifted with *Free.liftF*

```
def println(out: String): Free[Inout, Unit] = Free.liftF(Println(out))
def getline: Free[Inout, String] = Free.liftF(Getline)
def ask(prompt: String): Free[Inout, String] =
  println(prompt).flatMap(_ => getline)
```

DSL partially lifted with *Free.inject*

```
class IoOps[F[_]](implicit IO: InjectK[Inout, F]) {
  def println(out: String): Free[F, Unit] = Free.inject[Inout, F](Println(out)
  def getline: Free[F, String] = Free.inject[Inout, F](Getline)
  def ask(prompt: String): Free[F, String] =
    println(prompt).flatMap(_ => getline)
}

object IoOps {
  // provides an instance of IoOps in implicit scope
  implicit def ioOps[F[_]](
                  implicit IO: InjectK[Inout, F]): IoOps[F] = new IoOps[F]
}
```

Free.inject instead of *Free.liftF* (2/4)

- Instead of providing the DSL functions directly, pack them into a class.
- In the class constructor provide an implicit *InjectK[YourDSL, F]*. (*F[_]* is a place holder for some DSL that we provide later.)
- Implement the DSL functions inside the class with *Free.inject*.
- Implement the DSL extension function (*ask*) also inside the new class.
- Provide an implicit instance of this class inside the companion object of the class (= implicit scope).

This is a bit more boilerplate than before.
But it gives us more flexibility for DSL composition, as we will see later.

Free.inject instead of *Free.liftF* (3/4)

Program for DSL with *Free.liftF*:

```scala
def prog: Free[Inout, (String, Int)] = for {
  name <- ask("What's your name?")
  age <- ask("What's your age?")
  _ <- println(s"Hello $name! Your age is $age!")
} yield (name, age.toInt)
```

Program for DSL with *Free.inject*:

```scala
def prog(implicit io: IoOps[Inout]): Free[Inout, (String, Int)] = for {
  name <- io.ask("What's your name?")
  age <- io.ask("What's your age?")
  _ <- io.println(s"Hello $name! Your age is $age!")
} yield (name, age.toInt)
```

Free.inject instead of *Free.liftF* (4/4)

In the definition of *IoOps* we already defined an *Inout* as the first type parameter of *InjectK[Inout, F]*. Here in the program definition we replace the place holder DSL *F* with the higher kinded type of another DSL, which in this case is also Inout. We have composed *Inout* with another *Inout*.

The benefit of this technique becomes obvious shortly.
We will create one composed DSL out of two different component DSLs.

7. Composing two DSLs and two Interpreters

See: *app4.MyApp*

Two DSLs

Inout

```
trait Inout[A]
final case class Println(out: String)
        extends Inout[Unit]
final case object Getline
        extends Inout[String]



class IoOps[F[_]](
    implicit IO: InjectK[Inout, F]) {
  def println(out: String) =
        Free.inject(Println(out))
  def getline = Free.inject(Getline)
  def ask(prompt: String) =
    println(prompt)
      .flatMap(_ => getline)
}

object IoOps {
  implicit def ioOps[F[_]](
    implicit IO: InjectK[Inout, F]) =
      new IoOps[F]
}
```

KVStore

```
trait KVStore[A]
final case class Put(key: String,
    value: Int) extends KVStore[Unit]
final case class Get(key: String
    ) extends KVStore[Option[Int]]
final case class Delete(key: String
    ) extends KVStore[Option[Int]]

class KVSOps[F[_]](
      implicit KV: InjectK[KVStore, F])
  def put(key: String, value: Int) =
    Free.inject(Put(key: String, value:
  def get(key: String) =
    Free.inject(Get(key: String))
  def delete(key: String) =
    Free.inject(Delete(key: String))
}

object KVSOps {
  implicit def kvsOps[F[_]](
    implicit IO: InjectK[KVStore, F]) =
      new KVSOps[F]
}
```

Two Interpreters

Inout

```
object ConsoleInterpreter
            extends (Inout ~> Id) {


  override def apply[A](fa: Inout[A]
          ): Id[A] = fa match {

    case Println(out) =>
      println(out)
      (): Id[Unit]

    case Getline =>
      val in = scala.io.StdIn.readLine()
      in: Id[String]



  }
}
```

KVStore

```
object KVSInterpreter
            extends (dsl.KVStore ~> Id)

  var kvs: Map[String, Int] = Map.empty

  override def apply[A](fa: KVStore[A]
          ): Id[A] = fa match {

    case Put(key, value) =>
      kvs = kvs.updated(key, value)
      (): Id[Unit]

    case Get(key) =>
      kvs.get(key): Id[Option[Int]]

    case Delete(key) =>
      val value = kvs.get(key)
      kvs = kvs - key
      value: Id[Option[Int]]
  }
}
```

Note: Both interpreters have the same target type: *Id*

Composing DSLs and interpreters

```
type AppDSL[A] = EitherK[Inout, KVStore, A]

def prog(implicit io: IoOps[AppDSL],
                  kvs: KVSOps[AppDSL]): Free[AppDSL, (String, Option[Int])] = {
  for {
    name <- io.ask("What's your name?")
    age <- io.ask("What's your age?")
    _ <- kvs.put(name, age.toInt)
    _ <- io.printline(s"Hello $name! Your age is $age!")
    optAge <- kvs.get(name)
  } yield (name, optAge)
}

val composedInterpreter: AppDSL ~> Id =
            (ConsoleInterpreter: Inout ~> Id) or (KVSInterpreter: KVStore ~> Id)
val result: Id[(String, Option[Int])] = prog.foldMap(composedInterpreter)
```

- Define a type alias for an *EitherK* with two ADTs
- Provide DSLs as implicit parameters to your program
- The two component interpreters must have the same target type as the composed interpreter (*Id* in our case).
- The composition order of interpreters must be the same as the composition order of the DSLS.

  See: Definition of *FunctionK*

EitherK (= Coproduct)

Either is parameterized with two types *A* and *B*.
A is the type of the *Left*, *B* the type of the *Right*.

```
sealed abstract class Either[+A, +B] ... { ... }
```

EitherK is parameterized with two type constructors *F[_]* and *G[_]* and a regular type *A*. It's a case class wrapping a value called *run* of type *Either[F[A], G[A]]*.

```
final case class EitherK[F[_], G[_], A](run: Either[F[A], G[A]]) {
  // ...
}
```

EitherK is used to define a composed DSL.

In our example we define *AppDSL* as an *EitherK*:

```
type AppDSL[A] = EitherK[Inout, KVStore, A]
```

Partial Lifting with *InjectK* (1/2)

InjectK is used for partial lifting into a Free Monad for the composed DSL.

In class IoOps *Free.inject* internally injects the *Inout* into *InjectK[Inout, F]*, where the place holder F will be replaced by *AppDSL*.

In class *KVSOps Free.inject* internally injects the *KVStore* into *InjectK[KVStore, F]*, where the place holder *F* will be replaced by *AppDSL*.

```
// simplified def of inject
object Free {
  def inject[F[_], G[_], A](fa: F[A])(implicit I: InjectK[F, G]): Free[G, A] =
          liftF(I.inj(fa))
}
```

Partial Lifting with *InjectK* (2/2)

```
class IoOps[F[_]](implicit IO: InjectK[Inout, F]) {
  def println(out: String): Free[F, Unit] =
                           Free.inject[Inout, F](Println(out))
  // ...
}

class KVSOps[F[_]](implicit KV: InjectK[KVStore, F]) {
  def get(key: String): Free[F, Option[Int]] =
                           Free.inject[KVStore, F](Get(key: String))
  // ...
}

type AppDSL[A] = EitherK[Inout, KVStore, A]

def prog(implicit io: IoOps[AppDSL],
                  kvs: KVSOps[AppDSL]): Free[AppDSL, (String, Option[Int])] = ???
```

In the implicit parameters we fill the place holder *F* with the concrete type *AppDSL* (= *EitherK[Inout, KVStore, ?]*).

For more details see: *app4a.MyApp*

8. Composing three DSLs and three Interpreters

See: *app5.MyApp*

Third DSL and Interpreter

Our 3rd DSL and corresponding interpreter(s) is for monotonic sequence number generation.

```
sealed trait Sequence[A]
case object NextId extends Sequence[Long]

class SeqOps[F[_]](implicit KV: InjectK[Sequence, F]) {
  def nextId: Free[F, Long] = Free.inject[Sequence, F](NextId)
  def nextStringId: Free[F, String] = nextId.map(_.toString)
}

object SeqOps {
  implicit def seqOps[F[_]](implicit IO: InjectK[Sequence, F]): SeqOps[F] =
        new SeqOps[F]
}
```

Composing three DSLs and three Interpreters

- Compose 3 DSLs with 2 type aliases.
- Compose interpreters in the same order as the DSLs.
- All component interpreters must have the same target type as the composed interpreter (*Id* in our case).

```scala
type AppDSL0[A] = EitherK[Inout, KVStore, A]
type AppDSL[A] = EitherK[Sequence, AppDSL0, A]

def prog(implicit io: IoOps[AppDSL],
         kvs: KVSOps[AppDSL],
         seq: SeqOps[AppDSL]): Free[AppDSL, (String, Option[Cat])] = {
  for {
    name <- io.ask("What's your name?")
    age <- io.ask("What's your age?")
    id <- seq.nextId.map(_.toString)
    _ <- kvs.put(id, Cat(id, name, age.toInt))
    _ <- io.println(s"Hello cat $name! Your age is $age!")
    optCat <- kvs.get(id)
  } yield (id, optCat)
}

// compose interpreters in the same order as DSLs
val appInterpreter: AppDSL ~> Id =
        SeqInterpreter or (ConsoleInterpreter or KVSInterpreter)
val result: Id[(String, Option[Cat])] = prog.foldMap(appInterpreter)
```

9. Routing the workflow through DSLs

See: *app6.MyApp*

Logging DSL

In the next evolution step of the program we create a new DSL for logging and the corresponding interpreters(s). (interpreter code not shown here)

```
sealed trait Log[A] extends Product with Serializable
final case class Info(msg: String) extends Log[Unit]
final case class Warn(msg: String) extends Log[Unit]
final case class Error(msg: String) extends Log[Unit]

class LogOps[F[_]](implicit LG: InjectK[Log, F]) {
  def info(msg: String): Free[F, Unit] = Free.inject[Log, F](Info(msg))
  def warn(msg: String): Free[F, Unit] = Free.inject[Log, F](Warn(msg))
  def error(msg: String): Free[F, Unit] = Free.inject[Log, F](Error(msg))
}

object LogOps {
  implicit def logOps[F[_]](implicit LG: InjectK[Log, F]): LogOps[F] =
        new LogOps[F]
}
```

Routing the workflow through DSLs (1/2)

- We create a composed DSL from *KVStore*, *Sequence* and *Logging*.
- We create new DSL *CatManagement* (cats management business logic).
- The *CatManagement* interpreter is implemented with the above composed interpreter. It routes requests to the other interpreters.
- The main program is implemented in a DSL composed of *Inout* and *CatManagement*.

Routing the workflow through DSLs (2/2)

```
                -----------------
                | AppDSL |
                -----------------
                      |
        -----------------------------              Composition
        |                            |
  ------------          -----------------------------
  | Inout |             | CatManagement |
  ------------          -----------------------------
                                     |             Routing
                         ---------------------------
                         | ComposedDSL |
                         ---------------------------
                                     |
  -----------------------------------------------------    Composition
  |                          |                       |
-----------------       -----------------       ------------------
| Logging |             | KVStore |             | Sequence |
-----------------       -----------------       ------------------
```

Composing a DSL from *KVStore*, *Sequence* and *Logging*

We compose DSLs and interpreters as we did before.

Component and composed interpreters again with same target type: *Id*

```
// component interpreters, all having target type Id
object LogInterpreter extends (Log ~> Id) { ... }
object SeqInterpreter extends (Sequence ~> Id) { ... }
object KvsInterpreter extends (KVStore ~> Id) { ... }

// compose DSLs
type ComposedDSL0[A] = EitherK[Sequence, KVStore, A]
type ComposedDSL[A] = EitherK[Log, ComposedDSL0, A]

// composed interpreter also with target type Id
val ComposedLogSeqKvsInterpreter: ComposedDSL ~> Id =
        LogInterpreter or (SeqInterpreter or KvsInterpreter)
```

Lifting the composed DSL into a Free Monad

We need a monad for the natural transformation from the composed DSL to *Id*. By creating a type alias we lift *ComposedDSL* into the Free Monad. Then we define a new interpreter that translates from *FreeComposed ~> Id* by *foldMap*ping the interpreter we composed before.

```
// type alias for the Free Monad of the composed DSL
type FreeComposed[A] = Free[ComposedDSL, A]

// interpreter that translated from the composed Free Monad to the Id Monad
object FreeComposedLogSeqKvsInterpreter extends (FreeComposed ~> Id) {
  override def apply[A](fa: FreeComposed[A]): Id[A] =
              fa.foldMap(ComposedLogSeqKvsInterpreter)
}
```

CatManagement DSL

```
sealed trait CatManagement[A] extends Product with Serializable
final case class Create(cat: Cat) extends CatManagement[Cat]
final case class UpdateById(cat: Cat) extends CatManagement[Cat]
final case class DeleteById(id: String) extends CatManagement[Boolean]
final case class FindById(id: String) extends CatManagement[Option[Cat]]
final case class FindByName(name: String) extends CatManagement[List[Cat]]
case object FindAll extends CatManagement[List[Cat]]

class CatOps[F[_]](implicit KV: InjectK[CatManagement, F]) {
  def create(cat: Cat): Free[F, Cat] = Free.inject[CatManagement, F](Create(cat))
  def updateById(cat: Cat): Free[F, Cat] = Free.inject[CatManagement, F](UpdateByI
  def deleteById(id: String): Free[F, Boolean] = Free.inject[CatManagement, F](Del
  def findById(id: String): Free[F, Option[Cat]] = Free.inject[CatManagement, F](F
  def findByName(name: String): Free[F, List[Cat]] = Free.inject[CatManagement, F]
  def findAll: Free[F, List[Cat]] = Free.inject[CatManagement, F](FindAll)
}

object CatOps {
  implicit def catOps[F[_]](implicit CM: InjectK[CatManagement, F]): CatOps[F] =
          new CatOps[F]
}
```

CatLogicInterpreter (1/2)

CatLogicInterpreter transforms from *CatManagement ~> FreeComposed* and is implemented with the DSL composed from *Logging*, *Sequence* and *KVStore*.

```
class CatLogicInterpreter(implicit log: LogOps[ComposedDSL],
                                   seq: SeqOps[ComposedDSL],
                                   kvs: KVSOps[ComposedDSL])
                                   extends (CatManagement ~> FreeComposed) {

  override def apply[A](fa: CatManagement[A]): FreeComposed[A] = fa match {

    case Create(cat) =>
      kvsCreate(cat): FreeComposed[Cat]
    case UpdateById(cat) =>
      kvsUpdateById(cat): FreeComposed[Cat]
    case DeleteById(id) =>
      kvsDeleteById(id): FreeComposed[Boolean]
    case FindById(id) =>
      kvsFindById(id): FreeComposed[Option[Cat]]
    case FindByName(name) =>
      kvsFindByName(name): FreeComposed[List[Cat]]
    case FindAll =>
      kvsFindAll: FreeComposed[List[Cat]]
  }

  // ...
```

CatLogicInterpreter (2/2)

```scala
// ...

private def kvsFindAll[A]: FreeComposed[List[Cat]] =
  kvs.getAll

private def kvsFindById[A](id: String): FreeComposed[Option[Cat]] =
  kvs.get(id)

private def kvsFindByName[A](name: String): FreeComposed[List[Cat]] =
  kvs.getAll.map(_.filter(_.name == name))

private def kvsCreate[A](cat: Cat): FreeComposed[Cat] =
  for {
    maybeCat <- kvs.get(cat.id)
    _ = if (maybeCat.isDefined) {
      val message = s"cat with id ${cat.id} already exists"
      log.error(message)
      throw new RuntimeException(message)
    }
    newId <- seq.nextStringId
    _ <- kvs.put(newId, cat.copy(id = newId))
    newMaybeCat <- kvs.get(newId)
    _ <- log.info(s"Created: $cat")
  } yield newMaybeCat.get
```

Routing from one interpreter to the next

CatLogicInterpreter provides a natural transformation (*CatManagement ~> Id*). It transforms (*CatManagement ~> FreeComposed*) ***andThen*** propagates to *FreeComposedInterpreter* which transforms (*FreeComposed ~> Id*).

```
// Routing with FunctionK.andThen
val CatLogicInterpreter: CatManagement ~> Id =
              new CatLogicInterpreter andThen FreeComposedLogSeqKvsInterpreter
```

See: Definition of *FunctionK*

Program definition and execution

Technically nothing new here (just DSL composition).

```
type AppDSL[A] = EitherK[CatManagement, Inout, A]

def prog(implicit io: Inouts[AppDSL],
          co: CatOps[AppDSL]): Free[AppDSL, Option[Cat]] = {
  for {
    name <- io.ask("Cat's name?")
    age <- io.ask("Cat's age?")
    cat <- co.create(Cat(name, age.toInt))
    newAge <- io.ask("That was a lie! Tell me the correct age!")
    _ <- co.updateById(cat.copy(age = newAge.toInt))
    _ <- io.printline(s"Hello cat ${cat.name}! Your age is ${cat.age}!")
    optCat <- co.findById(cat.id)
  } yield optCat
}

val result: Id[Option[Cat]] =
        prog1.foldMap(CatLogicInterpreter or ConsoleInterpreter)
println(s"result = $result")
```

10. Pros & Cons

See the following video presentations:

Rather on the Pro side:
Chris Myers' Talk on Free Monads at Typelevel Summit Oslo, 2016
Rather on the Cons side:
Kelly Robinson's Talk on Free Monads at Scala Days Berlin, 2016

Pros

Pros

- FMs let us create a Monad from any (parameterized) ADT.

Pros

- FMs let us create a Monad from any (parameterized) ADT.
- FMs let us write programs in monadic style (for comprehensions)

Pros

- FMs let us create a Monad from any (parameterized) ADT.
- FMs let us write programs in monadic style (for comprehensions)
- Decoupled program description and execution/interpretation

Pros

- FMs let us create a Monad from any (parameterized) ADT.
- FMs let us write programs in monadic style (for comprehensions)
- Decoupled program description and execution/interpretation
- It is easy to add new ADTs/DSLs.

Pros

- FMs let us create a Monad from any (parameterized) ADT.
- FMs let us write programs in monadic style (for comprehensions)
- Decoupled program description and execution/interpretation
- It is easy to add new ADTs/DSLs.
- It is easy to add new interpreters for existing ADTs/DSLs.

Pros

- FMs let us create a Monad from any (parameterized) ADT.
- FMs let us write programs in monadic style (for comprehensions)
- Decoupled program description and execution/interpretation
- It is easy to add new ADTs/DSLs.
- It is easy to add new interpreters for existing ADTs/DSLs.
- E.g.: Implement different interpreters for prod and test.

Pros

- FMs let us create a Monad from any (parameterized) ADT.
- FMs let us write programs in monadic style (for comprehensions)
- Decoupled program description and execution/interpretation
- It is easy to add new ADTs/DSLs.
- It is easy to add new interpreters for existing ADTs/DSLs.
- E.g.: Implement different interpreters for prod and test.
- FMs are stack-safe (due to an internally used technique: Trampolining).

Cons

Cons

- Advanced technology not easily understood.

Cons

- Advanced technology not easily understood.
- Danger of over-enginieering

Cons

- Advanced technology not easily understood.
- Danger of over-enginieering
- Possible performance cost

Cons

- Advanced technology not easily understood.
- Danger of over-enginieering
- Possible performance cost

To minimize drawbacks:

Cons

- Advanced technology not easily understood.
- Danger of over-enginieering
- Possible performance cost

To minimize drawbacks:

- Be aware of the skill level of your team maintaining the code.

Cons

- Advanced technology not easily understood.
- Danger of over-enginieering
- Possible performance cost

To minimize drawbacks:

- Be aware of the skill level of your team maintaining the code.
- Consider using FMs only in library code or in well encapsulated modules.

Principle of Least Power

Given a choice of solutions,
pick the least powerful solution
capable of solving your problem.

-- Li Haoyi

11. Easing Free Monads with *FreeK*

See: *app5freek.MyApp*

FreeK - What is it?

It is a library, implemented with Shapeless,
designed to remove some of the boilerplate from your Free Monad code.

- Write only ADTs, no lifting, no injecting
- Simplified composition of DSLs
- Simplified composition of interpreters

FreeK Example: *app5freek.MyApp* (1/2)

```scala
// Algebra as an ADT
trait Inout[A]
final case class Println(out: String) extends Inout[Unit]
final case object Getline extends Inout[String]

// DSL to be omitted. No lifting or injecting needed with Freek

// ask is a small subroutine written with FreeK
type PRG = Inout :|: NilDSL
val prg = DSL.Make[PRG]

def ask(prompt: String): Free[prg.Cop, String] = for {
  _ <- Println(prompt).freek[PRG]
  input <- Getline.freek[PRG]
} yield input

def ask2(prompt: String): Free[prg.Cop, String] =
  Println(prompt).freek[PRG].flatMap(_ => Getline.freek[PRG])
```

FreeK Example: *app5freek.MyApp* (2/2)

```
type AppDSL = Inout :|: KVStore :|: Sequence :|: NilDSL
val appDSL = DSL.Make[AppDSL] // infer the right coproduct for AppDSL

def prog: Free[appDSL.Cop, (String, Option[Cat])] = {
  for {
    name <- ask("What's your name?").expand[AppDSL] // ask must be expanded
    age <- ask("What's your age?").expand[AppDSL]   // to Println/Getline
    idLong <- NextId.freek[AppDSL]          // freek performs the heavy lifting
    id = idLong.toString
    _ <- Put(id, Cat(id, name, age.toInt)).freek[AppDSL]
    _ <- Println(s"Hello cat $name! Your age is $age!").freek[AppDSL]
    optCat <- Get(id).freek[AppDSL]
  } yield (id, optCat)
}

// program execution with foldMap or with interpret
val composedInterpreter =  ConsoleInterpreter :&: KVSInterpreter :&: SeqInterprete
// foldMap is order-sensitive
val result1: Id[(String, Option[Cat])] = prog.foldMap(composedInterpreter.nat)
println(s"result1 = $result1")
// interpret is order-agnostic
val result2: Id[(String, Option[Cat])] = prog.interpret(composedInterpreter)
println(s"result2 = $result2")
```

FreeK - Howto in a nutshell

FreeK - Howto in a nutshell

- Write your ADTs as before

FreeK - Howto in a nutshell

- Write your ADTs as before
- Remove all the DSL generation code using *Free.liftF* and *Free.inject*.

FreeK - Howto in a nutshell

- Write your ADTs as before
- Remove all the DSL generation code using *Free.liftF* and *Free.inject*.
- Write your interpreters as before.

FreeK - Howto in a nutshell

- Write your ADTs as before
- Remove all the DSL generation code using *Free.liftF* and *Free.inject*.
- Write your interpreters as before.
- Type alias composed *AppDSL* with the *:|:* operator with *NilDSL* at the end.

FreeK - Howto in a nutshell

- Write your ADTs as before
- Remove all the DSL generation code using *Free.liftF* and *Free.inject*.
- Write your interpreters as before.
- Type alias composed *AppDSL* with the *:|:* operator with *NilDSL* at the end.
- The program doesn't need implicit parameters.

FreeK - Howto in a nutshell

- Write your ADTs as before
- Remove all the DSL generation code using *Free.liftF* and *Free.inject*.
- Write your interpreters as before.
- Type alias composed *AppDSL* with the *:|:* operator with *NilDSL* at the end.
- The program doesn't need implicit parameters.
- "Invoke" the uppercase case class constructors (*Getline* instead of *getline*).

FreeK - Howto in a nutshell

- Write your ADTs as before
- Remove all the DSL generation code using *Free.liftF* and *Free.inject*.
- Write your interpreters as before.
- Type alias composed *AppDSL* with the *:|:* operator with *NilDSL* at the end.
- The program doesn't need implicit parameters.
- "Invoke" the uppercase case class constructors (*Getline* instead of *getline*).
- Append an invocation of *freek[AppDSL]* (for the "heavy lifting").

FreeK - Howto in a nutshell

- Write your ADTs as before
- Remove all the DSL generation code using *Free.liftF* and *Free.inject*.
- Write your interpreters as before.
- Type alias composed *AppDSL* with the *:|:* operator with *NilDSL* at the end.
- The program doesn't need implicit parameters.
- "Invoke" the uppercase case class constructors (*Getline* instead of *getline*).
- Append an invocation of *freek[AppDSL]* (for the "heavy lifting").
- Append *expand[AppDSL]* to your own subroutines (such as *ask*).

FreeK - Howto in a nutshell

- Write your ADTs as before
- Remove all the DSL generation code using *Free.liftF* and *Free.inject*.
- Write your interpreters as before.
- Type alias composed *AppDSL* with the *:|:* operator with *NilDSL* at the end.
- The program doesn't need implicit parameters.
- "Invoke" the uppercase case class constructors (*Getline* instead of *getline*).
- Append an invocation of *freek[AppDSL]* (for the "heavy lifting").
- Append *expand[AppDSL]* to your own subroutines (such as *ask*).
- Compose interpreters with the *:&:* operator.

FreeK - Howto in a nutshell

- Write your ADTs as before
- Remove all the DSL generation code using *Free.liftF* and *Free.inject*.
- Write your interpreters as before.
- Type alias composed *AppDSL* with the *:|:* operator with *NilDSL* at the end.
- The program doesn't need implicit parameters.
- "Invoke" the uppercase case class constructors (*Getline* instead of *getline*).
- Append an invocation of *freek[AppDSL]* (for the "heavy lifting").
- Append *expand[AppDSL]* to your own subroutines (such as *ask*).
- Compose interpreters with the *:&:* operator.
- Program execution: use *prog.foldMap(myInterpreter.nat)*. *foldMap* is order-sensitive.

FreeK - Howto in a nutshell

- Write your ADTs as before
- Remove all the DSL generation code using *Free.liftF* and *Free.inject*.
- Write your interpreters as before.
- Type alias composed *AppDSL* with the *:|:* operator with *NilDSL* at the end.
- The program doesn't need implicit parameters.
- "Invoke" the uppercase case class constructors (*Getline* instead of *getline*).
- Append an invocation of *freek[AppDSL]* (for the "heavy lifting").
- Append *expand[AppDSL]* to your own subroutines (such as *ask*).
- Compose interpreters with the *:&:* operator.
- Program execution: use *prog.foldMap(myInterpreter.nat)*. *foldMap* is order-sensitive.
- Or: Use *prog.interpret(myInterpreter)* and omit the appendix *.nat*. *interpret* is order-agnostic.

FreeK - Howto in a nutshell

- Write your ADTs as before
- Remove all the DSL generation code using *Free.liftF* and *Free.inject*.
- Write your interpreters as before.
- Type alias composed *AppDSL* with the *:|:* operator with *NilDSL* at the end.
- The program doesn't need implicit parameters.
- "Invoke" the uppercase case class constructors (*Getline* instead of *getline*).
- Append an invocation of *freek[AppDSL]* (for the "heavy lifting").
- Append *expand[AppDSL]* to your own subroutines (such as *ask*).
- Compose interpreters with the *:&:* operator.
- Program execution: use *prog.foldMap(myInterpreter.nat)*. *foldMap* is order-sensitive.
- Or: Use *prog.interpret(myInterpreter)* and omit the appendix *.nat*. *interpret* is order-agnostic.

Detailed info about *FreeK* at: https://github.com/ProjectSeptemberInc/freek

12. Resources

Resources (1/3) - basic

- Code and Slides of this Talk:
  https://github.com/hermannhueck/free-monad-app
- Cats documentation on Free Monads:
  https://typelevel.org/cats/datatypes/freemonad.html
- Blog post on Free Monads by Pere Villega:
  http://perevillega.com/understanding-free-monads
- Blog post on FreeK by Pere Villega:
  http://perevillega.com/freek-and-free-monads
- The FreeK project on Github
  https://github.com/ProjectSeptemberInc/freek

Resources (2/3) - basic

- "A Year living Freely"
  Chris Myers' Talk on Free Monads at Typelevel Summit Oslo, 2016
  https://www.youtube.com/watch?v=rK53C-xyPWw
- "Why the free Monad isn't free"
  Kelly Robinson's Talk on Free Monads at Scala Days Berlin, 2016
  https://www.youtube.com/watch?v=U0lK0hnbc4U
- "Composable application architecture with reasonably priced monads"
  Runar Bjarnason's Talk on Free Monads at Scala Days Berlin, 2014
  https://www.youtube.com/watch?v=M258zVn4m2M

Resources (3/3) - advanced

- Cats documentation on FunctionK:
  https://typelevel.org/cats/datatypes/functionk.html
- Cats documentation on Free Applicatives:
  https://typelevel.org/cats/datatypes/freeapplicative.html
- "Free as in Monads" - Daniel Spiewak implements Free Monads
  Daniel Spiewak's live coding session at Northeast Scala Symposium, 2017
  https://www.youtube.com/watch?v=H28QqxO7Ihc
- "Move Over Free Monads: Make Way for Free Applicatives!"
  John de Goes' Talk on Free Applicatives at Scala World, 2015
  https://www.youtube.com/watch?v=H28QqxO7Ihc

Thanks for Listening

Q & A

https://github.com/hermannhueck/free-monad-app