Compositional Application Architecture With Reasonably Priced Monads

Rúnar Bjarnason
@runarorama
Scala Days, Berlin, 2014

Monad coproducts

- Subsumes "dependency injection", monad transformers
- Based on *Data Types à la Carte* (Swiestra)
- Similar to *Extensible Effects* (Kiselyov, Sabry, Swords)

Bank transfer

```
User must be authorized.
If insufficient funds,
   log the attempt,
   raise an error.
Otherwise update both accounts.
```

Concerns

- Authorization
- User interaction
- Logging
- Handling errors
- Persistent storage

Meta-concerns

- Testing
- Reuse
- Extensibility
- Compositionality

“Dependency injection”

```
def transfer(amount: Long, from: Account, to: Account,
             user: User, auth: Authorization,
             log: Logger, err: ErrorHandler,
             store: Storage): Unit
```

“Dependency injection”

```
def transfer(amount: Long, from: Account, to: Account,
             user: User, context: Authorization with
Logger with ErrorHandler with
Storage): Unit
```

Idea

Return a description of what we want to do.

Coproduct

```
def transfer(amount: Long, from: Account,
             to: Account, user: User): List[Instruction]
```

Where `Instruction` is the *coproduct* of:

- Log something
- Fail with an error
- Authorize a user to do something
- Read from storage
- Write to storage
- Interact with the user

User interaction

```
sealed trait Interact[A]

case class Ask(prompt: String)
  extends Interact[String]

case class Tell(msg: String)
  extends Interact[Unit]
```

User interaction

```
val prg =
  List(Ask("What's your first name?"),
  Ask("What's your last name?"),
  Tell("Hello, ???"))
```

User interaction

```
val prg = for {
  x <- Ask("What's your first name?")
  y <- Ask("What's your last name?")
  _ <- Tell(s"Hello, $x $y!")
} yield ()
```

Monads

```
trait Monad[M[_]] {
  def pure[A](a: A): M[A]
  def flatMap[A,B](ma: M[A])(f: A => M[B]): M[B]
}
```

Monads

```
trait Monad[M[_]] {
  def return[A](a: A): M[A]
  def bind[A,B](ma: M[A])(f: A => M[B]): M[B]
}
```

Can we write bind/return?

```
sealed trait Interact[A]

case class Ask(prompt: String)
  extends Interact[String]

case class Tell(msg: String)
  extends Interact[Unit]
```

Free monads

```
sealed trait Free[F[_],A]

case class Return[F[_],A](a: A)
  extends Free[F,A]

case class Bind[F[_],I,A](
  i: F[I],
  k: I => Free[F,A]) extends Free[F,A]
```

Free monads

```
sealed trait Free[F[_],A] {
  def flatMap[B](f: A => Free[F,B]): Free[F,B] =
    this match {
      case Return(a) => f(a)
      case Bind(i, k) =>
        Bind(i, k andThen (_ flatMap f))
    }
  def map[B](f: A => B): Free[F,B] =
    flatMap(a => Bind(f(a)))
}
```

Automatic lifting

```
implicit def lift[F[_],A](fa: F[A]): Free[F,A] =
  Bind(fa, (a: A) => Return(a))
```

User interaction

```
val prg: Free[Interact, Unit] =
  for {
    first <- Ask("What's your first name?")
    last  <- Ask("What's your last name?")
    _     <- Tell(s"Hello, $first $last!")
  } yield ()
```

Interaction

```
val prg: Free[Interact, Unit] =
  Ask("What's your first name?").flatMap(first =>
  Ask("What's your last name?").flatMap(last =>
  Tell(s"Hello, $first $last!").map(_ => ())))
```

Interaction

```
val prg: Free[Interact, Unit] =
  Bind(
    Ask("What's your first name?"), first =>
    Bind(
      Ask("What's your last name?"), last =>
        Bind(
          Tell(s"Hello, $first $last!"), _ =>
          Return(()))))
```

Running Free

```
sealed trait ~>[F[_],G[_]] {
  def apply[A](f: F[A]): G[A]
}

sealed trait Free[F[_],A] {
  def foldMap[G[_]:Monad](f: F ~> G): G[A] =
    this match {
      case Return(a) => Monad[G].pure(a)
      case Bind(fx, g) =>
        Monad[G].flatMap(f(fx)) { a =>
          g(a).foldMap(f)
        }
    }
}
```

Interaction interpreter

```
type Id[A] = A

object Console extends (Interact ~> Id) {
  def apply[A](i: Interact[A]) = i match {
    case Ask(prompt) =>
      println(prompt)
      readLine
    case Tell(msg) =>
      println(msg)
  }
}
```

Interaction interpreter

```
type Tester[A] =
  Map[String, String] => (List[String], A)

object Test extends (Interact ~> Tester) {
  def apply[A](i: Interact[A]) = i match {
    case Ask(prompt) => m => (List(), m(prompt))
    case Tell(msg) => _ => (List(msg), ())
  }
}
```

```scala
implicit val testerMonad = new Monad[Tester] {
  def pure[A](a: A) = _ => (List(), a)
  def flatMap[A,B](t: Tester[A])(f: A => Tester[B]) =
    m => {
      val (o1, a) = t(m)
      val (o2, b) = f(a)(m)
      (o1 ++ o2, b)
    }
}
```

```
scala> val m =
| Map("What's your first name?" -> "Harry",
|     "What's your last name?" -> "Potter"))
m: scala.collection.immutable.Map[String,String]

scala> val v = prg.foldMap(Test).apply(m)
v: (List[String], Unit) = (List(Hello, Harry Potter!),())
```

Let's add a feature

```
Ask the user for credentials.
If the user is authorized,
  tell the secret.
Otherwise
  tell the user to go away.
```

Security algebra

```
sealed trait Auth[A]

case class Login(u: UserID, p: Password)
    extends Auth[User]

case class HasPermission(
    u: User, p: Permission) extends Auth[Boolean]
```

What is the type here?

```
val prg2: Free[???, Unit] = for {
  uid <- Ask("What's your user ID?")
  pwd <- Ask("Password, please.")
  usr <- Login(uid, pwd)
  b <- HasPermission(usr, KnowTheSecret)
  _ <- if (b) Tell("UUDDLRLRBA")
       else   Tell("Go away!")
} yield ()
```

Coproduct

```
case class Coproduct[F[_],G[_],A](
  value: Either[F[A],G[A]])

type App[A] = Coproduct[Interact,Auth,A]

val prg: Free[App, A] = ...
```

Injection

```
sealed trait Inject[F[_],G[_]] {
  def inj[A](sub: F[A]): G[A]
}

object Inject {
  implicit def refl[F[_]]: Inject[F,F] = ???

  implicit def left[F[_],G[_]]:
    Inject[F,({type λ[x]=Coproduct[F,G,x]})#λ] = ???

  implicit def right[F[_],G[_],H[_]](
    implicit I: Inject[F,G]):
    Inject[F,({type λ[x]=Coproduct[H,G,x]})#λ] = ???
}
```

Lifting

```
def lift[F[_],G[_],A](f: F[A])(
  implicit I: Inject[F,G]): Free[G,A] =
    Bind(I.inj(f), Return(_:A))
```

```
class Interacts[F[_]](implicit I: Inject[Interact,F]) {
  def tell(msg: String): Free[F,Unit] =
    lift(Tell(msg))
  def ask(prompt: String): Free[F,String] =
    lift(Ask(prompt))
}
```

End result

```
def prg[F[_]](implicit I: Interacts[F],
                       A: Auths[F]): Free[F,Unit] = {
  import I._; import A._
  for {
    uid <- ask(“What’s your user ID?")
    pwd <- ask(“Password, please.”)
    u <- login(uid, pwd)
    b <- hasPermission(u, KnowSecret)
    _ <- if (b) tell("UUDDLRLRBA") else tell("Go away!")
  } yield ()
}
```

Composite interpreter

```
sealed trait ~>[F[_],G[_]] {
  def apply[A](f: F[A]): G[A]

  def or[H[_]](f: H ~> G) =
    new (({type t[x] = Coproduct[F,H,x]})#t ~> G) {
      def apply[A](c: Coproduct[F,H,A]): G[A] =
        c.run match {
          case Left(fa) => f(fa)
          case Right(ga) => g(ga)
        }
    }
}
```

Running a program

```
val app: Free[App, Unit] = prg[App]

def runApp = app.foldMap(MyAuth or Console))
```

Storage algebra

```
sealed trait Storage[K,V,A]

case class Get[K,V](key: K)
  extends Storage[K,V,V]

case class Put[K,V](key: K, value: V)
  extends Storage[K,V,Unit]

case class Del[K,V](key: K)
  extends Storage[K,V,Unit]
```

Logging algebra

```scala
sealed trait LogLevel
case class ErrorLevel extends LogLevel
case class WarnLevel extends LogLevel
case class InfoLevel extends LogLevel
case class DebugLevel extends LogLevel

case class Log[A](level: LogLevel, msg: String)
```

Composite type

```
type F0[A] = Coproduct[Interact, Auth, A]
type F1[A] = Coproduct[Logging, F0, A]
type F2[A] = Coproduct[Error, F1, A]
type App[A] = Coproduct[Storage, F2, A]

val app: Free[App, A] = prg[App]
```

Library code

- Define your individual algebras.
- Make smart constructors to lift them into coproducts.
- Define your individual interpreters.

User code

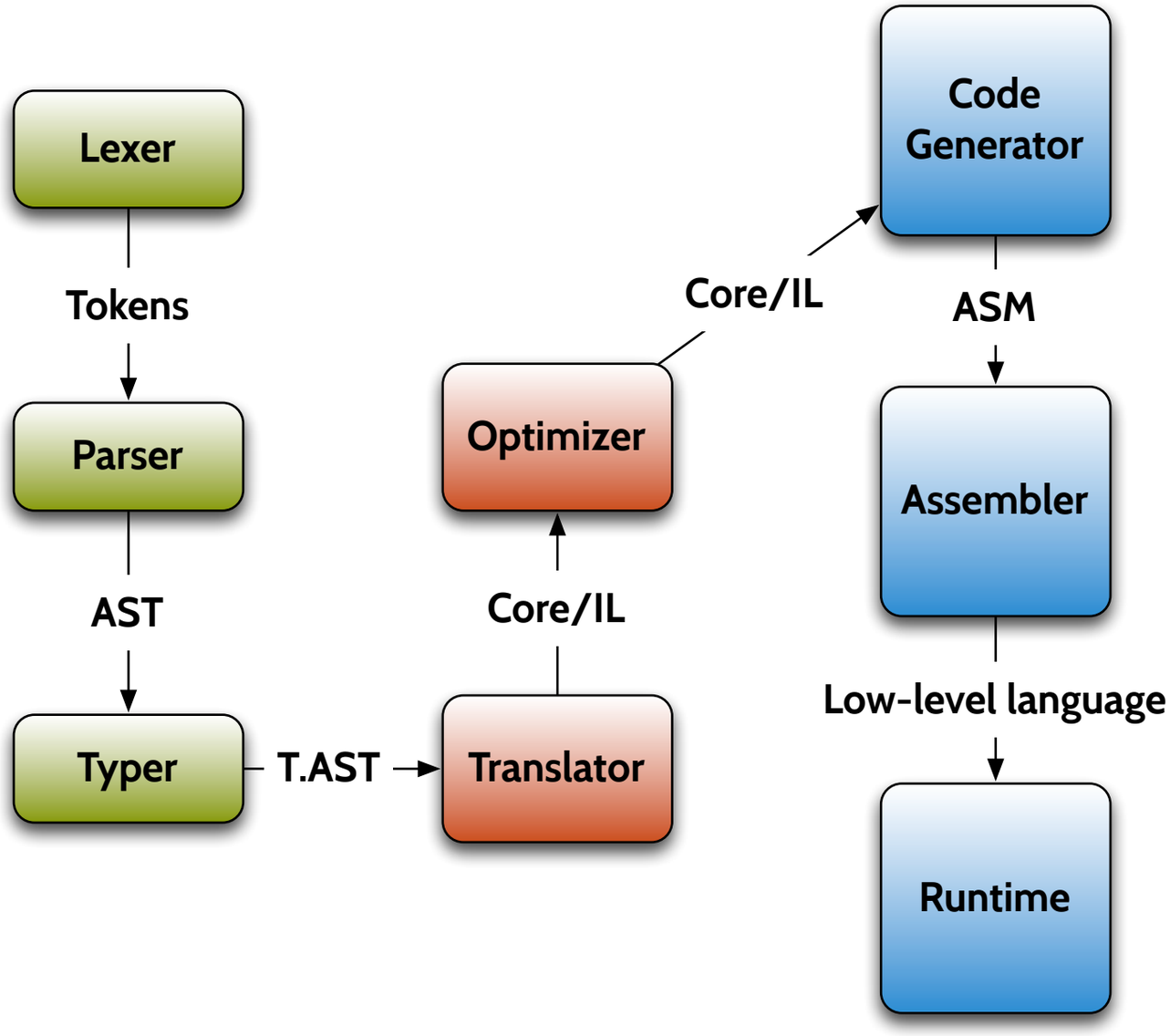
- Write your program using smart constructors.
- Compose the appropriate interpreter.
- Fold the program using the interpreter.

Stratified architecture

```
def foldMap[G[_]:Monad](f: F ~> G): G[A]
```

G could be a free monad!

Stratified architecture



Free monads

- Stack-safe
- Retry portions of the program (see scalaz.Task)
- Draw a diagram of the program (see github.com/puffnfresh/free-graphs)
- Step through the program, add breakpoints, etc, all programmatically.

Summary

- Write your program using exactly the language you need.
- Compose your language from smaller orthogonal languages, in a canonical way.
- Plug in interpreters that support the behavior you want.

Code from this presentation:

http://goo.gl/qhnPk1

`scalaz.Free`

https://github.com/scalaz/scalaz