Eff Monad

Binzi Cao @British Telecom

What is Monad?
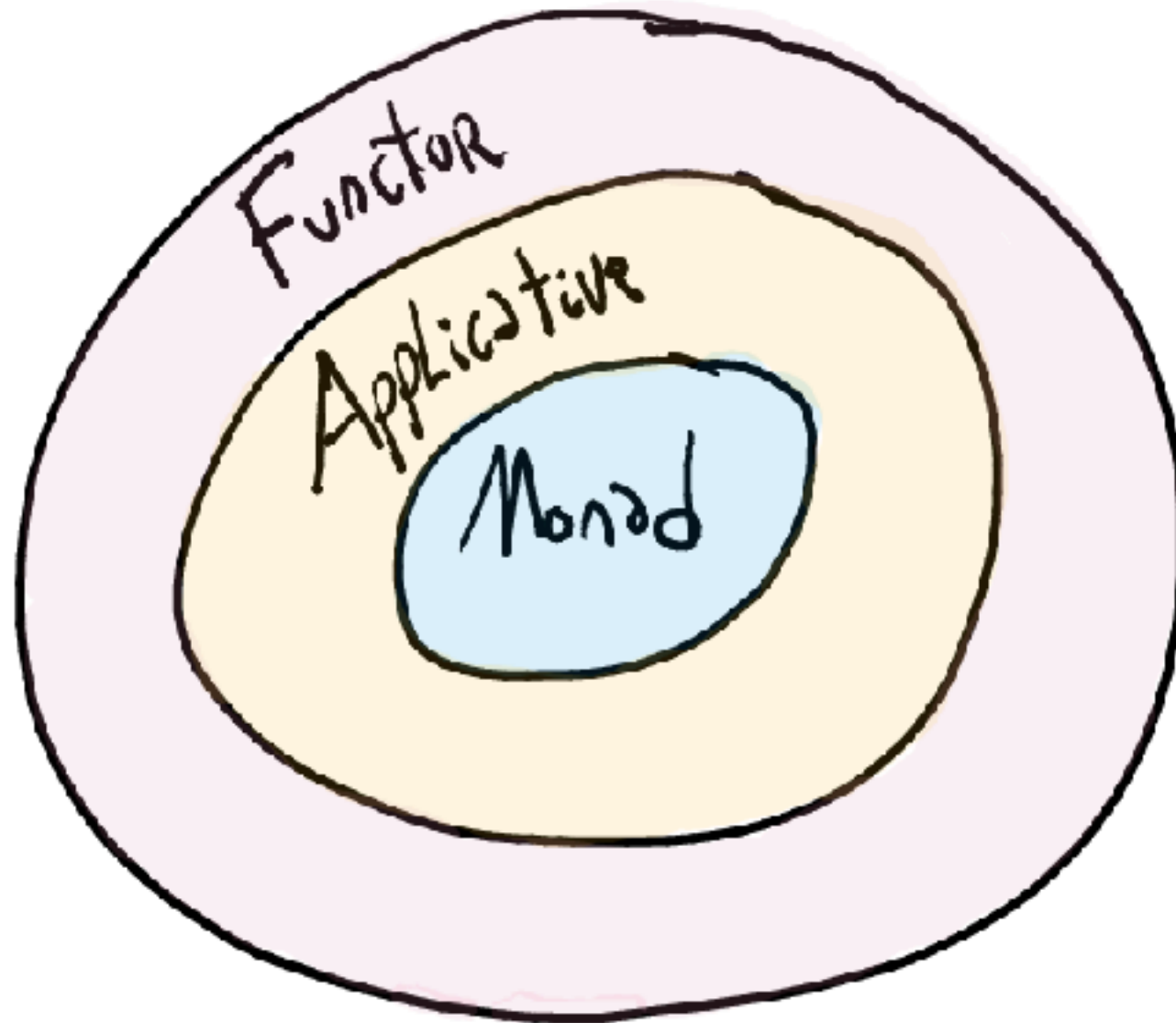
Type classes

|	
Functor	**def map[A, B](f: A => B): F[A]=>F[B]**
Applicative Functor	**def apply[A, B] (f: F[A => B]): F[A] => F[B]**
Monad	**def bind[A, B](f: A => F[B]): F[A] => F[B]**

Monad

```scala
import scalaz._
import Scalaz._
val double:Int => Int = 2*

//Functor
Option(2).map(double)
//Applicative
Option(2) <*> Option(double)
//Monad
Option(2) >>= (double(_).some)
```

Monads

- List, Option, Future

- Writer[W, A]: A monad for logging

- Reader[E, A]: A monad which read values

- State[S, A]: A monad for state machine

- ....

Life is not easy
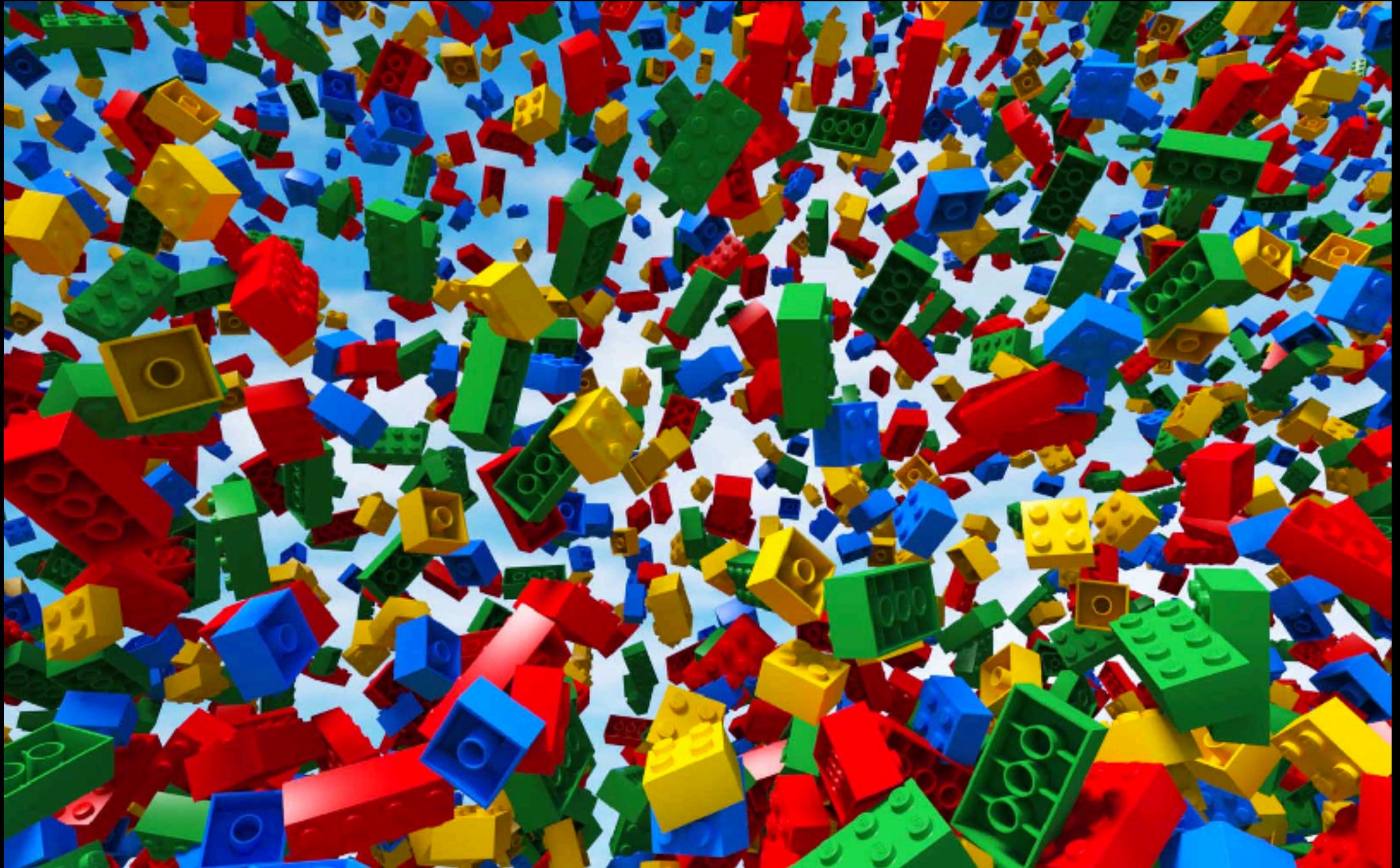
Life is beautiful

Monad Composition

- We always need to compose contexts or effects

- Option+Writer+Reader+…….

- How can I handle this?

Monad Transformer

- Either + Option

  - *type EOMonad[A] = EitherT[Option, String, A]*

- Either + Option + List

  - *type EOLMonad[A] = ListT[EitherT[Option, String, ?], A]*

- Either + Option + List + Reader + Writer????

Free Monad

- Monads + Interpreters

- Compose various monads via Coproduct

- Interpreters via NaturalTransformation

Some Helper Code

```scala
implicit class NaturalTransformationOrOps[F[_], H[_]]
(private val nt: F ~> H) extends AnyVal {
    def or[G[_]](f:G ~> H): Coproduct[F, G, ?] ~> H =
      new (Coproduct[F, G, ?] ~> H) {
        def apply[A](c: Coproduct[F, G, A]): H[A] = c.run match {
          case -\/(fa) => nt(fa)
          case \/-(ga) => f(ga)
        }
      }
}
type -~>[F[_], G[_]] = Inject[F, G]
object LiftImplicit {
  implicit def lift[F[_], G[_], A](
    fa: F[A]
  )(
  implicit I: F -~> G
): Free.FreeC[G, A] =
    Free liftFC I.inj(fa)
}
```

Monads Composition

Lift to Free Monad

Program

```scala
type PRG0[A] = Coproduct[Interact, Crud, A]
type PRG[A] = Coproduct[Log, PRG0, A]
type PRG1[A] = Coproduct[PPLog, PRG, A]
val program: Free[PRG1, Boolean] = prg[PRG1]
val interpreter0: PRG0 ~> Result = (Console andThen Id2Result) or CrudInterpreter
val interpreter: PRG ~> Result = (Printer andThen Id2Result) or interpreter0
val interpreter1: PRG1 ~> Result = (PPPrinter andThen Id2Result) or interpreter
val result0: Result[Boolean] = program.foldMap(interpreter1)
```

Program Stack

Interpreters

We need more

- What if I don't want to have boilerplate code , computer should be cleverer than me

- What if I want to compose, add and remove monads easily in one program

- What if I want to change a behaviour of a monad at run time

- Yes, Eff Monad!

Eff Monad

- Freer Monad from Haskell

  - http://okmij.org/ftp/Haskell/extensible/more.pdf

- A program is a big effect stack composed of multiple different effects

- An effect is modular and can be added and removed from a big effect stack at run time

- An effect can be translated to other effect

- Less boilerplate code

Eff Monad library

- https://github.com/atnos-org/eff

- Eric Torreborre

- Author of specs2

eff

build passing gitter join chat

Extensible effects are an alternative to monad transformers for computing with effects in a functional way. This library is based on the "free-er" monad and extensible effects described in Oleg Kiselyov in Freer monads, more extensible effects.

You can learn more in the User Guide:

- your first effects
- included effects: Reader , Writer , Eval , State ,...
- create your own effects
- use Member implicits
- working with different effect stacks
- a tutorial similar to the cats' tutorial for Free monads

You can also check out this presentation at flatMap Oslo 2016 (slides).

A basic sample

```scala
type Stack = Fx.fx3[WriterString, ReaderInt, Eval]
```

Effect Stack

```scala
def program[R: _readerInt: _writerString: _eval]: Eff[R, Int] =
  for {
    n <- ask[R, Int]
    _ <- tell("the required power is " + n)
    a <- delay(math.pow(2, n.toDouble).toInt)
    _ <- tell("the result is " + a)
  } yield a
```

Effects

```scala
println(program[Stack].runReader(6).runWriter.runEval.run)
//(64,List(the required power is 6, the result is 64))
println(program[Stack].runWriter.runReader(6).runEval.run)
//(64,List(the required power is 6, the result is 64))
```

Out of Box Effects

Name	Description	Link
EvalEffect	an effect for delayed computations	link
OptionEffect	an effect for optional computations, stopping when there's no available value	link
EitherEffect	an effect for computations with failures, stopping when there is a failure	link
ValidateEffect	an effect for computations with failures, allowing to collect failures	link
ErrorEffect	a mix of Eval and Either, catching exceptions and returning them as failures	link
ReaderEffect	an effect for depending on a configuration or an environment	link
WriterEffect	an effect to log messages	link
StateEffect	an effect to pass state around	link
ListEffect	an effect for computations returning several values	link
ChooseEffect	an effect for modeling non-determinism	link
MemoEffect	an effect for memoizing values	link
FutureEffect	an effect for asynchronous computations	link
SafeEffect	an effect for guaranteeing resource safety	link

Eff Step by Step

- Create your ADTs

- Lift your ADT to Eff[R, A] via Eff.send

- Build your program

- Write you interpreters

- Run it

An IVR Billing APP

ADT
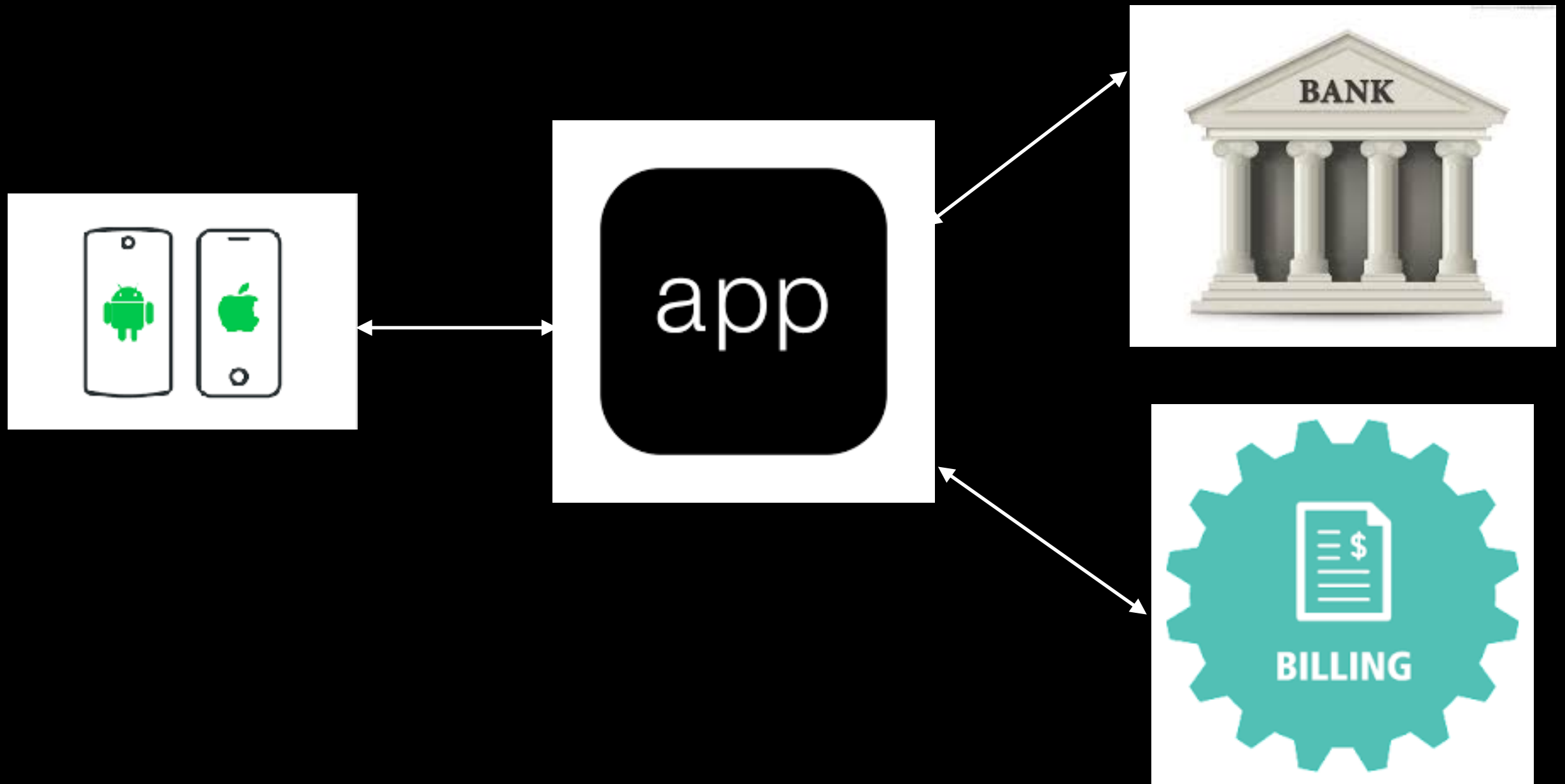
```scala
sealed trait BillOp[A]

case class CheckBill(bill: String) extends BillOp[Boolean]

case class UpdateBill(bill: String, status: String) extends BillOp[Option[String]]




sealed trait BankOp[A]

case class Purchase(bill: String, card: String) extends BankOp[Option[String]]
case class Refund(bill:   String, card: String) extends BankOp[Option[String]]
```

IVR

```scala
sealed trait IvrOp[A]

sealed trait Result
case object Continue extends Result
case object RequestAgain extends Result
case object Stop extends Result


case class Request(prompt: String) extends IvrOp[String]
case class Response(msg:    String) extends IvrOp[Unit]
case class CheckInput(msg:   String) extends IvrOp[Result]
```

Lift ADT to Eff

```scala
type _bankOp[R] = BankOp |= R
def purchase[R: _bankOp](bill: String, card: String): Eff[R, Option[String]] =
  Eff.send[BankOp, R, Option[String]](Purchase(bill, card))
```

Call Eff.send explicitly

```scala
object EffHelper {
  implicit def liftEff[A, F[_], R: F |= ?](s: F[A]): Eff[R, A] =
    Eff.send[F, R, A](s)
  }
}
```

Call Off.send implicitly

App

```scala
def program[R: _ivrOp: _billOp: _bankOp]: Eff[R, Unit] =
  for {
    bill      <- Request("Please type in your bill reference ")
    _         <- Response(s"Your bill reference: ${bill}")
    card      <- Request("Please type in your credit card info ")
    _         <- Response(s"Your credit card is : ${card}, we are processing n
    reference <- Purchase(bill, card)
    receipt   <- UpdateBill(bill, "Paid")
    _         <- Response(s"Your payment refrence is ${receipt}")
  } yield ()

type Stack = Fx.fx3[IvrOp, BillOp, BankOp]
program[Stack].runBill.runIvr.runBank.run

// Please type in your bill reference
//1
//Your bill reference: 1
//Please type in your credit card info
//2
//Your credit card is : 2, we are processing now
//Your payment refrence is Some(Ok)
```

The minimum effect stack

Interpreters

```scala
trait Recurser[M[_], R, A, B] {
  def onPure(a: A): B
  def onEffect[X](m: M[X]): X Either Eff[R, B]
  def onApplicative[X, T[_]: Traverse](ms: T[M[X]]): T[X] Either M[T[X]]
}

trait Translate[T[_], U] {
  def apply[X](kv: T[X]): Eff[U, X]
}

trait SideEffect[T[_]] {
  def apply[X](tx: T[X]): X
  def applicative[X, Tr[_] : Traverse](ms: Tr[T[X]]): Tr[X]
}
```

Interpreters

```scala
type _billOp[R] = BillOp |= R
def runBillOp[R, A](effect: Eff[R, A])(implicit m: BillOp <= R): Eff[m.Out, A] = {
  val memDataSet = new scala.collection.mutable.ListBuffer[String]

  recurse(effect)(new Recurser[BillOp, m.Out, A, A] {
    def onPure(a:       A): A = a
    def payBill(bill: String, card: String) = "Ok".some
    def check(bill:     String) = bill === "1234"


    def onEffect[X](i: BillOp[X]): X Either Eff[m.Out, A] = Left {
      i match {
        case UpdateBill(bill, card) => payBill(bill, card)
        case CheckBill(bill)        => check(bill)
      }
    }


    def onApplicative[X, T[_]: Traverse](ms: T[BillOp[X]]): T[X] Either BillOp[T[X]]
      Left(ms.map {
        case UpdateBill(bill, card) => payBill(bill, card)
        case CheckBill(bill)        => check(bill)
      })
  })(m)
}
```

The rest of effects tack

Effect Interpreter

I want check user's inputs effects!

```scala
def checkInput[R: _ivrOp](input: String): Eff[R, Option[String]] =
  (CheckInput(input): Eff[R, Result]) >>= { r =>
    r match {
      case Continue    => Eff.pure(input.some)
      case Stop        => Eff.pure(None)
      case RequestAgain => askForBill[R]
    }
  }

def askForBill[R: _ivrOp]: Eff[R, Option[String]] =
  for {
    input <- Request("Please type in your bill reference or type 0 to stop")
    bill  <- checkInput(input)
  } yield bill
```

Recursive Effect

Option effect is in !

```scala
def program[R: _ivrOp: _billOp: _bankOp: _option]: Eff[R, Unit]
  for {
    billOption <- askForBill
    bill       <- fromOption(billOption)
    _          <- Response(s"Your bill reference: ${bill}")
    card       <- Request("Please type in your credit card info ")
    _          <- Response(s"Your credit card is : ${card}, we are processing now"
    reference  <- Purchase(bill, card)
    receipt    <- finishCall(bill, reference)
  } yield ()

type Stack = Fx.fx4[IvrOp, BillOp, BankOp, Option]
program[Stack].runBill.runIvr.runBank.runOption.run
```

Option Effect

How can I miss logging?

```scala
type WriterString[A]  = Writer[String, A]
type _writerString[R] = WriterString |= R

def program[R: _ivrOp: _billOp: _bankOp: _option: _writerString]: Eff[R, Unit] =
  for {
    _          <- tell("A customer called in, let's start ")
    billOption <- askBill
    bill       <- fromOption(billOption)
    _          <- Response(s"Your bill reference: ${bill}")
    card       <- Request("Please type in your credit card info ")
    _          <- Response(s"Your credit card is : ${card}, we are processing now")
    reference  <- Purchase(bill, card)
    receipt    <- finishCall(bill, reference)
    _          <- Response(s"Your payment refrence is ${receipt}")
    _          <- tell("We have finished everyting")
  } yield ()

type Stack = Fx.fx5[IvrOp, BankOp, BillOp, Option, WriterString]
program[Stack].runBill.runIvr.runBank.runOption.runWriter.run
```

Writer Effect

One More thing

- I have a performance issue and I want to log the time taken for each operation and I hate below code

```scala
def getTimeStamp = s"${new java.util.Date}"
def program[R: _ivrOp: _billOp: _bankOp: _writerString]: Eff[R, Unit] =
  for {
    _        <- tell("<<<${getTimeStamp}>>>:Request for bill reference Start")
    bill     <- Request("Please type in your bill reference ")
    _        <- tell("<<<${getTimeStamp}>>>:Request for bill reference End")
    _        <- tell("<<<${getTimeStamp}>>>:Response for bill reference Start")
    _        <- Response(s"Your bill r
    _        <- tell("<<<${getTimeStamp
    _        <- tell("<<<${getTimeStamp
    card     <- Request("Please type i
    _        <- tell("<<<${getTimeStamp
    _        <- tell("<<<${getTimeStamp
    _        <- Response(s"Your credit
    _        <- tell("<<<${getTimeStamp
    _        <- tell("<<<${getTimeStamp
    reference <- Purchase(bill, card)
    _        <- tell("<<<${getTimeStamp
    _        <- tell("<<<${getTimeStamp
    receipt  <- UpdateBill(bill, refere
    _        <- tell("<<<${getTimeStamp
    _        <- tell("<<<${getTimeStamp
    _        <- Response(s"Your payment
    _        <- tell("<<<${getTimeStamp
  } yield ()
```

Eff Translation

```scala
type WriterString[A] = Writer[String, A]
implicit class LogTimesOps[R, A](e:     Eff[R, A]) {
  def logTimes[T[_]](implicit memberT: MemberInOut[T, R],
                     writer:           MemberIn[WriterString, R]): Eff[R, A] =
    LogHelper.logTimes[R, T, A](e)
}

def logTimes[R, T[_], A](eff:                    Eff[R, A])(implicit memberT: MemberInOut[T, R]
                                   writer: MemberIn[WriterString, R]): Eff[R, A] = {

  translateInto(eff)(new Translate[T, R] {
    def apply[X](tx: T[X]): Eff[R, X] =
      for {
        _ <- tell[R, String](s"${new java.util.Date}:$tx start")
        x <- send[T, R, X](tx)
        _ <- tell[R, String](s"${new java.util.Date}:$tx end")
      } yield x
  })
}
```

Writer effect injected

Eff Translation

```scala
def program[R: _ivrOp: _billOp: _bankOp: _option]: Eff[R, Unit] =
  for {
    billOption <- askBill
    bill       <- fromOption(billOption)
    cats       <- Response(s"Your bill reference: ${bill}")
    card       <- Request("Please type in your credit card info ")
    cats       <- Response(s"Your credit card is : ${card}, we are processing now")
    reference  <- Purchase(bill, card)
    receipt    <- UpdateBill(bill, "Paid")
    _          <- Response(s"Your payment refrence is ${receipt}")
  } yield ()

type Stack = Fx.fx5[IvrOp, BillOp, BankOp, Writer[String, ?], Option]

val (result, logs) =
  program[Stack].logTimes[BillOp].runBill.runBank.runIvr.runOption.runWriter.run
logs.foreach(println)
```

Writer effect injected

Eff[R, A]=> Eff[U,A]

into	Move current effect to a larger stack
transform	Transform an effect into another one by using a natural transformation
translate	Translate an effect into other effects in the stack
translateInto	Translate one effect of the stack into other effects in a larger stack

Links

- https://github.com/atnos-org/eff

- http://atnos-org.github.io/eff/