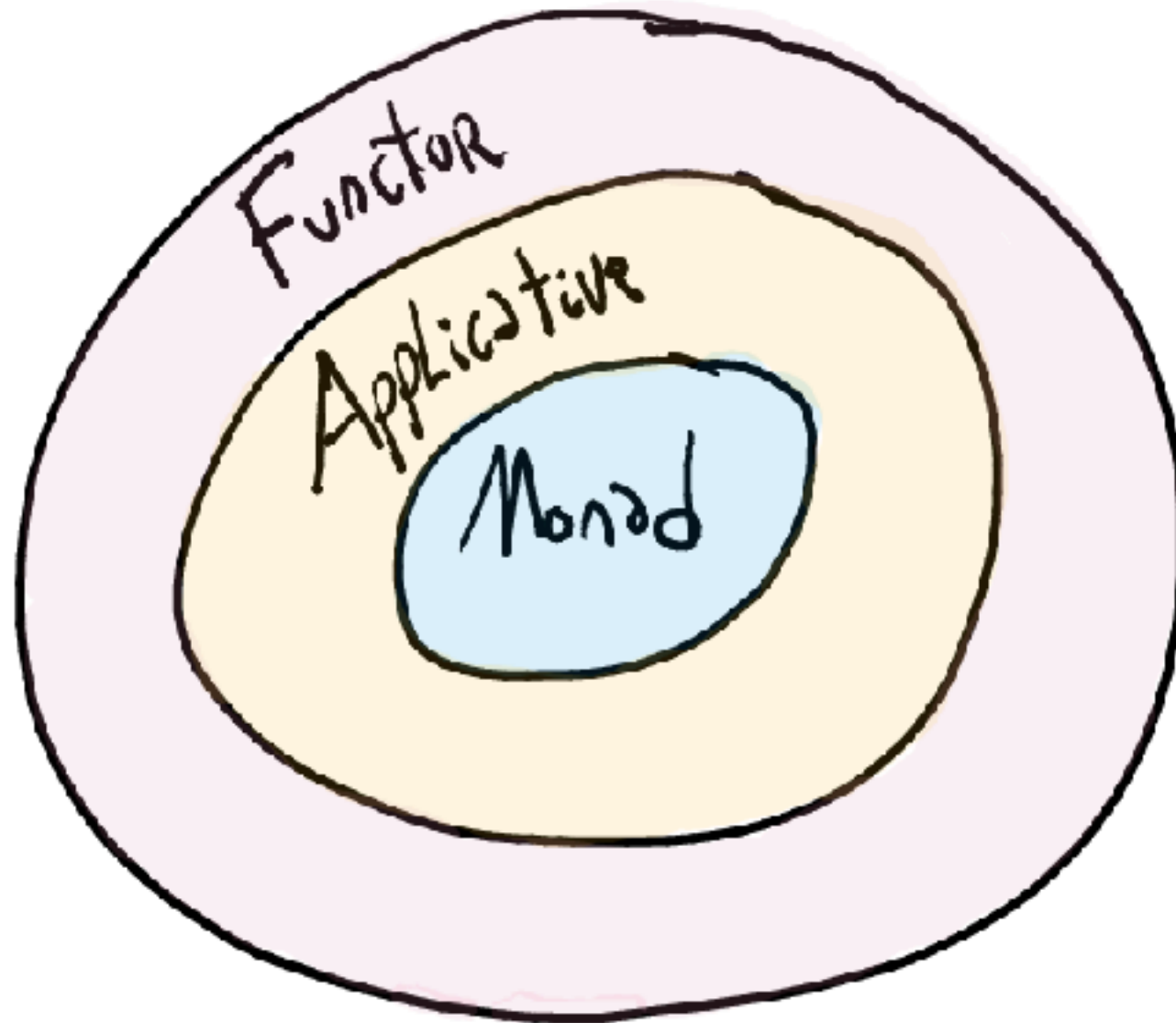


Eff Monad

Binzi Cao @BT

What is Monad?



Type classes

Functor	<code>def map[A, B](f: A => B): F[A] => F[B]</code>
Applicative Functor	<code>def apply[A, B] (f: F[A] => B): F[A] => F[B]</code>
Monad	<code>def bind[A, B](f: A => F[B]): F[A] => F[B]</code>

Monad

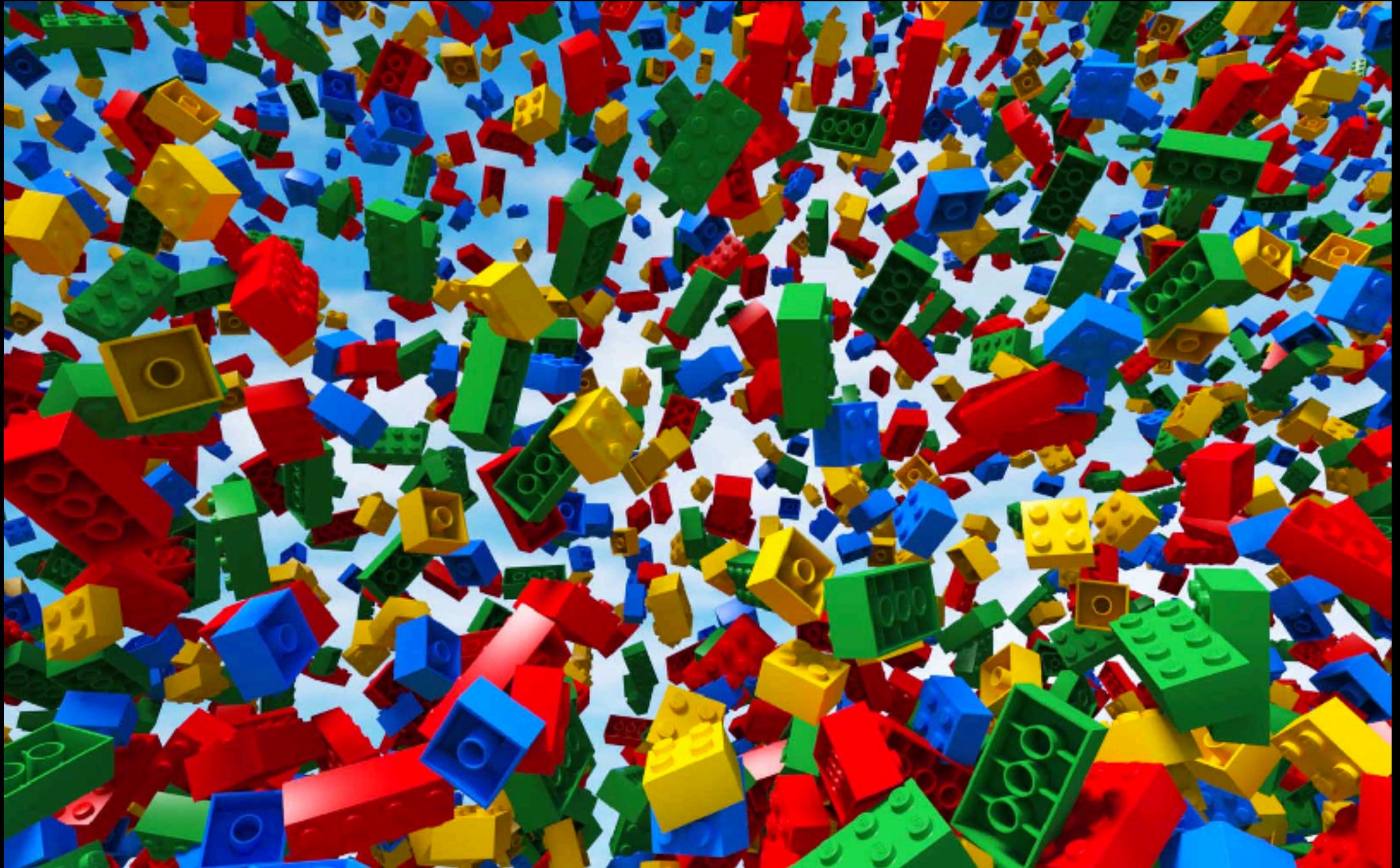
```
import scalaz._
import Scalaz._
val double: Int => Int = 2*

//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 reads 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 ADTs via Coproduct
- Interpreters via NaturalTransformation

Some Helper Code

```
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)
}
```

Interpreters
Composition

Lift to Free Monad

Program

Program Stack

```
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 = (PPPprinter andThen Id2Result) or interpreter
val result0: Result[Boolean] = program.foldMap(interpreter1)
```

Interpreters

We need more

- What if I don't want to have so much boilerplate code , computer should be cleverer than me
- What if I want to compose, add and remove effects easily in one program
- What if I want to change a behaviour of an effect 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                                   

A basic sample

Effect
Stack

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

```
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

```
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))
```

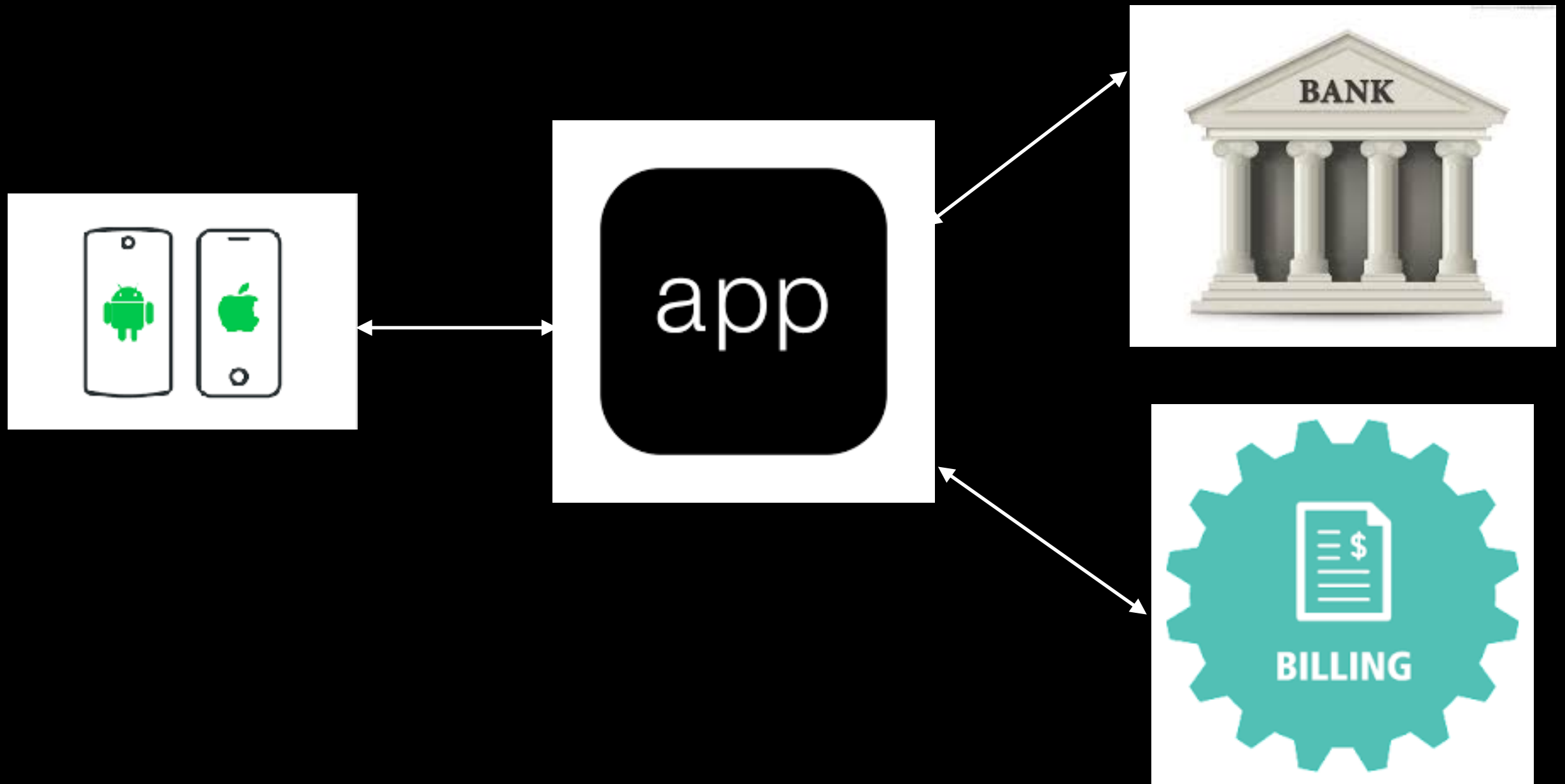

Out of Box Effects

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

Eff Step by Step

- Create your ADTs
- Lift your ADT to $\text{Eff}[R, A]$ via `Eff.send`
- Build your program
- Write your interpreters
- Run it

An IVR Billing App



ADT

```
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]]
```


ADT

```
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

```
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

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

Call Eff.send implicitly

The App

```
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 now")  
    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
```

The minimum effect
stack

```
// 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)
```

Interpreters

```
trait Recursor[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

```
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 Recursor[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)
    })
}
```

The rest of effects stack

Effect Interpreter

I want to check user's inputs!

```
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]  
    }  
  }
```

Recursive Effect

```
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
```

Option effect is in !

```
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?

```
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

```

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 reference: ${bill}")
    _      <- tell("<<<${getTimestamp}>>>:Response for bill reference End")
    _      <- tell("<<<${getTimestamp}>>>:Request for card info Start")
    card    <- Request("Please type in your credit card info ")
    _      <- tell("<<<${getTimestamp}>>>:Request for card info End")
    _      <- tell("<<<${getTimestamp}>>>:Response for card info Start")
    _      <- Response(s"Your credit card info: ${card}")
    _      <- tell("<<<${getTimestamp}>>>:Response for card info End")
    reference <- Purchase(bill, card)
    _      <- tell("<<<${getTimestamp}>>>:Purchase Success")
    _      <- tell("<<<${getTimestamp}>>>:Purchase Failure")
    receipt  <- UpdateBill(bill, reference)
    _      <- tell("<<<${getTimestamp}>>>:Update Bill Success")
    _      <- tell("<<<${getTimestamp}>>>:Update Bill Failure")
    _      <- Response(s"Your payment reference: ${reference}")
    _      <- tell("<<<${getTimestamp}>>>:Response for payment reference End")
  } yield ()

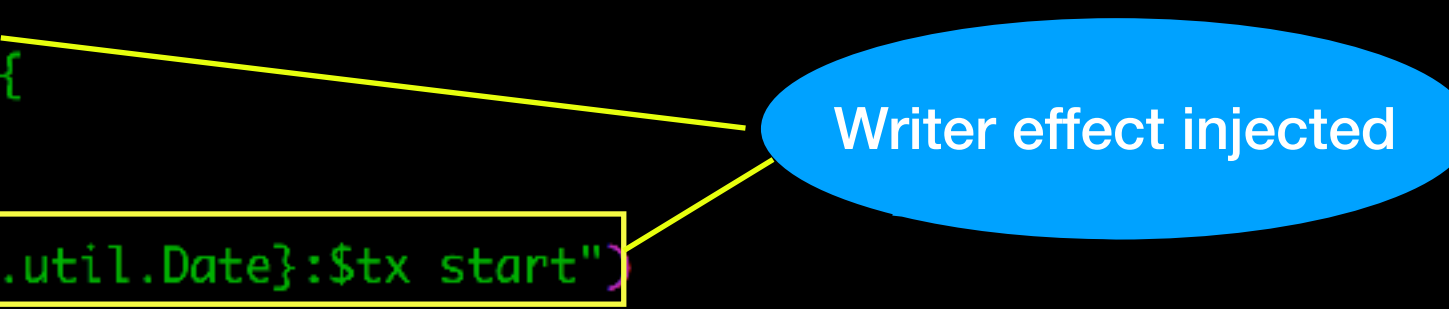
```



Eff Translation

```
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
  })
}
```



The diagram illustrates the injection of the Writer effect into the `logTimes` function. A blue oval labeled "Writer effect injected" has two yellow lines pointing to the `tell` calls within the `Translate` object's `apply` method. The first line points to `tell[R, String](s"${new java.util.Date}:$tx start")`, and the second line points to `tell[R, String](s"${new java.util.Date}:$tx end")`. Both `tell` calls are highlighted with yellow boxes.

Eff Translation

```
def program[R: _ivrOp: _billOp: _bankOp: _option]: Eff[R, Unit] =  
  for {  
    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   <- 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



$$\text{Eff}[R, A] \Rightarrow \text{Eff}[U, A]$$

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

Monad Loop

```
def program[R: _kvstore]: Eff[R, Unit] =  
  for {  
    _ <- Put("wild-cats", 2)  
    _ <- Put("tame-cats", 5)  
    n <- Get("wild-cats")  
    r = n.map(_ * 2)  
    _ <- Delete("tame-cats")  
  } yield ()
```

```
def done[R: _kvstore]: Eff[R, Boolean] = Check
```

```
type Stack = Fx.fx2[Option, KVStore]
```

```
program[Stack].untilM_(done[Stack])
```

Monad with a
Control Loop

Links

- <https://github.com/atnos-org/eff>
- <http://atnos-org.github.io/eff/>
- Some other similar libs
 - <https://github.com/frees-io/freestyle>
 - <https://github.com/b-studios/scala-effekt>

Thanks