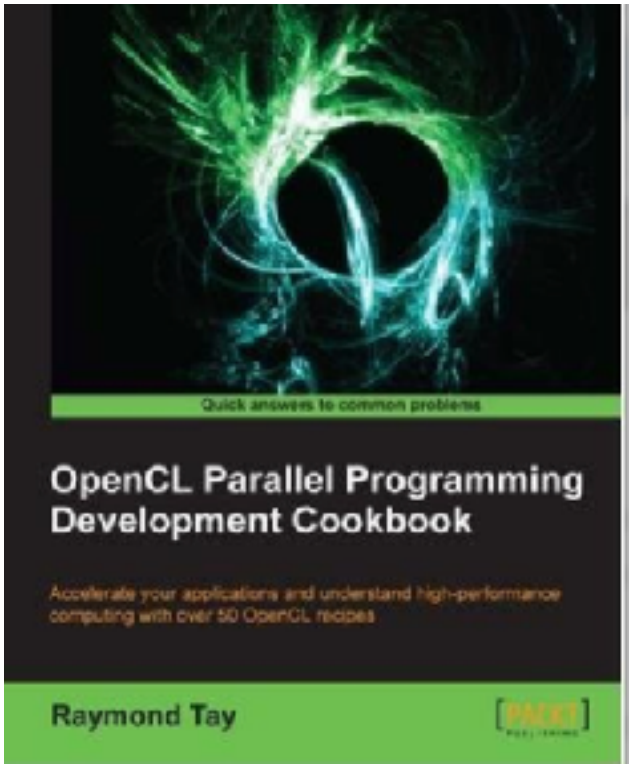
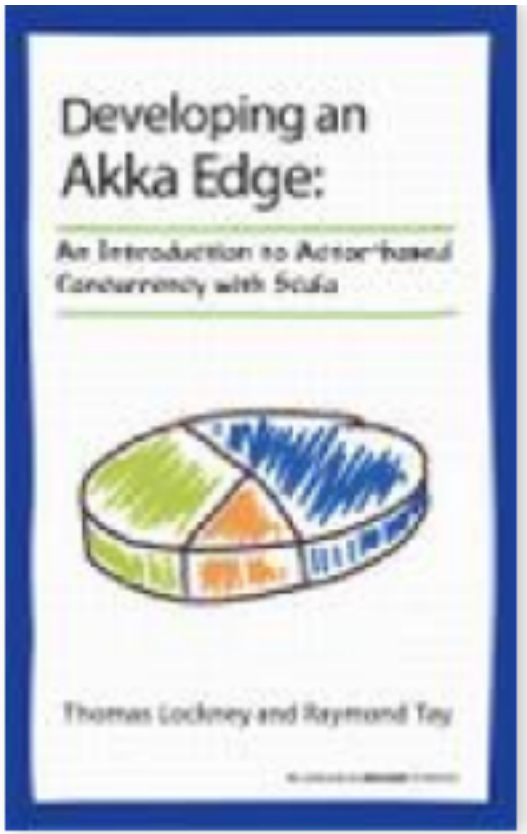
Developing a principled approach to programming IO in Scala

Raymond Tay

A little about me





A little bit more about me

Contributed to cats, dotty, eff-monad, akka, akka-http, scala compiler.

Effects

What is it? **Anything** but you are probably thinking about ***side-effect(s)***.

Controlling how side-effects are effected is the purpose of the **IO Monad**.

Effects (Scala 2)

```
scala> println("Hello World") // eager evaluation !
Hello World
scala> implicit val ec = scala.concurrent.ExecutionContext.global
ec: scala.concurrent.ExecutionContext = scala.concurrent.impl.ExecutionContextImpl@54ea33ba

scala> Future(println("Hello World")) // still eager
Hello World
res2: scala.concurrent.Future[Unit] = Future(<not completed>)
```

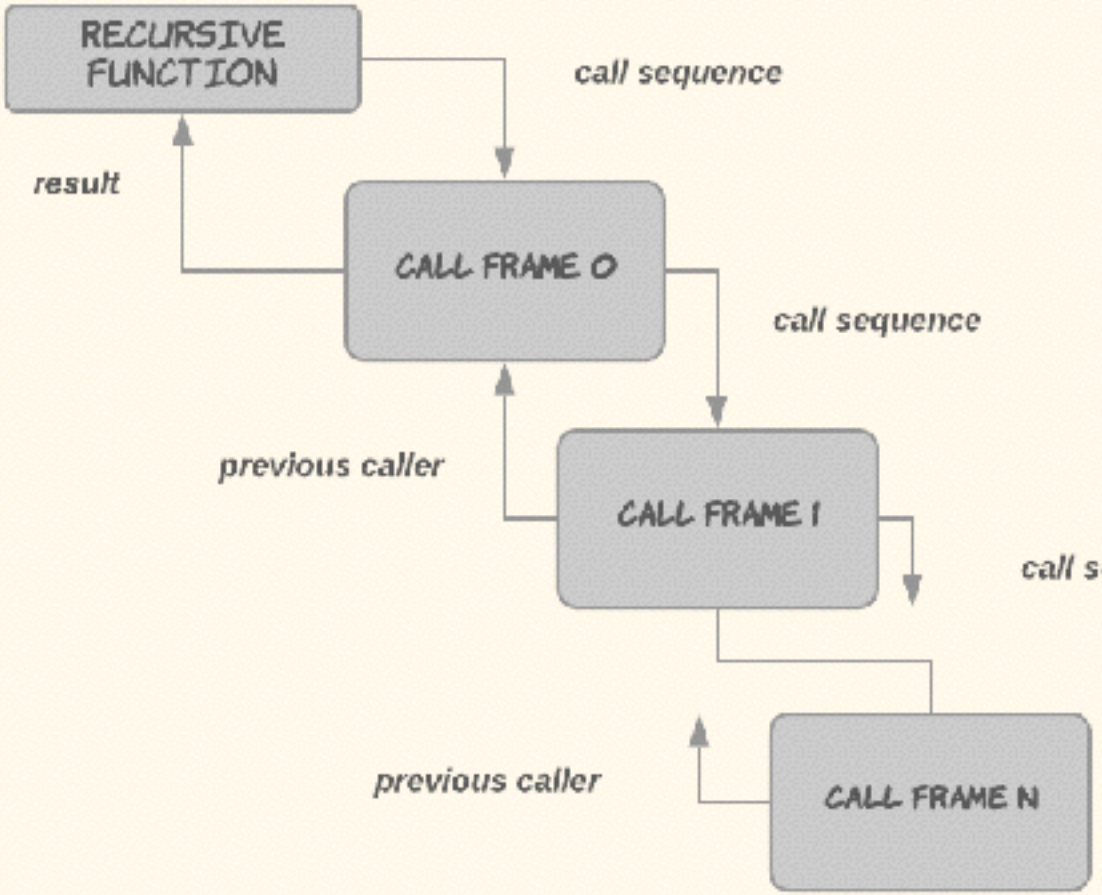
What is needed ...

- **Sequencing computations**
- Stack safety
- Support evaluation modes
  - Lazy
  - Asynchronous
  - Strict

What is needed ...

- Sequencing computations
- **Stack safety**
- Support evaluation modes
  - Lazy
  - Asynchronous
  - Strict

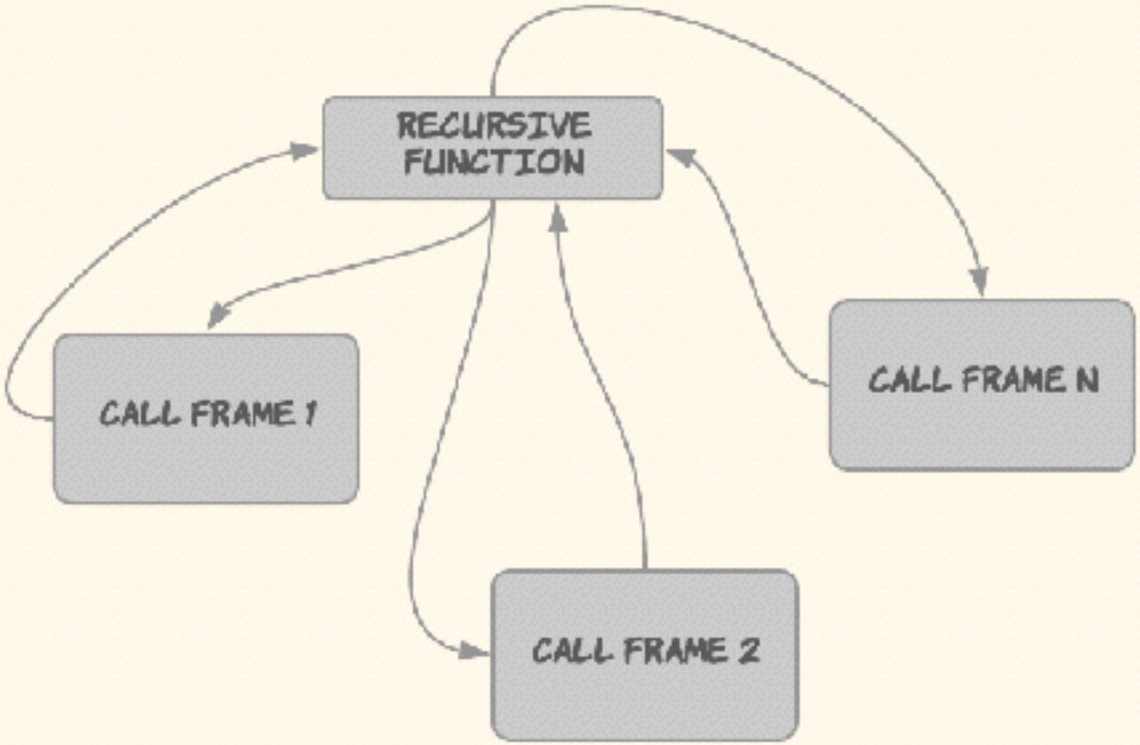
Stack execution



The computer program's flow refers to the interaction between functions and other ***control constructs***; the ***call-stack*** is never explicit.

A ***call-stack*** is only present when the program is executing.

Trampoline



A ***Trampoline*** is a construct that resembles a *plain-object with the purpose of carrying a payload where this payload is a* ***FUNCTION*** *to continue the next phase of the computation aka* ***continuation.***

What is needed ...

- Sequencing computations
- Stack safety
- **Support evaluation modes**
  - **Lazy**
  - **Asynchronous**
  - **Strict**

Getting started with IO

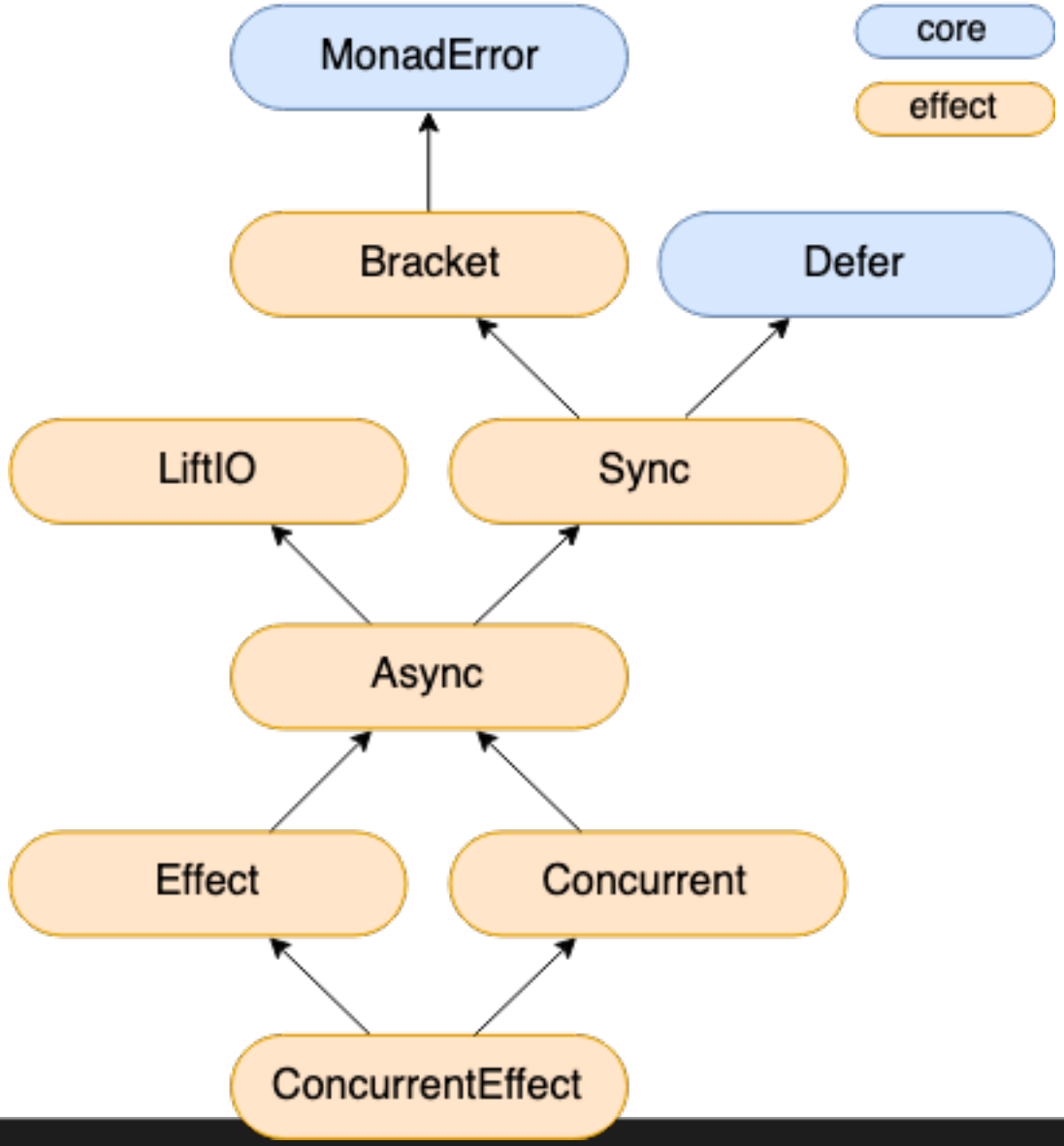
```
scala> val ioa = IO(println("Hello World"))
ioa: cats.effect.IO[Unit] = IO$647835100

scala> ioa.unsafeRunSync
Hello World
```

What exactly is IO?

A value of type `IO[A]` is a computation which, when evaluated, can perform effects before returning a value of type `A`

`IO` values are **pure**, **immutable** values and thus preserves **referential transparency**, being usable in functional programming. An `IO` is a data structure that represents just a description of a side effectful computation.

IO Typeclass Hierarchy



Asynchronous & Cancelable

```
def processServiceResultConc[F[_]: Concurrent](idx: Int) : F[String] =
  Concurrent[F].asyncF{ (cb: Either[Throwable, String] => Unit) =>
    val active = new AtomicBoolean(true)
    if (active.getAndSet(false))
      service.getResult().onComplete {
        case Success(s) => println(s"=> Concurrent-
[${Thread.currentThread.getName}] Yes:$idx"); cb(Right(s+"_"+idx))
        case Failure(e) => println(s"=> Concurrent-
[${Thread.currentThread.getName}] No:$idx "); cb(Left(e))
      }

    Concurrent[F].delay{
      if (active.getAndSet(false))
        println(s"Task-${idx} is canceled and set to false!")
      else
        println(s"Task-${idx} is no longer active canceled")
    }
  }
```

Asynchronous & Cancelable in action

```
===== Start =======>
=> Concurrent-[scala-execution-context-global-180] Yes:5
=> Concurrent-[scala-execution-context-global-178] Yes:2
=> Concurrent-[scala-execution-context-global-181] Yes:4
Task-5 is no longer active canceled
Task-3 is no longer active canceled
Task-4 is no longer active canceled
Task-6 is no longer active canceled
=> Concurrent-[scala-execution-context-global-179] Yes:3
Task-2 is no longer active canceled
Task-8 is no longer active canceled
Task-7 is no longer active canceled
Task-9 is no longer active canceled
Task-10 is no longer active canceled
Task-1 is no longer active canceled
…
=> Concurrent-[scala-execution-context-global-179] Yes:8

Left(Hello!_2)
```

Asynchronous & Non-Cancelable

```
def processServiceResultAsync[F[_]: Async](idx: Int) : F[String] =
  Async[F].async{ (cb: Either[Throwable, String] => Unit) =>
    service.getResult().onComplete {
      case Success(s) => println(s"=> Async-[${Thread.currentThread.getName}]
Yes:$idx"); cb(Right(s+"_"+idx))
      case Failure(e) => println(s"=> Async-[${Thread.currentThread.getName}]
No:$idx "); cb(Left(e))
    }
  }
```

Asynchronous & Non-Cancelable

```
===== Start =======>
=> Async-[scala-execution-context-global-179] Yes:2
=> Async-[scala-execution-context-global-179] Yes:3
…
=> Async-[scala-execution-context-global-179] Yes:4
=> Async-[scala-execution-context-global-179] Yes:7
=> Async-[scala-execution-context-global-179] Yes:8
Left(Hello!_2)

===== End =======>
```

Cancelable

```
object IO {
  def cancelable[A](k: (Either[Throwable, A] => Unit) => CancelToken[IO]):
IO[A]
}
```

Cancelable

```
def beep(implicit SC : ScheduledExecutorService) =
  IO.cancelable[Unit] { cb =>
    lazy val beeper : Runnable = new Runnable {
      def run() = {
        println(s"[${Thread.currentThread.getName}] >> beep! <<")
      }
    }
    val beeperHandle = SC.scheduleAtFixedRate(beeper, 1, 1, SECONDS)

    IO {
      println(s"[${Thread.currentThread.getName}] >> Beeping canceled!
<<")  beeperHandle.cancel(false)
    }
  }

(for {
  f          <- beep.start
  aftermath <- timer.sleep(3 seconds) *> f.cancel *> IO(sc.shutdown)
 } yield aftermath).unsafeRunSync

// [pool-12-thread-1] >> beep! <<
// [pool-12-thread-1] >> beep! <<
// [pool-12-thread-1] >> beep! <<
// [ioapp-compute-1] >> Beeping canceled! <<
```

Error Handling

```
def convertToAsync[A:Numeric,F[_]:Async](value : A, compareBy: A, op: A => A) =
  Async[F].async[A]{ cb =>
    val G = Functor[Id]
    if (implicitly[Numeric[A]].gt(value, compareBy))
      cb(Right(G.fmap[A,A](value)(op(_))))
    else
      cb(Left(new Exception(s"No negative values: $value")))
  }

lazy val result4 =
  List(1,-2,3)
    .map(x =>
      convertToAsync[Int,IO](x, 0, increOp)
        .handleErrorWith(err =>
IO(increOp(math.abs(x))))).parSequence.unsafeRunSync
```

Resource management

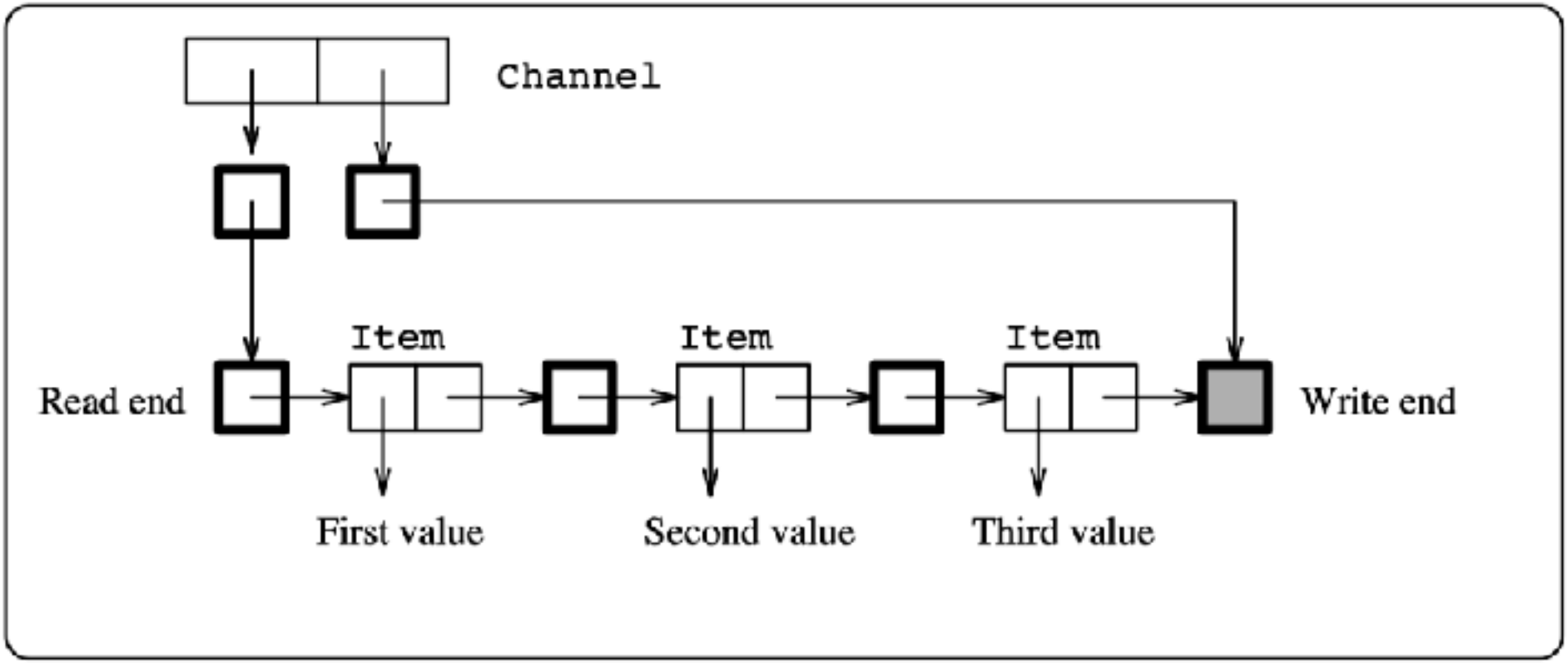
```
sealed abstract class Resource[F[_], A] {
  def use[B](f: A => F[B])(implicit F: Bracket[F, Throwable]):
F[B]
}
```

Safely Acquire & Release

```
def makeHttp1xResource = {
  val acquire = IO(Http())
  def release(http1x : HttpExt) = IO.unit /* nothing to release in Http1.x */
  Resource.make(acquire)(release)
}

def requestHttp1x(client: HttpExt, site: String) : IO[Unit] = {
 IO.fromFuture(IO.pure[Future[HttpResponse]](client.singleRequest(HttpRequest(uri = site))))
   .flatMap(result => IO(println(result.status)))
   .handleErrorWith(error => IO(println("Error"))) *> IO.unit
 }

 lazy val responses0 : IO[List[Unit]] =
   sites traverse (site => makeHttp1xResource.use(client => requestHttp1x(client, site)))
```

Concurrent Modelling

Concurrent Datatypes (I) - Buffered Channel



Encoding the Datatype

```
type Stream[A, F[_]] = MVar[F, Item[A,F]]

case class Item[A,F[_]](head : A, tail : Stream[A,F])

case class Channel[A, F[_]:Concurrent](
  reader : MVar[F, Stream[A,F]],
  writer : MVar[F, Stream[A,F]]
)
```

```scala
def newChannel[A,F[_]:Concurrent] : F[Channel[A,F]] =
  for {
    r <- MVar.uncancelableEmpty[F, Item[A,F]]
    a <- MVar.uncancelableOf[F, Stream[A,F]](r)
    b <- MVar.uncancelableOf[F, Stream[A,F]](r)
  } yield Channel(a, b)

def readChannel[A, F[_]:Concurrent](ch: Channel[A,F]) : F[A] = {
  for {
    stream <- ch.reader.take
    item   <- stream.read
    _      <- ch.reader.put(item.tail)
  } yield item.head
}

def writeChannel[A, F[_]:Concurrent](ch: Channel[A,F], value : A) : F[Unit] = {
  for {
    oldHole <- ch.writer.take
    newHole <- MVar.empty[F, Item[A,F]]
    _ <- oldHole.put(Item(value, newHole))
    _ <- ch.writer.put(newHole)
  } yield ()
}
```

Buffered Channel in Action

```
def sumTask =
  for {
    channel <- newChannel[Int,IO]
    _ <- writeChannel(channel, 1).start
    _ <- writeChannel(channel, 2).start
    _ <- writeChannel(channel, 3).start
    _ <- writeChannel(channel, 4).start
    _ <- writeChannel(channel, 5).start
    _ <- writeChannel(channel, 6).start
    _ <- writeChannel(channel, 7).start
    _ <- writeChannel(channel, 8).start
    _ <- writeChannel(channel, 9).start
    sum <- sumChannel(channel, 0L)
  } yield sum

sumTask.unsafeRunSync // returns 45
```

Concurrent Datatypes (II) - Skip Channel

```
case class SkipChan[A, F[_]](main : MVar[F, (A, List[MVar[F,Unit]])], sem: MVar[F,
Unit])

def newSkipChan[A:Monoid, F[_]:Concurrent] : F[SkipChan[A, F]] = ...

def getSkipChan[A, F[_]:Concurrent](sCh: SkipChan[A, F]) : F[A] = ...

def putSkipChan[A, F[_]:Concurrent](sCh: SkipChan[A, F], value:A ) : F[Unit] = {
  val F = implicitly[Concurrent[F]]
  for {
    pair <- sCh.main.take
    _ <- sCh.main.put((value, List()))
    f <- F.start(F.delay(pair._2.map(sem => sem.put(()))))
    _ <- f.join
  } yield { }
}
```

Skip Channel in action

```
val putNDrainTask = for {
  channel <- newSkipChan[Int, IO]
  _ <- putSkipChan(channel, 0)
  _ <- putSkipChan(channel, 1)
  a <- getSkipChan(channel)      // this should return "1"
  _ <- putSkipChan(channel, 2)
  _ <- putSkipChan(channel, 3)
  b <- getSkipChan(channel)      // this should return "3"
  _ <- putSkipChan(channel, 4)
  _ <- IO.sleep(10.millis)
  c <- getSkipChan(channel)      // this should return "4"
} yield (a, b, c)

putNDrainTask.unsafeRunSync // returns (1,3,4)
```

Logging with Cats-effect

Setting the stage...

```scala
def fib(n: Int, a: Long = 0, b: Long = 1)(implicit cs: ContextShift[IO]): IO[Long] =
  IO.suspend {
    if (n == 0) IO.pure(a) else {

      val next = fib(n - 1, b, a + b)
      // Every 100 cycles, introduce a logical thread fork
      if (n % 100 == 0)
        cs.shift *> next
      else
        next
    }
  }
```

Logging ...

```
def fibW(n: Int, a: Long = 0, b: Long = 1)
        (implicit cs: ContextShift[IO],
         W: IO[Writer[List[String],Long]]): IO[Writer[List[String],Long]] =
  IO.suspend {
    if (n == 0) W >>= { writer => IO{ Writer(writer.written :+ s"=> Done $a", a) } } else {

      def next(WW: IO[Writer[List[String], Long]]) =
        WW >>= {writer => fibW(n - 1, b, a + b)(cs, IO{writer.tell(List(s"=> Next $a"))})}

      // Every 100 cycles, introduce a logical thread fork
      if (n % 10 == 0)
        W >>= { writer => cs.shift *> next(IO{writer.tell(List(s"=> Context Shift !"))}) }
      else
        W >>= { writer => next(IO{writer.tell(List(s"=> regular "))}) }
    }
  }
```

Do I have to do this every time ?

```scala
implicit def unsafeLogger[F[_]: Sync] = Slf4jLogger.getLogger[F]

def fib[F[_]:Sync:ContextShift](n: Int, a: Long = 0, b: Long = 1): F[Long] =
  Sync[F].suspend {
    if (n == 0) Logger[F].info(s"=> Done $a") *> Sync[F].pure(a) else {

      val next = Logger[F].info(s"=> Next $a") *> fib(n - 1, b, a + b)
      // Every 100 cycles, introduce a logical thread fork
      if (n % 100 == 0)
        Logger[F].info(s"Context Shift!") *> ContextShift[F].shift *> next
      else
        next
    }
  }

def doSomething[F[_]:Sync:ContextShift](n : Int, a: Long = 0, b: Long = 1): F[Long] =
  Logger[F].info("Logging Started.") *>
  fib(n, a, b) >>= { result => Logger[F].info("Logging Ended.") *> Sync[F].pure(result) }
```

Gotchas

Stack safety

```
val k : Kleisli[Id, Int, Int] = Kleisli{ (x: Int) => k(x) }
val iok = IO(k).map(_(2)).unsafeRunSync // run it!
...
  at scala.runtime.java8.JFunction1$mcII$sp.apply(JFunction1$mcII$sp.java:23)
  at cats.data.Kleisli.apply(Kleisli.scala:119)
  at .$anonfun$k$1(<console>:24)
  at scala.runtime.java8.JFunction1$mcII$sp.apply(JFunction1$mcII$sp.java:23)
  at cats.data.Kleisli.apply(Kleisli.scala:119)
```

Cancellation assumes no-blocking

```
def problematic(f: Fiber[IO,Int], sem1: Semaphore[IO], sem2: Semaphore[IO]) = {

  val Timeout = 2.milli
  val promise = Promise[Int] // will never finish
  val loooongIO : IO[Int] = IO.fromFuture(IO(promise.future))

  (
    IO(println("Acquiring")) *> // kinda like logging but its not; don't do this.
    sem1.acquire *>
    opThatMightVomits *>
    sem2.acquire *>
    IO(println("Acquired"))
  ).handleErrorWith{err => IO(println("Error caught, cancelling Fiber...")) *> f.cancel}
   .bracket{ _ =>
     IO(print("Start..")) *> IO(0) >>=
     ((x:Int) => IO(x+1)) >>=
     ((x:Int) => IO(println(s"intermediate evaluation: $x")) *>
       loooongIO.guaranteeCase{
         case ExitCase.Completed  => IO(println("Completed"))
         case ExitCase.Canceled   => IO(println("Canceled: Before")) *> IO.sleep(Timeout) *>
IO(println("Canceled: After"))
         case ExitCase.Error(err) => IO(println(s"Error($err)"))
       }
     )
   }{ _ => sem2.release *> sem1.release *> IO(println("Released!"))} // this is never
called, what happens to this?
}
```

Just because it compiles ...

```scala
def fib(n: Int, a: Long = 0, b: Long = 1): IO[Long] =
  IO.suspend {
    if (n <= 0) IO.pure(a) else {
      val next = fib(n - 1, b, a + b)

      // Every 3-th cycle, check cancellation status
      if (n % 3 == 0)
        IO.cancelBoundary *> next
      else
        next
    }
    // Cancellation logic is missing ;)
  }

val fibTask =
  for {
    f <- fib(100).start
    _ <- f.cancel /* Just because it compiles does not mean it will behave
properly; know your types! */
    r <- f.join
  } yield r
```

Thank You JCConf 2019 !

@RaymondTayBL

@https://www.linkedin.com/in/raymondtayboonleong

References

=> code repo : https://github.com/raymondtay/JCConf_2019

=> cats effect : https://typelevel.org/cats-effect

=> cats effect tutorial : https://typelevel.org/cats-effect/tutorial/tutorial.html

=> monix : http://monix.io/

=> fs2 : https://fs2.io

=> log4cats : https://christopherdavenport.github.io/log4cats/

=> concurrent Haskell paper : https://www.microsoft.com/en-us/research/wp-content/uploads/1996/01/concurrent-haskell.pdf