BATCHING MONADS

COMBINING FREE MONAD AND FREE APPLICATIVE FTW

Cary Robbins

March 20, 2018

BoomTown!

PROBLEM

We need to integrate with Google Calendar

However, we project that we will be rate limited.

SOLUTION

Google provides a batch API, let's just use that!

RESTRICTIONS

The Google batch API only permits 50 requests per batch request.

THE PROBLEM WITH MONADS

```scala
for {
  x <- process1
  y <- process2
  z <- process3
} yield (x, y, z)

// Desugars to

process1.flatMap(x =>
  process2.flatMap(y =>
    process3.map(z =>
      (x, y, z)
    )
  )
)
```

APPLICATIVE TO THE RESCUE

```
(process1, process2, process3).tupled

// Rougly equivalent to

val ff = ((a: A) => (b: B) => (c: C) => (a, b, c)).pure[F]
// ff: F[A => B => C => (A, B, C)]

val f1 = ff.ap(process1)
// f1: F[B => C => (A, B, C)]

val f2 = f1.ap(process2)
// f2: F[C => (A, B, C)]

val f3 = f2.ap(process3)
// f3: F[(A, B, C)]
```

THE FREE MONAD

```
sealed abstract class Free[S[_], A]

final case class Pure[S[_], A](a: A)       extends Free[S, A]

final case class Suspend[S[_], A](a: S[A]) extends Free[S, A]

final case class FlatMapped[S[_], B, C](
  c: Free[S, C], f: C => Free[S, B]
) extends Free[S, B]
```

FREE APPLICATIVE

```scala
sealed abstract class FreeApplicative[F[_], A]

final case class Pure[F[_], A](a: A)     extends FreeApplicative[F, A]

final case class Lift[F[_], A](fa: F[A]) extends FreeApplicative[F, A]

final case class Ap[F[_], P, A](
  fn: FreeApplicative[F, P => A], fp: FreeApplicative[F, P]
) extends FreeApplicative[F, A]
```

A TALE OF TWO ALGEBRAS

```
object GoogleCalendarClient {
  sealed trait Action[A]
  final case class CalendarsGet(...) extends Action[Option[GCalendar]]
  final case class EventsGet(...)    extends Action[Option[GCalendarEvent]]
  final case class EventsInsert(...) extends Action[GCalendarEvent]
  final case class EventsUpdate(...) extends Action[GCalendarEvent]
  final case class EventsDelete(...) extends Action[Unit]
}
```

A TALE OF TWO ALGEBRAS

```
object ExternalCalendarClient {
  sealed trait Action[A]
  final case class AddEvent(...)    extends Action[Unit]
  final case class DeleteEvent(...) extends Action[Unit]
  final case class EventExists(...) extends Action[Boolean]
  final case class UpdateEvent(...) extends Action[Unit]
}
```

BoomTown!

APPLICATIVE REQUESTS

```
import GoogleCalendarClient.{Methods => G}

val request: Request[(Option[GCalendar], GCalendarEvent)] =
  (G.calendars.get(...), G.events.insert(...))).tupled

val response: F[(Option[GCalendar], GCalendarEvent)] =
  client.run(request)
```

COMMANDS AND REQUESTS

```
// We need to follow this pattern for ExternalCalendarClient as well.
object GoogleCalendarClient {
  sealed trait Command[A]
  final case class Pure[A](value: A)          extends Command[A]
  final case class Exec[A](action: Action[A]) extends Command[A]

  type Request[A] = FreeApplicative[Command, A]

  def exec[A](action: Action[A]): Request[A] =
    FreeApplicative.lift(Command.Exec(action))
}
```

COMMANDS AND REQUESTS

```
// We need to follow this pattern for ExternalCalendarClient as well.
object GoogleCalendarClient {
  object Methods {
    object calendars {
      def get(...):    Request[Option[GCalendar]]      = exec(CalendarsGet(...))
    }
    object events {
      def insert(...): Request[GCalendarEvent]         = exec(EventsInsert(...))
      def update(...): Request[GCalendarEvent]         = exec(EventsUpdate(...))
      def delete(...): Request[Unit]                   = exec(EventsDelete(...))
      def get(...):    Request[Option[GCalendarEvent]] = exec(EventsGet(...))
    }
  }
}
```

COMMANDS AND REQUESTS

```scala
import GoogleCalendarClient.{Methods => G}

val request: Request[Option[GCalendar]] = G.calendars.get(...)

val response: F[Option[GCalendar]] = client.run(request)
```

APPLICATIVE REQUESTS REVISITED

```
import GoogleCalendarClient.{Methods => G}

val request: Request[(Option[GCalendar], GCalendarEvent)] =
  (G.calendars.get(...), G.events.insert(...)).tupled

val response: F[(Option[GCalendar], GCalendarEvent)] =
  client.run(request)
```

HTTP CLIENT INTERFACE

```
trait GoogleCalendarClient[F[_]] {
  def run[A](r: GoogleCalendarClient.Request[A]): F[A]
}
```

IMPLEMENTING THE INTERFACE

```
final class BatchingGoogleCalendarClient[F[_]](
  implicit F: MonadError[F, Throwable]
) extends GoogleCalendarClient[F] {
  override def run[A](r: GoogleCalendarClient.Request[A]): F[A] = ???
}
```

FUNCTIONK

```
trait FunctionK[F[_], G[_]] {
  def apply[A](fa: F[A]): G[A]
}

type ~>[F[_], G[_]] = FunctionK[F, G]
```

FUNCTIONK

```scala
val optionToList = new (Option ~> List) {
  override def apply[A](fa: Option[A]): List[A] = fa match {
    case None    => Nil
    case Some(x) => List(x)
  }
}

// Using kind-projector

val optionToList = λ[Option ~> List] {
  case None    => Nil
  case Some(x) => List(x)
}
```

FREEAPPLICATIVE#COMPILE

```
sealed abstract class FreeApplicative[F[_], A] {
  ...
  /**
    * Interpret this algebra into another algebra.
    * Stack-safe.
    */
  def compile[G[_]](f: F ~> G): FreeApplicative[G, A] = ...
}
```

BABY'S FIRST INTERPRETER

```
type CommandWithId[A] = (UUID, Command[A])

val idGenCompiler = λ[Command ~> CommandWithId] {
  (c: Command[A]) => (UUID.randomUUID(), c)
}

r: Request[A] = ...
r: FreeApplicative[Command, A] = ...
val commandsWithIds: FreeApplicative[CommandWithId, A] = r.compile(idGenCompiler)
```

FREEAPPLICATIVE#ANALYZE

```
sealed abstract class FreeApplicative[F[_], A] {
  ...
  /** Interpret this algebra into a Monoid. */
  def analyze[M: Monoid](f: F ~> λ[α => M]): M = ...
}
```

ACCUMULATING INTERPRETER

```scala
type Requests[_] = Vector[(UUID, Exec[_])]

val requestsBuilder = λ[CommandWithId ~> Requests] {
  case   (_, _: Pure[_]) => Vector.empty
  case x@(_, _: Exec[_]) => Vector(x)
}


val commandsWithIds: FreeApplicative[CommandWithId, A] = ...
val requests: Vector[(UUID, Exec[_])] = commandsWithIds.analyze(requestsBuilder)
```

BoomTown!

FREEAPPLICATIVE#FOLDMAP

```
sealed abstract class FreeApplicative[F[_], A] {
  ...
  /**
    * Interprets/Runs the sequence of operations using the semantics of
    * `Applicative` G[_]. Tail recursive.
    */
  def foldMap[G[_]](f: F ~> G)(implicit G: Applicative[G]): G[A] = ...
}
```

READER INTERPRETER

```scala
type Env       = Map[UUID, EncodedResponse]
type Reader[A] = Kleisli[Either[Throwable, ?], Env, A]

val readerInterpreter = λ[CommandWithId ~> Reader] {
  case (_, Pure(v)) => Kleisli.pure(v)
  case (id, Exec(action)) =>
    Kleisli[Either[Throwable, ?], Env, A](
      _.get(id) match {
        case Some(response) => decode[A](action, response)
        case None => Left(new NoSuchElementException(...))
      }
    )
}
val commandsWithIds: FreeApplicative[CommandWithId, A] = ...
val reader: Reader[A] = commandsWithIds.foldMap(readerInterpreter)
```

PUTTING IT ALL TOGETHER

```
// Remember: Request[A] =:= FreeApplicative[Command, A]
override def run[A](r: GoogleCalendarClient.Request[A]): F[A] = {
  val commandsWithIds: FreeApplicative[CommandWithId, A] = r.compile(idGenCompiler)
  val requests: Vector[(UUID, Exec[_])] = commandsWithIds.analyze(requestsBuilder)
  val reader: Reader[A] = commandsWithIds.foldMap(readerInterpreter)
  val batches: Vector[Vector[(UUID, Exec[_])]] = requests.grouped(batchLimit).toVector
  val responsesF: Vector[F[Env]] = batches.map(runBatch)
  val sequenced: F[Vector[Env]] = responsesF.sequence
  val envF: F[Env] = sequenced.map(_.foldLeft(Map.empty: Env)(_ ++ _))
  envF.flatMap(env => eitherToF(reader(env)))
}

def runBatch(rs: Vector[(UUID, Exec[_])]): F[Env] = ...
def eitherToF[A](either: Either[Throwable, A]): F[A] = ...
```

APPLICATIVE REQUESTS REVISITED AGAIN

```scala
import GoogleCalendarClient.{Methods => G}

val request: Request[(Option[GCalendar], GCalendarEvent)] =
  (G.calendars.get(...), G.events.insert(...)).tupled

val response: F[(Option[GCalendar], GCalendarEvent)] =
  client.run(request)
```

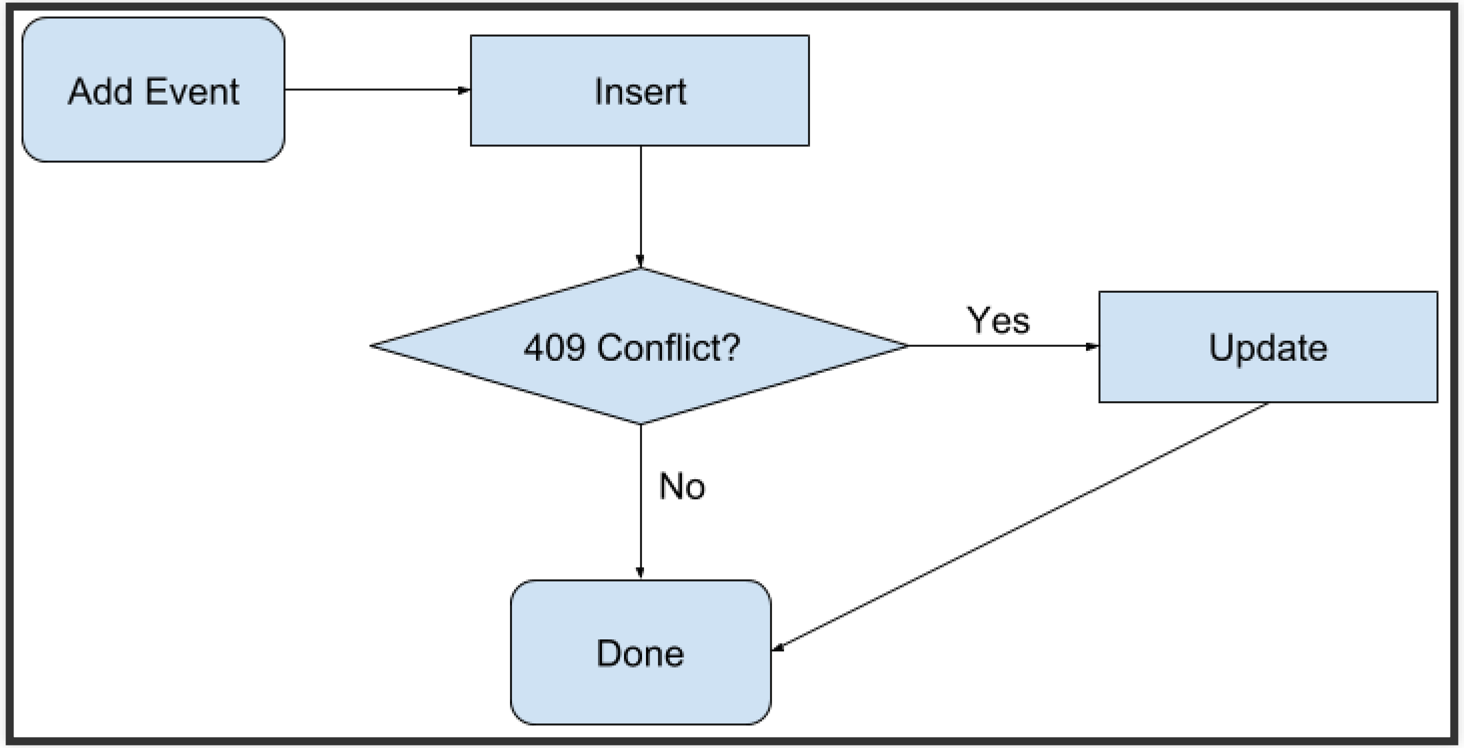
BUT WAIT...

What does this have to do with batching Monads?

ALGEBRA SHMALGEBRA

```scala
object ExternalCalendarClient {
  sealed trait Action[A]
  final case class AddEvent(...)    extends Action[Unit]
  final case class DeleteEvent(...) extends Action[Unit]
  final case class EventExists(...) extends Action[Boolean]
  final case class UpdateEvent(...) extends Action[Unit]
}
```

BUSINESS LOGIC

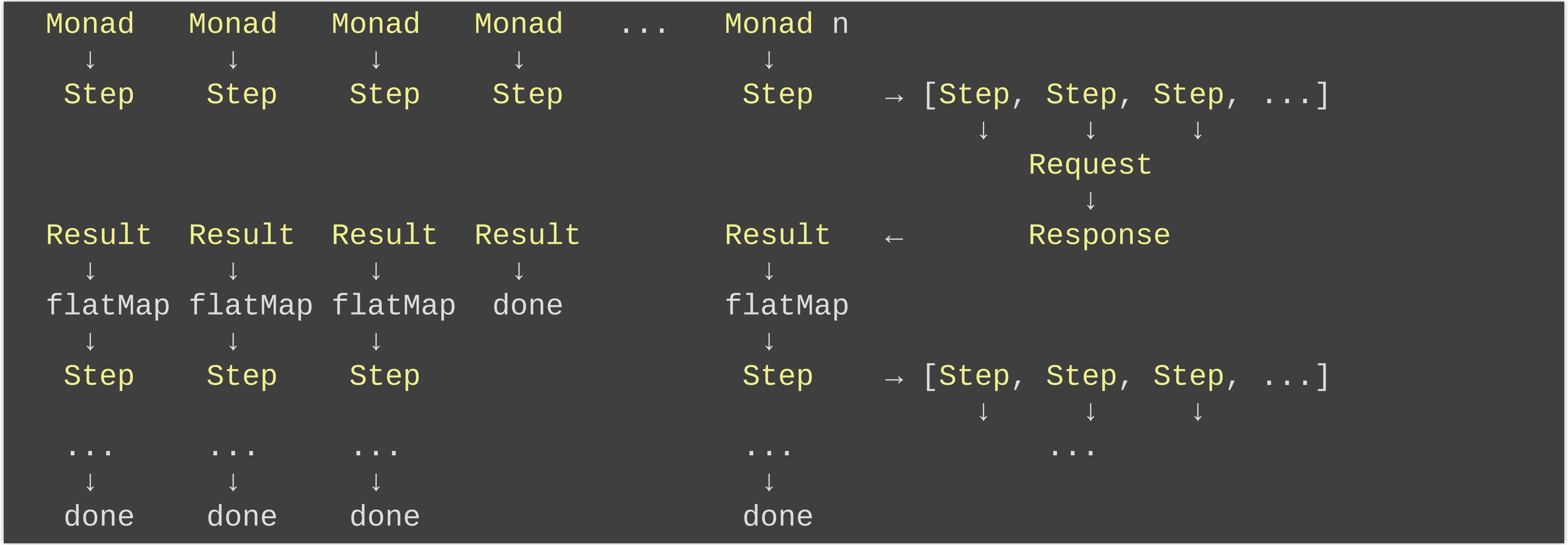


BoomTown!

BUSINESS LOGIC MAKES FP CRY 😫

```
def addEvent =
  insert.flatMap(res =>
    if (res.conflict) update
    else res
  )
```

WHAT IF...



```
Monad   Monad   Monad   Monad   ...   Monad n
  ↓       ↓       ↓       ↓             ↓
 Step    Step    Step    Step          Step    → [Step, Step, Step, ...]
                                                    ↓     ↓     ↓
                                                       Request
                                                          ↓
Result  Result  Result  Result        Result   ←       Response
  ↓       ↓       ↓       ↓             ↓
flatMap flatMap flatMap  done         flatMap
  ↓       ↓       ↓                     ↓
 Step    Step    Step                  Step    → [Step, Step, Step, ...]
                                                    ↓     ↓     ↓
 ...     ...     ...                   ...              ...
  ↓       ↓       ↓                     ↓
 done    done    done                  done
```

BATCHED MONADS EXEMPLIFIED

```
addEvent eventExists updateEvent
   ↓         ↓            ↓
 insert    exists       update  → [insert, exists, update]
                                     ↓       ↓       ↓
                                          Request
                                             ↓
 Conflict Pure(true)  Not Found ←         Response
   ↓                     ↓
flatMap               flatMap
   ↓                     ↓
 update                insert   →    [update, insert]
                                             ↓
                                          Request
                                             ↓
Pure(())              Pure(())  ←        Response
```

FREE INCEPTION

```
type GStep[A] = GoogleCalendarClient.Request[A]
// FreeApplicative[GoogleCalendarClient.Command, A]

type GLogic[A] = Free[GStep, A]
// Free[λ[a => FreeApplicative[GoogleCalendarClient.Command[a]], A]
object GLogic {
  def pure[A](a: A): GLogic[A]            = Free.pure(a)
  def suspend[A](fa: GStep[A]): GLogic[A] = Free.liftF(fa)
}
```

FREE INCEPTION

```
def actionToGRequest[A](action: ExternalCalendarClient.Action[A]): GLogic[A] = {
  action match {
    case e: Action.AddEvent    => addEvent(e)
    case e: Action.DeleteEvent => deleteEvent(e)
    case e: Action.EventExists => eventExists(e)
    case e: Action.UpdateEvent => updateEvent(e)
  }
}
```

FREE INCEPTION

```scala
def addEvent(e: Action.AddEvent): GLogic[Unit] = {
  GLogic.suspend(G.events.insert(...)).flatMap { res =>
    if (res.isConflict) GLogic.suspend(G.events.update(...))
    else GLogic.pure(())
  }
}
```

ANOTHER CLIENT INTERFACE

```
trait ExternalCalendarClient[F[_]] {
  def run[A](r: ExternalCalendarClient.Request[A]): F[A]
}

class GoogleExternalCalendarClient[F[_]](
  implicit F: MonadError[F, Throwable]
) extends ExternalCalendarClient[F] {
```

FREE#RESUME

```scala
sealed abstract class Free[S[_], A] {
  ...
  /** Evaluate a single layer of the free monad. */
  def resume(implicit S: Functor[S]): Either[S[Free[S, A]], A] = ...
}
```

INTERPRETER INPUTS

```
type Inputs = Map[UUID, Either[GStep[GLogic[_]], Any]]

def getInitialInputs(actions: Vector[(UUID, Action[_])]): Inputs =
  actions.iterator.map { case (id, a) => (id, actionToGRequest(a).resume) }.toMap
```

INTERPRETER EVALUATION

```
type ResultMap = Map[UUID, Any]

/** Monadic recursive function for building our ResultsMap. */
def buildResultsRec(inputs: Inputs): F[Either[Inputs, ResultMap]] = {
  val completed = inputs.collect { case (cmdId, Right(x)) => (cmdId, x) }
  // If all of our steps are completed, we're done.
  if (completed.size == inputs.size) {
    F.pure(Either.right(completed))
  } else {
    // Obtain the next set of steps and their indices so we can run them and
    // bind the results to the next set of inputs.
    val steps: Vector[GStep[(CmdId, GLogic[_])]] = collectSteps(inputs)
    runSteps(steps).map(rs => Either.left(rebuildInputs(rs, inputs)))
  }
}
```

FREE#FOLD

```scala
sealed abstract class Free[S[_], A] {
  /**
   * Catamorphism. Run the first given function if Pure, otherwise,
   * the second given function.
   */
  def fold[B](r: A => B, s: S[Free[S, A]] => B)(implicit S: Functor[S]): B = ...
}
```

INTERPRETER EVALUATION

```
type Inputs = Map[UUID, Either[GStep[GLogic[_]], Any]]

def rebuildInputs(
  responses: Vector[(CmdId, GLogic[_])],
  inputs: Inputs
): Inputs = responses.foldLeft(inputs) { case (accInputs, (cmdId, gLogic)) =>
  val newValue = gLogic.fold(Either.right, Either.left)
  accInputs.updated(index, newValue)
}
```

DYNAMIC EXTRACTOR

```scala
type ResultMap = Map[UUID, Any]

def extractor(results: ResultMap) = λ[CommandWithId ~> F] {
  case (_,     Command.Pure(a))       => F.pure(a)
  case (cmdId, Command.Exec(action)) =>

    def extractDynamic[B](implicit ct: ClassTag[B]): F[B] = ...

    action match {
      case _: Action.AddEvent     => extractDynamic
      case _: Action.UpdateEvent => extractDynamic
      case _: Action.EventExists => extractDynamic
      case _: Action.DeleteEvent => extractDynamic
    }
}
```

PUTTING IT ALL TOGETHER...AGAIN

```scala
override def run[A](request: ExternalCalendarClient.Request[A]): F[A] = {
  val requestWithIds: RequestWithIds[A] = request.compile(idGenCompiler)
  val actions: Vector[(UUID, Action[_])] = requestWithIds.analyze(actionAccumulator)
  F.tailRecM(getInitialInputs(actions))(buildResultsRec).flatMap { resultMap =>
    requestWithIds.foldMap(extractor(resultMap))
  }
}
```

WHEW 😅

Questions?