Unfolding programs with composable interpreters

errr … with a little change of plan

```
f(a)
```

```
def apply[A, B](f: A ⇒ B)(a: A): B = f(a)
```

```
def uncurry[A, B, C](f: A ⇒ B ⇒ C)(ab: (A, B)): C =
  f(ab._1)(ab._2)
```

```
def pair[F[_], G[_], A, B](fab: F[A ⇒ B])(ga: G[A]): B =
  ???
```

```
trait Pairing[F[_], G[_]] {
  def pair[A, B](fa: F[A])(gab: G[A ⇒ B]): B
}
```

```
trait Pairing[F[_], G[_]] {
  def pair[A, B](fa: F[A])(gab: G[A ⇒ B]): B
}
```

```
trait Pairing[F[_], G[_]] {
  def pair[A, B, C](fa: F[A], gb: G[B])(f: (A, B) ⇒ C): C
}
```

```
type Id[A] = A

new Pairing[Id, Id] {
  def pair[A, B, C](fa: Id[A], gb: Id[B])(f: (A, B) ⇒ C) =
    f(fa, gb)
}
```

```
new Pairing[(X, ?), X ⇒ ?] {
  def pair[A, B, C](fa: (X, A), gb: X ⇒ B)(f: (A, B) ⇒ C) =
    f(fa._2, gb(fa._1))
}
```

```
new Pairing[(X, ?), X ⇒ ?] {
  def pair[A, B, C](fa: (X, A), gb: X ⇒ B)(f: (A, B) ⇒ C) =
    f(fa._2, gb(fa._1))
}
```

```
new Pairing[(X, ?), X ⇒ ?] {
  def pair[A, B, C](fa: (X, A), gb: X ⇒ B)(f: (A, B) ⇒ C) =
    f(fa._2, gb(fa._1))
}
```

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

```scala
case class Product[F[_], G[_], A](
  fst: F[A],
  snd: G[A])
```

```
case class Coproduct[F[_], G[_], A](run: Either[F[A], G[A]])
case class Product[F[_], G[_], A](fst: F[A], snd: G[A])

def product[F1[_], F2[_], G1[_], G2[_]](implicit
  P1: Pairing[F1, G1],
  P2: Pairing[F2, G2]
) = new Pairing[Coproduct[F1, F2, ?], Product[G1, G2, ?]] {
  def pair[A, B, C](
    ffa: Coproduct[F1, F2, A],
    ggb: Product[G1, G2, B]
  )(f: (A, B) ⇒ C) =
   ffa.run match {
      case Left(f1a)  ⇒ P1.pair(f1a, ggb.fst)(f)
      case Right(f2a) ⇒ P2.pair(f2a, ggb.snd)(f)
    }
}
```

```scala
case class Coproduct[F[_], G[_], A](run: Either[F[A], G[A]])
case class Product[F[_], G[_], A](fst: F[A], snd: G[A])

def product[F1[_], F2[_], G1[_], G2[_]](implicit
  P1: Pairing[F1, G1],
  P2: Pairing[F2, G2]
) = new Pairing[Coproduct[F1, F2, ?], Product[G1, G2, ?]] {
  def pair[A, B, C](
    ffa: Coproduct[F1, F2, A],
    ggb: Product[G1, G2, B]
  )(f: (A, B) ⇒ C) =
   ffa.run match {
      case Left(f1a)  ⇒ P1.pair(f1a, ggb.fst)(f)
      case Right(f2a) ⇒ P2.pair(f2a, ggb.snd)(f)
    }
}
```

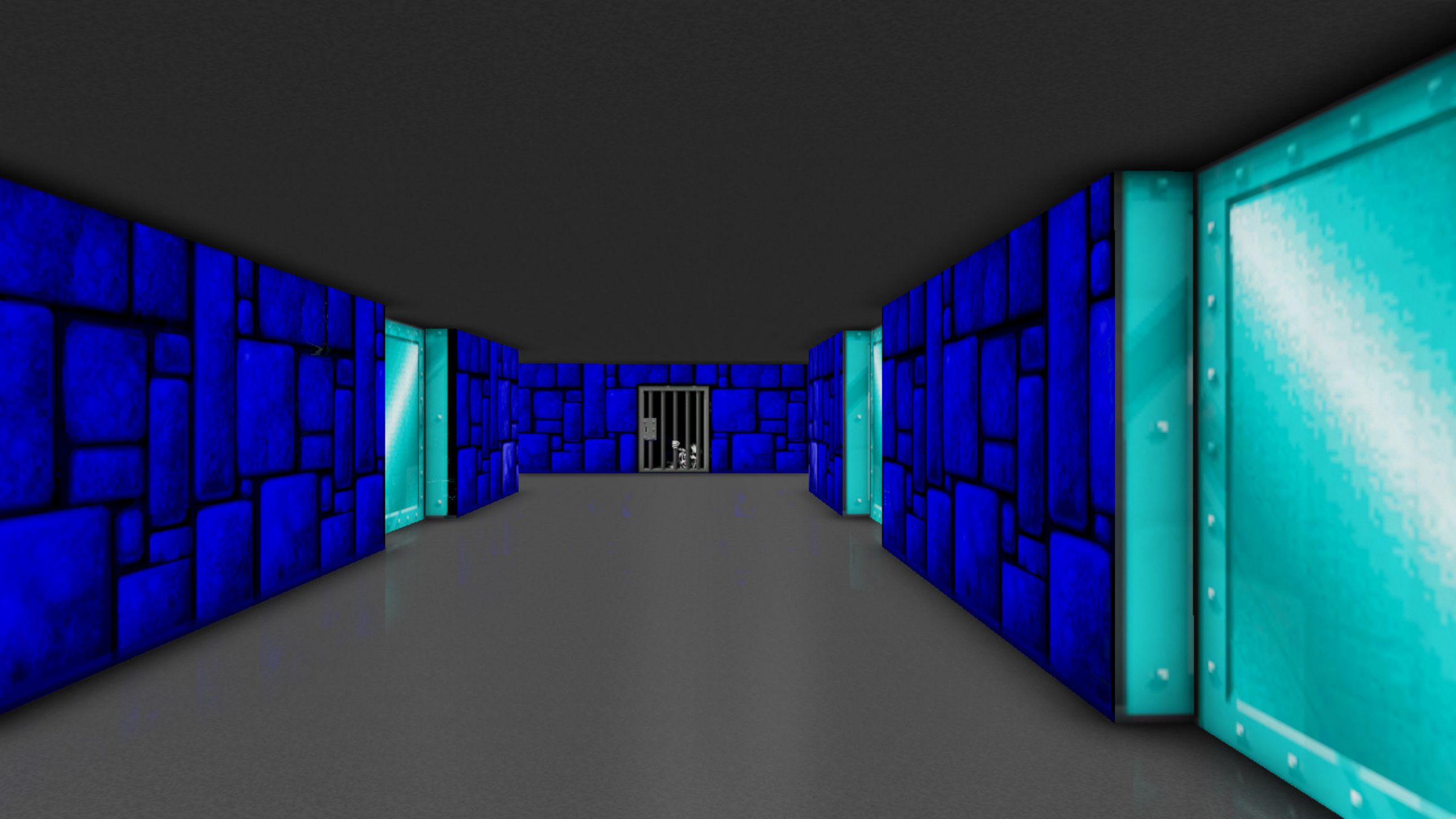
```scala
case class Coproduct[F[_], G[_], A](run: Either[F[A], G[A]])
case class Product[F[_], G[_], A](fst: F[A], snd: G[A])

def product[F1[_], F2[_], G1[_], G2[_]](implicit
  P1: Pairing[F1, G1],
  P2: Pairing[F2, G2]
) = new Pairing[Coproduct[F1, F2, ?], Product[G1, G2, ?]] {
  def pair[A, B, C](
    ffa: Coproduct[F1, F2, A],
    ggb: Product[G1, G2, B]
  )(f: (A, B) ⇒ C) =
   ffa.run match {
      case Left(f1a)  ⇒ P1.pair(f1a, ggb.fst)(f)
      case Right(f2a) ⇒ P2.pair(f2a, ggb.snd)(f)
    }
}
```

```scala
case class Coproduct[F[_], G[_], A](run: Either[F[A], G[A]])
case class Product[F[_], G[_], A](fst: F[A], snd: G[A])

def product[F1[_], F2[_], G1[_], G2[_]](implicit
  P1: Pairing[F1, G1],
  P2: Pairing[F2, G2]
) = new Pairing[Coproduct[F1, F2, ?], Product[G1, G2, ?]] {
  def pair[A, B, C](
    ffa: Coproduct[F1, F2, A],
    ggb: Product[G1, G2, B]
  )(f: (A, B) ⇒ C) =
   ffa.run match {
      case Left(f1a)  ⇒ P1.pair(f1a, ggb.fst)(f)
      case Right(f2a) ⇒ P2.pair(f2a, ggb.snd)(f)
    }
}
```

```scala
case class Coproduct[F[_], G[_], A](run: Either[F[A], G[A]])
case class Product[F[_], G[_], A](fst: F[A], snd: G[A])

def product[F1[_], F2[_], G1[_], G2[_]](implicit
  P1: Pairing[F1, G1],
  P2: Pairing[F2, G2]
) = new Pairing[Coproduct[F1, F2, ?], Product[G1, G2, ?]] {
  def pair[A, B, C](
    ffa: Coproduct[F1, F2, A],
    ggb: Product[G1, G2, B]
  )(f: (A, B) ⇒ C) =
   ffa.run match {
      case Left(f1a)  ⇒ P1.pair(f1a, ggb.fst)(f)
      case Right(f2a) ⇒ P2.pair(f2a, ggb.snd)(f)
    }
}
```

```scala
case class Coproduct[F[_], G[_], A](run: Either[F[A], G[A]])
case class Product[F[_], G[_], A](fst: F[A], snd: G[A])

def product[F1[_], F2[_], G1[_], G2[_]](implicit
  P1: Pairing[F1, G1],
  P2: Pairing[F2, G2]
) = new Pairing[Coproduct[F1, F2, ?], Product[G1, G2, ?]] {
  def pair[A, B, C](
    ffa: Coproduct[F1, F2, A],
    ggb: Product[G1, G2, B]
  )(f: (A, B) ⇒ C) =
   ffa.run match {
      case Left(f1a)  ⇒ P1.pair(f1a, ggb.fst)(f)
      case Right(f2a) ⇒ P2.pair(f2a, ggb.snd)(f)
    }
}
```

```scala
case class Coproduct[F[_], G[_], A](run: Either[F[A], G[A]])
case class Product[F[_], G[_], A](fst: F[A], snd: G[A])

def product[F1[_], F2[_], G1[_], G2[_]](implicit
  P1: Pairing[F1, G1],
  P2: Pairing[F2, G2]
) = new Pairing[Coproduct[F1, F2, ?], Product[G1, G2, ?]] {
  def pair[A, B, C](
    ffa: Coproduct[F1, F2, A],
    ggb: Product[G1, G2, B]
  )(f: (A, B) ⇒ C) =
   ffa.run match {
      case Left(f1a)  ⇒ P1.pair(f1a, ggb.fst)(f)
      case Right(f2a) ⇒ P2.pair(f2a, ggb.snd)(f)
    }
}
```



Comonads

I don’t know

Comonads as Spaces

by Phil Freeman on 2016/08/07

https://blog.functorial.com/posts/2016-08-07-Comonads-As-Spaces.html


World Tracker
Choose Research
Feudalism
Turns: 2
Build 6 Farms.
31
17
10 Athens
3
20
10
6 Pergamon
24
Coast
Fish
Movement Cost: 1
Continent: none
2 Food
1 Gold
Choose Research

```
abstract class Comonad[F[_]]
```

```
abstract class Comonad[W[_]] {
  def extract[A](wa: W[A]): A
}
```

```
abstract class Comonad[W[_]] {
  def extract[A](wa: W[A]): A

  def extend[A, B](wa: W[A])(f: W[A] ⇒ B): W[B]
}
```

```
abstract class Comonad[W[_]] {
  def extract[A](wa: W[A]): A

  def extend[A, B](wa: W[A])(f: W[A] ⇒ B): W[B]

  def duplicate[A](wa: W[A]): W[W[A]]
}
```

```
abstract class Comonad[W[_]] {
  def extract[A](
    wa: W[A]): A

  def extend[A, B](wa: W[A])(
    f: W[A] ⇒ B): W[B]

  def duplicate[A](
    wa: W[A]): W[W[A]]
}
```

```
abstract class Monad[M[_]] {
  def pure[A](
    a: A): M[A]

  def bind[A, B](ma: M[A])(
    f: A ⇒ M[B]): M[B]

  def flatten[A](
    mma: M[M[A]]): M[A]
}
```

```scala
def move[M[_], W[_]: Comonad, A](
  action: M[Unit],
  wmap0: W[A]
)(implicit P: Pairing[M, W]): W[A] =
  P.pair(action, wmap0.duplicate)((_, wmap) ⇒ wmap)
```

```
def move[M[_], W[_]: Comonad, A](
  action: M[Unit],
  wmap0: W[A]
)(implicit P: Pairing[M, W]): W[A] =
  P.pair(action, wmap0.duplicate)((_, wmap) ⇒ wmap)
```

```
def move[M[_], W[_]: Comonad, A](
  action: M[Unit],
  wmap0: W[A]
)(implicit P: Pairing[M, W]): W[A] =
  P.pair(action, wmap0.duplicate)((_, wmap) ⇒ wmap)
```

```scala
def move[M[_], W[_]: Comonad, A](
  action: M[Unit],
  wmap0: W[A]
)(implicit P: Pairing[M, W]): W[A] =
  P.pair(action, wmap0.duplicate)((_, wmap) ⇒ wmap)
```

```
case class Zipper[A](
  pred  : Stream[A],
  focus : A,
  succ  : Stream[A]
) {
  def left: Zipper[A] = ???
  def right: Zipper[A] = ???
}
```

```
case class Zipper[A](
  pred  : Stream[A],
  focus : A,
  succ  : Stream[A]
) {
  def left: Zipper[A] = ???
  def right: Zipper[A] = ???
}
```

```
case class Zipper[A](
  pred  : Stream[A],
  focus : A,
  succ  : Stream[A]
) {
  def left: Zipper[A] = ???
  def right: Zipper[A] = ???
}
```

```
sealed abstract class ZipMove[A]
case object ZipStop[A](value: A)          extends ZipMove[A]
case class  ZipLeft[A](next: ZipMove[A])  extends ZipMove[A]
case class  ZipRight[A](next: ZipMove[A]) extends ZipMove[A]
```

```
sealed abstract class ZipMove[A]
case object ZipStop[A](value: A)          extends ZipMove[A]
case class  ZipLeft[A](next: ZipMove[A])  extends ZipMove[A]
case class  ZipRight[A](next: ZipMove[A]) extends ZipMove[A]
```

```
sealed abstract class ZipMove[A]
case object ZipStop[A](value: A)          extends ZipMove[A]
case class  ZipLeft[A](next: ZipMove[A])  extends ZipMove[A]
case class  ZipRight[A](next: ZipMove[A]) extends ZipMove[A]
```

```
sealed abstract class ZipMove[A]
case object ZipStop[A](value: A)          extends ZipMove[A]
case class  ZipLeft[A](next: ZipMove[A])  extends ZipMove[A]
case class  ZipRight[A](next: ZipMove[A]) extends ZipMove[A]
```

```
new Pairing[ZipMove, Zipper] {
  def pair[A, B, C](
    fa: ZipMove[A],
    gb: Zipper[B]
  )(f: (A, B) ⇒ C) =
    fa match {
      case ZipStop(a)     ⇒ f(a, gb.focus)
      case ZipLeft(next)  ⇒ pair(next, gb.left)(f)
      case ZipRight(next) ⇒ pair(next, gb.right)(f)
    }
}
```

```
new Pairing[ZipMove, Zipper] {
  def pair[A, B, C](
    fa: ZipMove[A],
    gb: Zipper[B]
  )(f: (A, B) ⇒ C) =
    fa match {
      case ZipStop(a)     ⇒ f(a, gb.focus)
      case ZipLeft(next)  ⇒ pair(next, gb.left)(f)
      case ZipRight(next) ⇒ pair(next, gb.right)(f)
    }
}
```

```
new Pairing[ZipMove, Zipper] {
  def pair[A, B, C](
    fa: ZipMove[A],
    gb: Zipper[B]
  )(f: (A, B) ⇒ C) =
    fa match {
      case ZipStop(a)     ⇒ f(a, gb.focus)
      case ZipLeft(next)  ⇒ pair(next, gb.left)(f)
      case ZipRight(next) ⇒ pair(next, gb.right)(f)
    }
}
```

```
type ⋈[F[_], G[_]] = Pairing[F, G]

Store[S, ?] ⋈ State[S, ?]

F1 ⋈ G1 ∧ F2 ⋈ G2
          ⇓
Compose[F1, F2, ?] ⋈ Compose[G1, G2, ?]

F ⋈ G
  ⇓
Free[F, ?] ⋈ Cofree[G, ?]

M ⋈ W
  ⇓
StateT[M, ?] ⋈ StoreT[W, ?]
```

```
type ⋈[F[_], G[_]] = Pairing[F, G]

State[S, ?] ⋈ Store[S, ?]

F1 ⋈ G1 ∧ F2 ⋈ G2
          ⇓
Compose[F1, F2, ?] ⋈ Compose[G1, G2, ?]

F ⋈ G
  ⇓
Free[F, ?] ⋈ Cofree[G, ?]

M ⋈ W
  ⇓
StateT[M, ?] ⋈ StoreT[W, ?]
```

Declarative UIs

Declarative UIs are comonads

```
type ReactComponent[Model] = (Model, Model ⇒ VDOM[Model])
```

```
case class Store[S, A](pos: S, run: S ⇒ A)

type ReactComponent[Model] = Store[Model, VDOM[Model]]
```

```
case class Store[S, A](
  pos: S,
  run: S ⇒ A)
```

```
case class State[S, A](
  run: S ⇒ (S, A))
```

```
new Pairing[State[S, ?], Store[S, ?]] {
  def pair[A, B, C](fa: State[S, A], gb: Store[S, B])(f: (A, B) ⇒ C) = {
    val (s, a) = fa.run(gb.pos)
    val b = gb.run(s)
    f(a, b)
  }
}
```

```
new Pairing[State[S, ?], Store[S, ?]] {
  def pair[A, B, C](fa: State[S, A], gb: Store[S, B])(f: (A, B) ⇒ C) = {
    val (s, a) = fa.run(gb.pos)
    val b = gb.run(s)
    f(a, b)
  }
}
```

```
new Pairing[State[S, ?], Store[S, ?]] {
  def pair[A, B, C](fa: State[S, A], gb: Store[S, B])(f: (A, B) ⇒ C) = {
    val (s, a) = fa.run(gb.pos)
    val b = gb.run(s)
    f(a, b)
  }
}
```

```
def move[Model](
  action: State[Model, Unit],
  wmap0: ReactComponent[Model]
)(implicit
  P: Pairing[State[Model, ?], Store[Model, ?]]
): ReactComponent[Model] =
  P.pair(action, wmap0.duplicate)((_, wmap) ⇒ wmap)
```

```
val counter: ReactComponent[Int] = Store(
  0,
  count ⇒ (
    <button onClick={_ ⇒ count + 1}>
      {count}
    </button>
  )
)

counter.extract   : VDOM[Int]

counter.duplicate : Store[Int, ReactComponent[Int]]
```

```
type ReactComponent[Model] = Store[Model, VDOM[Model]]
```

```
type ReactComponent[Model] = Store[Model, VDOM[State[Model, Unit]]]
```

```
val counter: ReactComponent[Int] = Store(
  0,
  count ⇒ (
    <button onClick={_ ⇒ State[Int].modify(_ + 1)}>
      {count}
    </button>
  )
)
```

React

$$\texttt{Store[S, ?]} \bowtie \texttt{State[S, ?]}$$

Elm

$$\texttt{Free[(X, ?), ?]} \bowtie \texttt{Cofree[(X} \Rightarrow \texttt{?), ?]}$$

Halogen

$$\texttt{F} \bowtie \texttt{G}$$

$$\Downarrow$$

$$\texttt{Free[F, ?]} \bowtie \texttt{Cofree[G, ?]}$$

```
type Component
```

```
type Component[A] = A
```

```
type Component[W[_], A] = W[A]
```

```
type Component[W[_], M[_], A] = W[M[Unit] ⇒ A]
```

```
type Component[Effect[_], W[_], M[_], A] =
  W[(M[Unit] ⇒ Effect[Unit]) ⇒ A]
```

```
type UI[Effect[_], M[_], A] =
  (M[Unit] ⇒ Effect[Unit]) ⇒ A

type Component[Effect[_], W[_], M[_], A] =
  W[UI[Effect, M, A]]
```

Demo time

Section 1 | Section 2 | Section 3 | Section 4

Section-1

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras nec urna aliquam, ornare eros vel, malesuada lorem. Nullam faucibus lorem at eros consectetur lobortis. Maecenas nec nibh congue, placerat sem id, rutrum velit. Phasellus porta enim at facilisis condimentum. Maecenas pharetra dolor vel elit tempor pellentesque sed sed eros. Aenean vitae mauris tincidunt, imperdiet orci semper, rhoncus ligula. Vivamus scelerisque.

```
case class Tab(
  label     : String,
  content   : String,
  keystroke : String ⇒ IO[IOException, Unit]
)

type TabbedUI =
  Component[IO[IOException, ?], Zipper, ZipMove, Tab]
```

```
def render(
  label: String,
  content: String
): UI[IO[IOException, ?], ZipMove, Tab] =
  (send: ZipMove[Unit] ⇒ IO[IOException, Unit]) ⇒
    Tab(
      label,
      content,
      keystroke = {
        case "p" ⇒ send(ZipLeft(ZipStop(())))
        case "n" ⇒ send(ZipRight(ZipStop(())))
        case _   ⇒ IO.unit
      }
    )
```

```
def render(
  label: String,
  content: String
): UI[IO[IOException, ?], ZipMove, Tab] =
  (send: ZipMove[Unit] ⇒ IO[IOException, Unit]) ⇒
    Tab(
      label,
      content,
      keystroke = {
        case "p" ⇒ send(ZipLeft(ZipStop(())))
        case "n" ⇒ send(ZipRight(ZipStop(())))
        case _   ⇒ IO.unit
      }
    )
```

```
def render(
  label: String,
  content: String
): UI[IO[IOException, ?], ZipMove, Tab] =
  (send: ZipMove[Unit] ⇒ IO[IOException, Unit]) ⇒
    Tab(
      label,
      content,
      keystroke = {
        case "p" ⇒ send(ZipLeft(ZipStop(())))
        case "n" ⇒ send(ZipRight(ZipStop(())))
        case _   ⇒ IO.unit
      }
    )
```

```
def tabs(
  fst: (String, String)
  rst: *(String, String)
): TabbedUI =
  Zipper(
    Stream.empty,
    render(fst._1, fst._2),
    Stream(rst:_*).map { case (l, c) ⇒ render(l, c) }
  )
```

```
def tabs(
  fst: (String, String)
  rst: *(String, String)
): TabbedUI =
  Zipper(
    Stream.empty,
    render(fst._1, fst._2),
    Stream(rst:_*).map { case (l, c) ⇒ render(l, c) }
  )
```

```
def tabs(
  fst: (String, String)
  rst: *(String, String)
): TabbedUI =
  Zipper(
    Stream.empty,
    render(fst._1, fst._2),
    Stream(rst:_*).map { case (l, c) ⇒ render(l, c) }
  )
```

```
def tabs(
  fst: (String, String)
  rst: *(String, String)
): TabbedUI =
  Zipper(
    Stream.empty,
    render(fst._1, fst._2),
    Stream(rst:_*).map { case (l, c) ⇒ render(l, c) }
  )
```

```
def run(component: TabbedUI): IO[IOException, Unit] =
  for {
    state ← Ref(component)
    _     ← (state.get.flatMap { current ⇒ ??? }).forever
  } yield ()
```

```
def run(component: TabbedUI): IO[IOException, Unit] =
  for {
    state ← Ref(component)
    _     ← (state.get.flatMap { current ⇒ ??? }).forever
  } yield ()
```

```
def run(component: TabbedUI): IO[IOException, Unit] =
  for {
    state ← Ref(component)
    _     ← (state.get.flatMap { current ⇒ ??? }).forever
  } yield ()
```

```
def run(component: TabbedUI): IO[IOException, Unit] =
  for {
    state ← Ref(component)
    _     ← (state.get.flatMap { current ⇒
      val ui: UI[IO[IOException, ?], ZipMove, Tab] =
        current.extract
    }).forever
  } yield ()
```

```
def send(
  state: Ref[TabbedUI],
  component: TabbedUI
): ZipMove[Unit] ⇒ IO[IOException, Unit] =
  action ⇒ state.set(move(action, component))
```

```
def send(
  state: Ref[TabbedUI],
  component: TabbedUI
): ZipMove[Unit] ⇒ IO[IOException, Unit] =
  action ⇒ state.set(move(action, component))
```

```
def send(
  state: Ref[TabbedUI],
  component: TabbedUI
): ZipMove[Unit] ⇒ IO[IOException, Unit] =
  action ⇒ state.set(move(action, component))
```

```
def run(component: TabbedUI): IO[IOException, Unit] =
  for {
    state ← Ref(component)
    _ ← (state.get.flatMap { current ⇒
      val ui: UI[IO[IOException, ?], ZipMove, Tab] =
        current.extract
    }).forever
  } yield ()
```

```
def run(component: TabbedUI): IO[IOException, Unit] =
  for {
    state ← Ref(component)
    _ ← (state.get.flatMap { current ⇒
      val ui: UI[IO[IOException, ?], ZipMove, Tab] =
        current.extract
      val tab: Tab = ui(send(state, current))
    }).forever
  } yield ()
```

```
def run(component: TabbedUI): IO[IOException, Unit] =
  for {
    state ← Ref(component)
    _ ← (state.get.flatMap { current ⇒
      val ui: UI[IO[IOException, ?], ZipMove, Tab] =
        current.extract
      val tab: Tab = ui(send(state, current))
      display(current, tab) andThen (getStrLn.flatMap {
        str ⇒ tab.keystroke(str)
      })
    }).forever
  } yield ()
```

```
def run(component: TabbedUI): IO[IOException, Unit] =
  for {
    state ← Ref(component)
    _ ← (state.get.flatMap { current ⇒
      val ui: UI[IO[IOException, ?], ZipMove, Tab] =
        current.extract
      val tab: Tab = ui(send(state, current))
      display(current, tab) andThen (getStrLn.flatMap {
        str ⇒ tab.keystroke(str)
      })
    }).forever
  } yield ()
```

```
def run(component: TabbedUI): IO[IOException, Unit] =
  for {
    state ← Ref(component)
    _ ← (state.get.flatMap { current ⇒
      val ui: UI[IO[IOException, ?], ZipMove, Tab] =
        current.extract
      val tab: Tab = ui(send(state, current))
      display(current, tab) andThen (getStrLn.flatMap {
        str ⇒ tab.keystroke(str)
      })
    }).forever
  } yield ()
```

Reading

- *The Cofree Comonad and the Expression Problem*, Ed Kmett
  http://comonad.com/reader/2008/the-cofree-comonad-and-the-expression-problem/

- *Cofun with cofree interpreters*, Dave Laing
  http://dlaing.org/cofun/

- *Comonads as spaces* (and all the rest), Phil Freeman
  https://blog.functorial.com/posts/2016-08-07-Comonads-As-Spaces.html

- *Comonads for user interfaces*, Arthur Xavier
  https://arthurxavierx.github.io/ComonadsForUIs.pdf

- *A Real-World Application with a Comonadic User Interface*, Arthur Xavier
  https://arthurxavierx.github.io/RealWorldAppComonadicUI.pdf

Voilà

Check out

https://github.com/regiskuckaertz/scala-exchange-2018

This presentation will be soon available on the Scala eXchange London website at the following link

https://skillsmatter.com/conferences/10488-scala-exchange-2018#skillscasts

Free & Cofree

```
sealed abstract class Free[F[_], A]
case class Pure[F[_], A](value: A) extends Free[F, A]
case class Bind[F[_], A](value: F[Free[F, A]]) extends Free[F, A]

case class Cofree[F[_], A](head: A, tail: F[Cofree[F, A]])
```

Free & Cofree

```
implicit def cofree[F[_]: Functor, G[_]](implicit P: Pairing[F, G]) =
  new Pairing[Free[F, ?], Cofree[G, ?]] {
    def pair[A, B, C](fa: Free[F, A], gb: Cofree[G, B])(f: (A, B) ⇒ C) =
      ???
  }
```

Free & Cofree

```
implicit def cofree[F[_]: Functor, G[_]](implicit P: Pairing[F, G]) =
  new Pairing[Free[F, ?], Cofree[G, ?]] {
    def pair[A, B, C](fa: Free[F, A], gb: Cofree[G, B])(f: (A, B) ⇒ C) =
      fa.resume match {
        ???
      }
  }
```

Free & Cofree

```
implicit def cofree[F[_]: Functor, G[_]](implicit P: Pairing[F, G]) =
  new Pairing[Free[F, ?], Cofree[G, ?]] {
    def pair[A, B, C](fa: Free[F, A], gb: Cofree[G, B])(f: (A, B) ⇒ C) =
      fa.resume match {
        case \/-(a) ⇒ f(a, gb.head)
        ???
      }
  }
```

Free & Cofree

```
implicit def cofree[F[_]: Functor, G[_]](implicit P: Pairing[F, G]) =
  new Pairing[Free[F, ?], Cofree[G, ?]] {
    def pair[A, B, C](fa: Free[F, A], gb: Cofree[G, B])(f: (A, B) ⇒ C) =
      fa.resume match {
        case \/-(a) ⇒ f(a, gb.head)
        case -\/(ffa) ⇒ P.pair(ffa, gb.tail)(pair(_, _)(f))
      }
  }
```

Sample program

```
val program: Free[UserApi, List[User]] = for {
  _ ← UserApi.add(User(“Kate”))
  _ ← UserApi.add(User(“Adam”))
  _ ← UserApi.add(User(“Pascal”))
  users ← UserApi.getAll
} yield users
```

Sample interpreter

```
type State = (Map[Int, User], Int)
```

Sample interpreter

```
type State = (Map[Int, User], Int)

def makeCoUserApi(state: State): CoUserApi[State] = ???
```

Sample interpreter

```
type State = (Map[Int, User], Int)

def makeCoUserApi(state: State): CoUserApi[State] = CoUserApi(
  getAll = (state._1.values.toList, state),
  get = id ⇒ (state._1.get(id), state),
  add = user ⇒ {
    val uid = state._2 + 1
    val users = state._1 + (uid → user)
    (uid, (users, uid))
  }
)
```

Sample interpreter

```
object Cofree {
  def unfoldC[F[_]: Functor, A](a: A)(f: A ⇒ F[A]): Cofree[F, A]
}
```

Sample interpreter

```
type State = (Map[Int, User], Int)

def makeCoUserApi(state: State): CoUserApi[State] = CoUserApi(
  getAll = (state._1.values.toList, state),
  get = id ⇒ (state._1.get(id), state),
  add = u ⇒ {
    val uid = state._2 + 1
    val users = state._1 + (uid → u)
    (uid, (users, uid))
  }
)

val interpreter: Cofree[CoUserApi, State] =
  Cofree.unfoldC((Map.empty, 0))(makeCoUserApi)
```

Run

```
def evalPure[F[_], G[_], A, B](
  program: Free[F, A],
  interpreter: Cofree[G, B]
)(implicit
  P: Pairing[Free[F, ?], Cofree[G, ?]]
): A =
  P.pair(program, interpreter)((a, _) ⇒ a)

evalPure(program, interpreter)
// List(User(Kate), User(Adam), User(Pascal))
```

Combine DSLs

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

type Program[A] = Coproduct[UserApi, ProductApi, A]
```

Combine DSLs

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

type Program[A] = Coproduct[UserApi, ProductApi, A]

case class Produkt[F[_], G[_], A](fst: F[A], snd: G[A])

type Interpreter[A] = Produkt[CoUserApi, CoProductApi, A]
```

Combine DSLs

```
implicit def product[F1[_], F2[_], G1[_], G2[_]](implicit
  P1: Pairing[F1, F2],
  P2: Pairing[G1, G2]
) = new Pairing[Coproduct[F1, G1, ?], Produkt[F2, G2, ?]] {
  def pair[A, B, C](
    fa: Coproduct[F1, G1, A],
    gb: Produkt[F2, G2, B]
  )(f: (A, B) ⇒ C) =
    fa.run.fold(P1.pair(_, gb.fst)(f), P2.pair(_, gb.snd)(f))
}
```

Combine DSLs

```
type UserState = (Map.empty, Int)
type ProductState = (Map.empty, Int)

def makeCoUserApi(state: UserState): CoUserApi[UserState] = ???
def makeCoProductApi(state: ProductState): CoProductApi[ProductState] = ???
```

Combine DSLs

```
type UserState = (Map.empty, Int)
type ProductState = (Map.empty, Int)
type State = (UserState, ProductState)

def makeCoUserApi(state: UserState): CoUserApi[UserState] = ???
def makeCoProductApi(state: ProductState): CoProductApi[ProductState] = ???

def makeInterpreter(state: State): Interpreter[State] =
  Produkt(makeCoUserApi(state._1), makeCoProductApi(state._2))
```

Combine DSLs

```
type UserState = (Map.empty, Int)
type ProductState = (Map.empty, Int)
type State = (UserState, ProductState)

def makeCoUserApi(state: UserState): CoUserApi[UserState] = ???
def makeCoProductApi(state: ProductState): CoProductApi[ProductState] = ???

def makeInterpreter(state: State): Interpreter[State] =
  Produkt(makeCoUserApi(state._1), makeCoProductApi(state._2))

val interpreter: Cofree[Interpreter, State] =
  Cofree.unfoldC(((Map.empty, 0), (Map.empty, 0)))(makeInterpreter)
```

I can haz effects?

```
sealed trait UserApi[A]
case class GetAllUsers[A](next: List[User] ⇒ A) extends UserApi[A]
case class GetUser[A](id: Int, next: Option[User] ⇒ A) extends UserApi[A]
case class AddUser[A](user: User, next: Int ⇒ A) extends UserApi[A]

final case class CoUserApi[A](
  getAll: (List[User], A),
  get: Int ⇒ (Option[User], A),
  add: User ⇒ (Int, A)
)
```

I can haz effects?

```
sealed trait UserApiM[M[_], A]
case class GetAllUsersM[M[_], A](next: List[User] ⇒ M[A])
  extends UserApiM[M, A]
case class GetUserM[M[_], A](id: Int, next: Option[User] ⇒ M[A])
  extends UserApiM[M, A]
case class AddUserM[M[_], A](user: User, next: Int ⇒ M[A])
  extends UserApiM[M, A]

final case class CoUserApiM[M[_], A](
  getAll: M[(List[User], A)],
  get: Int ⇒ M[(Option[User], A)],
  add: User ⇒ M[(Int, A)]
)
```

I can haz effects?

```
trait PairingM[M[_], F[_], G[_]] {
    def pairM[A, B, C](fa: F[A], gb: G[B])(f: (A, B) ⇒ M[C]): M[C]
}
```

I can haz effects?

```
def pair[F[_], G[_], A, B, C](fa: F[A], gb: G[B])(f: (A, B) ⇒ C)(implicit
  P: PairingM[Id, F, G]
): C =
  P.pairM(fa, gb)(f)
```

I can haz effects?

```
implicit def pairingM[M[_]: Monad] =
  new PairingM[M, UserApiM[M, ?], CoUserApiM[M, ?]] {
    def pairM[A, B, C](
      fa: UserApiM[M, A],
      gb: CoUserApiM[M, B]
    )(f: (A, B) ⇒ M[C]): M[C] = fa match {
      case GetAllUsersM(next) ⇒
        gb.getAll.flatMap { case (us, b) ⇒ next(us).flatMap(f(_, b)) }
      case GetUserM(id, next) ⇒
        gb.get(id).flatMap { case (ou, b) ⇒ next(ou).flatMap(f(_, b)) }
      case AddUserM(user, next) ⇒
        gb.add(user).flatMap { case (uid, b) ⇒ next(uid).flatMap(f(_, b)) }
    }
}
```