Monads Are *Not* About Sequencing

Lawful Monads in a Concurrent Setting

functional scala

Tomas Mikula

Dec 1st, 2023

“Monads are about sequencing.”

–folk knowledge

What Does *Sequencing* Mean?

What Does *Sequencing* Mean?

¿ Sequence-like syntax ?

What Does *Sequencing* Mean?

¿ Sequence-like syntax ?

¿ Enforced sequential execution ?

What Does *Sequencing* Mean?

¿ Sequence-like syntax ?

¿ Enforced sequential execution ?

¿ Enforced order of “effects” ?

What Does *Sequencing* Mean?

Spoiler

¿ Sequence-like syntax ?

¿ Enforced sequential execution ?

¿ Enforced order of “effects” ?

What Does *Sequencing* Mean?

Spoiler

¿ Sequence-like syntax ? ⬅ the only defensible claim

¿ Enforced sequential execution ?

¿ Enforced order of “effects” ?

Monads: Refresher

Monads: Refresher

```
trait Monad[M[_]]:
```

Monads: Refresher

```
trait Monad[M[_]]:

    def map[A, B] (ma: M[A]) (f: A => B) : M[B]
```

Monads: Refresher

```
trait Monad[M[_]]:

    def map[A, B] (ma: M[A]) (f: A => B) : M[B]

    def pure[A] (a: A) : M[A]
```

Monads: Refresher

```
trait Monad[M[_]]:

    def map[A, B] (ma: M[A]) (f: A => B) : M[B]

    def pure[A] (a: A) : M[A]

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

Monads: Refresher

```
trait Monad[M[_]]:

    def map[A, B] (ma: M[A]) (f: A => B) : M[B]

    def pure[A] (a: A) : M[A]

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


    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B] =
```

Monads: Refresher

```
trait Monad[M[_]]:
    def map[A, B] (ma: M[A]) (f: A => B) : M[B]
    def pure[A] (a: A) : M[A]
    def flatten[A] (mma: M[M[A]]) : M[A]

    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B] =
        flatten(map(ma)(f))
```

Monads: Refresher

```scala
trait Monad[M[_]]:
    def map[A, B] (ma: M[A]) (f: A => B) : M[B]
    def pure[A] (a: A) : M[A]
    def flatten[A] (mma: M[M[A]]) : M[A]

    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B] =
        flatten(map(ma)(f))
```

… plus the monad laws …

Monads: Refresher

```
trait Monad[M[_]]:

    def map[A, B] (ma: M[A]) (f: A => B) : M[B]

    def pure[A] (a: A) : M[A]

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

    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B]
```

Monads: Refresher

```scala
trait Monad[M[_]]:
    def map[A, B] (ma: M[A]) (f: A => B) : M[B]
    def pure[A] (a: A) : M[A]
    def flatten[A] (mma: M[M[A]]) : M[A]
    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B]

/** Extensions to let us write ma.map(f), ma.flatMap(f). */
```

Monads: Refresher

```scala
trait Monad[M[_]]:

    def map[A, B] (ma: M[A]) (f: A => B) : M[B]

    def pure[A] (a: A) : M[A]

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

    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B]

/** Extensions to let us write ma.map(f), ma.flatMap(f). */
extension [M[_], A] (ma: M[A]) (using M: Monad[M])
```

Monads: Refresher

```
trait Monad[M[_]]:

    def map[A, B] (ma: M[A]) (f: A => B) : M[B]

    def pure[A] (a: A) : M[A]

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

    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B]

/** Extensions to let us write ma.map(f), ma.flatMap(f). */
extension [M[_], A] (ma: M[A]) (using M: Monad[M])

    def map[B]     (f: A =>   B ) : M[B]  =  M.map(ma)(f)

    def flatMap[B] (f: A => M[B]) : M[B]  =  M.flatMap(ma)(f)
```

Monads: Refresher

```
trait Monad[M[_]]:

    def map[A, B] (ma: M[A]) (f: A => B) : M[B]

    def pure[A] (a: A) : M[A]

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

    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B]
```

Monads: Refresher

```
trait Monad[M[_]]:

    def map[A, B] (ma: M[A]) (f: A => B) : M[B]

    def pure[A] (a: A) : M[A]

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

    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B]

/** Kleisli composition: f >=> g */
```

Monads: Refresher

```
trait Monad[M[_]]:

    def map[A, B] (ma: M[A]) (f: A => B) : M[B]

    def pure[A] (a: A) : M[A]

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

    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B]

/** Kleisli composition: f >=> g */
extension [M[_], A, B] (f: A => M[B]) (using Monad[M])
```

Monads: Refresher

```scala
trait Monad[M[_]]:
    def map[A, B] (ma: M[A]) (f: A => B) : M[B]
    def pure[A] (a: A) : M[A]
    def flatten[A] (mma: M[M[A]]) : M[A]
    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B]

/** Kleisli composition: f >=> g */
extension [M[_], A, B] (f: A => M[B]) (using Monad[M])
    def >=>[C] (g: B => M[C]) : A => M[C] =
```

Monads: Refresher

```
trait Monad[M[_]]:

    def map[A, B] (ma: M[A]) (f: A => B) : M[B]

    def pure[A] (a: A) : M[A]

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

    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B]

/** Kleisli composition: f >=> g */
extension [M[_], A, B] (f: A => M[B]) (using Monad[M])

    def >=>[C] (g: B => M[C]) : A => M[C] =

        f(_).flatMap(g)
```

Monads in Action

Monads in Action

```
given M: Monad[M]

val ma: M[A]

val f: A => M[B]

val g: B => M[C]

val h: C => M[D]
```

Monads in Action

```scala
given M: Monad[M]

val ma: M[A]

val f: A => M[B]

val g: B => M[C]

val h: C => M[D]
```

```scala
for
  a <- ma
  b <- f(a)
  c <- g(b)
  d <- h(c)
yield d
```

Monads in Action

```scala
given M: Monad[M]

val ma: M[A]

val f: A => M[B]

val g: B => M[C]

val h: C => M[D]
```

```scala
for
  a <- ma
  b <- f(a)
  c <- g(b)
  d <- h(c)
yield d
```

```scala
f >=> g >=> h  :  A => M[D]
```

Monads in Action

```
given M: Monad[M]

val ma: M[A]

val f: A => M[B]

val g: B => M[C]

val h: C => M[D]
```

```
for
  a <- ma
  b <- f(a)
  c <- g(b)
  d <- h(c)
yield d
```

Sequence!

```
f >=> g >=> h  :  A => M[D]
```

Monads in Action

```
given M: Monad[M]

val ma: M[A]

val f: A => M[B]

val g: B => M[C]

val h: C => M[D]
```

```
for
  a <- ma
  b <- f(a)
  c <- g(b)
  d <- h(c)
yield d
```

```
f >=> g >=> h  :  A => M[D]
```

Sequence!

Monads in Action

```scala
given M: Monad[M]

val ma: M[A]

val f: A => M[B]

val g: B => M[C]

val h: C => M[D]
```

```scala
for
  a <- ma
  b <- f(a)
  c <- g(b)
  d <- h(c)
yield d
```

```scala
f >=> g >=> h  :  A => M[D]
```

Sequence!

```scala
M[M[M[M[D]]]]
```

Of course **Monads** are about **Sequencing**!

Of course **Monads** are about **Sequencing**!

Right?

Monads: Generally

```
trait Monad[M[_]]:

    def map[A, B] (ma: M[A]) (f: A => B) : M[B]

    def pure[A] (a: A) : M[A]

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

    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B]
```

Monads: Generally

```
trait Monad[M[_]]:
    def map[A, B] (ma: M[A]) (f: A => B) : M[B]
    def pure[A] (a: A) : M[A]
    def flatten[A] (mma: M[M[A]]) : M[A]
    def flatMap[A, B] (ma: M[A]) (f: A => M[B]) : M[B]
```

Monads: Generally

```
trait Monad[M[_]]:
    def map[A, B] (f: A => B) (ma: M[A]) : M[B]
    def pure[A] (a: A) : M[A]
    def flatten[A] (mma: M[M[A]]) : M[A]
    def flatMap[A, B] (f: A => M[B]) (ma: M[A]) : M[B]
```

Monads: Generally

```
trait Monad[M[_]]:
    def map[A, B] (f: A => B)         (ma: M[A]) : M[B]
    def pure[A]                          (a: A) : M[A]
    def flatten[A]                  (mma: M[M[A]]) : M[A]
    def flatMap[A, B] (f: A => M[B]) (ma: M[A]) : M[B]
```

Monads: Generally

```
trait Monad[M[_]]:
    def map[A, B] (f: A => B)          :   M[A]  => M[B]
    def pure[A]                        :     A   => M[A]
    def flatten[A]                     : M[M[A]] => M[A]
    def flatMap[A, B] (f: A => M[B]) :   M[A]  => M[B]
```

Monads: Generally

```
trait Monad[->[_, _], M[_]]:

    def map[A, B] (f: A -> B)          :   M[A]  -> M[B]
    def pure[A]                        :     A   -> M[A]
    def flatten[A]                     : M[M[A]] -> M[A]
    def flatMap[A, B] (f: A -> M[B]) :   M[A]  -> M[B]
```

Monads: Generally

```
trait Monad[->[_, _], M[_]]:


    def map[A, B] (f: A -> B)           :   M[A]   -> M[B]
    def pure[A]                         :     A    -> M[A]
    def flatten[A]                      : M[M[A]]  -> M[A]
    def flatMap[A, B] (f: A -> M[B])    :   M[A]   -> M[B]
```

Monads: Generally

```
trait Monad[->[_, _], M[_]]:

    given cat : Category[->]

    def map[A, B] (f: A -> B)          :   M[A]   -> M[B]

    def pure[A]                        :     A    -> M[A]

    def flatten[A]                     : M[M[A]] -> M[A]

    def flatMap[A, B] (f: A -> M[B]) :   M[A]   -> M[B]
```

Monads: Generally

```
trait Category[->[_, _]]:
    def andThen[A, B, C](f: A -> B, g: B -> C): A -> C
    def id[A]: A -> A

trait Monad[->[_, _], M[_]]:
    given cat : Category[->]
    def map[A, B] (f: A -> B)          :   M[A]  -> M[B]
    def pure[A]                        :     A   -> M[A]
    def flatten[A]                     : M[M[A]] -> M[A]
    def flatMap[A, B] (f: A -> M[B])   :   M[A]  -> M[B]
```

Monads: Generally

```
trait Category[●-○[_, _]]:
    def andThen[A, B, C](f: A ●-○ B, g: B ●-○ C): A ●-○ C
    def id[A]: A ●-○ A

trait Monad[●-○[_, _], M[_]]:
    given cat : Category[●-○]
    def map[A, B] (f: A ●-○ B)          :   M[A]  ●-○ M[B]
    def pure[A]                         :     A   ●-○ M[A]
    def flatten[A]                      : M[M[A]] ●-○ M[A]
    def flatMap[A, B] (f: A ●-○ M[B]) :   M[A]  ●-○ M[B]
```

Subtype Relation <:< Forms a Category

```scala
given Category[<:<] with

    def andThen[A, B, C](f: A <:< B, g: B <:< C): A <:< C  =  f andThen g

    def id[A]:                                    A <:< A  =  <:<.refl
```

Monads in <:<

```
given Category[<:<] with
    def andThen[A, B, C](f: A <:< B, g: B <:< C): A <:< C  =  f andThen g
    def id[A]:                                    A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???
    def pure[A]                :     A   <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

Monads in <:<

```
given Category[<:<] with
    def andThen[A, B, C](f: A <:< B, g: B <:< C): A <:< C  =  f andThen g
    def id[A]:                                    A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???
    def pure[A]                :     A   <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

Monads in <:<

```scala
given Category[<:<] with
    def andThen[A, B, C](f: A <:< B, g: B <:< C): A <:< C  =  f andThen g
    def id[A]:                                    A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???
    def pure[A]                :     A   <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

➡ Monotone

Monads in <:<

```scala
given Category[<:<] with
    def andThen[A, B, C](f: A <:< B, g: B <:< C): A <:< C  =  f andThen g
    def id[A]:                                    A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???
    def pure[A]                :     A   <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

➡ Monotone

➡ Extensive

Monads in <:<

```scala
given Category[<:<] with

    def andThen[A, B, C](f: A <:< B, g: B <:< C): A <:< C  =  f andThen g

    def id[A]:                                    A <:< A  =  <:<.refl

given Monad[<:<, M] with

    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???

    def pure[A]                :     A   <:< M[A] = ???

    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in `<:<`

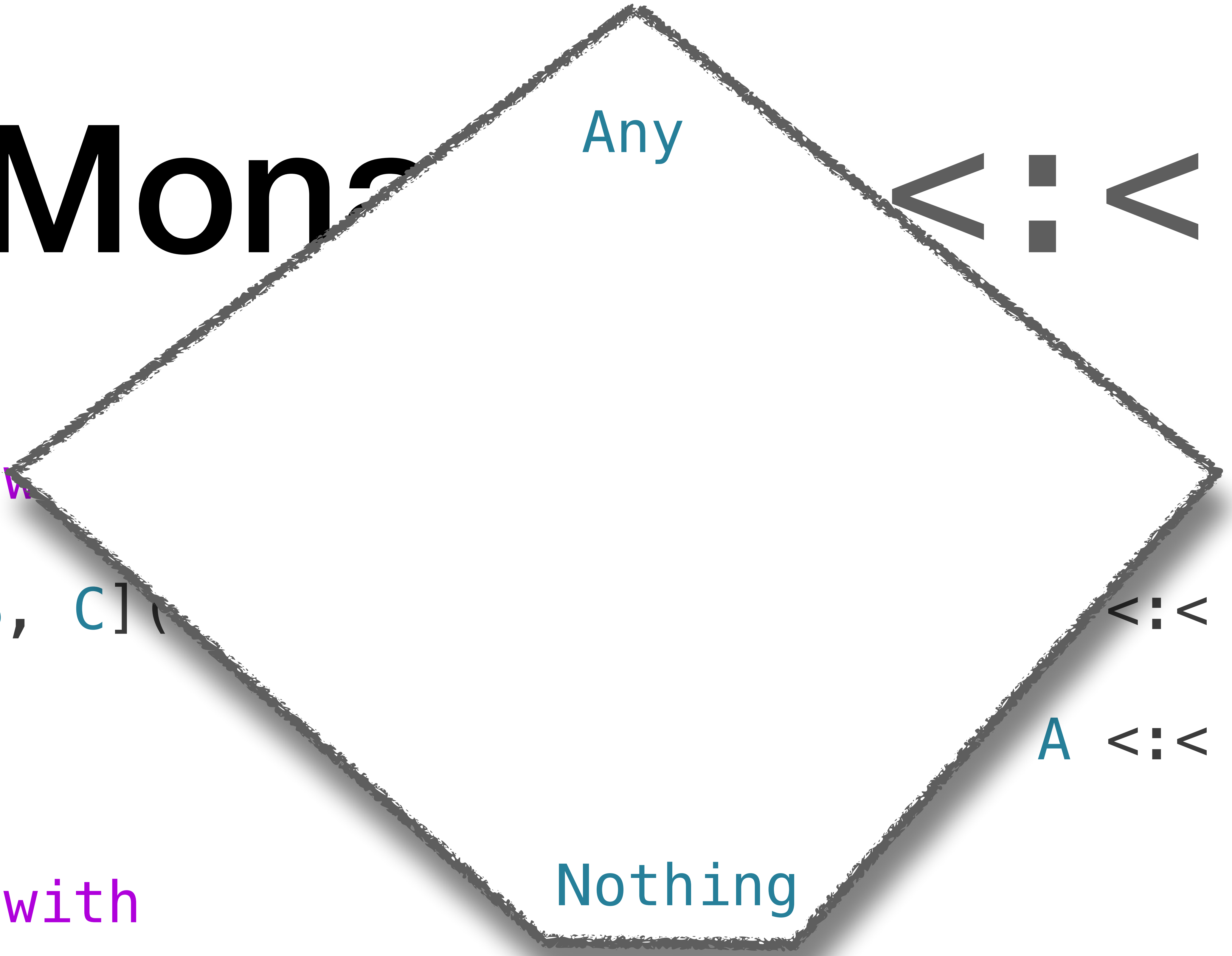
```
given Category[<:<] with
    def andThen[A, B, C](f: A <:< B, g: B <:< C): A <:< C  =  f andThen g
    def id[A]:                                    A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???
    def pure[A]                :     A   <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category `<:<` are **Closure Operators.**

Mon… <:<



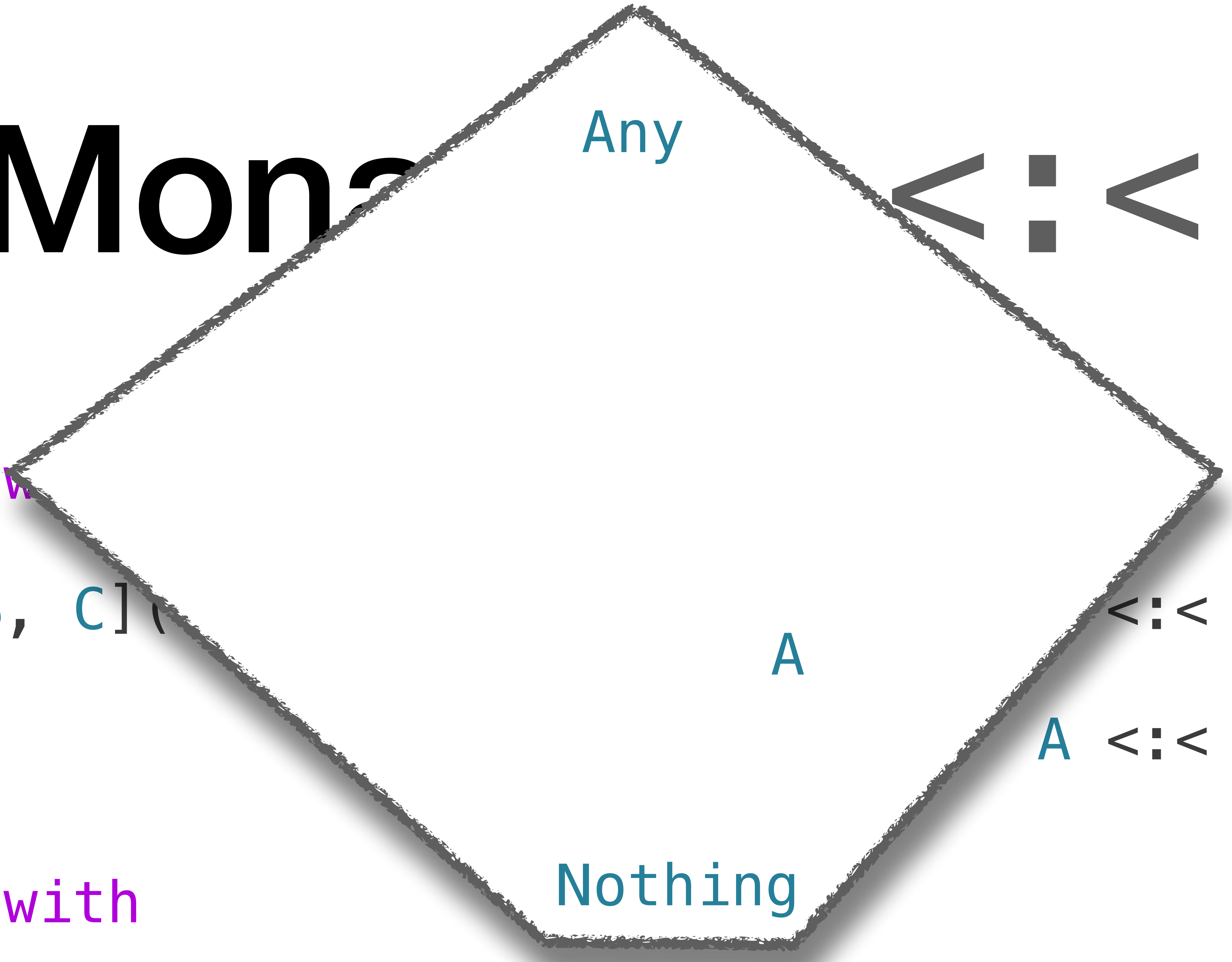

```
given Category[<:<] w
    def andThen[A, B, C](                 <:< C  =  f andThen g
    def id[A]:                          A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???
    def pure[A]                :    A    <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mon <:<




```
given Category[<:<] w

    def andThen[A, B, C](                           <:< C  =  f andThen g

    def id[A]:                                    A <:< A  =  <:<.refl

given Monad[<:<, M] with

    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???

    def pure[A]                :     A   <:< M[A] = ???

    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

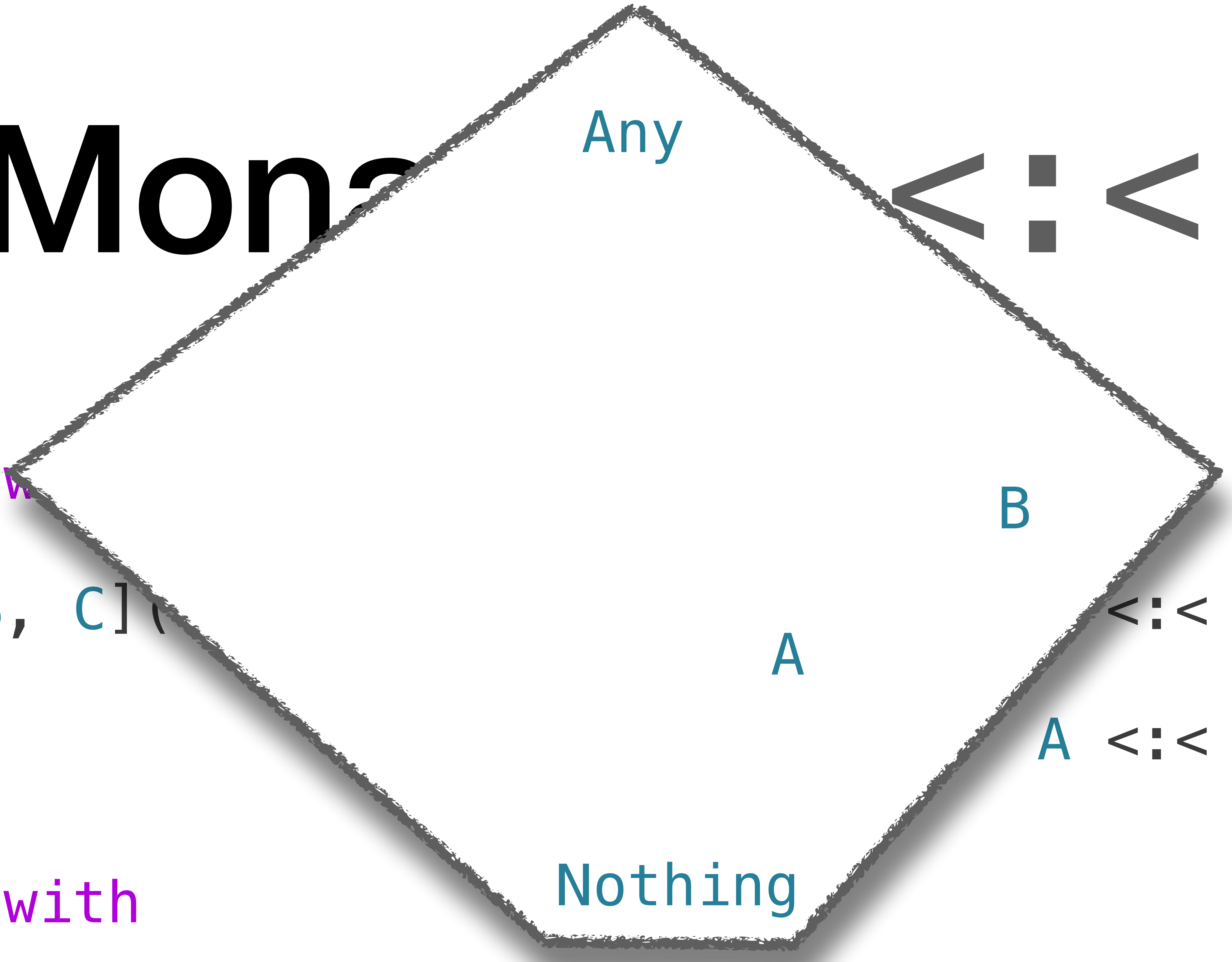
➡ Monotone

➡ Extensive

➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mona <:<




```
given Category[<:<] w

    def andThen[A, B, C](                 <:< C  =  f andThen g

    def id[A]:                          A <:< A  =  <:<.refl

given Monad[<:<, M] with

    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???

    def pure[A]                :     A   <:< M[A] = ???

    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

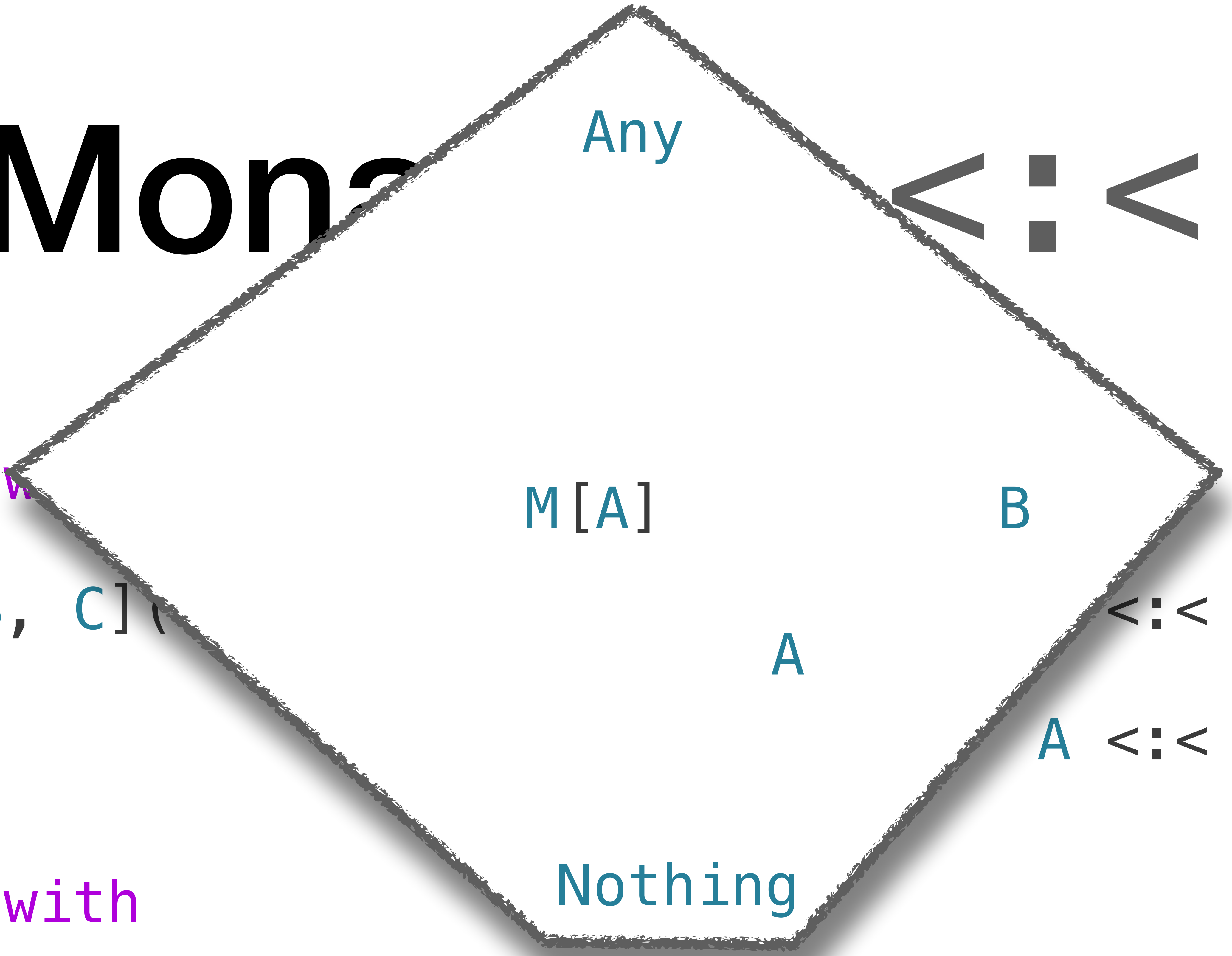
➡ Monotone

➡ Extensive

➡ Idempotent

Monads in the category `<:<` are **Closure Operators.**

Mon <:<



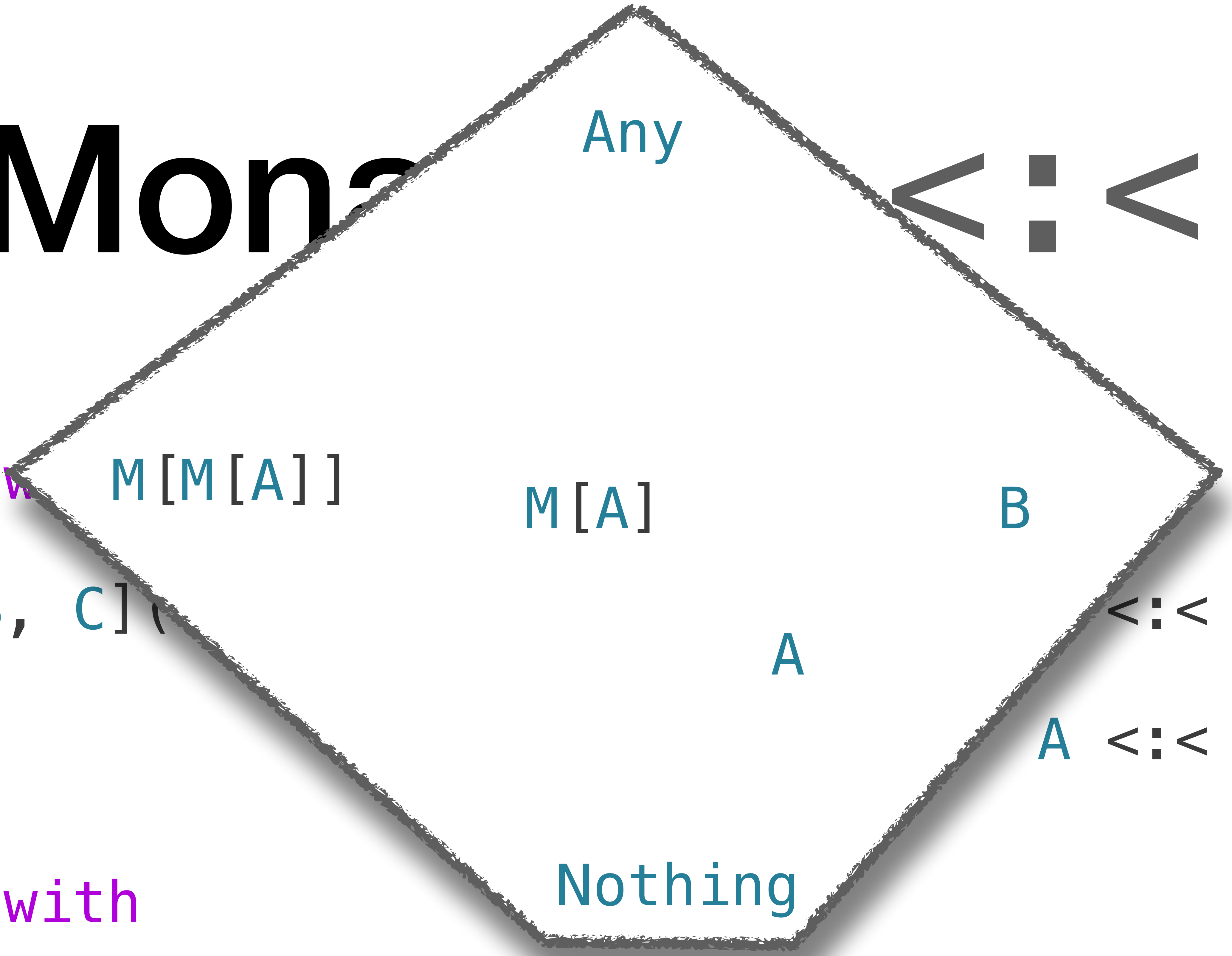

```
given Category[<:<] w
    def andThen[A, B, C](                                  <:< C  =  f andThen g
    def id[A]:                                        A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :    M[A]  <:< M[B] = ???
    def pure[A]                :      A   <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mona <:<



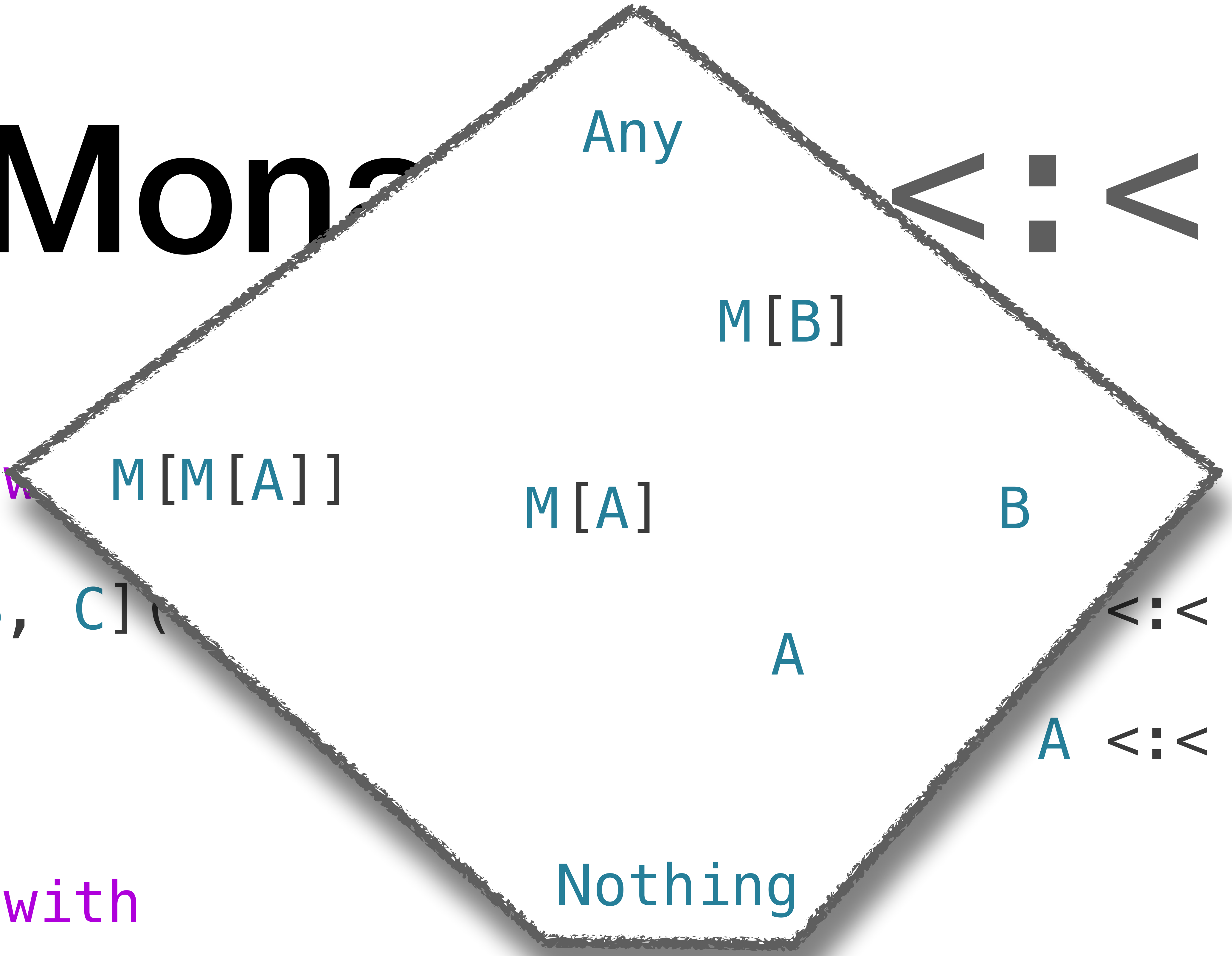

```
given Category[<:<] w
    def andThen[A, B, C](                    <:< C  =  f andThen g
    def id[A]:                          A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]   <:< M[B] = ???
    def pure[A]                :     A    <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mona… <:<



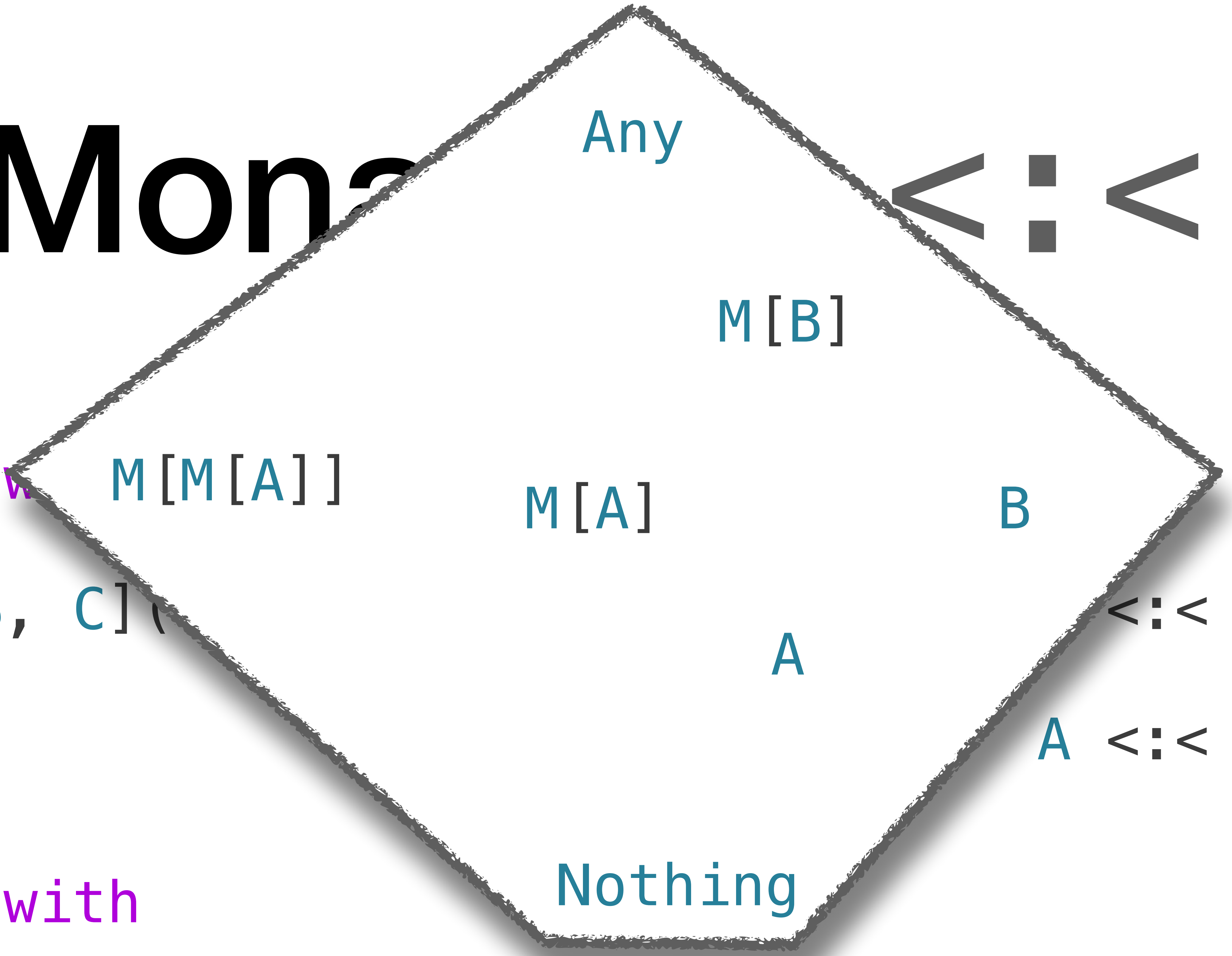

```
given Category[<:<] w
    def andThen[A, B, C](                          <:< C  =  f andThen g
    def id[A]:                                   A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???
    def pure[A]                :     A   <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mon <:<



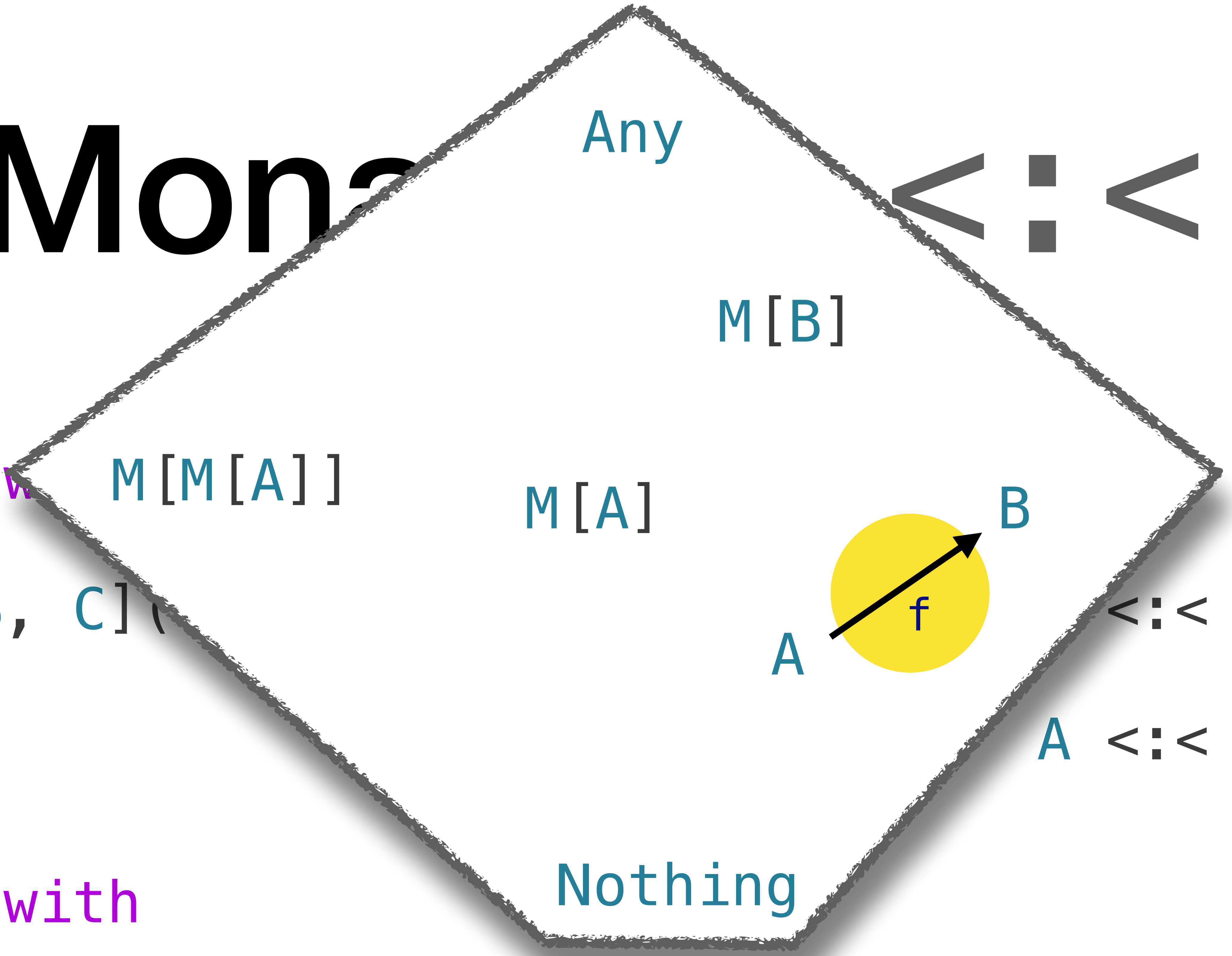

```
given Category[<:<] w

    def andThen[A, B, C](                         <:< C  =  f andThen g

    def id[A]:                                   A <:< A  =  <:<.refl

given Monad[<:<, M] with

    def map[A, B] (f: A <:< B) :    M[A]   <:< M[B] = ???

    def pure[A]                :      A    <:< M[A] = ???

    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mona <:<



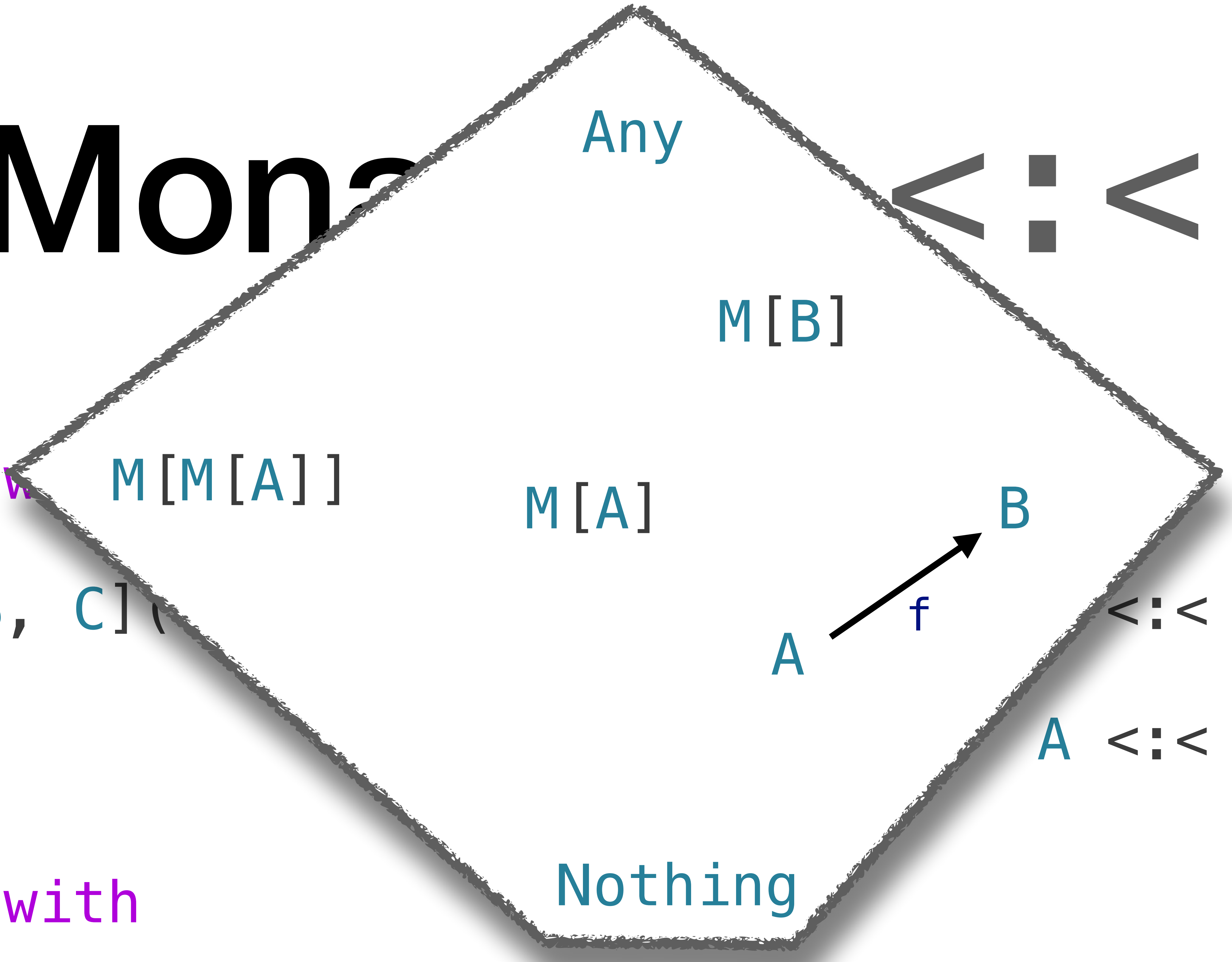

```
given Category[<:<] w
    def andThen[A, B, C](                       <:< C  =  f andThen g
    def id[A]:                                A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :    M[A]  <:< M[B] = ???
    def pure[A]                :      A   <:< M[A] = ???
    def flatten[A]             : M[M[A]]  <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mona <:<



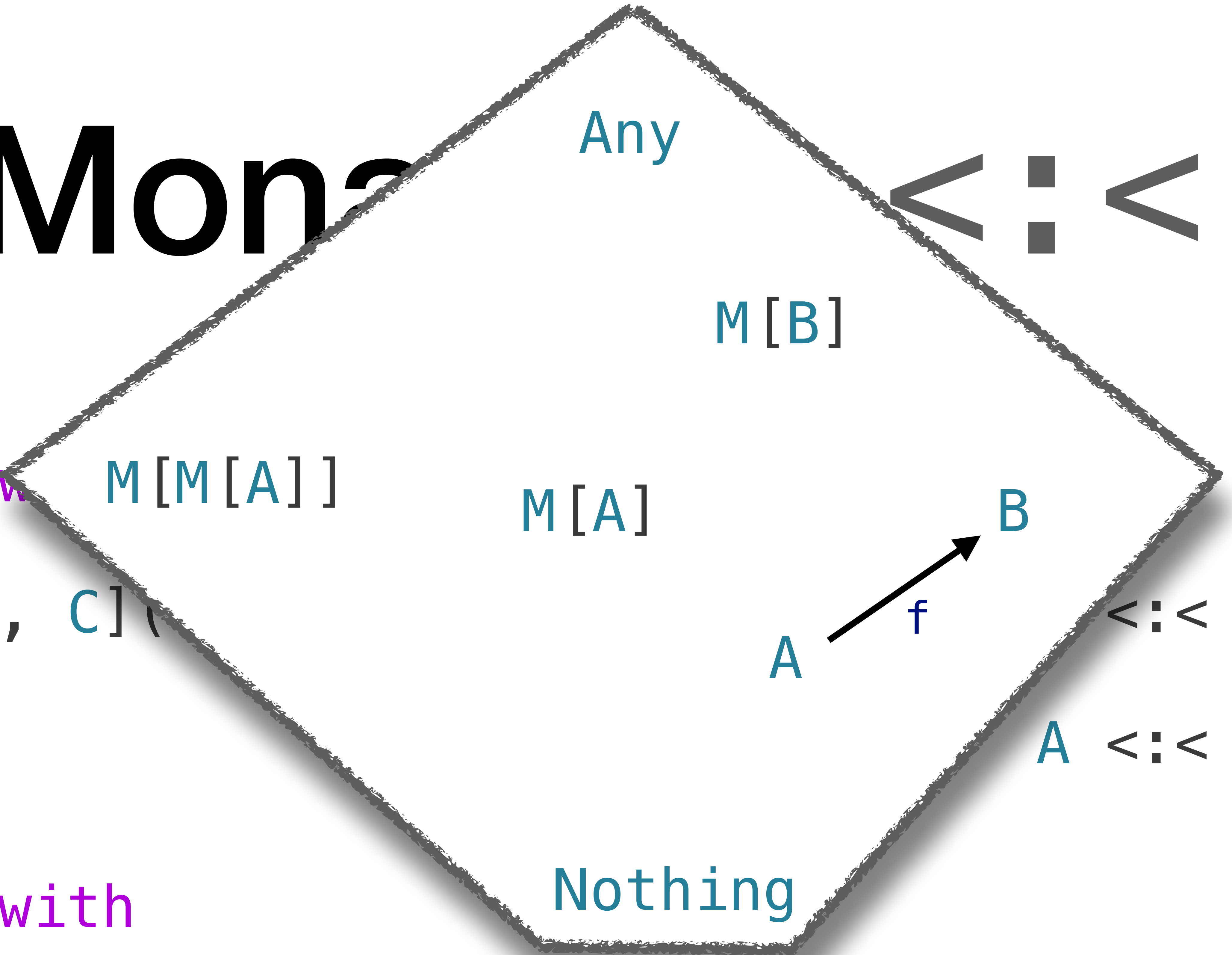

```
given Category[<:<] w
    def andThen[A, B, C](          <:< C  =  f andThen g
    def id[A]:                   A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???
    def pure[A]                :    A    <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mona… <:<




```
given Category[<:<] w…
    def andThen[A, B, C](…                <:< C  =  f andThen g
    def id[A]:                          A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :  M[A]   <:< M[B] = ???
    def pure[A]                :     A   <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

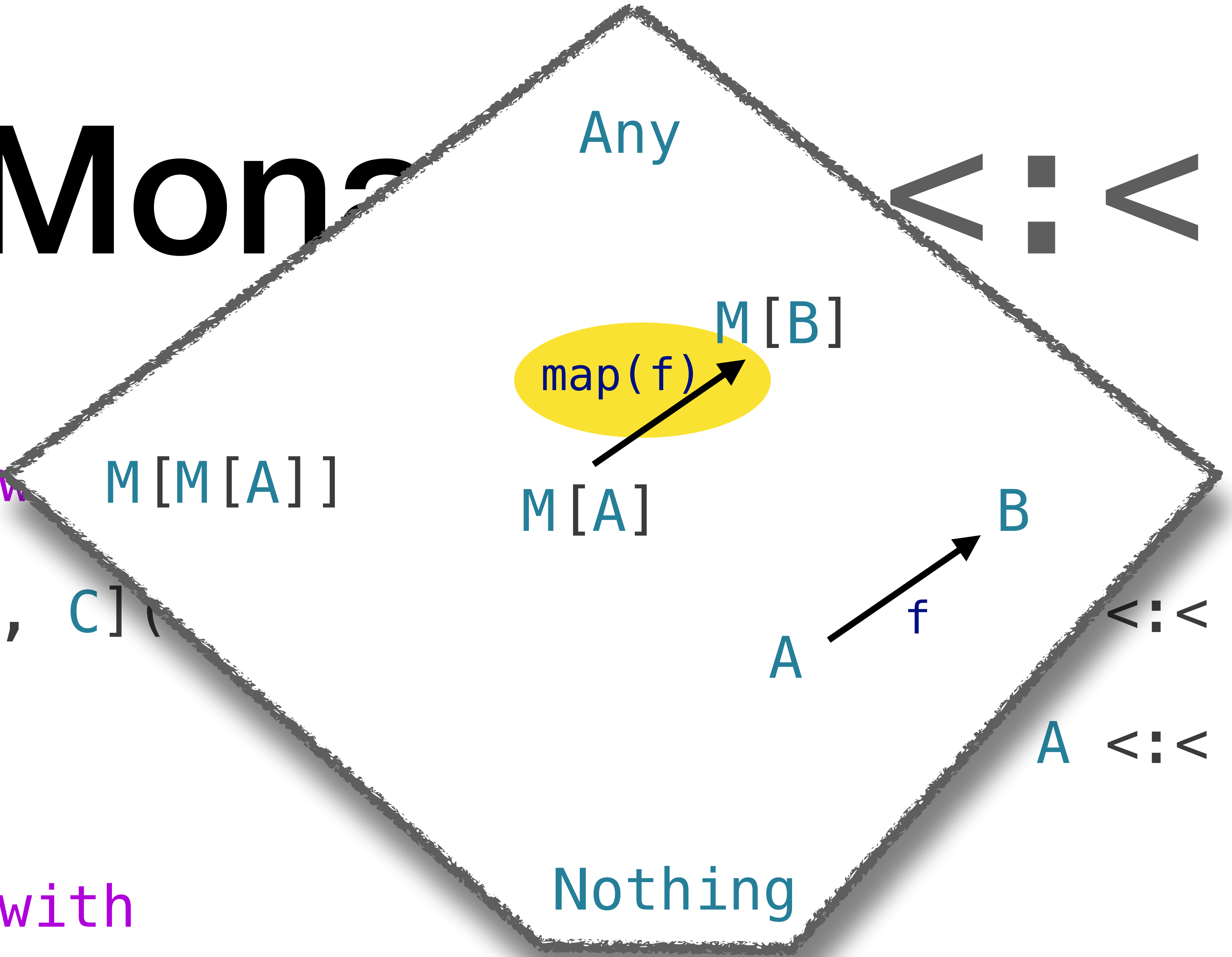
➡ Monotone

➡ Extensive

➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mon… <:<



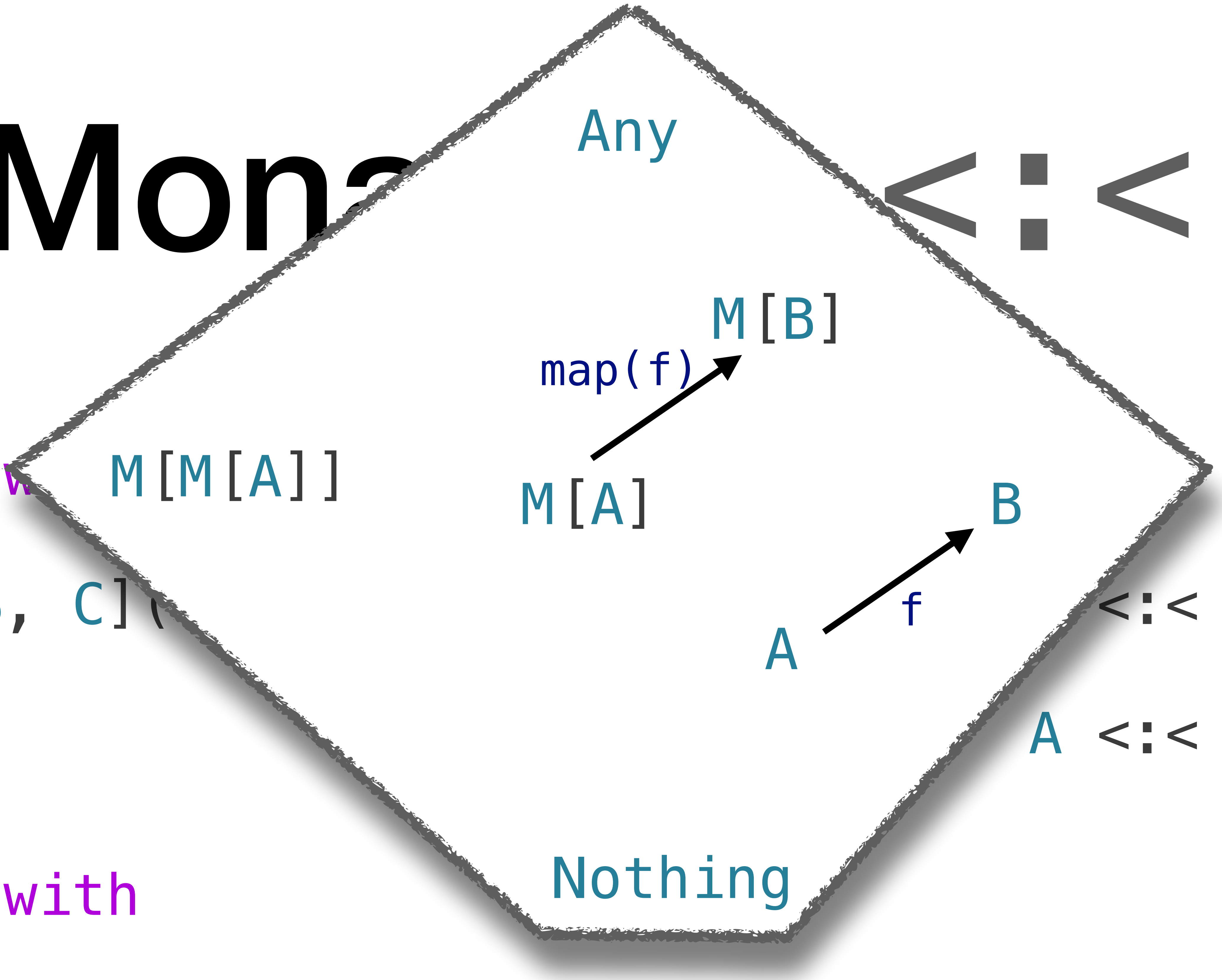

```
given Category[<:<] w…
    def andThen[A, B, C](…  <:< C  =  f andThen g
    def id[A]:                A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :  M[A]    <:< M[B] = ???
    def pure[A]                :     A    <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mona <:<



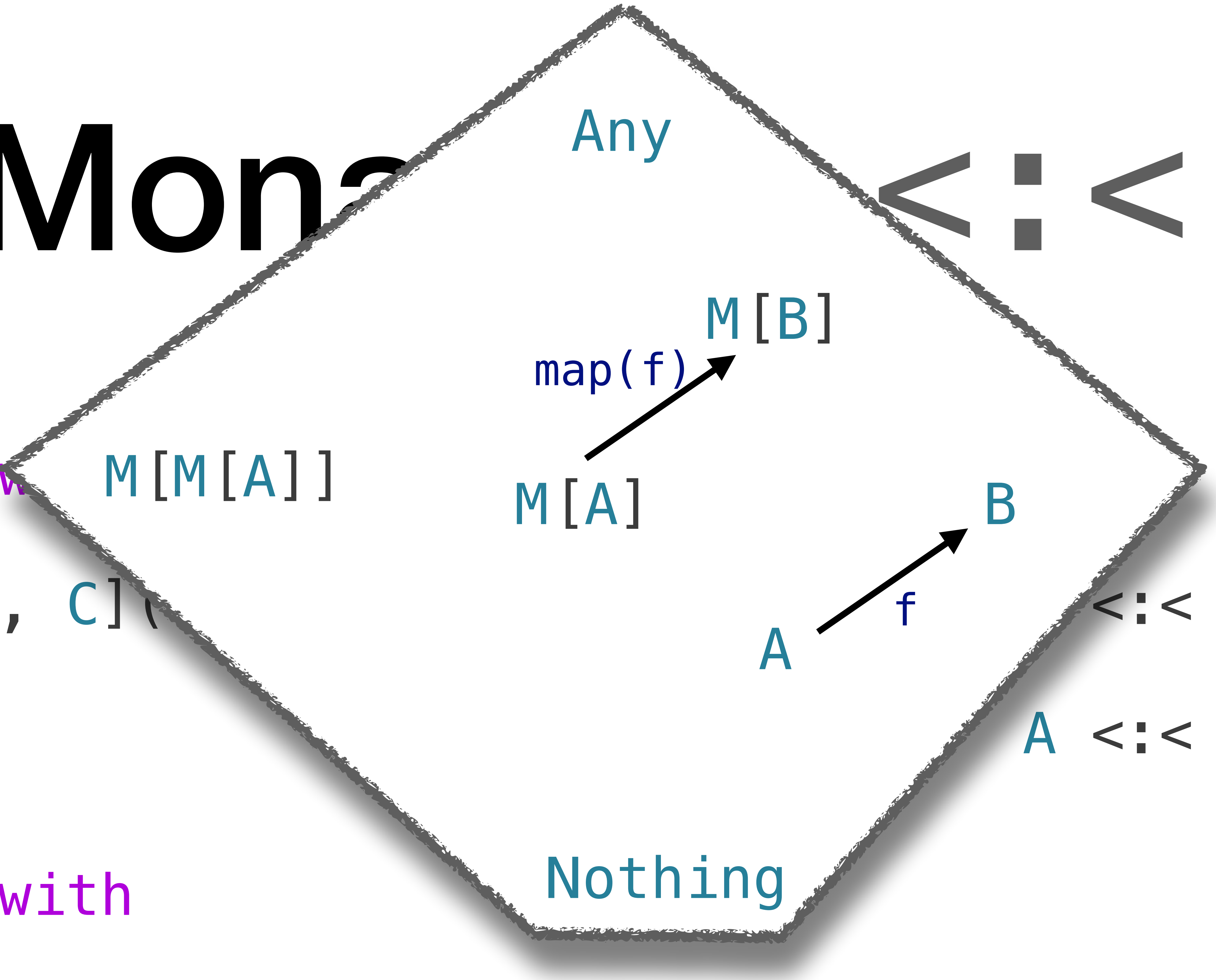

```
given Category[<:<] w
    def andThen[A, B, C](                                  <:< C  =  f andThen g
    def id[A]:                                           A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :    M[A]   <:< M[B] = ???
    def pure[A]                :      A    <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mona <:<



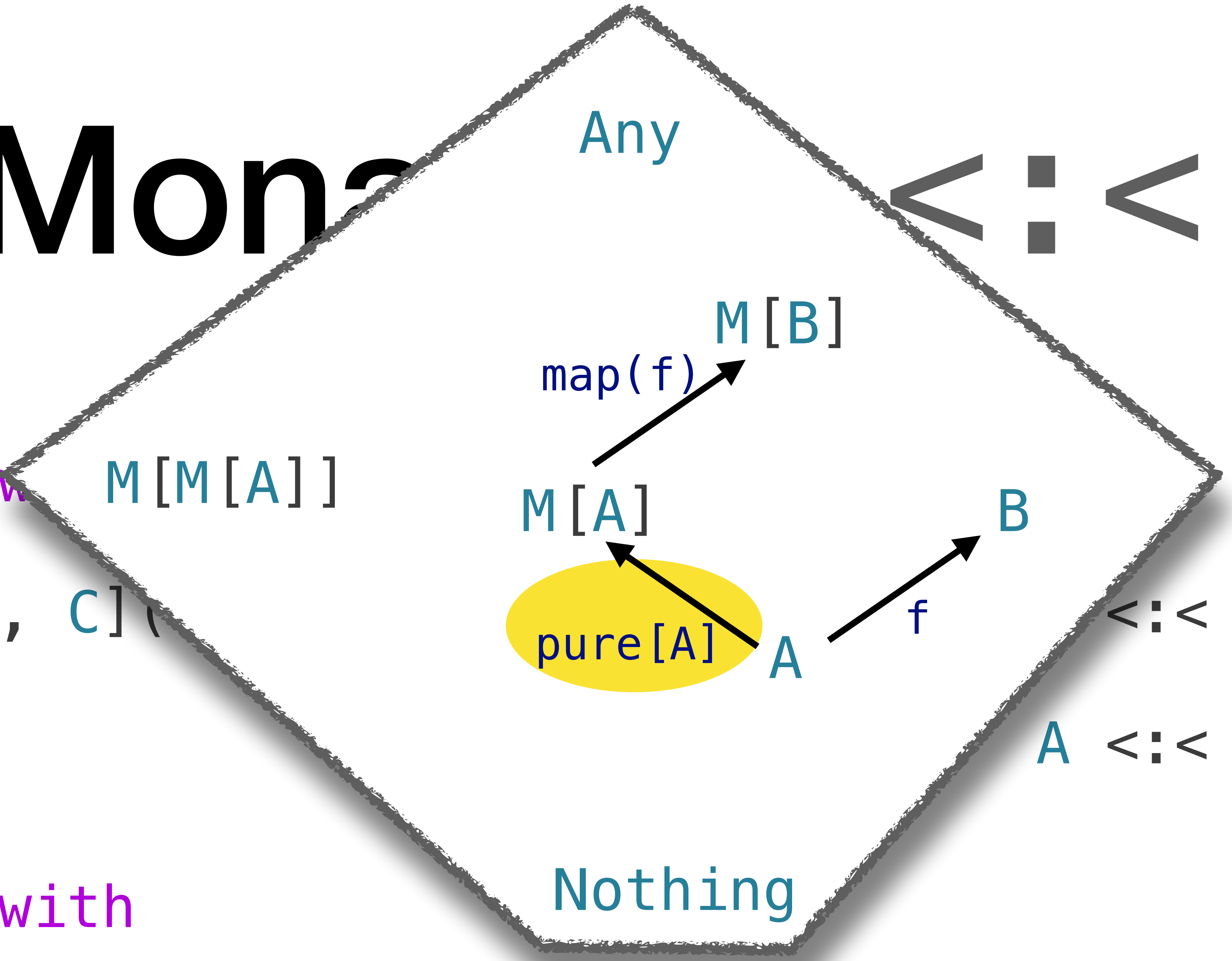

```
given Category[<:<] w
    def andThen[A, B, C](                    <:< C  =  f andThen g
    def id[A]:                          A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???
    def pure[A]                :   A     <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mon <:<



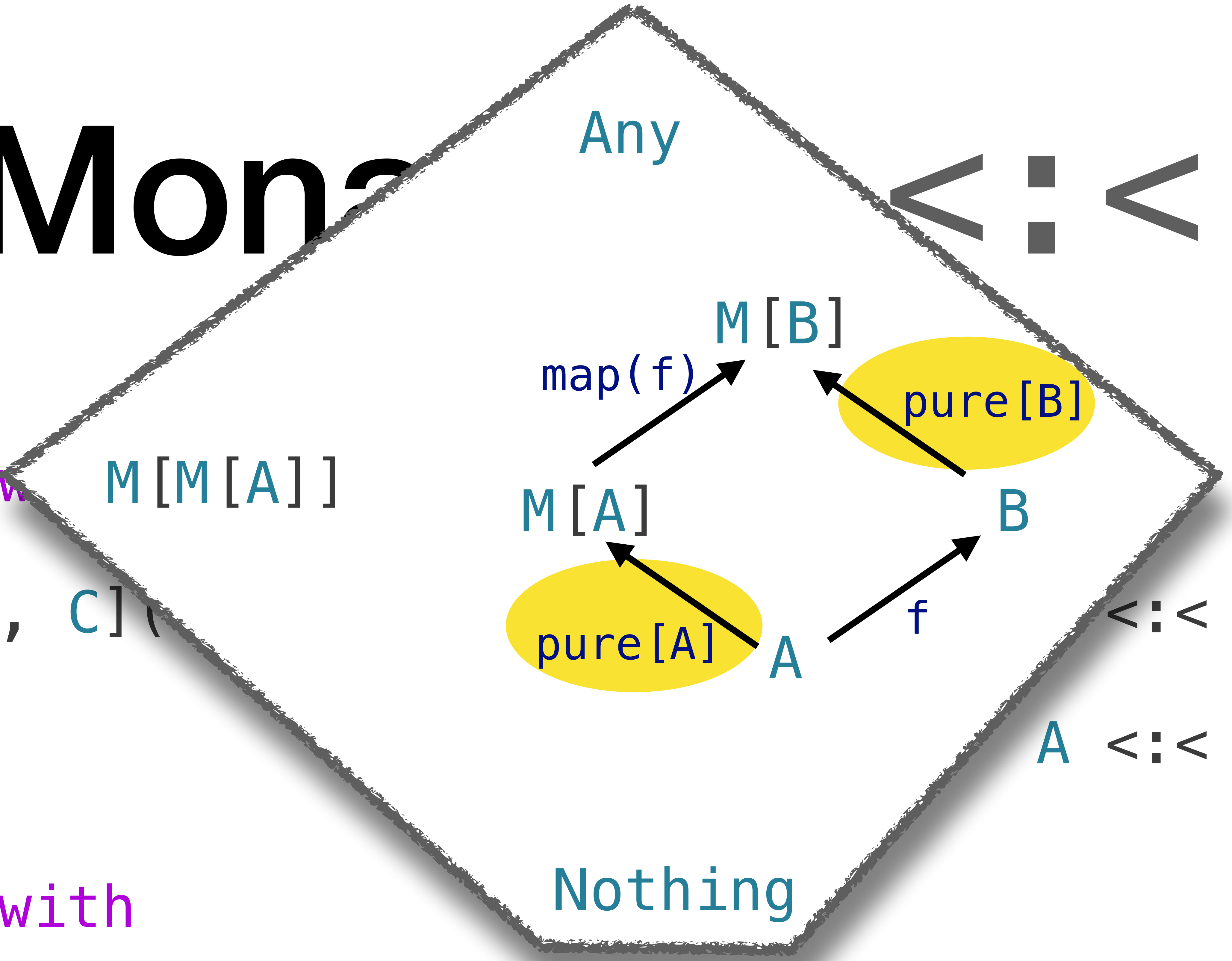

```
given Category[<:<] w
    def andThen[A, B, C](                         <:< C  =  f andThen g
    def id[A]:                                  A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :    M[A]  <:< M[B] = ???
    def pure[A]                :    A     <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mon <:<



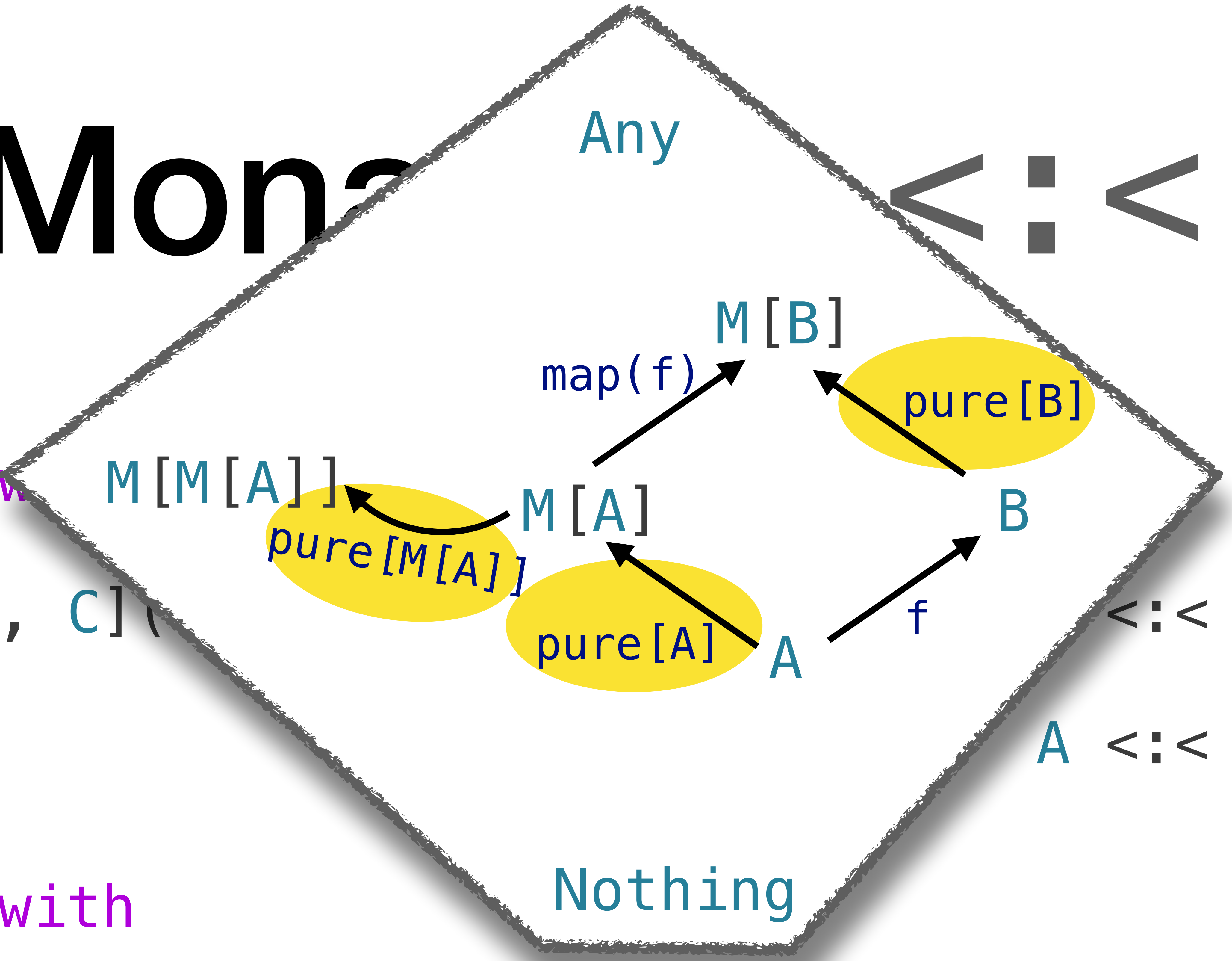

```
given Category[<:<] w
    def andThen[A, B, C](          <:< C  =  f andThen g
    def id[A]:                  A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???
    def pure[A]                :   A     <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mona <:<




```
given Category[<:<] w
    def andThen[A, B, C](                                   <:< C  =  f andThen g
    def id[A]:                                        A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???
    def pure[A]                :   A     <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

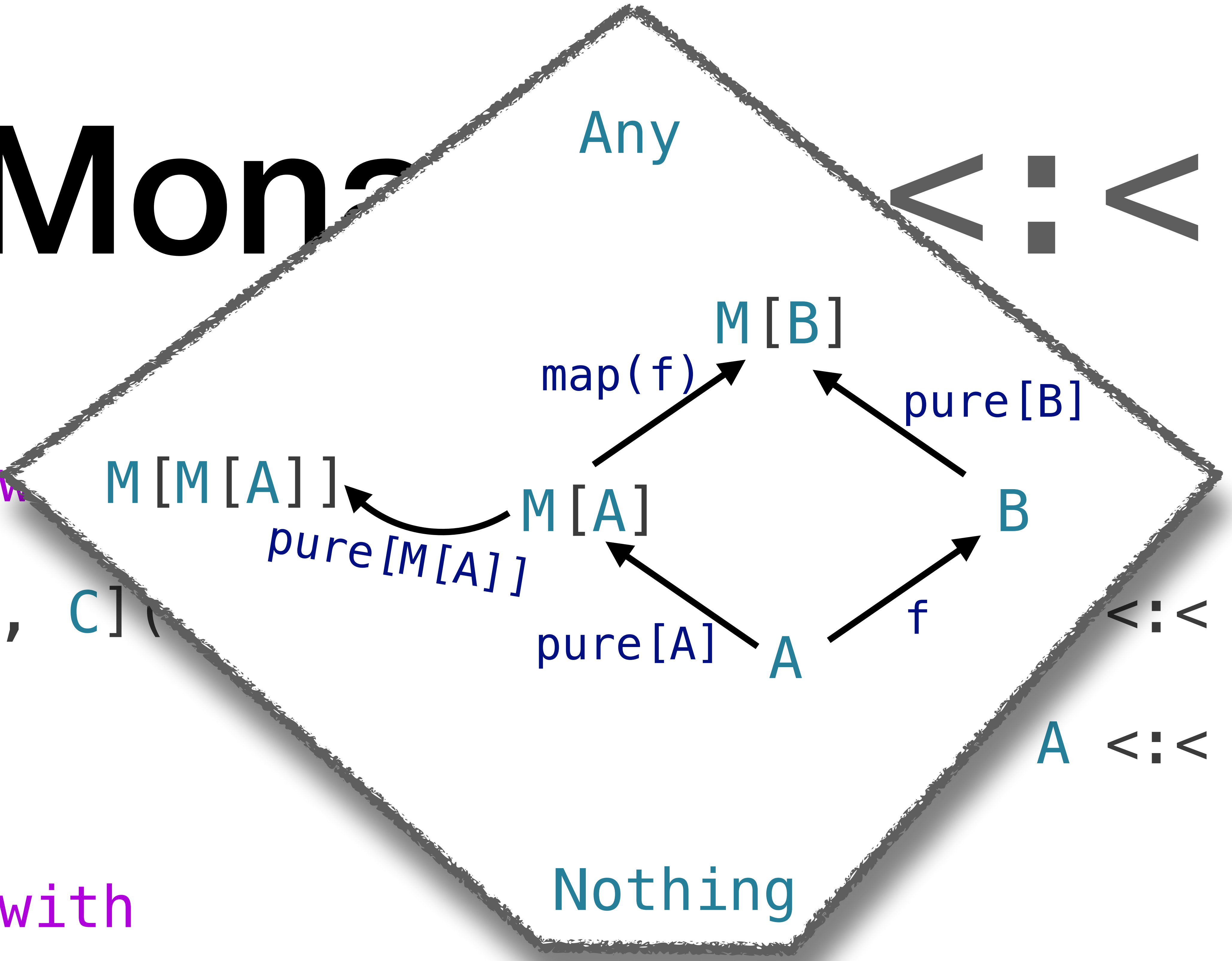
➡ Monotone

➡ Extensive

➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mona <:<




```
given Category[<:<] w
    def andThen[A, B, C](                          <:< C  =  f andThen g
    def id[A]:                                   A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]   <:< M[B] = ???
    def pure[A]                :     A    <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

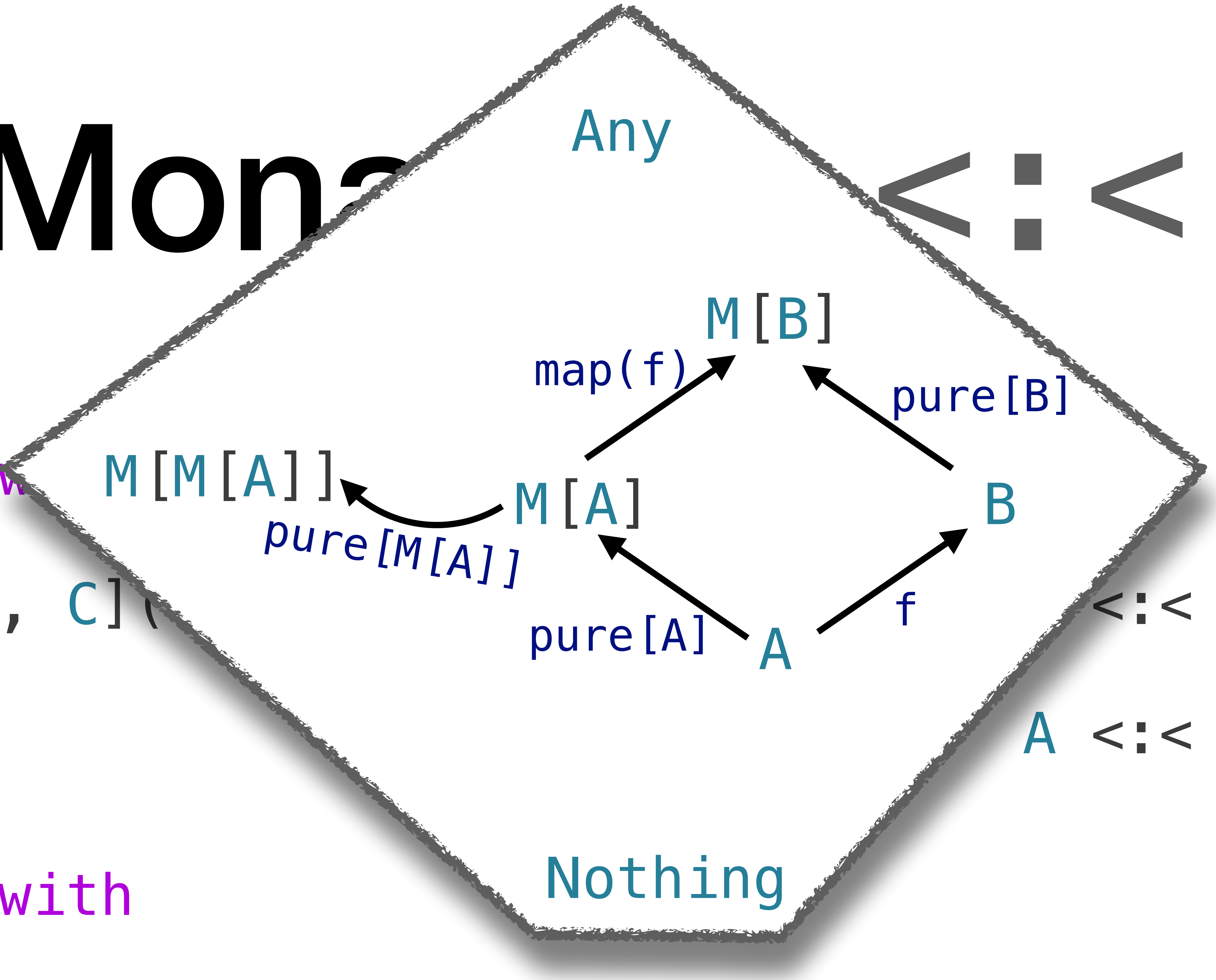
➡ Monotone

➡ Extensive

➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mon <:<



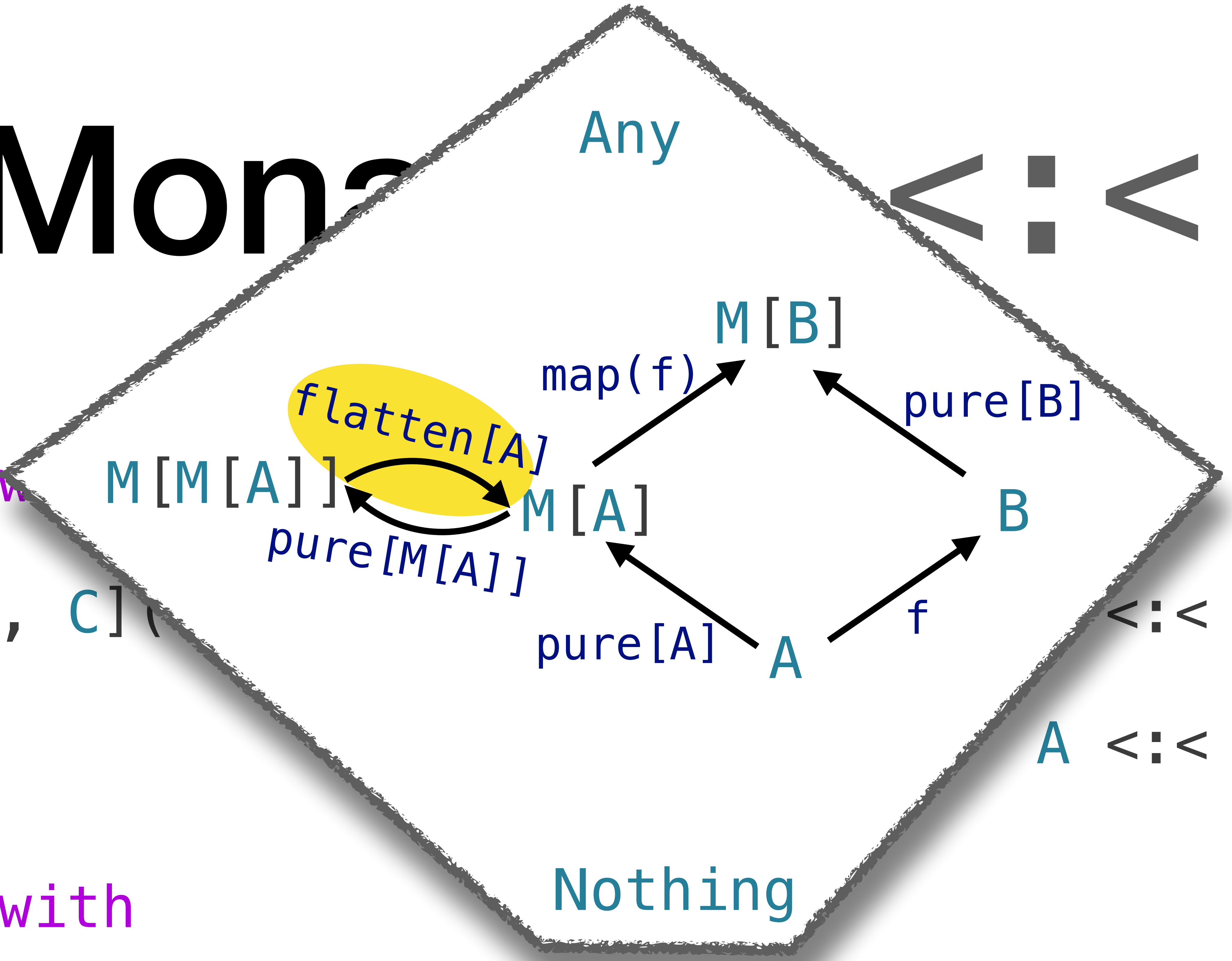

```
given Category[<:<] w
    def andThen[A, B, C](                               <:< C  =  f andThen g
    def id[A]:                                      A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :    M[A]  <:< M[B] = ???
    def pure[A]                :      A   <:< M[A] = ???
    def flatten[A]             : M[M[A]]  <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mona <:<



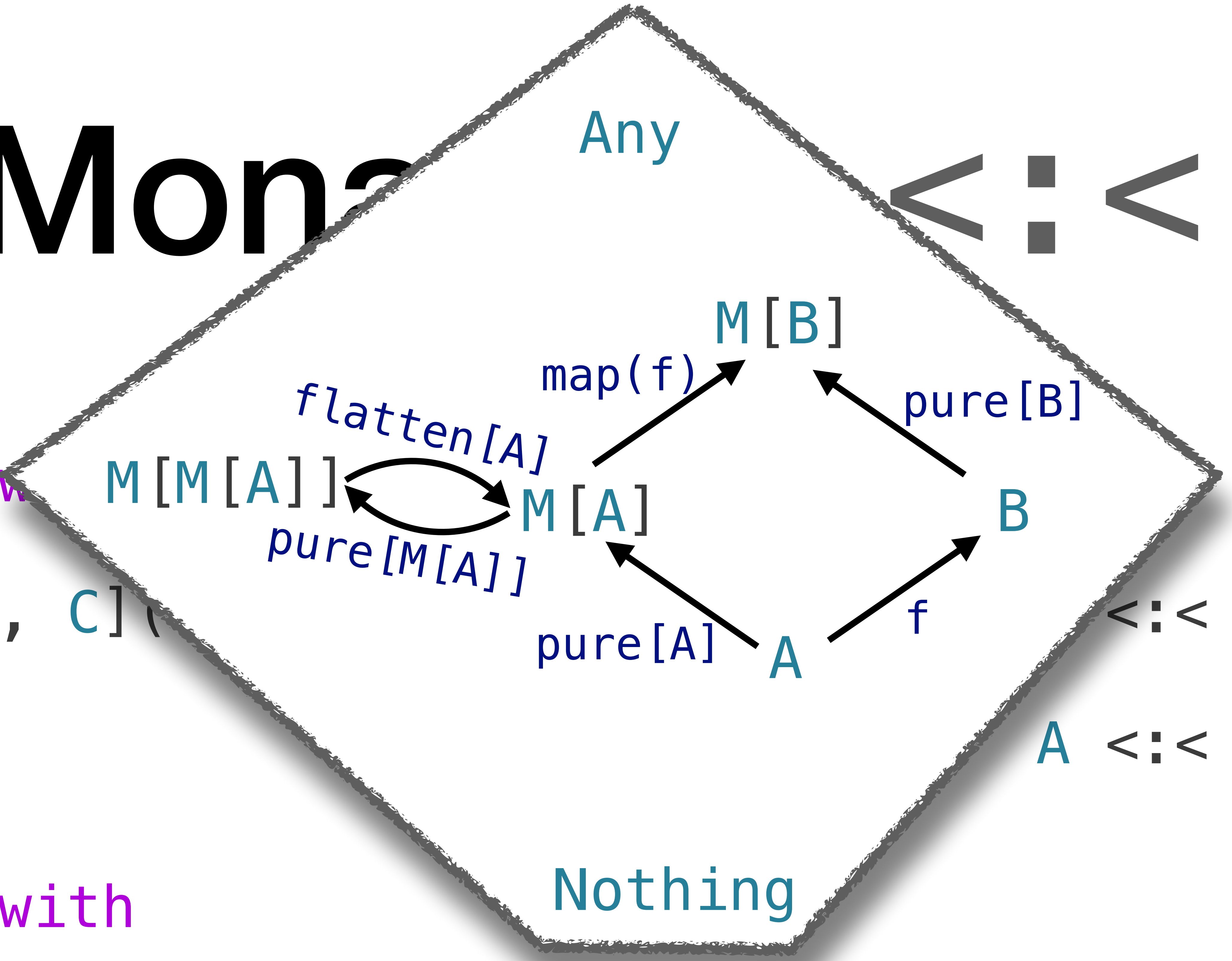

```
given Category[<:<] w
    def andThen[A, B, C](                    <:< C  =  f andThen g
    def id[A]:                          A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???
    def pure[A]                :     A   <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mon <:<



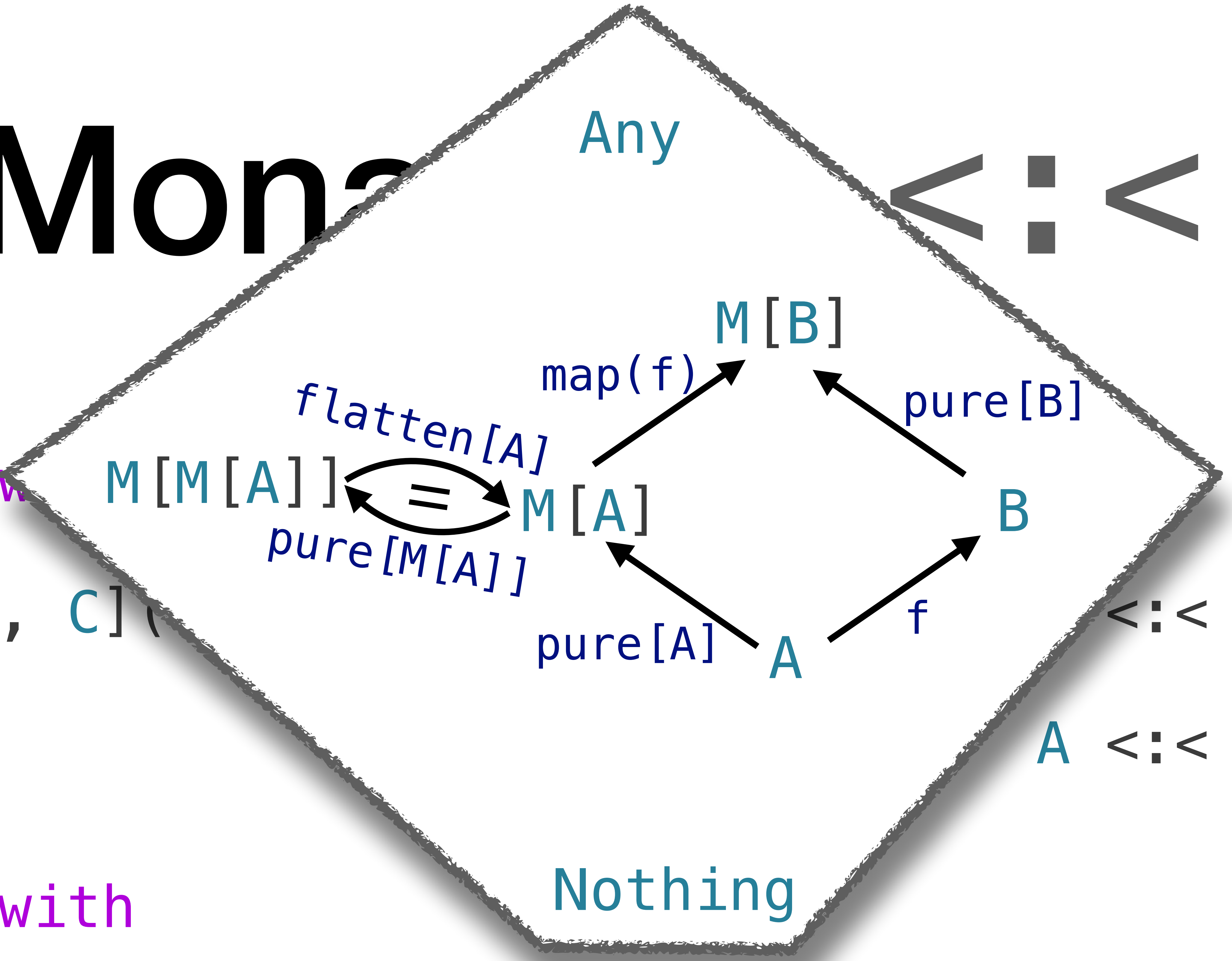

```
given Category[<:<] w
    def andThen[A, B, C](                    <:< C  =  f andThen g
    def id[A]:                             A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :    M[A]  <:< M[B] = ???
    def pure[A]                :      A   <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Mona <:<



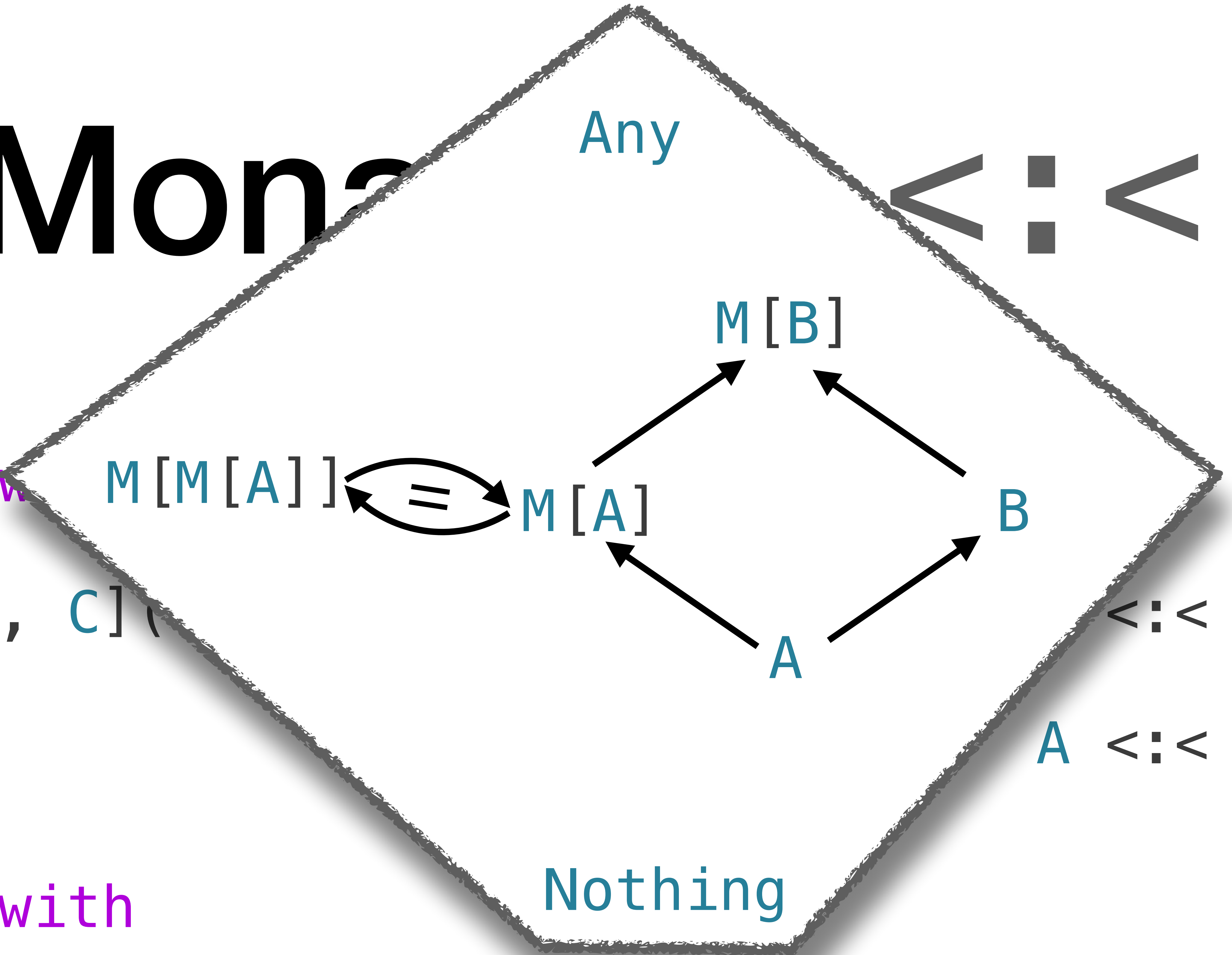

```
given Category[<:<] w
    def andThen[A, B, C](                                   <:< C  =  f andThen g
    def id[A]:                                            A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :   M[A]  <:< M[B] = ???
    def pure[A]                :     A   <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category `<:<` are **Closure Operators.**

Mon… <:<




```
given Category[<:<] w
    def andThen[A, B, C](                                  <:< C  =  f andThen g
    def id[A]:                                           A <:< A  =  <:<.refl

given Monad[<:<, M] with
    def map[A, B] (f: A <:< B) :    M[A]  <:< M[B] = ???
    def pure[A]                :      A   <:< M[A] = ???
    def flatten[A]             : M[M[A]] <:< M[A] = ???
```

M[_] must be:

- ➡ Monotone
- ➡ Extensive
- ➡ Idempotent

Monads in the category <:< are **Closure Operators.**

Monads in <:< : Example

```
type Res[+A] = Error | A

given Monad[<:<, Res] with

    def map[A, B] (f: A <:< B) : Res[A] <:< Res[B] =

        ???

    def pure[A] : A <:< Res[A] =

        ???

    def flatten[A] : Res[Res[A]] <:< Res[A] =

        ???
```

Monads in <:< : Example

```scala
type Res[+A] = Error | A

given Monad[<:<, Res] with

    def map[A, B] (f: A <:< B) : Res[A] <:< Res[B] =

        f.liftCo[Res]

    def pure[A] : A <:< Res[A] =

        ???

    def flatten[A] : Res[Res[A]] <:< Res[A] =

        ???
```

Monads in <:< : Example

```
type Res[+A] = Error | A

given Monad[<:<, Res] with
    def map[A, B] (f: A <:< B) : Res[A] <:< Res[B] =
        f.liftCo[Res]
    def pure[A] : A <:< Res[A] =
        summon[A <:< (Error | A)]
    def flatten[A] : Res[Res[A]] <:< Res[A] =
        ???
```

Monads in <:< : Example

```
type Res[+A] = Error | A

given Monad[<:<, Res] with
    def map[A, B] (f: A <:< B) : Res[A] <:< Res[B] =
        f.liftCo[Res]
    def pure[A] : A <:< Res[A] =
        summon[A <:< (Error | A)]
    def flatten[A] : Res[Res[A]] <:< Res[A] =
        summon[(Error | Error | A) <:< (Error | A)]
```

Monads in <:< : Example

```
type Res[+A] = Error | A

given Monad[<:<, Res] with
    def map[A, B] (f: A <:< B) : Res[A] <:< Res[B] =
        f.liftCo[Res]
    def pure[A] : A <:< Res[A] =
        summon[A <:< (Error | A)]
    def flatten[A] : Res[Res[A]] <:< Res[A] =
        summon[(Error | Error | A) <:< (Error | A)]
```

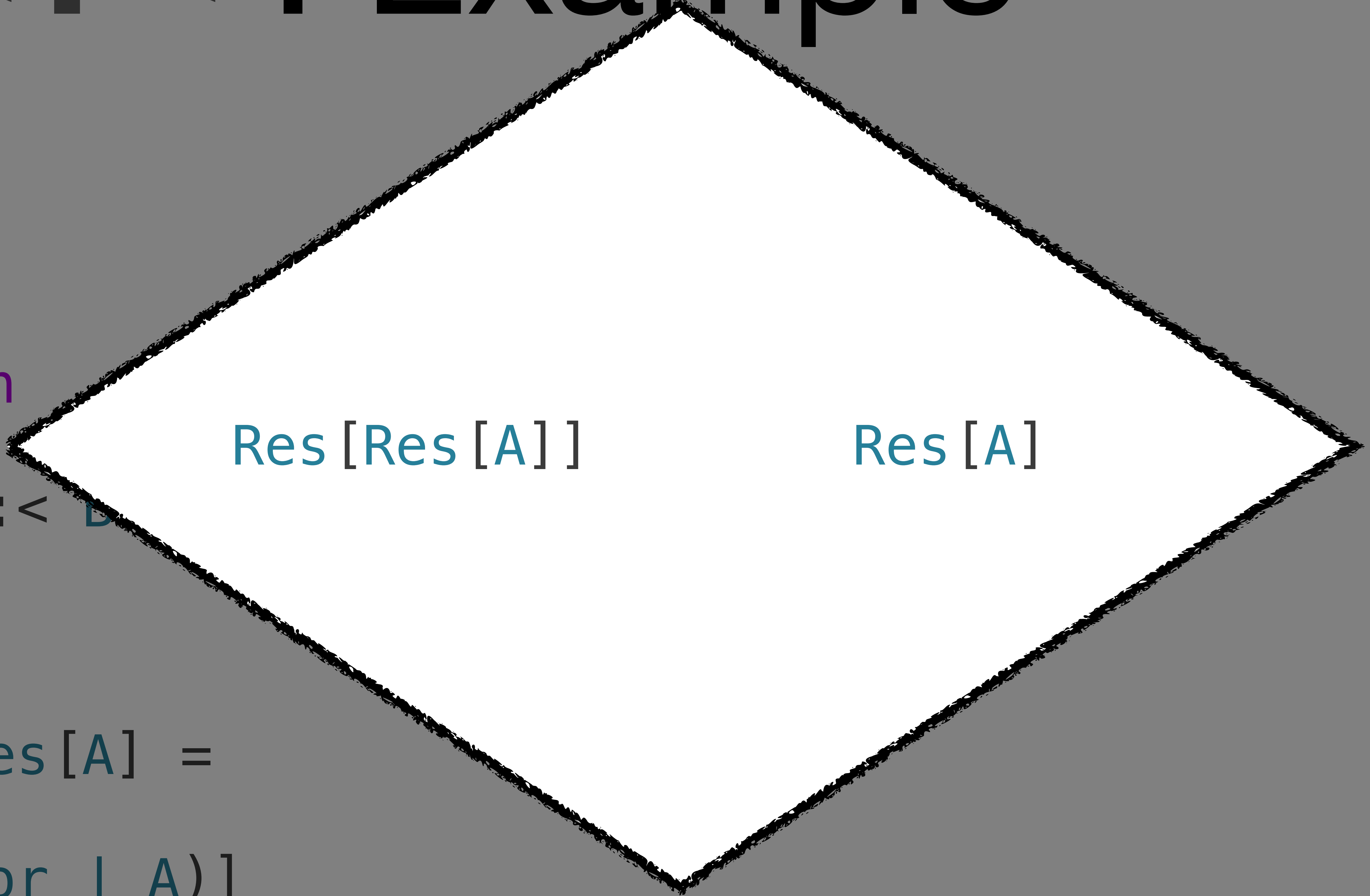
Sequencing?

Monads in <:< : Example

```
type Res[+A] = Error | A

given Monad[<:<, Res] with
    def map[A, B] (f: A <:< B) : Res[A] <:< Res[B] =
        f.liftCo[Res]
    def pure[A] : A <:< Res[A] =
        summon[A <:< (Error | A)]
    def flatten[A] : Res[Res[A]] <:< Res[A] =
        summon[(Error | Error | A) <:< (Error | A)]
```

Sequencing?

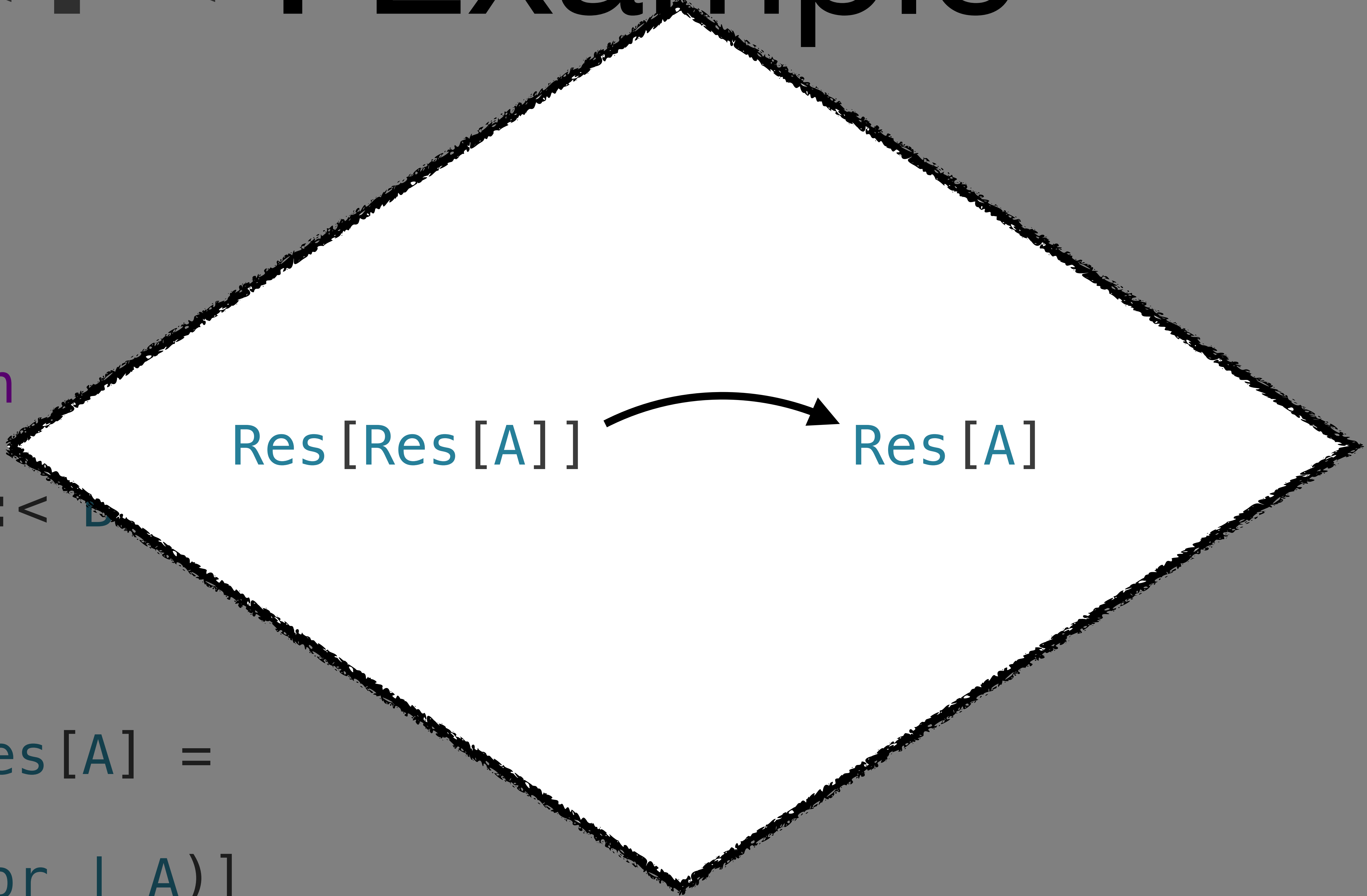
Monads in <:< : Example




```
type Res[+A] = Error | A

given Monad[<:<, Res] with
    def map[A, B] (f: A <:< B
        f.liftCo[Res]
    def pure[A] : A <:< Res[A] =
        summon[A <:< (Error | A)]
    def flatten[A] : Res[Res[A]] <:< Res[A] =
        summon[(Error | Error | A) <:< (Error | A)]
```

Sequencing?

Monads in <:< : Example

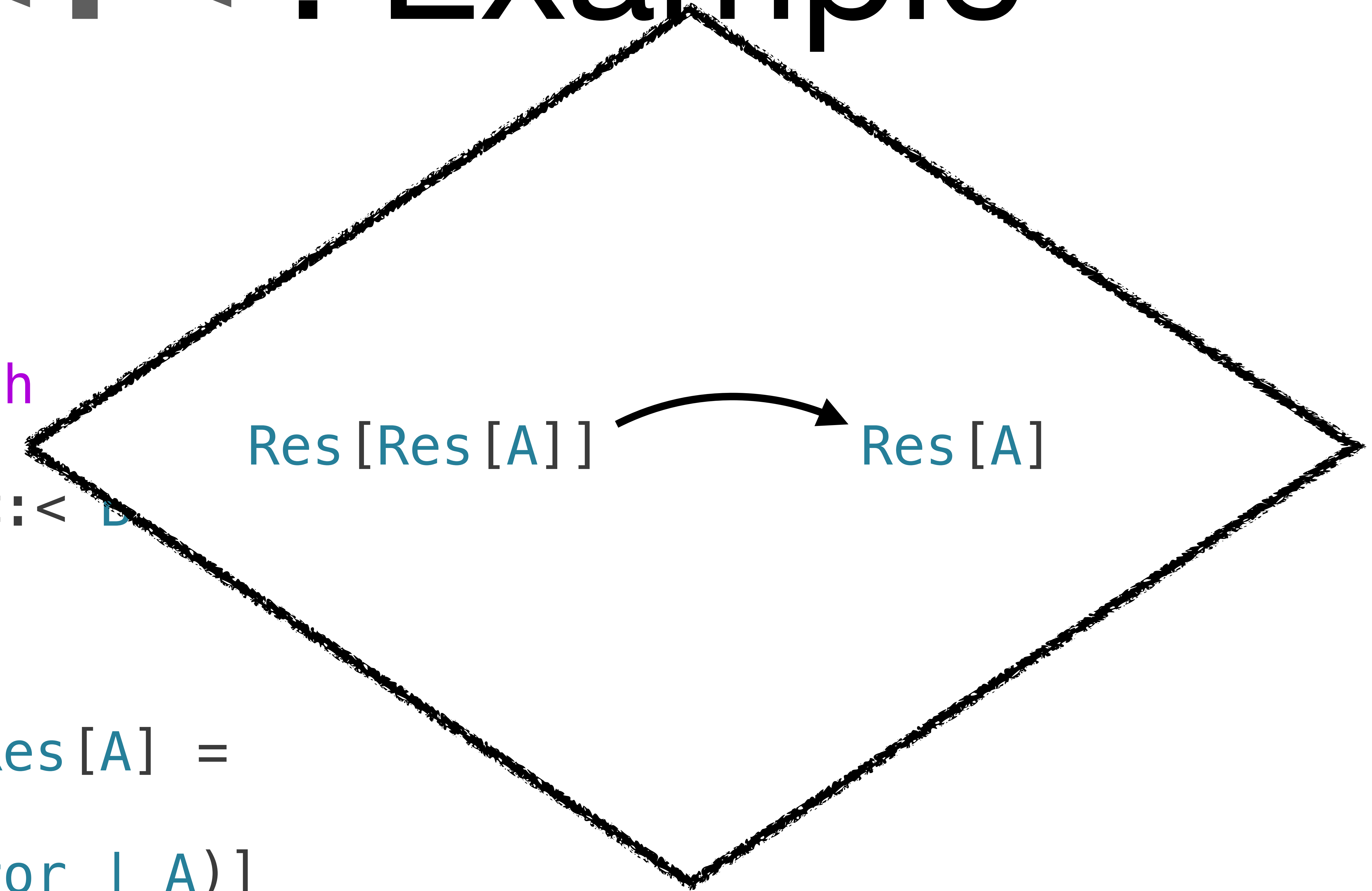



```
type Res[+A] = Error | A

given Monad[<:<, Res] with
    def map[A, B] (f: A <:< B
        f.liftCo[Res]
    def pure[A] : A <:< Res[A] =
        summon[A <:< (Error | A)]
    def flatten[A] : Res[Res[A]] <:< Res[A] =
        summon[(Error | Error | A) <:< (Error | A)]
```

Sequencing?

Monads in <:< : Example

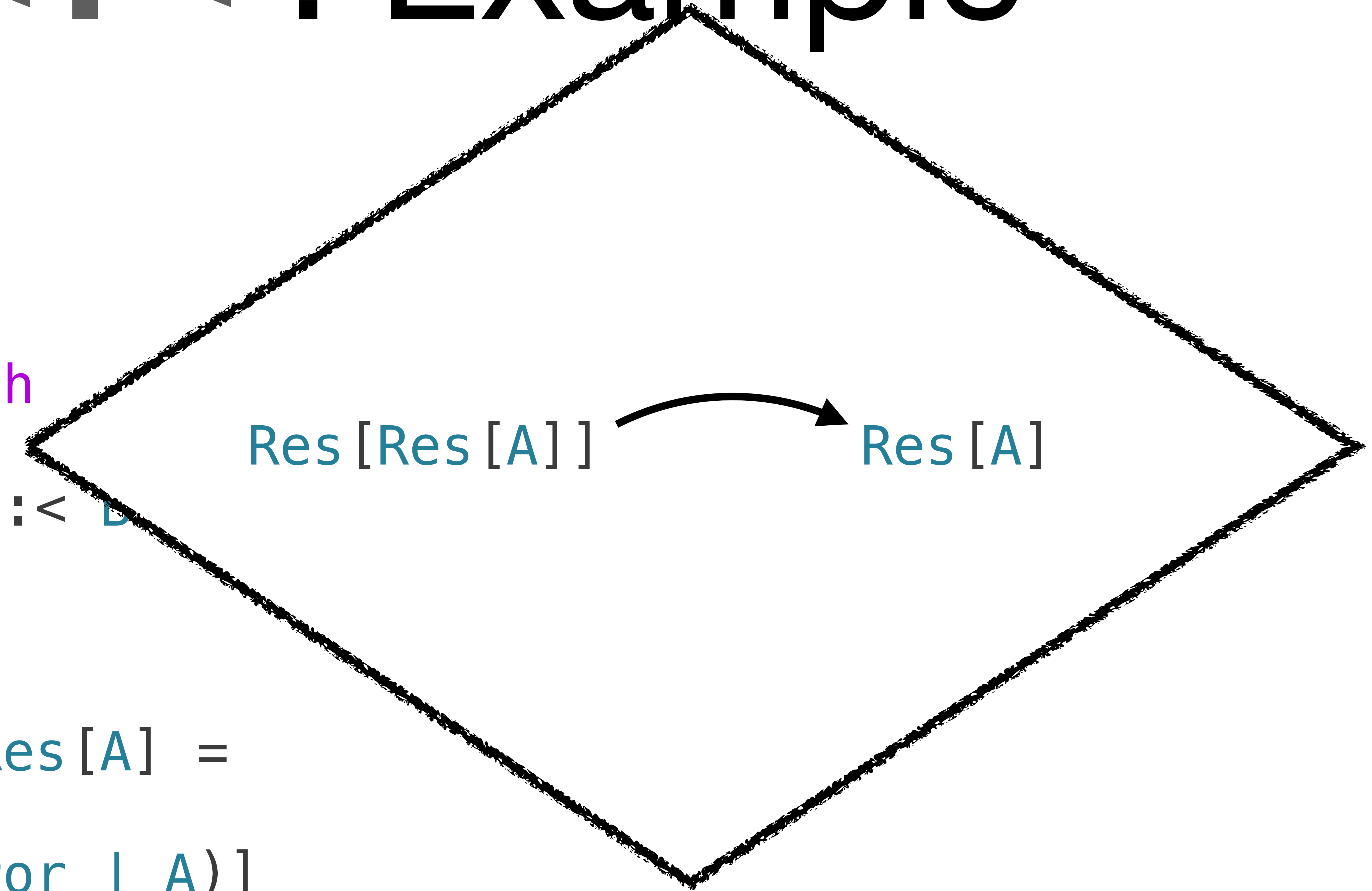
```
type Res[+A] = Error | A

given Monad[<:<, Res] with
    def map[A, B] (f: A <:< B
        f.liftCo[Res]
    def pure[A] : A <:< Res[A] =
        summon[A <:< (Error | A)]
    def flatten[A] : Res[Res[A]] <:< Res[A] =
        summon[(Error | Error | A) <:< (Error | A)]
```



Sequencing?

Monads in <:< : Example

```
type Res[+A] = Error | A

given Monad[<:<, Res] with
    def map[A, B] (f: A <:< B
        f.liftCo[Res]
    def pure[A] : A <:< Res[A] =
        summon[A <:< (Error | A)]
    def flatten[A] : Res[Res[A]] <:< Res[A] =
        summon[(Error | Error | A) <:< (Error | A)]
```



Sequencing? Not even execution! Just statements about types.

Lesson So Far

- Monads definable in **any** Category (even non-executable one, like `<:<`)

Game Over

GAME OVER

EXIT

GAME OVER

EXIT

PLAY AGAIN

GAME OVER

EXIT

PLAY AGAIN

The Opposite Category

```
case class Op[A, B](run: B => A)
```

The Opposite Category

```scala
case class Op[A, B](run: B => A)

type <= [A, B] = Op[A, B]
```

The Opposite Category

```scala
case class Op[A, B](run: B => A)

type <= [A, B] = Op[A, B]

given Category[<=] with
```

The Opposite Category

```scala
case class Op[A, B](run: B => A)

type <= [A, B] = Op[A, B]

given Category[<=] with
    def andThen[A, B, C](f: A <= B, g: B <= C): A <= C =
```

The Opposite Category

```scala
case class Op[A, B](run: B => A)

type <= [A, B] = Op[A, B]

given Category[<=] with
    def andThen[A, B, C](f: A <= B, g: B <= C): A <= C =
        Op(g.run andThen f.run)
```

The Opposite Category

```
case class Op[A, B](run: B => A)

type <= [A, B] = Op[A, B]

given Category[<=] with
    def andThen[A, B, C](f: A <= B, g: B <= C): A <= C =
        Op(g.run andThen f.run)
    def id[A]: A <= A =
```

The Opposite Category

```scala
case class Op[A, B](run: B => A)

type <= [A, B] = Op[A, B]

given Category[<=] with
    def andThen[A, B, C](f: A <= B, g: B <= C): A <= C =
        Op(g.run andThen f.run)
    def id[A]: A <= A =
        Op(a => a)
```

Monads in <=

Monads in <=

Monads in **<=** are exactly the *co-monads* in **=>**.

Monads in <=

Monads in **<=** are exactly the *co-monads* in **=>**.

Example: The **`Id`** (co-)monad

```
case class Id[A](a: A)
```

Monads in <=

Monads in **<=** are exactly the *co-monads* in **=>**.

Example: The **`Id`** (co-)monad

```
case class Id[A](a: A)


given Monad[<=, Id] with
```

Monads in <=

Monads in **<=** are exactly the *co-monads* in **=>**.

Example: The **Id** (co-)monad

```
case class Id[A](a: A)

given Monad[<=, Id] with
    def map[A, B](f: A <= B): Id[A] <= Id[B] =
```

Monads in <=

Monads in **<=** are exactly the *co-monads* in **=>**.

Example: The **`Id`** (co-)monad

```
case class Id[A](a: A)

given Monad[<=, Id] with

    def map[A, B](f: A <= B): Id[A] <= Id[B] =
        Op { case Id(b) => Id(f.run(b)) }
```

Monads in <=

Monads in **<=** are exactly the *co-monads* in **=>**.

Example: The **Id** (co-)monad

```
case class Id[A](a: A)

given Monad[<=, Id] with
  def map[A, B](f: A <= B): Id[A] <= Id[B] =
    Op { case Id(b) => Id(f.run(b)) }
  def pure[A]: A <= Id[A] =
```

Monads in <=

Monads in **<=** are exactly the *co-monads* in **=>**.

Example: The **Id** (co-)monad

```
case class Id[A](a: A)

given Monad[<=, Id] with
  def map[A, B](f: A <= B): Id[A] <= Id[B] =
    Op { case Id(b) => Id(f.run(b)) }
  def pure[A]: A <= Id[A] =
    Op(_.a)
```

Monads in <=

Monads in **<=** are exactly the *co-monads* in **=>**.

Example: The **Id** (co-)monad

```
case class Id[A](a: A)

given Monad[<=, Id] with

    def map[A, B](f: A <= B): Id[A] <= Id[B] =
        Op { case Id(b) => Id(f.run(b)) }

    def pure[A]: A <= Id[A] =
        Op(_.a)

    def flatten[A]: Id[Id[A]] <= Id[A] =
```

Monads in <=

Monads in **<=** are exactly the *co-monads* in **=>**.

Example: The **Id** (co-)monad

```
case class Id[A](a: A)

given Monad[<=, Id] with
  def map[A, B](f: A <= B): Id[A] <= Id[B] =
    Op { case Id(b) => Id(f.run(b)) }
  def pure[A]: A <= Id[A] =
    Op(_.a)
  def flatten[A]: Id[Id[A]] <= Id[A] =
    Op(Id(_))
```

Monads in <=

```
val f: String  <= Id[Boolean]   = Op { case Id(b)  => println("f"); String.valueOf(b) }
val g: Boolean <= Id[Int]       = Op { case Id(i)  => println("g"); i % 2 == 0 }
val h: Int     <= Id[List[Int]] = Op { case Id(xs) => println("h"); xs.sum }
```

Full code at https://github.com/TomasMikula/non-sequencing-monads/

Monads in <=

```scala
val f: String  <= Id[Boolean]     = Op { case Id(b)  => println("f"); String.valueOf(b) }
val g: Boolean <= Id[Int]         = Op { case Id(i)  => println("g"); i % 2 == 0 }
val h: Int     <= Id[List[Int]]   = Op { case Id(xs) => println("h"); xs.sum }
```

Full code at https://github.com/TomasMikula/non-sequencing-monads/

Monads in <=

```
val f: String  <= Id[Boolean]   = Op { case Id(b)  => println("f"); String.valueOf(b) }
val g: Boolean <= Id[Int]       = Op { case Id(i)  => println("g"); i % 2 == 0 }
val h: Int     <= Id[List[Int]] = Op { case Id(xs) => println("h"); xs.sum }
```

Full code at https://github.com/TomasMikula/non-sequencing-monads/

Monads in <=

```scala
val f: String  <= Id[Boolean]   = Op { case Id(b)  => println("f"); String.valueOf(b) }
val g: Boolean <= Id[Int]       = Op { case Id(i)  => println("g"); i % 2 == 0 }
val h: Int     <= Id[List[Int]] = Op { case Id(xs) => println("h"); xs.sum }
```

Full code at https://github.com/TomasMikula/non-sequencing-monads/

Monads in <=

```scala
val f: String  <= Id[Boolean]   = Op { case Id(b)  => println("f"); String.valueOf(b) }
val g: Boolean <= Id[Int]       = Op { case Id(i)  => println("g"); i % 2 == 0 }
val h: Int     <= Id[List[Int]] = Op { case Id(xs) => println("h"); xs.sum }


(f >=> g >=> h)
    .run(Id(Nil))
```

Full code at https://github.com/TomasMikula/non-sequencing-monads/

Monads in <=

```scala
val f: String  <= Id[Boolean]   = Op { case Id(b)  => println("f"); String.valueOf(b) }
val g: Boolean <= Id[Int]       = Op { case Id(i)  => println("g"); i % 2 == 0 }
val h: Int     <= Id[List[Int]] = Op { case Id(xs) => println("h"); xs.sum }


(f >=> g >=> h)
    .run(Id(Nil))

// Output:
```

Full code at https://github.com/TomasMikula/non-sequencing-monads/

Monads in <=

```
val f: String  <= Id[Boolean]   = Op { case Id(b)  => println("f"); String.valueOf(b) }
val g: Boolean <= Id[Int]       = Op { case Id(i)  => println("g"); i % 2 == 0 }
val h: Int     <= Id[List[Int]] = Op { case Id(xs) => println("h"); xs.sum }


(f >=> g >=> h)
    .run(Id(Nil))

// Output:
//
//   h
//   g
//   f
```

Full code at https://github.com/TomasMikula/non-sequencing-monads/

Monads in <=

```
val f: String  <= Id[Boolean]   = Op { case Id(b)  => println("f"); String.valueOf(b) }
val g: Boolean <= Id[Int]       = Op { case Id(i)  => println("g"); i % 2 == 0 }
val h: Int     <= Id[List[Int]] = Op { case Id(xs) => println("h"); xs.sum }


(f >=> g >=> h)
    .run(Id(Nil))

// Output:
//
//   h
//   g
//   f
```

Reverse Order!

Full code at https://github.com/TomasMikula/non-sequencing-monads/

Monads in <=

```scala
val f: String  <= Id[Boolean]   = Op { case Id(b)  => println("f"); String.valueOf(b) }
val g: Boolean <= Id[Int]       = Op { case Id(i)  => println("g"); i % 2 == 0 }
val h: Int     <= Id[List[Int]] = Op { case Id(xs) => println("h"); xs.sum }
```

```scala
(f >=> g >=> h)
    .run(Id(Nil))

// Output:
//
//   h
//   g
//   f
```

Reverse Order!

```scala
(for
    s <- summon[Monad[<=, Id]].pure[String]
    b <- f(s)
    i <- g(b)
    l <- h(i)
yield l)
    .run(Id(Nil))
```

Full code at https://github.com/TomasMikula/non-sequencing-monads/

Monads in <=

```scala
val f: String  <= Id[Boolean]   = Op { case Id(b)  => println("f"); String.valueOf(b) }
val g: Boolean <= Id[Int]       = Op { case Id(i)  => println("g"); i % 2 == 0 }
val h: Int     <= Id[List[Int]] = Op { case Id(xs) => println("h"); xs.sum }
```

```scala
(f >=> g >=> h)
    .run(Id(Nil))

// Output:
//
//   h
//   g
//   f
```

Reverse Order!

```scala
(for
    s <- summon[Monad[<=, Id]].pure[String]
    b <- f(s)
    i <- g(b)
    l <- h(i)
yield l)
    .run(Id(Nil))
```

```scala
// Output:
//
//   h
//   g
//   f
```

Full code at https://github.com/TomasMikula/non-sequencing-monads/

Monads in <=

```scala
val f: String  <= Id[Boolean]   = Op { case Id(b)  => println("f"); String.valueOf(b) }
val g: Boolean <= Id[Int]       = Op { case Id(i)  => println("g"); i % 2 == 0 }
val h: Int     <= Id[List[Int]] = Op { case Id(xs) => println("h"); xs.sum }
```

```scala
(f >=> g >=> h)
    .run(Id(Nil))

// Output:
//
//   h
//   g
//   f
```

Reverse Order!

```scala
(for
    s <- summon[Monad[<=, Id]].pure[String]
    b <- f(s)
    i <- g(b)
    l <- h(i)
yield l)
    .run(Id(Nil))

// Output:
//
//   h
//   g
//   f
```

Reverse Order!

Full code at https://github.com/TomasMikula/non-sequencing-monads/

Lessons So Far

- Monads definable in **any** Category (even non-executable one, like `<:<`)

Lessons So Far

- Monads definable in **any** Category (even non-executable one, like `<:<`)
- **Syntactically,** monads *do* support **sequential composition**

Lessons So Far

- Monads definable in **any** Category (even non-executable one, like **`<:<`**)
- **Syntactically,** monads *do* support **sequential composition**
- Sequential composition ≠ sequential **execution** (e.g. monads in **`<=`**)

Y U NO UNDERSTAND

IT’S THE **EFFECTS** THAT ARE SEQUENCED

“`flatten` sequences the *effects*.”

“`flatten` sequences the *effects*.”

What?

“`flatten` sequences the *effects*.”

What?

Hidden assumptions:

“`flatten` sequences the *effects*.”

What?

Hidden assumptions:

- an “*effect*” associated with every monad

“`flatten` sequences the *effects*.”

What?

Hidden assumptions:

- an *“effect”* associated with every monad
- clear what *“sequencing of effects”* means

“`flatten` sequences the *effects*.”

What?

Hidden assumptions:

- an *“effect”* associated with every monad
- clear what *“sequencing of effects”* means

A **vague** statement that can be **reinterpreted** at will.

“`flatten` sequences the *effects*.”

What?

Hidden assumptions:

- an *“effect”* associated with every monad
- clear what *“sequencing of effects”* means

A **vague** statement that can be **reinterpreted** at will.

Just reject this game. 🙅‍♂️

Demand clarity.

“`flatten` sequences the *effects*.”

What?

Hidden assumptions:

- an *“effect”* associated with every monad
- clear what *“sequencing of effects”* means

A **vague** statement that can be **reinterpreted** at will.

Just reject this game. 🙅‍♂️

Demand clarity.

Play some more.

“`flatten` sequences the effects.”

What?

Hidden assumptions:

- an *“effect”* associated with every monad
- clear what *“sequencing of effects”* means

A **vague** statement that can be **reinterpreted** at will.

Just reject this game. 🙅‍♂️
Demand clarity.

Play some more.

Writer Monad with a Twist

Writer Monad with a Twist

```
type Writer[A] = (List[String], A)
```

Writer Monad with a Twist

```scala
type Writer[A] = (List[String], A)

def flatten[A]: (List[String], (List[String], A)) => (List[String], A)
```

Writer Monad with a Twist

```
type Writer[A] = (List[String], A)
```

```
def flatten[A]: (List[String], (List[String], A)) => (List[String], A)
```

Effect: writing items to a log

Writer Monad with a Twist

```scala
type Writer[A] = (List[String], A)
```

```scala
def flatten[A]: (List[String], (List[String], A)) => (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

Writer Monad with a Twist

```scala
type Writer[A] = (List[String], A)
```

```scala
def flatten[A]: (List[String], (List[String], A)) => (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

```scala
(List("Fun"), (List("Scala"), (List("2023"), a)))
```

Writer Monad with a Twist

```scala
type Writer[A] = (List[String], A)
```

```scala
def flatten[A]: (List[String], (List[String], A)) => (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

```scala
(List("Fun"), (List("Scala"), (List("2023"), a)))
```

➡ `(List("Fun", "Scala", "2023"), a)`

Writer Monad with a Twist

```scala
type Writer[A] = (List[String], A)
```

```scala
def flatten[A]: (List[String], (List[String], A)) => (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

```scala
(List("Fun"), (List("Scala"), (List("2023"), a)))
```

➡ `(List("Fun", "Scala", "2023"), a)`

Twist:

Writer Monad with a Twist

```scala
type Writer[A] = (List[String], A)
```

```scala
def flatten[A]: (List[String], (List[String], A)) => (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

```scala
(List("Fun"), (List("Scala"), (List("2023"), a)))
```

➡ `(List("Fun", "Scala", "2023"), a)`

Twist:

`Writer[A]` is a lawful *Monad* for **any** lawful *Monoid* on `List[String]`.

Writer Monad with a Twist

```
type Writer[A] = (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

`Writer[A]` is a lawful *Monad* for **any** lawful *Monoid* on `List[String]`.

Writer Monad with a Twist

```
type Writer[A] = (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

`Writer[A]` is a lawful *Monad* for **any** lawful *Monoid* on `List[String]`.

```
given Monoid[List[String]] with
    def combine(a: List[String], b: List[String]): List[String] = b ++ a
    def unit: List[String] = Nil
```

Writer Monad with a Twist

```
type Writer[A] = (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

`Writer[A]` is a lawful *Monad* for **any** lawful *Monoid* on `List[String]`.

```
given Monoid[List[String]] with
    def combine(a: List[String], b: List[String]): List[String] = b ++ a
    def unit: List[String] = Nil
```

Writer Monad with a Twist

```
type Writer[A] = (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

`Writer[A]` is a lawful *Monad* for **any** lawful *Monoid* on `List[String]`.

```
given Monoid[List[String]] with
    def combine(a: List[String], b: List[String]): List[String] = b ++ a
    def unit: List[String] = Nil
```

reverse order

Writer Monad with a Twist

```
type Writer[A] = (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

`Writer[A]` is a lawful *Monad* for **any** lawful *Monoid* on `List[String]`.

```
given Monoid[List[String]] with
    def combine(a: List[String], b: List[String]): List[String] = b ++ a
    def unit: List[String] = Nil
```

reverse order

Lawful Monoid ✓

Writer Monad with a Twist

```
type Writer[A] = (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

`Writer[A]` is a lawful *Monad* for **any** lawful *Monoid* on `List[String]`.

```
given Monoid[List[String]] with
    def combine(a: List[String], b: List[String]): List[String] = b ++ a
    def unit: List[String] = Nil
```

reverse order

Lawful Monoid ✓

```
(List("Fun"), (List("Scala"), (List("2023"), a)))
```

Writer Monad with a Twist

```
type Writer[A] = (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

`Writer[A]` is a lawful *Monad* for **any** lawful *Monoid* on `List[String]`.

```
given Monoid[List[String]] with
    def combine(a: List[String], b: List[String]): List[String] = b ++ a
    def unit: List[String] = Nil
```

reverse order

Lawful Monoid ✓

```
(List("Fun"), (List("Scala"), (List("2023"), a)))
```

➡ `(List("2023", "Scala", “Fun"), a)`

Writer Monad with a Twist

```scala
type Writer[A] = (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

`Writer[A]` is a lawful *Monad* for **any** lawful *Monoid* on `List[String]`.

```scala
given Monoid[List[String]] with
    def combine(a: List[String], b: List[String]): List[String] = b ++ a
    def unit: List[String] = Nil
```

reverse order

Lawful Monoid ✓

```scala
(List("Fun"), (List("Scala"), (List("2023"), a)))
```

➡ `(List("2023", "Scala", “Fun"), a)`

reverse order (inner first)

Writer Monad with a Twist

```
type Writer[A] = (List[String], A)
```

Effect: writing items to a log

Sequencing: items appear in the same order as in code (outer first)

`Writer[A]` is a lawful *Monad* for **any** lawful *Monoid* on `List[String]`.

```
given Monoid[List[String]] with
    def combine(a: List[String], b: List[String]): List[String] = b ++ a
    def unit: List[String] = Nil
```

reverse order

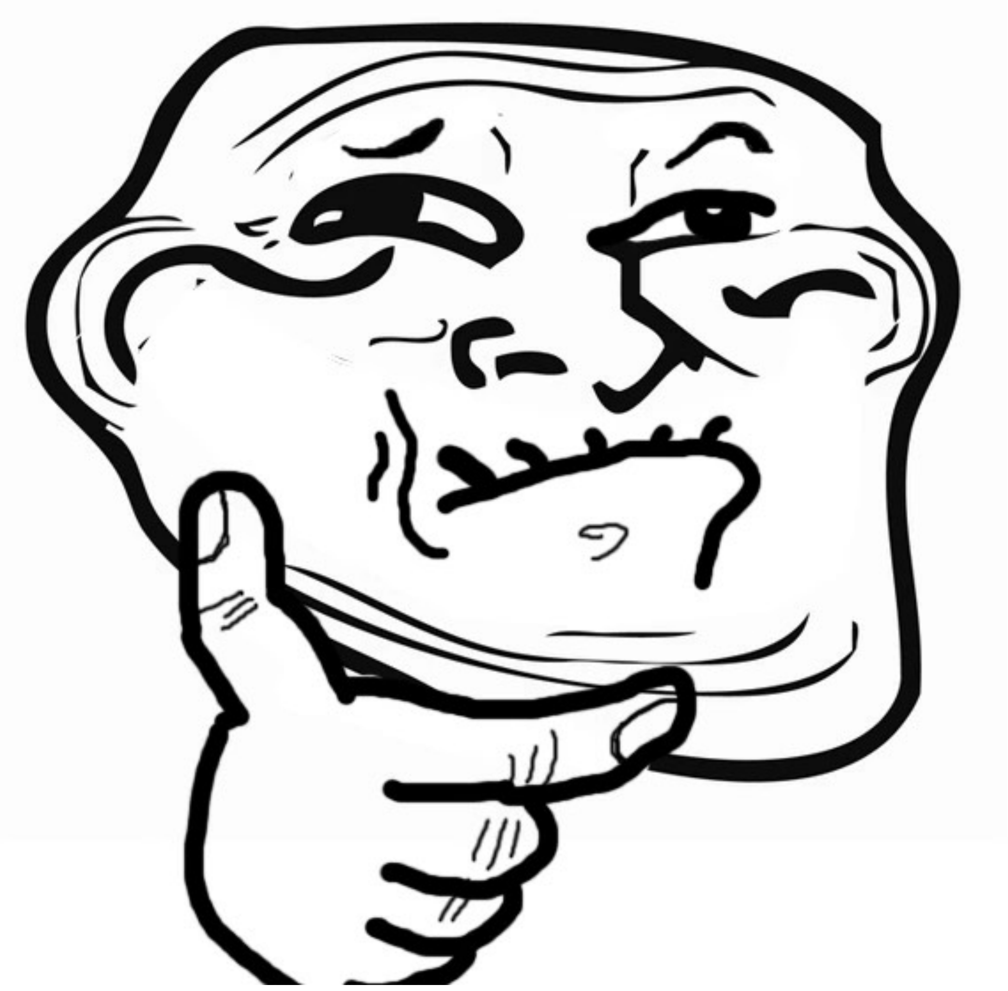
Lawful Monoid ✓

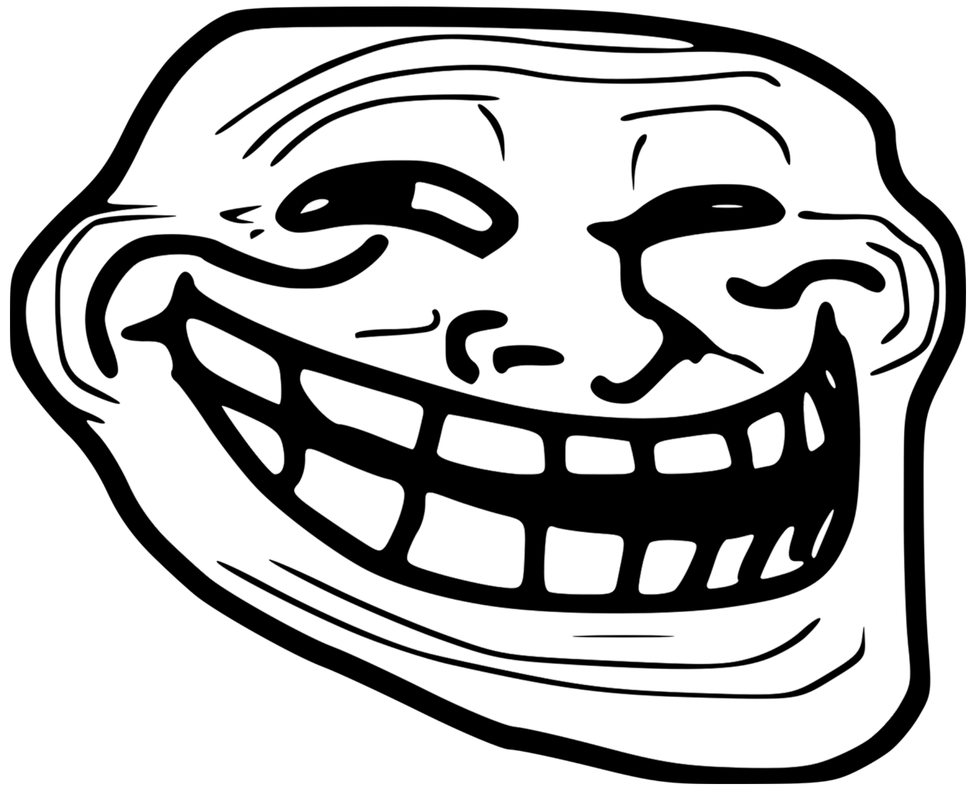
```
(List("Fun"), (List("Scala"), (List("2023"), a)))
```

➡ `(List("2023", "Scala", “Fun"), a)`

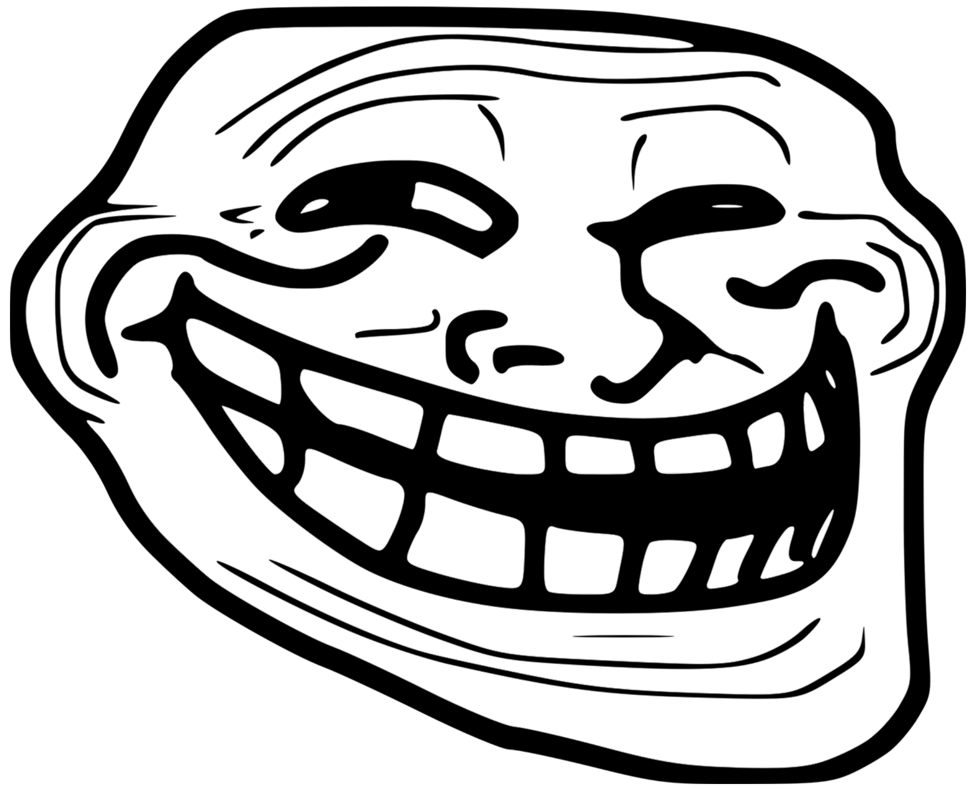
reverse order (inner first)

[… REINTERPRETING …]



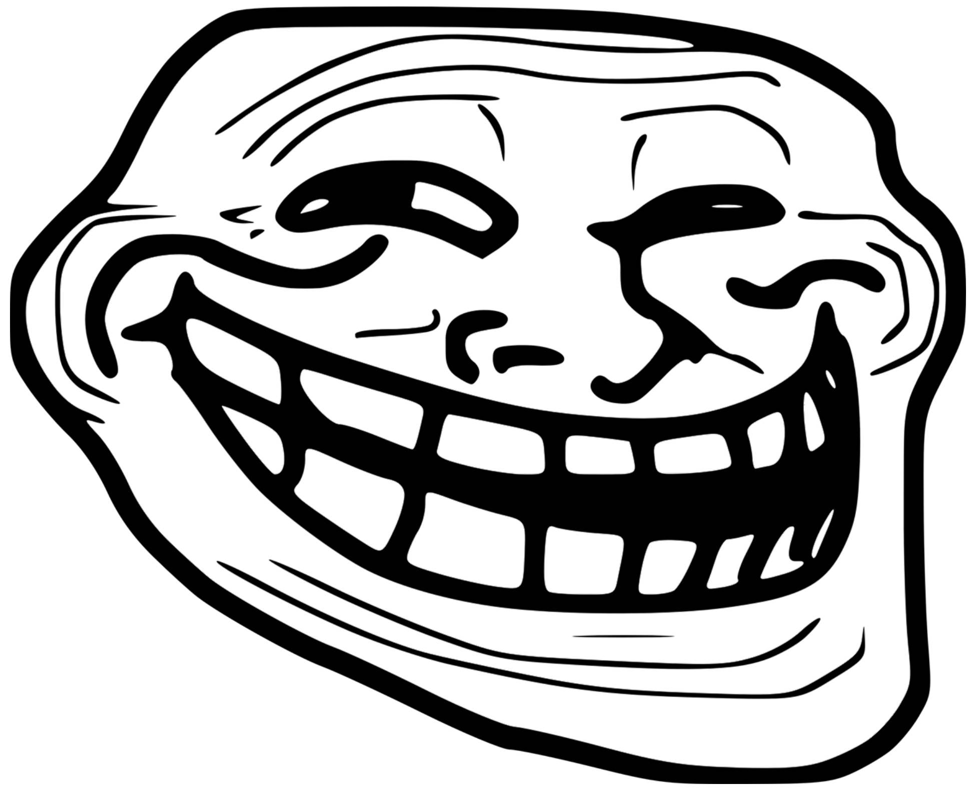


“The effect of **that** monad is **prepending** to the log.”



“The effect of **that** monad is **prepending** to the log.”

“Monads are all about sequencing!”



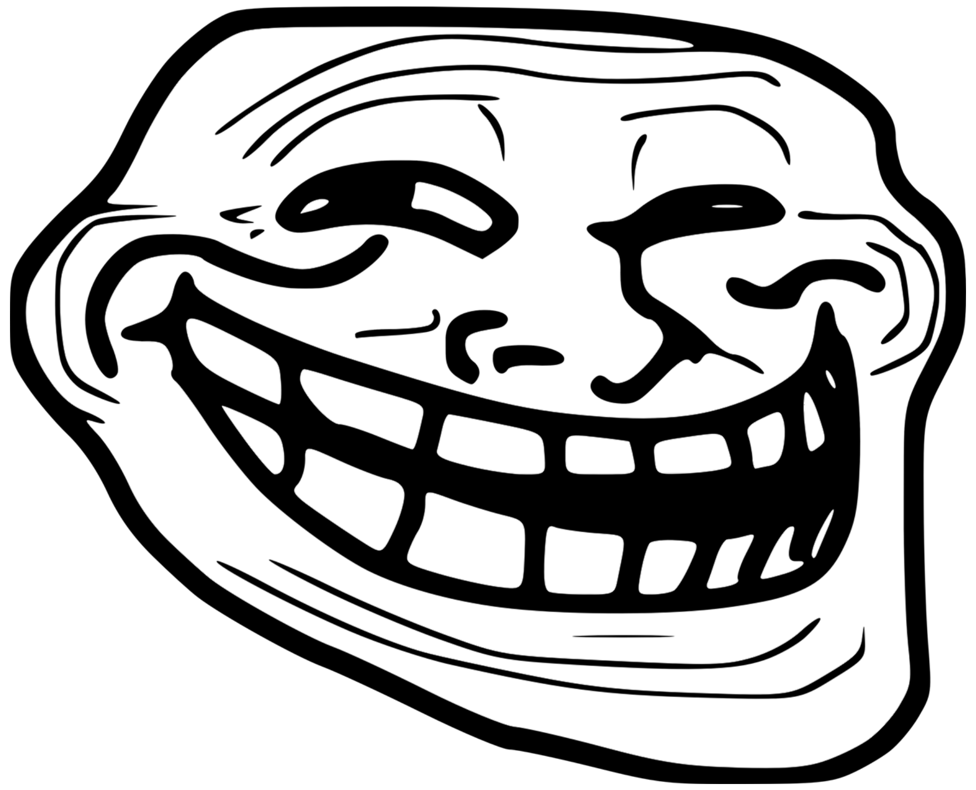
“The effect of **that** monad is **prepending** to the log.”

“Monads are all about sequencing!”

“The effect of **that** monad is **prepending** to the log.”

“Monads are all about sequencing!”

OK, so

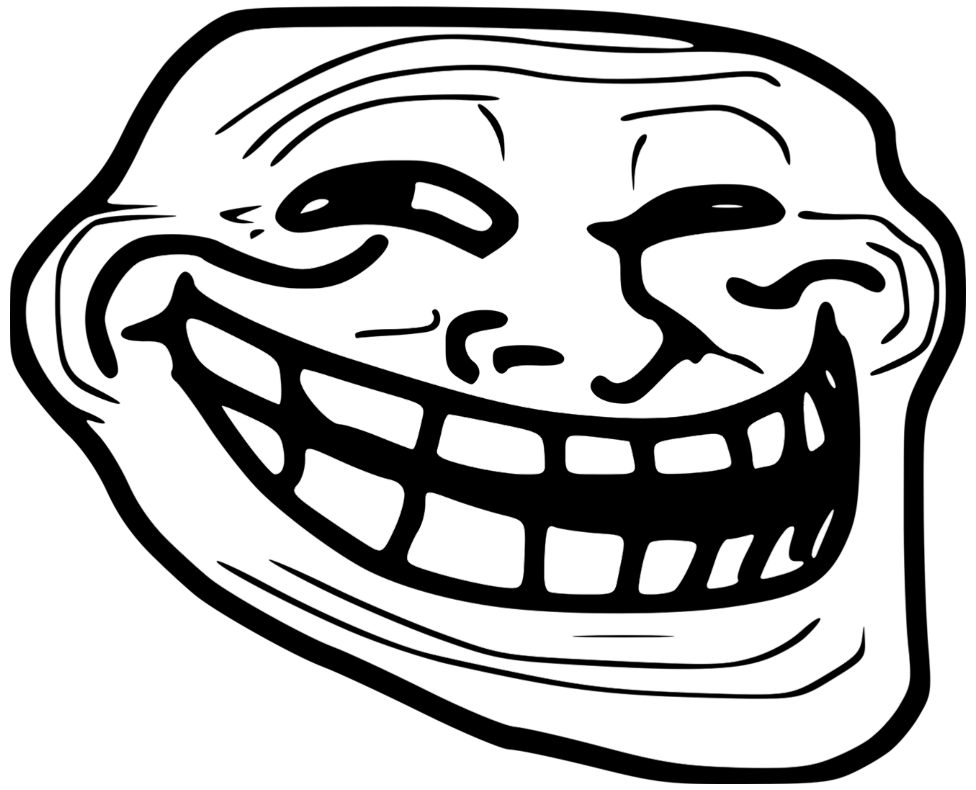


“The effect of **that** monad is **prepending** to the log.”

“Monads are all about sequencing!”

OK, so

effect = monad



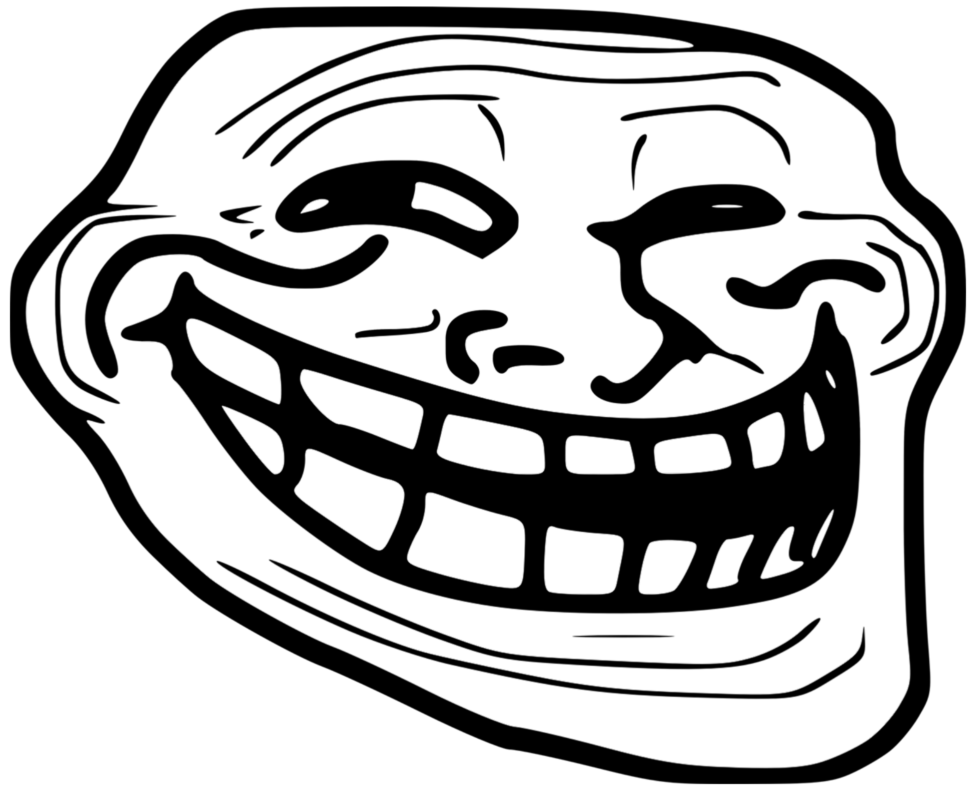
“The effect of **that** monad is **prepending** to the log.”

“Monads are all about sequencing!”

OK, so

effect = monad

sequencing = whatever **`flatten`** does



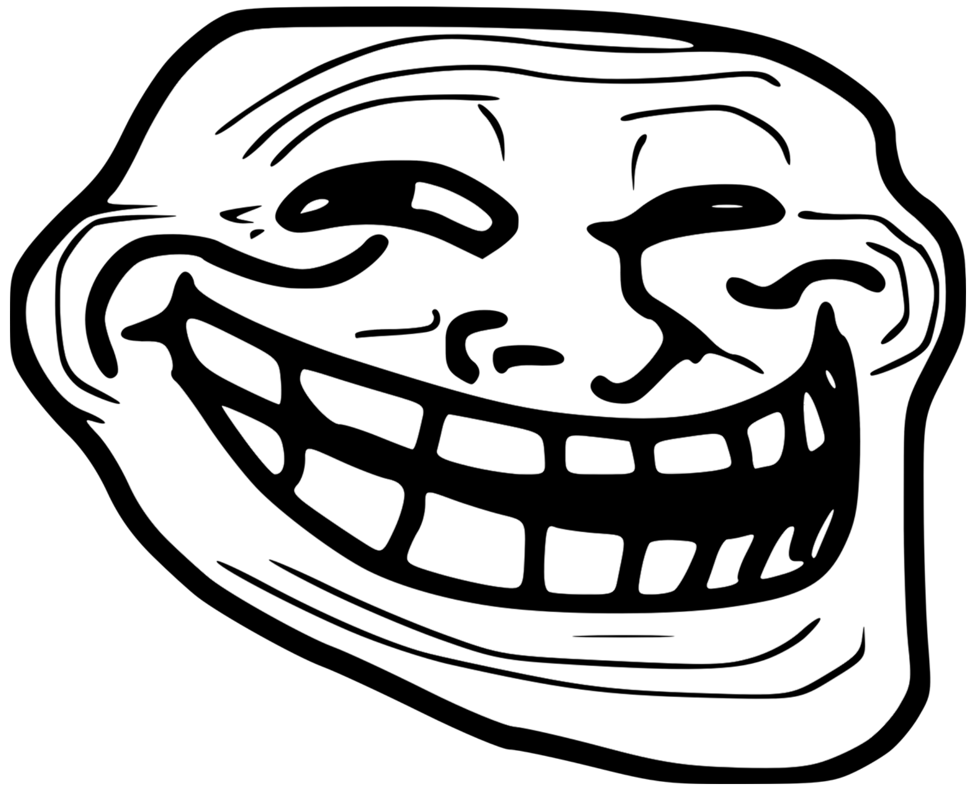
“The effect of **that** monad is **prepending** to the log.”

“Monads are all about sequencing!”

OK, so

effect	=	monad
sequencing	=	whatever **`flatten`** does

?



“The effect of **that** monad is **prepending** to the log.”

“Monads are all about sequencing!”

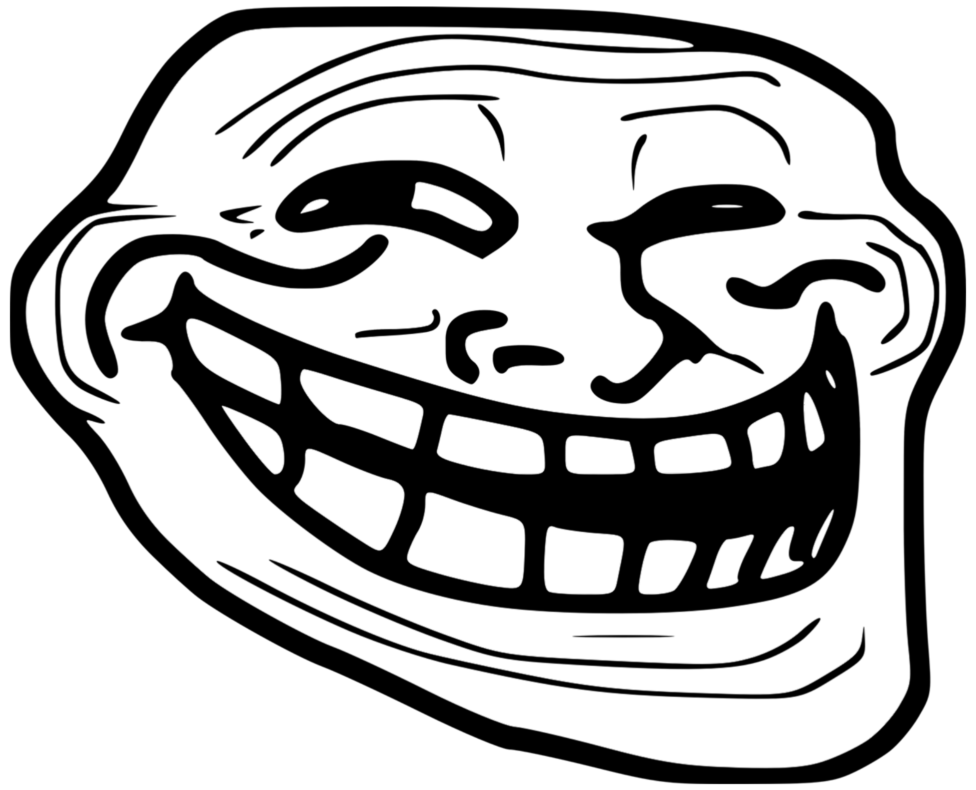
OK, so

effect = monad

sequencing = whatever **`flatten`** does

?

✓ Bulletproof



“The effect of **that** monad is **prepending** to the log.”

“Monads are all about sequencing!”

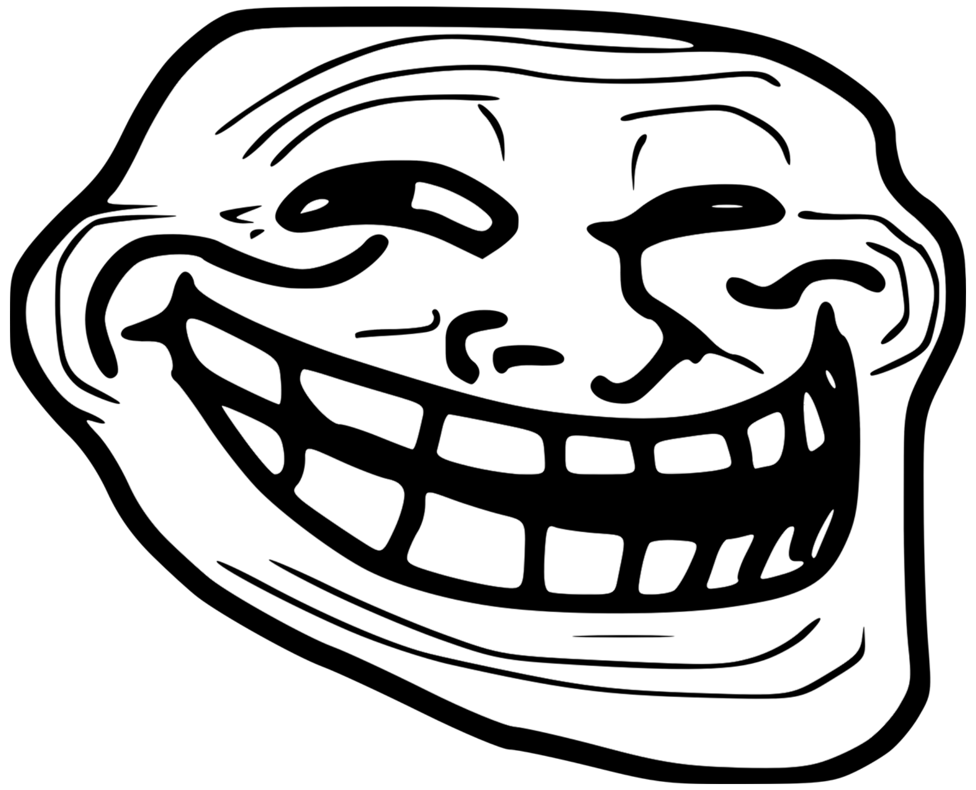
OK, so

effect	=	monad
sequencing	=	whatever **`flatten`** does

?

✓ **Bulletproof**

✓ **Tautological**



“The effect of **that** monad is **prepending** to the log.”

“Monads are all about sequencing!”

OK, so

effect = monad

sequencing = whatever **`flatten`** does

?

✓ Bulletproof

✓ Tautological

✓ Useless

Lessons So Far

- Monads definable in **any** Category (even non-executable one, like `<:<`)
- **Syntactically,** monads *do* support **sequential composition**
- Sequential composition ≠ sequential **execution** (e.g. monads in **<=**)

Lessons So Far

- Monads definable in **any** Category (even non-executable one, like **<:<**)
- **Syntactically,** monads *do* support **sequential composition**
- Sequential composition ≠ sequential **execution** (e.g. monads in **<=**)
- *“Sequencing of effects”* is **vague,** definable only **tautologically**

IT IS SEQUENCING!!!

(the behavior of twisted `Writer`)

IT IS SEQUENCING!!!

(the behavior of twisted `Writer`)

Just say NO! 🙅‍♂️

Don’t feed the trolls.

IT IS SEQUENCING!!!

(the behavior of twisted `Writer`)

Just say NO! 🙅‍♂️

Don’t feed the trolls.

Accept & Continue

IT IS SEQUENCING!!!

(the behavior of twisted `Writer`)

Just say NO! 🙅‍♂️

Don't feed the trolls.

Accept & Continue

Up Next:
Concurrent, Non-deterministic `Writer`

Up Next:
Concurrent, Non-deterministic `Writer`

But first, a quick introduction to

Libretto,

a *concurrent-by-default* DSL embedded in Scala.

Libretto for Scala Programmers

Scala	Libretto	

Libretto for Scala Programmers

Scala	Libretto	
`(A, B)`	`A ⊗ B`	*Concurrent* Pair

Libretto for Scala Programmers

Scala	**Libretto**	
`(A, B)`	`A ⊗ B`	*Concurrent* Pair
`Either[A, B]`	`A ⊕ B`	Meaning closer to `Future[Either[A, B]]`

Libretto for Scala Programmers

Scala	Libretto	
`(A, B)`	`A ⊗ B`	*Concurrent* Pair
`Either[A, B]`	`A ⊕ B`	Meaning closer to `Future[Either[A, B]]`
`Unit`	`One`	

Libretto for Scala Programmers

Scala	Libretto	
`(A, B)`	`A ⊗ B`	*Concurrent* Pair
`Either[A, B]`	`A ⊕ B`	Meaning closer to `Future[Either[A, B]]`
`Unit`	`One`	
`A => B`	`A -o B`	Functions in Libretto are *linear.*

Libretto for Scala Programmers

Scala	Libretto	
`(A, B)`	`A ⊗ B`	*Concurrent* Pair
`Either[A, B]`	`A ⊕ B`	Meaning closer to `Future[Either[A, B]]`
`Unit`	`One`	
`A => B`	`A -○ B`	Functions in Libretto are *linear.*
`Promise[A]`	`-[A]`	Cannot be ignored. Cannot be completed twice.

`List` in Scala vs. Libretto

Scala

```
type List[A] = Either[Unit, (A, List[A])]
```

Libretto

```
type List[A] = One ⊕ (A ⊗ List[A])
```

Libretto `List` : Intuition

Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction

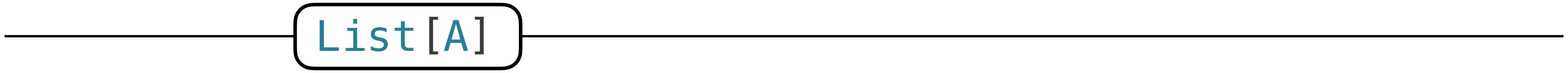
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction

`List[A]`

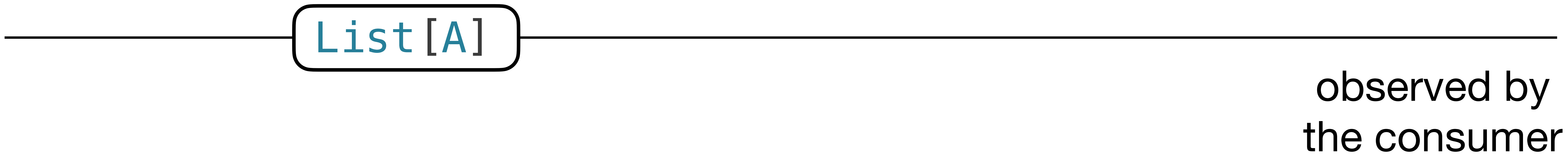
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction



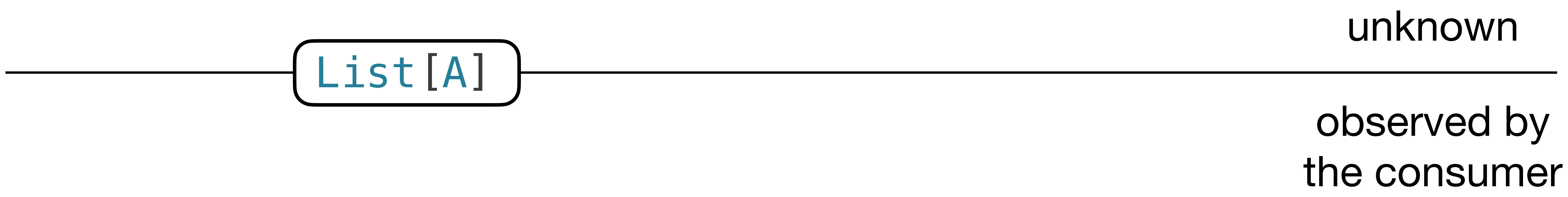
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction



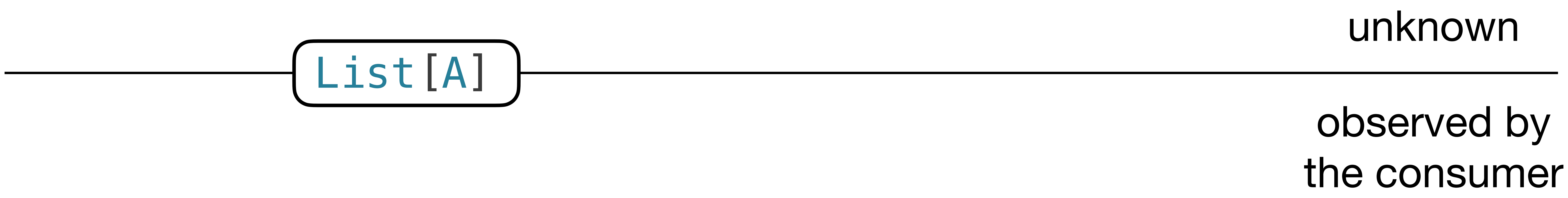
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction



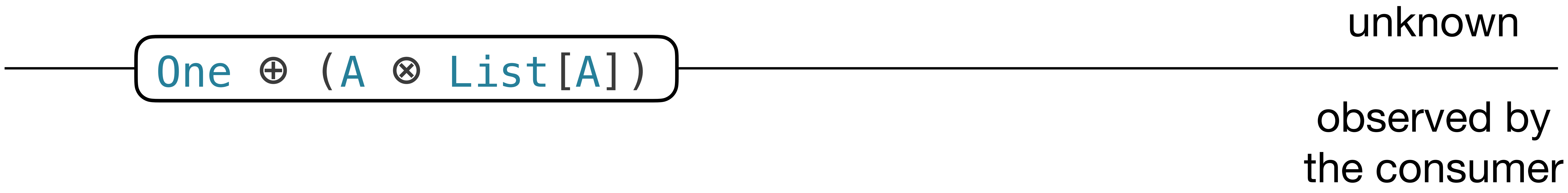
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**



Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**

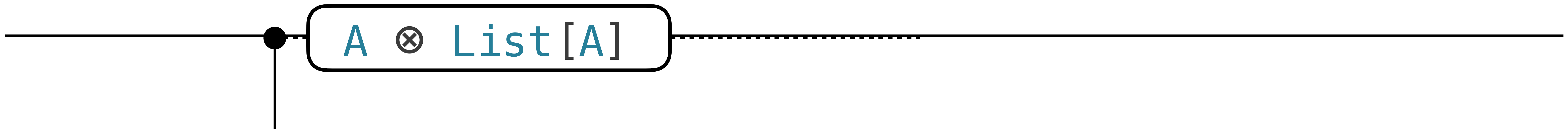


Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**

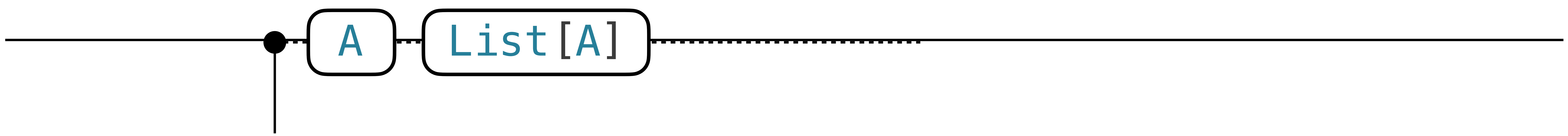
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**



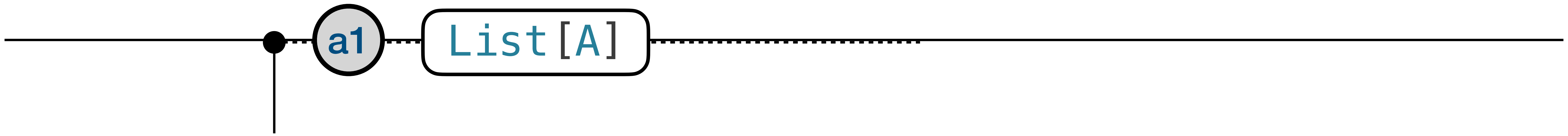
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**



Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**



Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**

Libretto `List` : Intuition

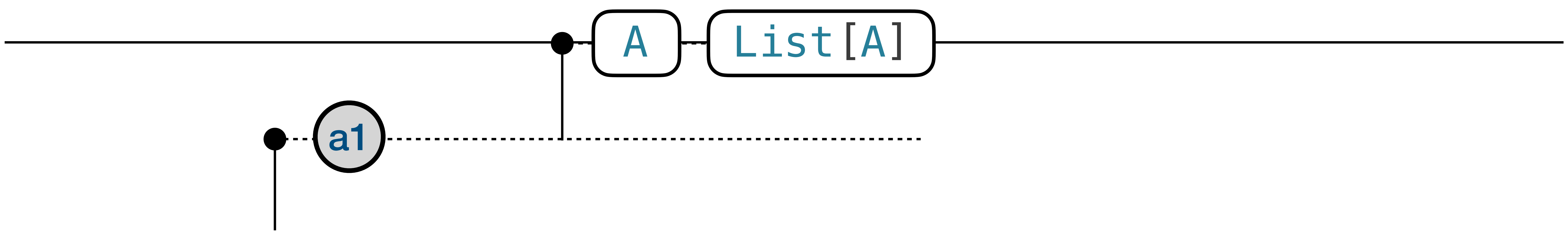
- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**

Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**

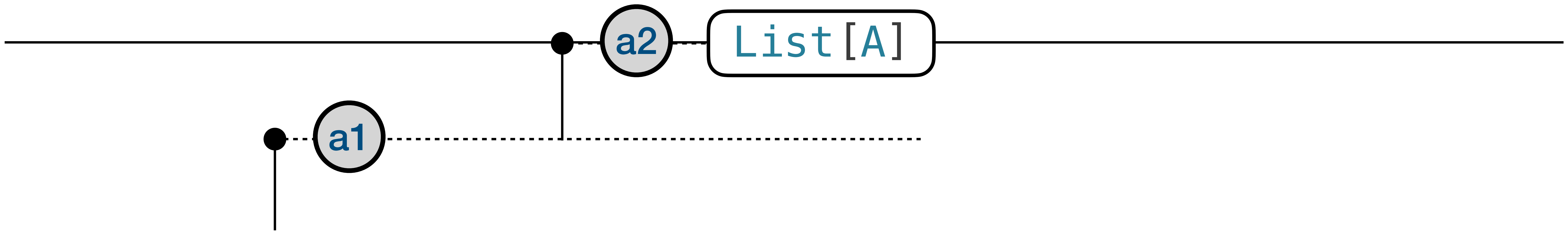
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**



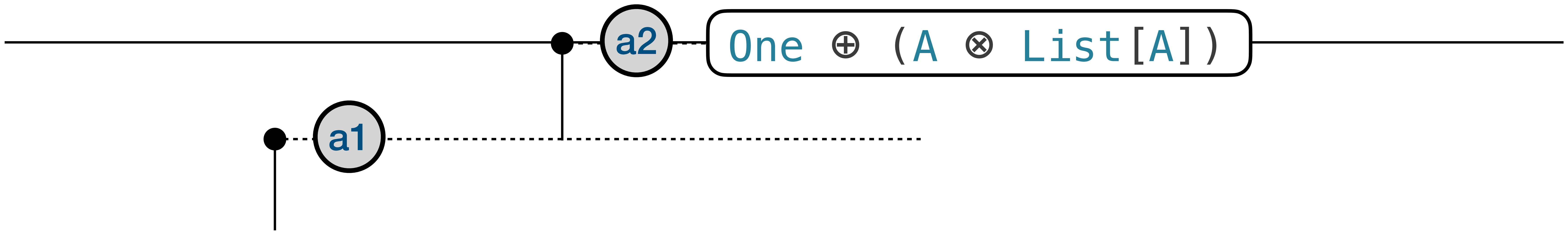
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**



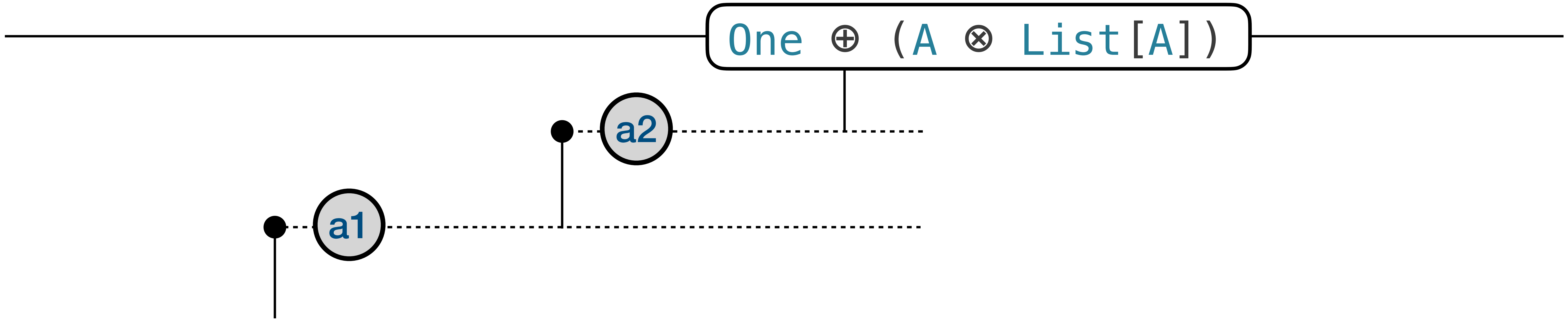
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**



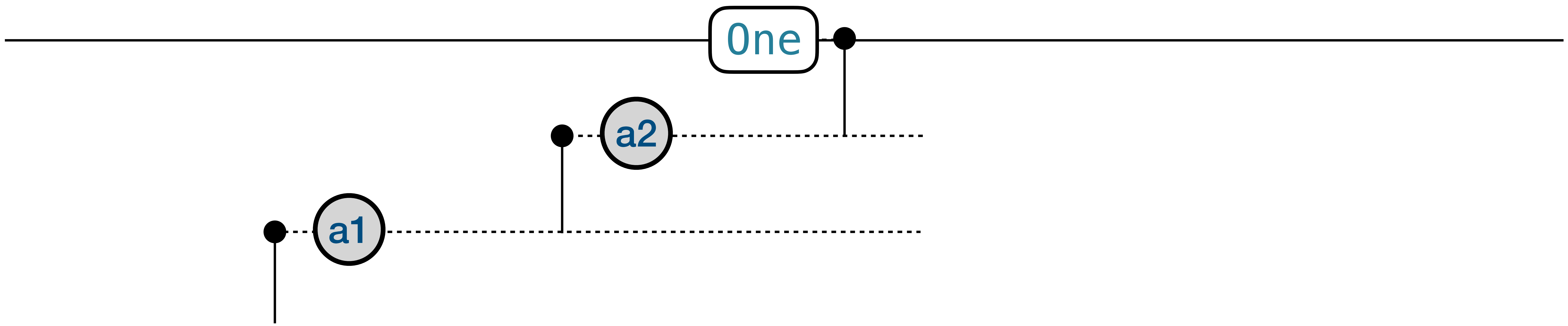
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**



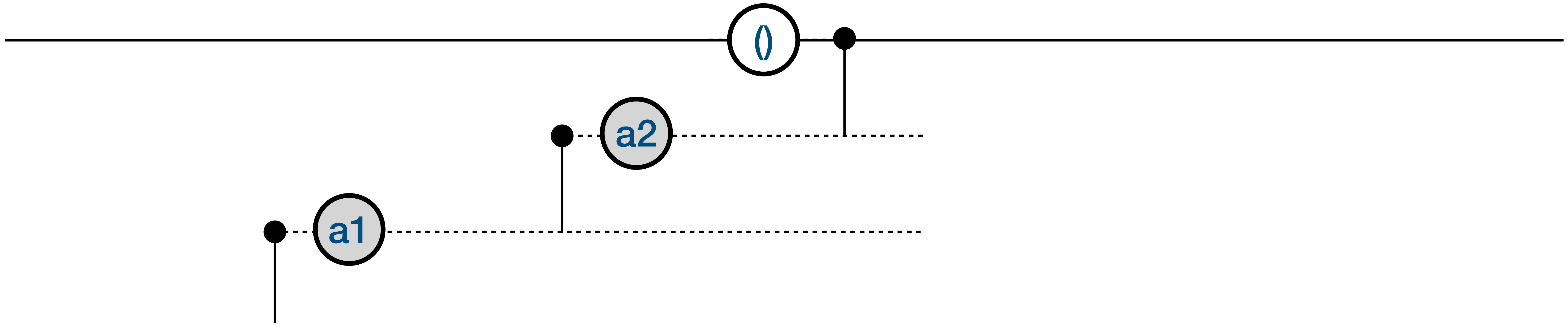
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**



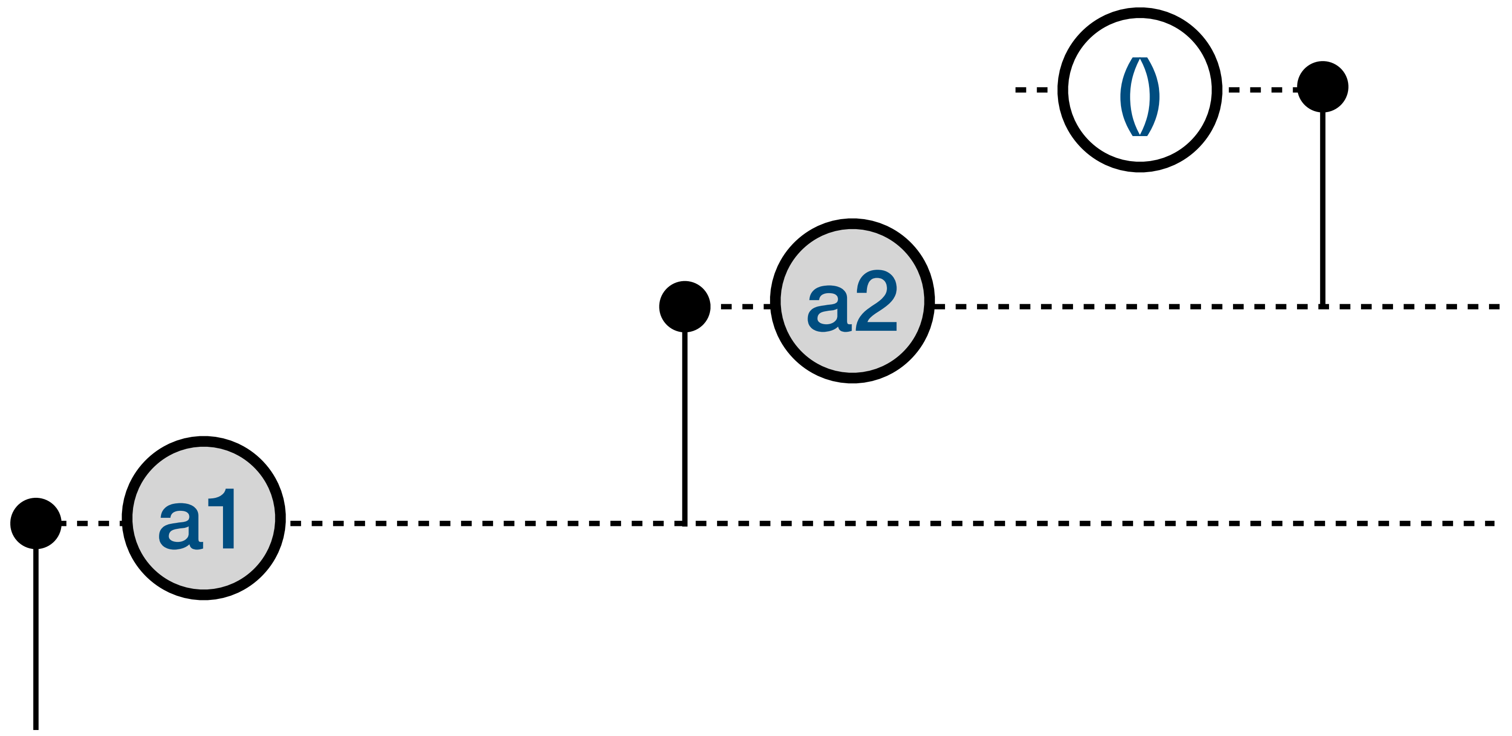
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**



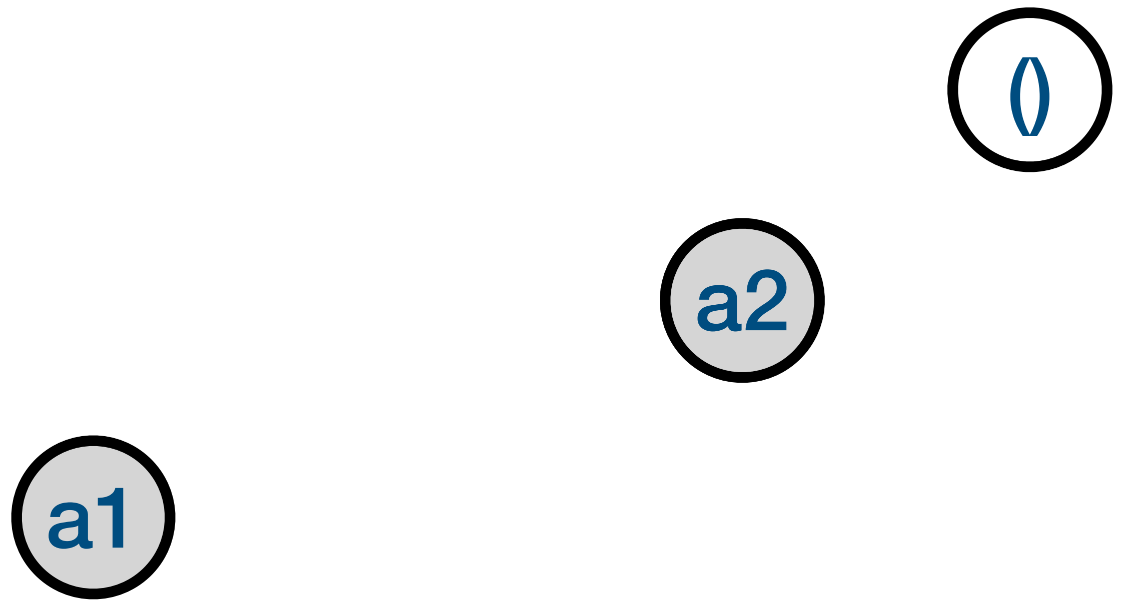
Libretto `List` : Intuition

- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**

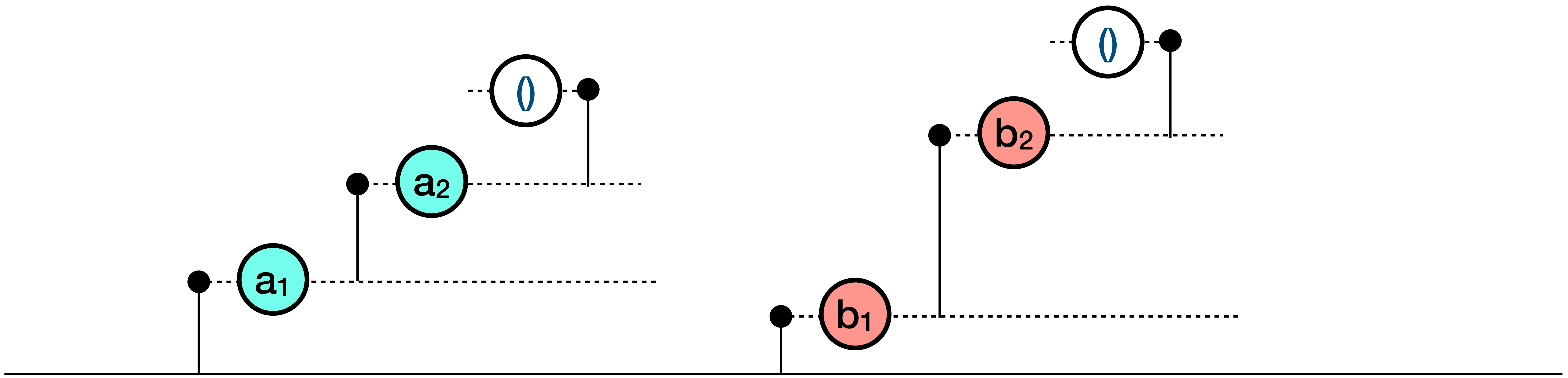


Libretto `List` : Intuition

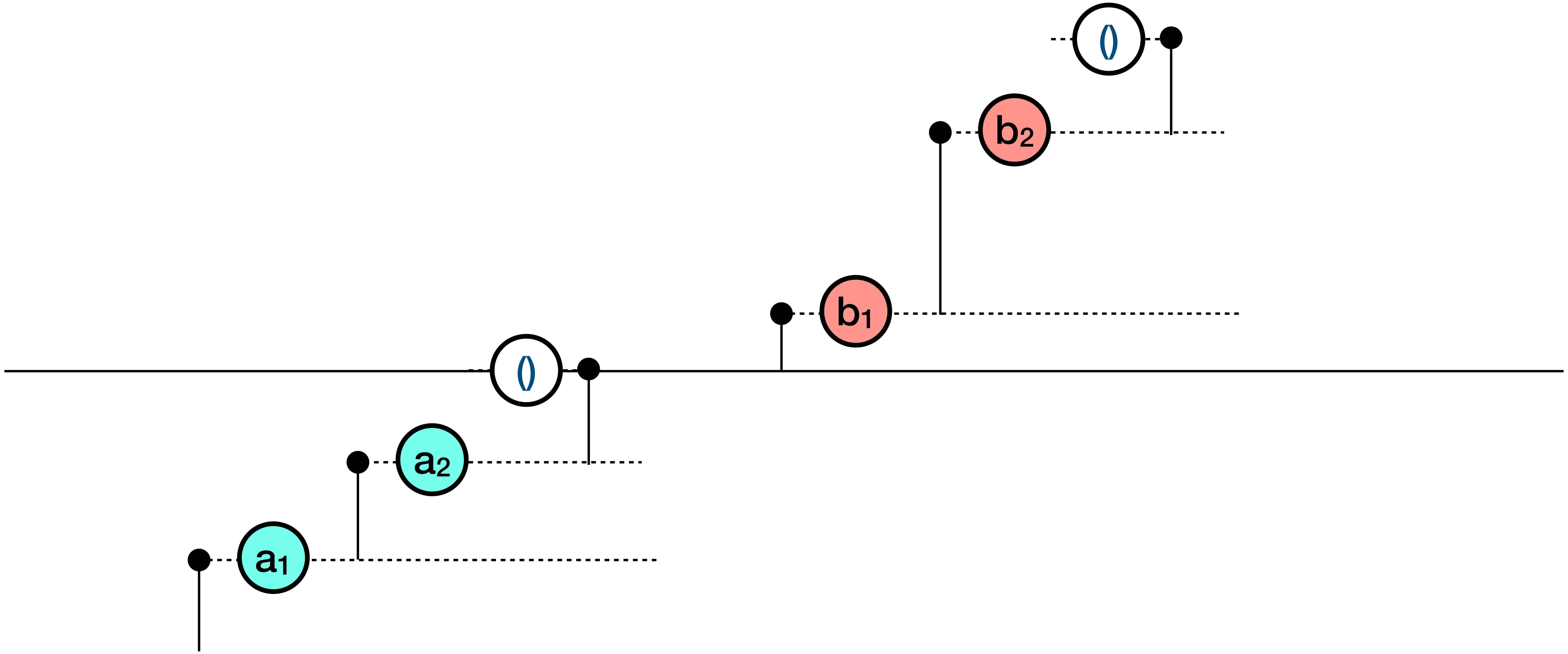
- Types in Libretto describe interfaces of interaction
- `List` is produced (and observed) **gradually**



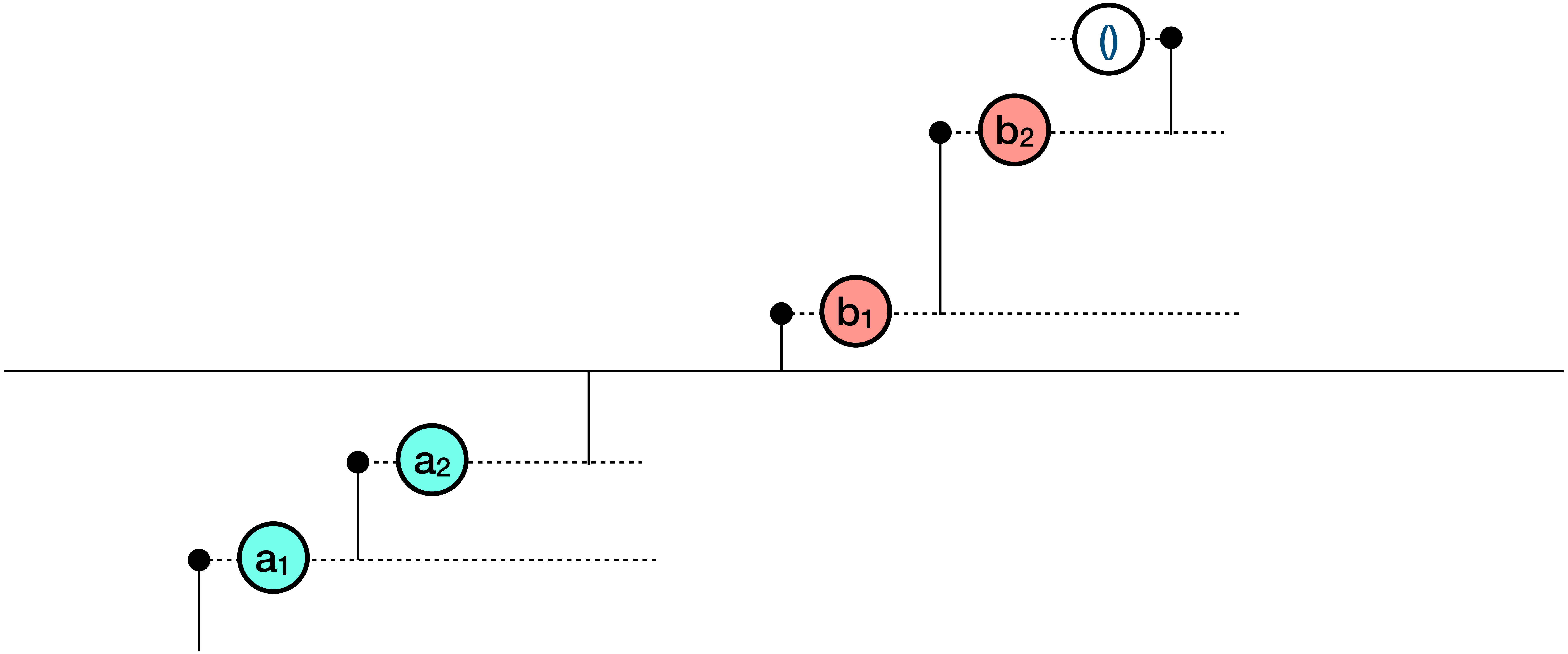
Libretto `List:concat`



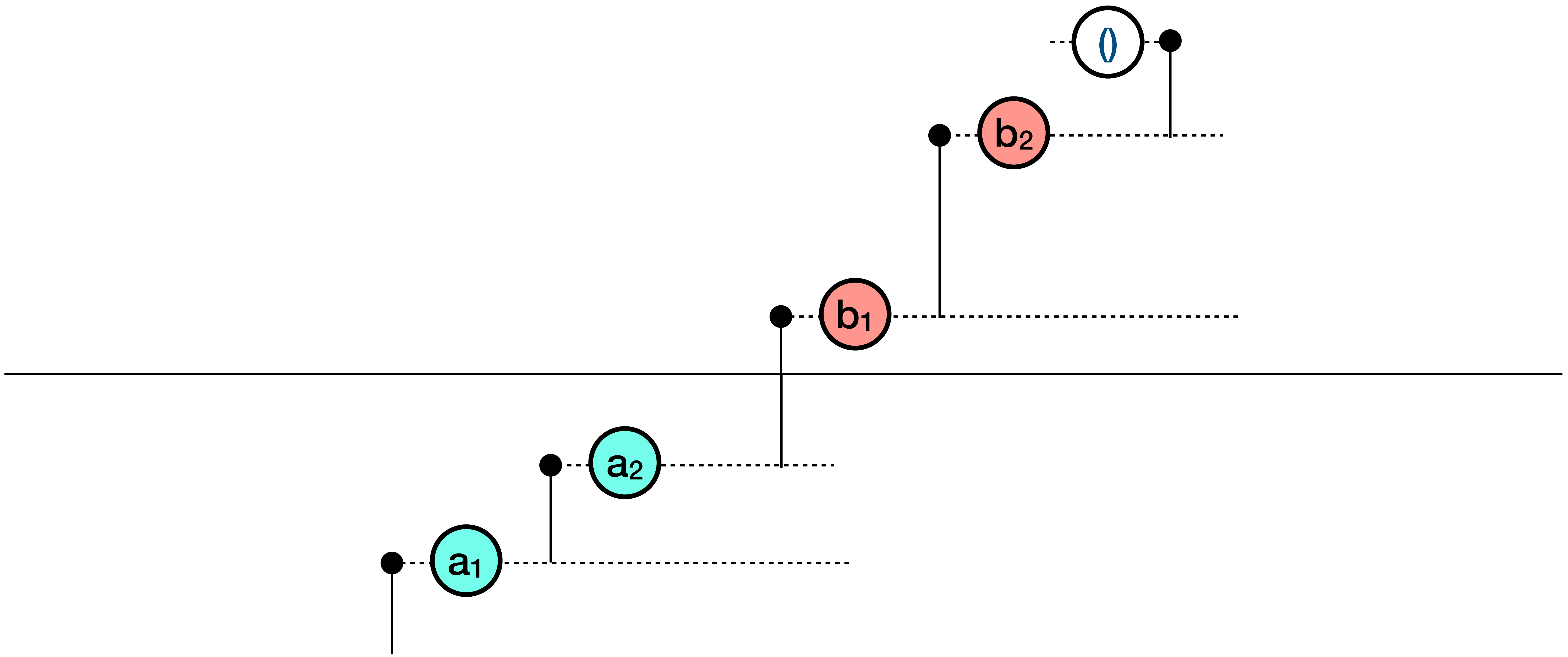
Libretto `List:concat`



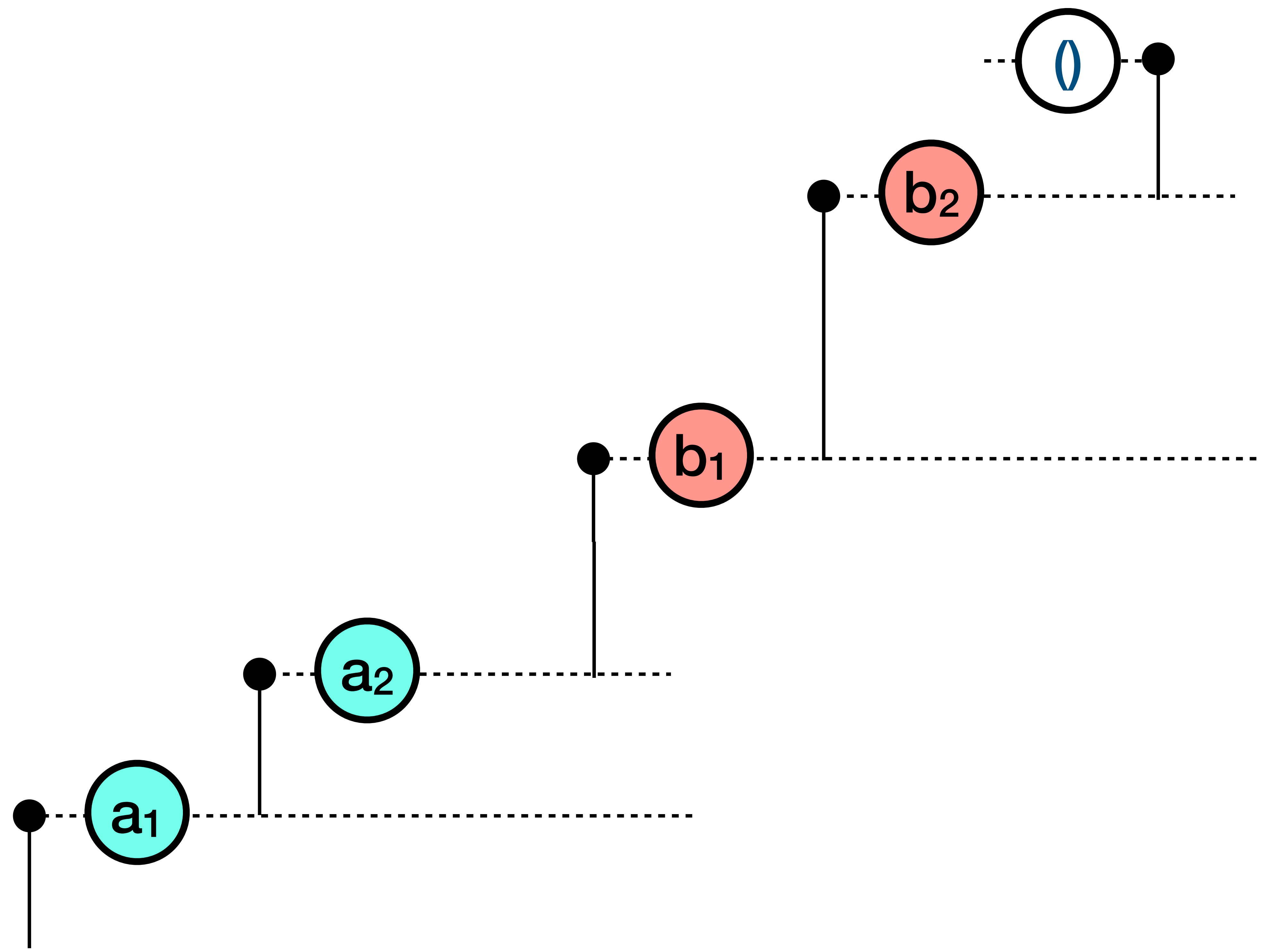
Libretto `List`: `concat`



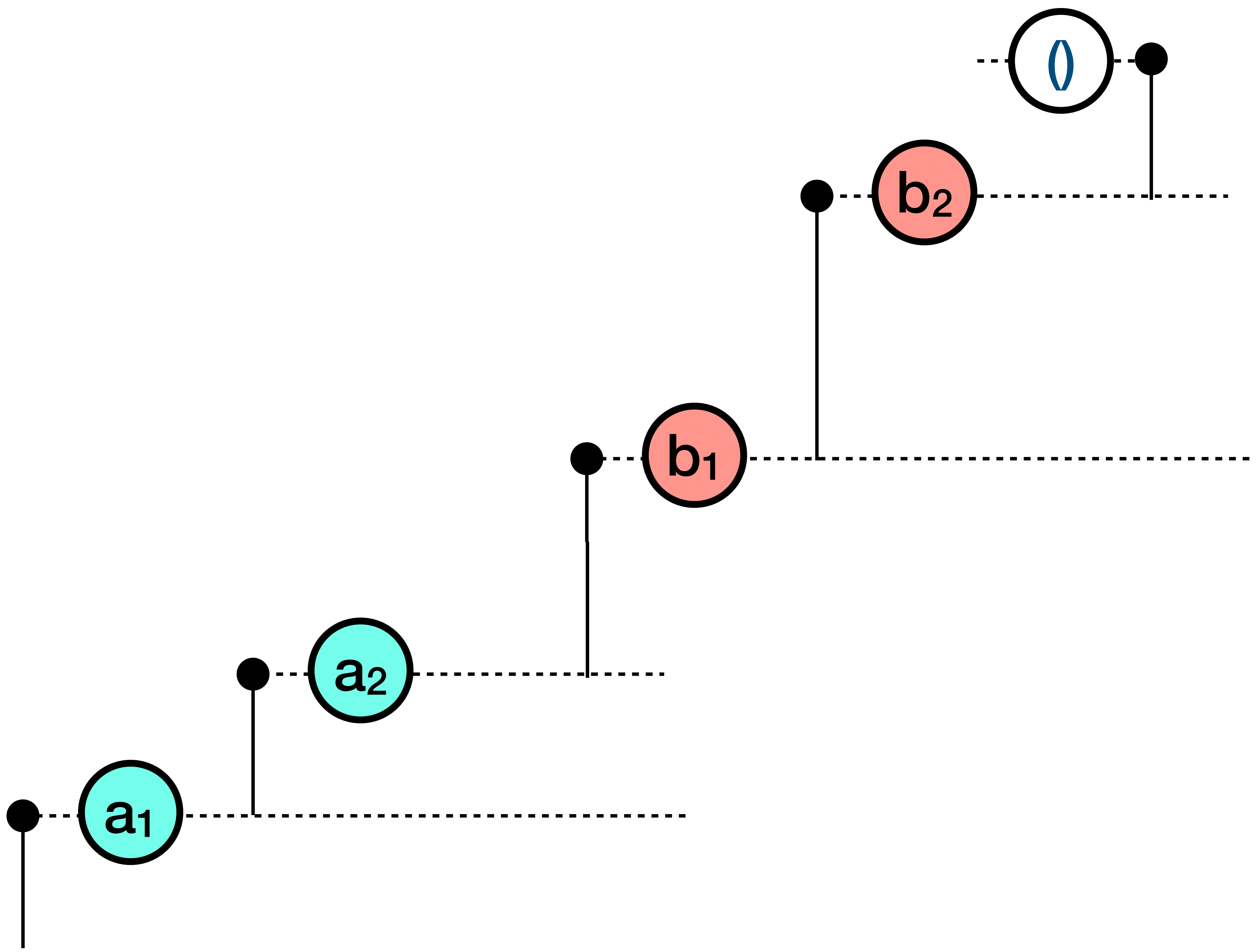
Libretto `List`:`concat`



Libretto `List`: `concat`

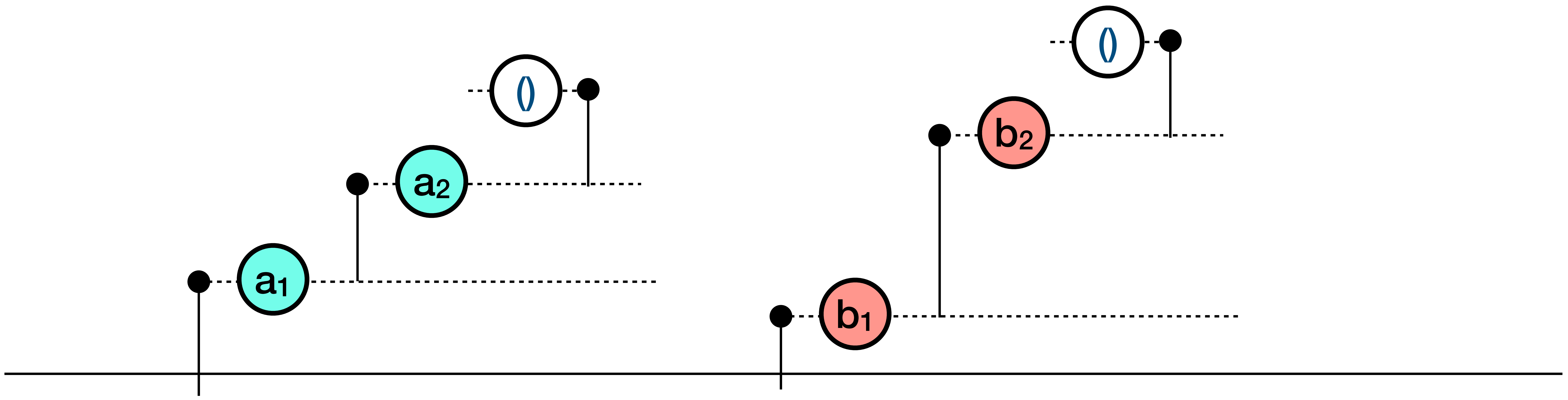


Libretto `List`: `concat`

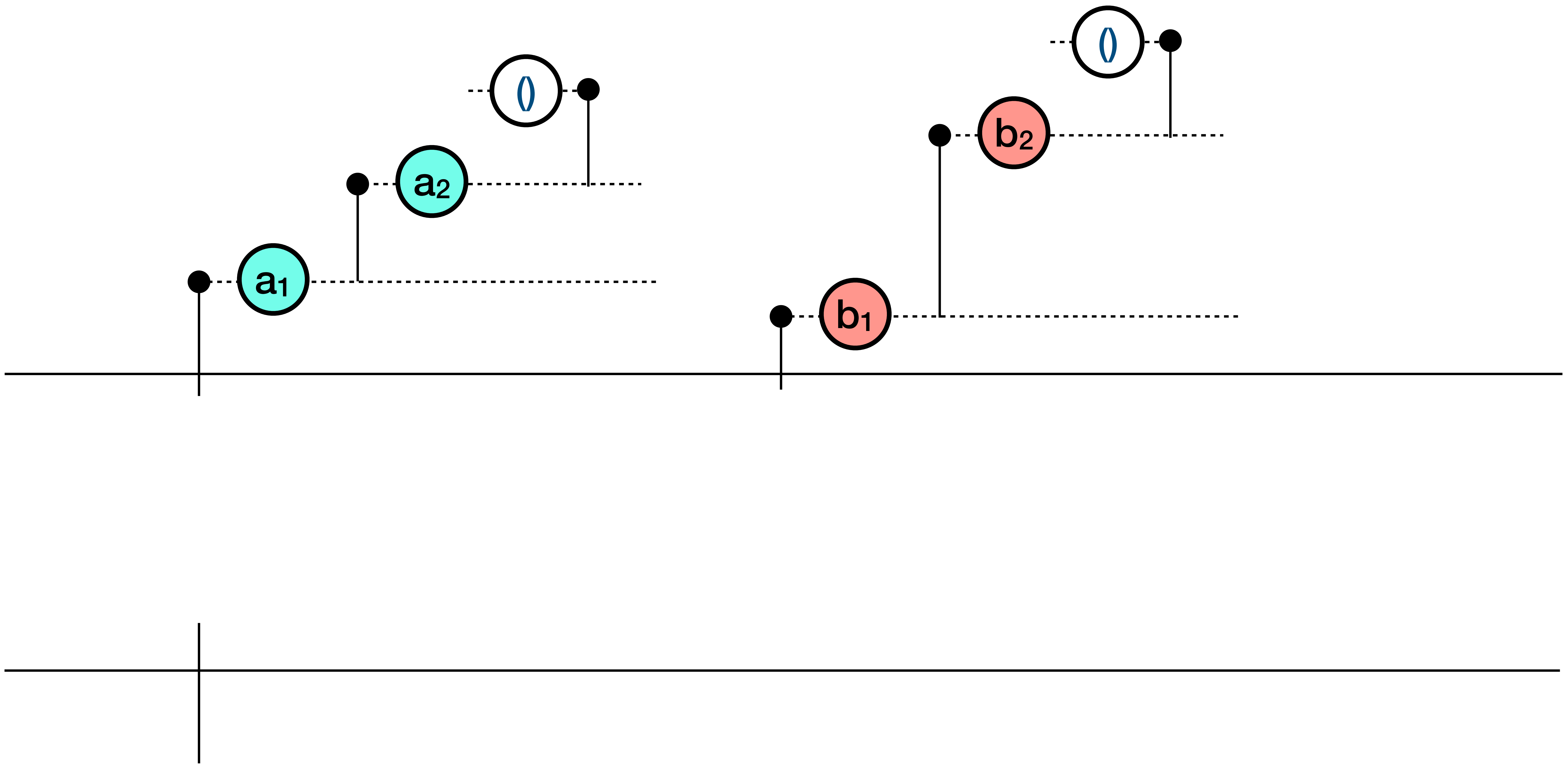


Forms the usual Monoid on `List`

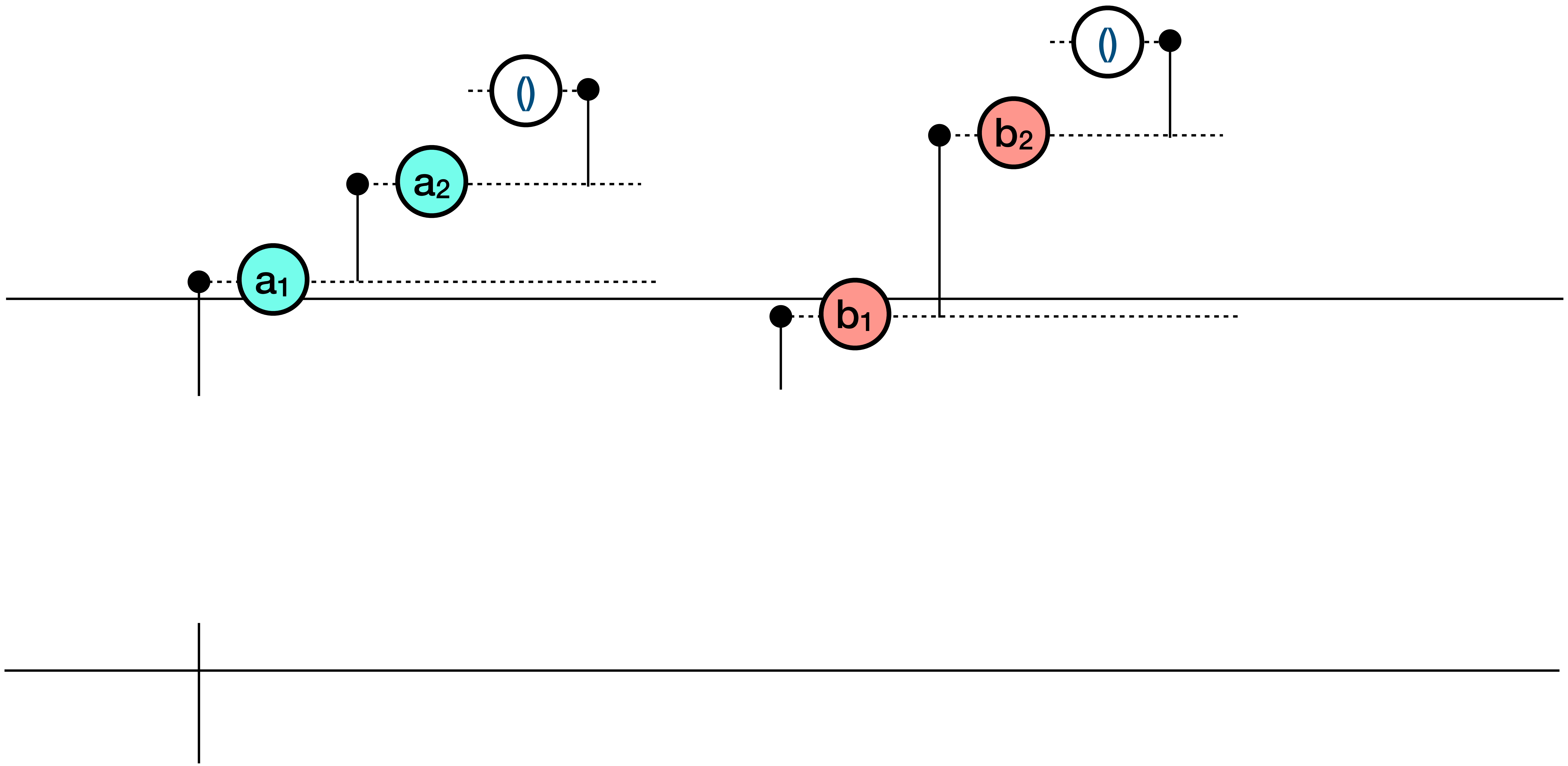
Libretto `List:merge`



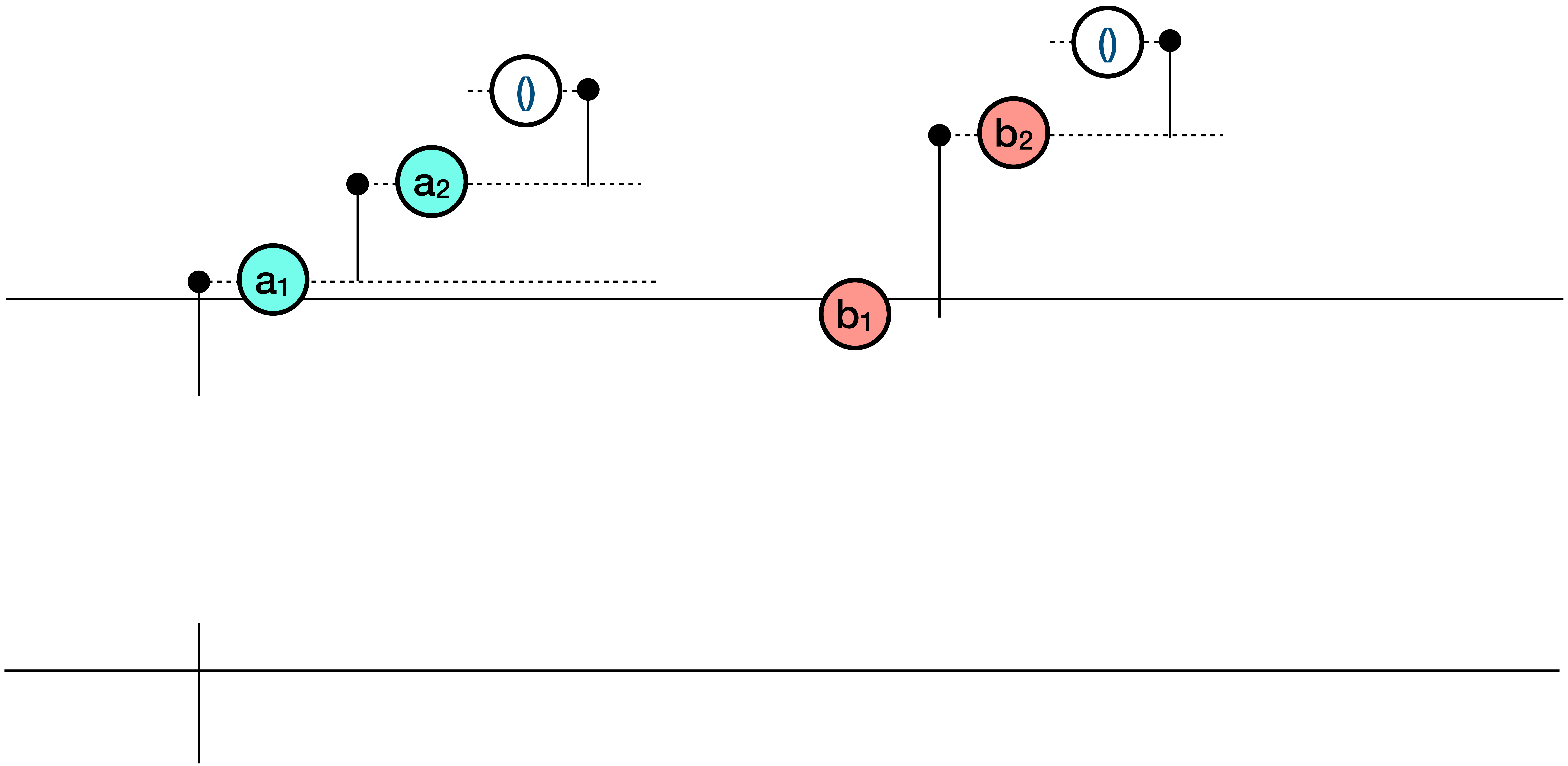
Libretto `List:merge`



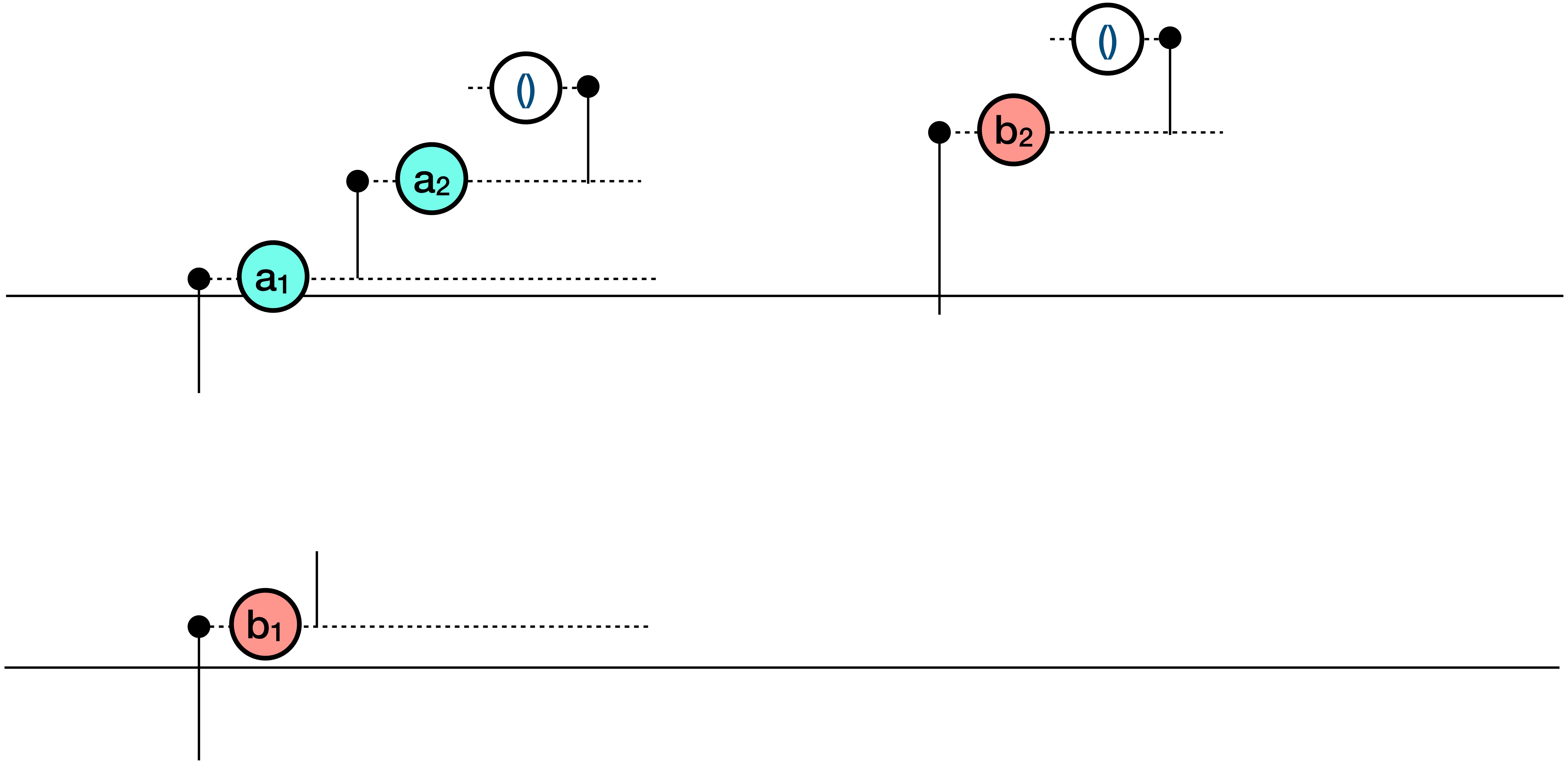
Libretto `List`:`merge`



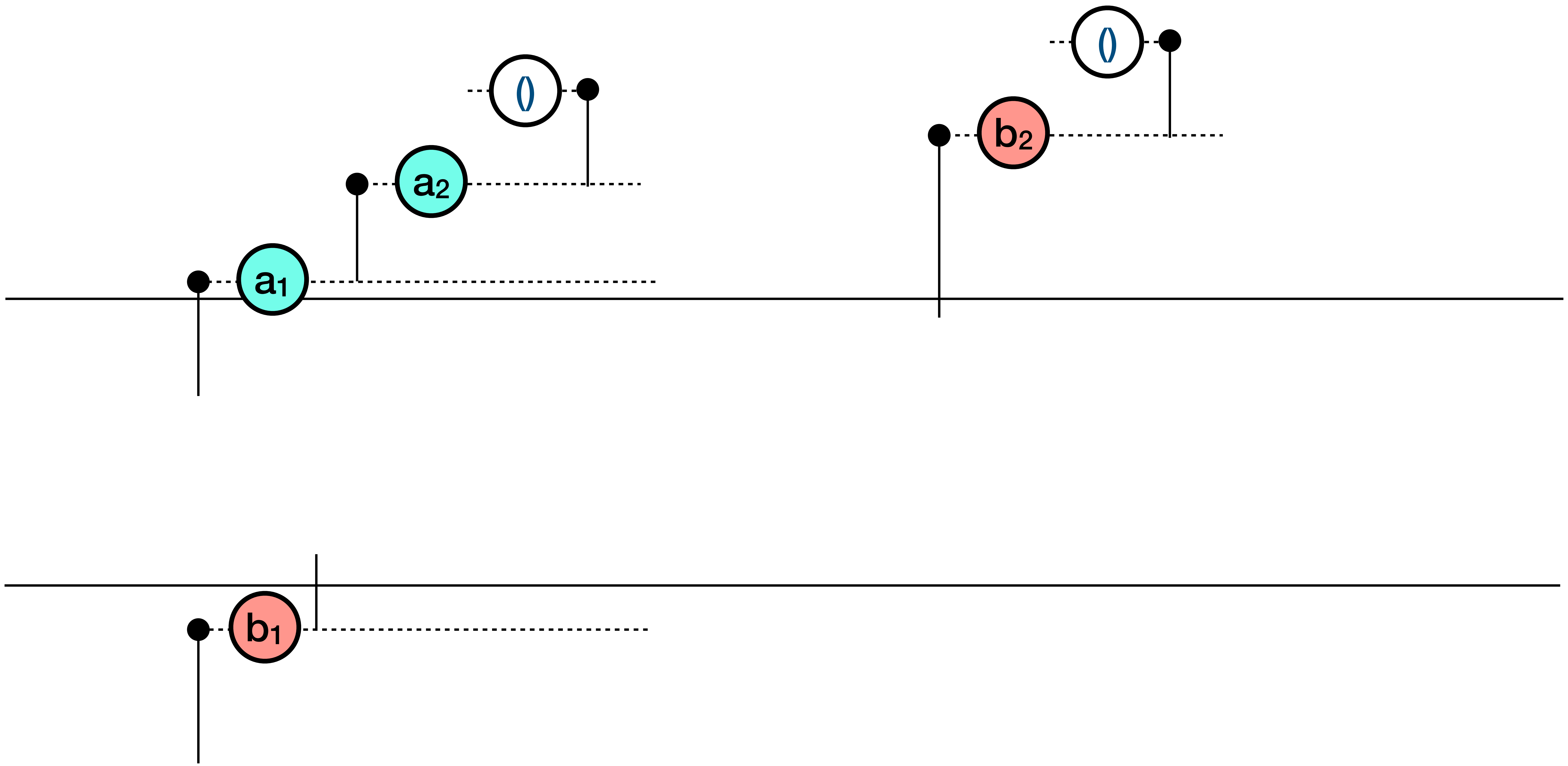
Libretto `List:merge`



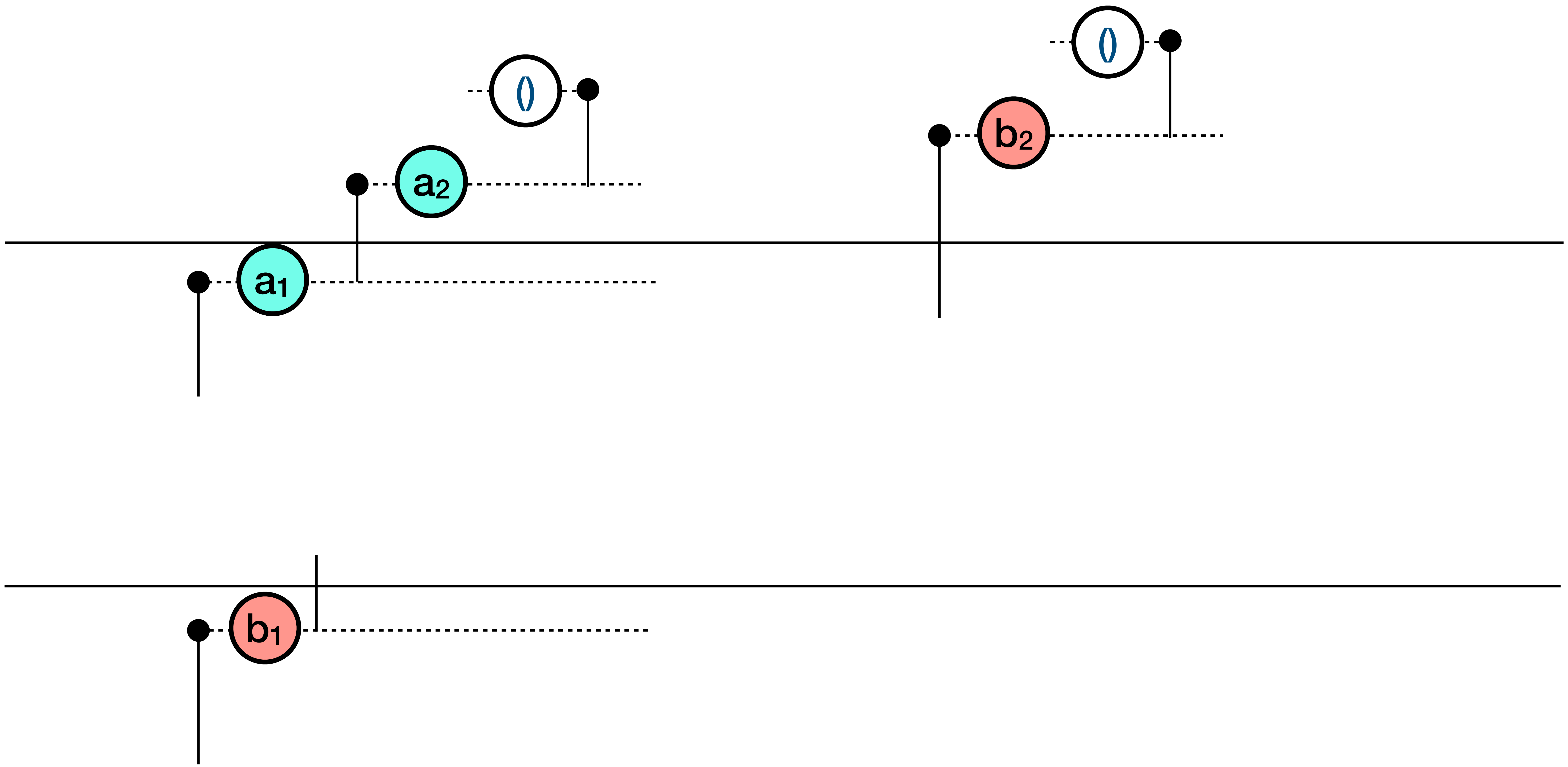
Libretto `List:merge`



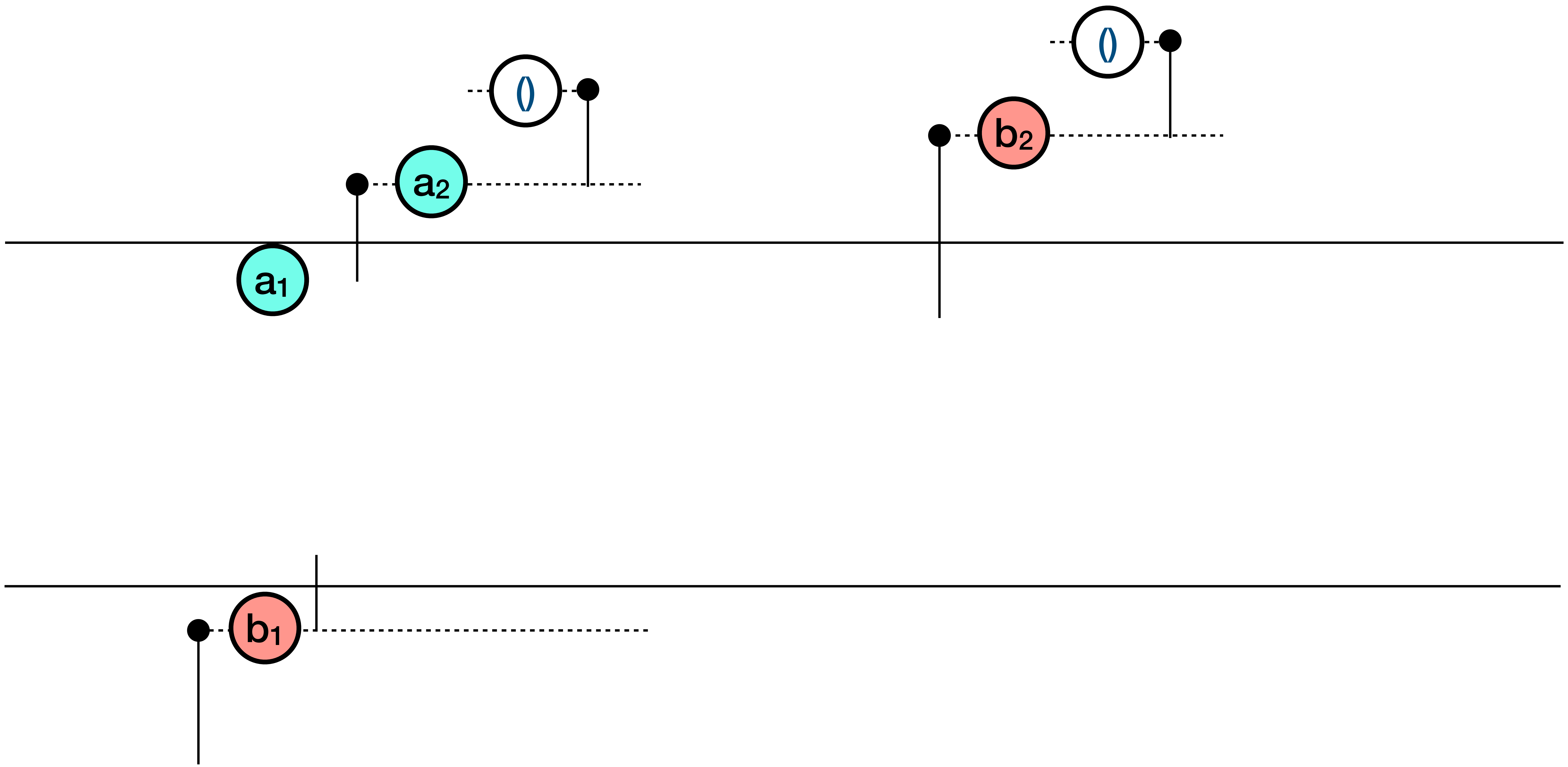
Libretto `List:merge`



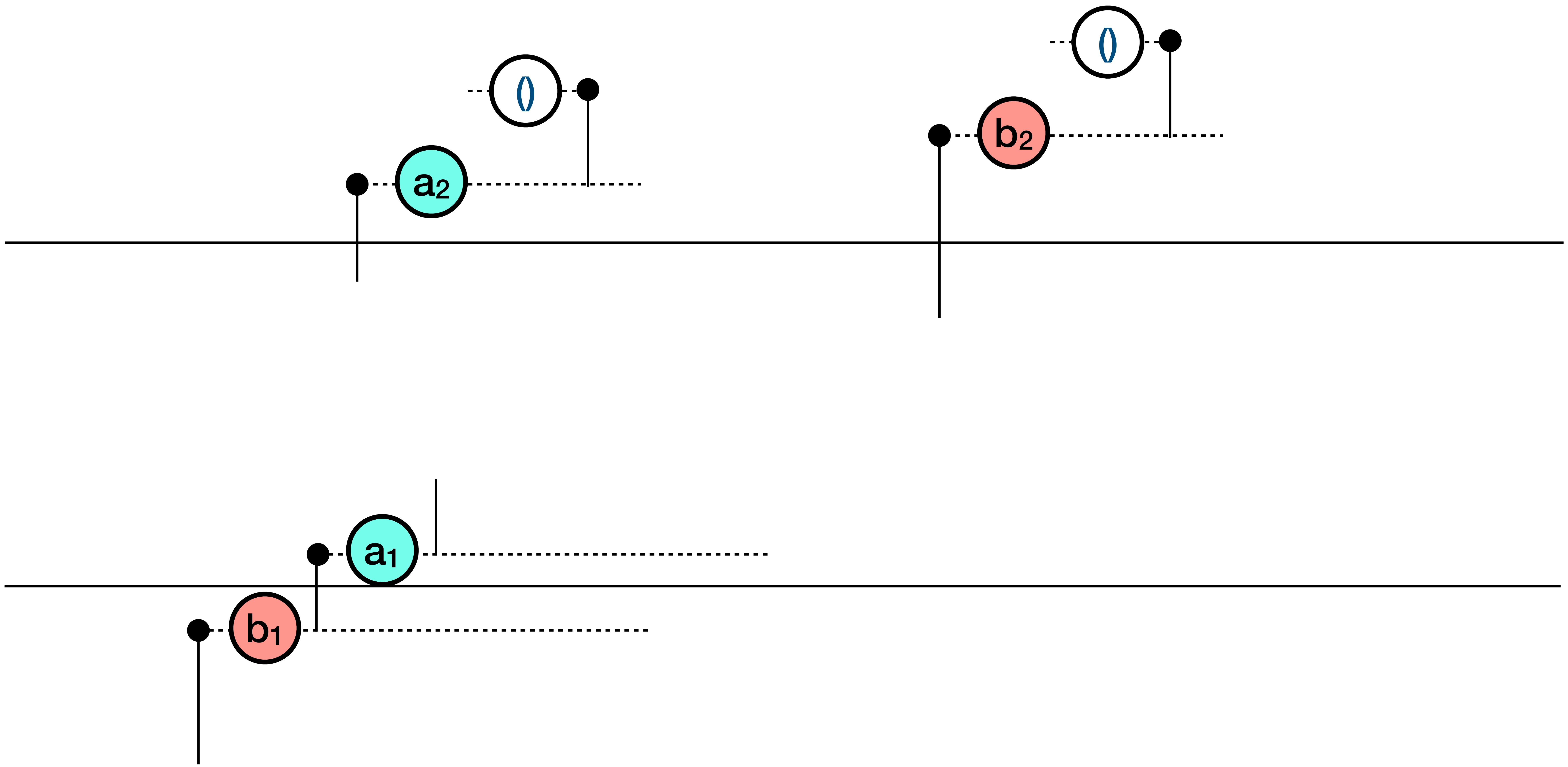
Libretto `List:merge`



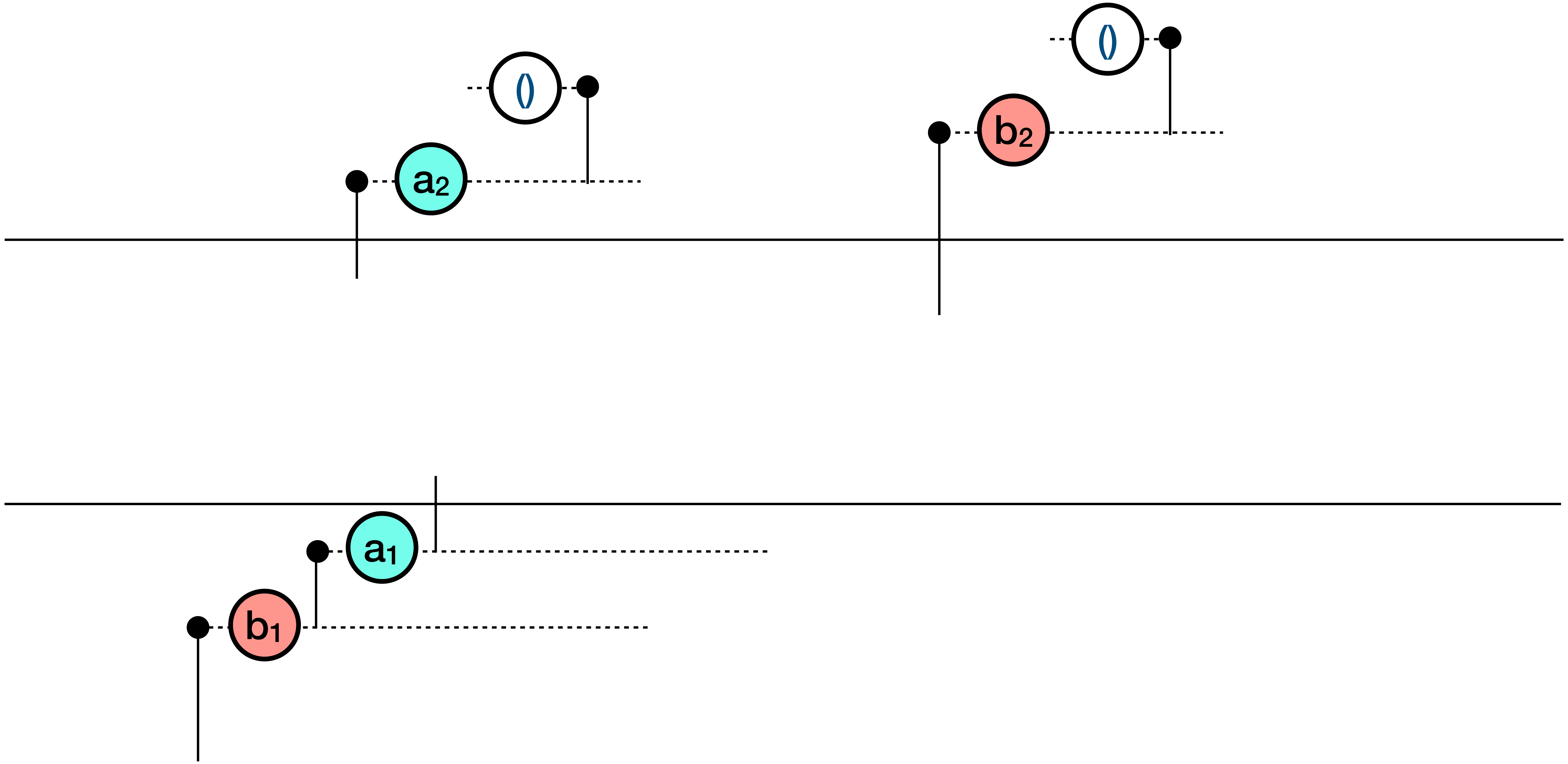
Libretto `List:merge`



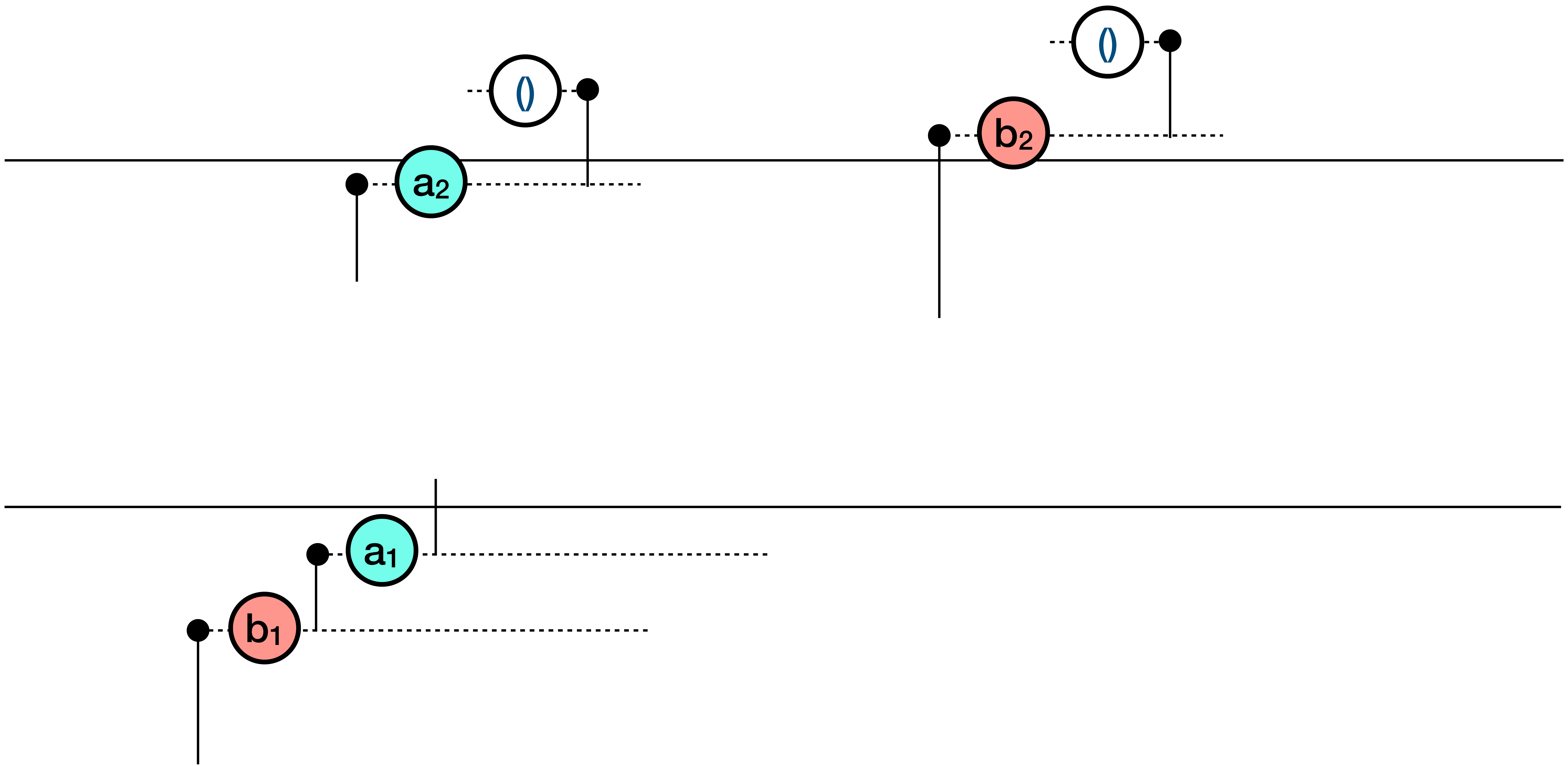
Libretto `List:merge`



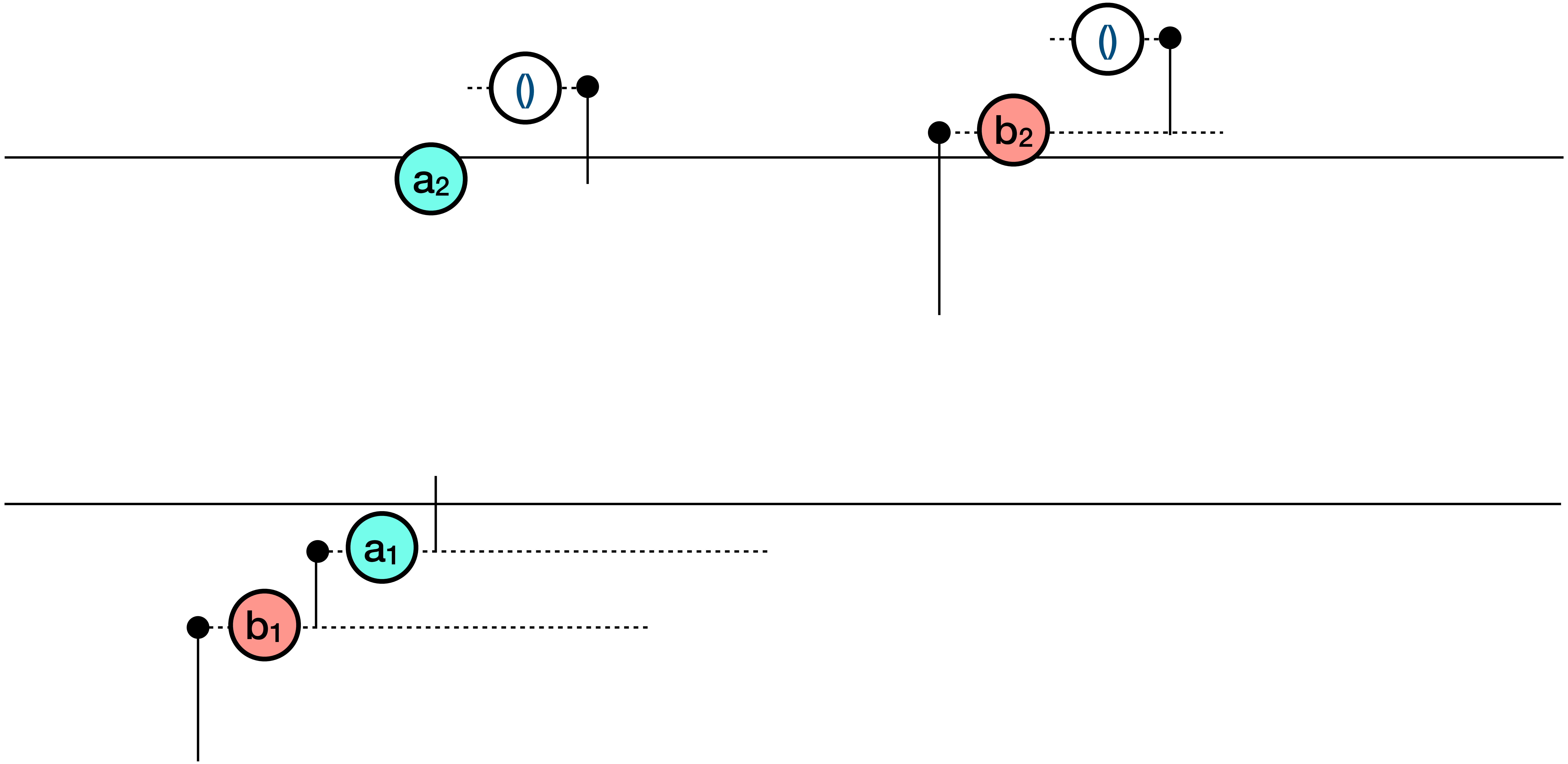
Libretto `List:merge`



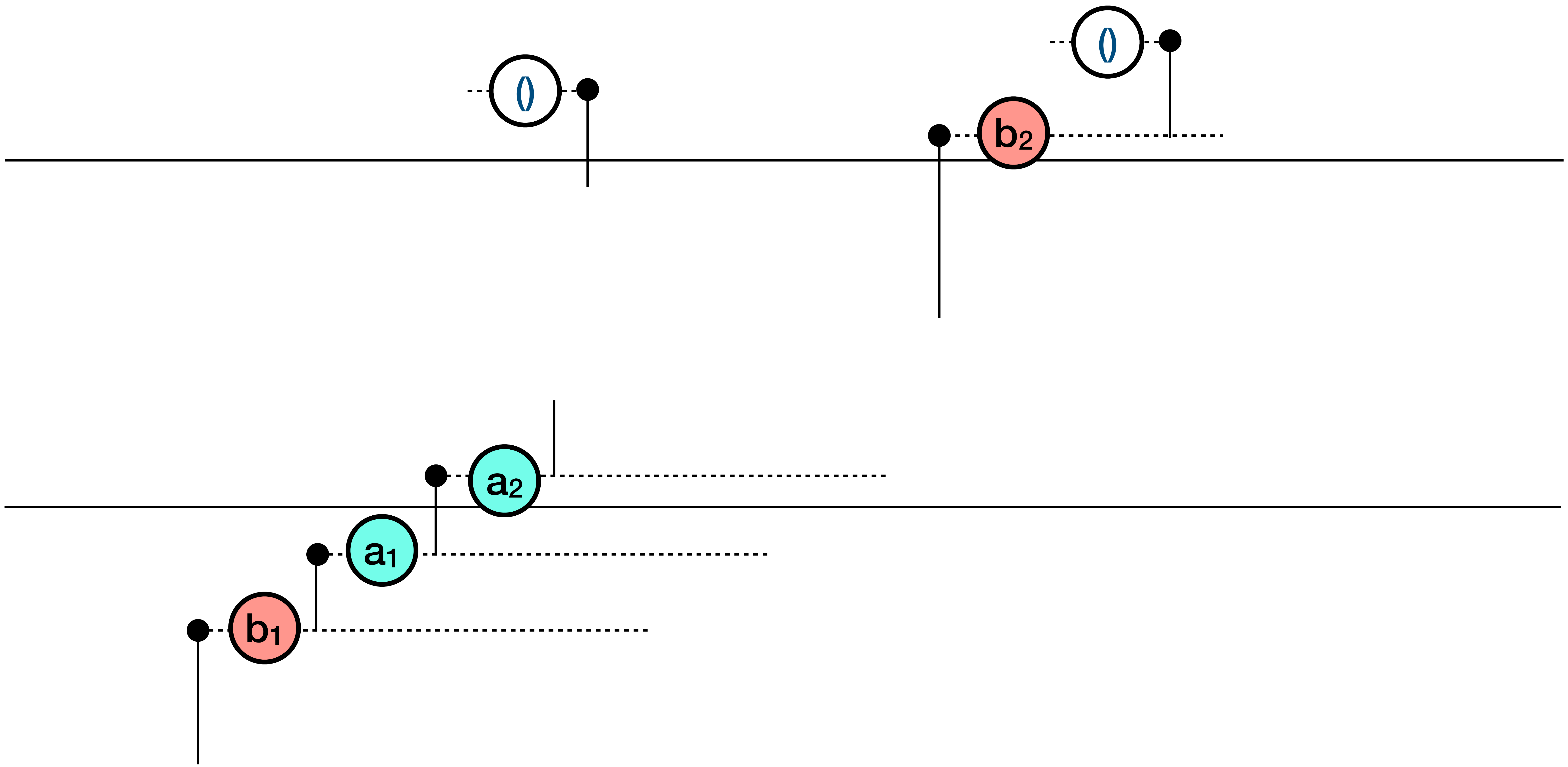
Libretto `List:merge`



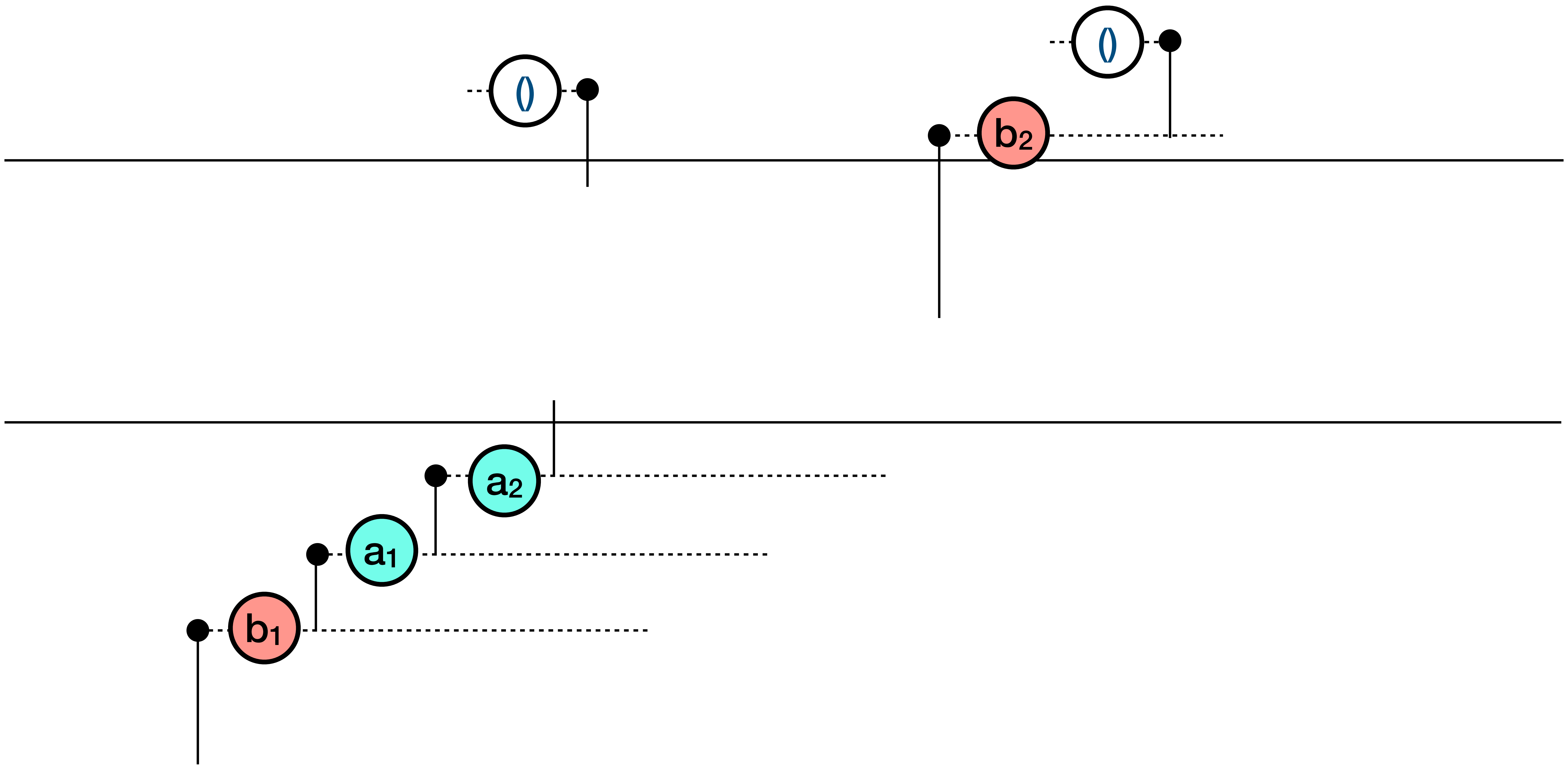
Libretto `List : merge`



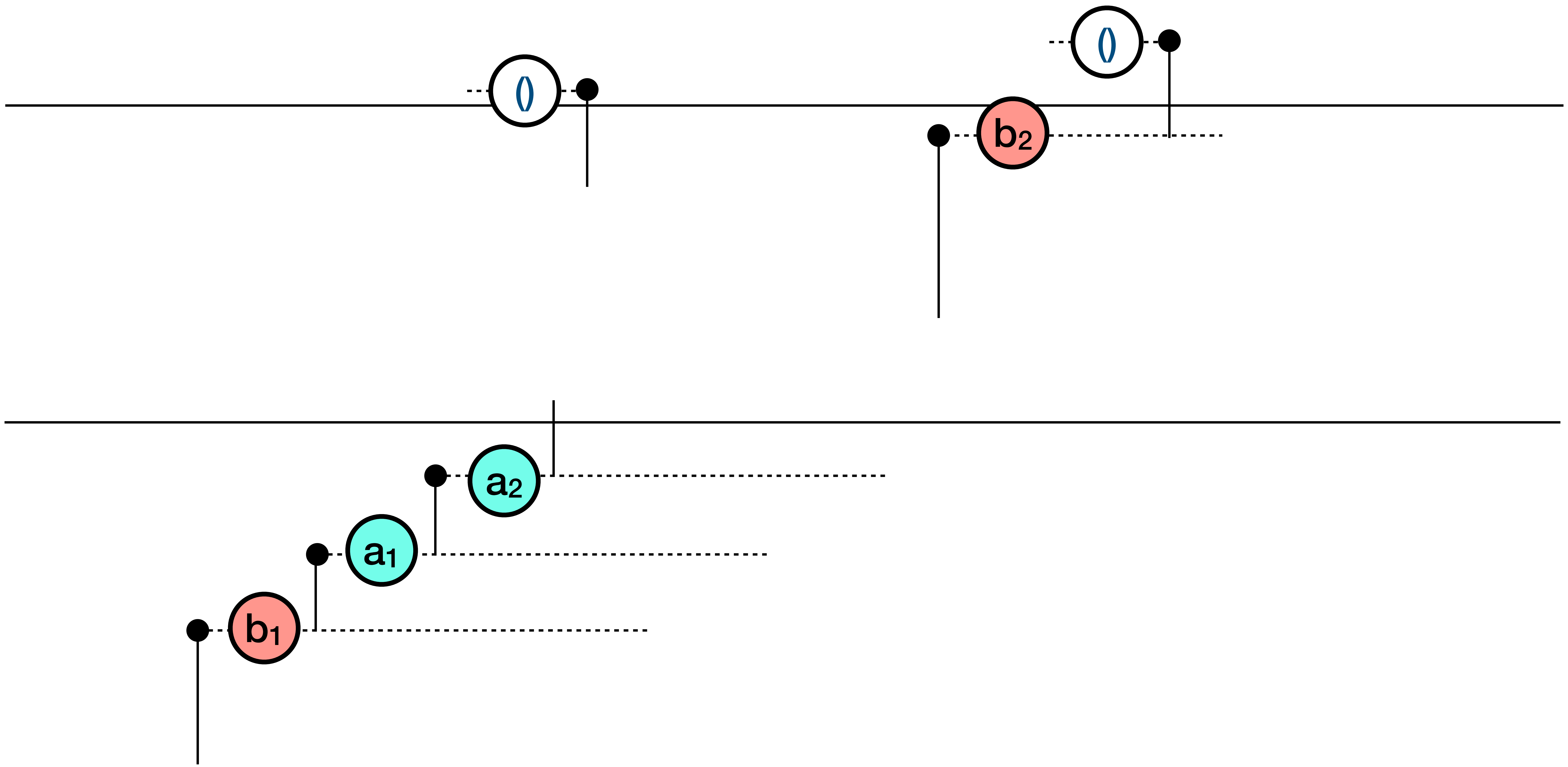
Libretto `List:merge`



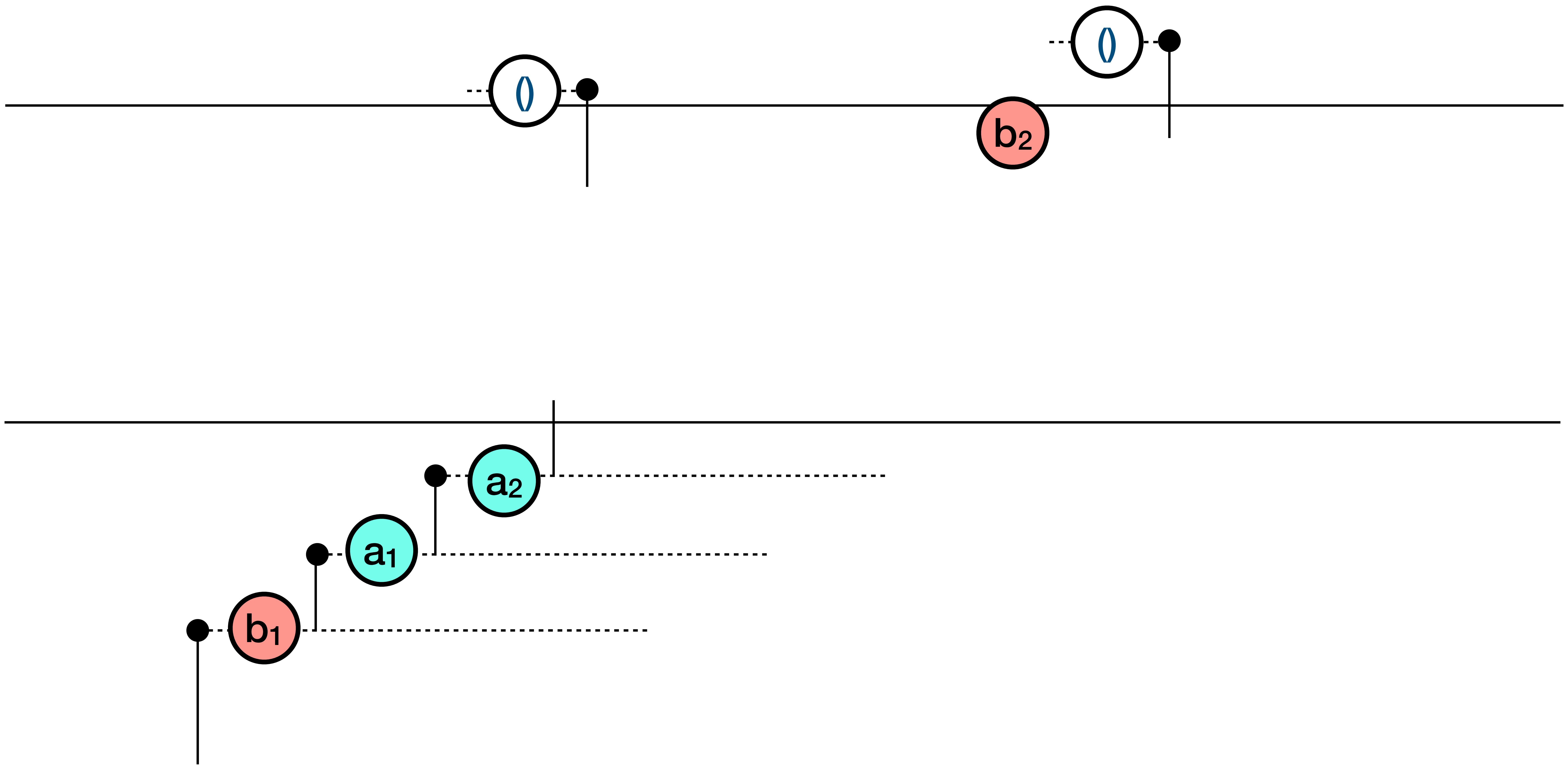
Libretto `List:merge`



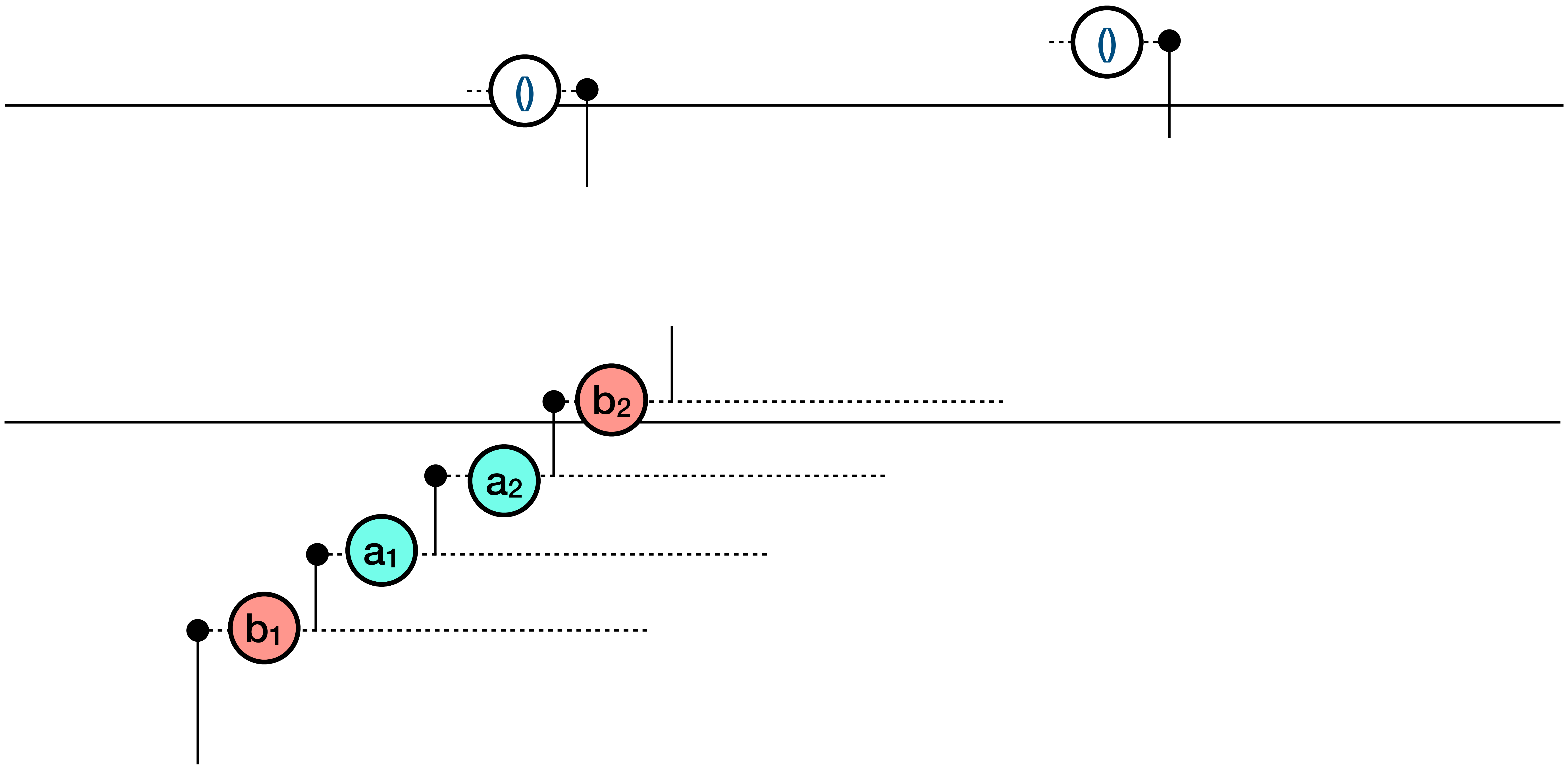
Libretto `List:merge`



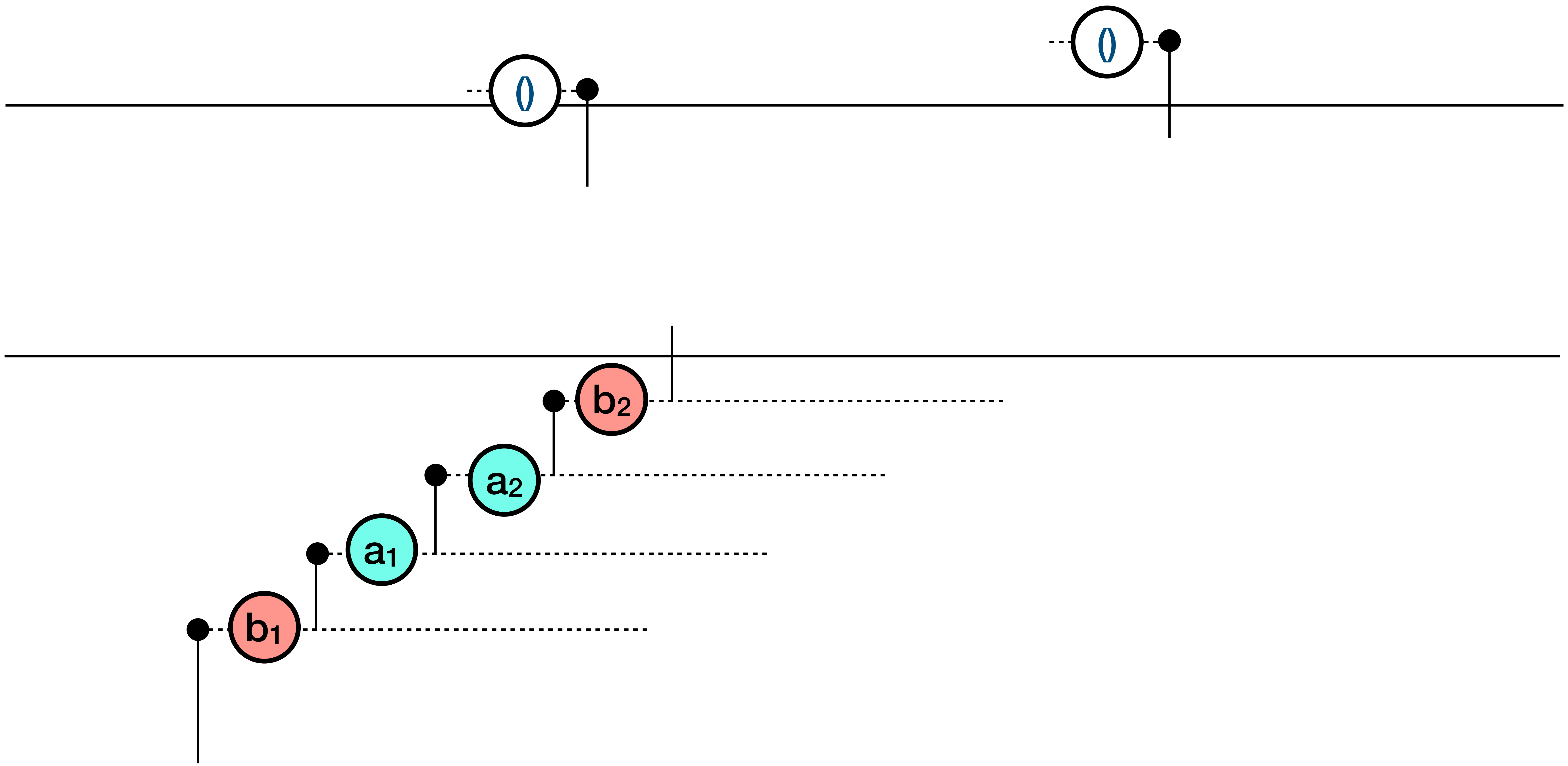
Libretto `List:merge`



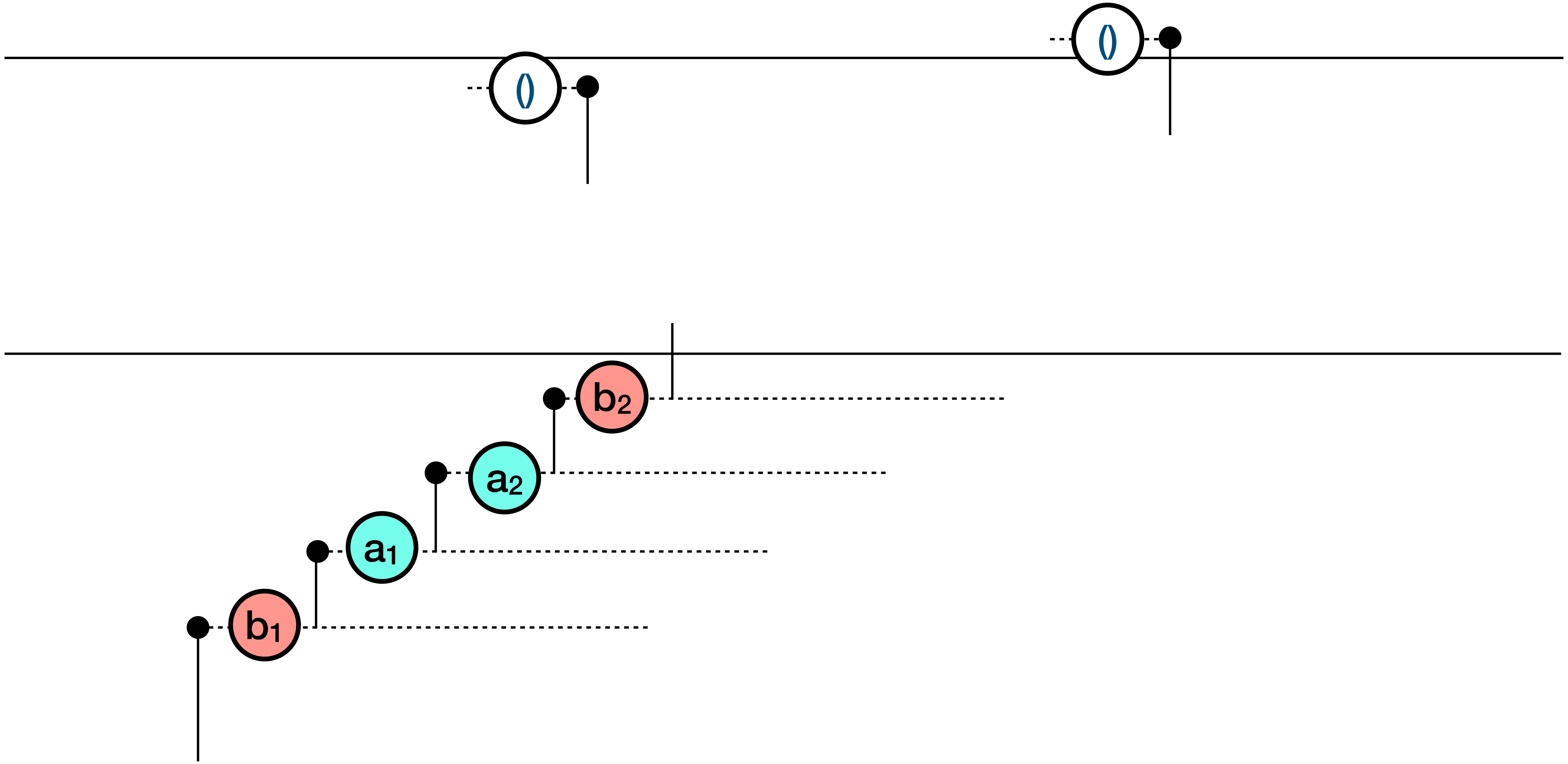
Libretto `List:merge`



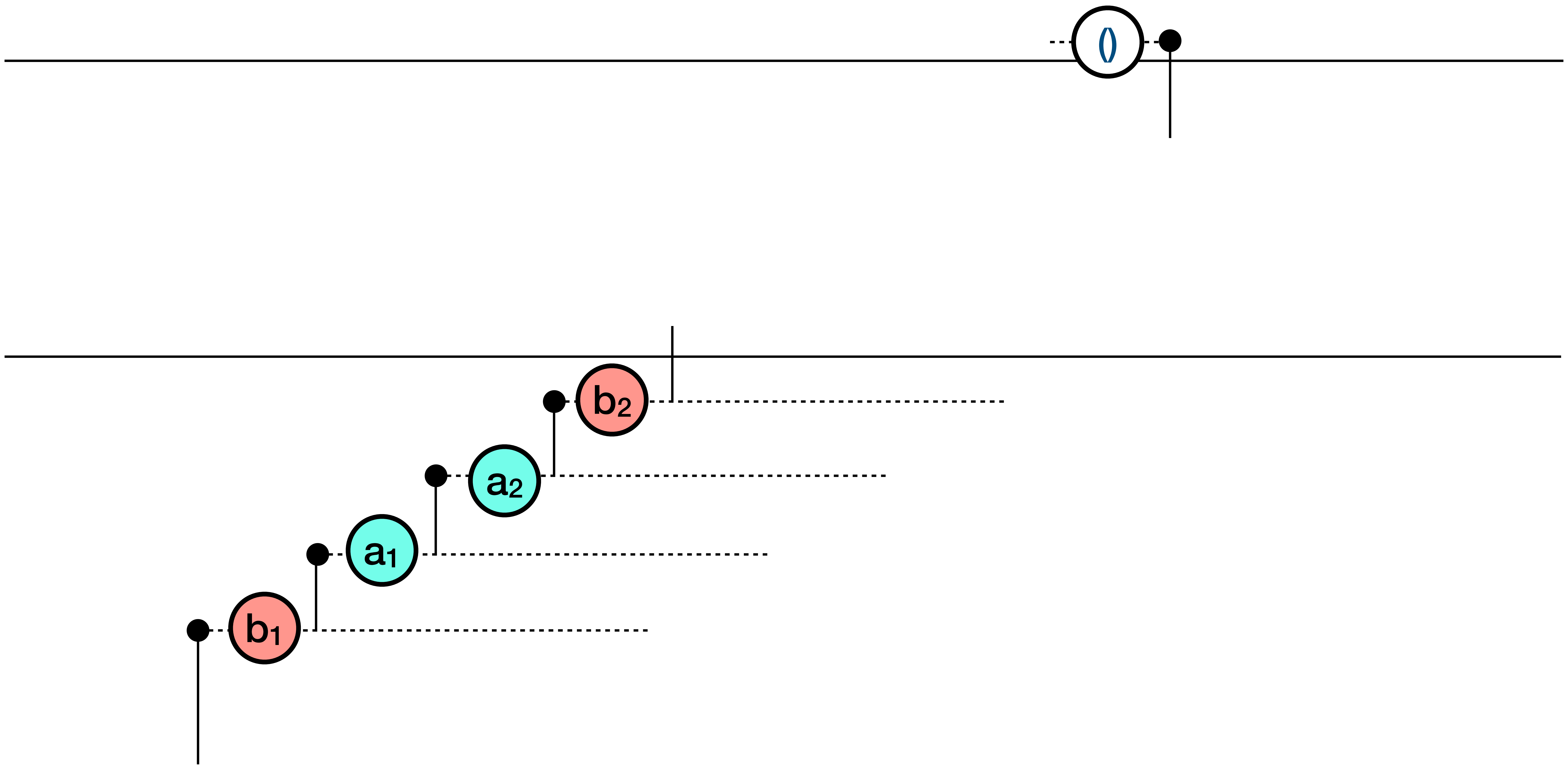
Libretto `List:merge`



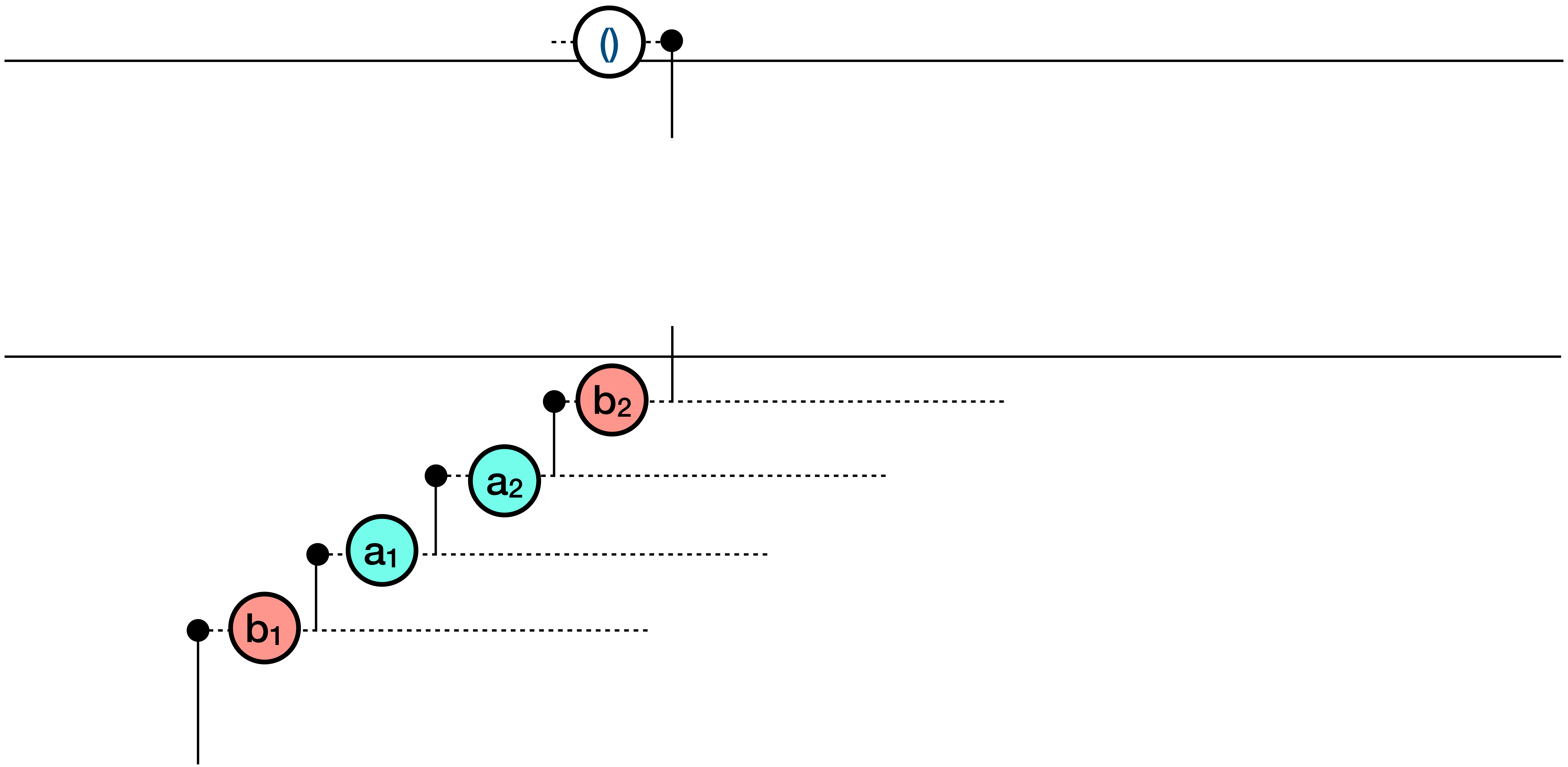
Libretto `List:merge`



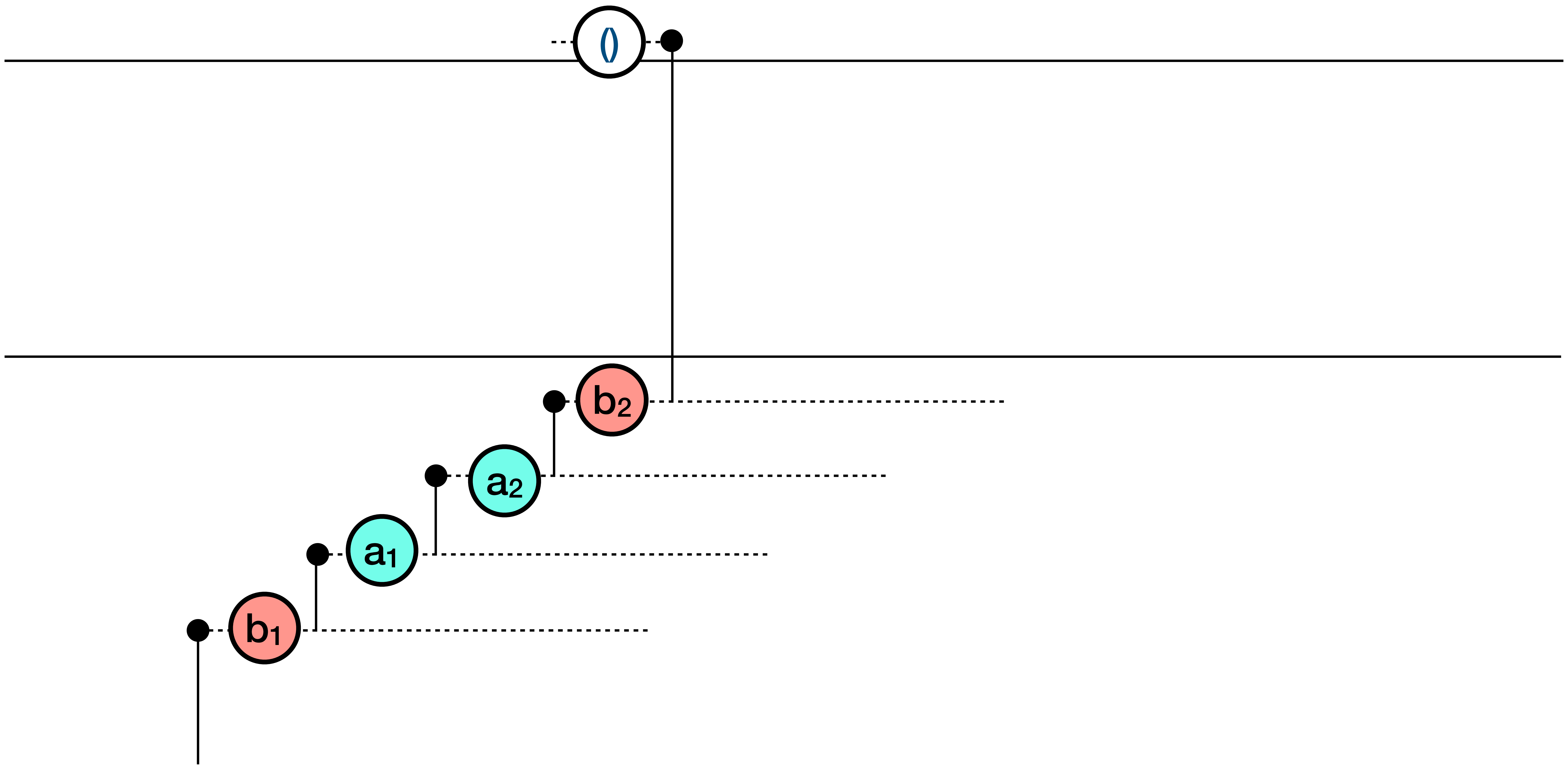
Libretto `List:merge`



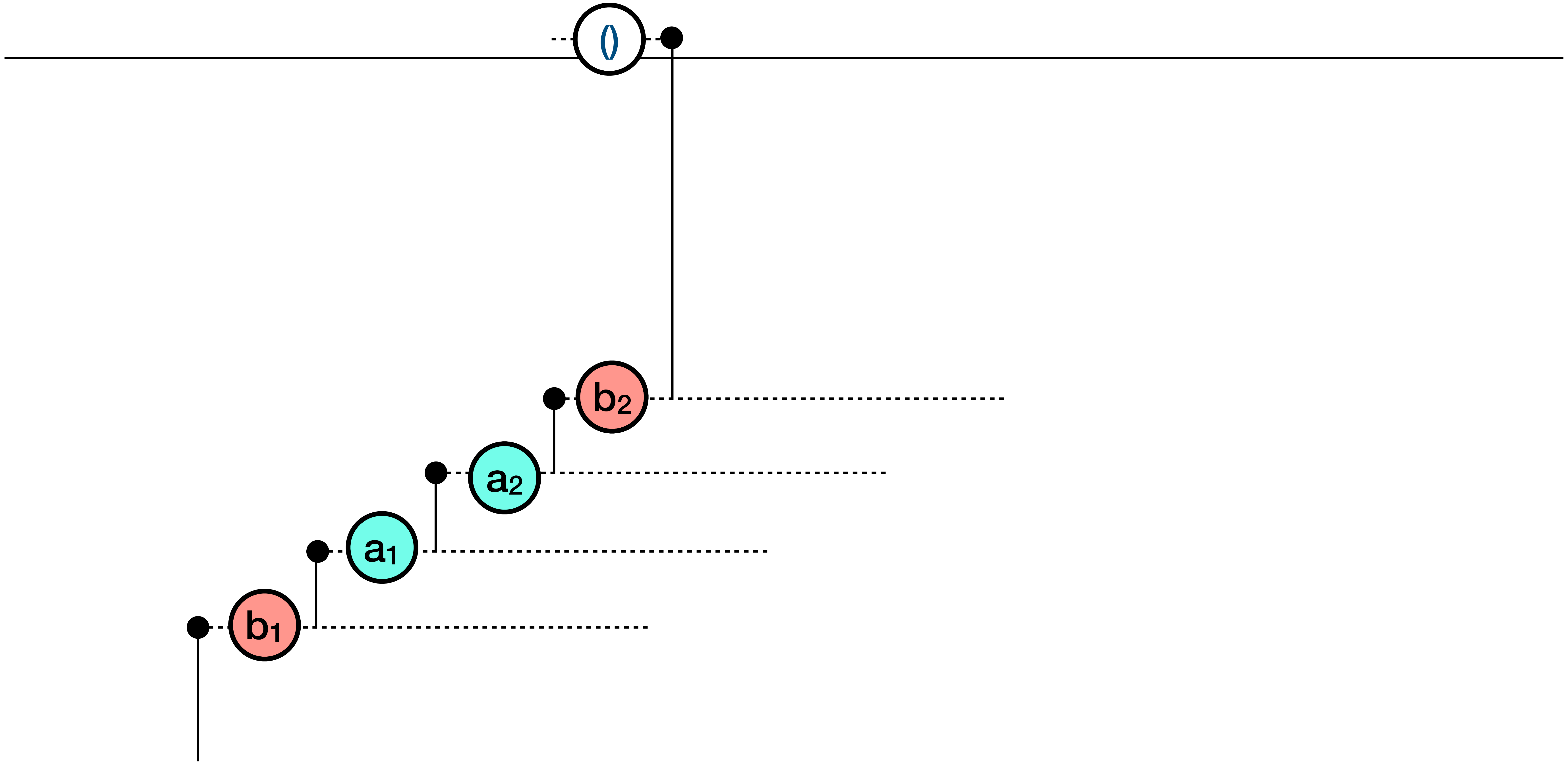
Libretto `List:merge`



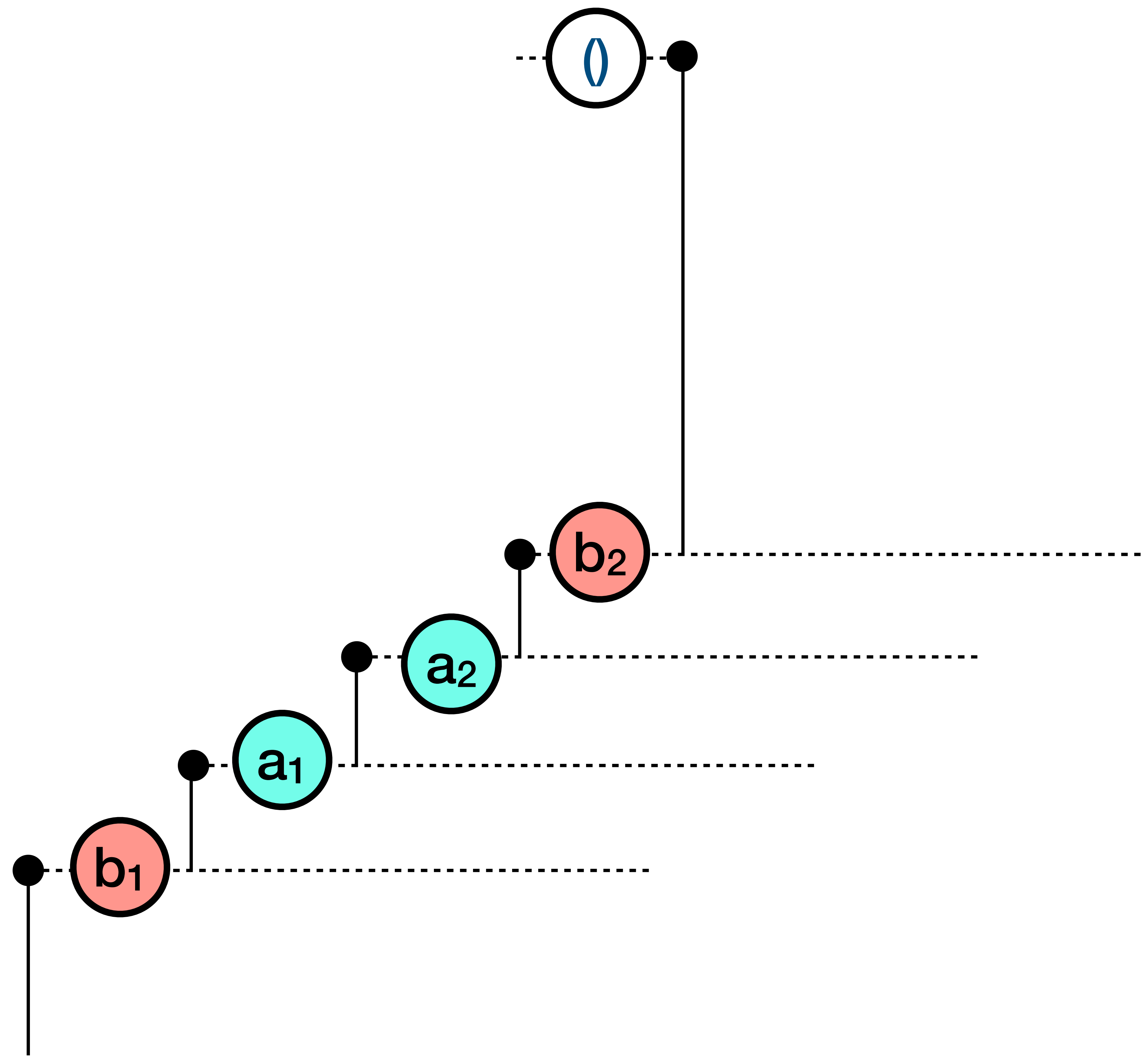
Libretto `List:merge`



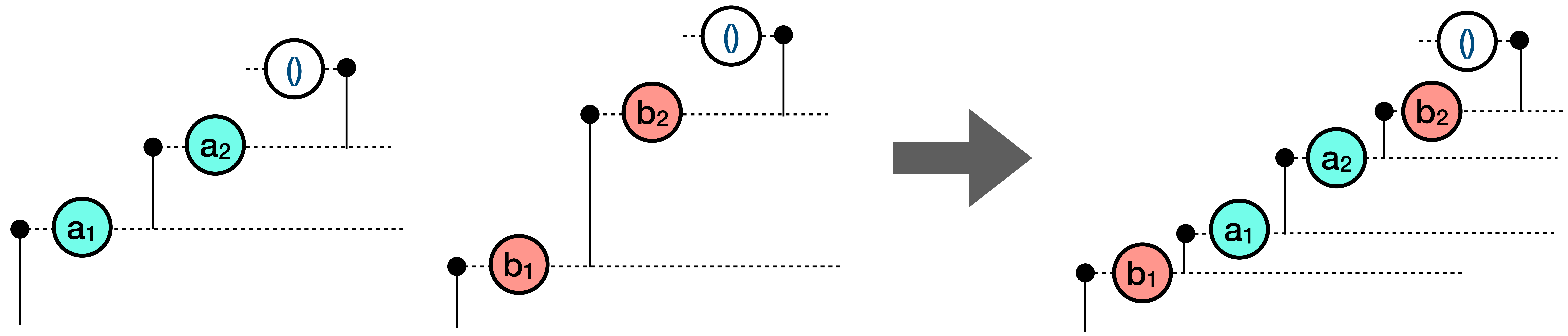
Libretto `List:merge`



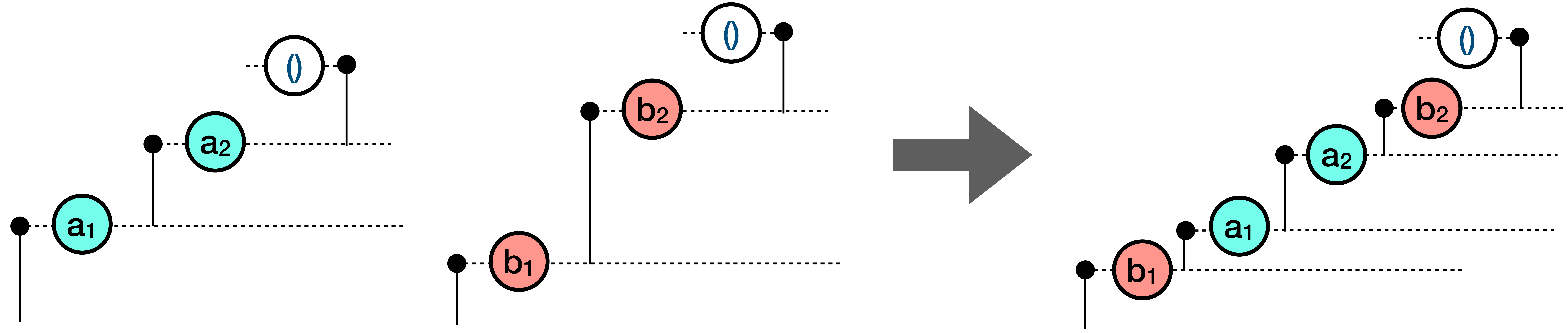
Libretto `List:merge`



Libretto `List:merge`



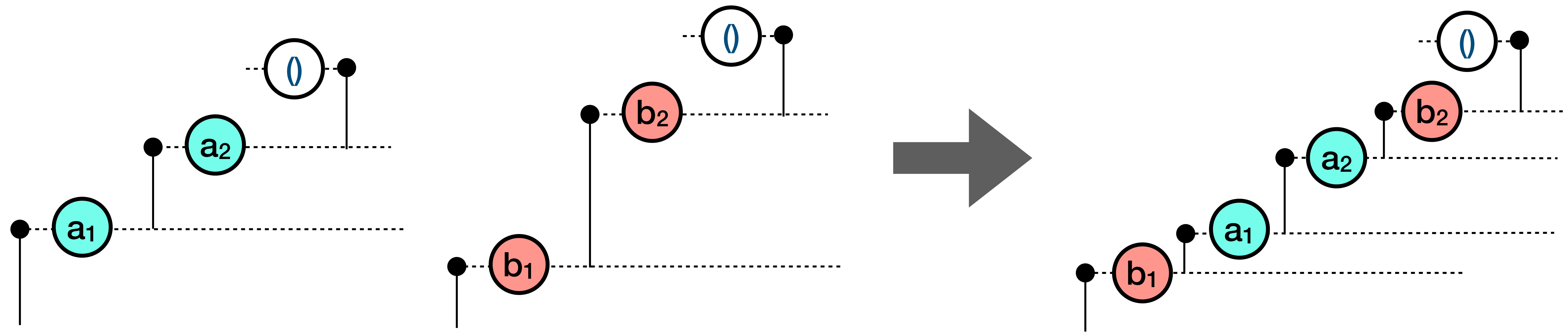
Libretto `List:merge`



Preserves *"temporal"*(*) order

(*) but there's no global time

Libretto `List`:`merge`

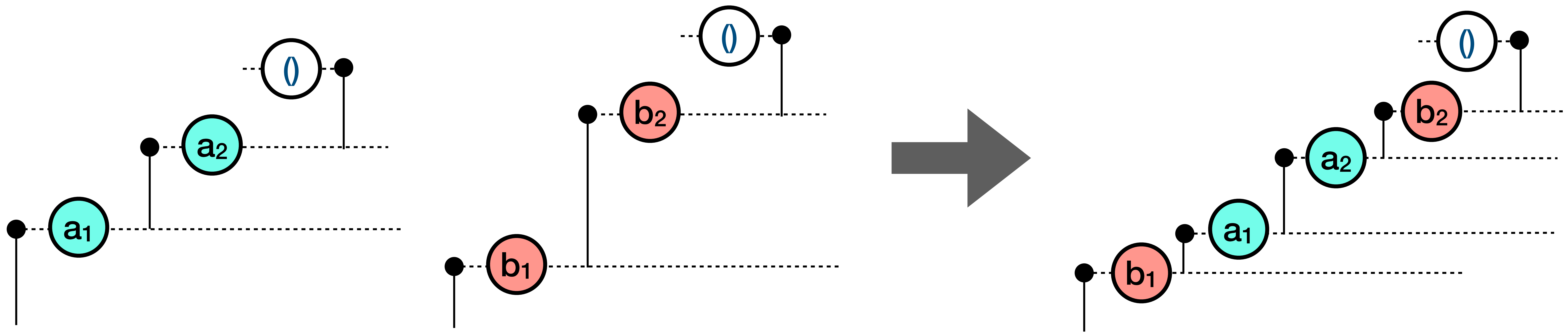


Preserves *"temporal"*(*) order

Is it *associative?*

(*) but there's no global time

Libretto `List : merge`



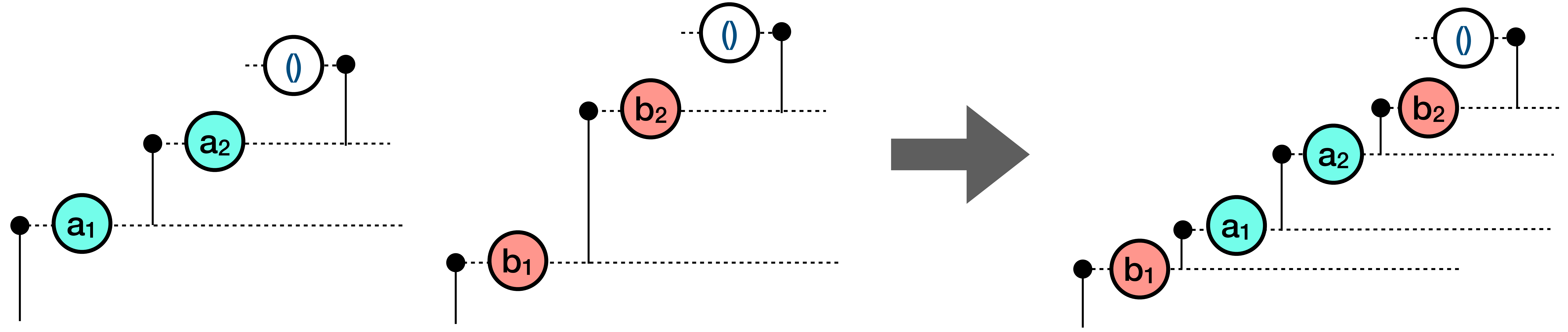
Preserves *“temporal”*(*) order

Is it *associative?*

```
(as merge bs) merge cs  =?=  as merge (bs merge cs)
```

(*) but there’s no global time

Libretto `List` : `merge`



Preserves *"temporal"*(*) order

Is it *associative?*

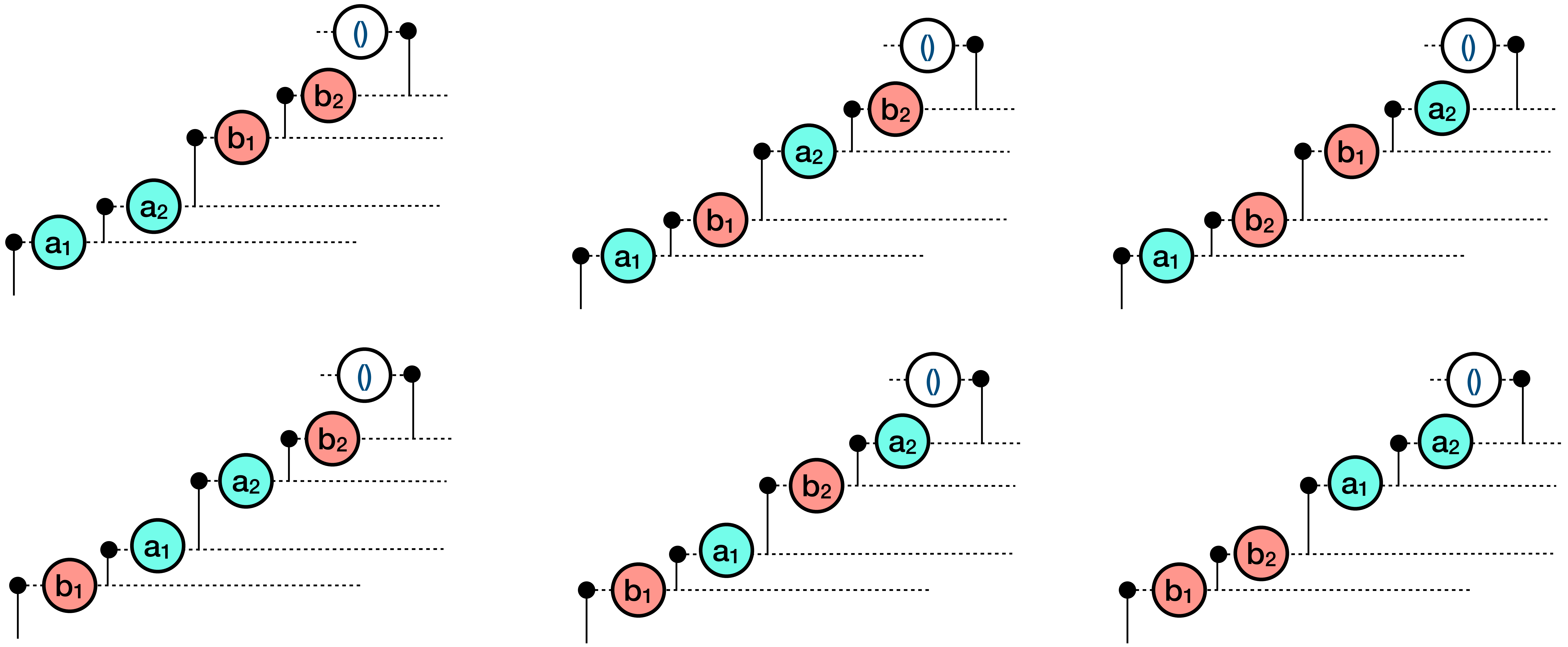
```
(as merge bs) merge cs  =?=  as merge (bs merge cs)
```

It seems so, but what about the **non-determinism?**

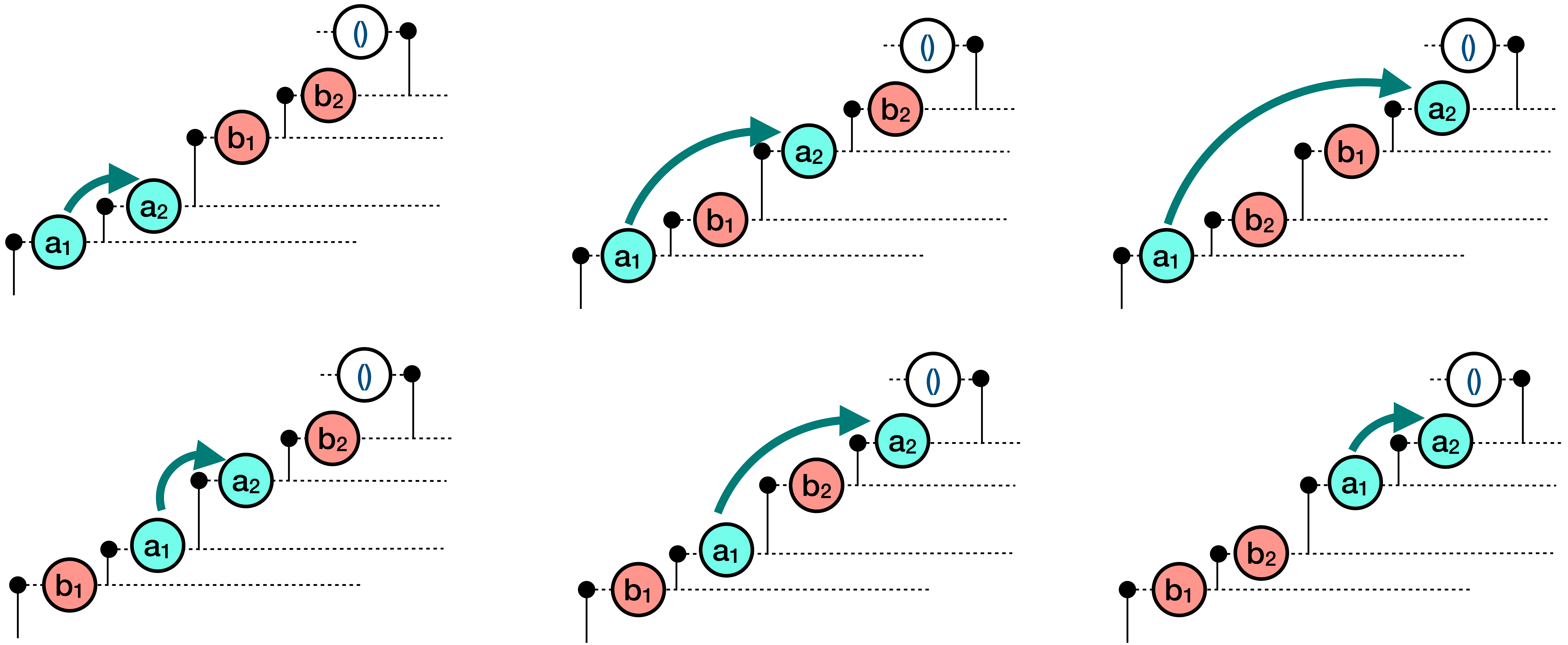
(*) but there's no global time

`merge` : possible outcomes

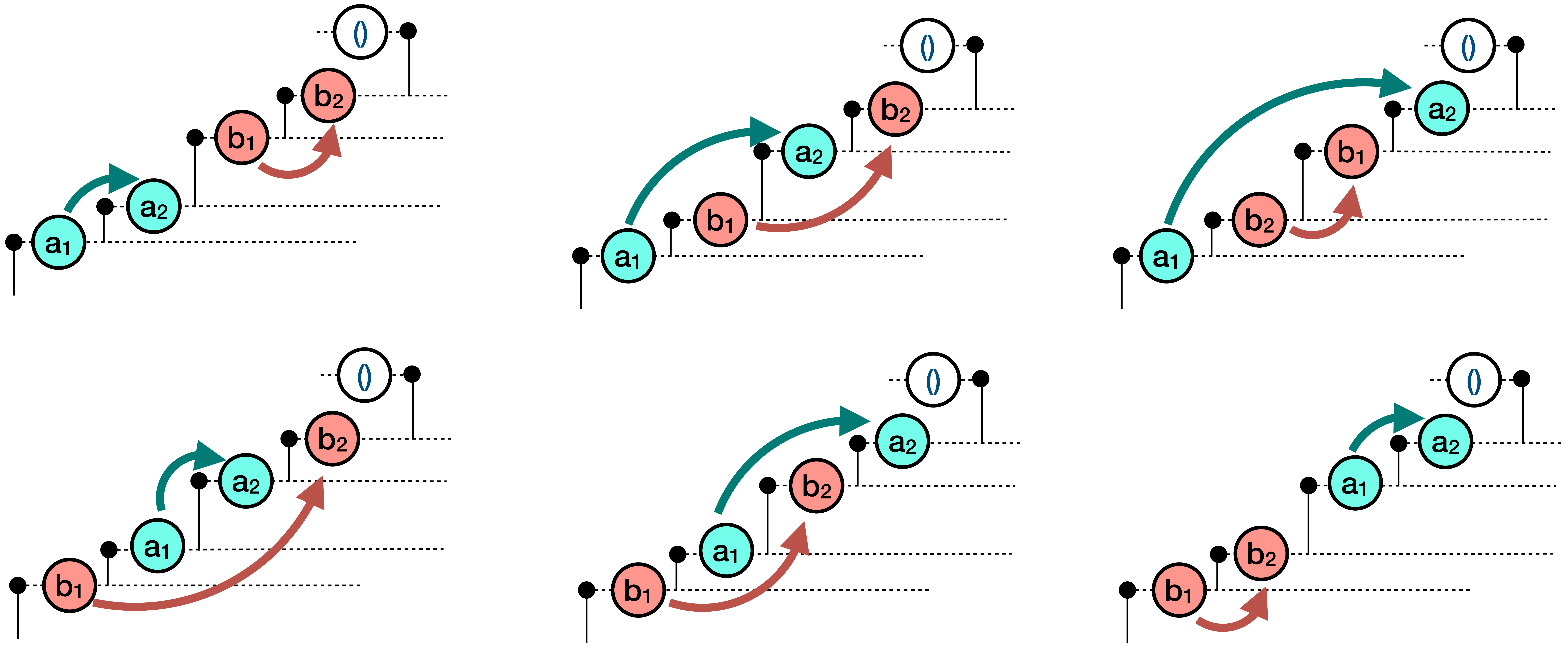
`merge` : possible outcomes



`merge` : possible outcomes



`merge` : possible outcomes



Equality of Non-deterministic Functions

`f = g`

What does it mean for **non-deterministic** *f, g?*

Equality of Non-deterministic Functions

`f  =  g`

What does it mean for **non-deterministic** *f, g?*

Deterministic functions

Equality of Non-deterministic Functions

`f  =  g`

What does it mean for **non-deterministic** *f, g?*

Deterministic functions

- *"same input leads to same output":* $\forall\, a: A \,.\, f(a) = g(a)$

Equality of Non-deterministic Functions

`f = g`

What does it mean for **non-deterministic** *f, g?*

Deterministic functions

- *"same input leads to same output":* $\forall\, a{:}\, A \,.\, f(a) = g(a)$
- more generally: *"same observable behavior"*

Equality of Non-deterministic Functions

`f = g`

What does it mean for **non-deterministic** *f, g?*

Deterministic functions

- *"same input leads to same output":* $\forall\, a{:}\, A\ .\ f(a) = g(a)$
- more generally: *"same observable behavior"*

Non-deterministic functions

Equality of Non-deterministic Functions

`f   =   g`

What does it mean for **non-deterministic** *f, g?*

Deterministic functions

- *"same input leads to same output":* $\forall\, a: A\ .\ f(a) = g(a)$
- more generally: *"same observable behavior"*

Non-deterministic functions

- *"same **set** of observable behaviors"*

Equality of Non-deterministic Functions

`f = g`

What does it mean for **non-deterministic** *f, g?*

Deterministic functions

- *"same input leads to same output":* $\forall\, a: A\ .\ f(a) = g(a)$
- more generally: *"same observable behavior"*

Non-deterministic functions

- *"same **set** of observable behaviors"*
- not necessarily the same probabilistic distribution of them

`merge` : associativity

```
(as merge bs) merge cs   =?=   as merge (bs merge cs)
```

`merge` : associativity

```
(as merge bs) merge cs   =?=   as merge (bs merge cs)
```

Strategy: (intuitively, not a formal proof)

`merge` : associativity

```
(as merge bs) merge cs   =?=   as merge (bs merge cs)
```

Strategy: (intuitively, not a formal proof)

1. Consider any observable behavior of the left side.

`merge` : associativity

```
(as merge bs) merge cs   =?=   as merge (bs merge cs)
```

Strategy: (intuitively, not a formal proof)

1. Consider any observable behavior of the left side.
2. Show that the same observable behavior may be exhibited by right side.

`merge` : associativity

```
(as merge bs) merge cs   =?=   as merge (bs merge cs)
```

Strategy: (intuitively, not a formal proof)

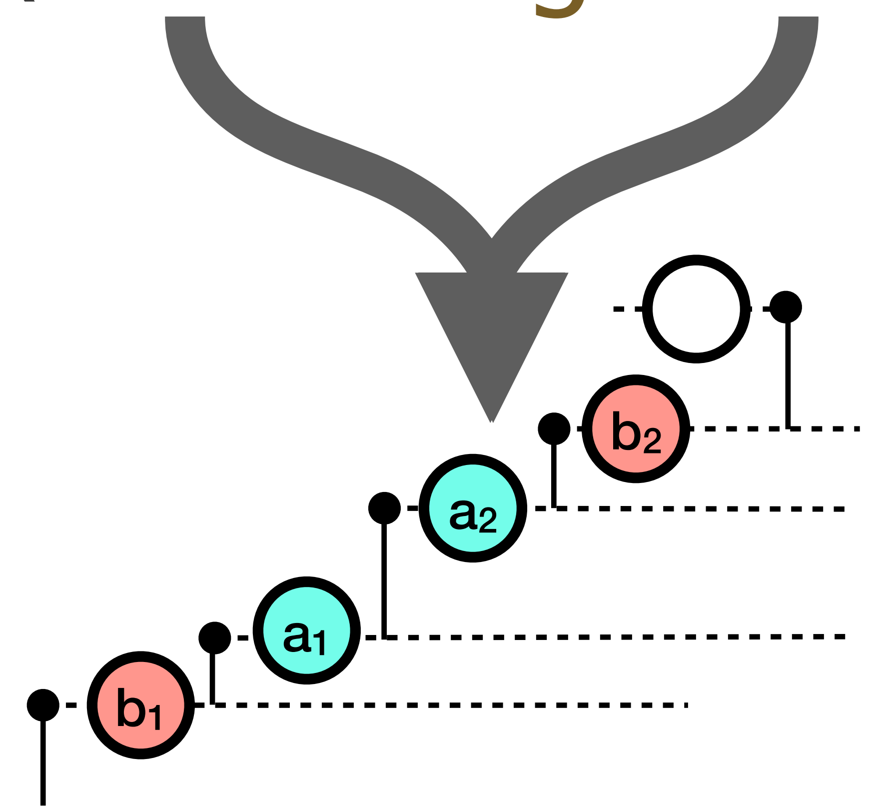
1. Consider any observable behavior of the left side.
2. Show that the same observable behavior may be exhibited by right side.
3. And vice versa.

`merge` : associativity

```
(as merge bs) merge cs   =?=   as merge (bs merge cs)
```

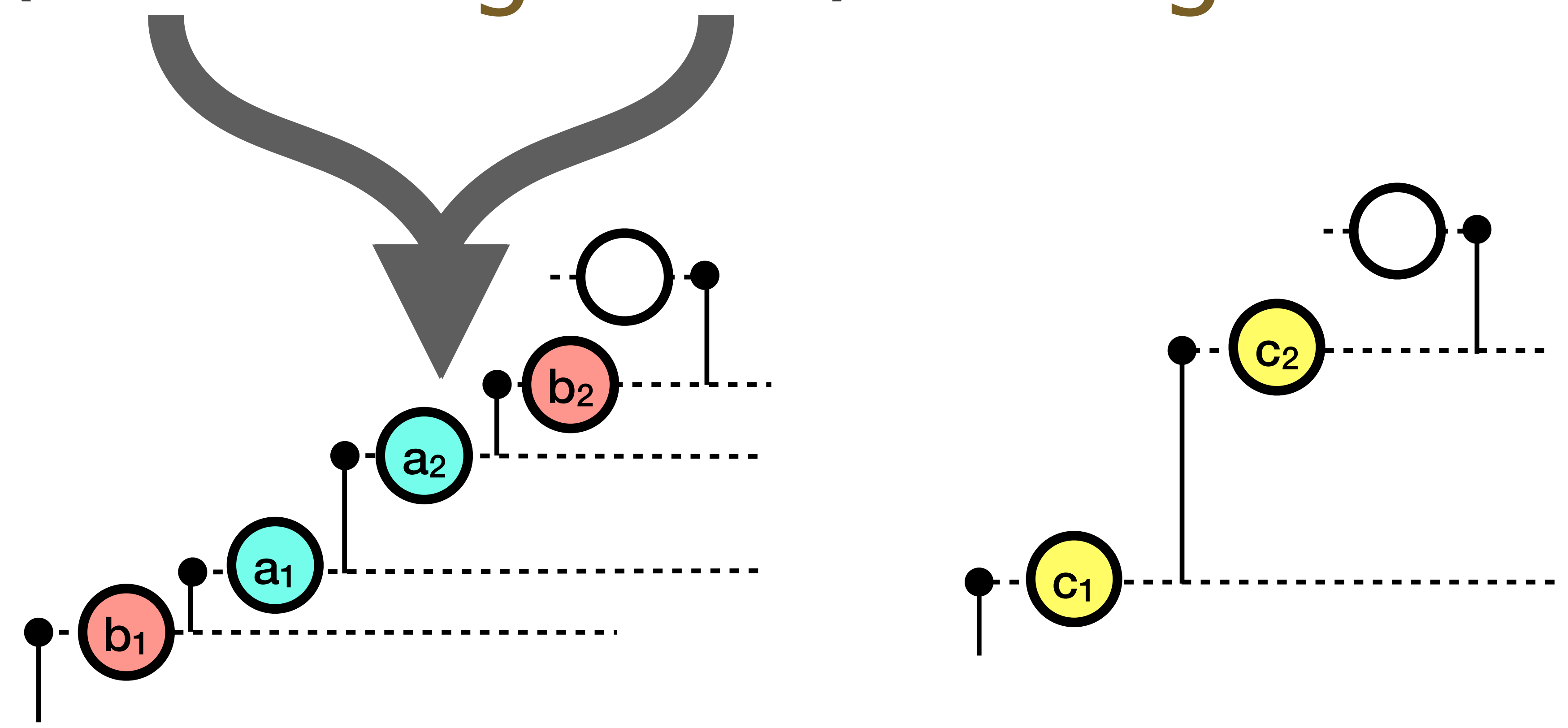
`merge` : associativity

`(as merge bs) merge cs    =?=    as merge (bs merge cs)`



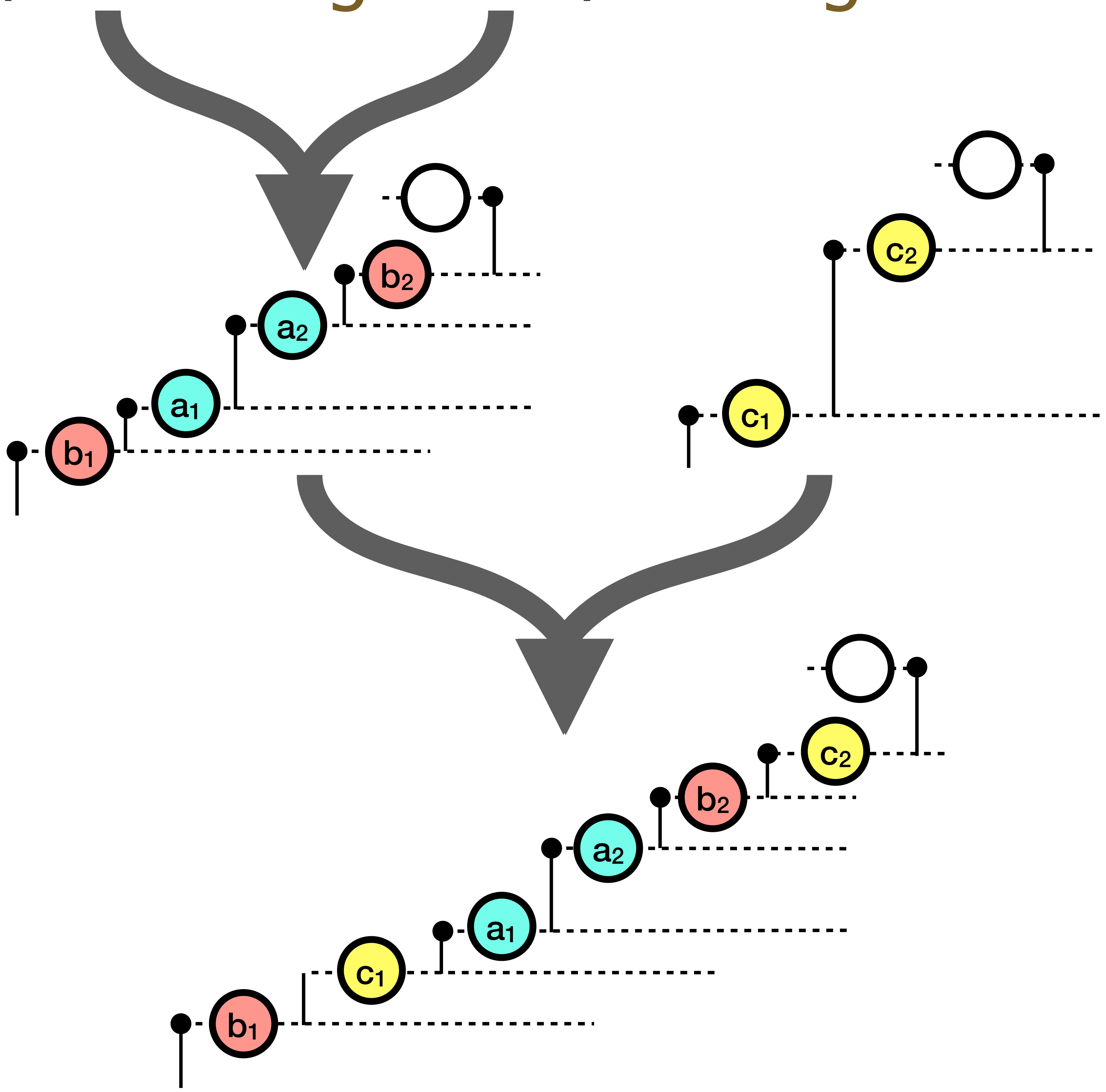
`merge` : associativity

`(as merge bs) merge cs    =?=    as merge (bs merge cs)`



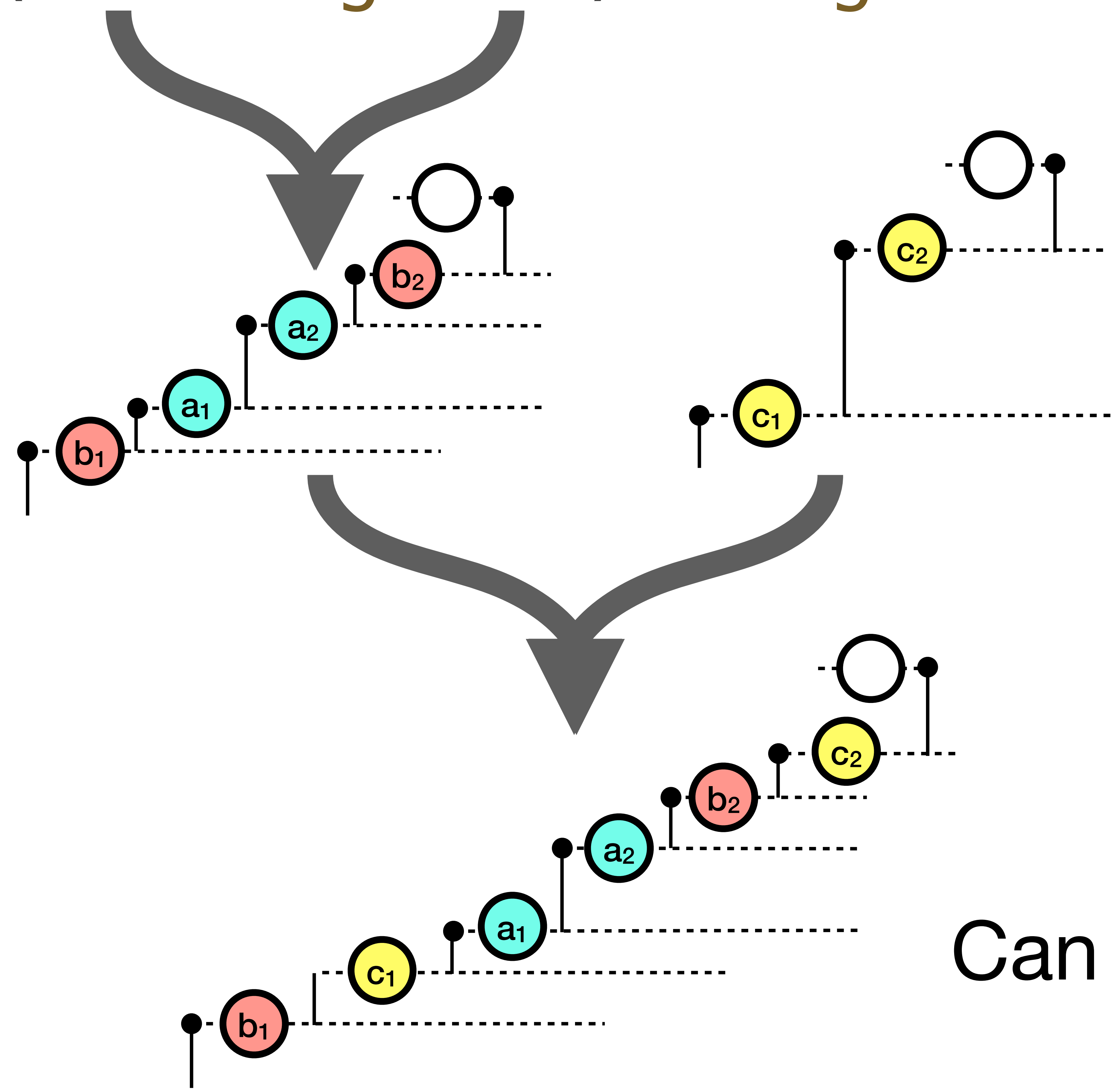
`merge` : associativity

`(as merge bs) merge cs   =?=   as merge (bs merge cs)`



`merge` : associativity

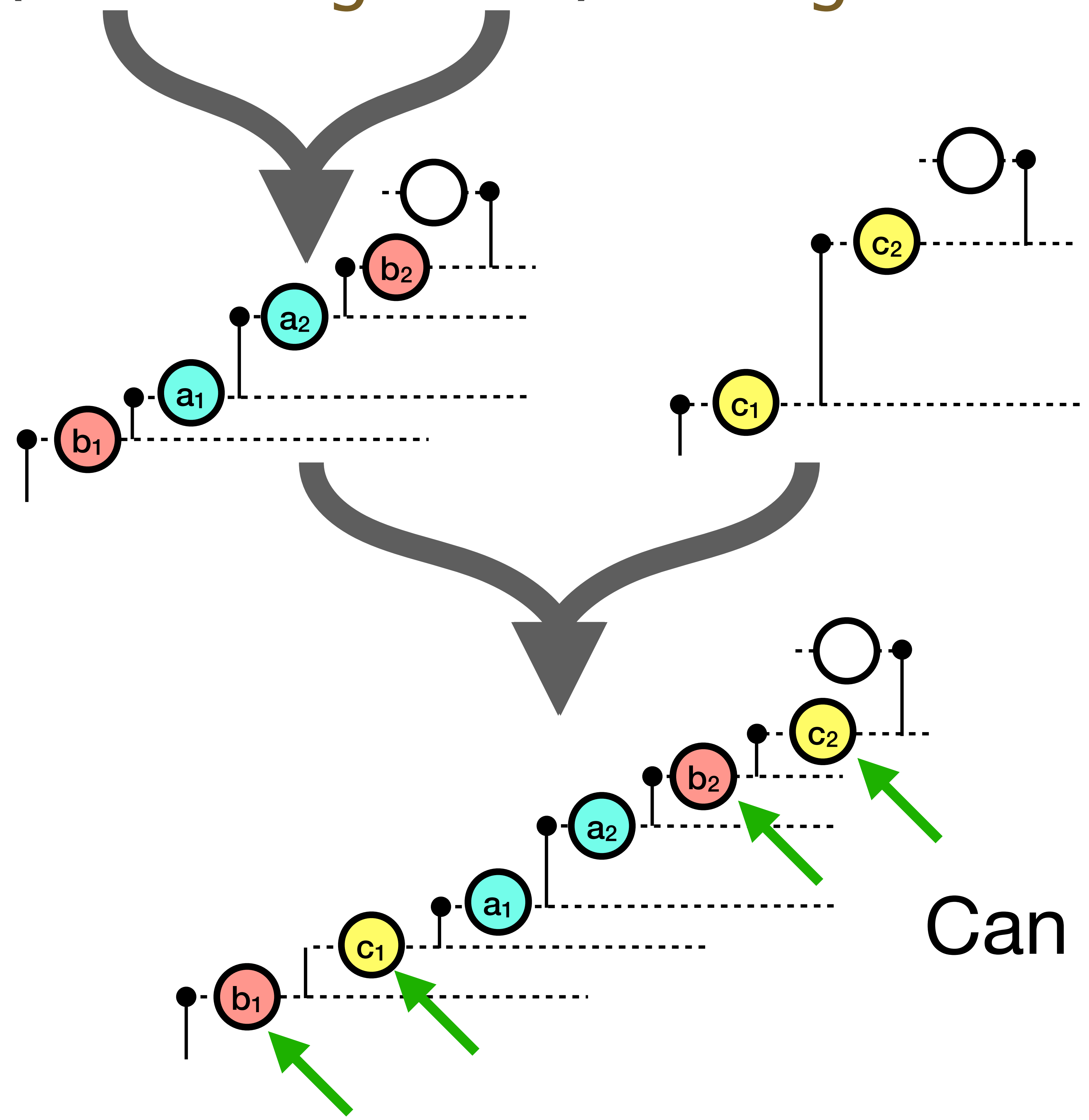
`(as merge bs) merge cs   =?=   as merge (bs merge cs)`



Can the same be obtained on the right?

`merge` : associativity

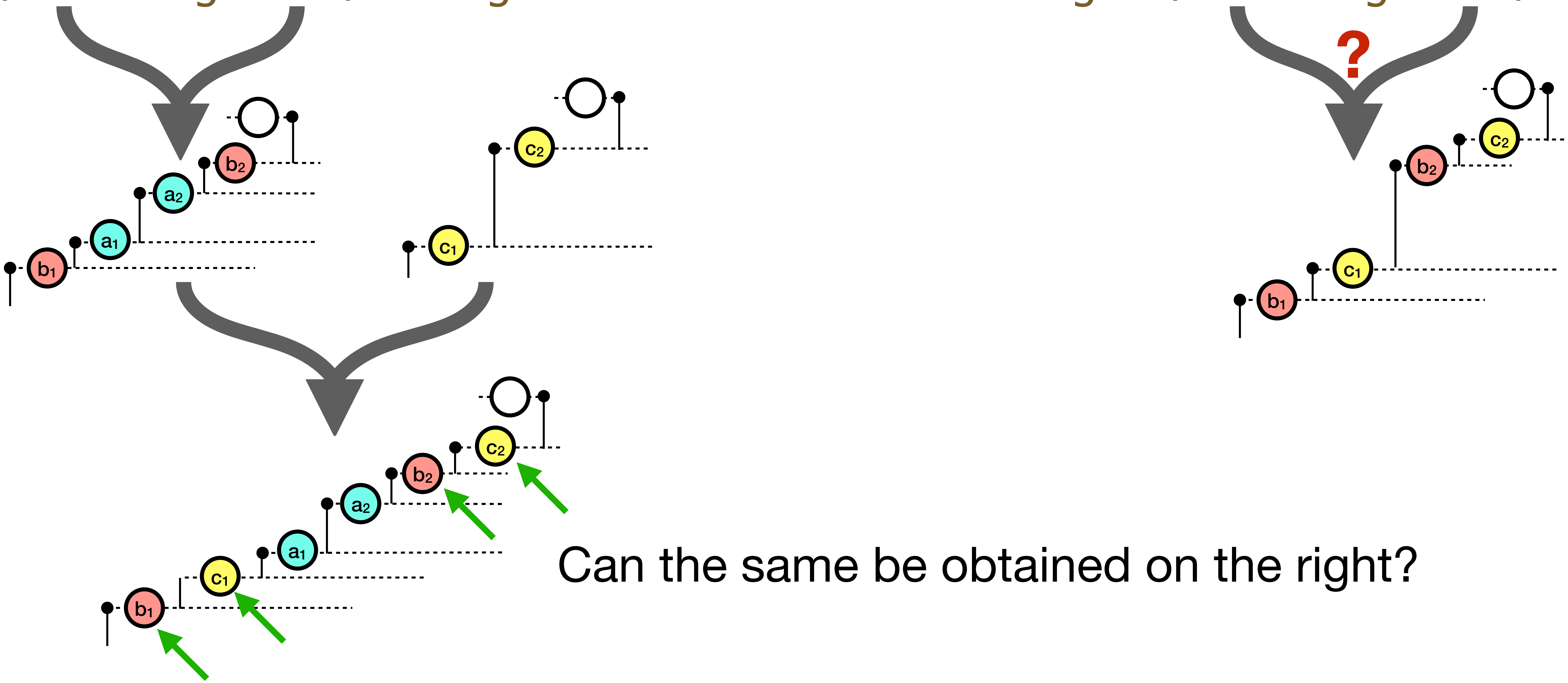
`(as merge bs) merge cs    =?=    as merge (bs merge cs)`



Can the same be obtained on the right?

`merge` : associativity

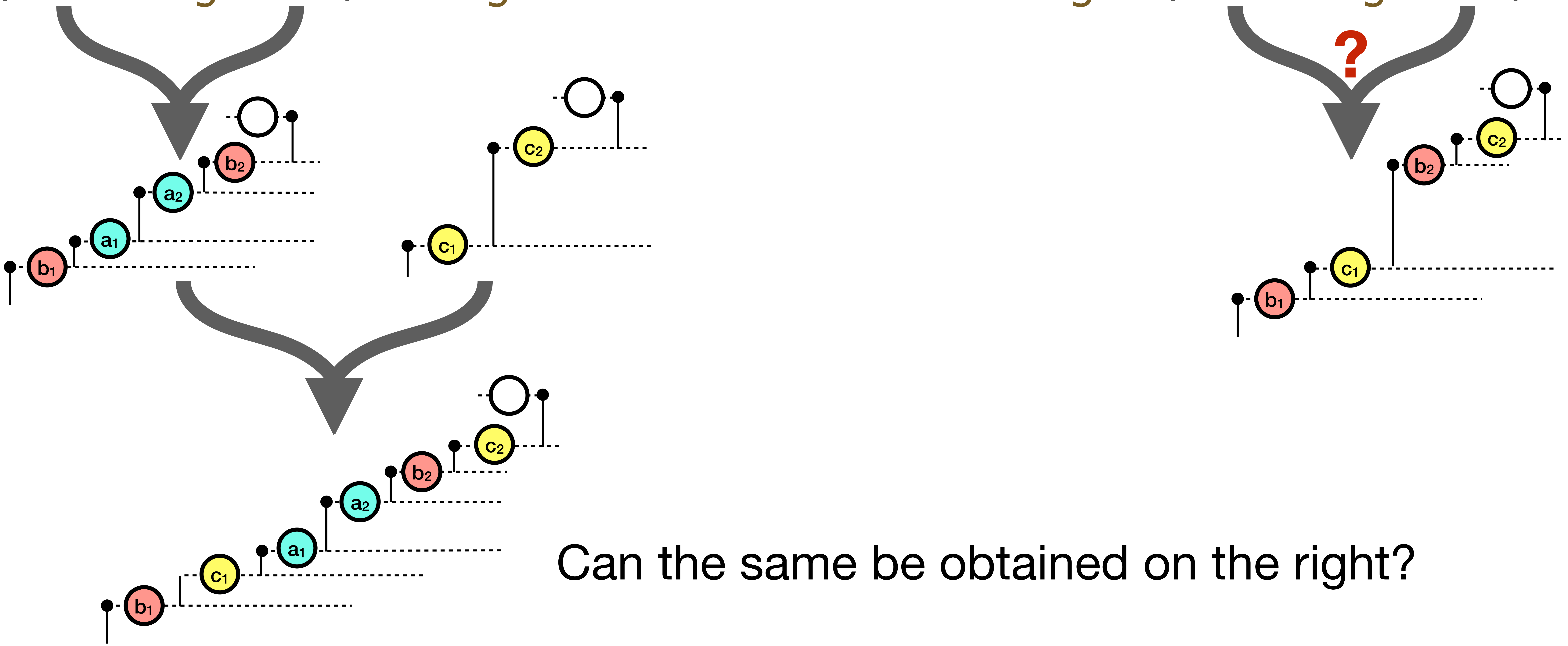
`(as merge bs) merge cs   =?=   as merge (bs merge cs)`



Can the same be obtained on the right?

`merge` : associativity

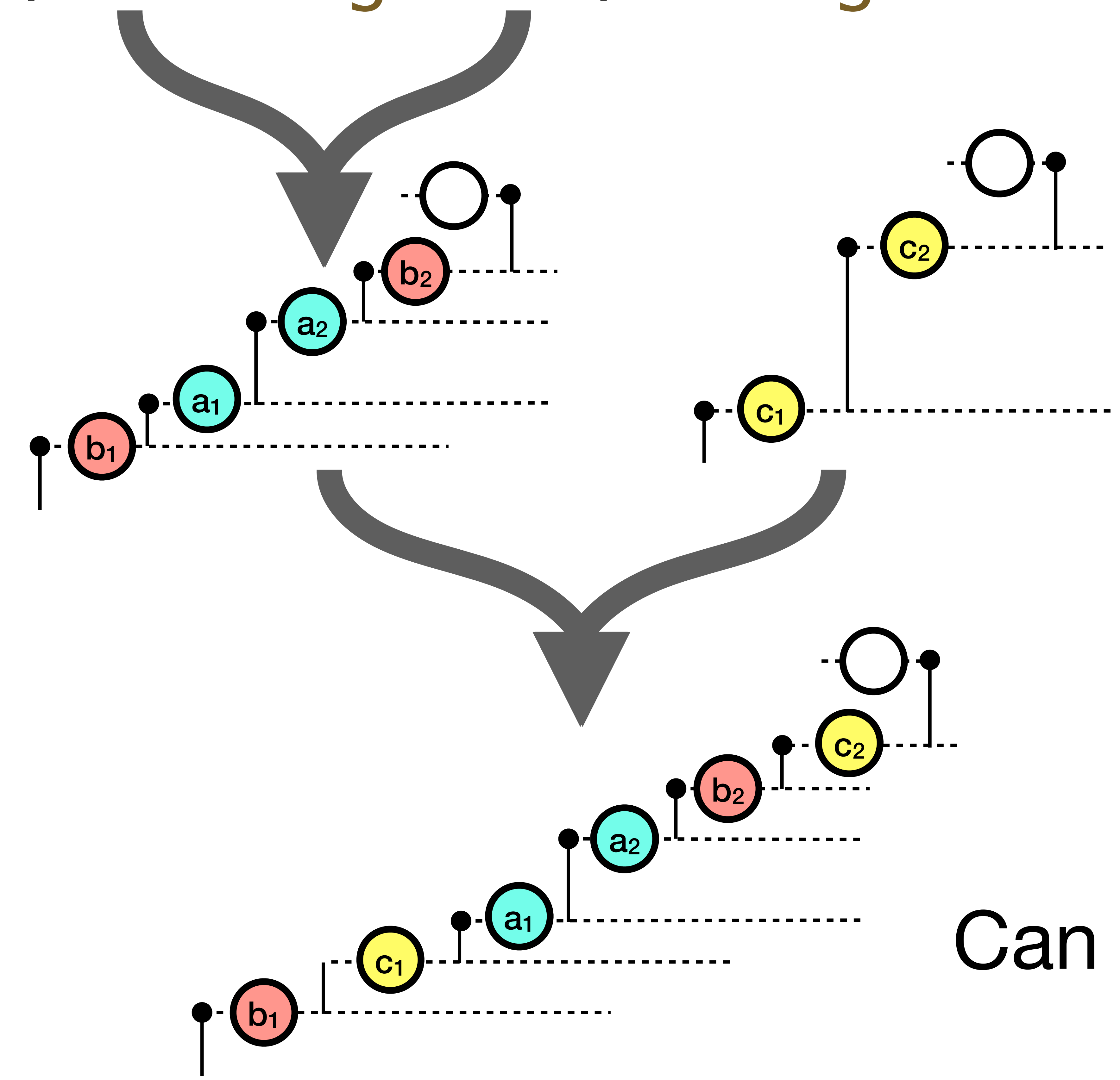
`(as merge bs) merge cs   =?=   as merge (bs merge cs)`

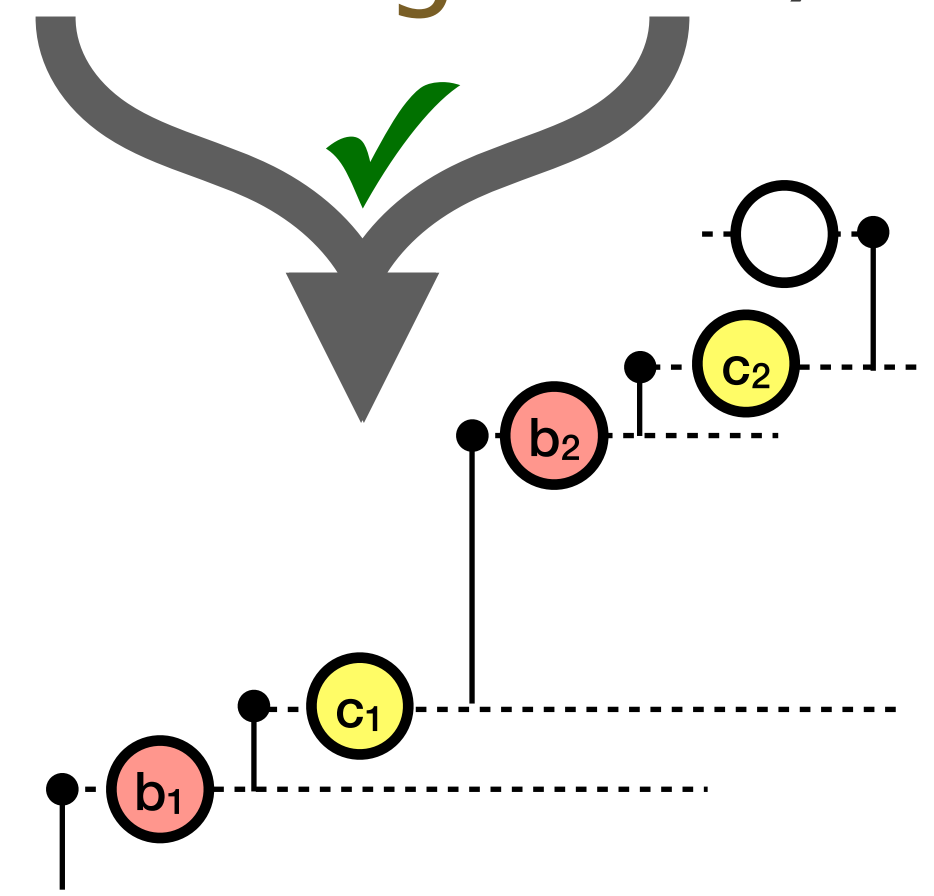


Can the same be obtained on the right?

`merge` : associativity

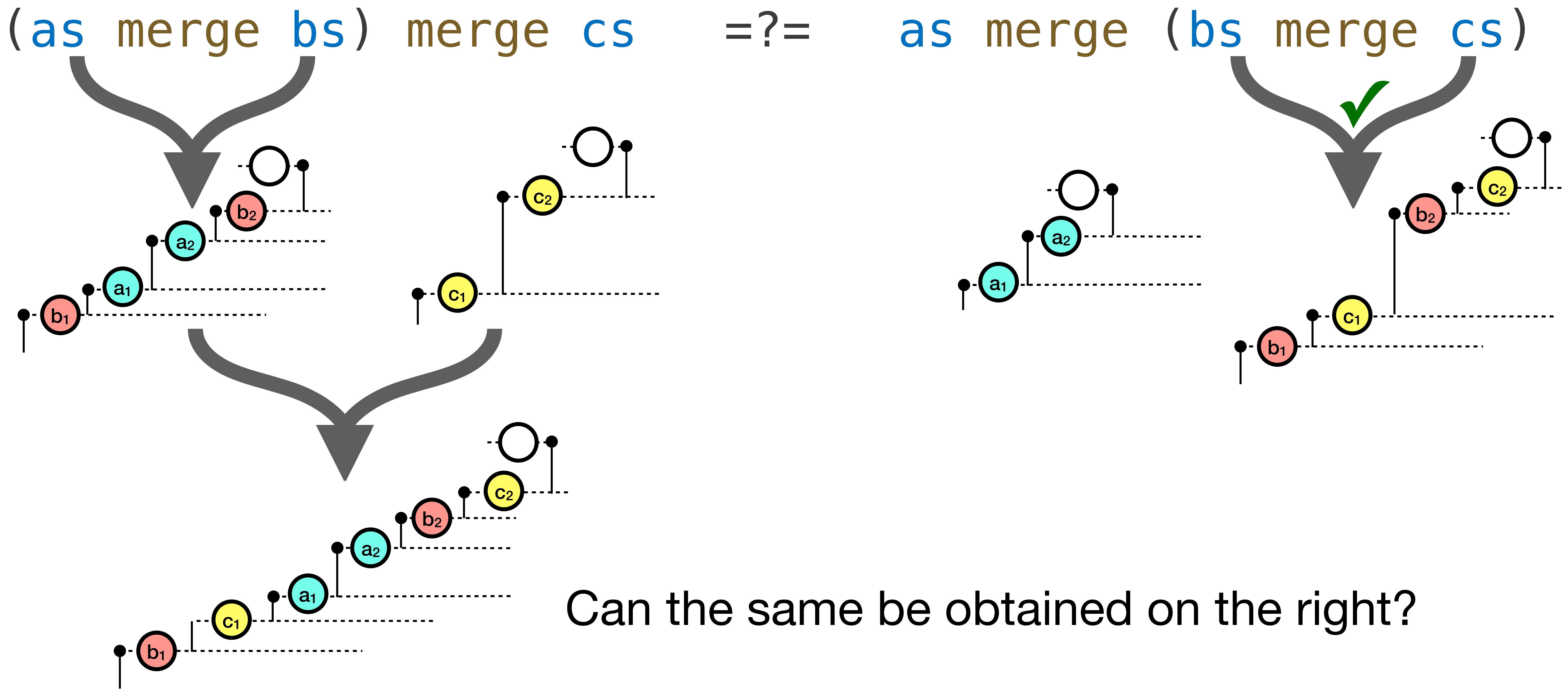
`(as merge bs) merge cs   =?=   as merge (bs merge cs)`





Can the same be obtained on the right?

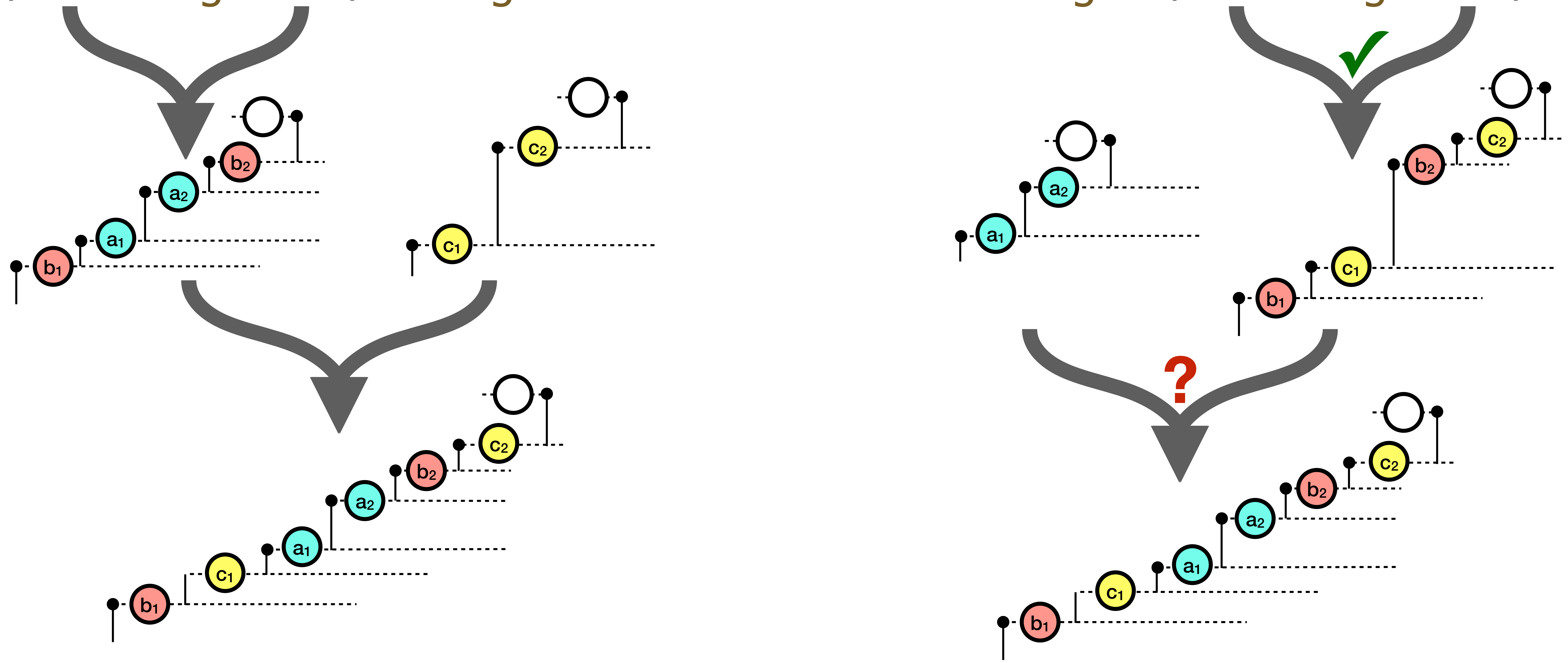
`merge` : associativity



Can the same be obtained on the right?

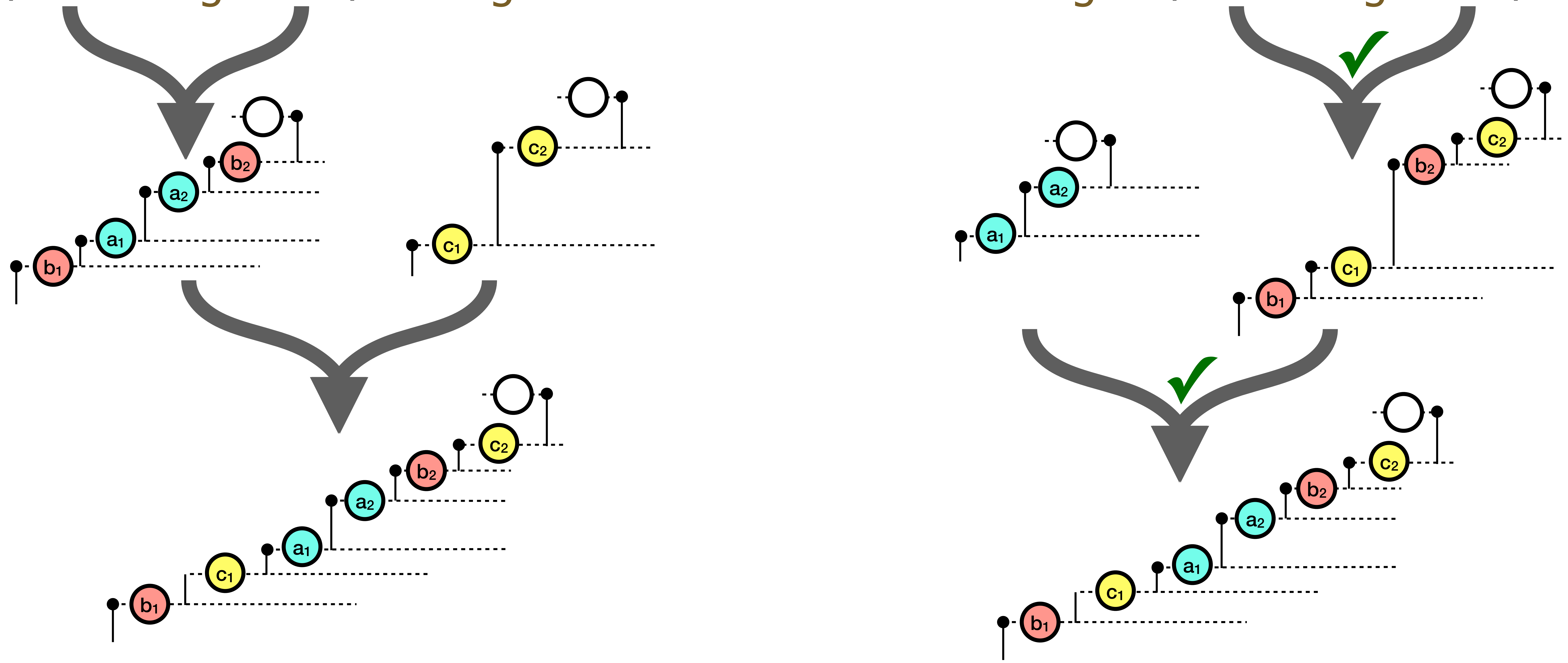
`merge` : associativity

`(as merge bs) merge cs   =?=   as merge (bs merge cs)`



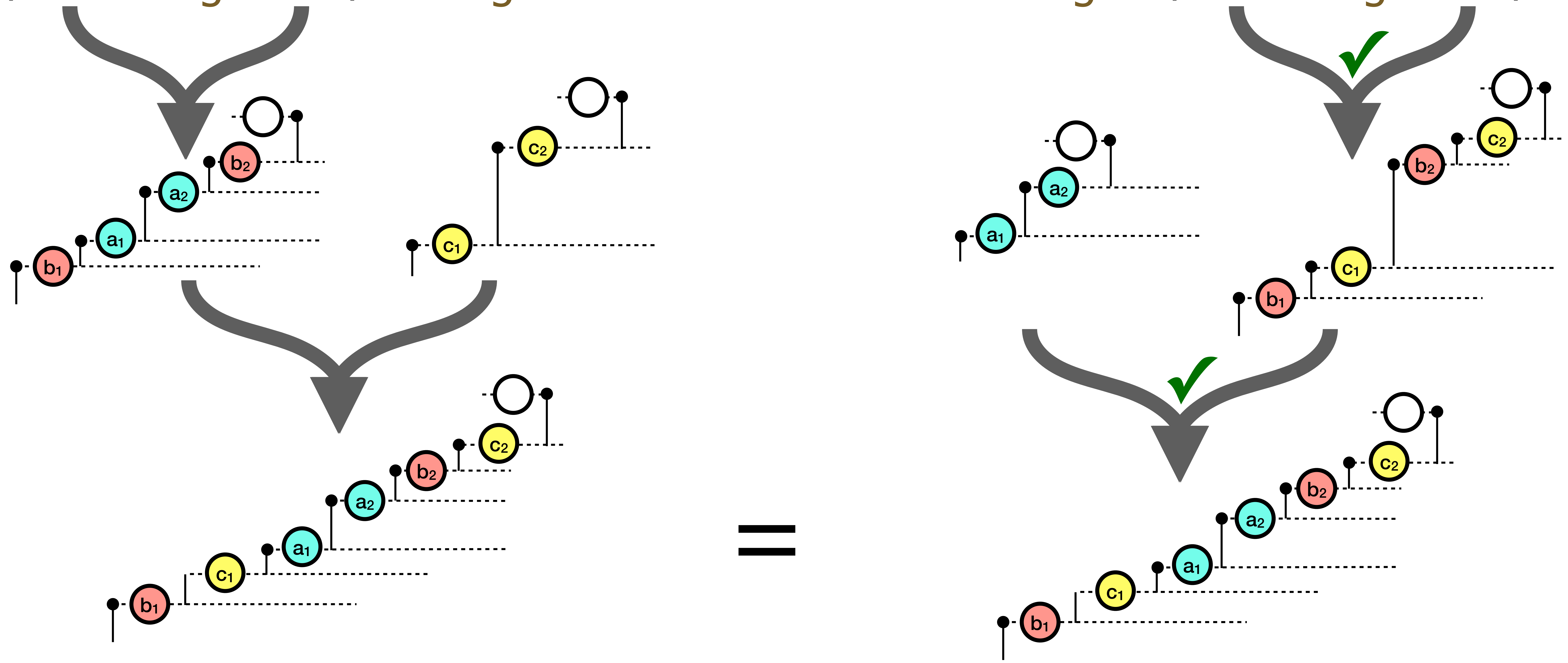
`merge` : associativity

`(as merge bs) merge cs    =?=    as merge (bs merge cs)`



`merge` : associativity

`(as merge bs) merge cs   =?=   as merge (bs merge cs)`



`List` Monoid via `merge`

`List` Monoid via `merge`

✓ `merge` is associative

`List` Monoid via `merge`

✓ `merge` is associative

✓ empty `List` as left and right unit element

`List` Monoid via `merge`

- ✓ `merge` is associative
- ✓ empty `List` as left and right unit element
- ✓ lawful Monoid

Non-deterministic `Writer`

Non-deterministic `Writer`

```
type Writer[A] = List[String] ⊗ A
```

Non-deterministic `Writer`

```
type Writer[A] = List[String] ⊗ A
```

- use the **merging** monoid to combine `Lists`

Non-deterministic `Writer`

```
type Writer[A] = List[String] ⊗ A
```

- use the **merging** monoid to combine `Lists`
- is a **lawful monad**

Non-deterministic `Writer`

```
type Writer[A] = List[String] ⊗ A
```

- use the **merging** monoid to combine `List`s
- is a **lawful monad**

```
List[String] ⊗ (List[String] ⊗ (List[String] ⊗ A))
```

Non-deterministic `Writer`

```
type Writer[A] = List[String] ⊗ A
```

- use the **merging** monoid to combine `List`s
- is a **lawful monad**

```
List[String] ⊗ (List[String] ⊗ (List[String] ⊗ A))
```

`flatten` via ↓ **non-deterministic** `merge`

Non-deterministic `Writer`

```
type Writer[A] = List[String] ⊗ A
```

- use the **merging** monoid to combine `List`s
- is a **lawful monad**

```
List[String] ⊗ (List[String] ⊗ (List[String] ⊗ A))
```

`flatten` via ⬇ **non-deterministic** `merge`

```
List[String] ⊗ A
```

Non-deterministic `Writer`

```
type Writer[A] = List[String] ⊗ A
```

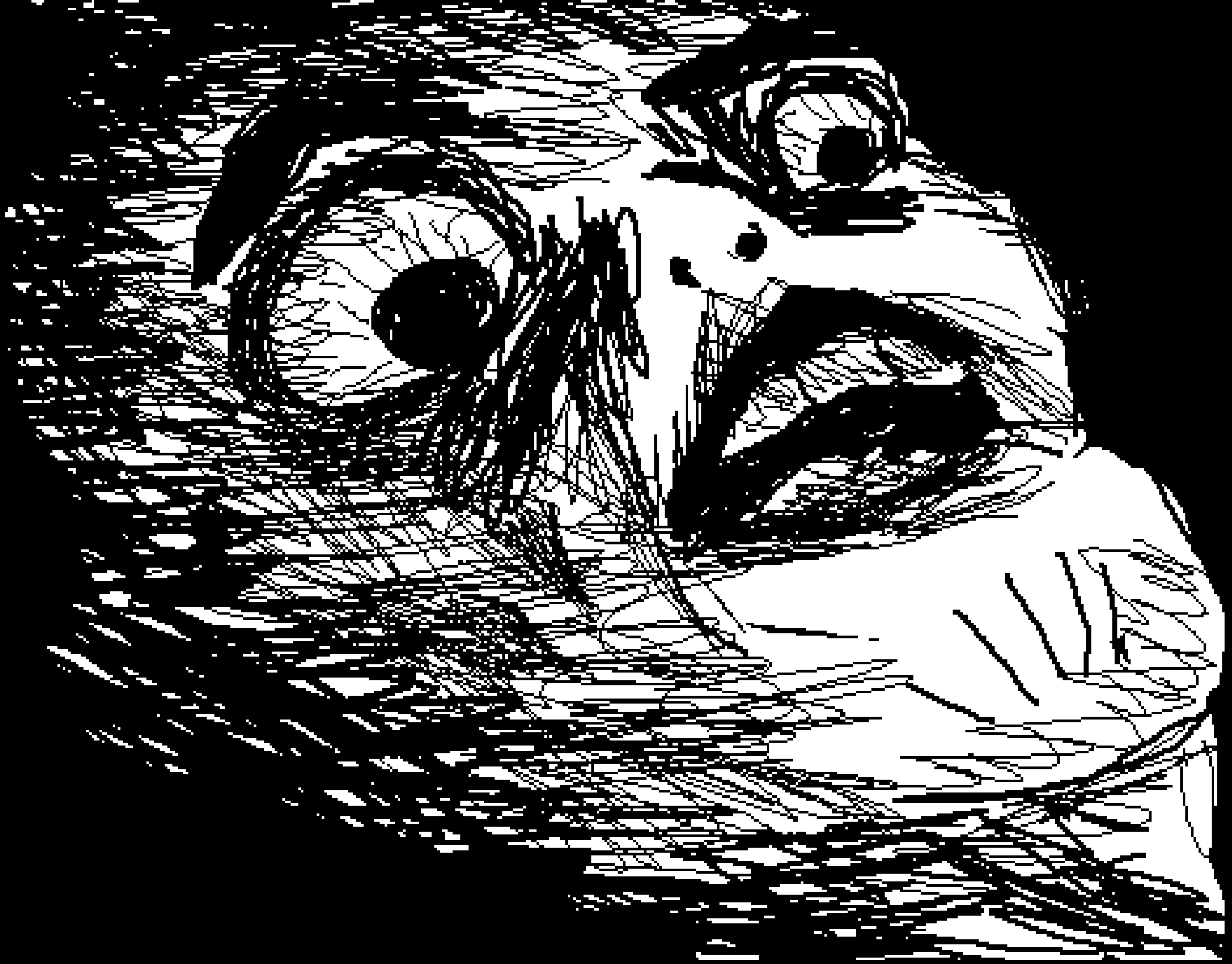
- use the **merging** monoid to combine `Lists`
- is a **lawful monad**

```
List[String] ⊗ (List[String] ⊗ (List[String] ⊗ A))
```

`flatten` via ⬇ **non-deterministic** `merge`

```
List[String] ⊗ A
```

Where's the sequencing?



THERE IS NO SEQUENCING

Lessons

- Monads definable in **any** Category (even non-executable one, like **`<:<`**)
- **Syntactically,** monads *do* support **sequential composition**
- Sequential composition ≠ sequential **execution** (e.g. monads in **`<=`**)
- *“Sequencing of effects”* is **vague,** definable only **tautologically**

Lessons

- Monads definable in **any** Category (even non-executable one, like **<:<**)
- **Syntactically,** monads *do* support **sequential composition**
- Sequential composition ≠ sequential **execution** (e.g. monads in **<=**)
- *"Sequencing of effects"* is **vague,** definable only **tautologically**
- There exist **lawful monads** with **non-deterministic** behavior

Lessons, Rephrased

Lessons, Rephrased

Monads (or their laws) do **not** require

Lessons, Rephrased

Monads (or their laws) do **not** require

- sequential execution

Lessons, Rephrased

Monads (or their laws) do **not** require

- sequential execution
  - or any execution at all

Lessons, Rephrased

Monads (or their laws) do **not** require

- sequential execution
  - or any execution at all
- ordering of effects

Lessons, Rephrased

Monads (or their laws) do **not** require

- sequential execution
  - or any execution at all
- ordering of effects
- determinism

Closing Remarks

Closing Remarks

- If not from monads, **where does** observed **sequencing come from?**

Closing Remarks

- If not from monads, **where does** observed **sequencing come from?**
  - Hint: Function application & evaluation strategy of the language

Closing Remarks

- If not from monads, **where does** observed **sequencing come from?**
    - Hint: Function application & evaluation strategy of the language
- **What *are* monads about?**

Closing Remarks

- If not from monads, **where does** observed **sequencing come from?**
  - Hint: Function application & evaluation strategy of the language
- **What *are* monads about?**
  - Flattening: Simplifying an arbitrarily nested structure to a single level.

Closing Remarks

- If not from monads, **where does** observed **sequencing come from?**
    - Hint: Function application & evaluation strategy of the language
- **What *are* monads about?**
    - Flattening: Simplifying an arbitrarily nested structure to a single level.
- *“Monads are about sequencing”* might have been a **useful crutch**

Closing Remarks

- If not from monads, **where does** observed **sequencing come from?**
  - Hint: Function application & evaluation strategy of the language
- **What *are* monads about?**
  - Flattening: Simplifying an arbitrarily nested structure to a single level.
- *“Monads are about sequencing”* might have been a **useful crutch**
  - Ultimately better off without crutches

Closing Remarks

- If not from monads, **where does** observed **sequencing come from?**
  - Hint: Function application & evaluation strategy of the language
- **What *are* monads about?**
  - Flattening: Simplifying an arbitrarily nested structure to a single level.
- *"Monads are about sequencing"* might have been a **useful crutch**
  - Ultimately better off without crutches
- **What else are we wrong about?**

Thank You!

Scala examples: https://github.com/TomasMikula/non-sequencing-monads/

Libretto: https://github.com/TomasMikula/libretto/