# Concurrent All The Way Down

## Functional Concurrency with Libretto

Tomas Mikula

10 YEARS lambda DAYS
5-6 JUNE 2023 KRAKÓW | POLAND

# **Functional** Concurrency with Libretto

# Functional Programming

# Functional Programming

👍 **Function Composition** 👍

# Functional Programming

## 👍 Function Composition 👍

- Input/output types as the only interface
- No hidden communication between functions

# Functional Programming

## 👍 Function Composition 👍

- Input/output types as the only interface
- No hidden communication between functions

## 👎 Side-Effects 👎

# Functional Programming

## 👍 Function Composition 👍

- Input/output types as the only interface
- No hidden communication between functions

## 👎 Side-Effects 👎

- Spooky action at a distance
- Erode local reasoning

# *Concurrent* Functional Programming

# *Concurrent* Functional Programming

- Start a bunch of **sequential** processes

# *Concurrent* Functional Programming

- Start a bunch of **sequential** processes

  (threads / actors / fibers / virtual threads / green threads)

# *Concurrent* Functional Programming

- Start a bunch of **sequential** processes

  (threads / actors / fibers / virtual threads / green threads)

- Let them communicate via **side-effects**

# *Concurrent* Functional Programming

- Start a bunch of **sequential** processes

  (threads / actors / fibers / virtual threads / green threads)

- Let them communicate via **side-effects**

  (shared mutable state, message passing, …)

# *Concurrent* Functional Programming

- Start a bunch of **sequential** processes

  (threads / actors / fibers / virtual threads / green threads)

- Let them communicate via **side-effects**

  (shared mutable state, message passing, …)

**Let that sink in …**

# *Concurrent* Functional Programming

- Start a bunch of **sequential** processes

  (threads / actors / fibers / virtual threads / green threads)

- Let them communicate via **side-effects**

  (shared mutable state, message passing, …)

**Let that sink in …**

**Functional**

# *Concurrent* Functional Programming

- Start a bunch of **sequential** processes

  (threads / actors / fibers / virtual threads / green threads)

- Let them communicate via **side-effects**

  (shared mutable state, message passing, …)

**Let that sink in …**

**Functional   concurrency**

# *Concurrent* Functional Programming

- Start a bunch of **sequential** processes

  (threads / actors / fibers / virtual threads / green threads)

- Let them communicate via **side-effects**

  (shared mutable state, message passing, …)

**Let that sink in …**

**Functional concurrency**

built on

**side-effects**

# *Concurrent* Functional Programming

- Start a bunch of **sequential** processes

  (threads / actors / fibers / virtual threads / green threads)

- Let them communicate via **side-effects**

  (shared mutable state, message passing, …)

**Let that sink in …**

**Functional concurrency**

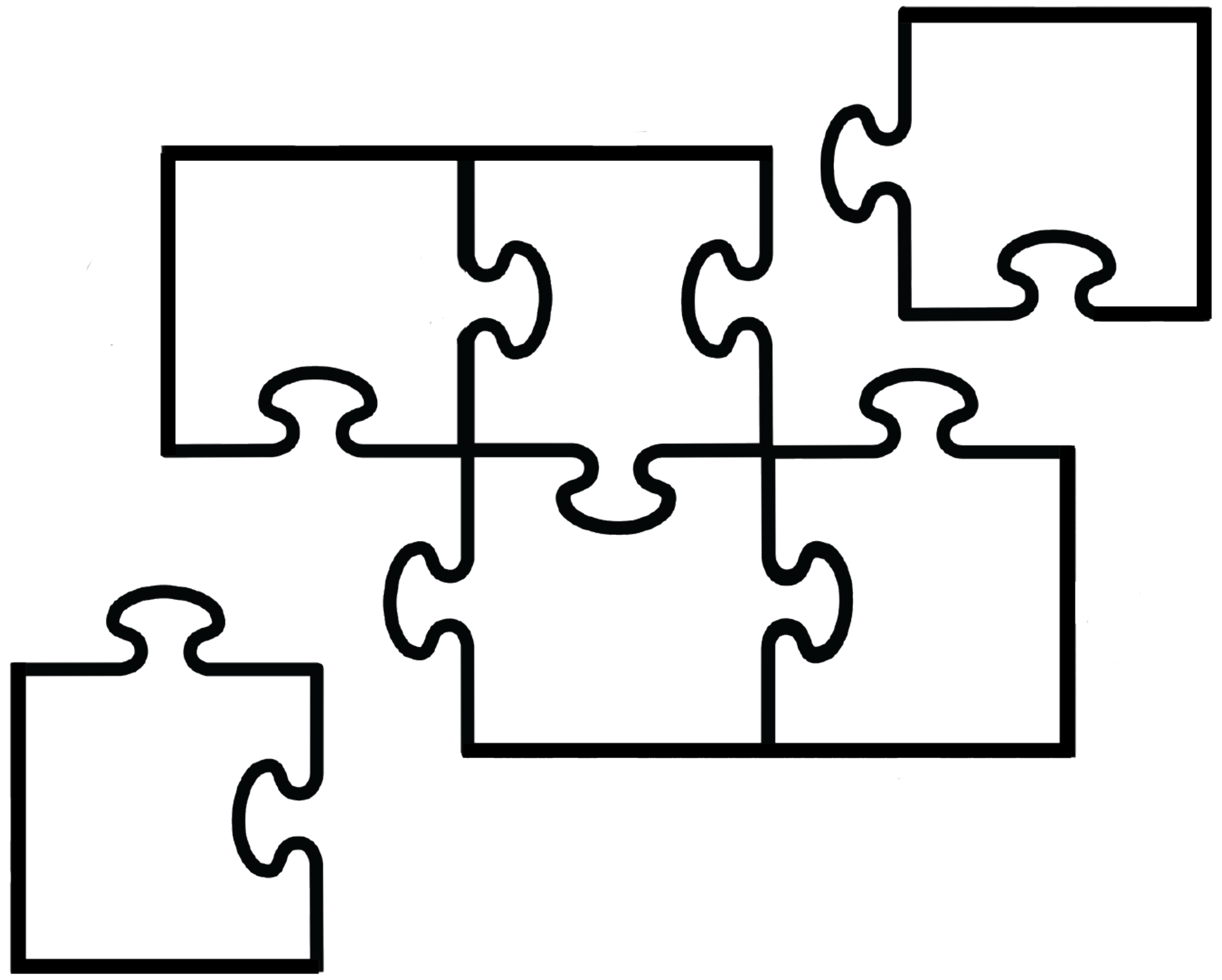
built on

**side-effects** **sequential** processes

**?**

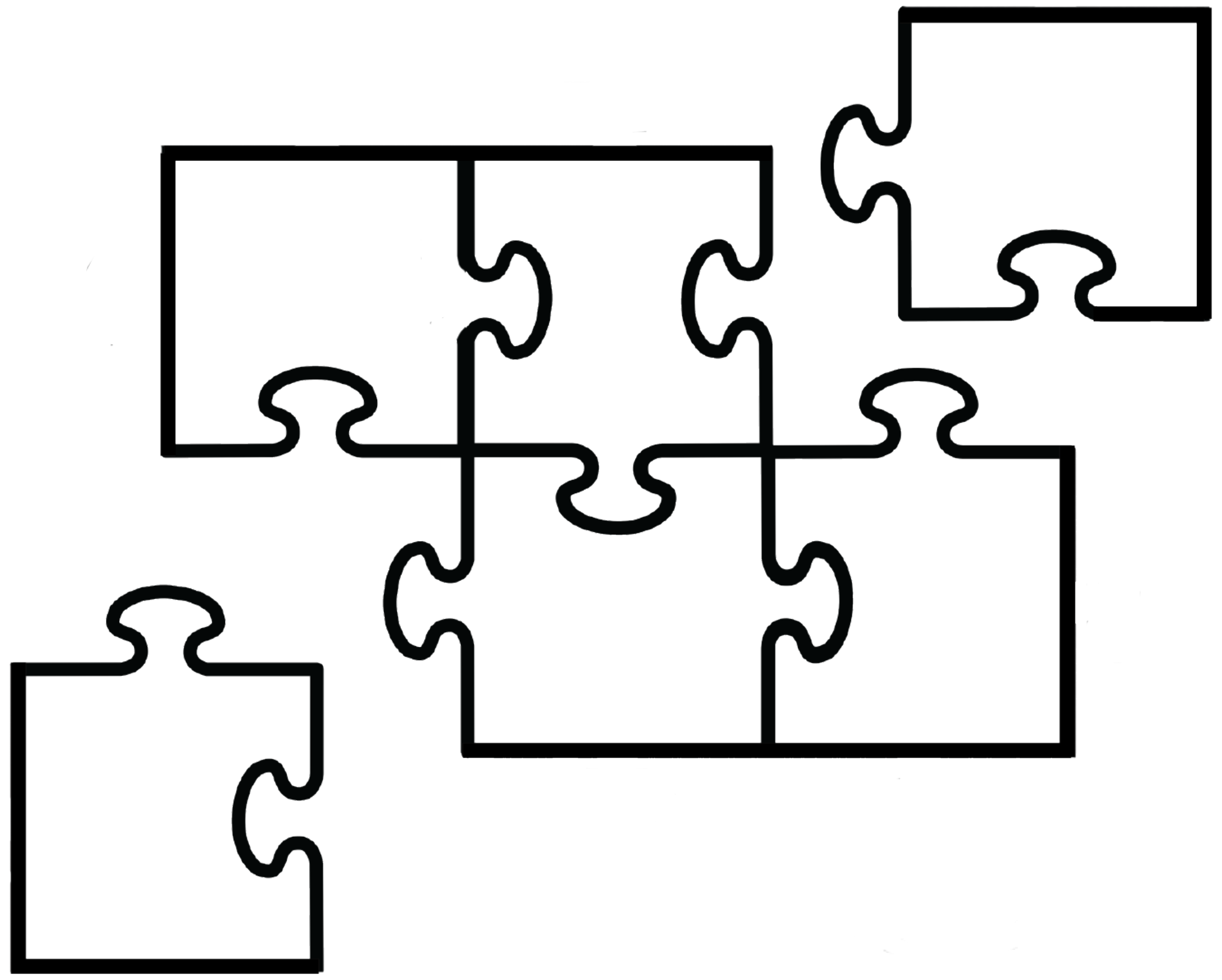
# Composing

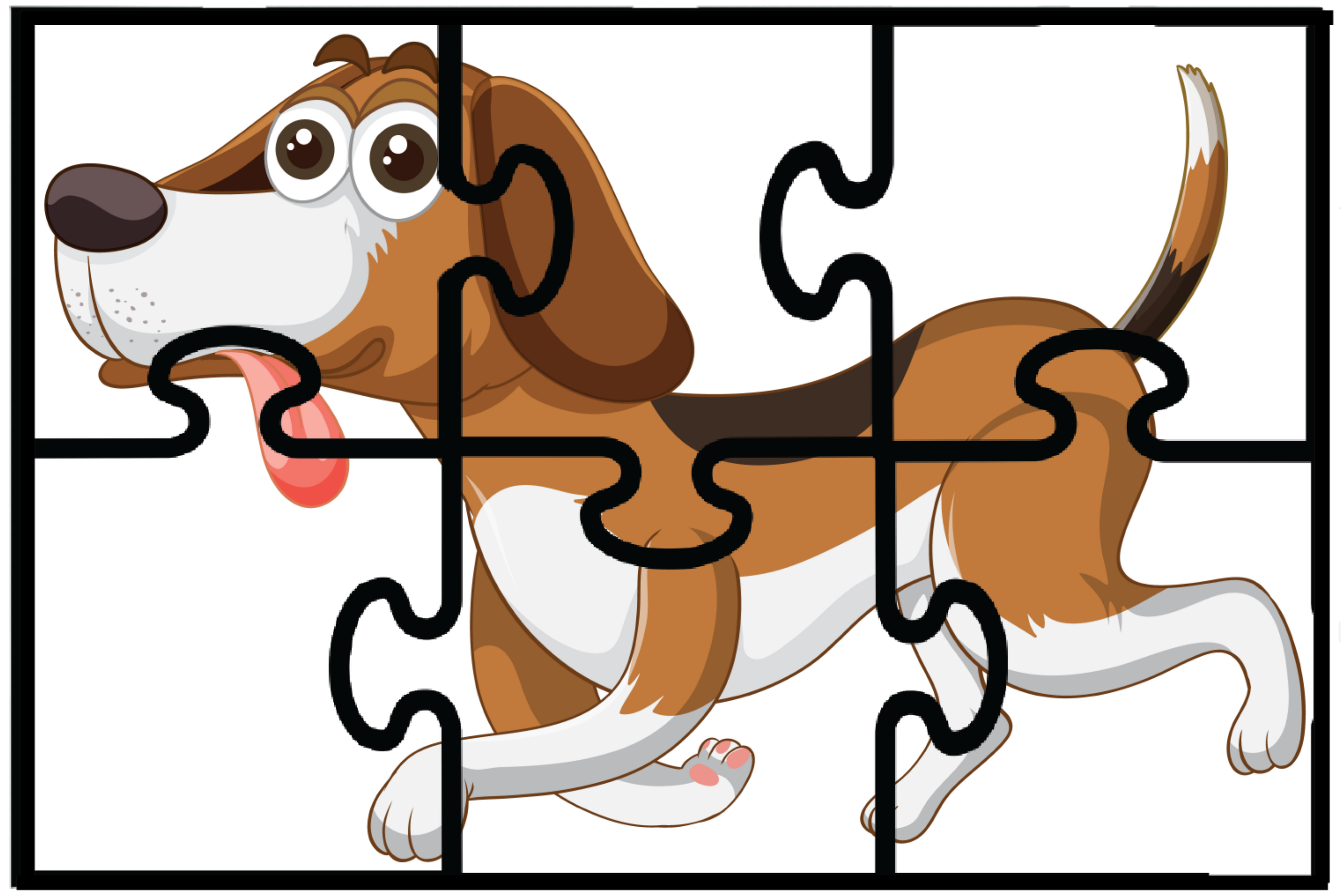
Functions



# Composing

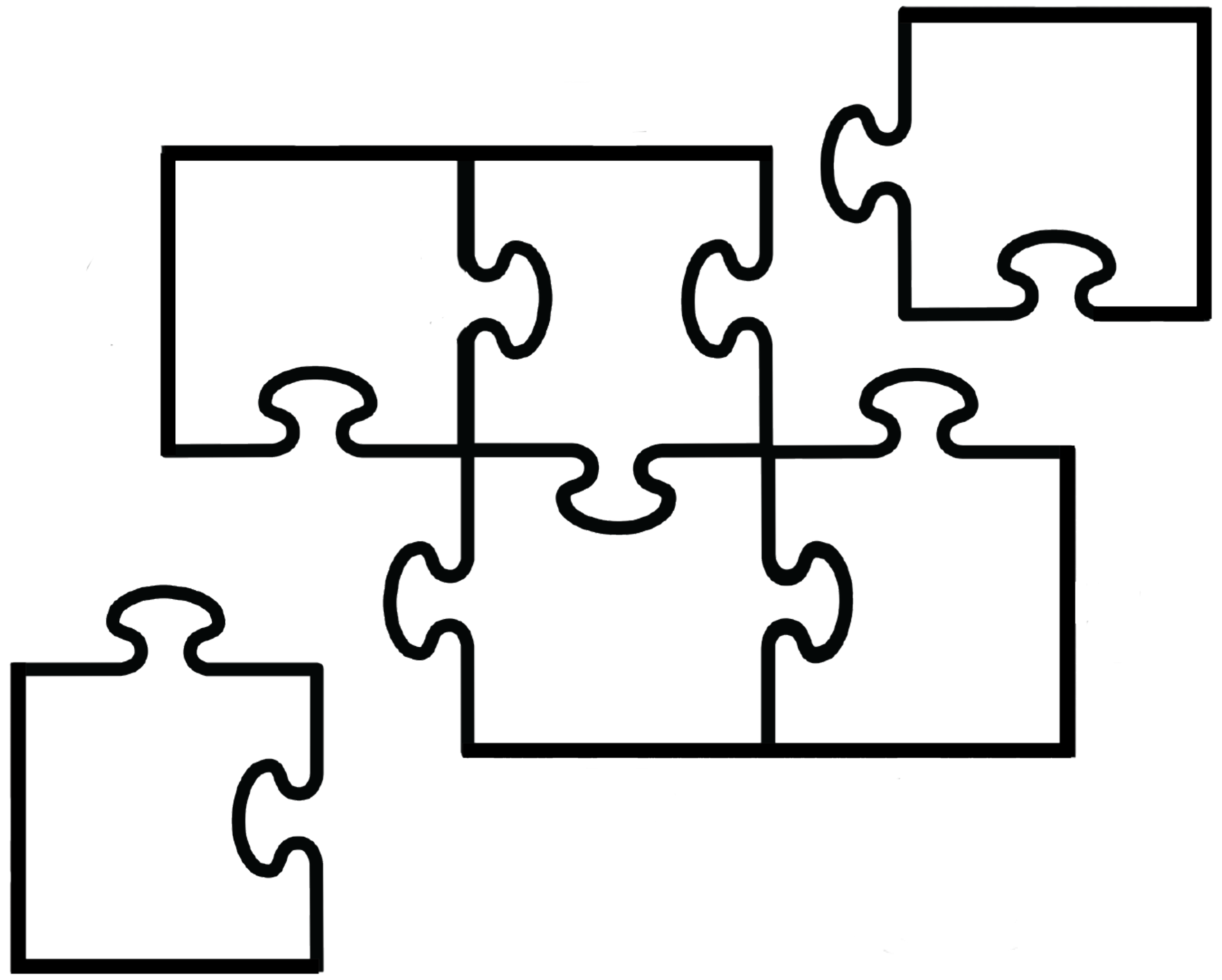
Functions

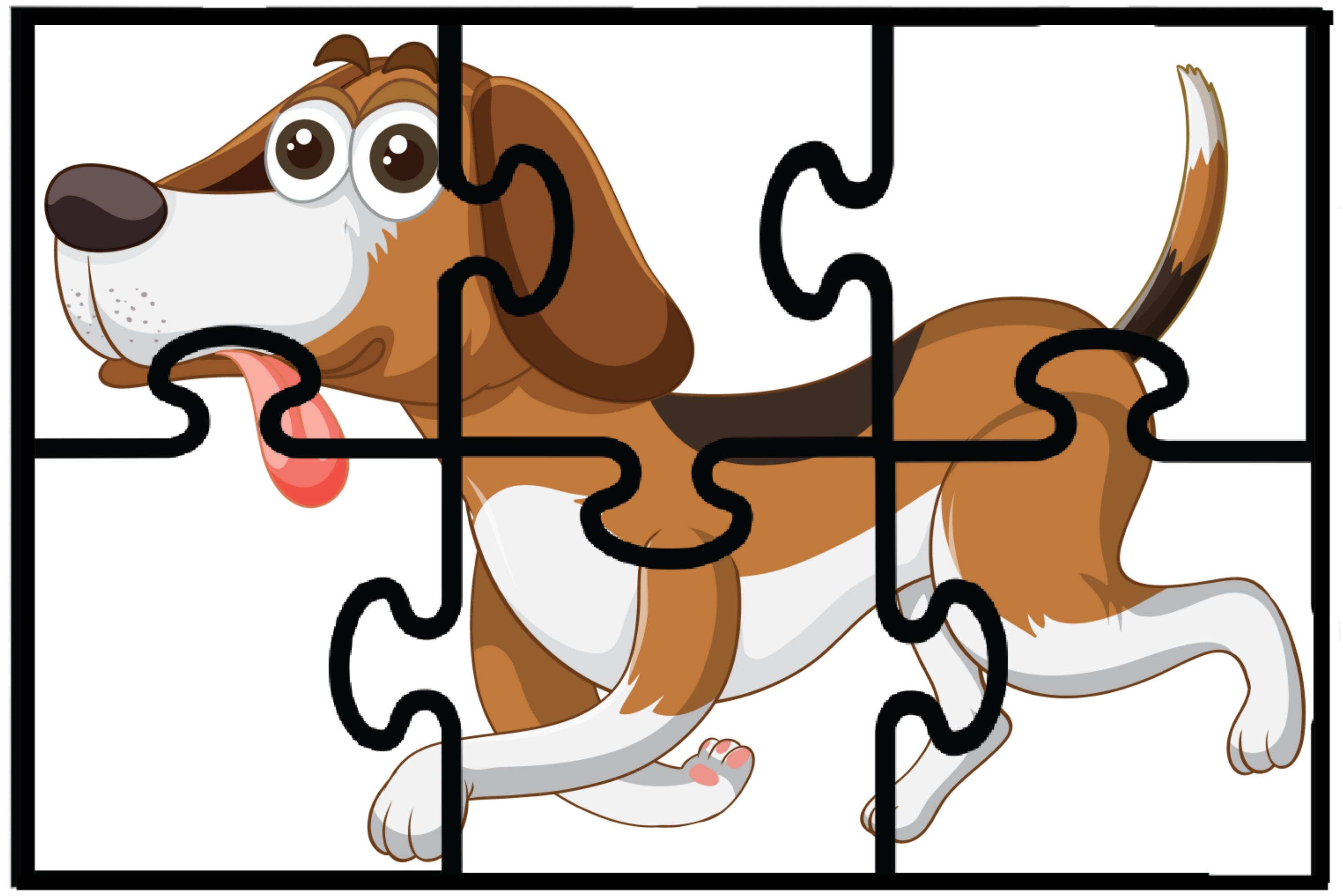




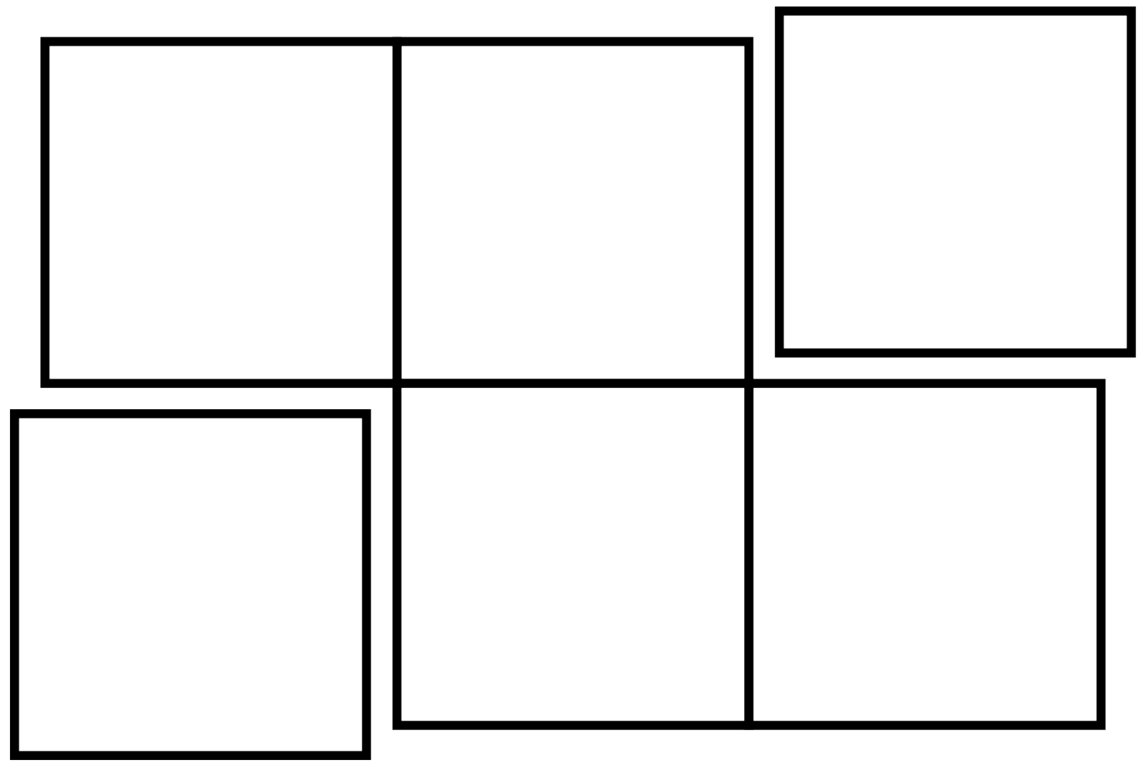
# Composing

Functions



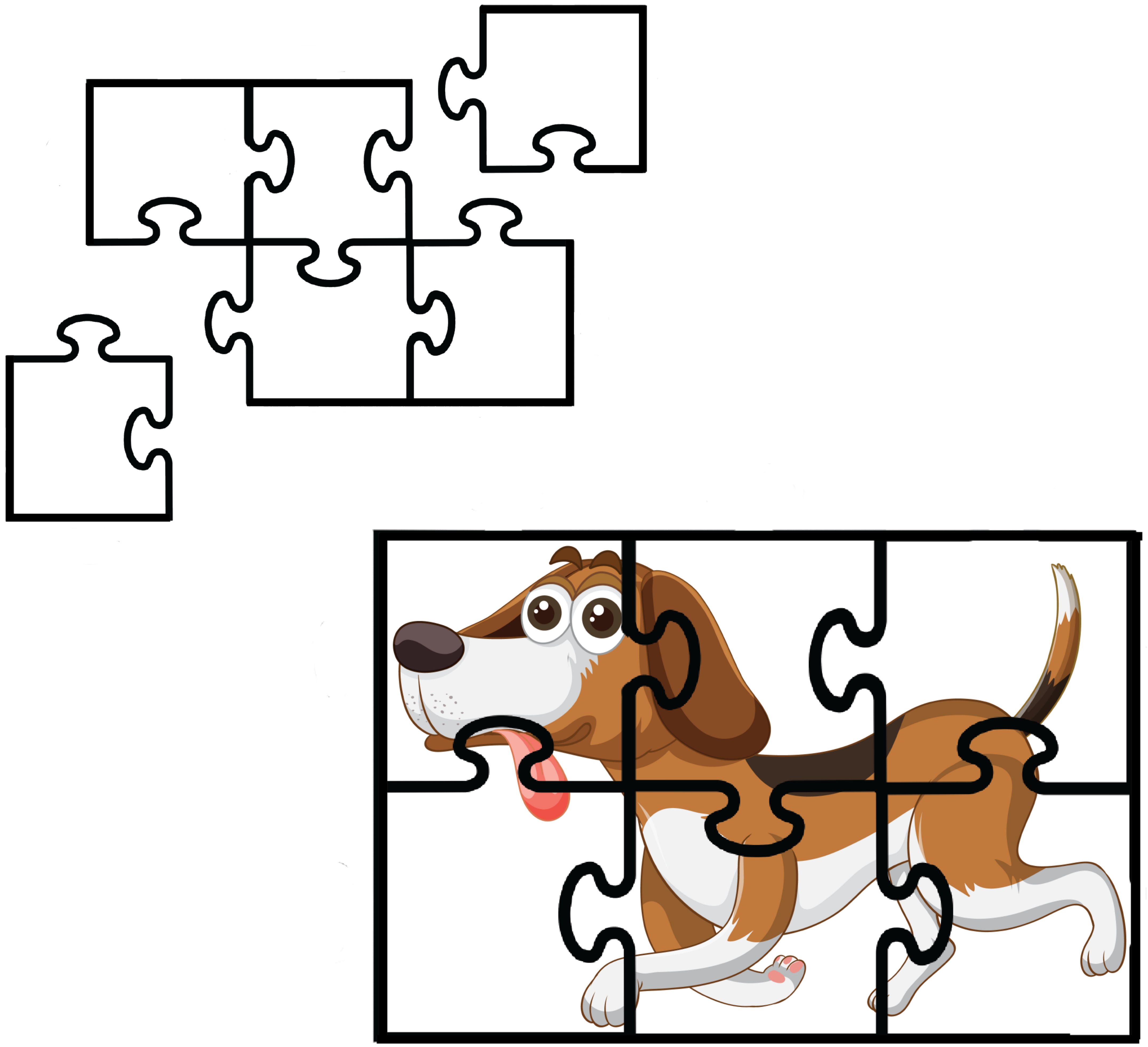


Threads

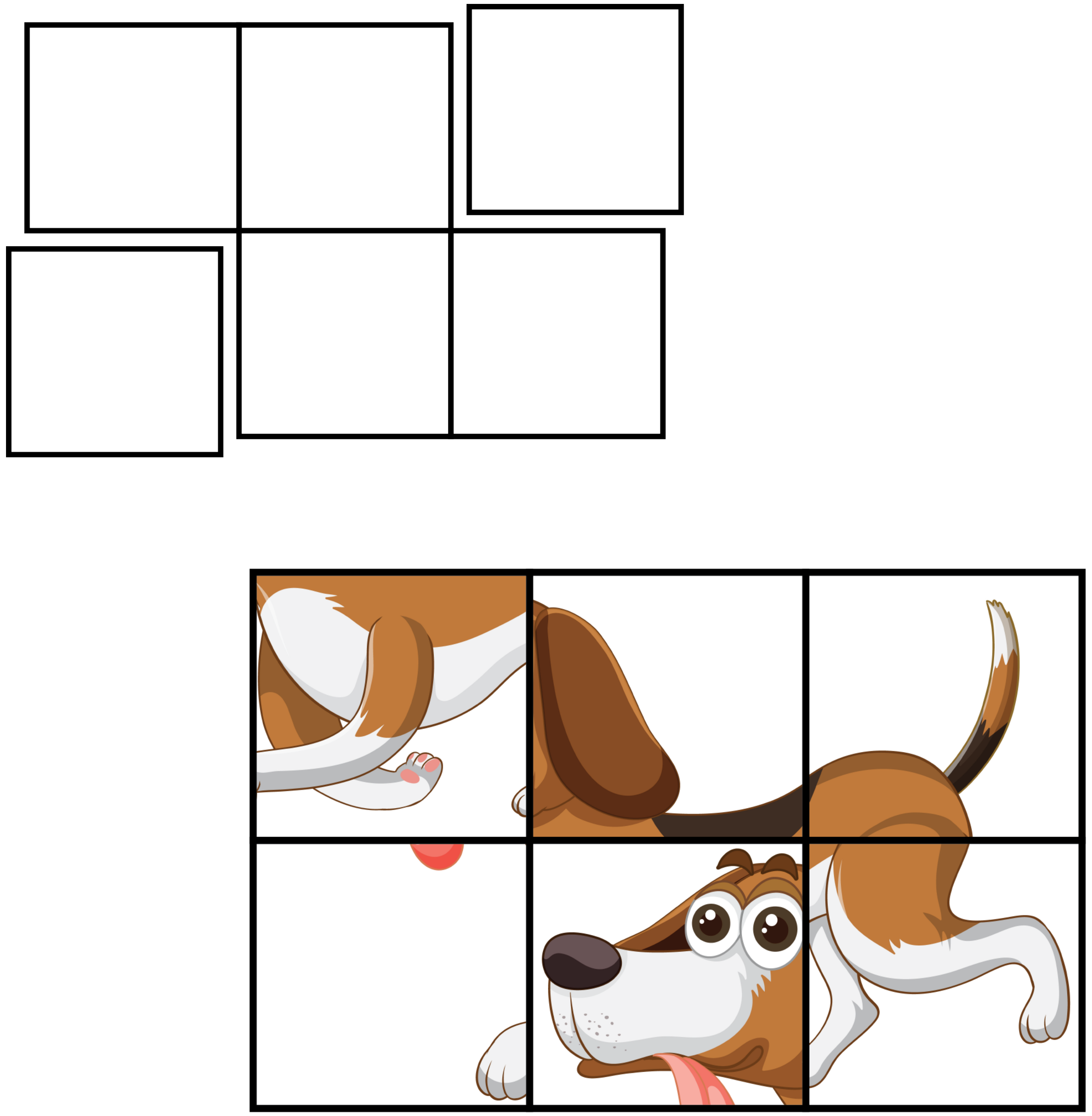


# Composing

## Functions



## Threads



We still don’t know how to do

**Concurrent Functional Programming**

We still don’t know how to do

**Concurrent Functional Programming**

Let’s keep trying!

# Goals

- **Compose** concurrent programs **like** we compose pure **functions**
- **No** reliance on **side-effects**
- **No** manual **thread** management
    - implicit concurrency
    - causal dependence as the only form of sequencing

- **Compose** concurrent programs **like** we compose pure **functions**
- **No** reliance on **side-effects**
- **No** manual **thread** management
    - implicit concurrency
    - causal dependence as the only form of sequencing

# Libretto

- concurrency DSL embedded in Scala
- **Compose** concurrent programs **like** we compose pure **functions**
- **No** reliance on **side-effects**
- **No** manual **thread** management
  - implicit concurrency
  - causal dependence as the only form of sequencing

# Agenda

1. A taste of Libretto
2. Santa Claus problem

# List in Libretto

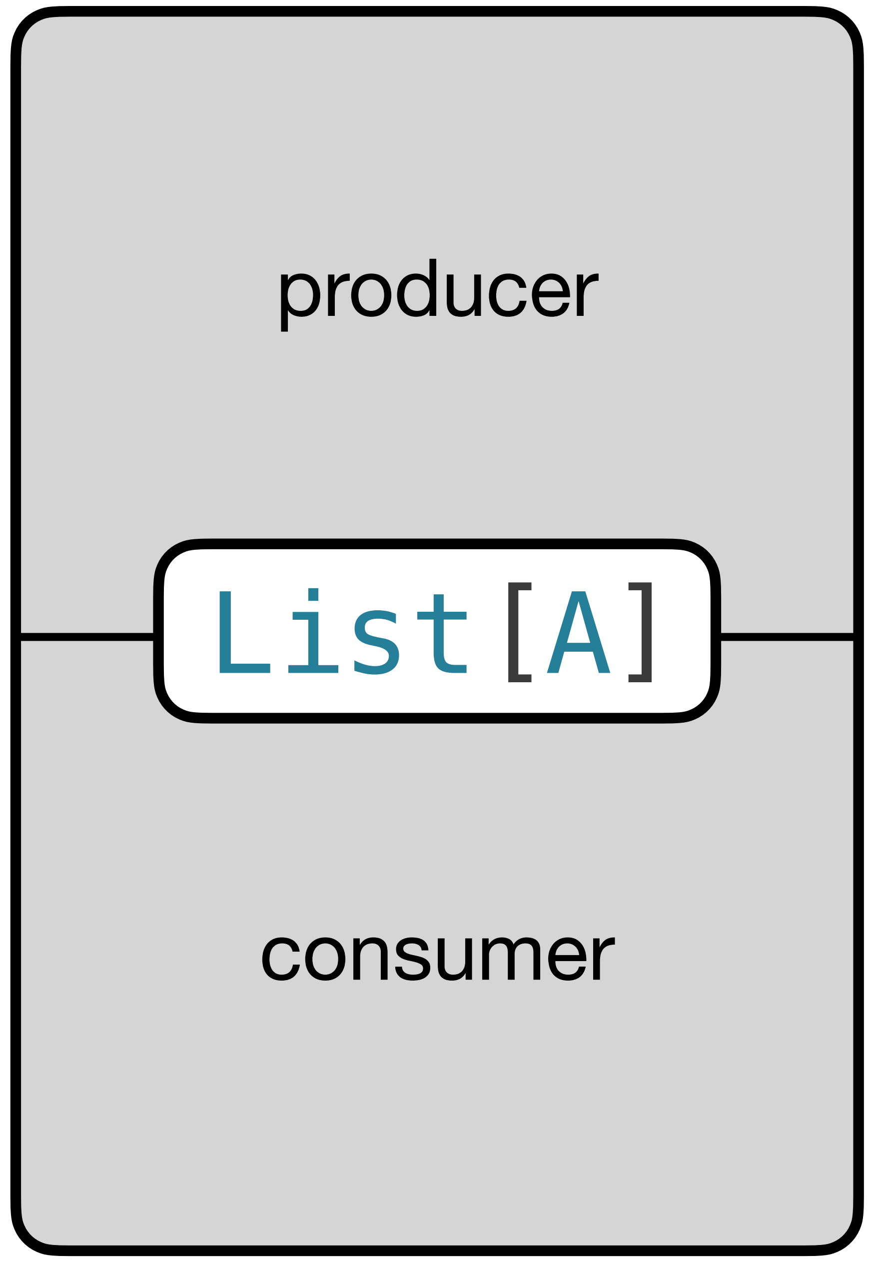
```
List[A] = One ⊕ (A ⊗ List[A])
```

# **List** in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```

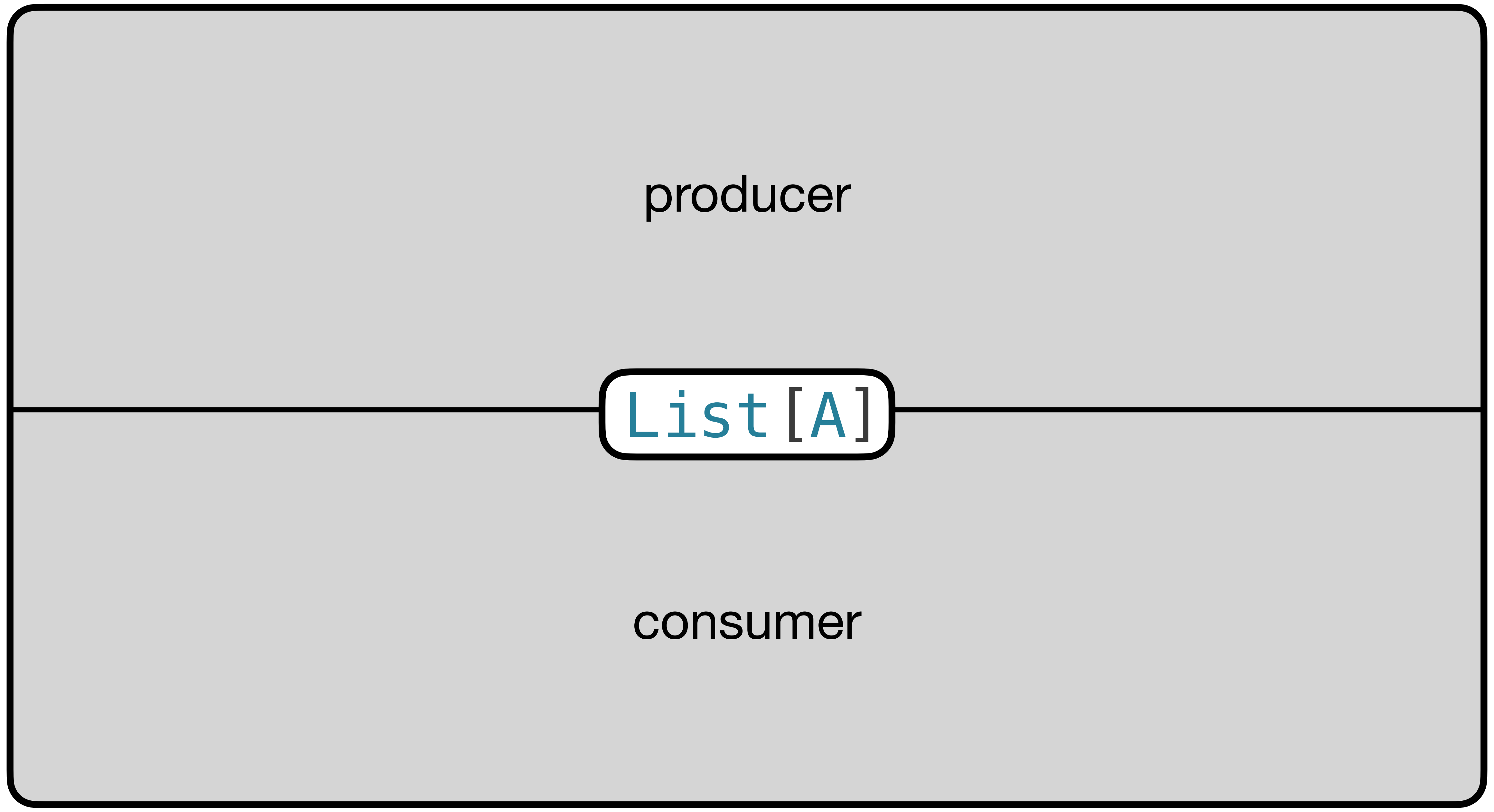
- One: empty
- ⊕: producer choice
- ⊗: concurrent pair
- (A ⊗ List[A]): non-empty

- Type is an **interface of interaction** between producer and consumer
- Producer decides
  - **how many** elements there are
  - **when** does each element become available



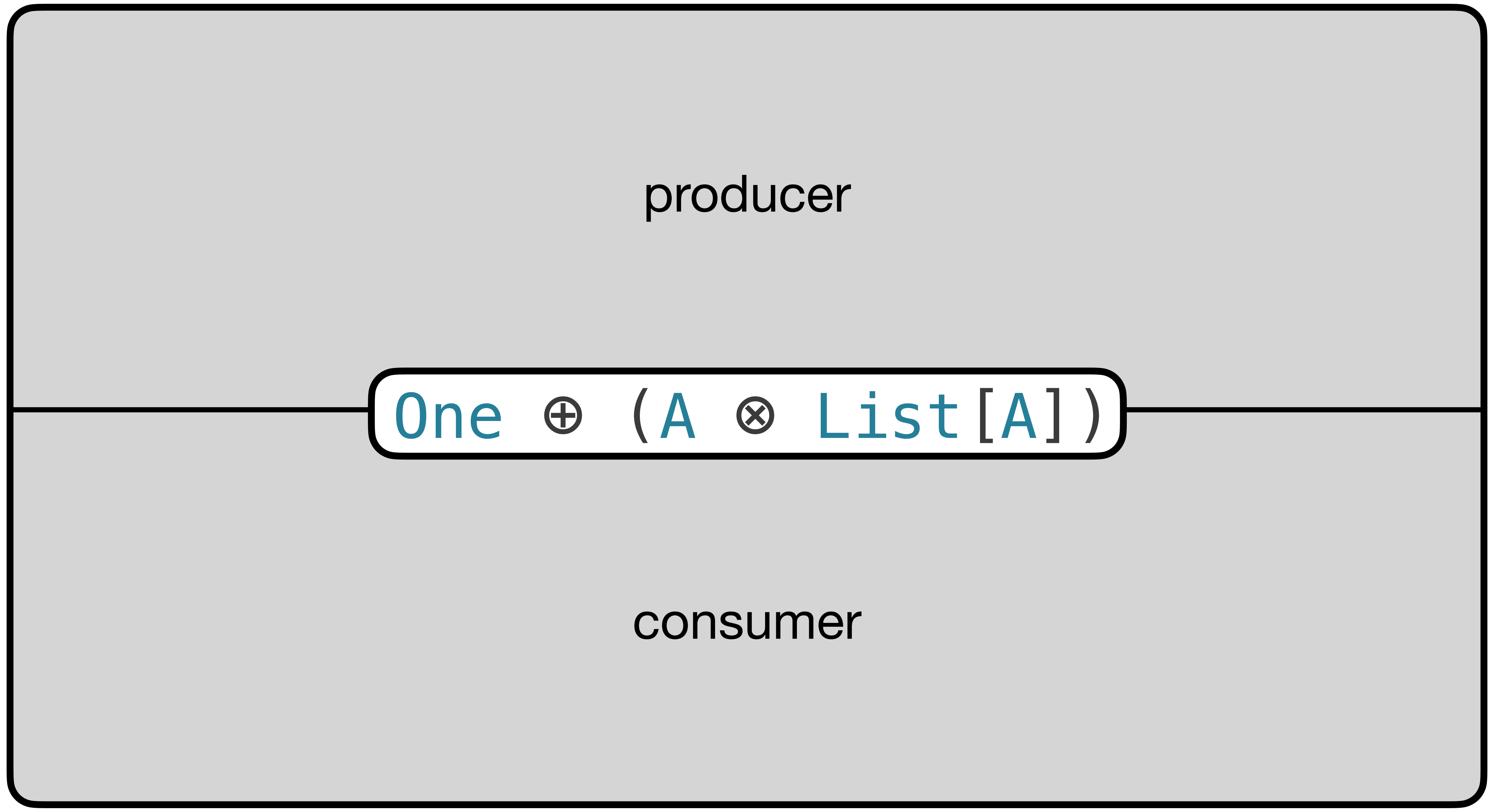
# List in Libretto

$$\texttt{List[A]} = \texttt{One} \oplus (\texttt{A} \otimes \texttt{List[A]})$$



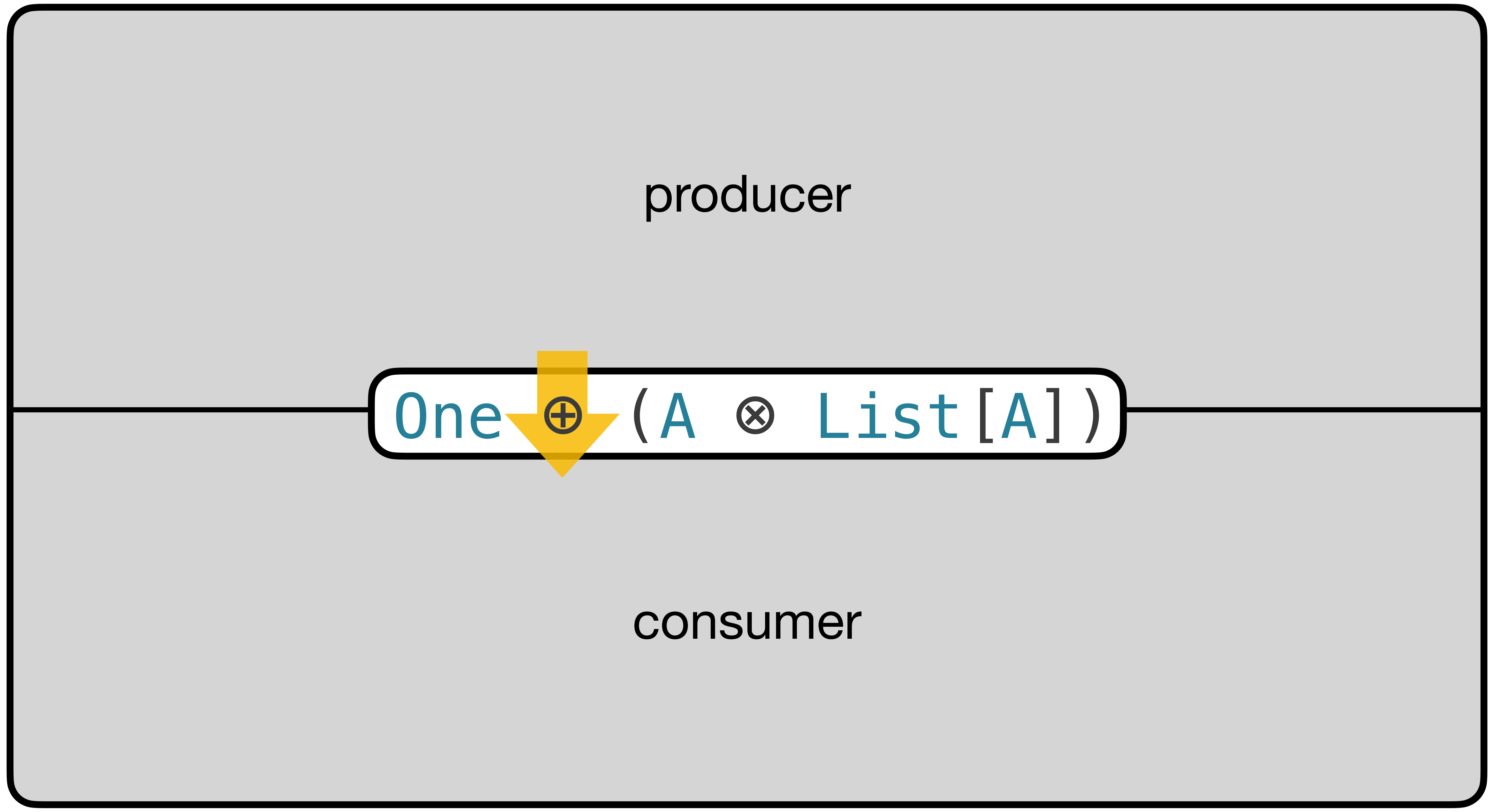
# List in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```



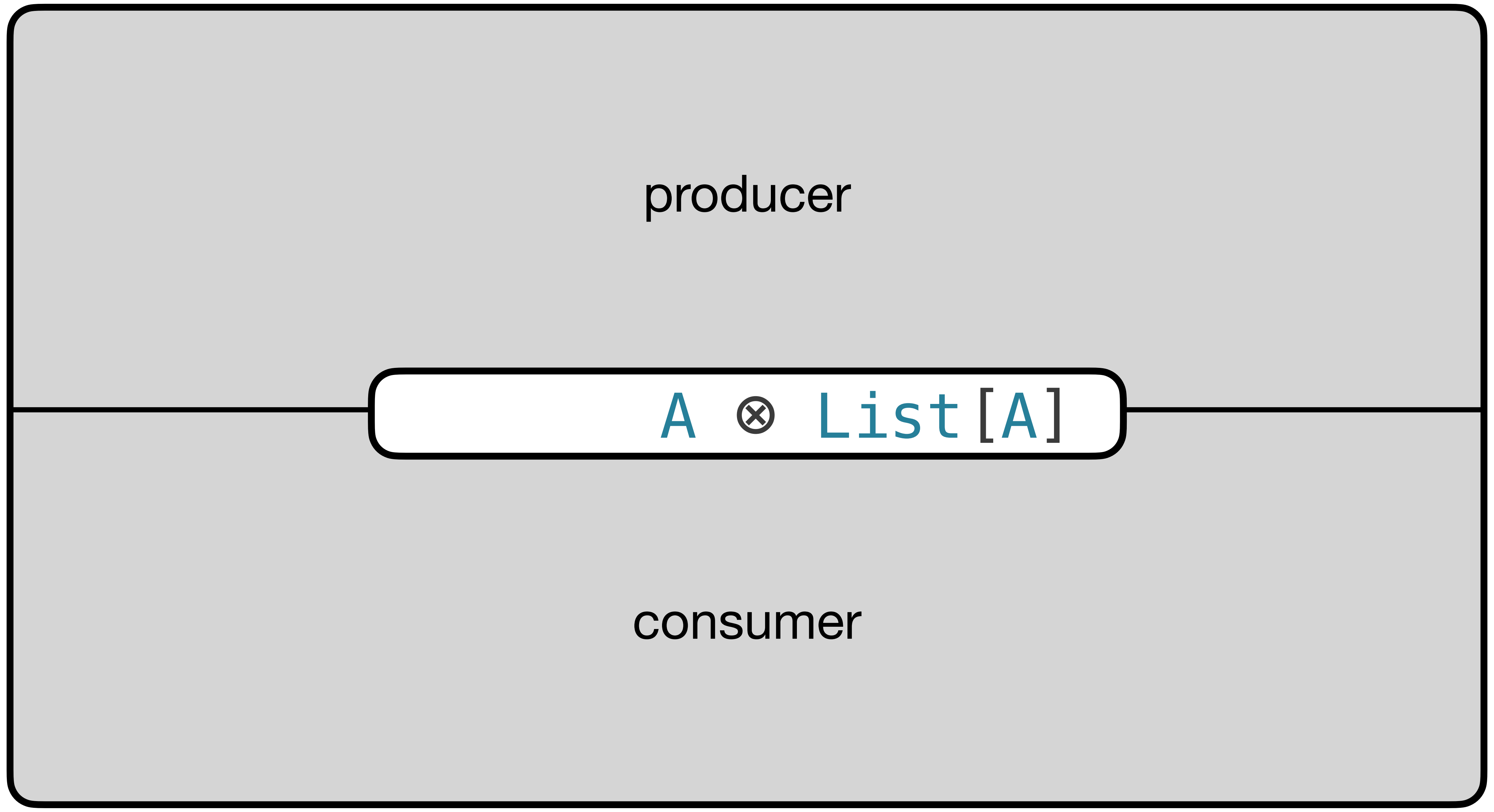
# List in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```



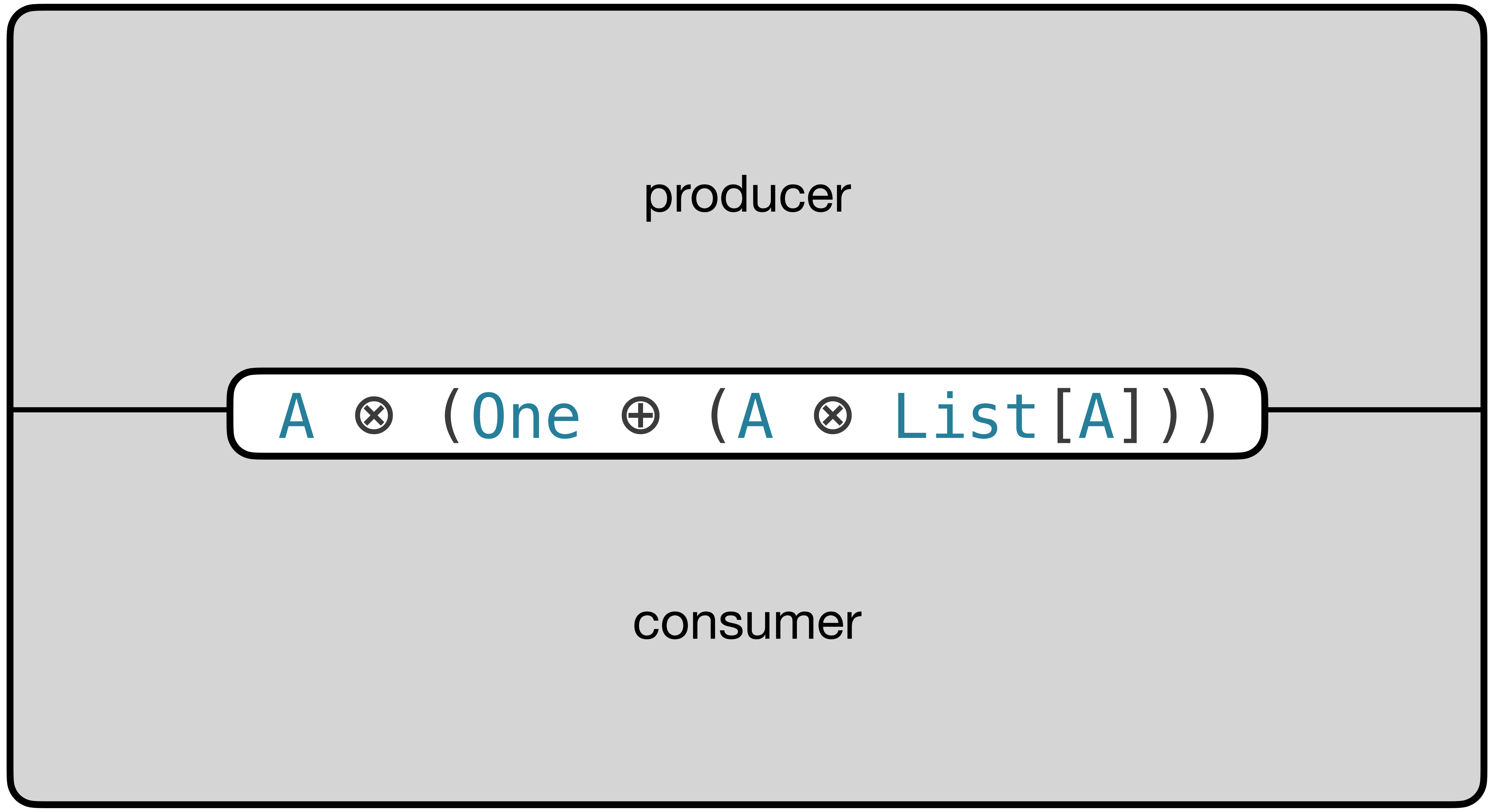
# List in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```



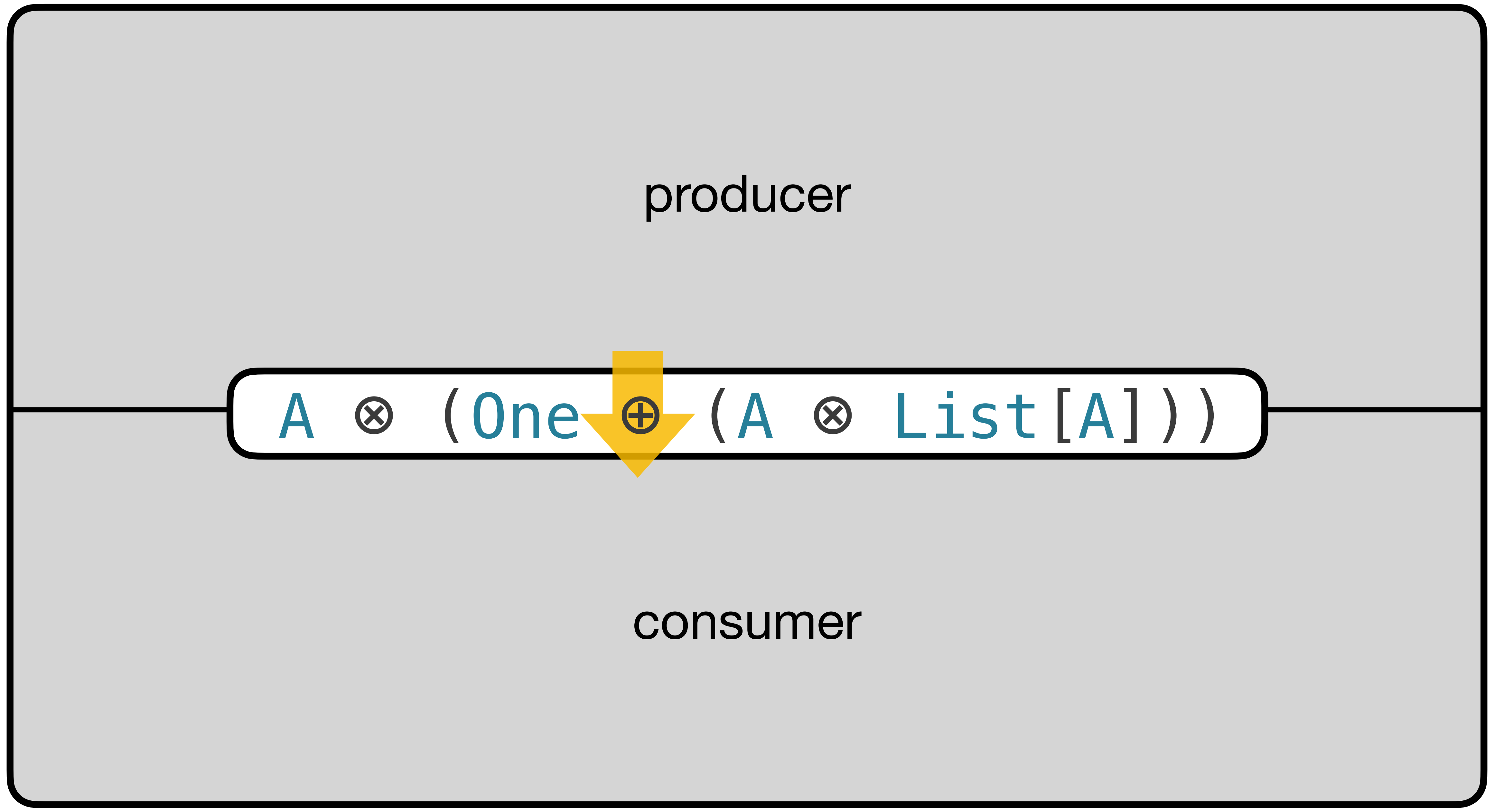
# List in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```



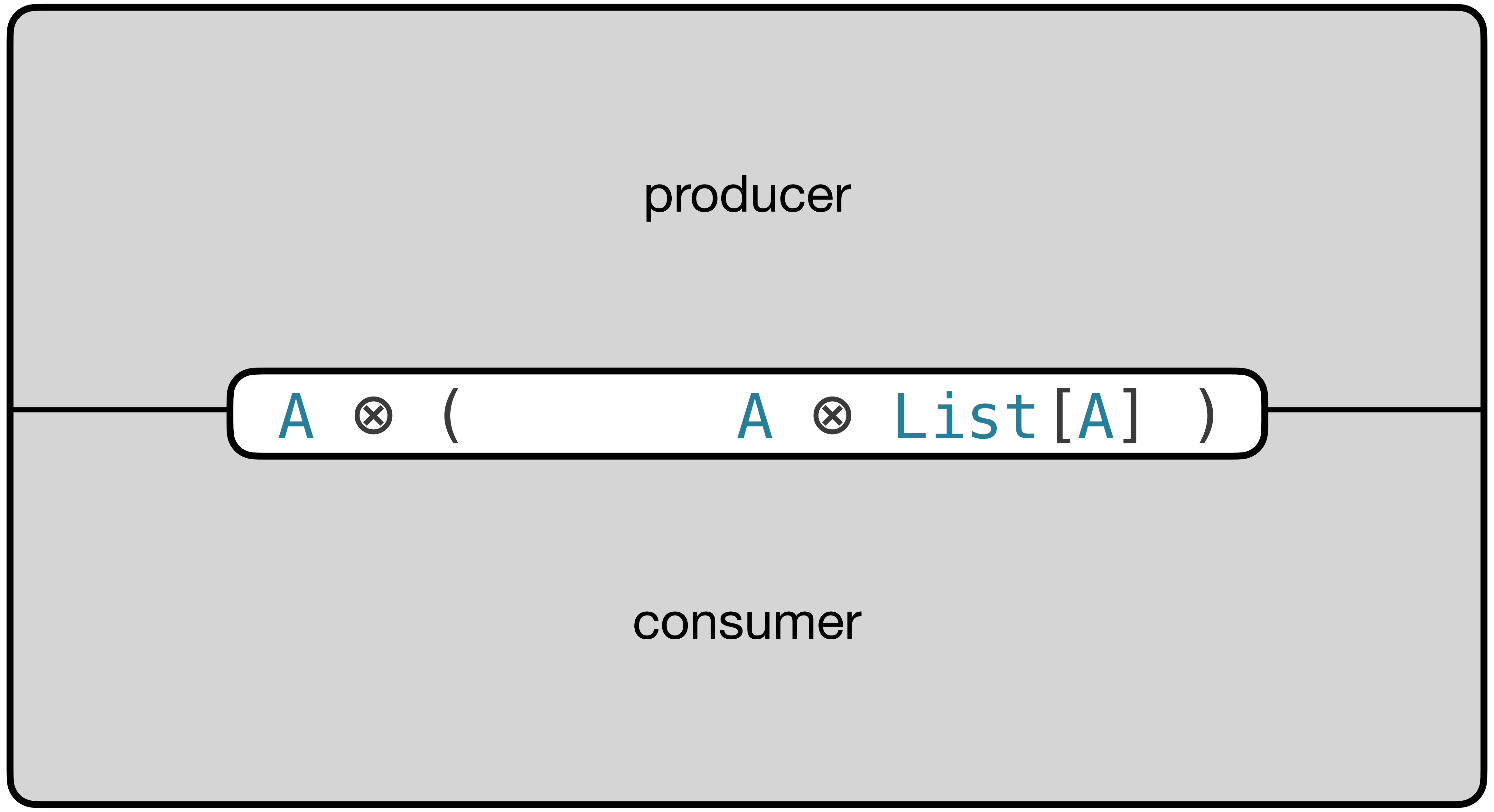
# List in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```



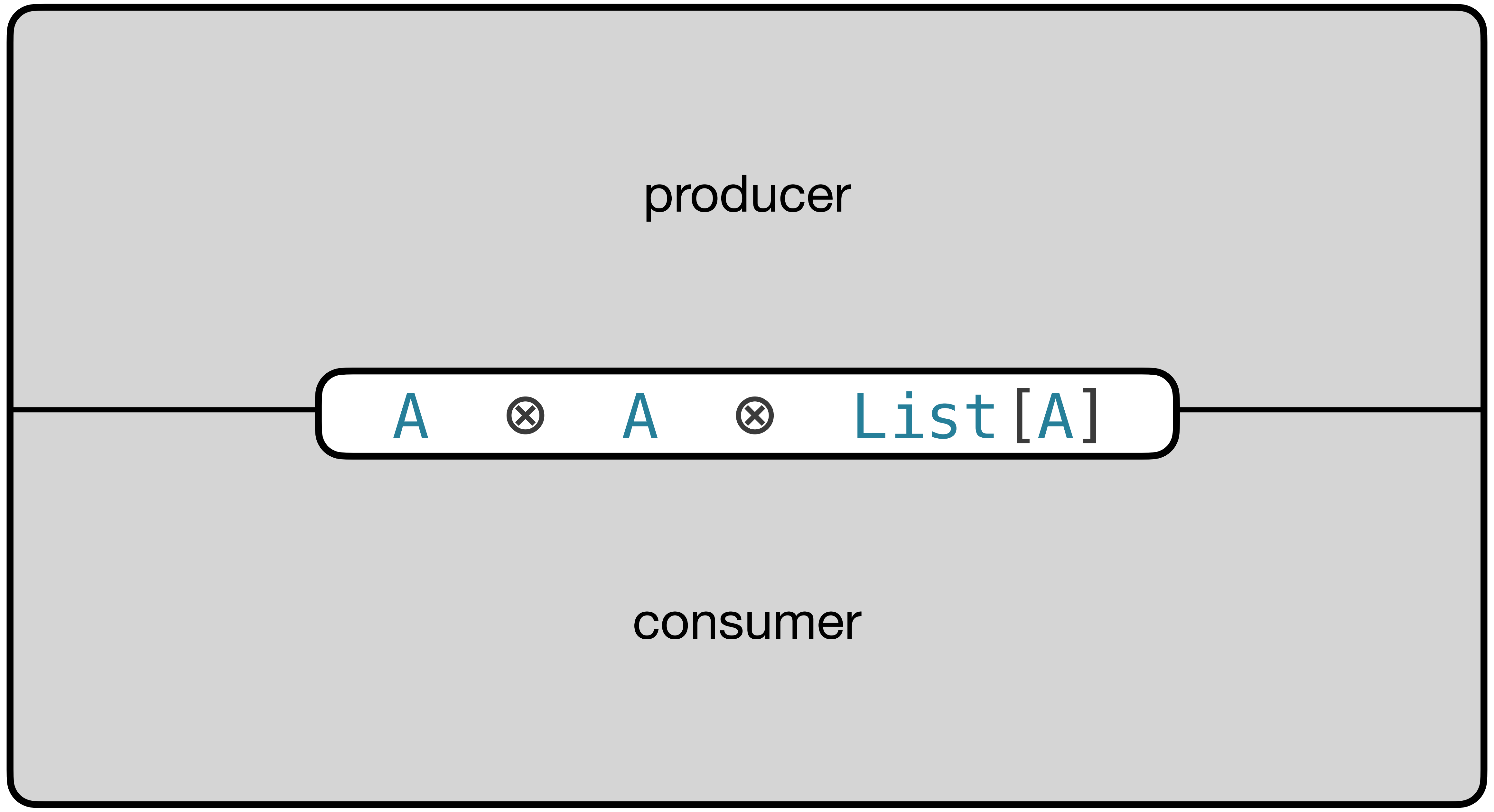
# List in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```



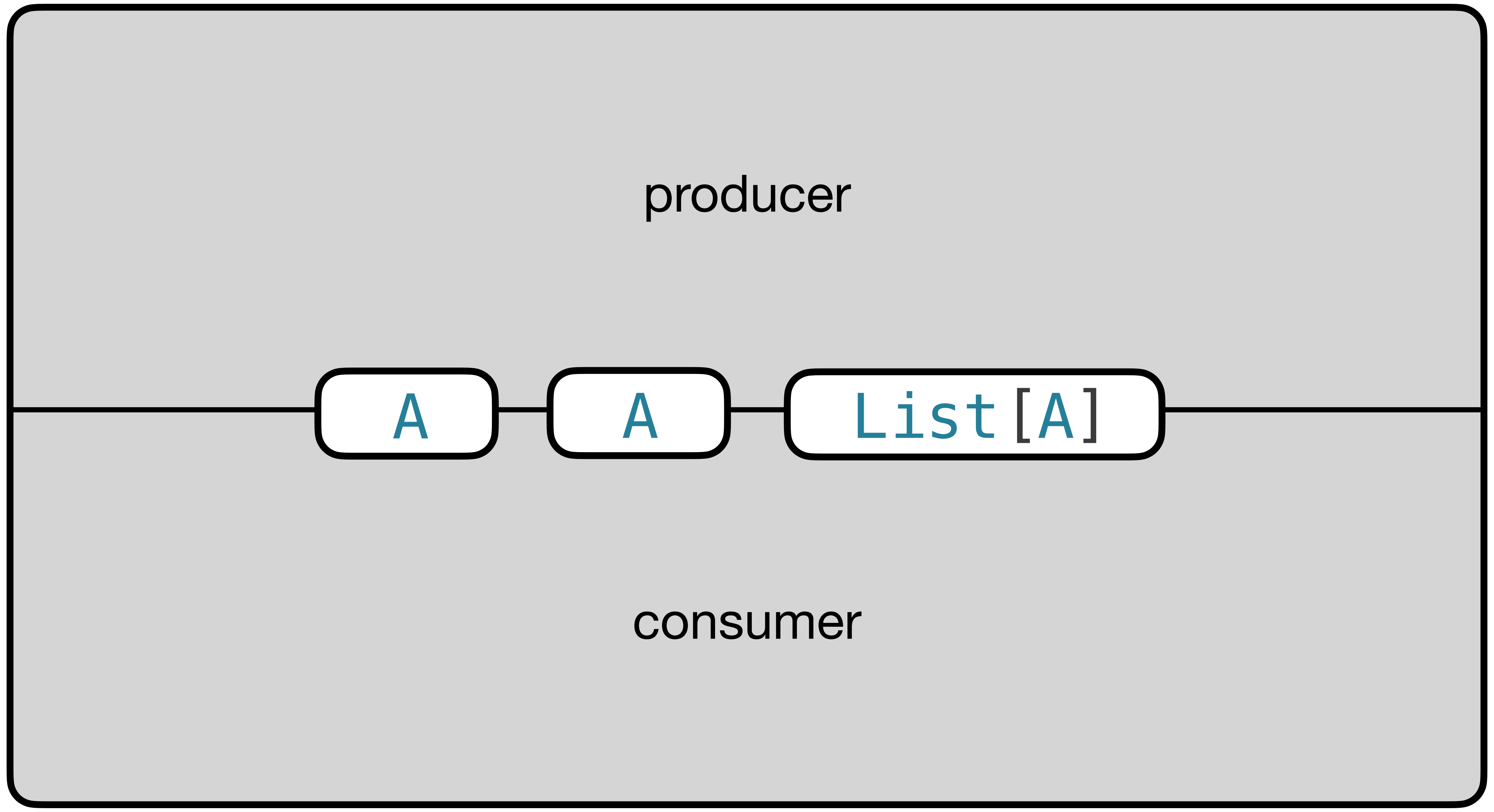
# List in Libretto

$$\texttt{List[A]} = \texttt{One} \oplus (\texttt{A} \otimes \texttt{List[A]})$$



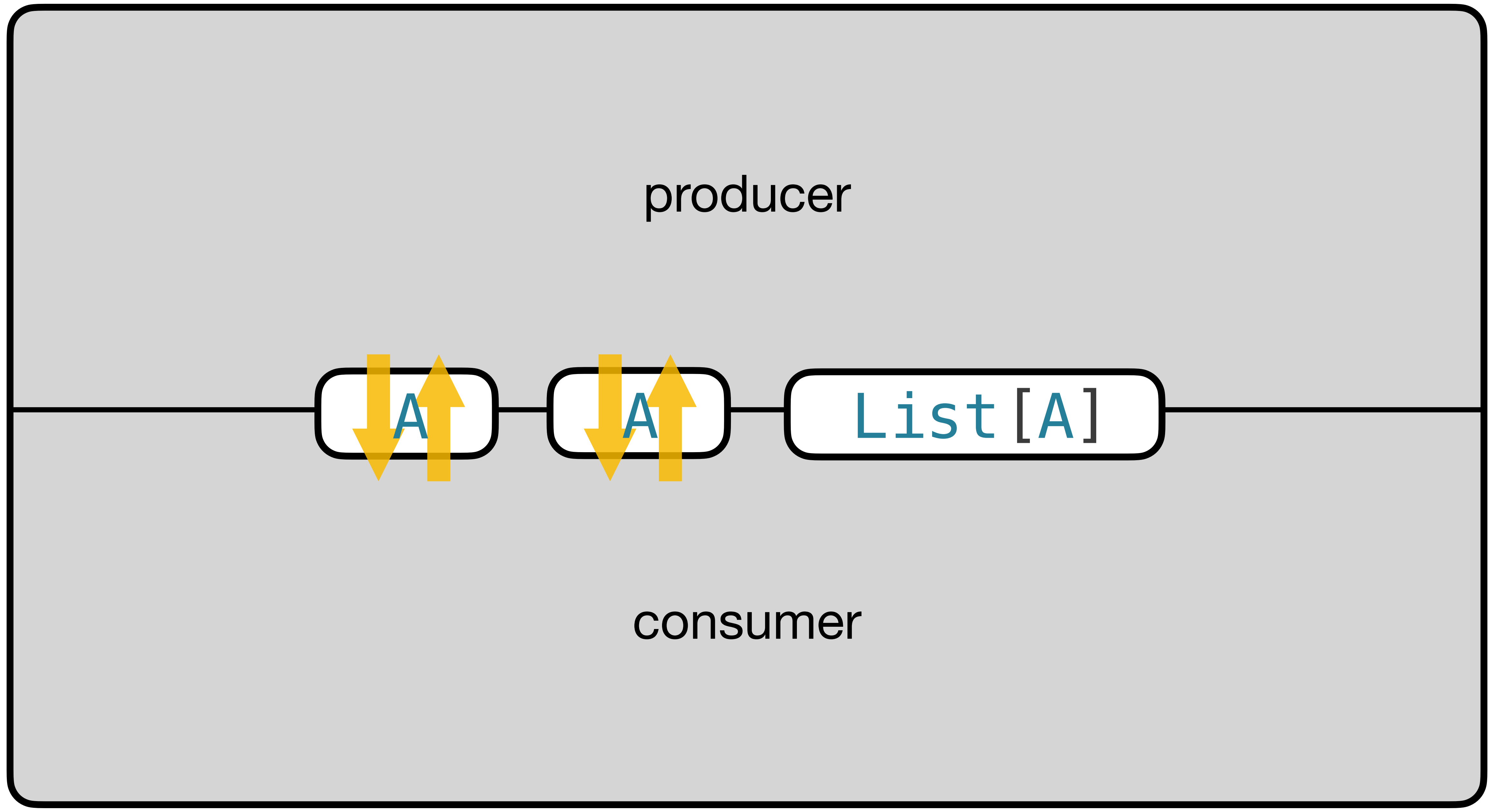
# List in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```



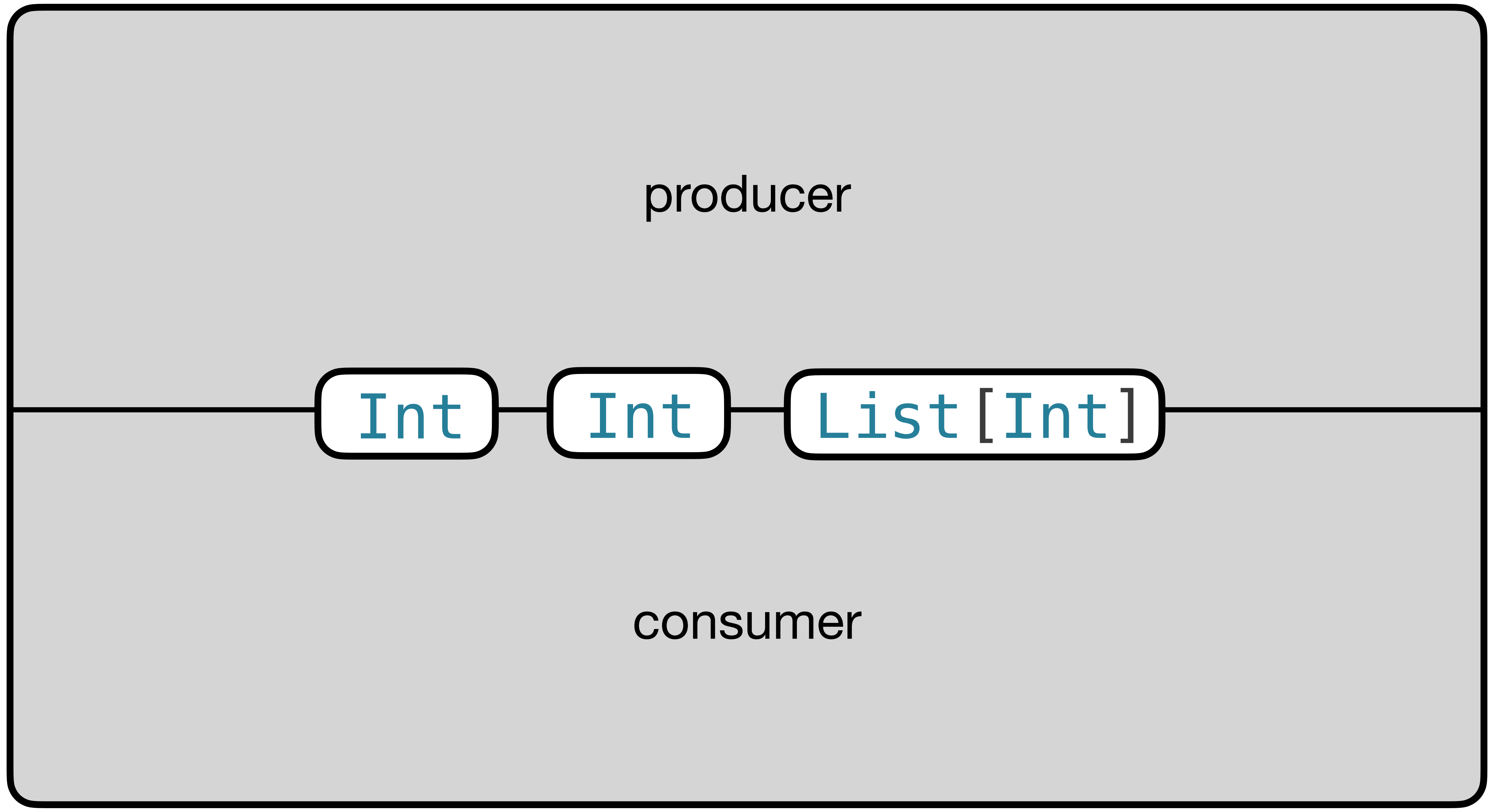
# List in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```



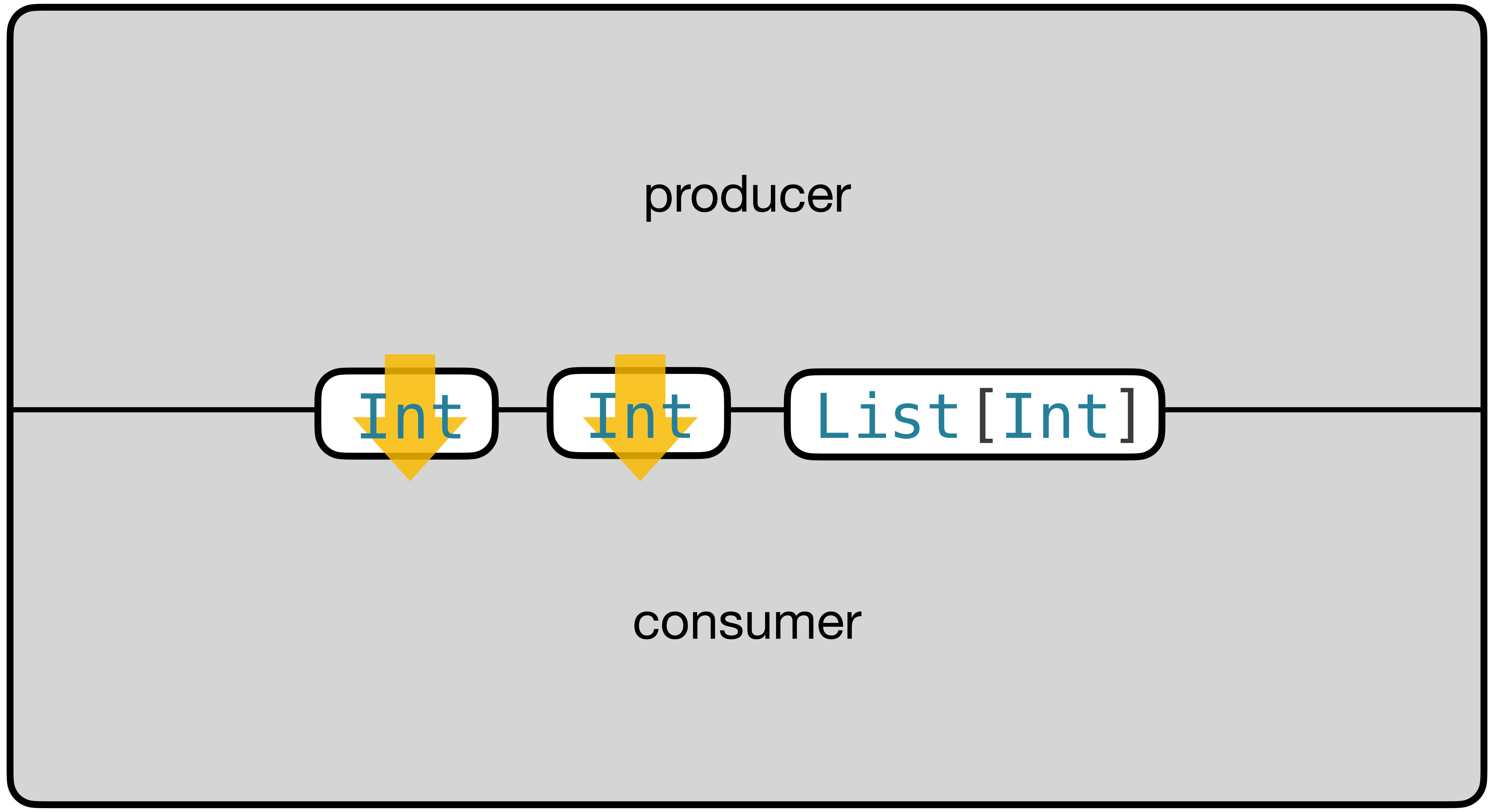
# List in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```



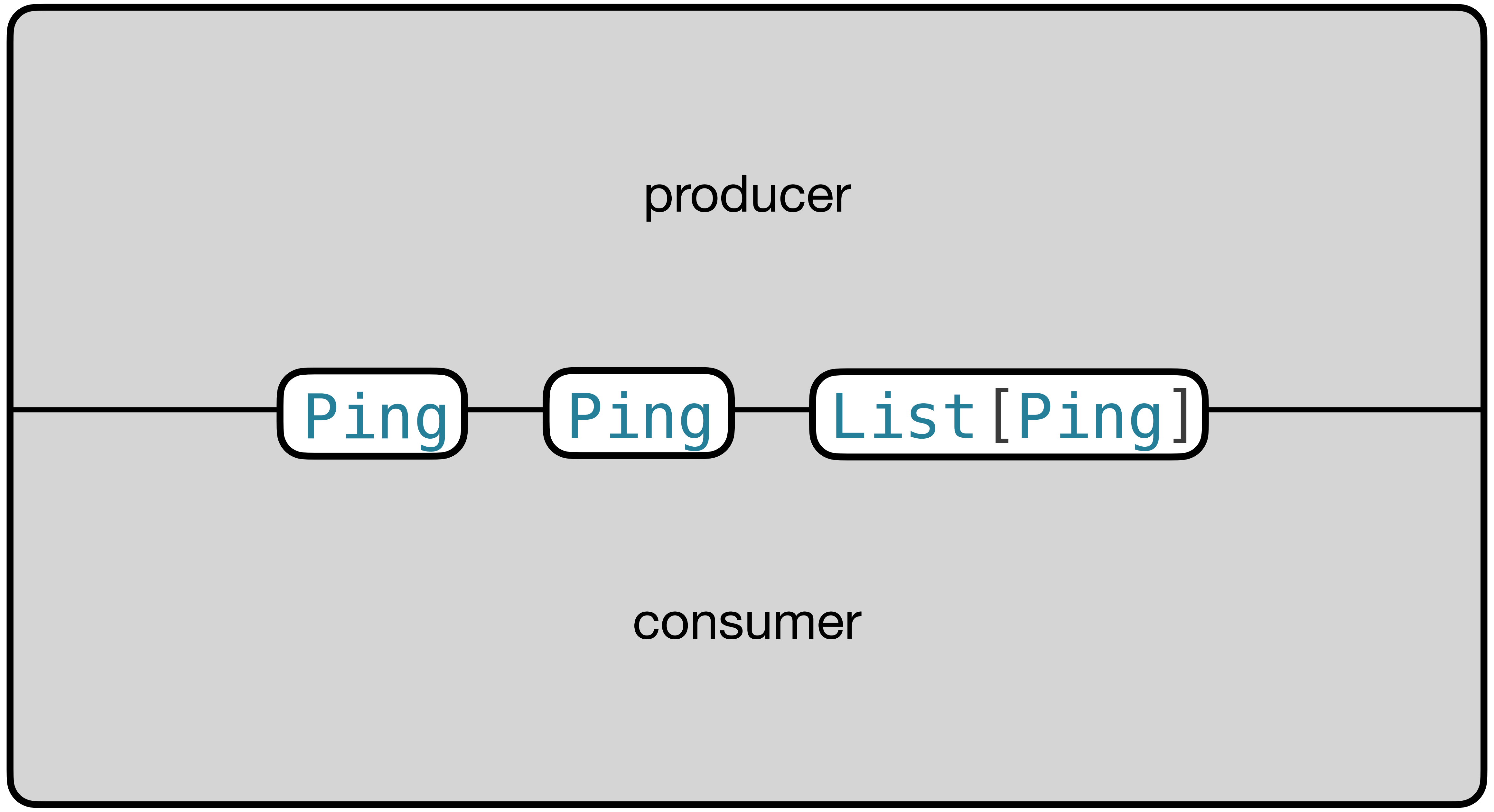
# List in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```



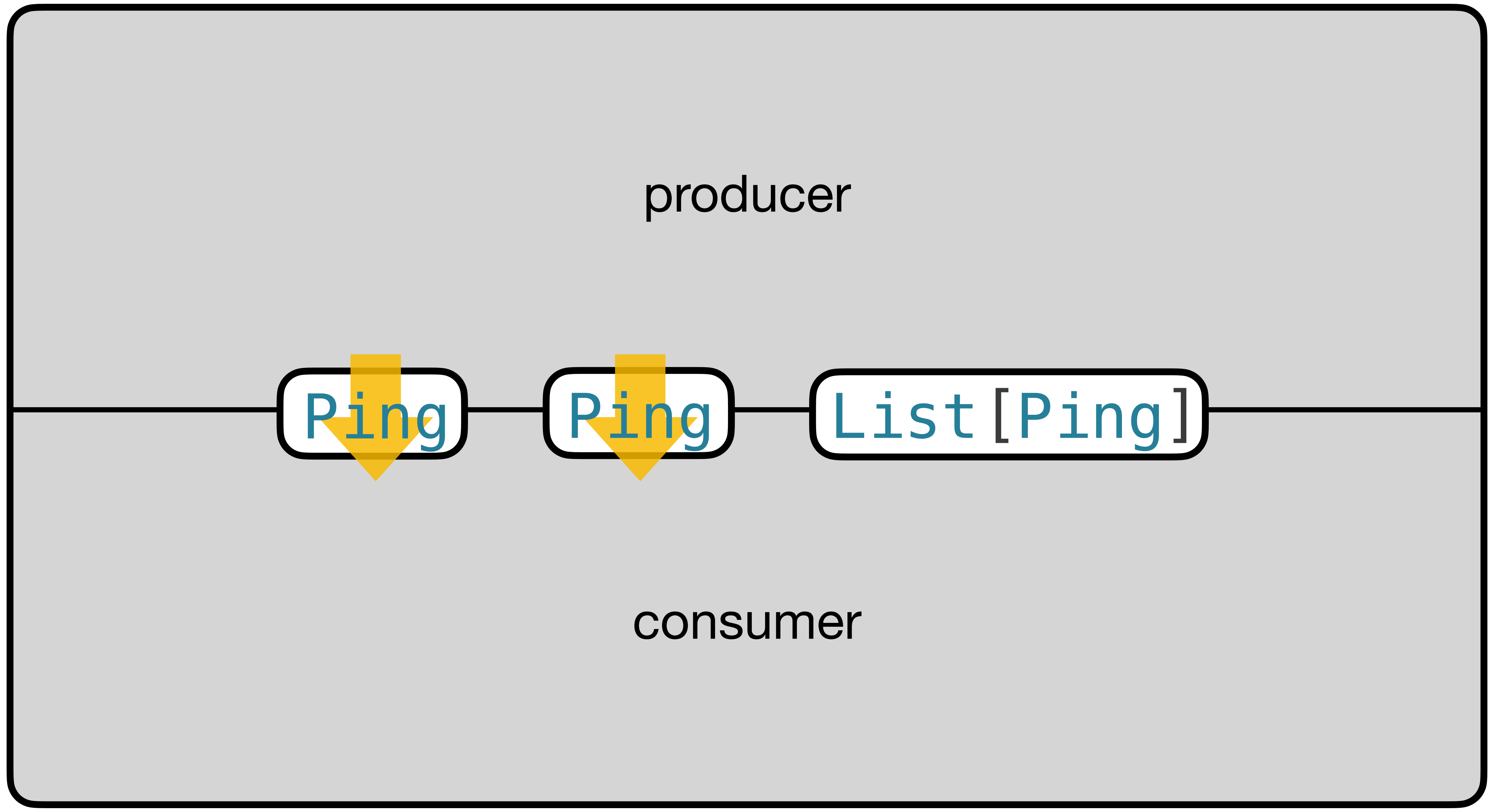
# List in Libretto

`List[A] = One ⊕ (A ⊗ List[A])`



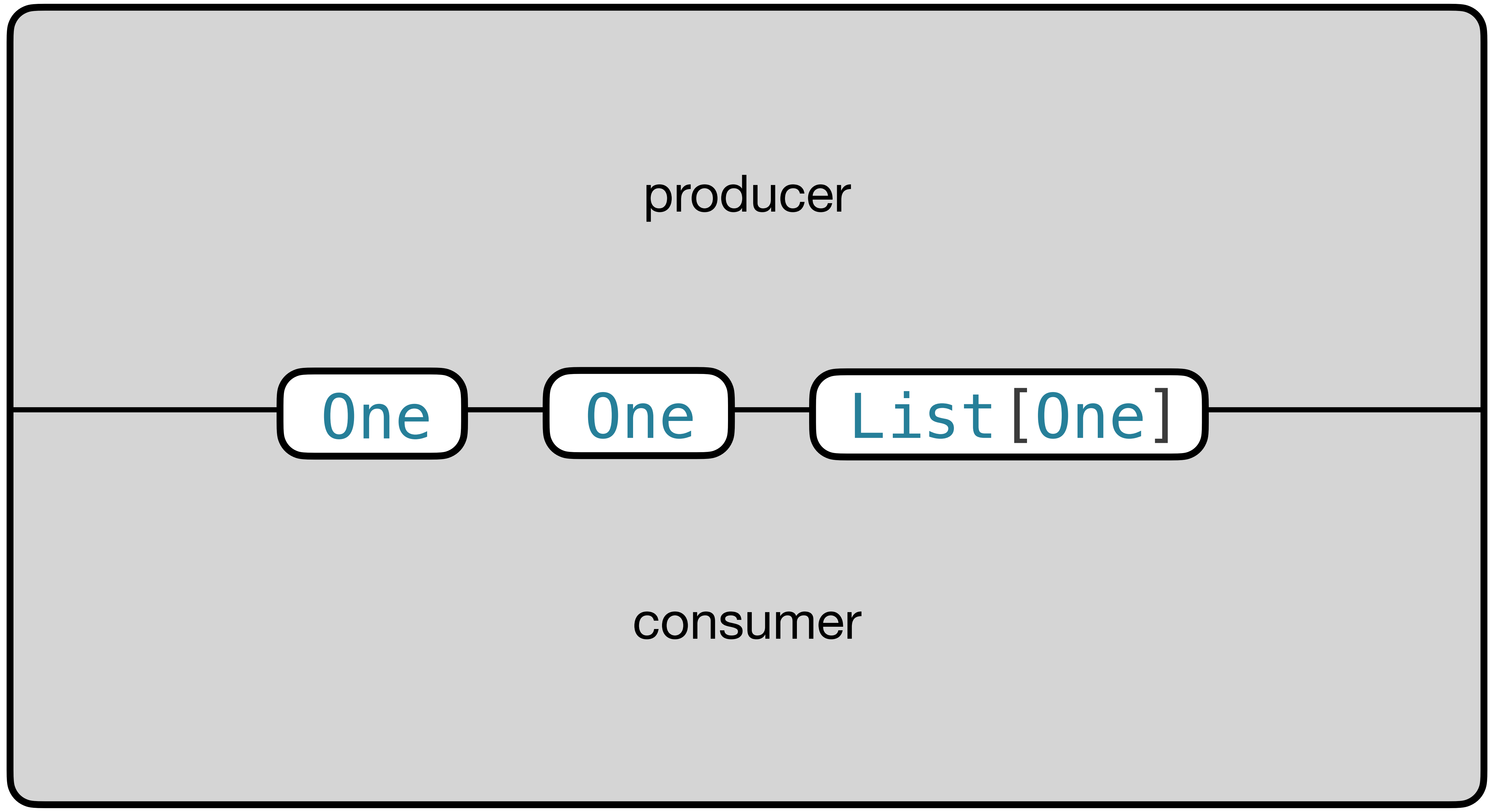
# List in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```



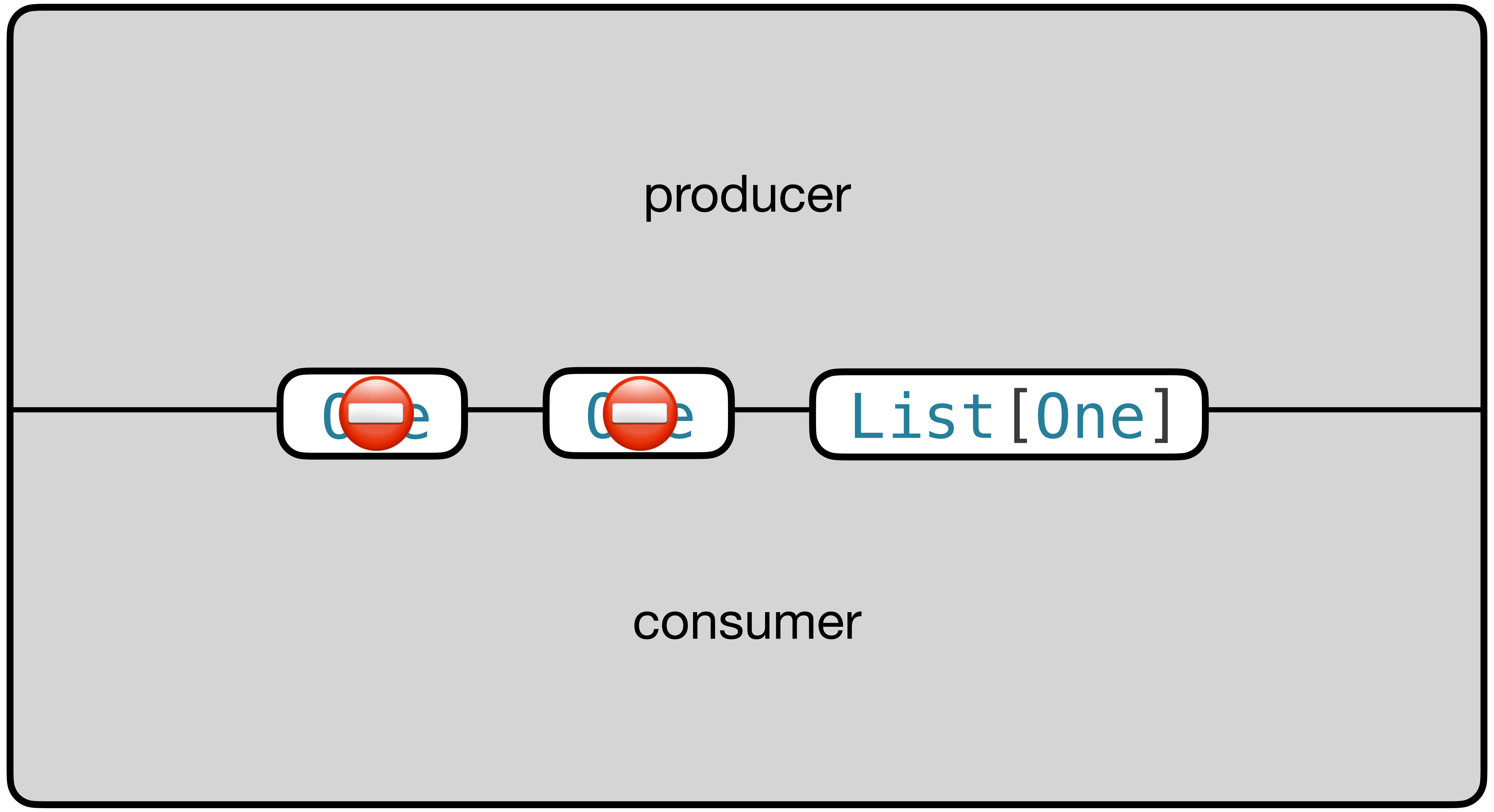
# List in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```



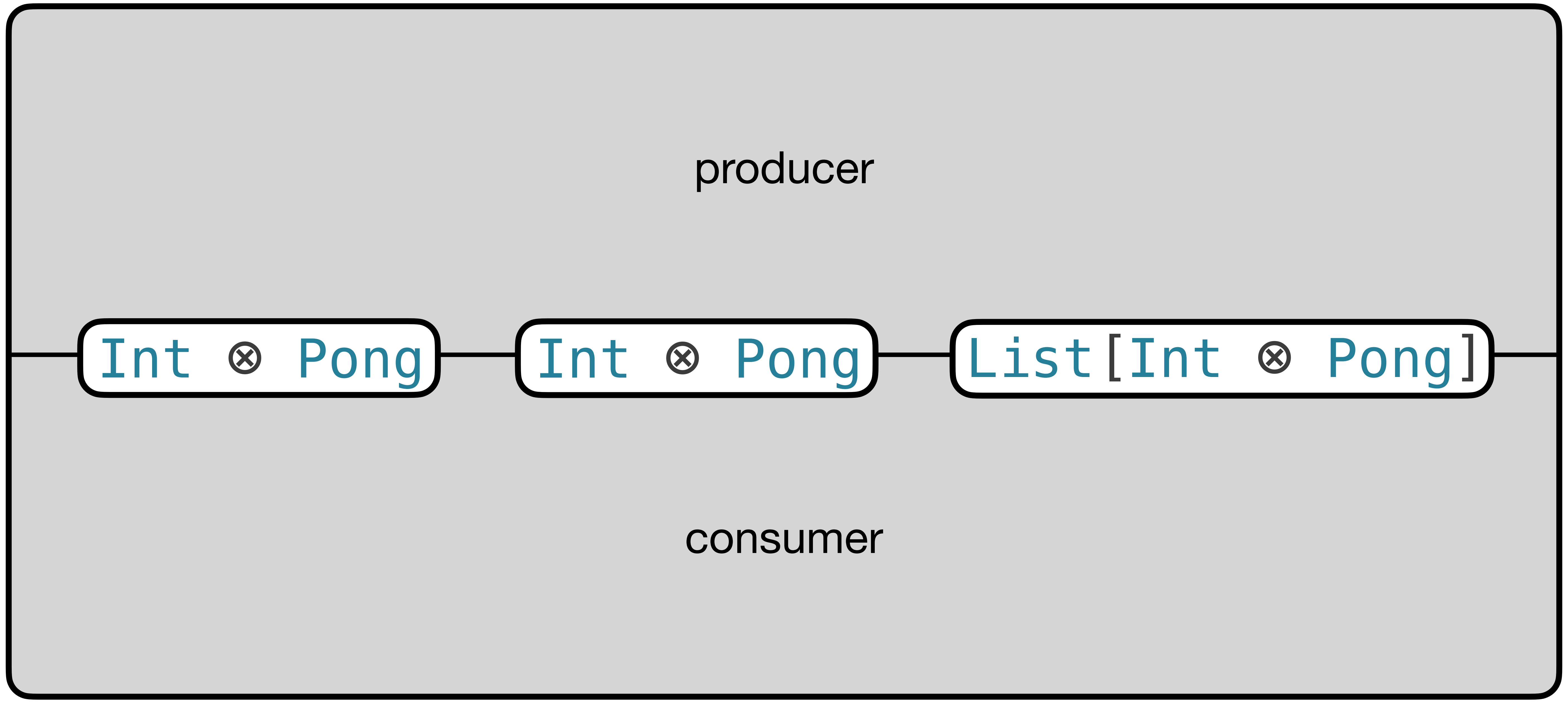
# List in Libretto

$$\texttt{List[A]} = \texttt{One} \oplus (\texttt{A} \otimes \texttt{List[A]})$$



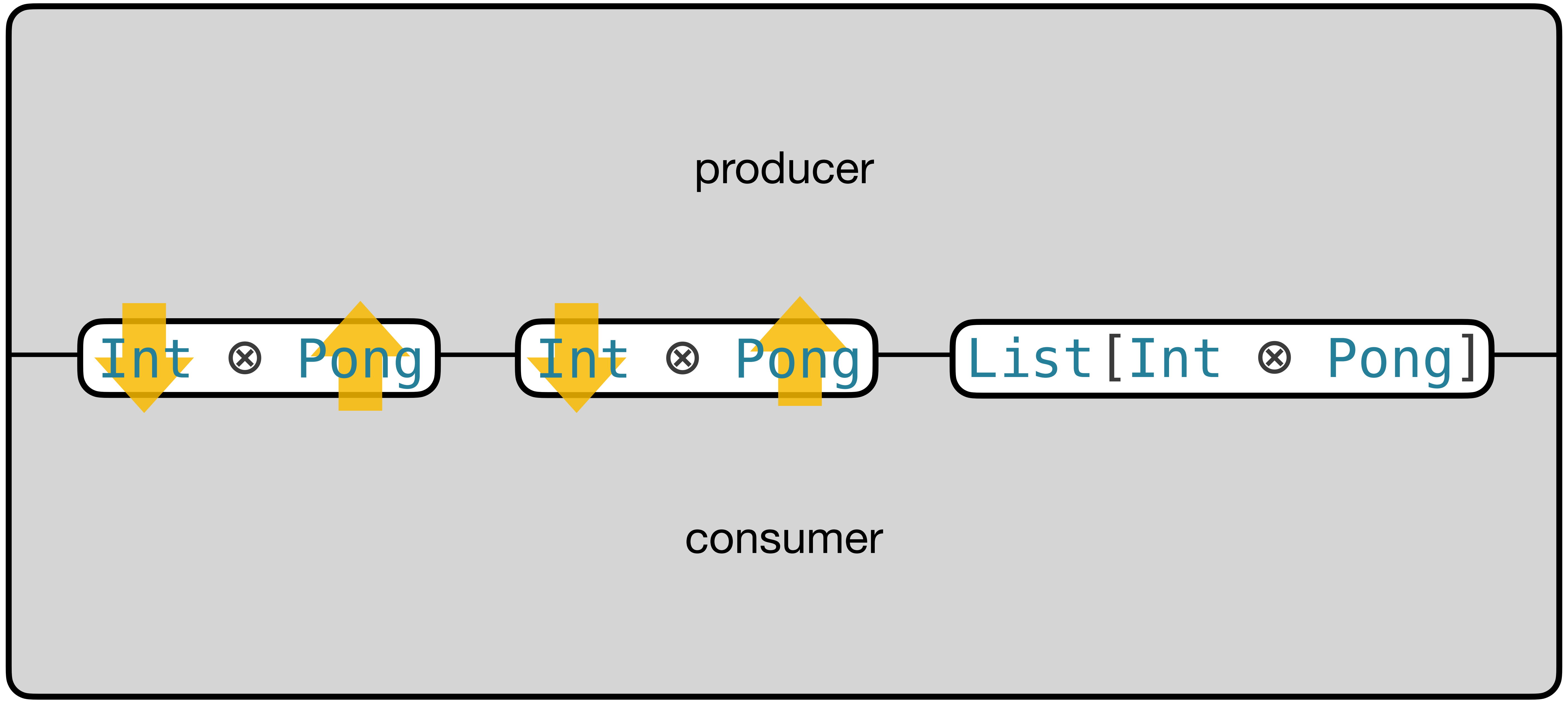
# List in Libretto

```
List[A] = One ⊕ (A ⊗ List[A])
```

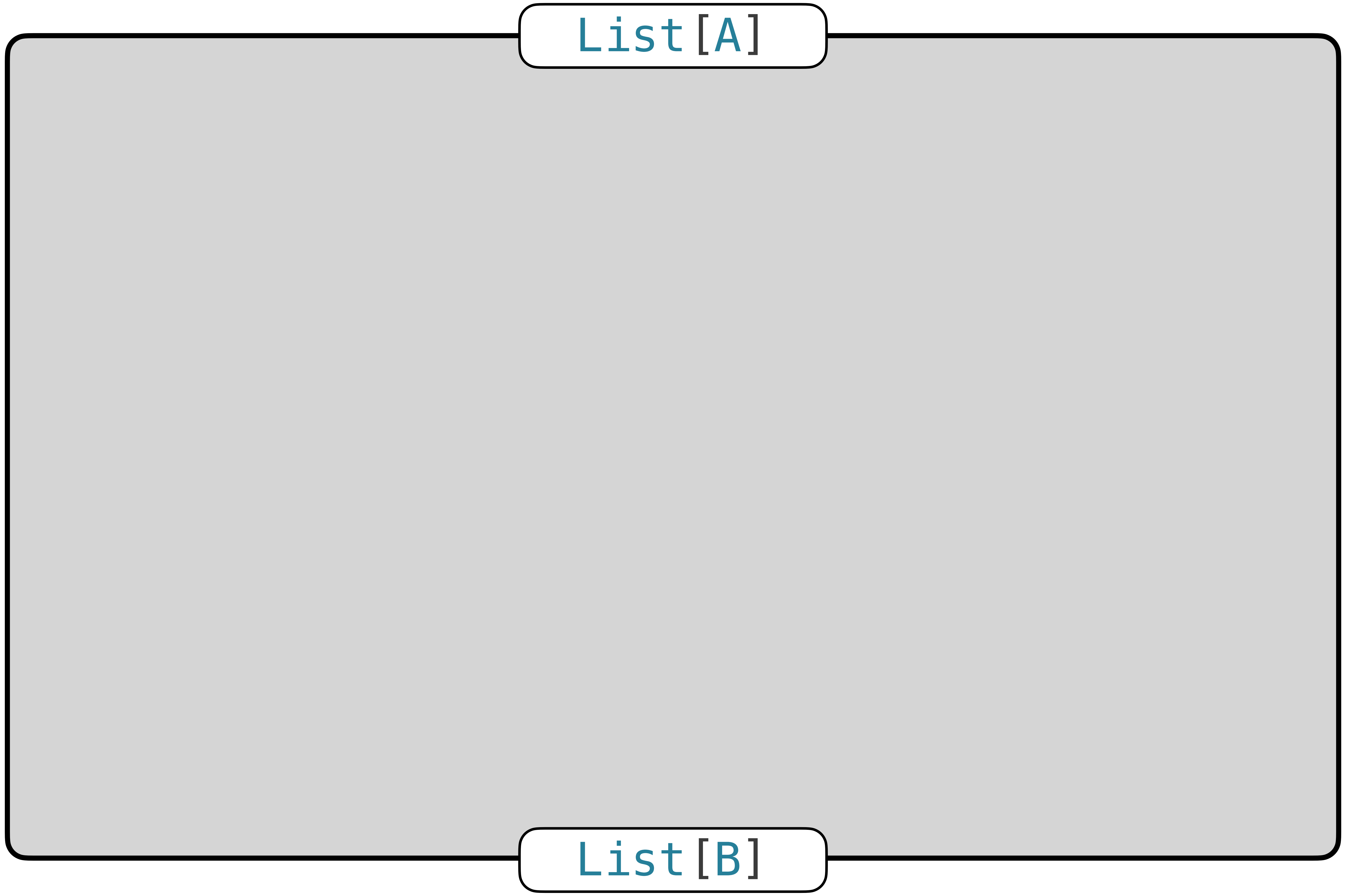


# List in Libretto

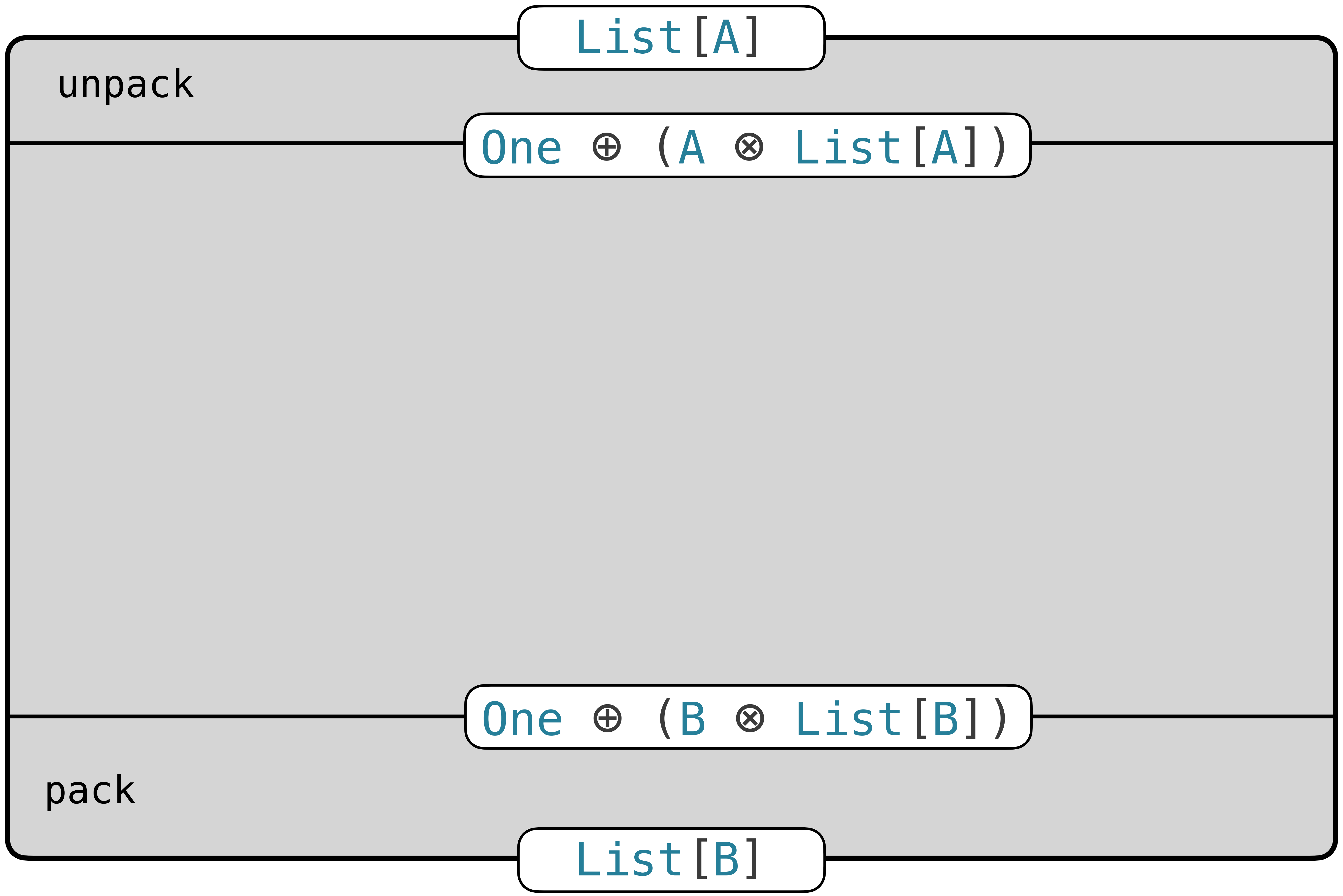
```
List[A] = One ⊕ (A ⊗ List[A])
```



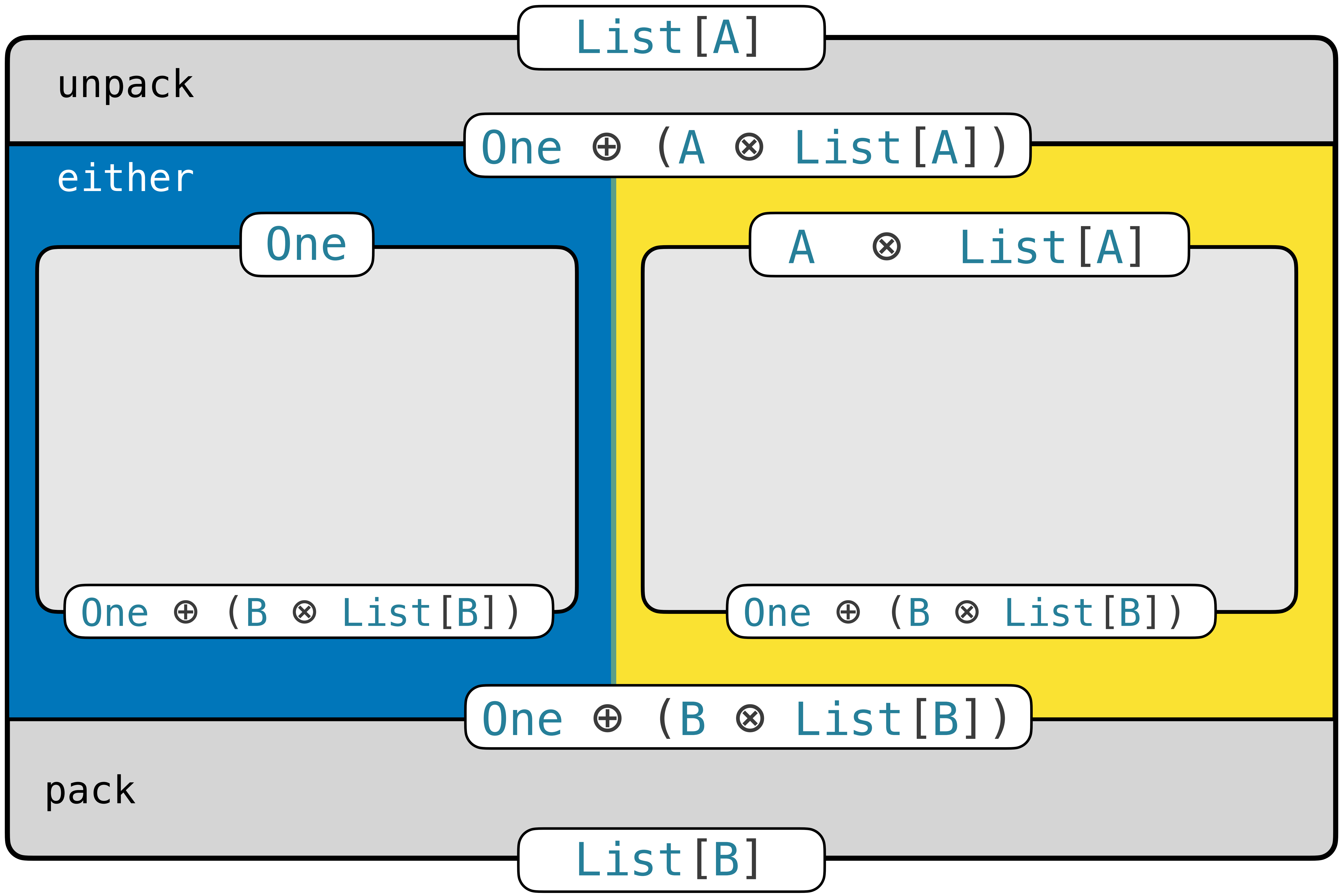
# List.map(f)



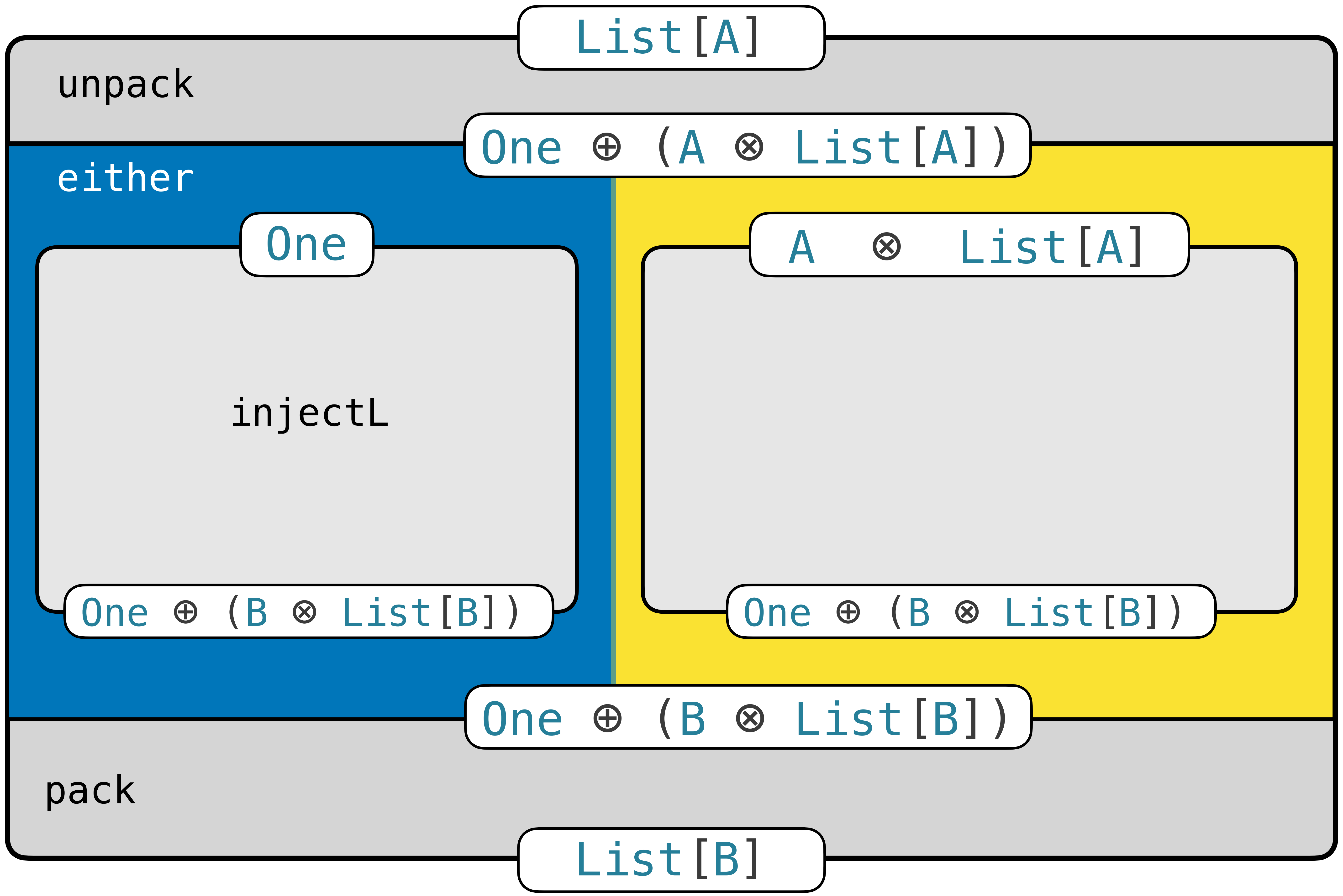
# List.map(f)



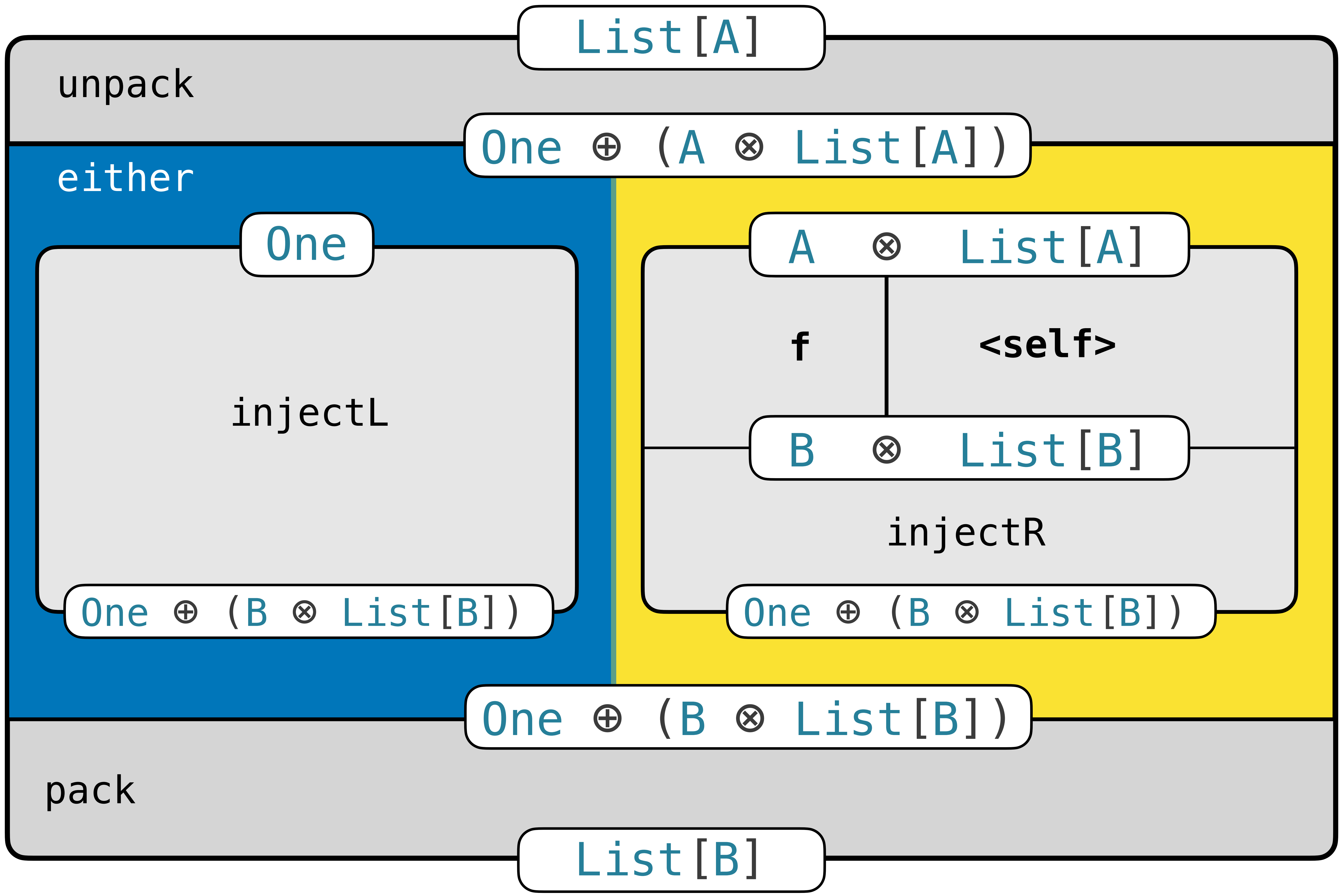
# List.map(f)


List[A]
unpack
One ⊕ (A ⊗ List[A])
either
One
A ⊗ List[A]
One ⊕ (B ⊗ List[B])
One ⊕ (B ⊗ List[B])
One ⊕ (B ⊗ List[B])
pack
List[B]

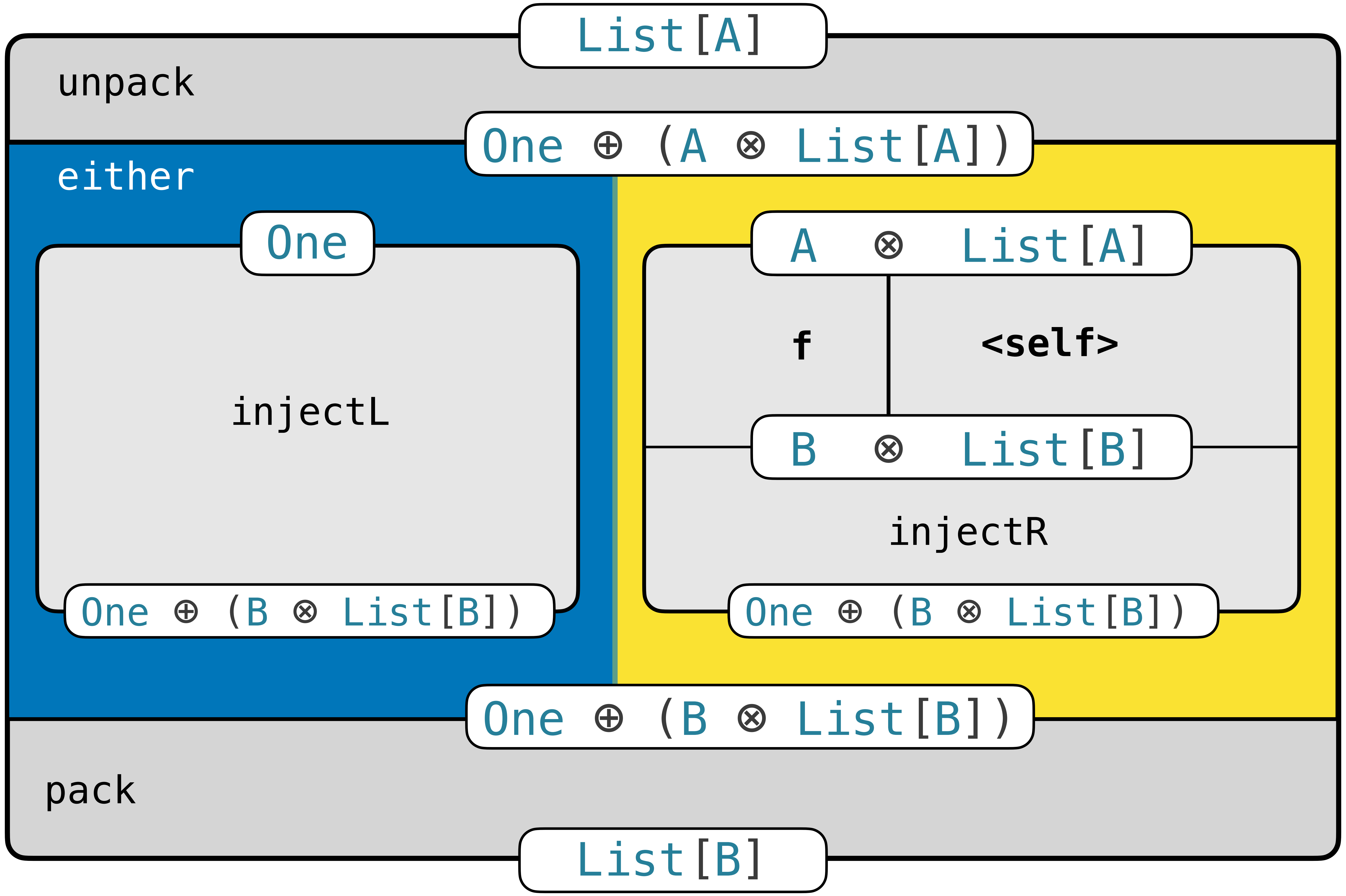
# List.map(f)


List[A]
unpack
One ⊕ (A ⊗ List[A])
either
One
injectL
One ⊕ (B ⊗ List[B])
A ⊗ List[A]
One ⊕ (B ⊗ List[B])
One ⊕ (B ⊗ List[B])
pack
List[B]

# List.map(f)

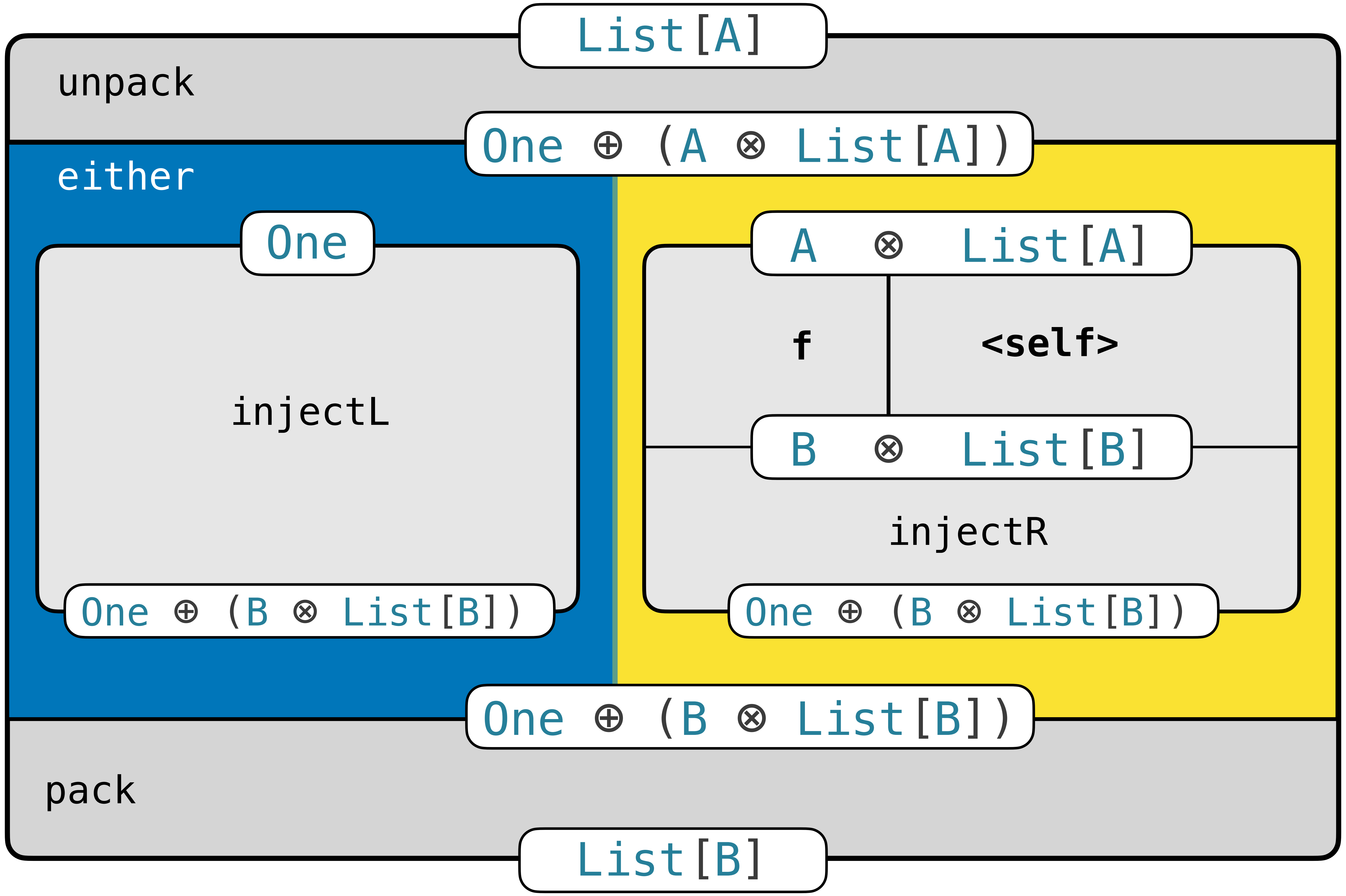

List[A]
unpack
One ⊕ (A ⊗ List[A])
either
One
injectL
One ⊕ (B ⊗ List[B])
A ⊗ List[A]
f
<self>
B ⊗ List[B]
injectR
One ⊕ (B ⊗ List[B])
One ⊕ (B ⊗ List[B])
pack
List[B]

# List.map(f)



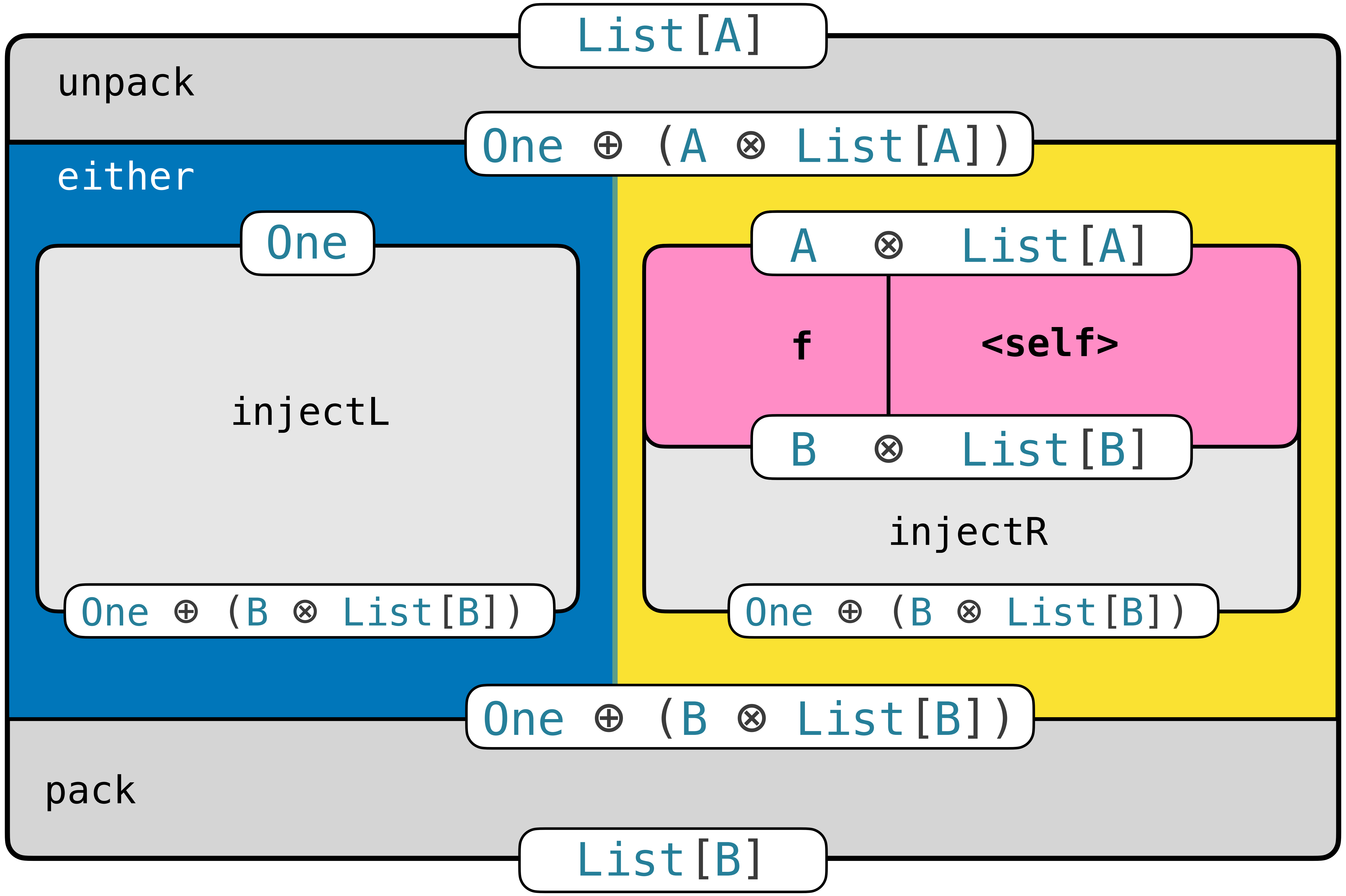

```
def map[A, B](
  f: A -∘ B
): List[A] -∘ List[B] =

// point-free

rec { self =>
  unpack >
  either(
    injectL,
    par(f, self) > injectR
  ) >
  pack
}
```

# List.map(f)



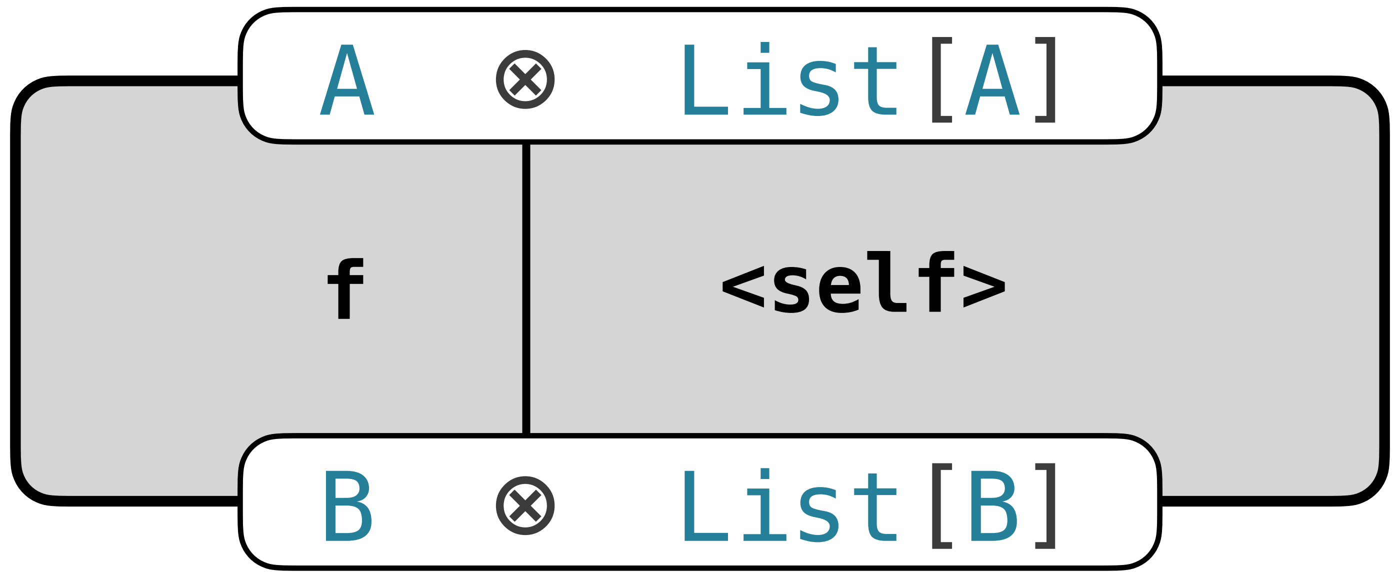

```
def map[A, B](
  f: A -∘ B
): List[A] -∘ List[B] =

// point-full

rec { self =>
 λ { as =>
  pack(
   unpack(as) switch {
    case Left(one) =>
     injectL(one)
    case Right(h ⊗ t) =>
     injectR(f(h) ⊗ self(t))
})}}
```

# List.map(f)

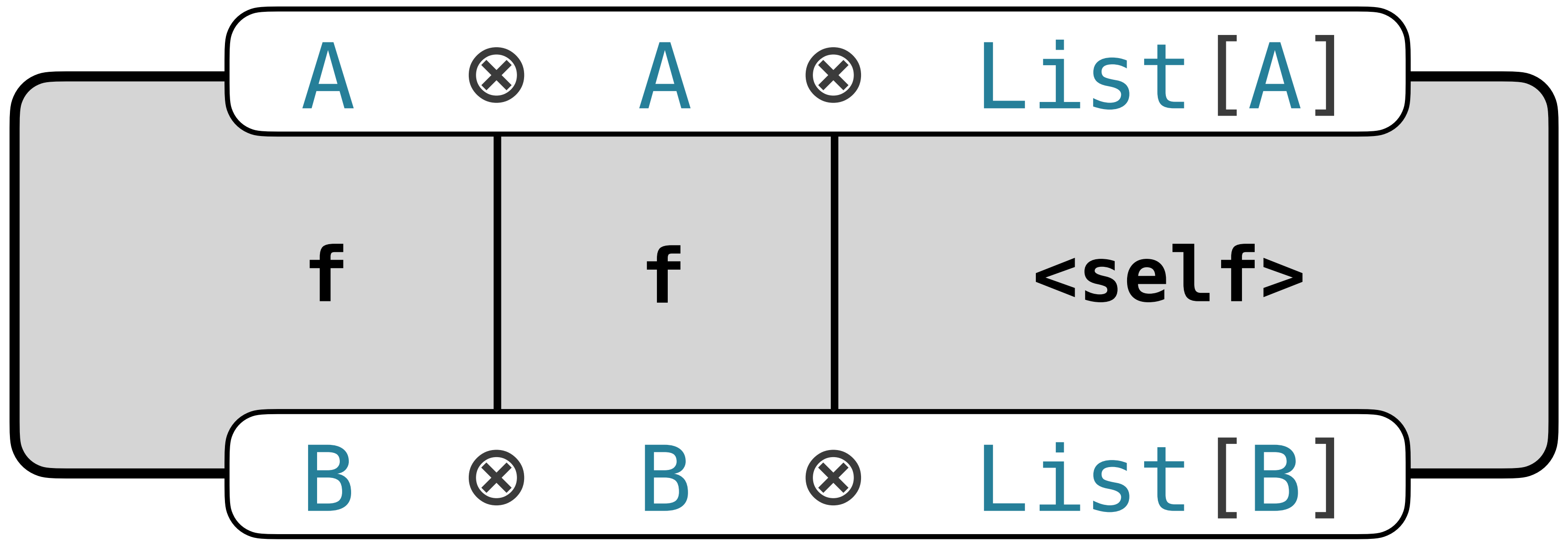



```
def map[A, B](
  f: A -∘ B
): List[A] -∘ List[B] =

// point-full

rec { self =>
 λ { as =>
  pack(
   unpack(as) switch {
    case Left(one) =>
     injectL(one)

    case Right(h ⊗ t) =>

     injectR(f(h) ⊗ self(t))
})}}
```

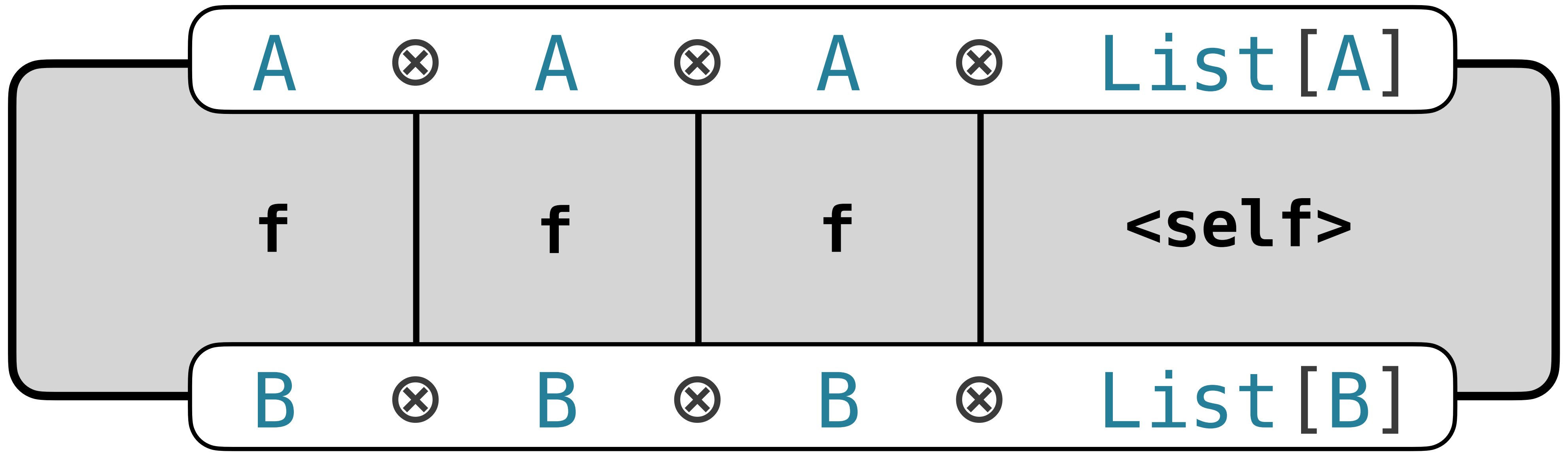
# List.map(f)



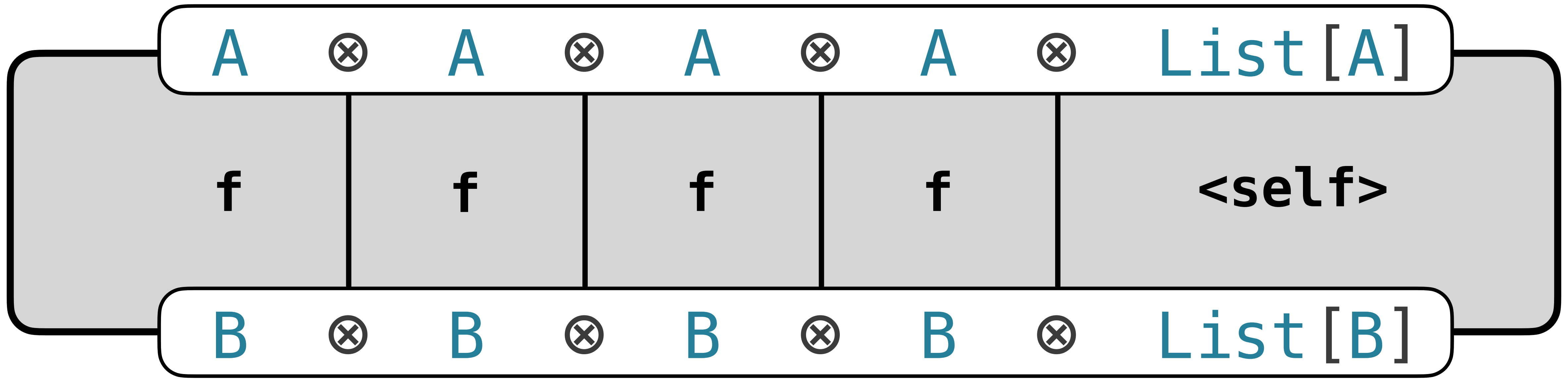
# List.map(f)



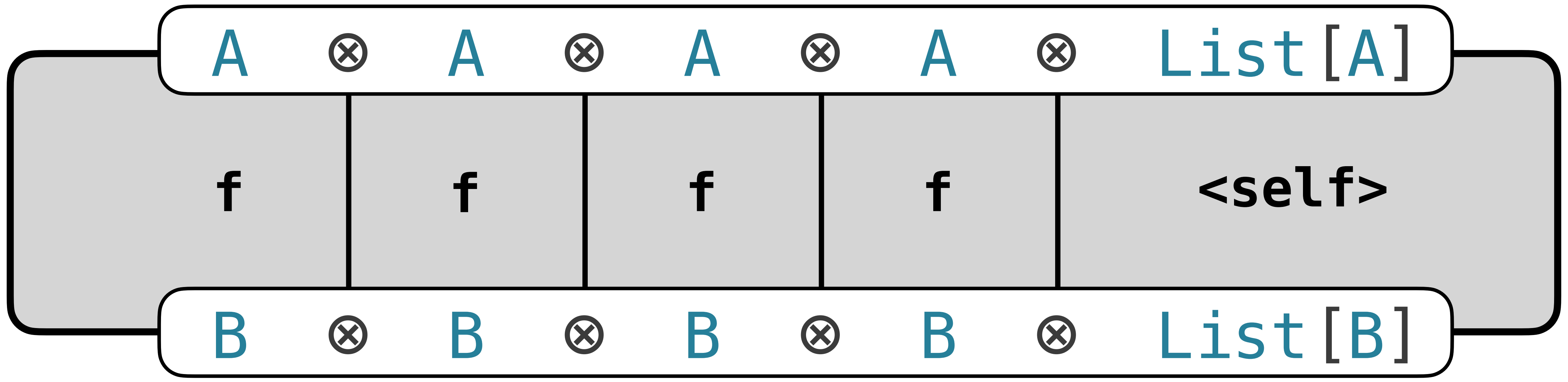
# List.map(f)



# List.map(f)



# List.map(f)




**Implicitly concurrent**

# Endless

```
Endless[A] = One & (A ⊗ Endless[A])
```

# Endless

consumer choice

```
Endless[A] = One & (A ⊗ Endless[A])
```

# Endless

consumer choice

```
Endless[A] = One & (A ⊗ Endless[A])
```

- consumer may
  - close
  - ask for next element
- producer has to oblige
- co-`List`

# Endless

consumer choice

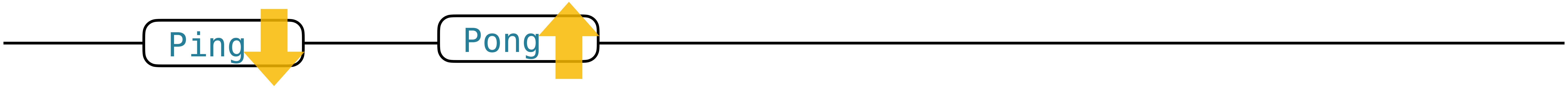
```
Endless[A] = One & (A ⊗ Endless[A])
```

- consumer may
  - close
  - ask for next element
- producer has to oblige
- co-`List`

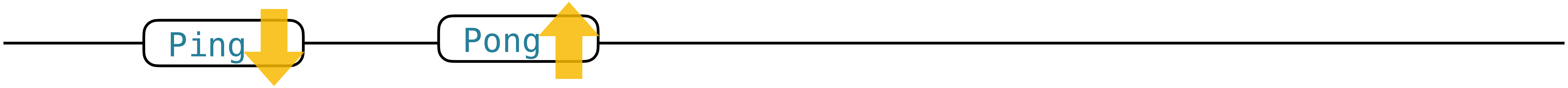
```
List[A] = One ⊕ (A ⊗ List[A])
```

producer choice

# Signals

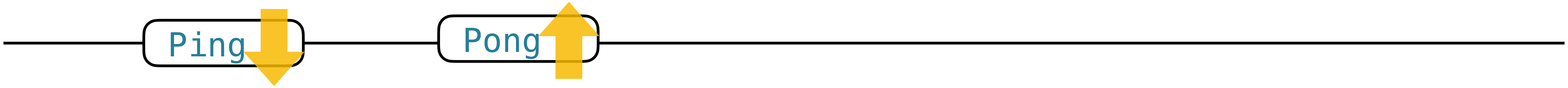


# Signals

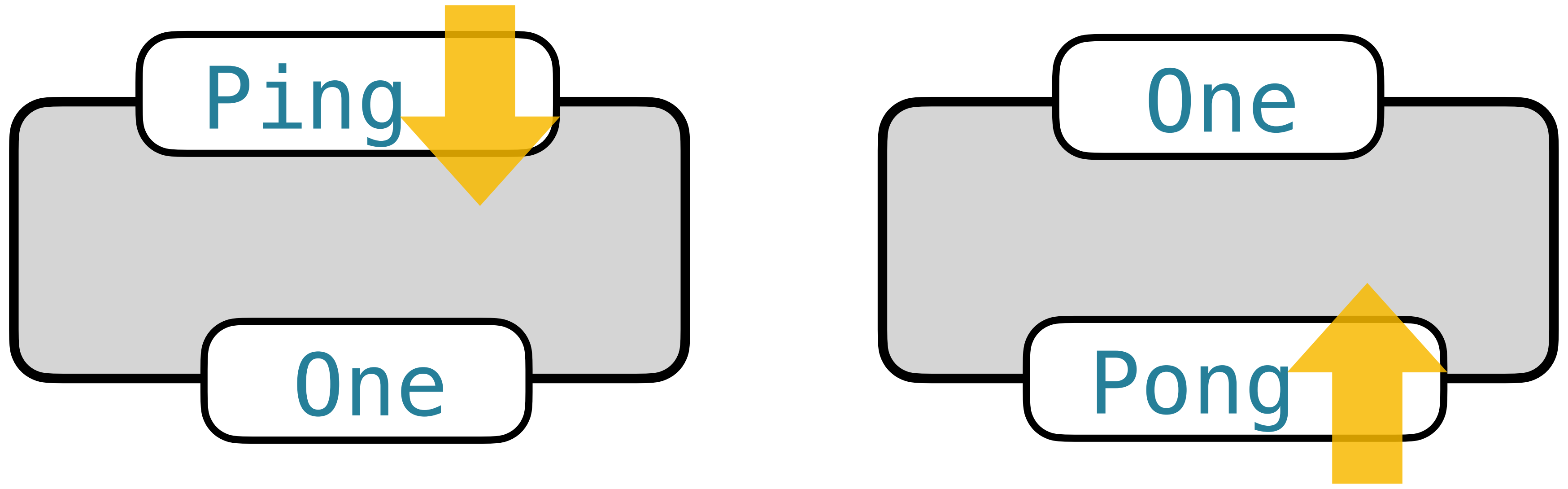



**dismissible**

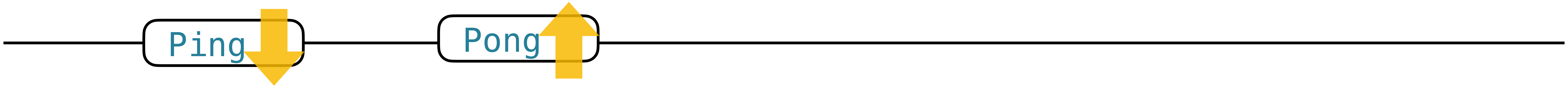
# Signals

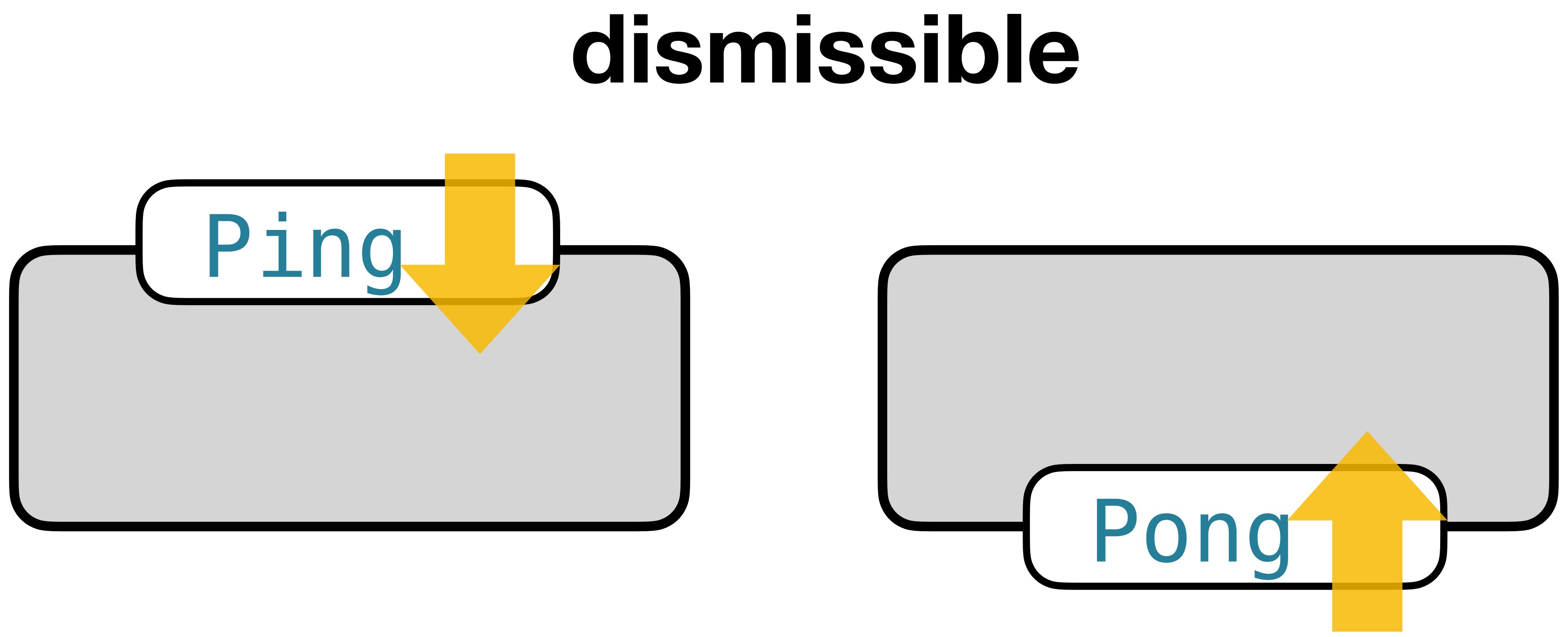



**dismissible**

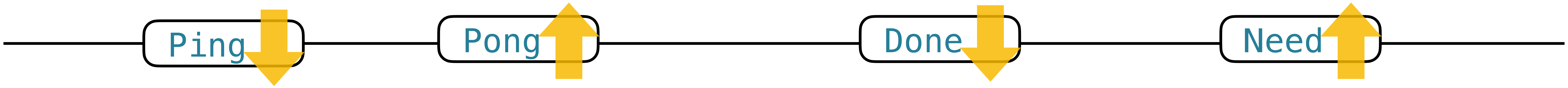


# Signals

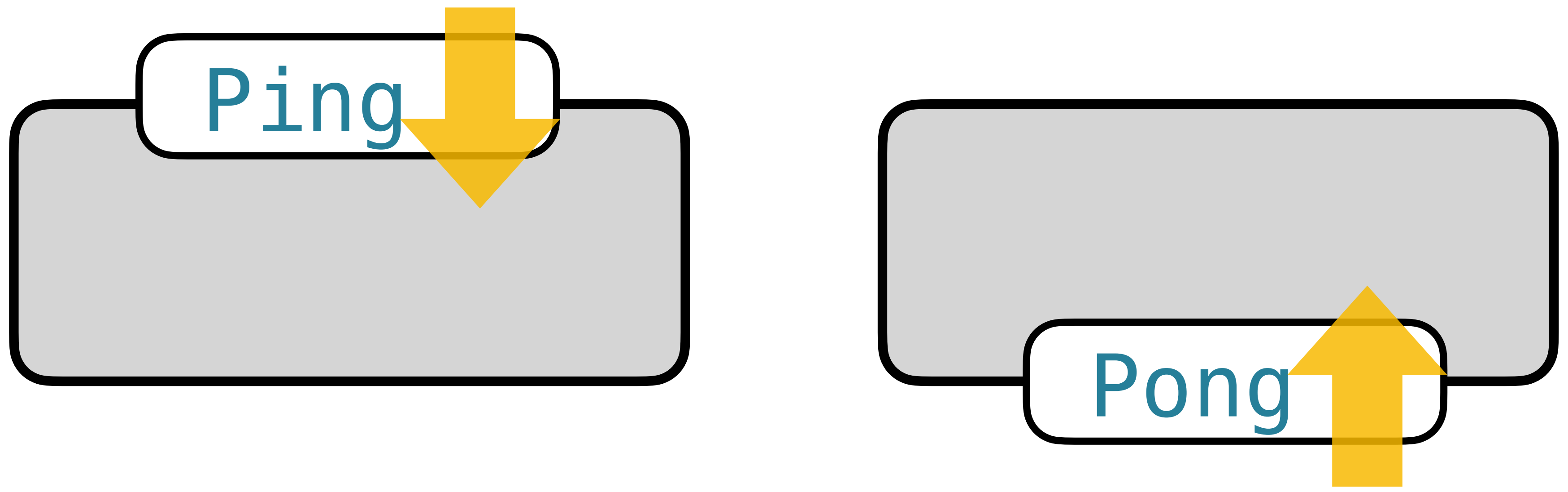


# Signals




**dismissible**

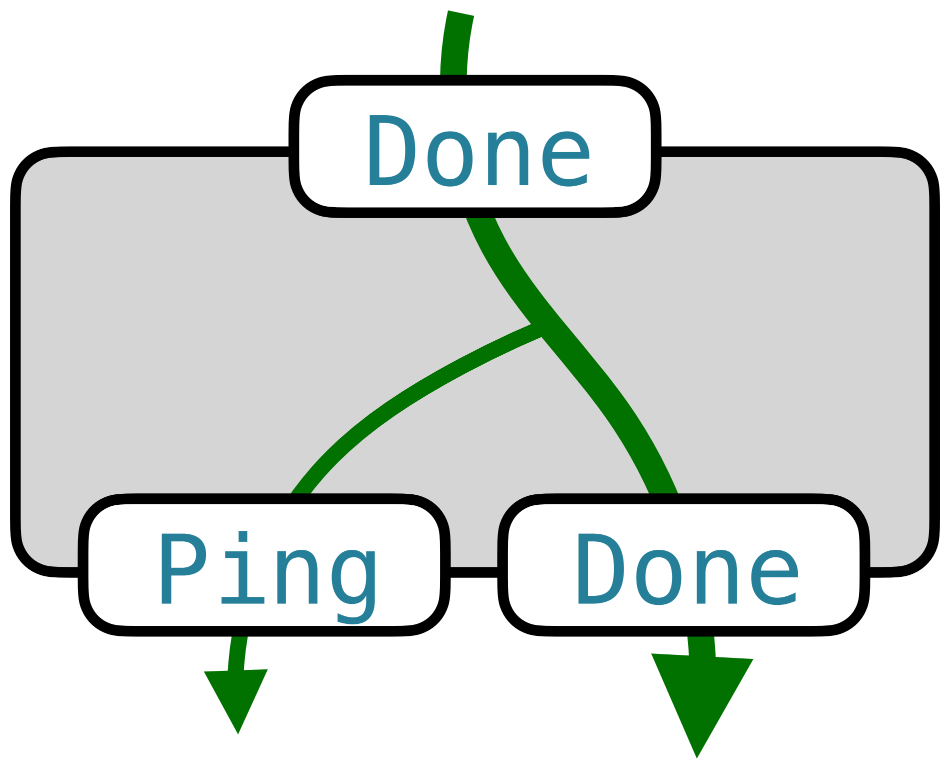


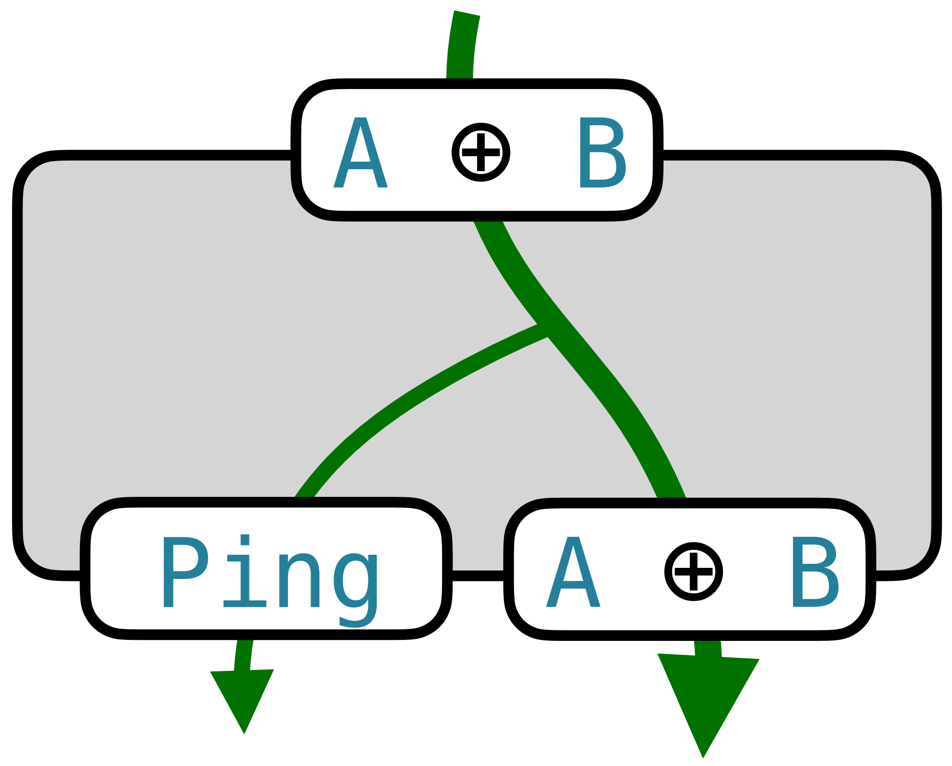

**non-dismissible**

- must be awaited
- signal completion of something expensive

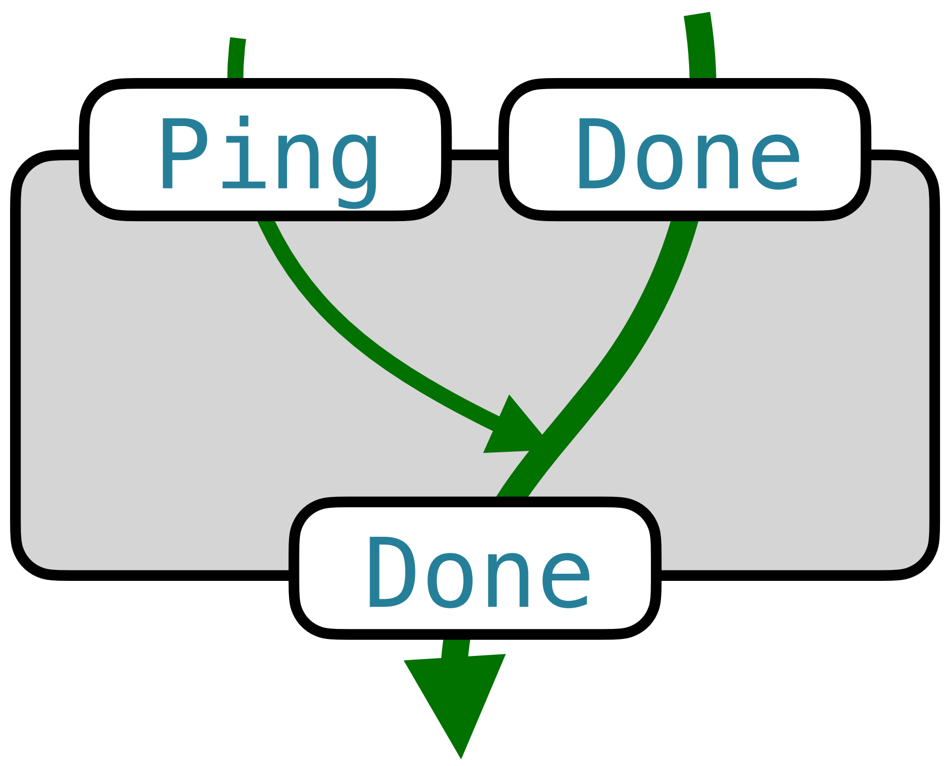
# Signals

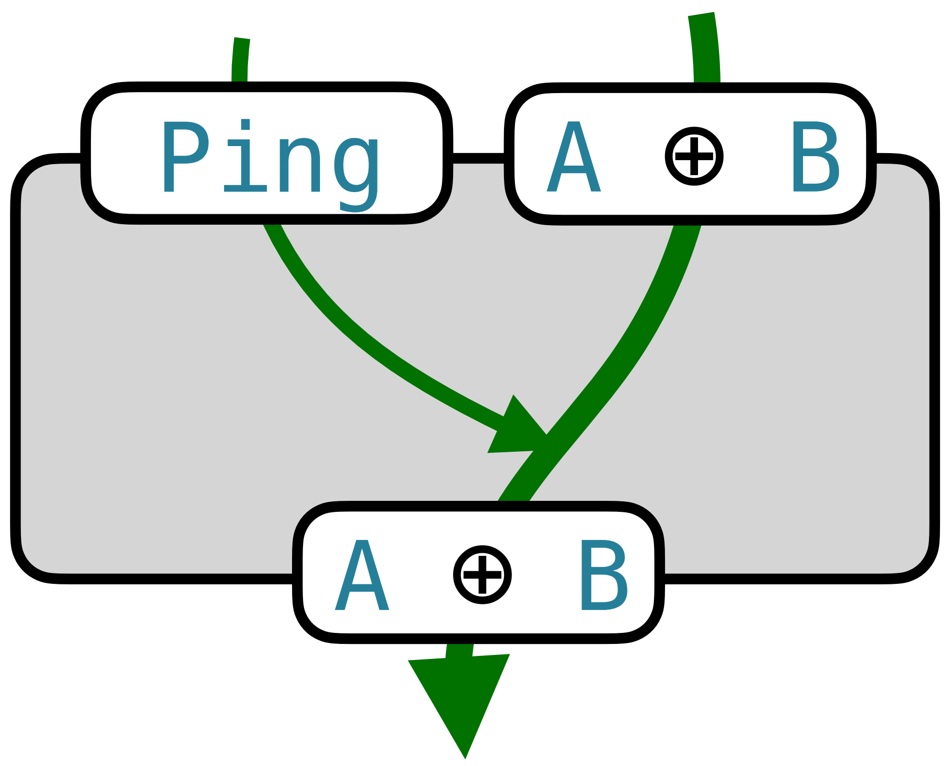
**Ping introduction** (e.g.)



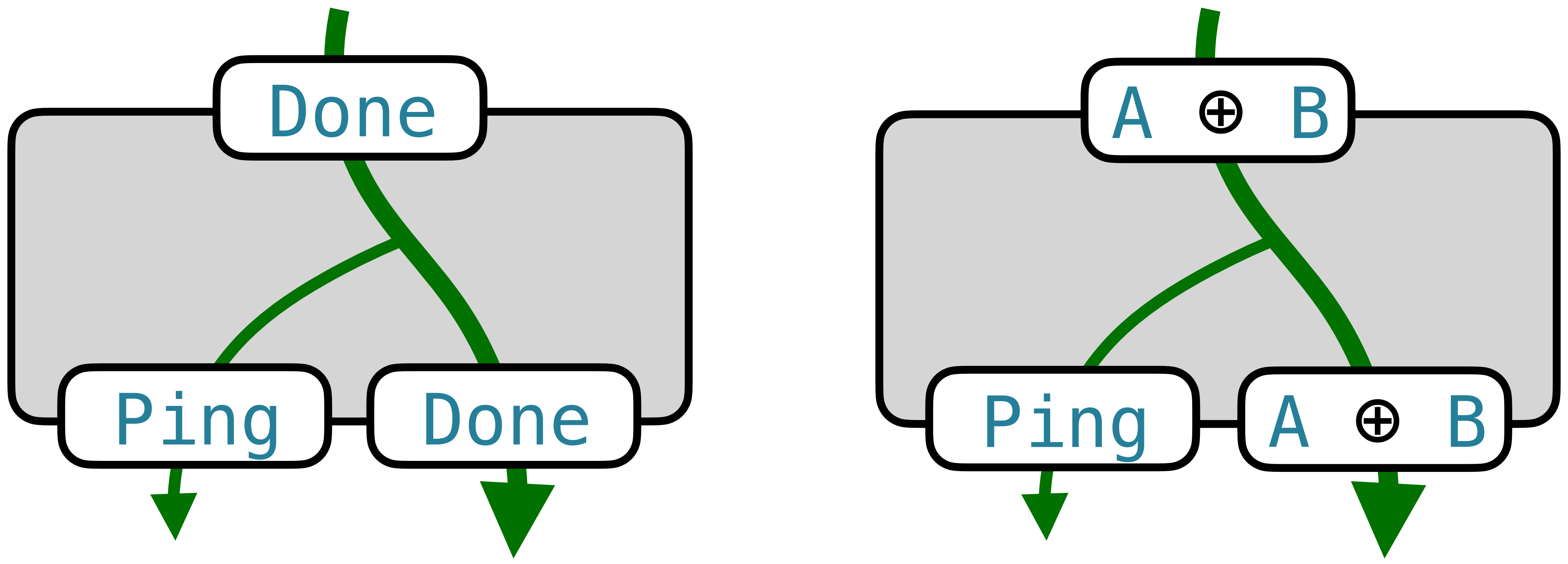

**Ping elimination** (e.g.)



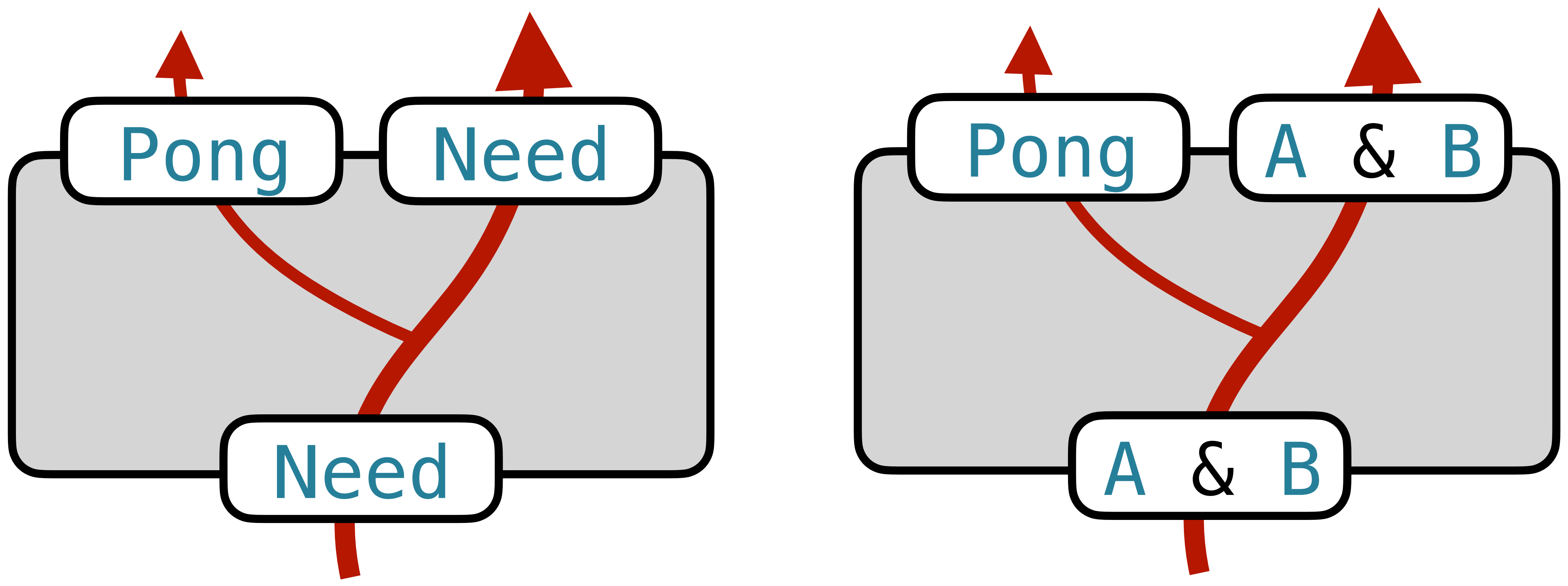



# Signals

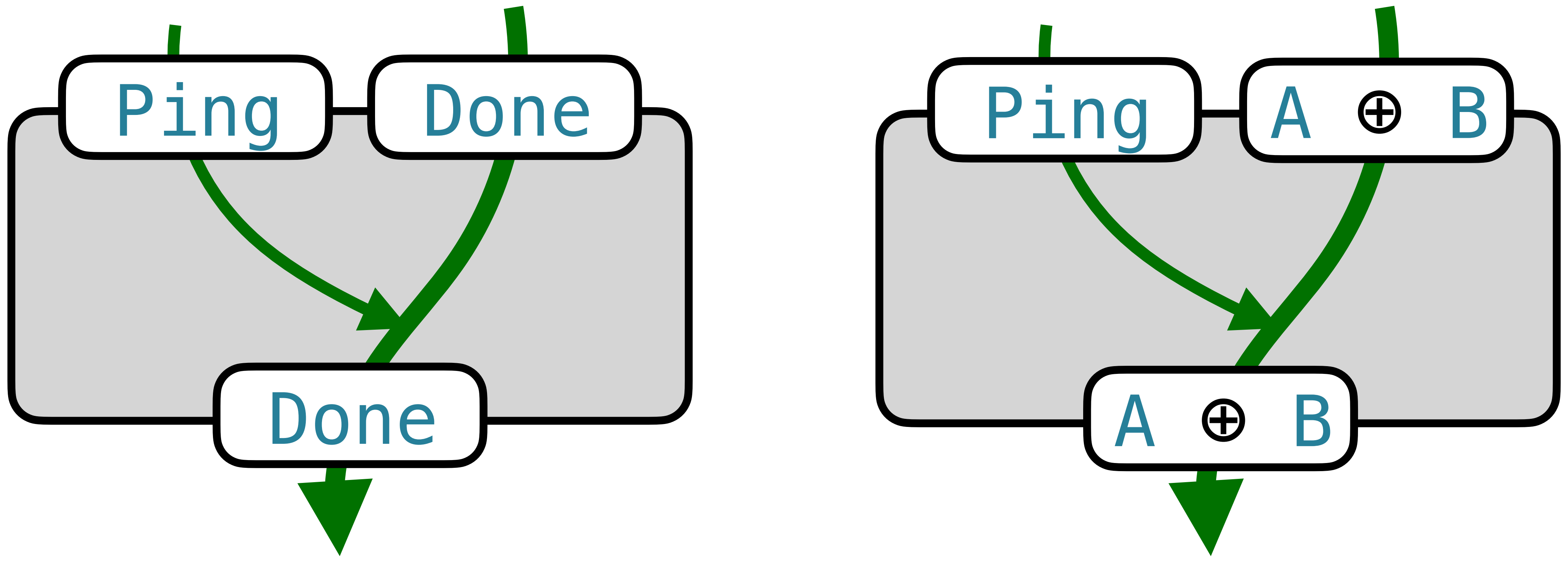
## Ping introduction (e.g.)



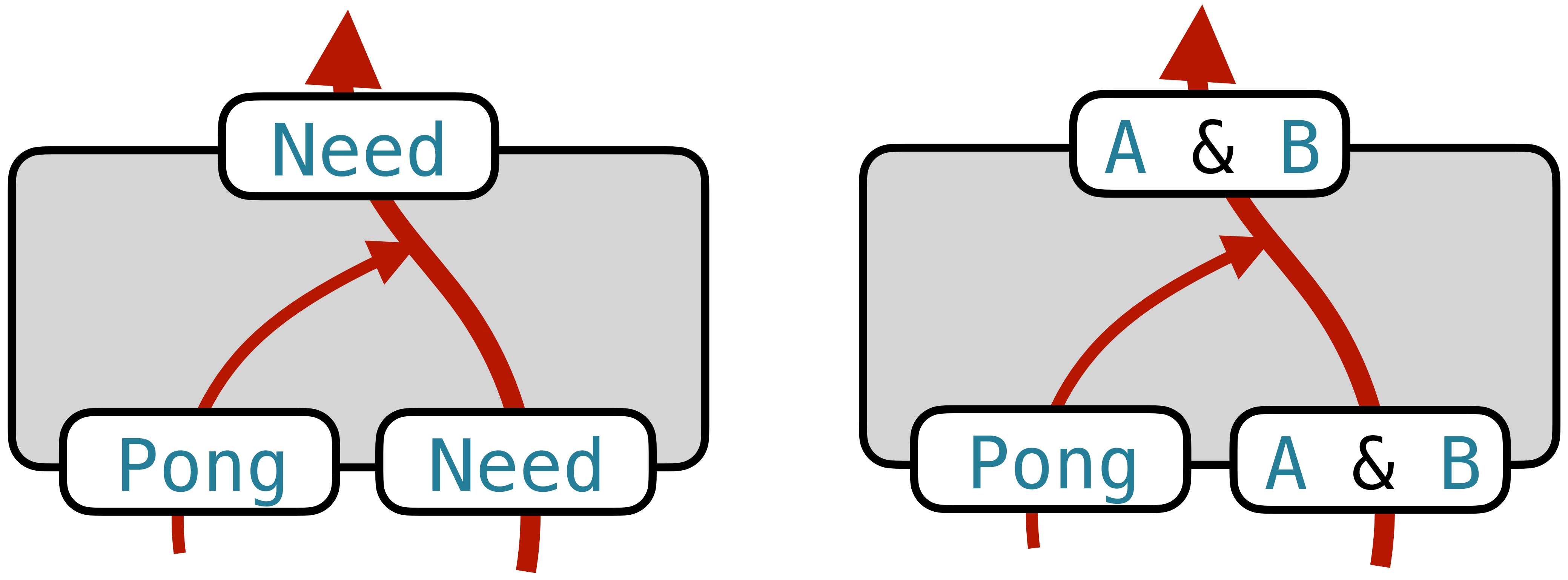

## Pong introduction (e.g.)




## Ping elimination (e.g.)

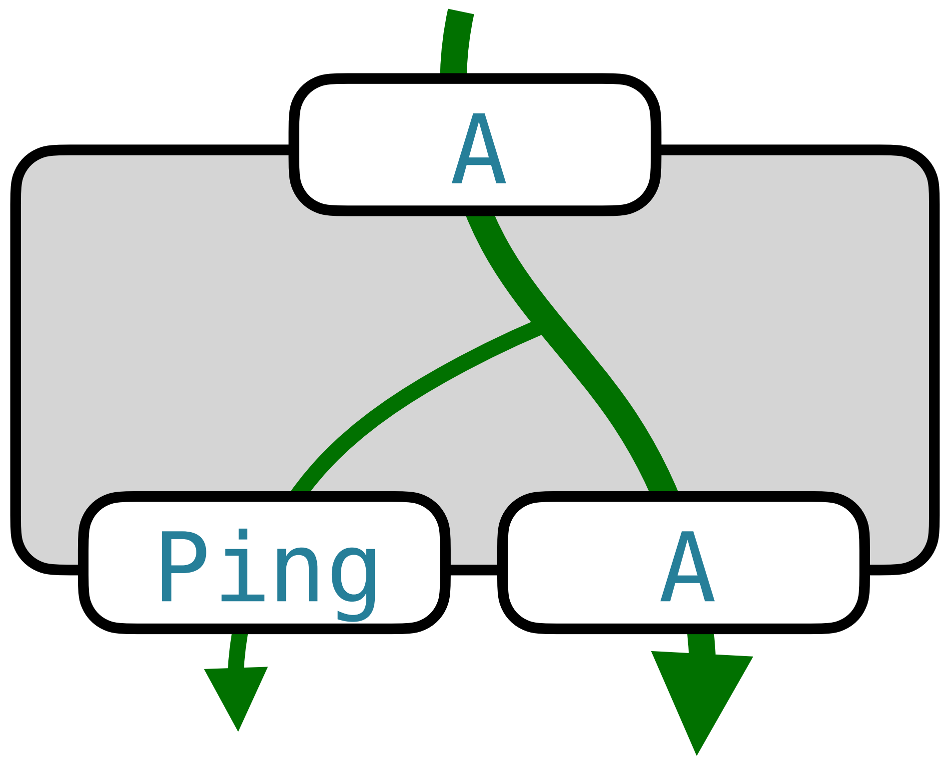



## Pong elimination (e.g.)

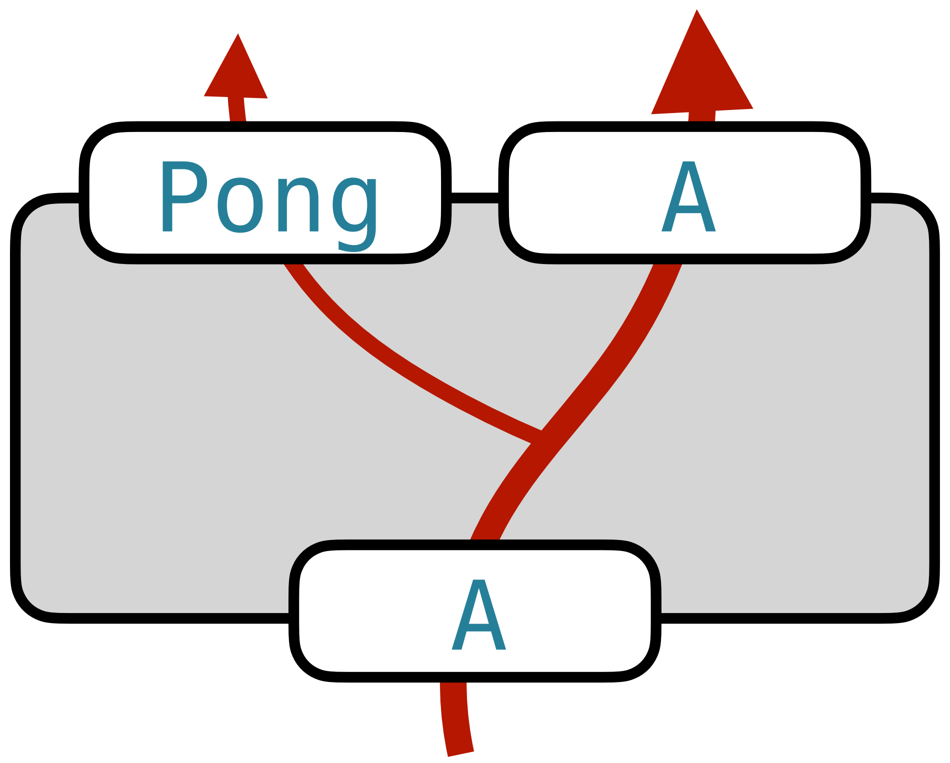


# Signals

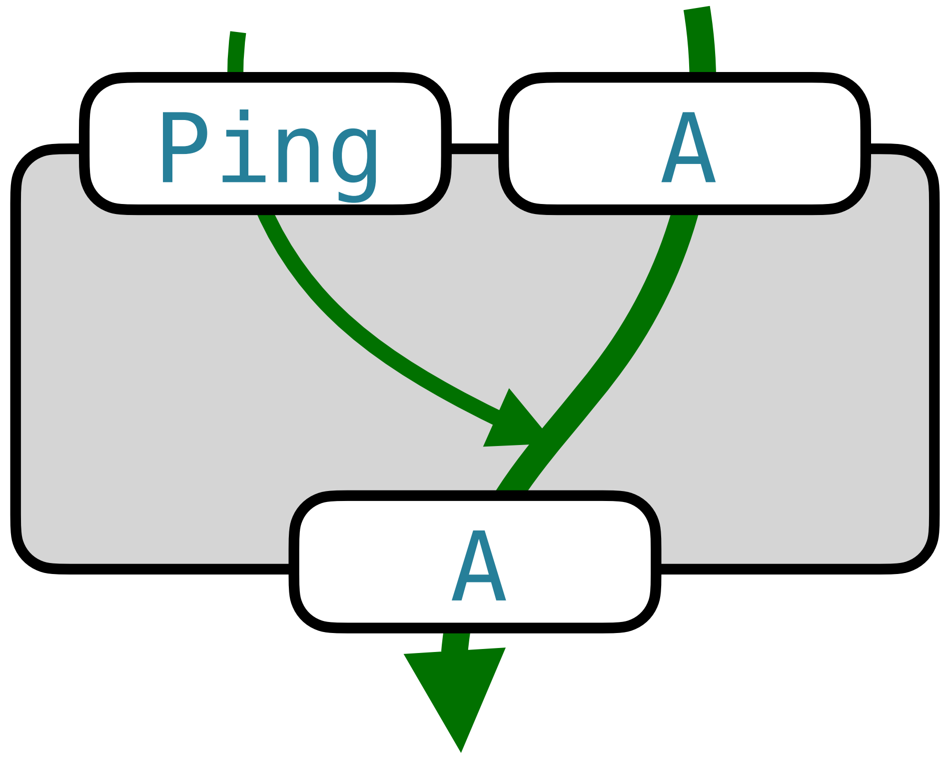
## Signaling.Positive[A]



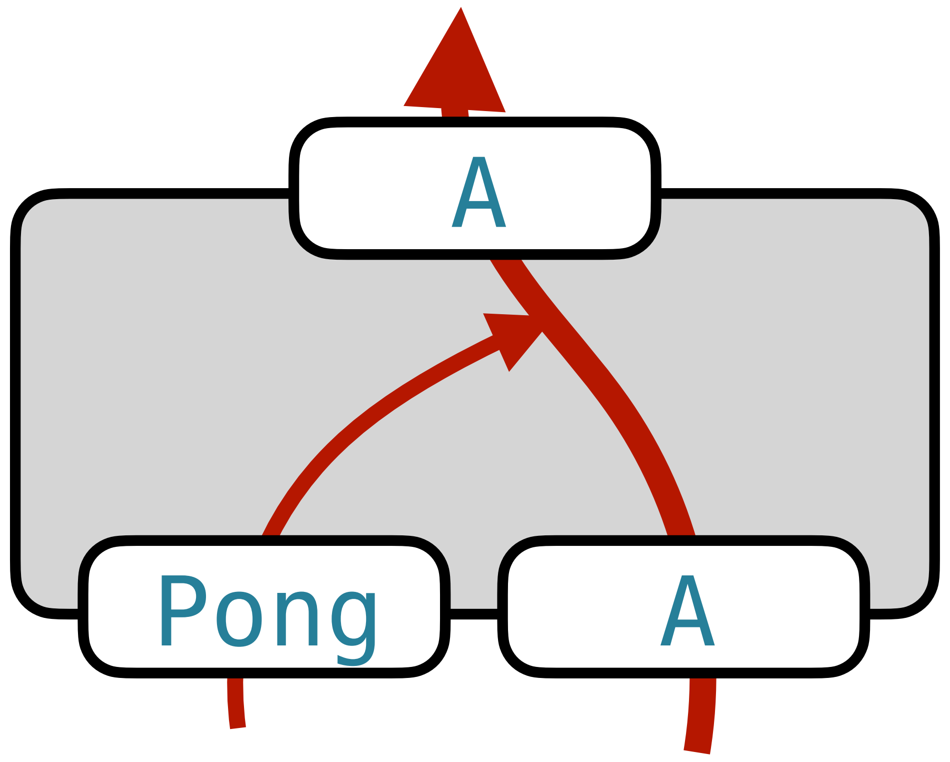

## Signaling.Negative[A]




## Deferrable.Positive[A]

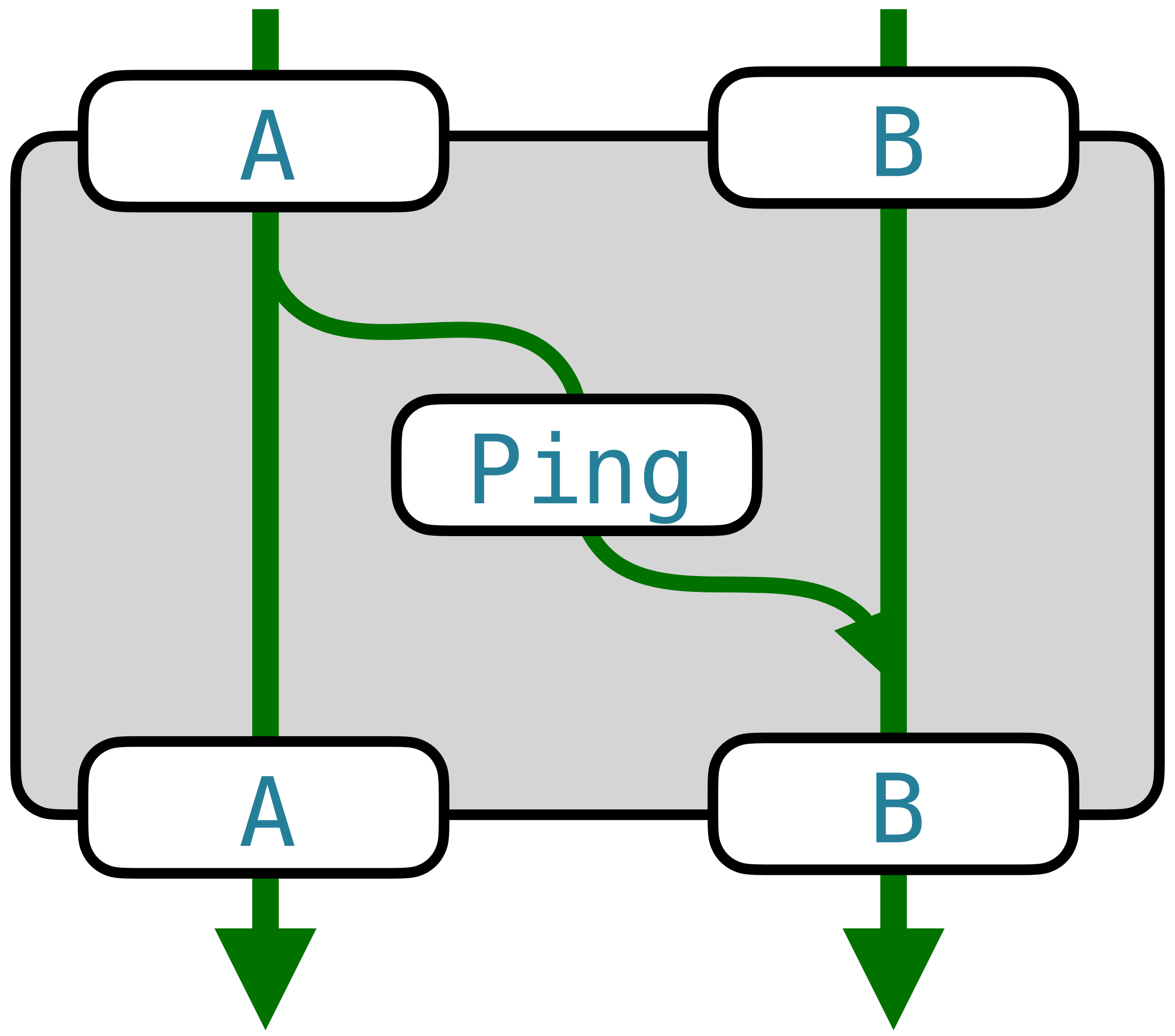



## Deferrable.Negative[A]



# Sequencing

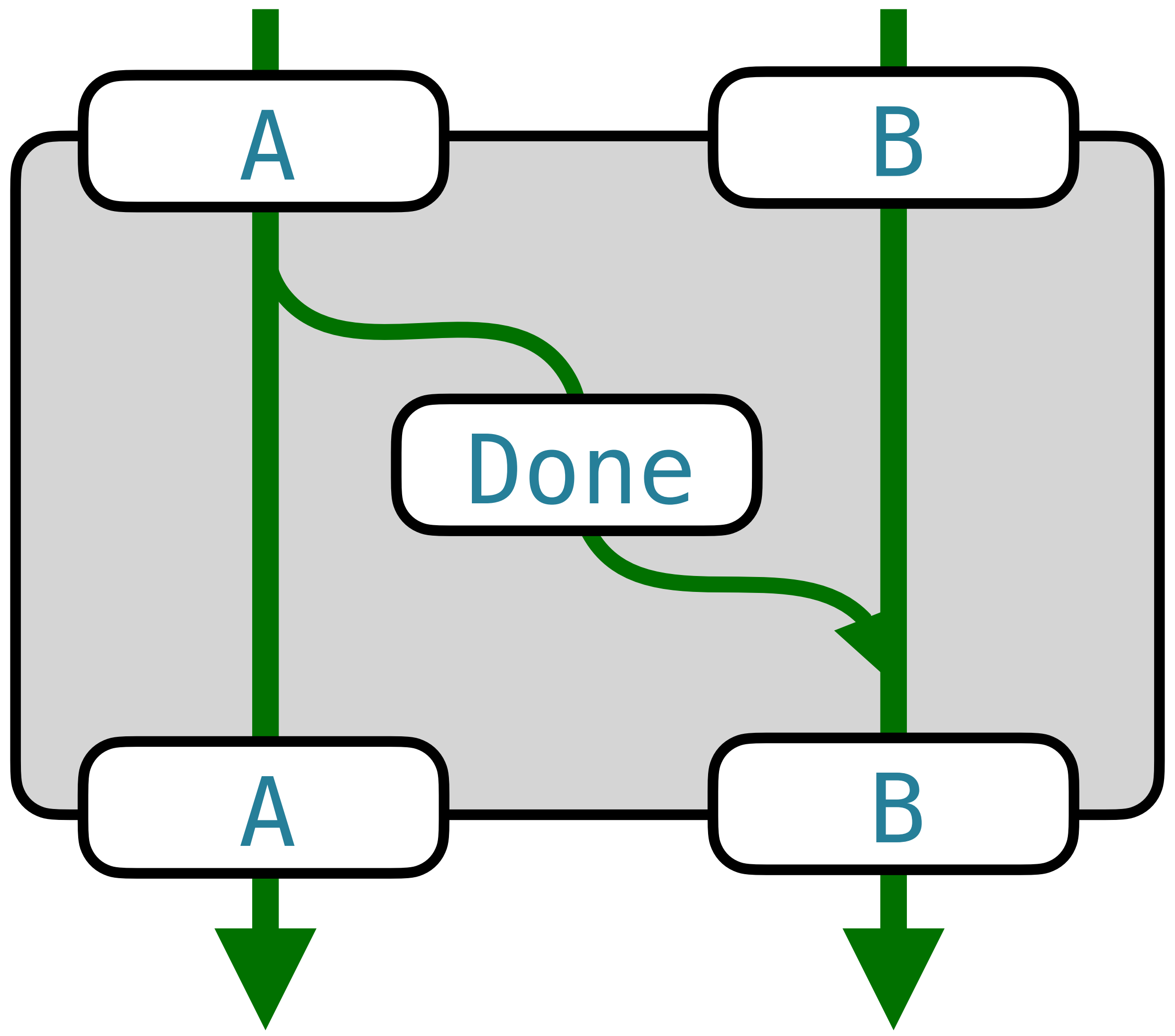
Signaling.Positive[A]




Deferrable.Positive[B]

# Sequencing

Signaling.Positive[A]



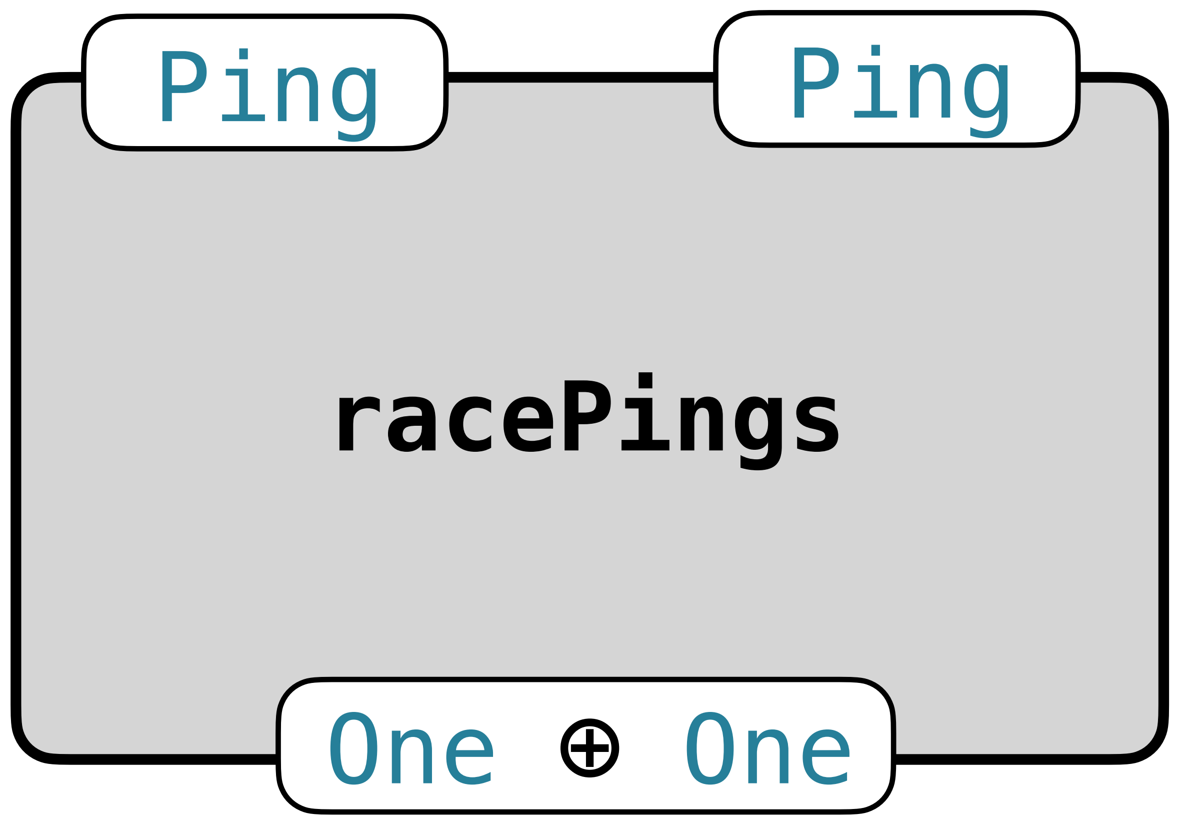

Junction.Positive[A]

# Racing

- Test which of two concurrent events occurred first
- Source of non-determinism

# Racing

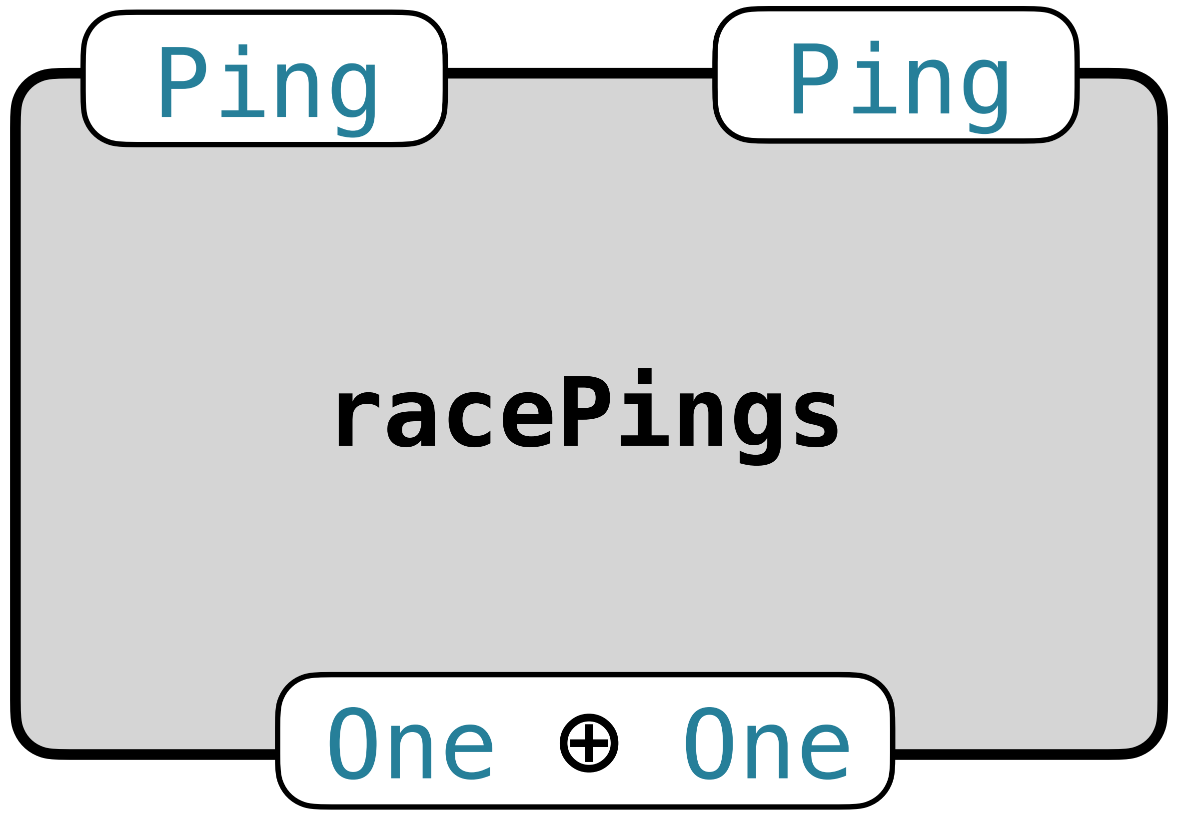
- Test which of two concurrent events occurred first
- Source of non-determinism

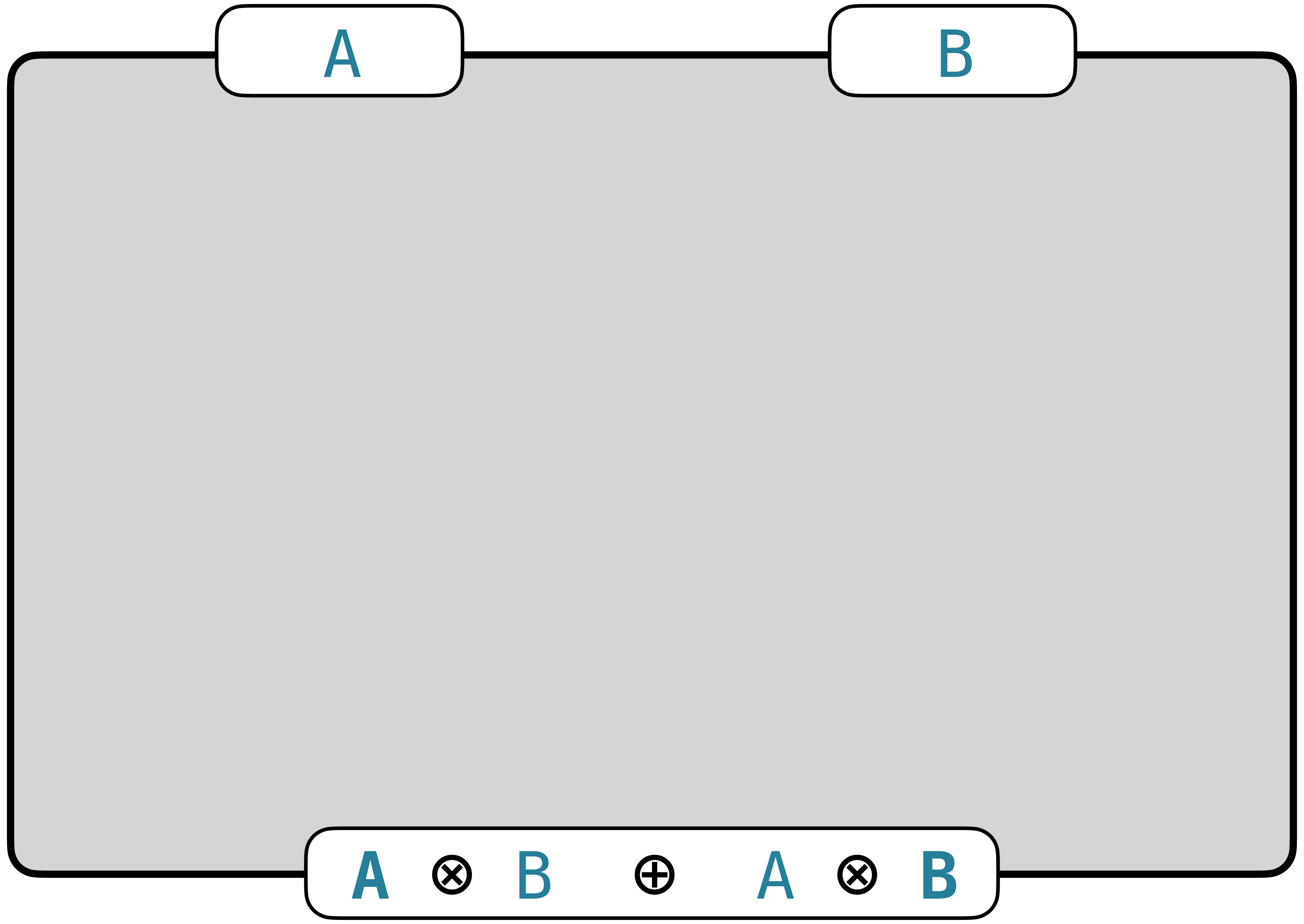


# Racing

- Test which of two concurrent events occurred first
- Source of non-determinism

```
def race[A, B](using
  Signaling.Positive[A],
  Signaling.Positive[B],
) =
```

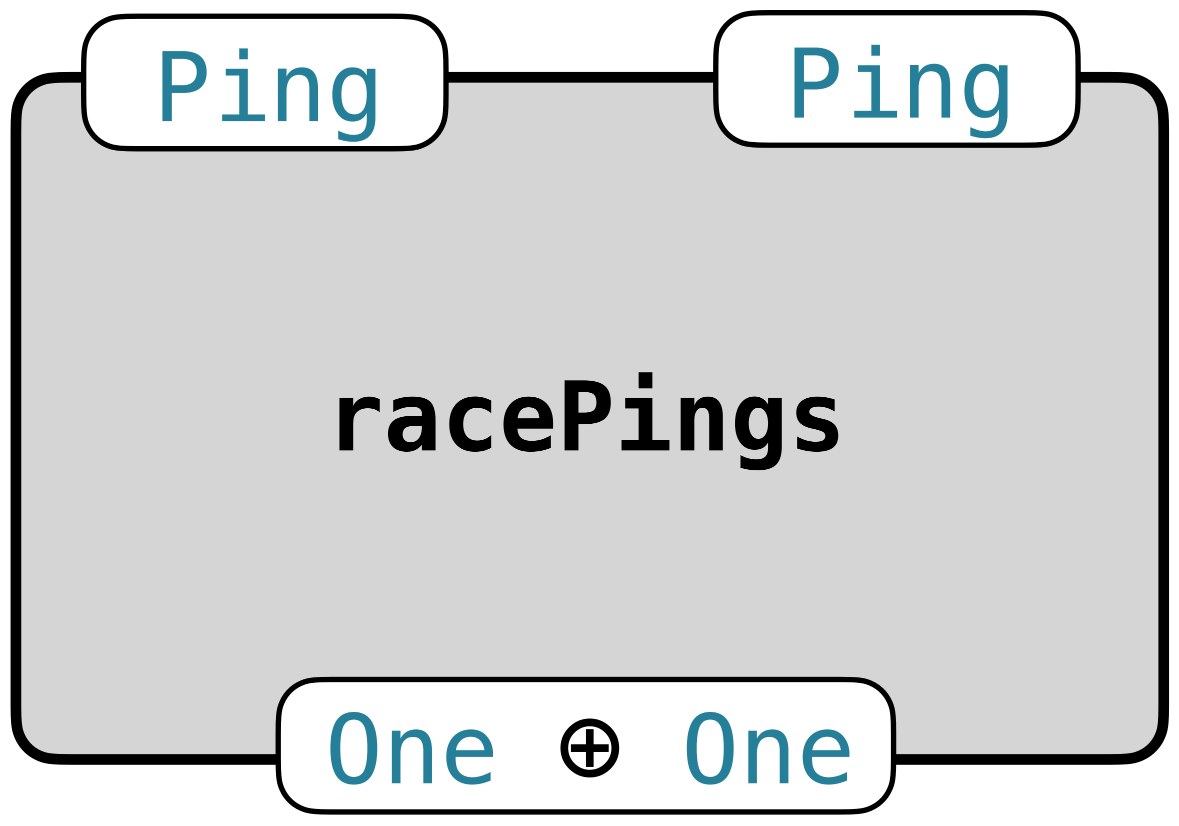


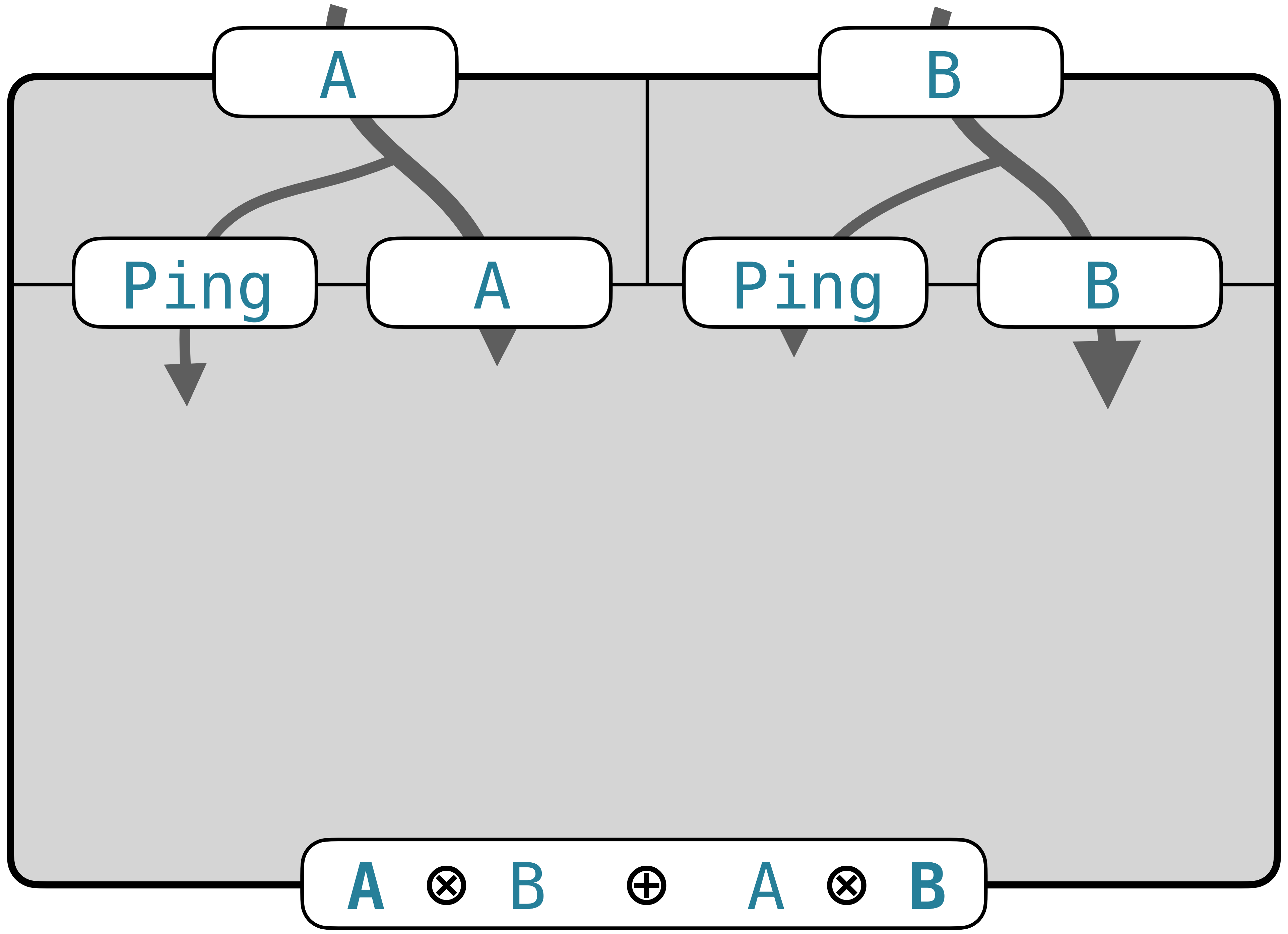



# Racing

- Test which of two concurrent events occurred first
- Source of non-determinism

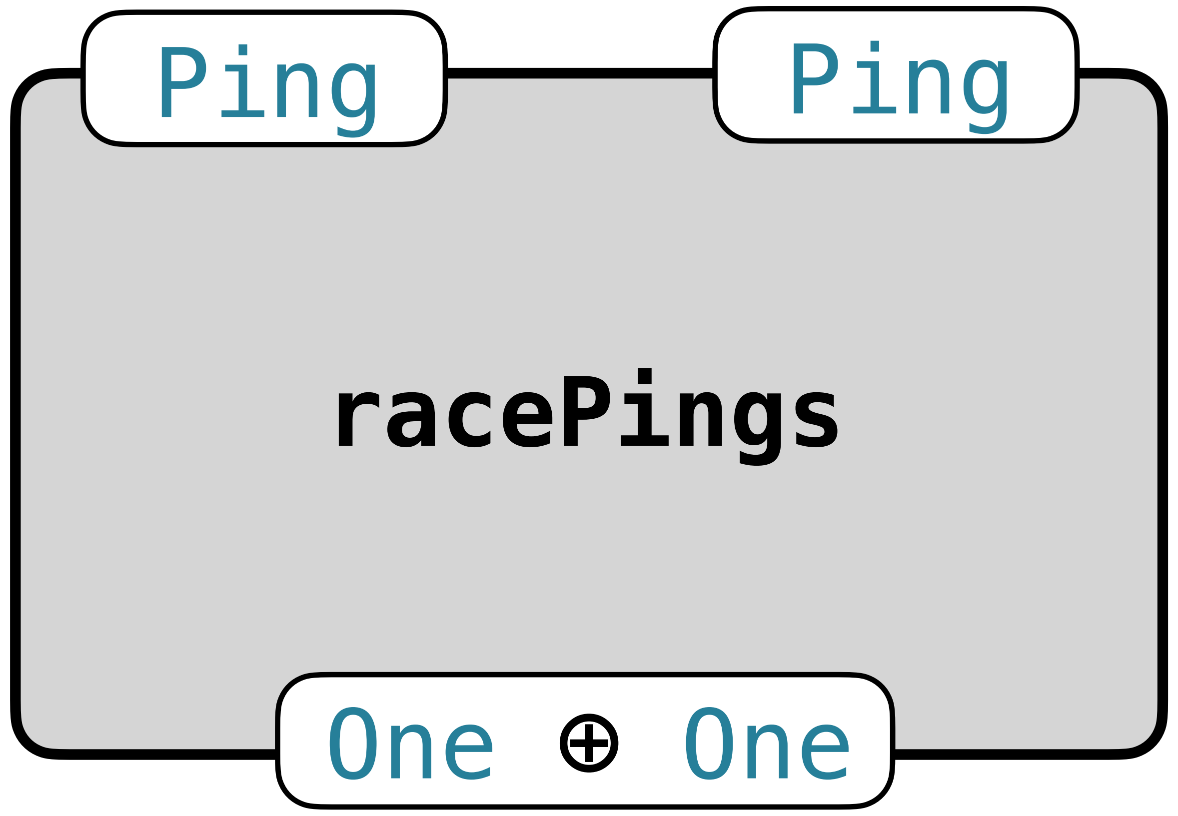
```
def race[A, B](using
  Signaling.Positive[A],
  Signaling.Positive[B],
) =
```



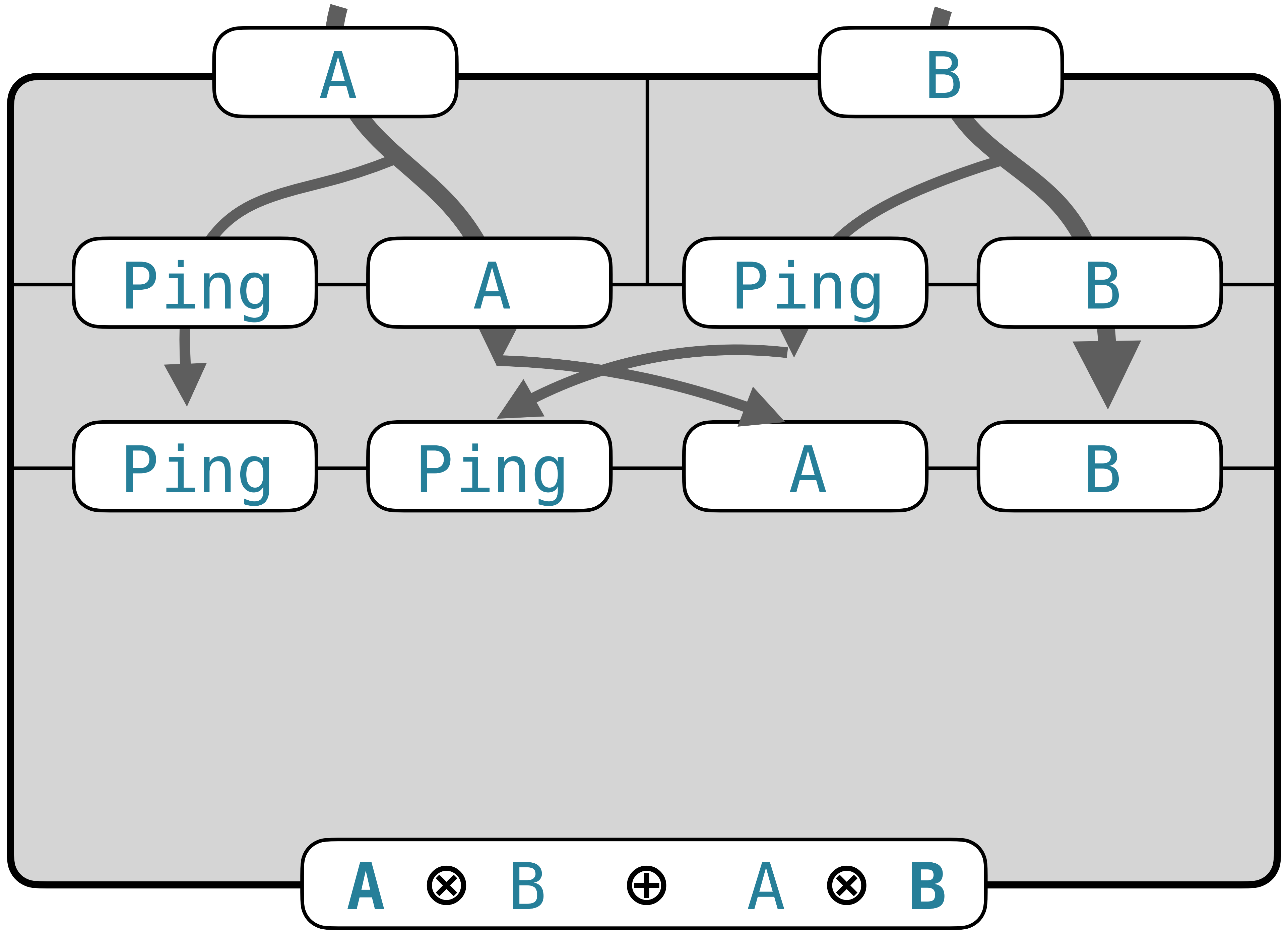



# Racing

- Test which of two concurrent events occurred first
- Source of non-determinism

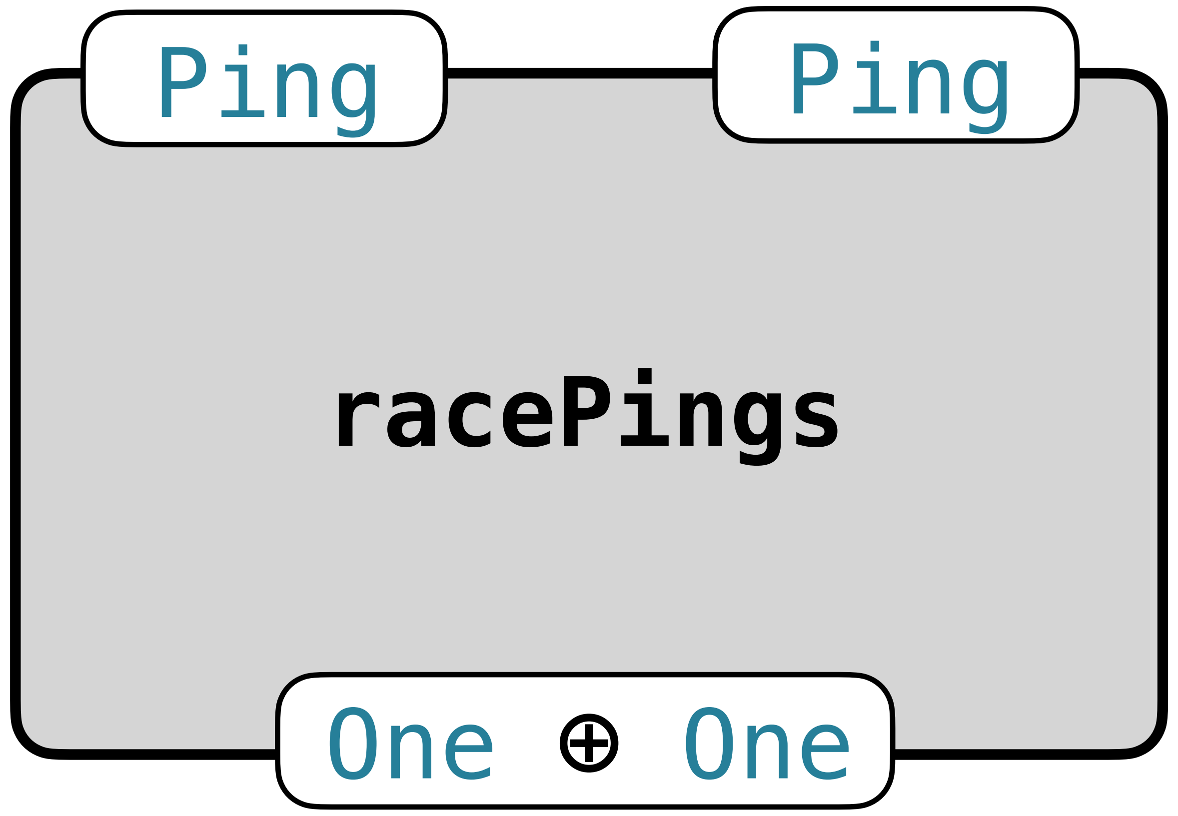



```
def race[A, B](using
  Signaling.Positive[A],
  Signaling.Positive[B],
) =
```

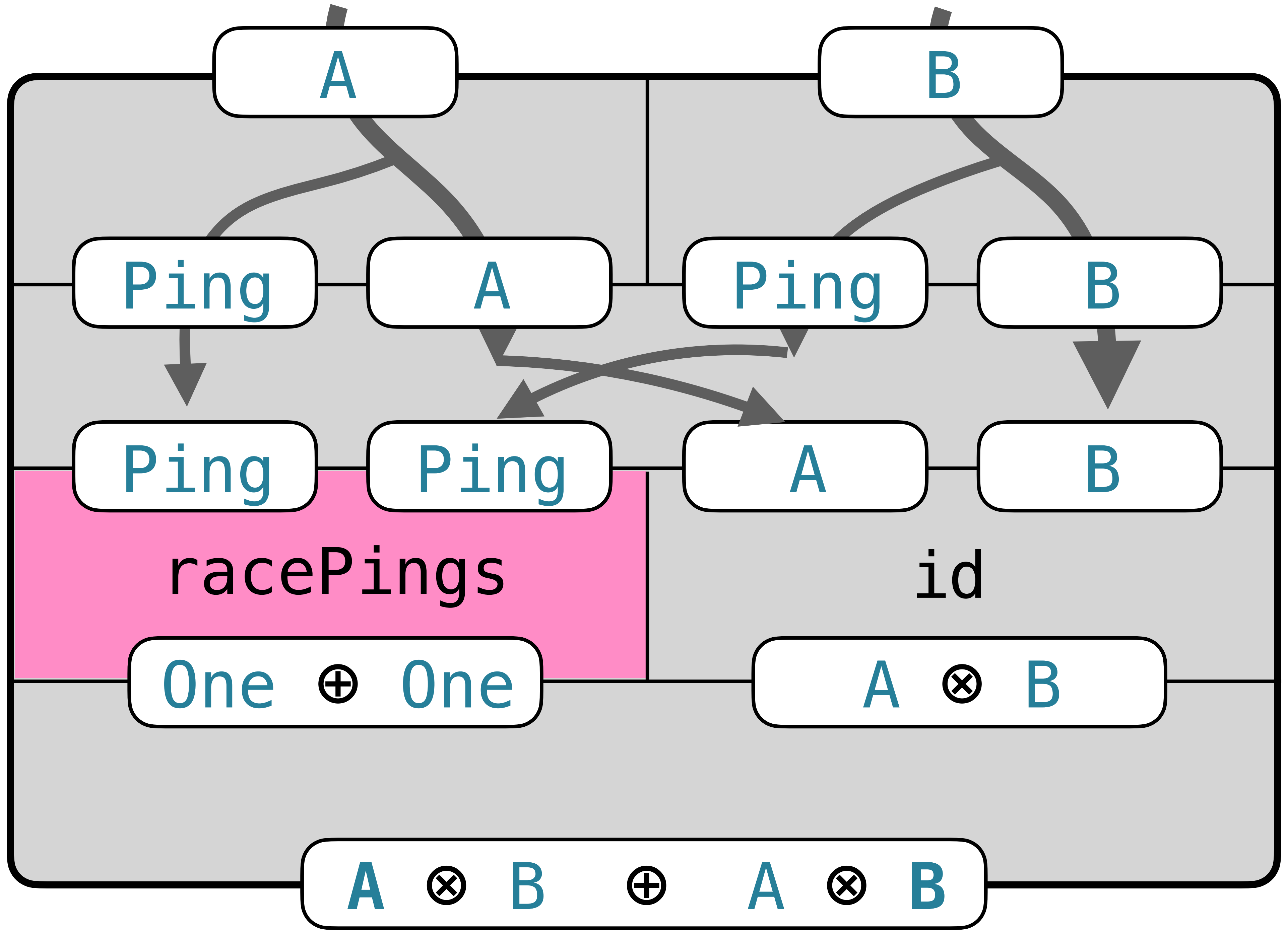


# Racing

- Test which of two concurrent events occurred first
- Source of non-determinism



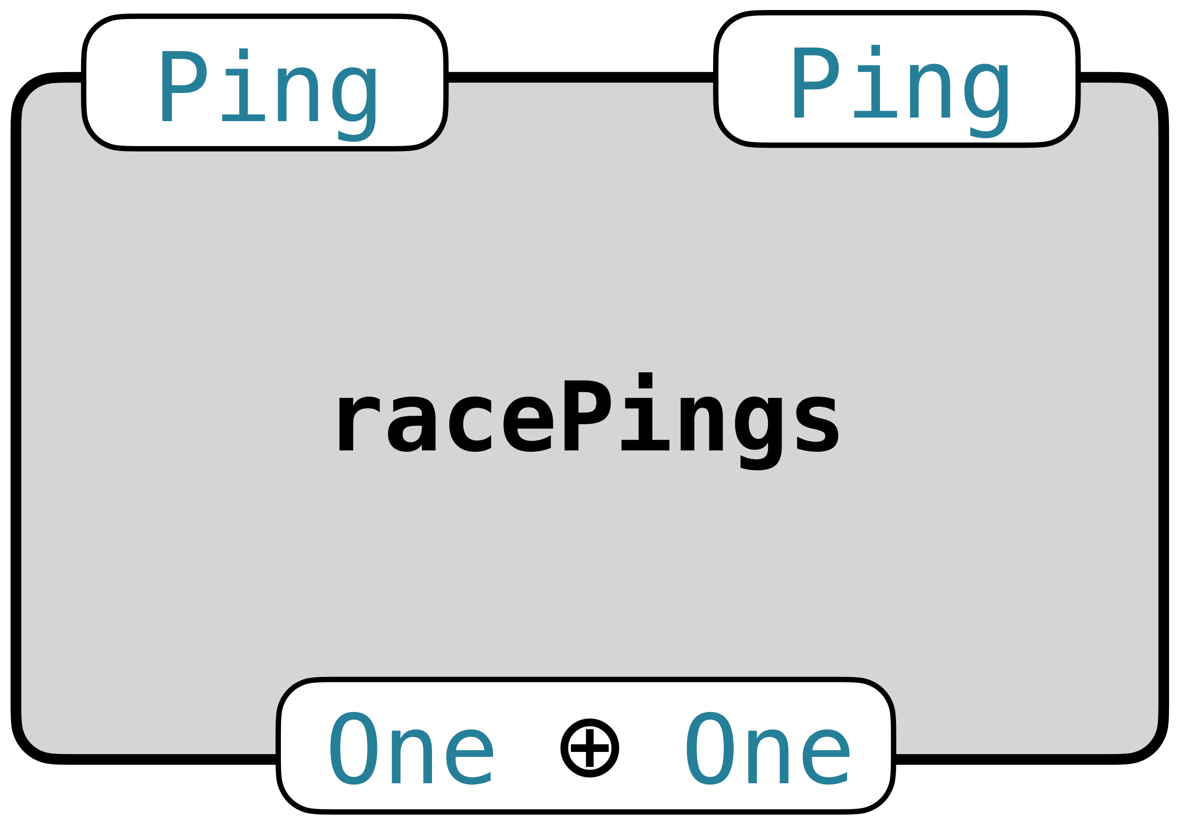

```
def race[A, B](using
  Signaling.Positive[A],
  Signaling.Positive[B],
) =
```

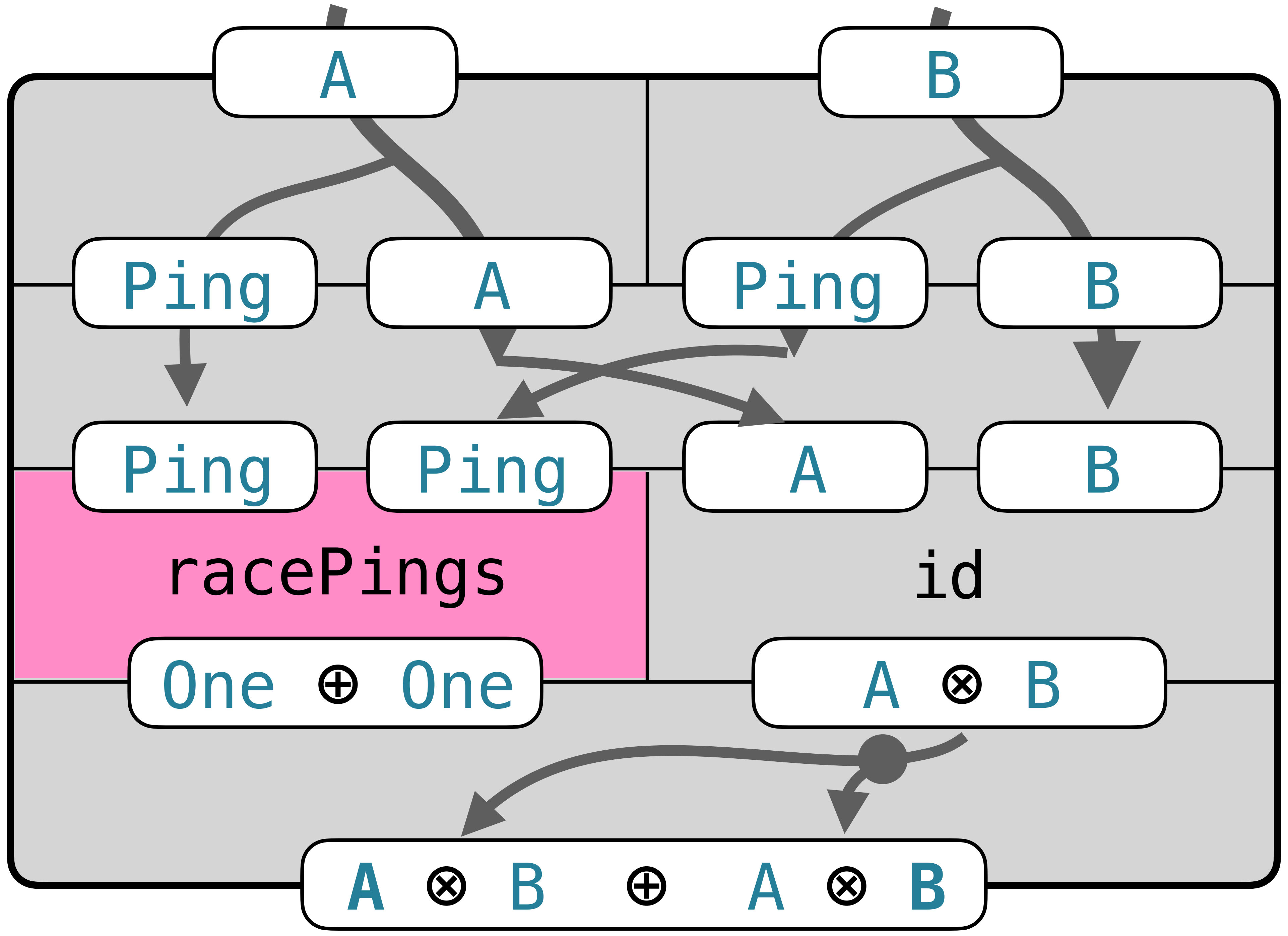


# Racing

- Test which of two concurrent events occurred first
- Source of non-determinism

```
def race[A, B](using
  Signaling.Positive[A],
  Signaling.Positive[B],
) =
```



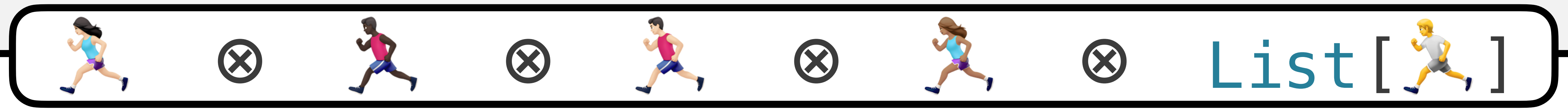
# `List.sortBySignal`

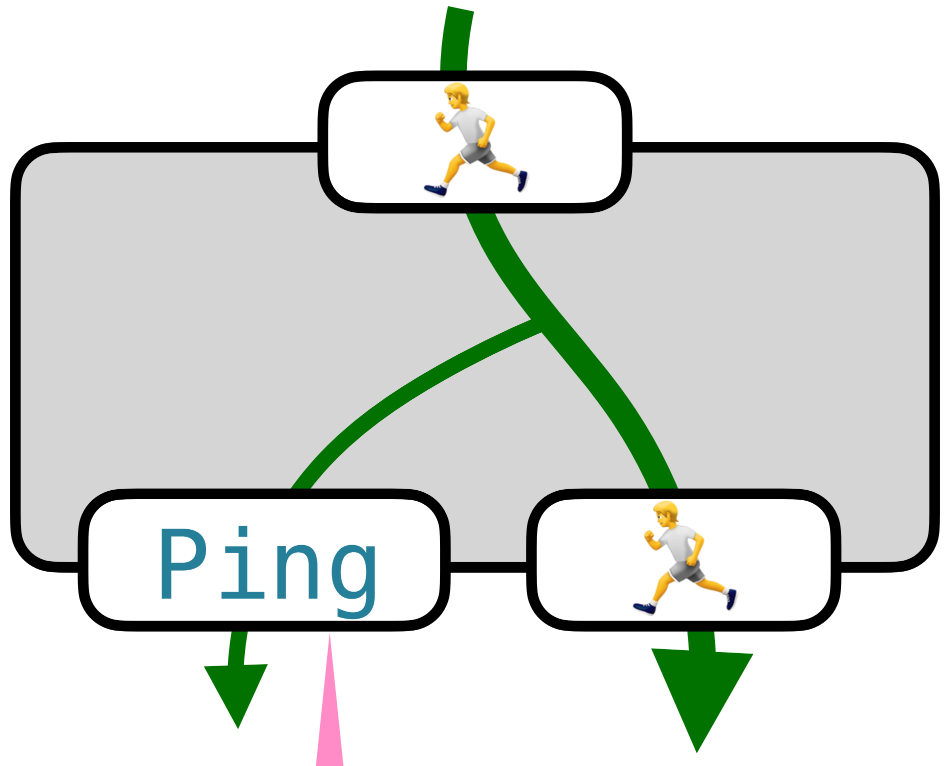
Runners added to the list
as they **register** for the marathon.

🏃‍♀️ ⊗ 🏃🏿 ⊗ 🏃‍♂️ ⊗ 🏃🏽‍♀️ ⊗ `List[🏃]`

# List.sortBySignal

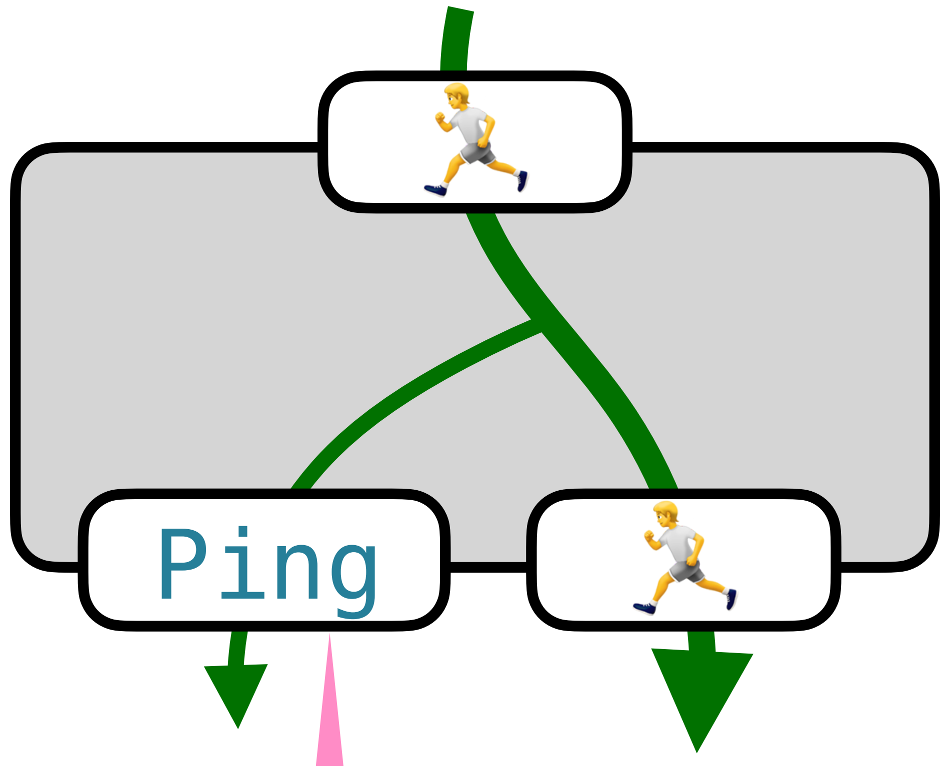
Runners added to the list
as they **register** for the marathon.



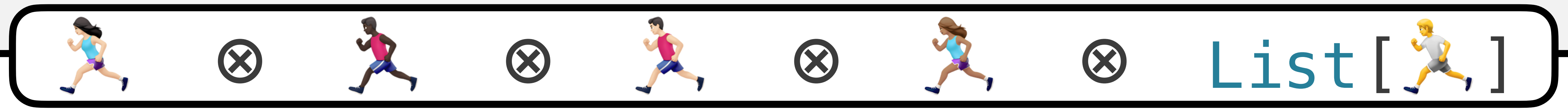

signals when the runner
**finished** the marathon

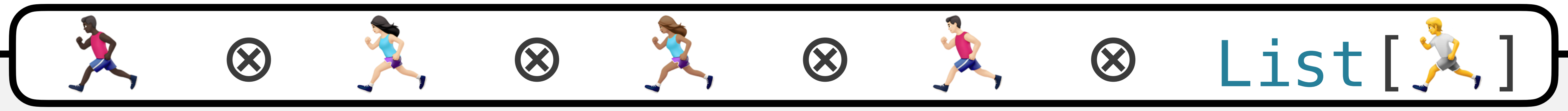
# List.sortBySignal




signals when the runner **finished** the marathon

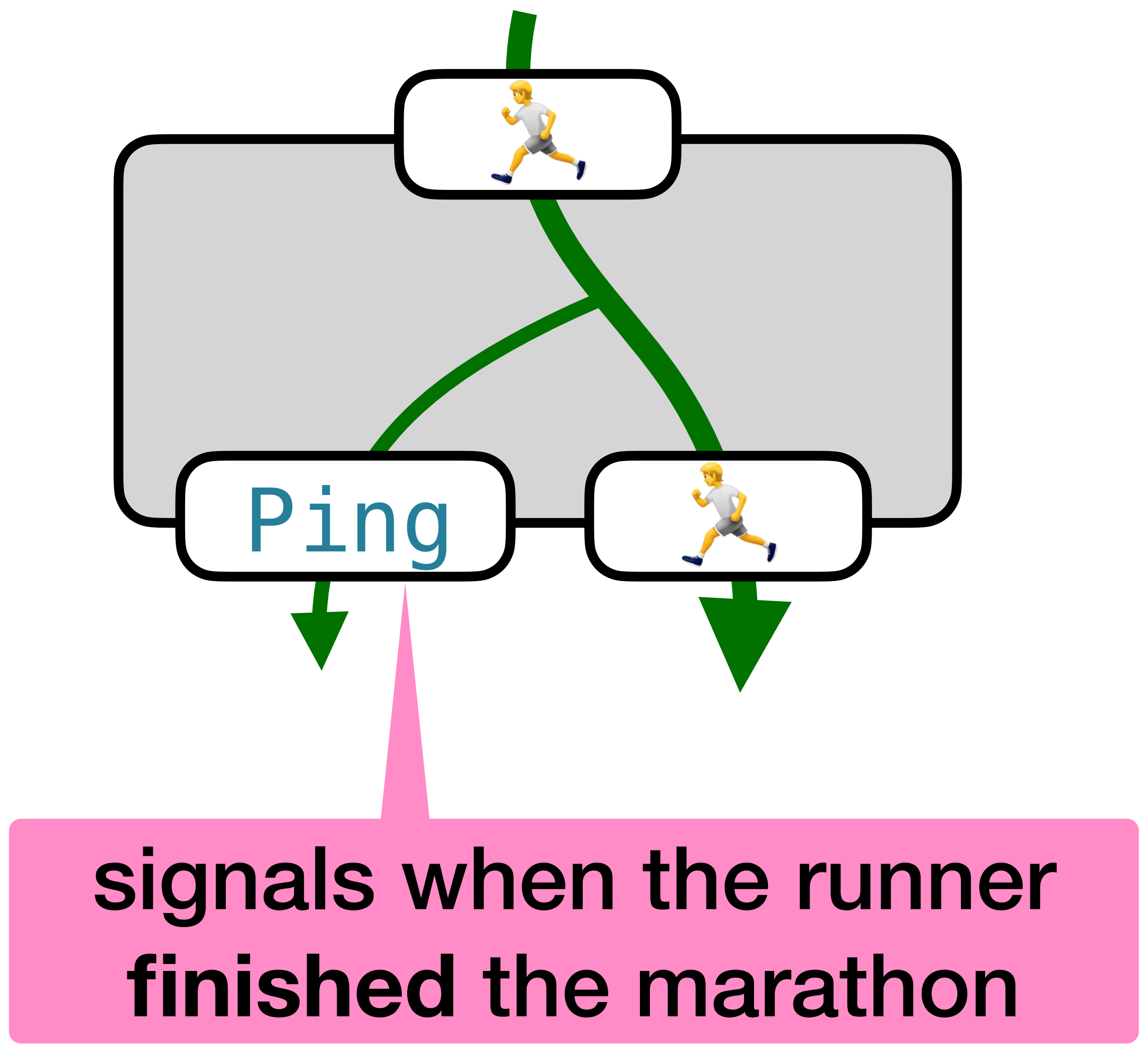
Runners added to the list
as they **register** for the marathon.

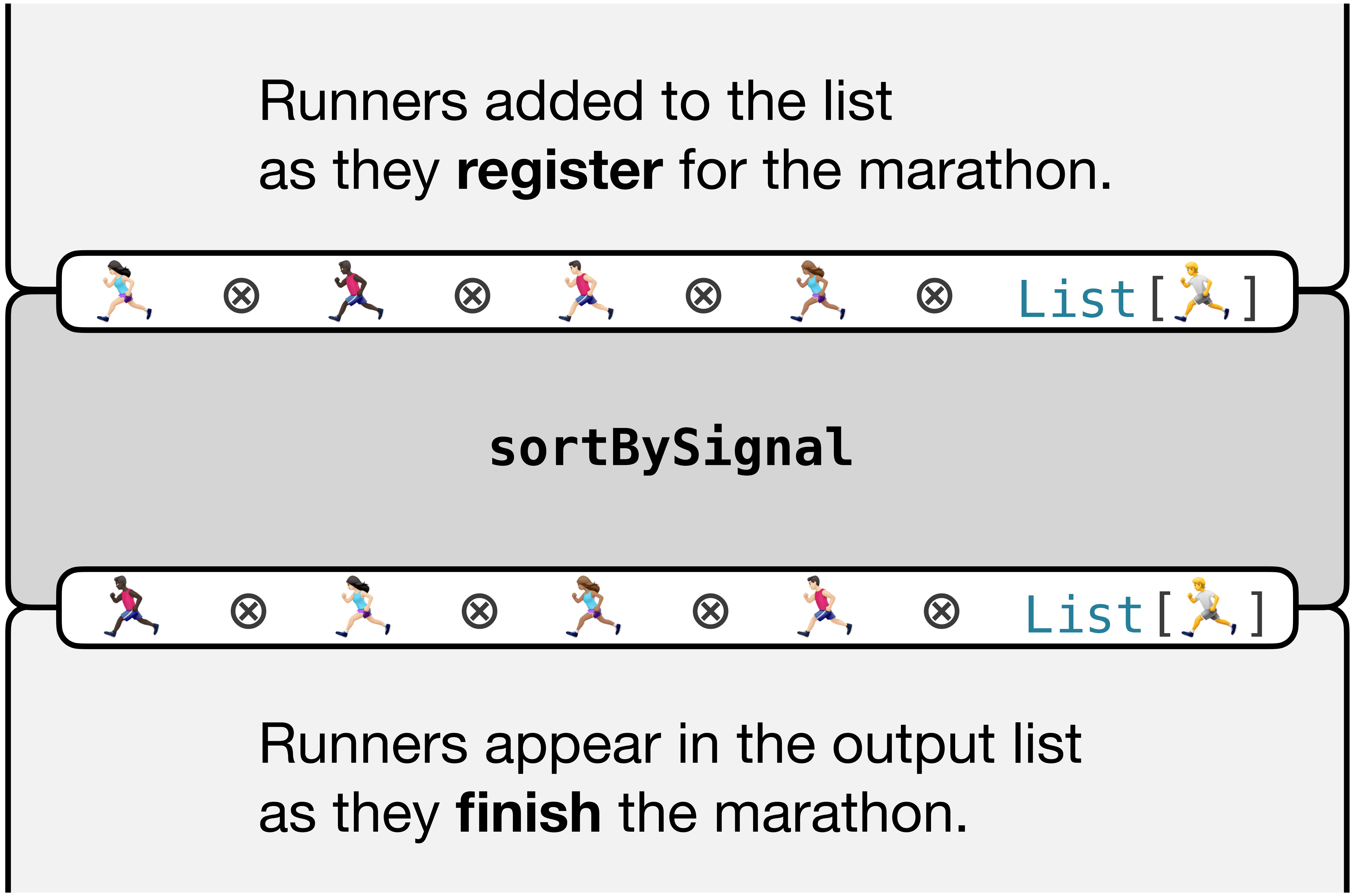


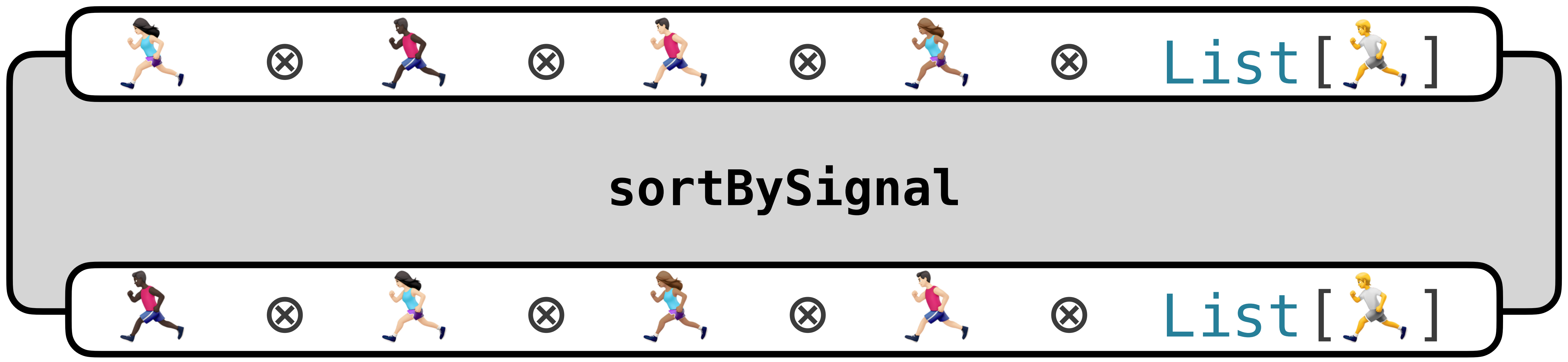

Runners appear in the output list
as they **finish** the marathon.

# List.sortBySignal

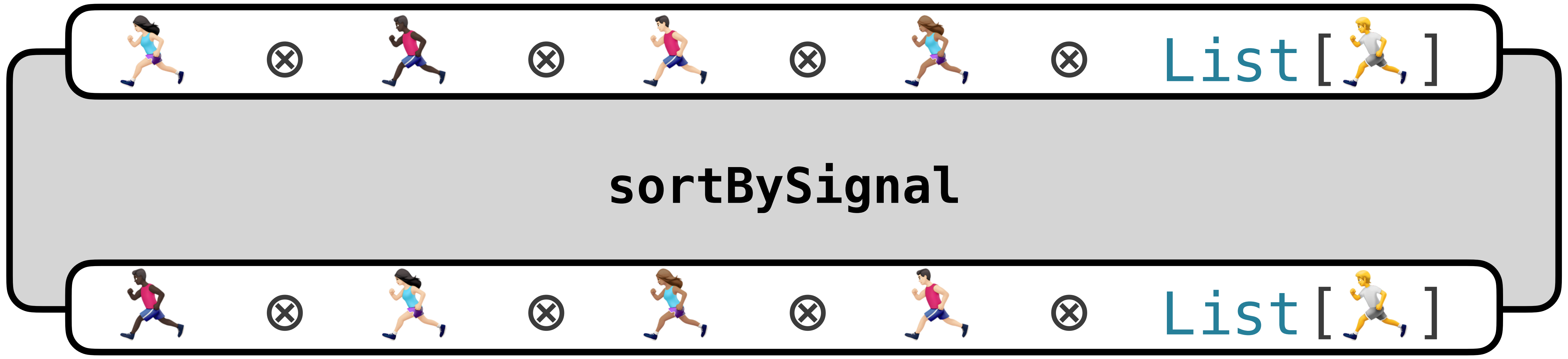


# List.sortBySignal

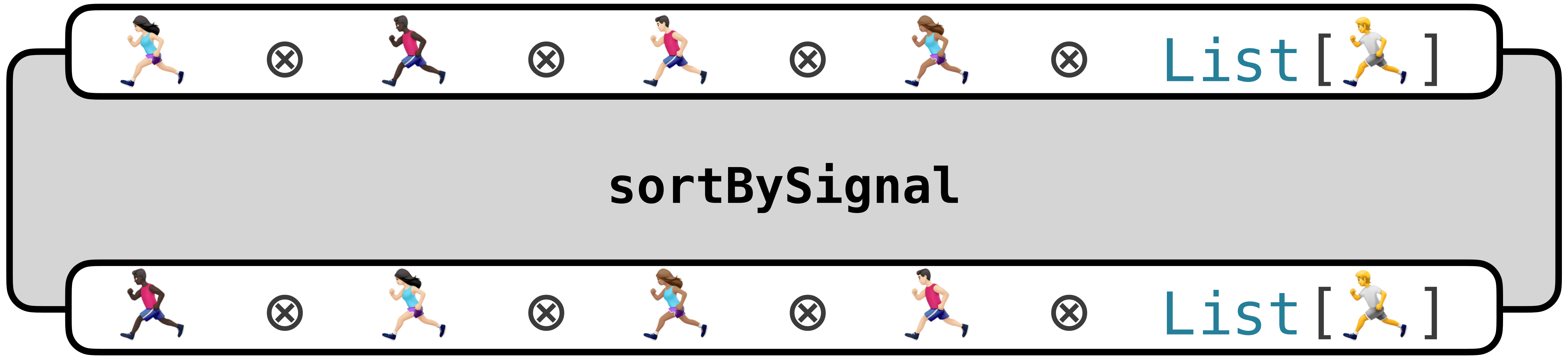


# List.sortBySignal




```
def sortBySignal[A](
 using Signaling.Positive[A]
): List[A] -∘ List[A] =
 rec { self =>
  λ { as =>
   uncons(as) switch {
    case Left(one) =>
     nil(one)
    case Right(a ⊗ as) =>
     insertBySignal(a ⊗ self(as))
 }}}
```

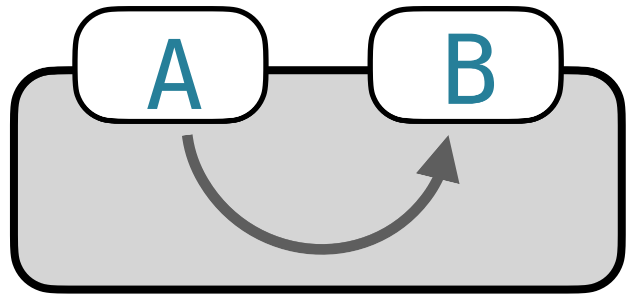
# List.sortBySignal

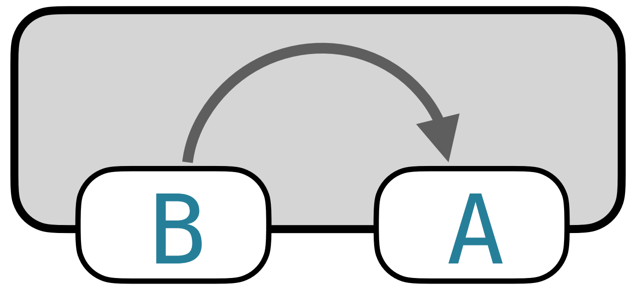



```
def sortBySignal[A](
 using Signaling.Positive[A]
): List[A] -∘ List[A] =
 rec { self =>
  λ { as =>
   uncons(as) switch {
    case Left(one) =>
     nil(one)
    case Right(a ⊗ as) =>
     insertBySignal(a ⊗ self(as))
 }}}
```

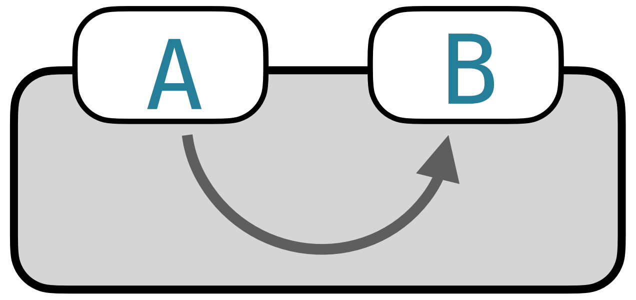
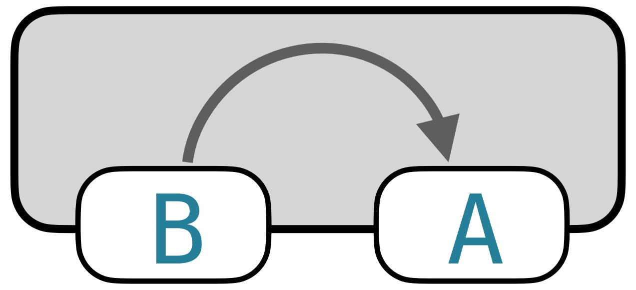
```
def insertBySignal[A](
 using Signaling.Positive[A]
): (A ⊗ List[A]) -∘ List[A] =
 rec { self =>
  λ { case a ⊗ as =>
   race(a ⊗ as) switch {
    case Left(a ⊗ as) =>
     cons(a ⊗ as)
    case Right(a ⊗ as) =>
     uncons(as) switch {
      case Left(?(one)) =>
       singletonOnSignal(a)
      case Right(a1 ⊗ as) =>
       cons(a1 ⊗ self(a ⊗ as))
 }}}}
```

# Duals

B is the *dual* of A if there exist 

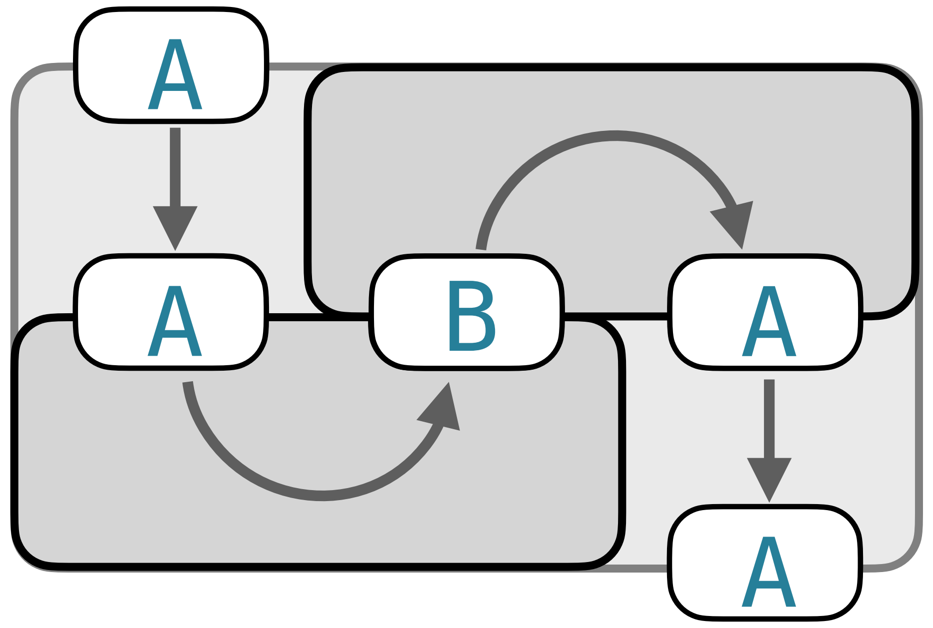
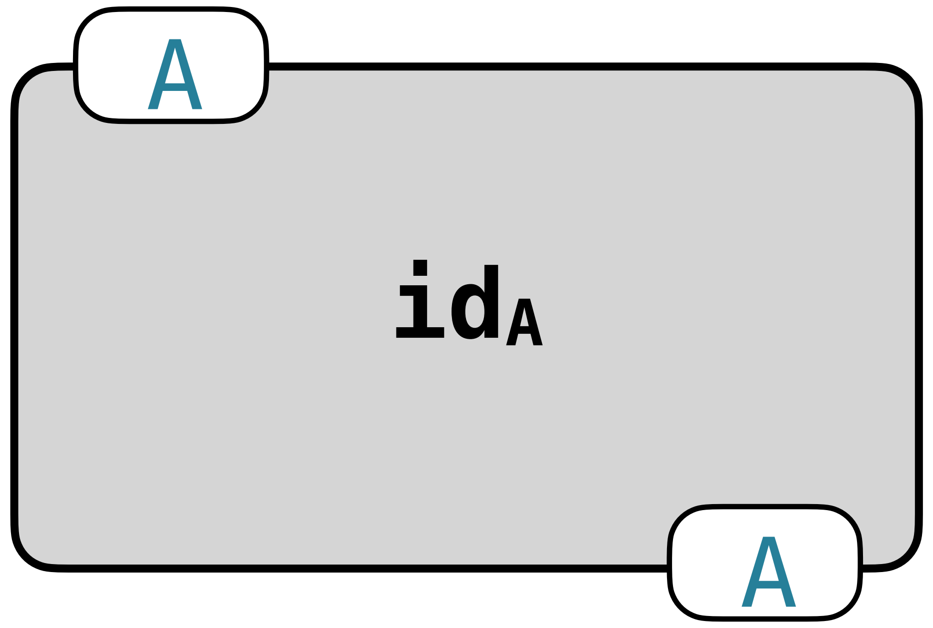
and 

# Duals

B is the *dual* of A if there exist  and 

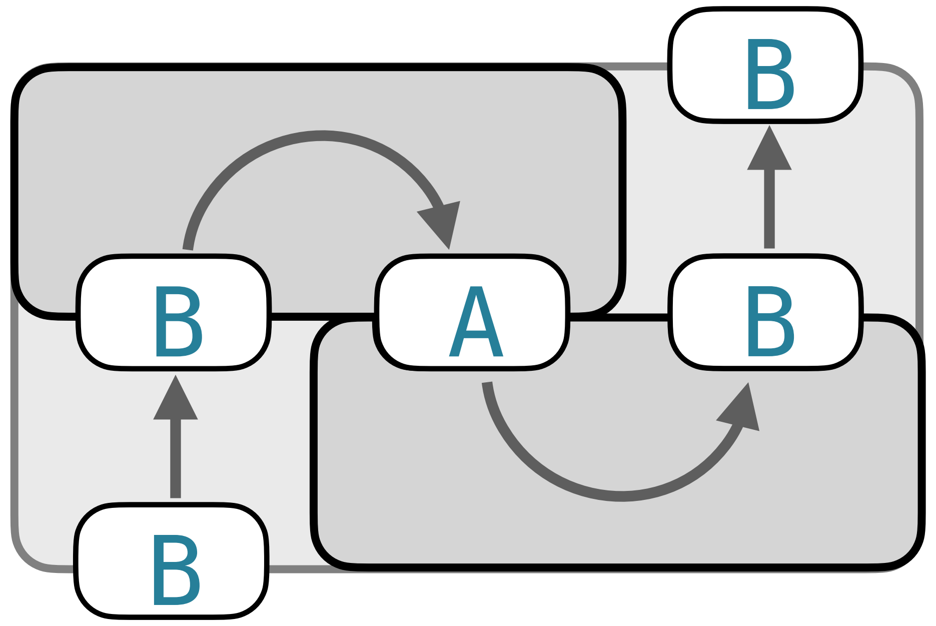
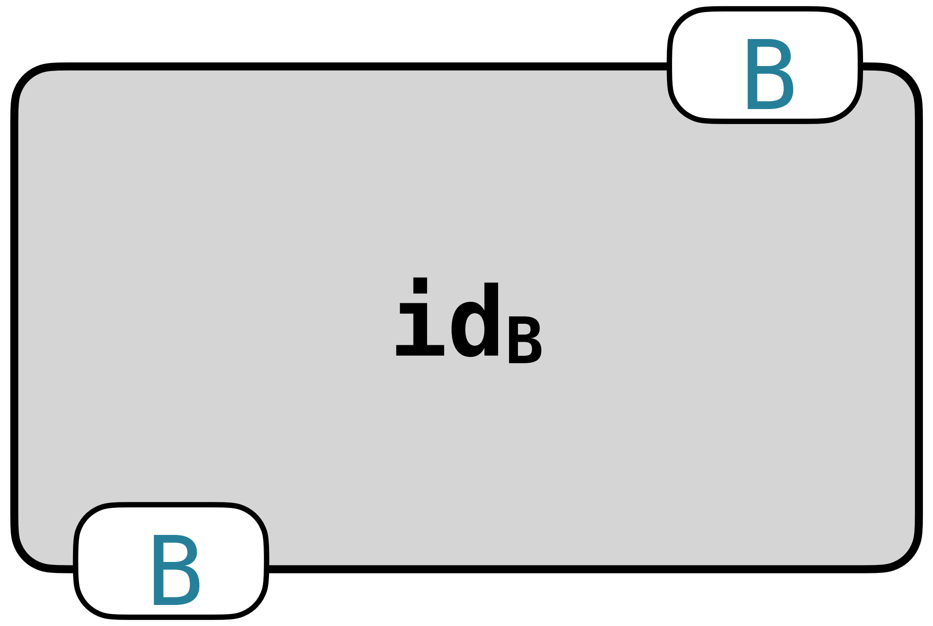




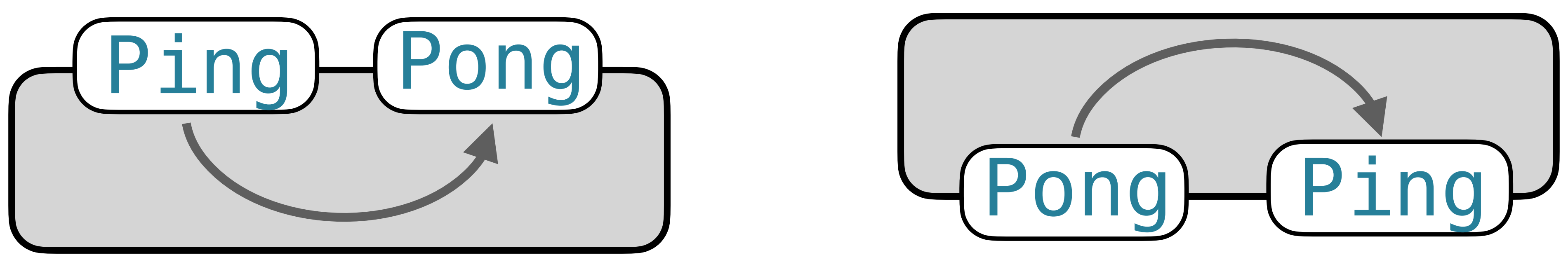
such that  = 



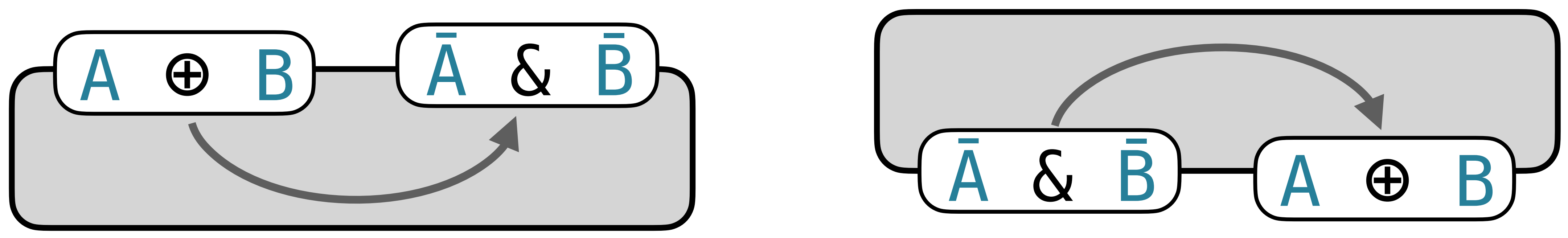


and  = 

# Examples of **Duals**

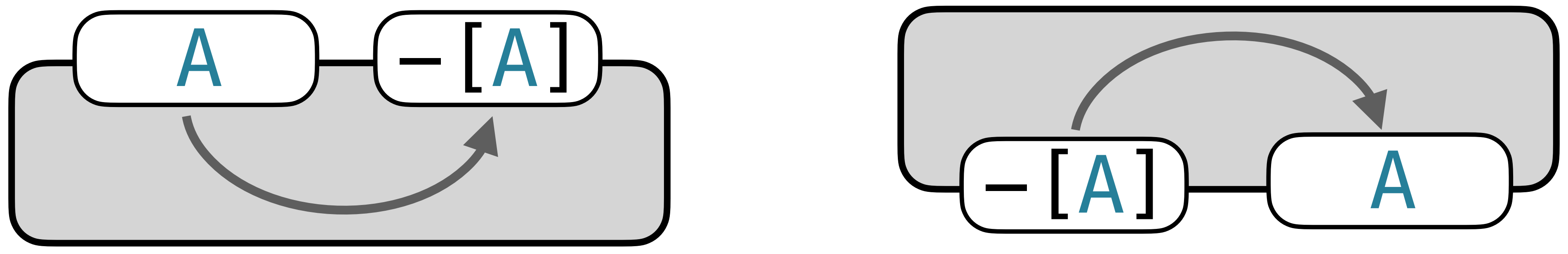



Given $\bar{A}$ dual of $A$, $\bar{B}$ dual of $B$



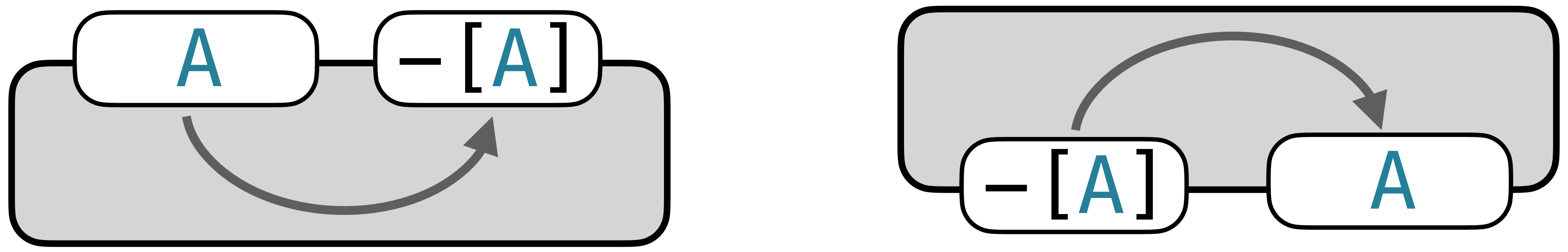
# Universal **Duals**

−[A]



# Universal Duals

−[A]

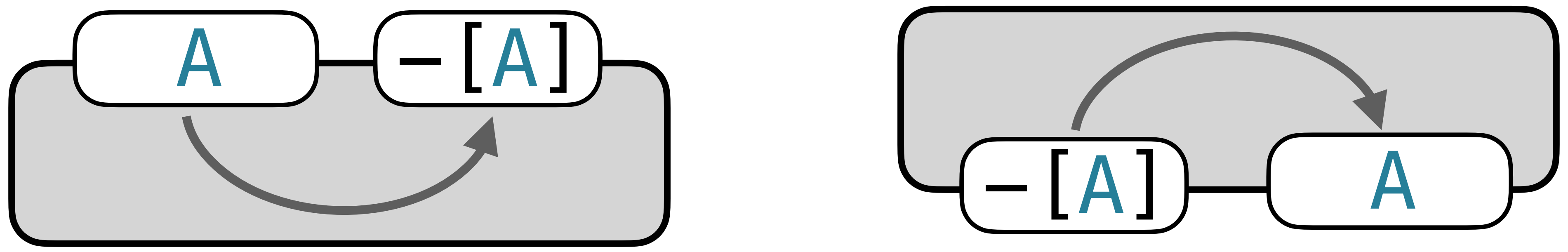



−[Ping] ≃ Pong

−[Pong] ≃ Ping

# Universal Duals

−[A]




−[Ping] ≃ Pong

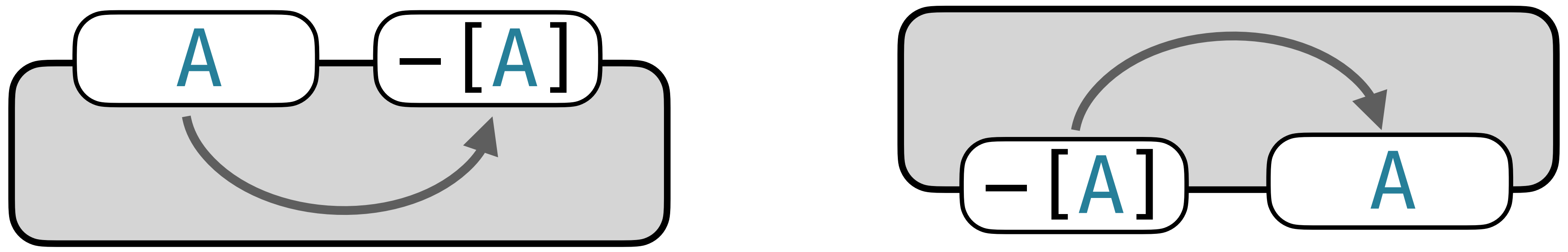
−[Pong] ≃ Ping

−[A ⊕ B] ≃ −[A] & −[B]

−[A & B] ≃ −[A] ⊕ −[B]

# Universal Duals

−[A]




−[Ping] ≃ Pong

−[Pong] ≃ Ping

−[A ⊕ B] ≃ −[A] & −[B]

−[A & B] ≃ −[A] ⊕ −[B]

−[List[A]] ≃ Endless[−[A]]

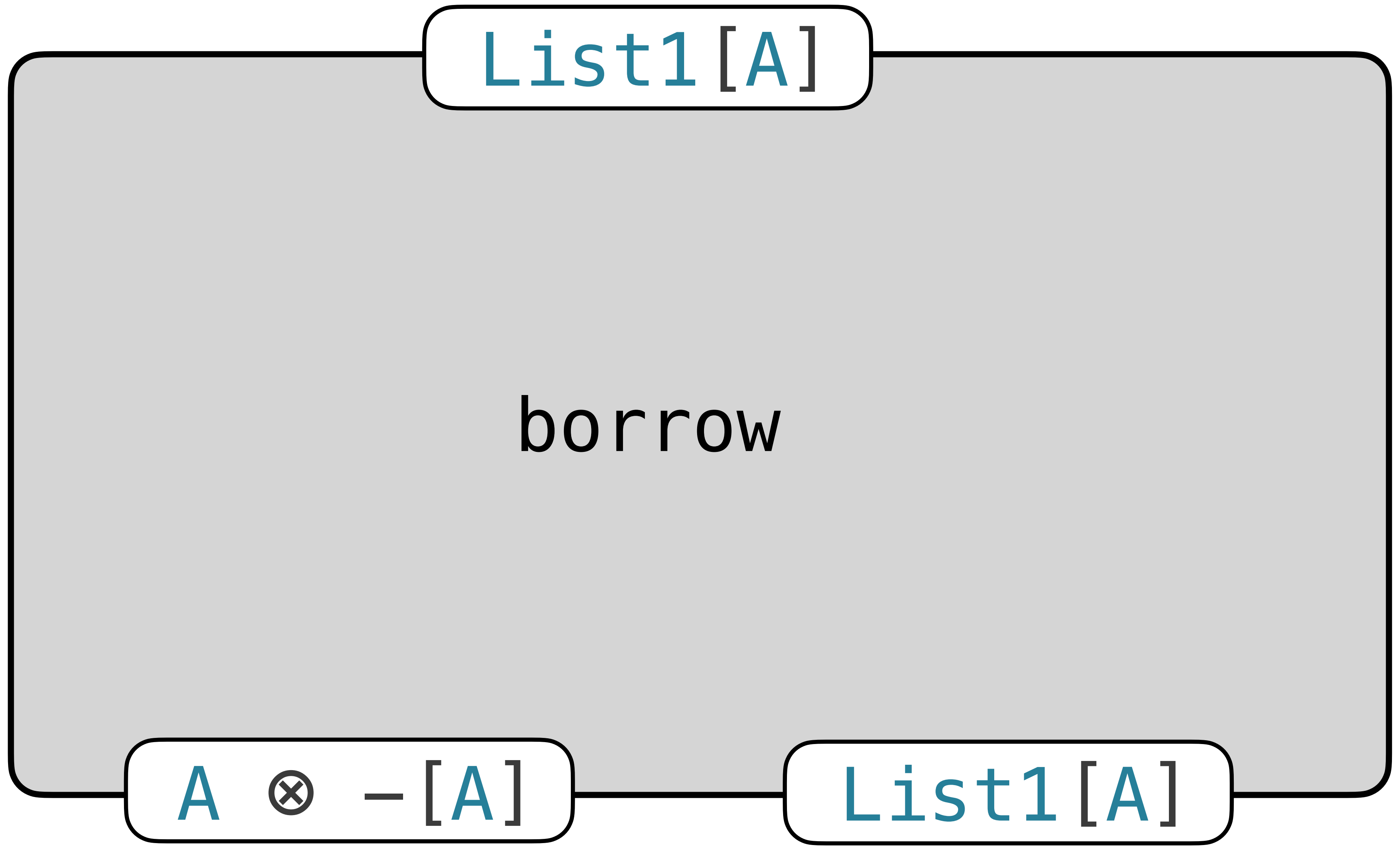
−[Endless[A]] ≃ List[−[A]]

# Non-empty List

```
List1[A] = A ⊗ List[A]
```

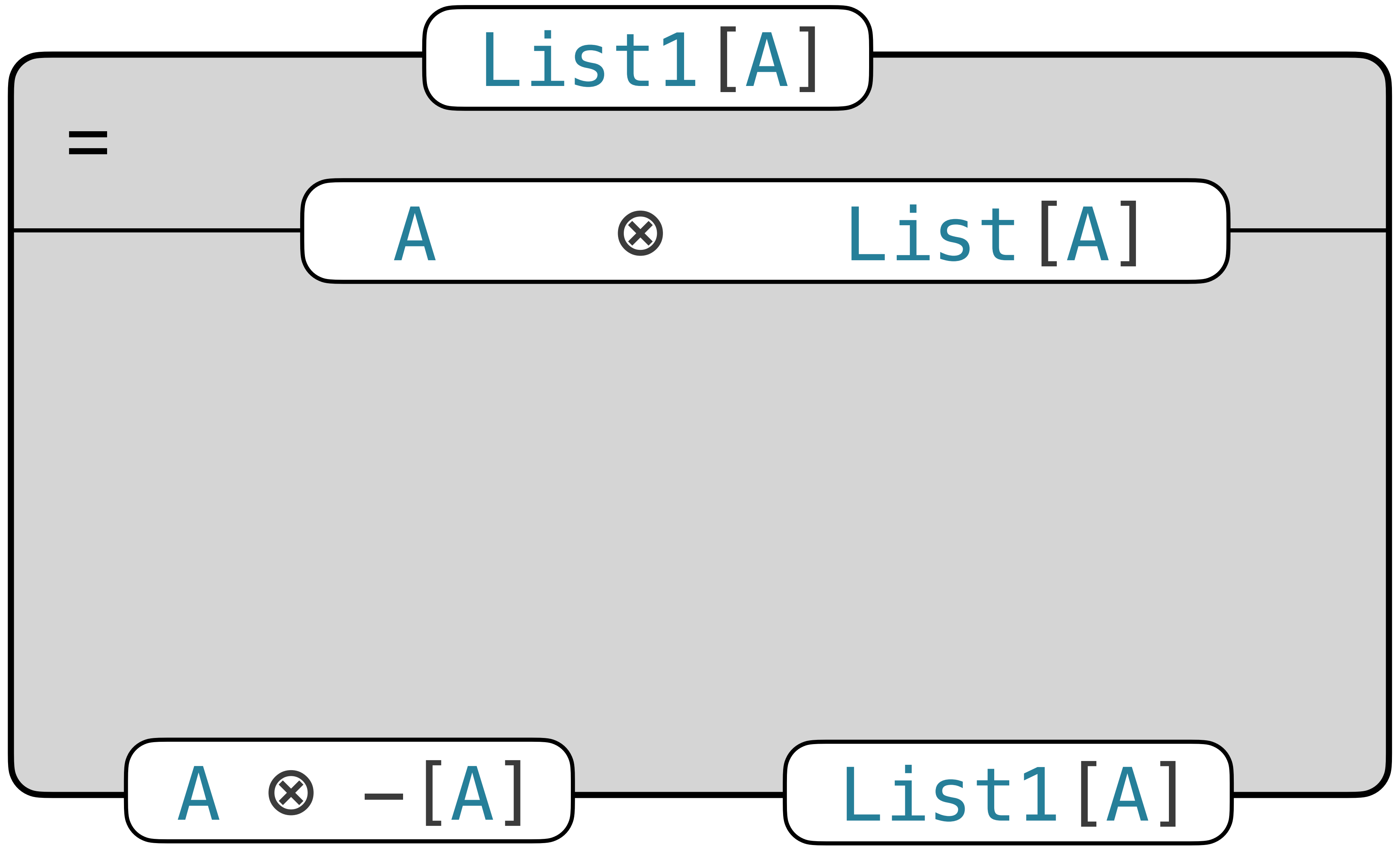
# Non-empty List

$$\texttt{List1[A]} = \texttt{A} \otimes \texttt{List[A]}$$



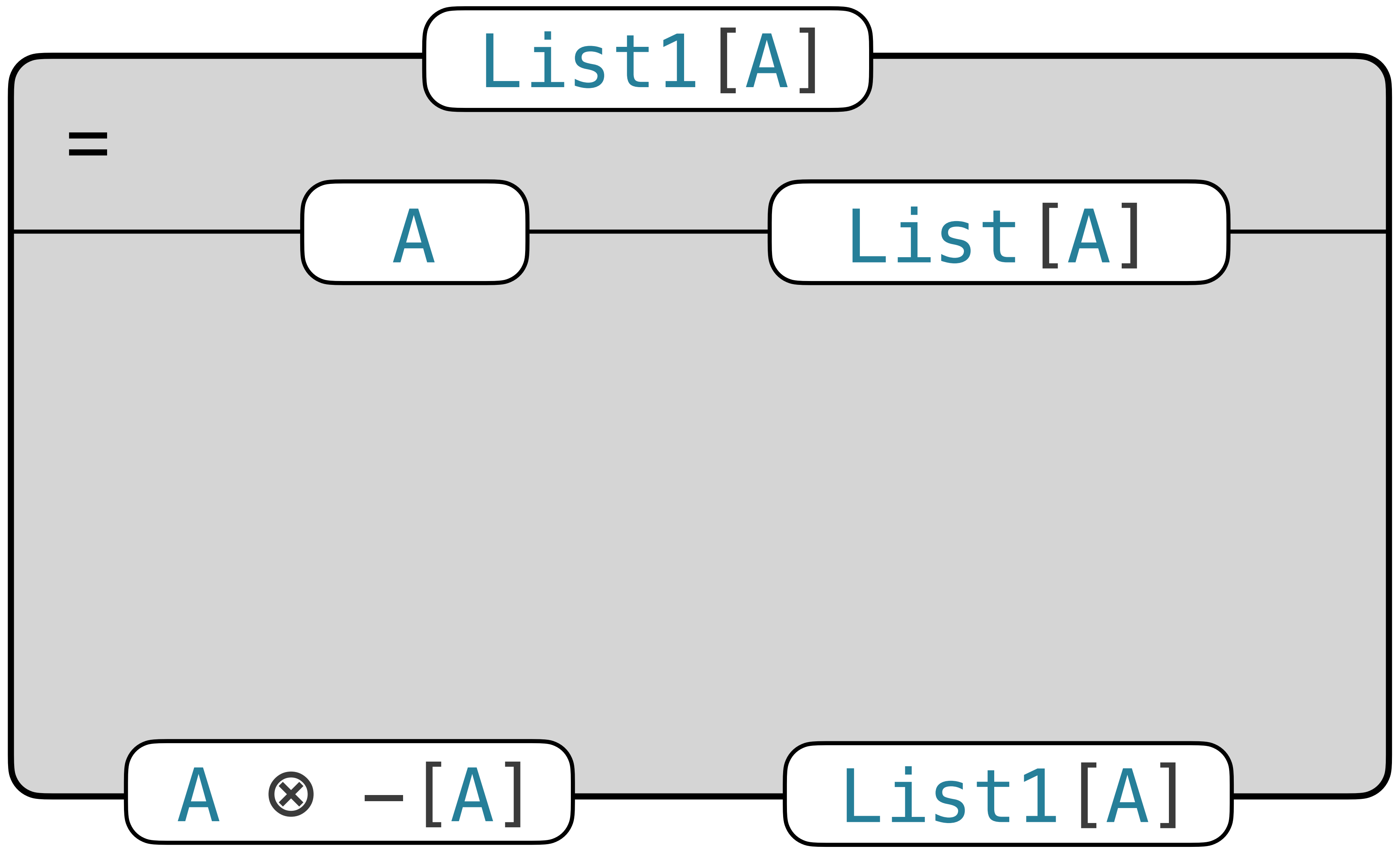
# Non-empty List

```
List1[A] = A ⊗ List[A]
```



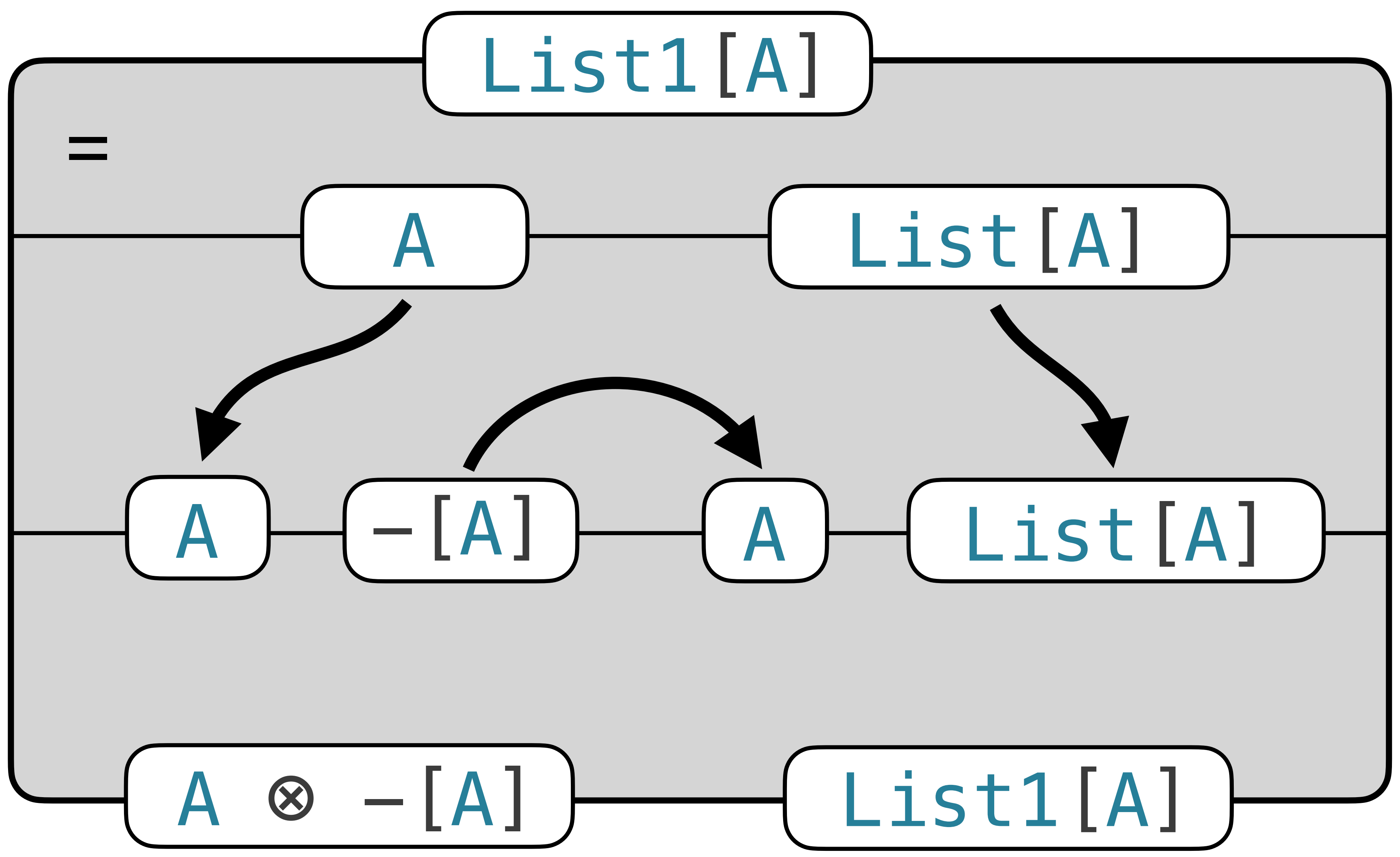
# Non-empty List

```
List1[A] = A ⊗ List[A]
```



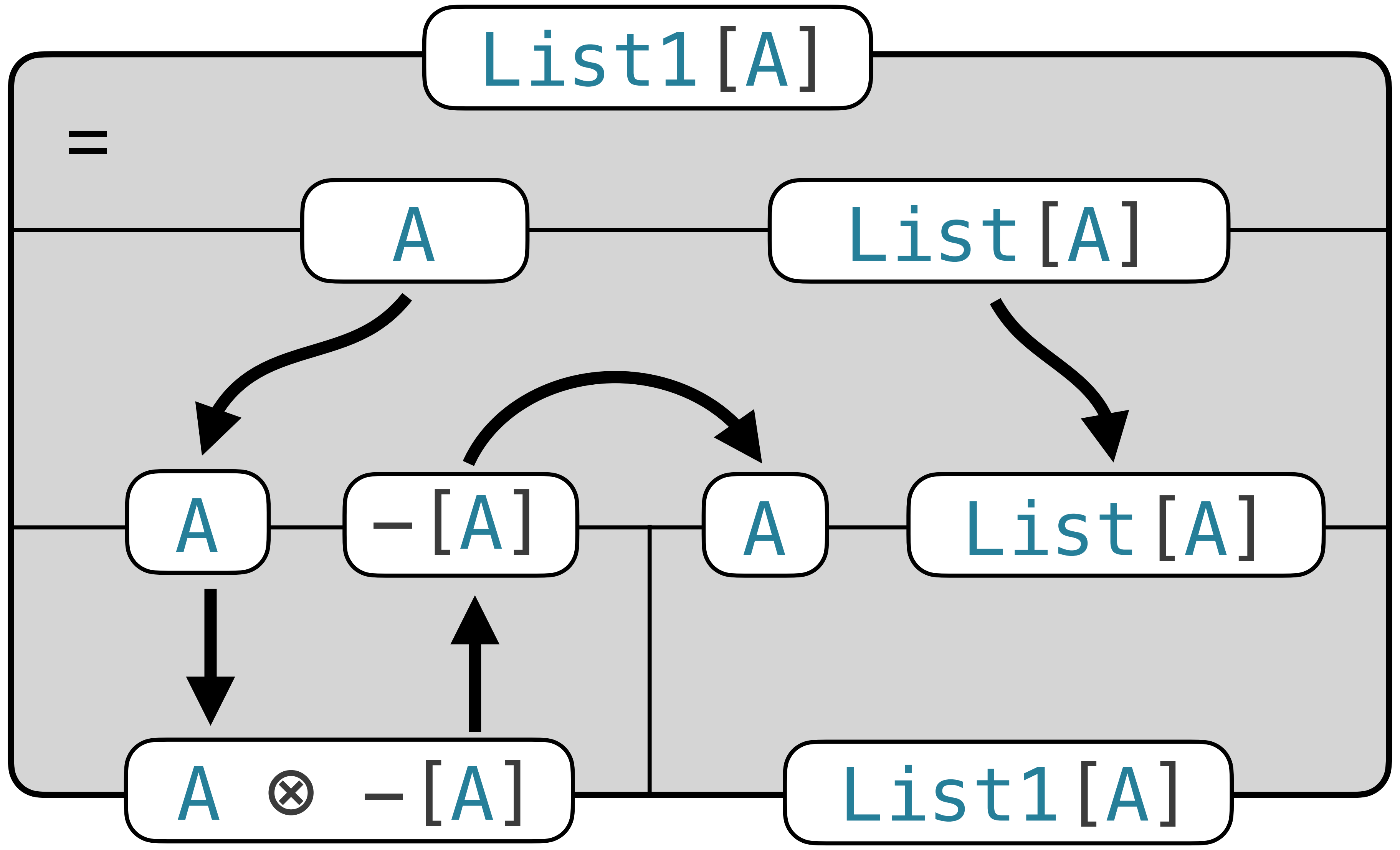
# Non-empty List

```
List1[A] = A ⊗ List[A]
```



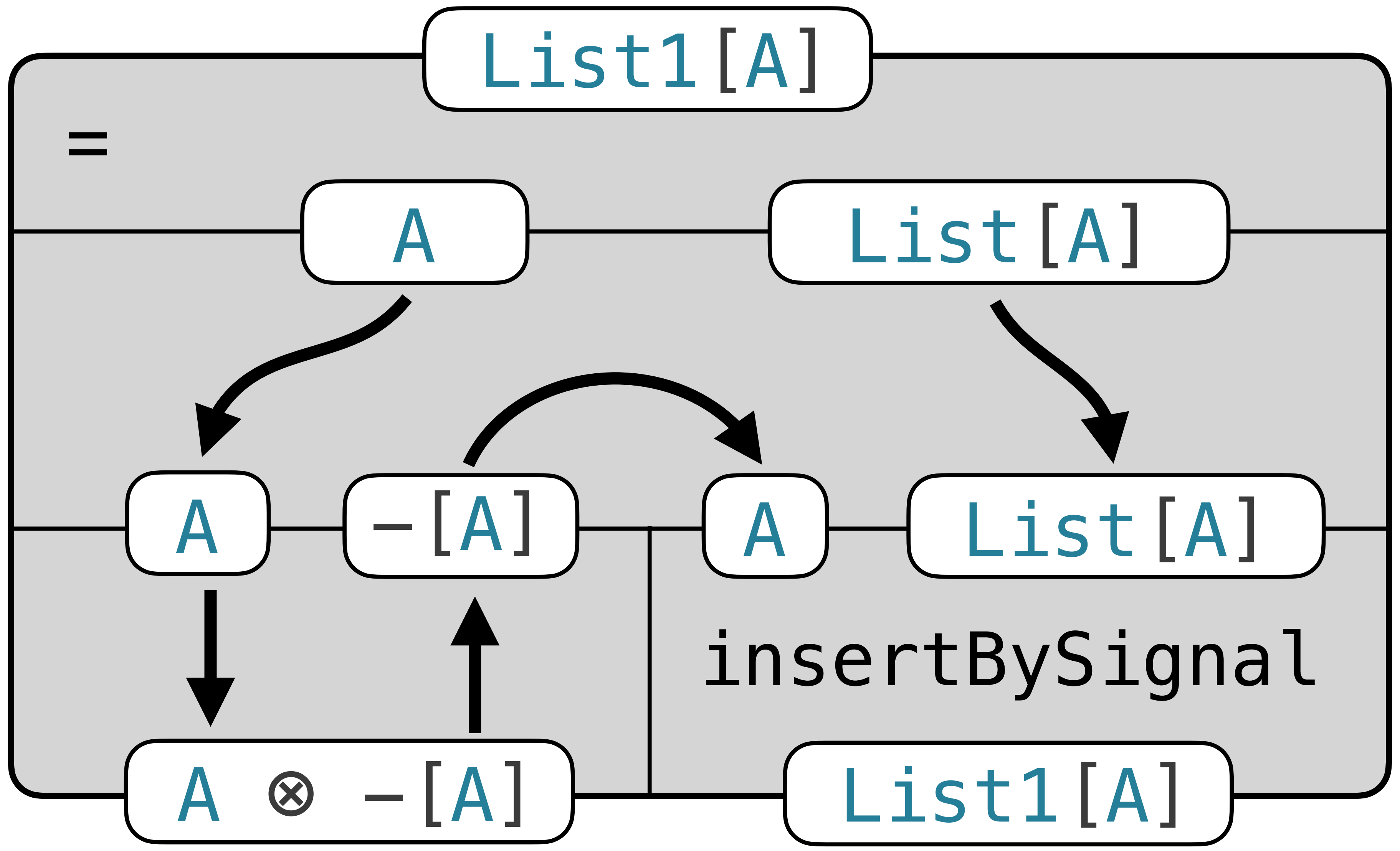
# Non-empty List

$$\texttt{List1[A]} = \texttt{A} \otimes \texttt{List[A]}$$



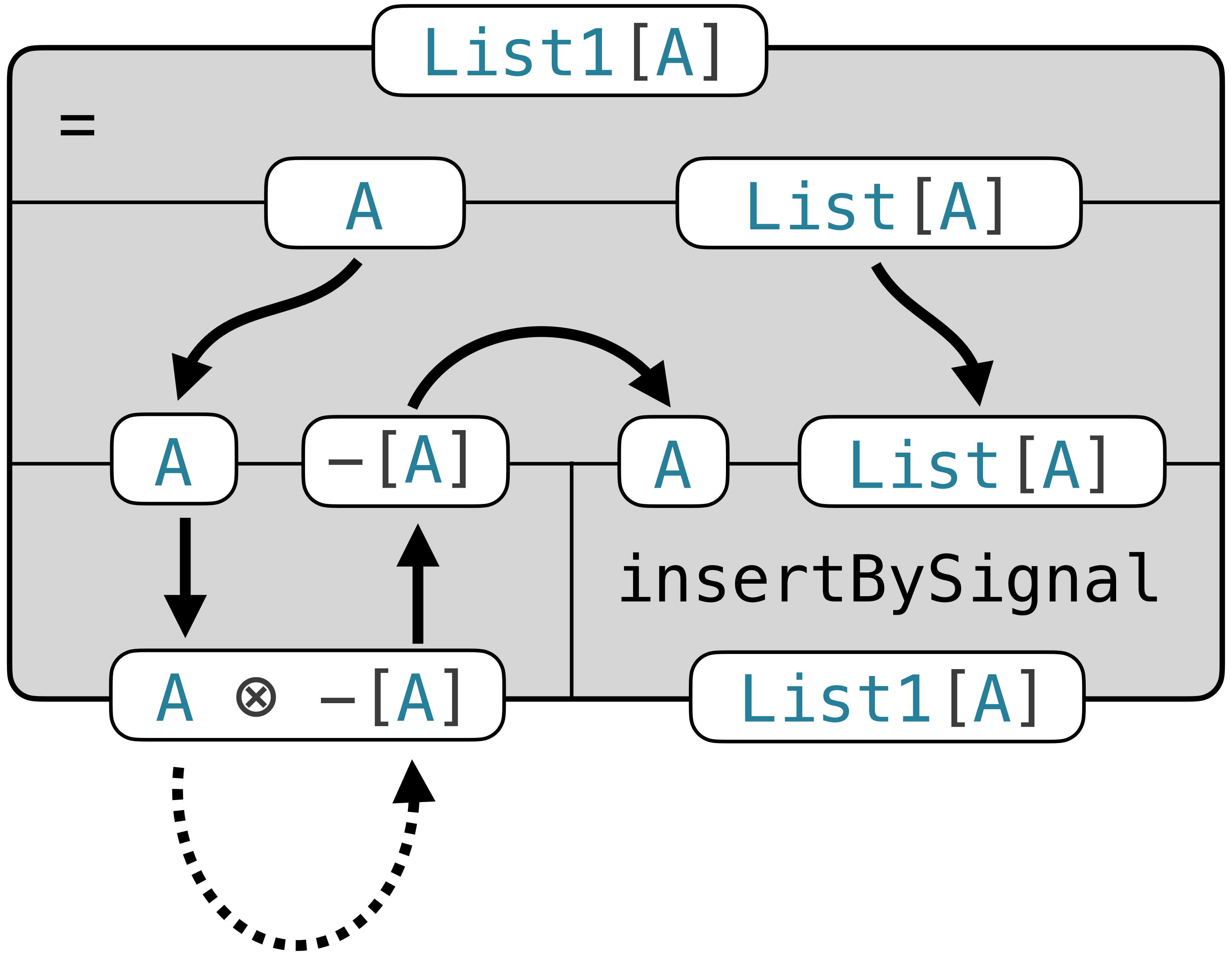
# Non-empty List

List1[A] = A ⊗ List[A]



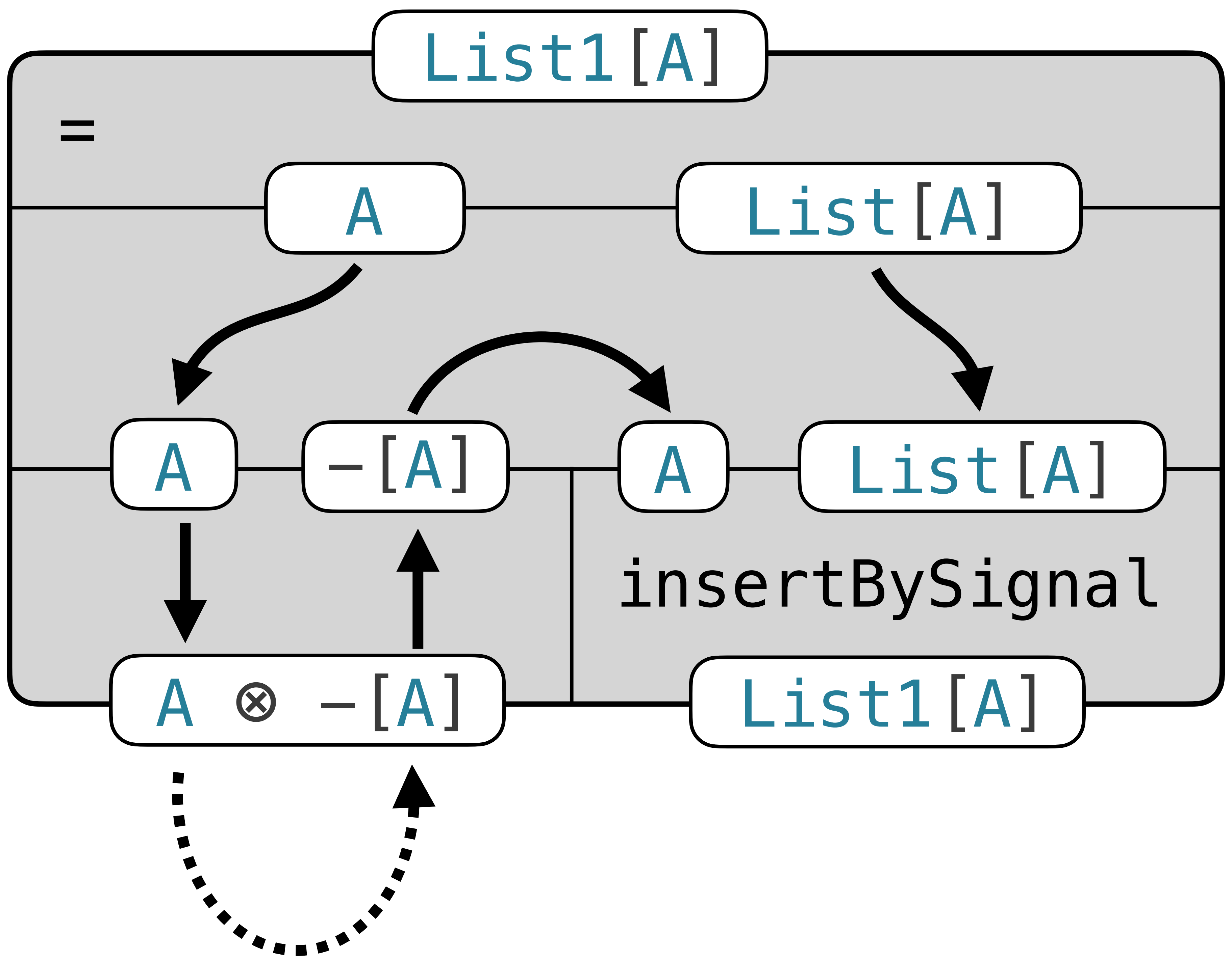
# Non-empty List

$$\texttt{List1[A]} = \texttt{A} \otimes \texttt{List[A]}$$



# Non-empty List

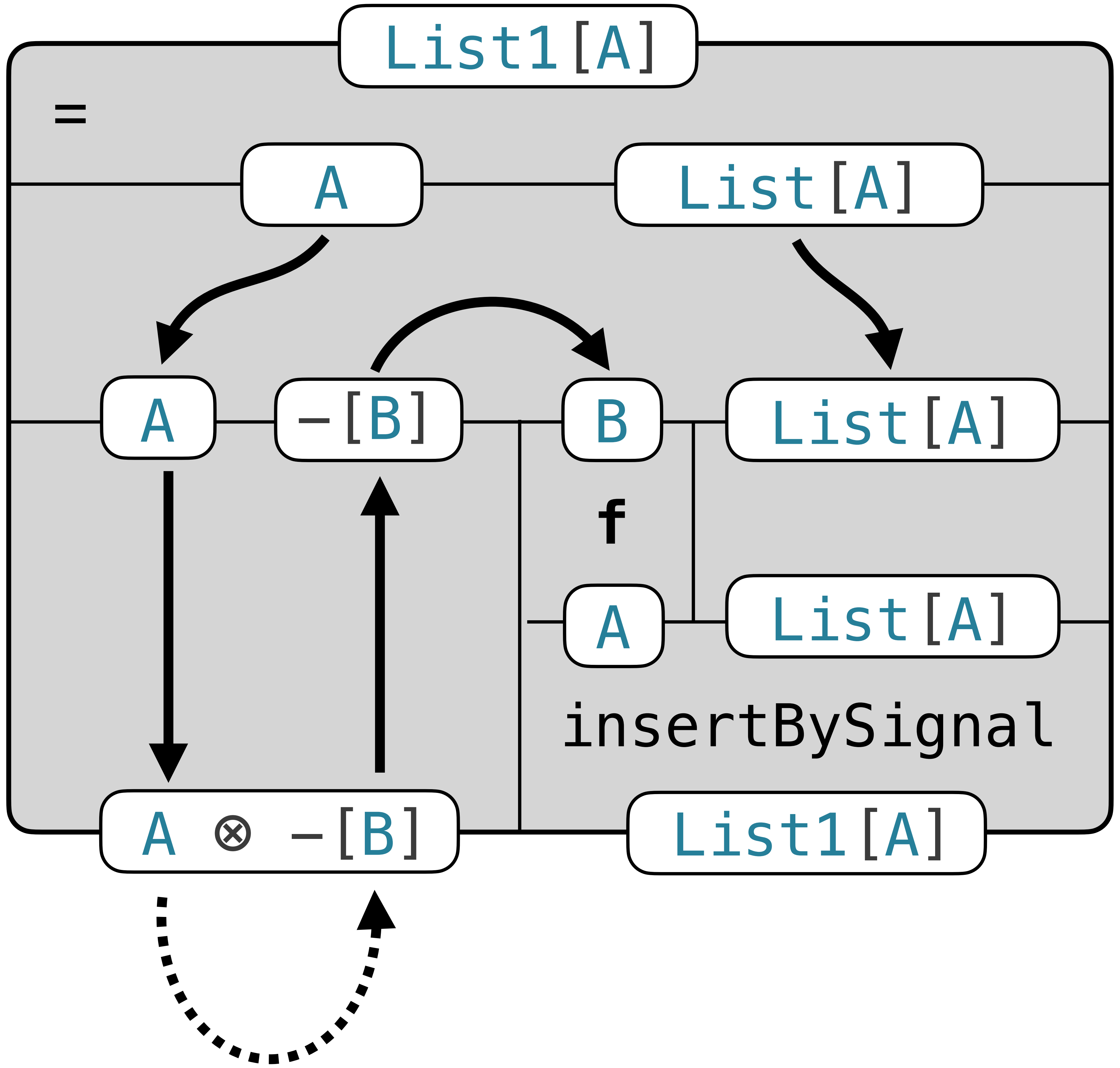
```
List1[A] = A ⊗ List[A]
```




```
def borrow[A](using
  Signaling.Positive[A],
): List1[A] -∘ (A ⊗ -[A] ⊗ List1[A]) =
  λ { case a ⊗ as =>
    val (na ⊗ a1) = constant(forevert)
    (a ⊗ na) ⊗ insertBySignal(a1 ⊗ as)
  }
```

# List1.borrowReset

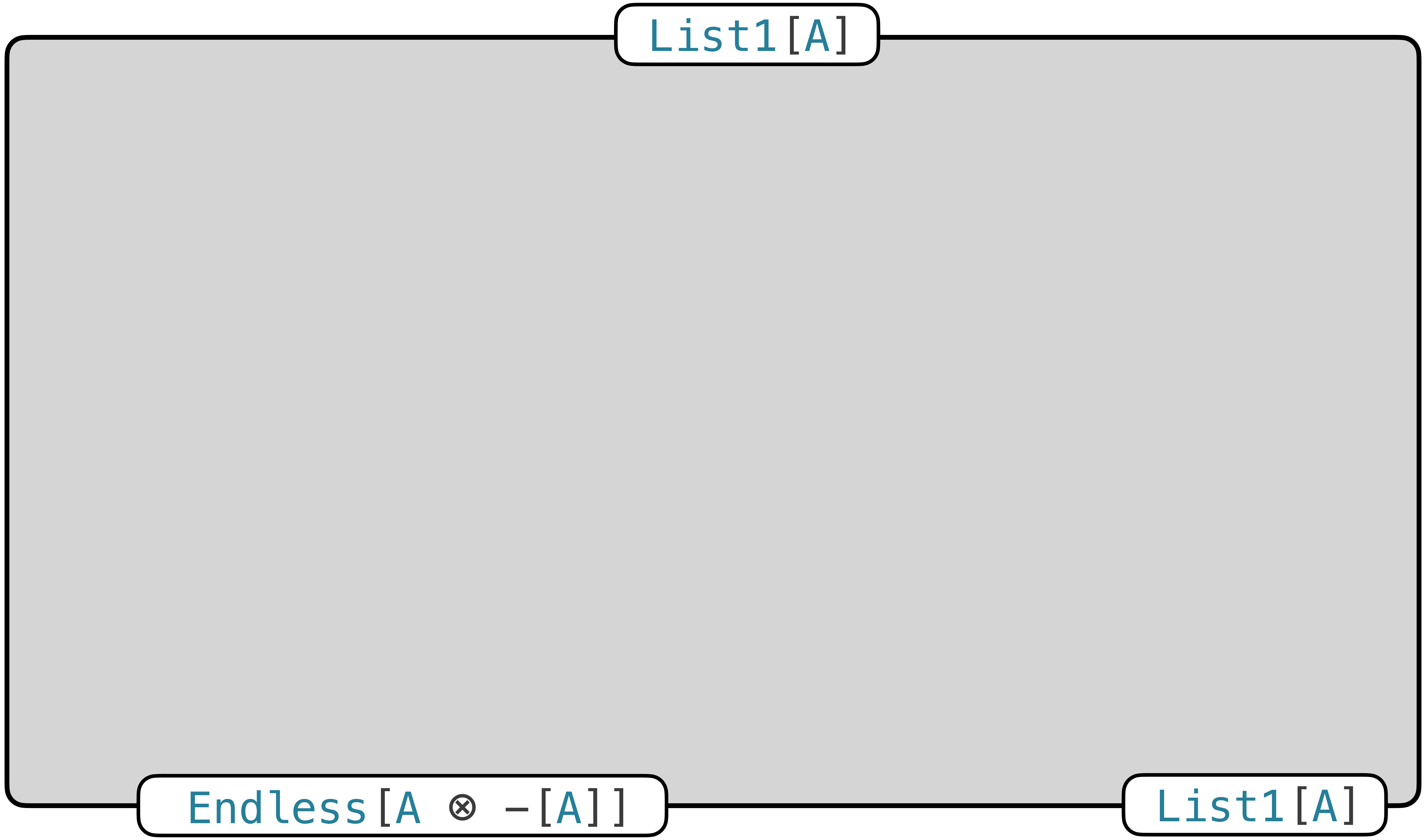
Different type B of returned element. Reset back to A by a given function.




```
def borrowReset[A](f: B -∘ A)(using
  Signaling.Positive[A],
): List1[A] -∘ (A ⊗ -[B] ⊗ List1[A]) =
  λ { case a ⊗ as =>
    val (nb ⊗ b) = constant(forevert)
    (a ⊗ nb) ⊗ insertBySignal(f(b) ⊗ as)
  }
```

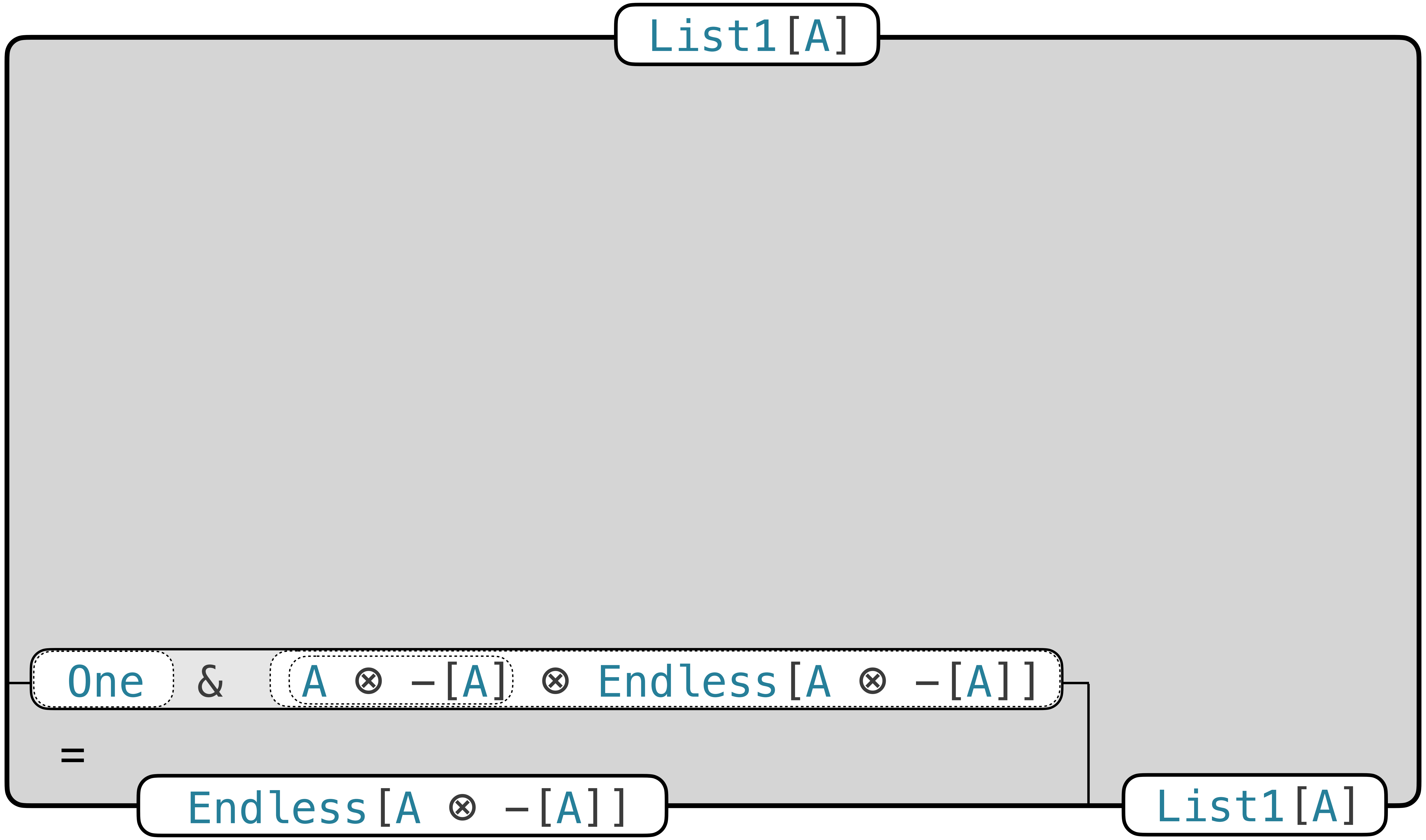
# Endless.pool

Present a *limited* supply of elements as an *endless* supply of *borrowed* elements.



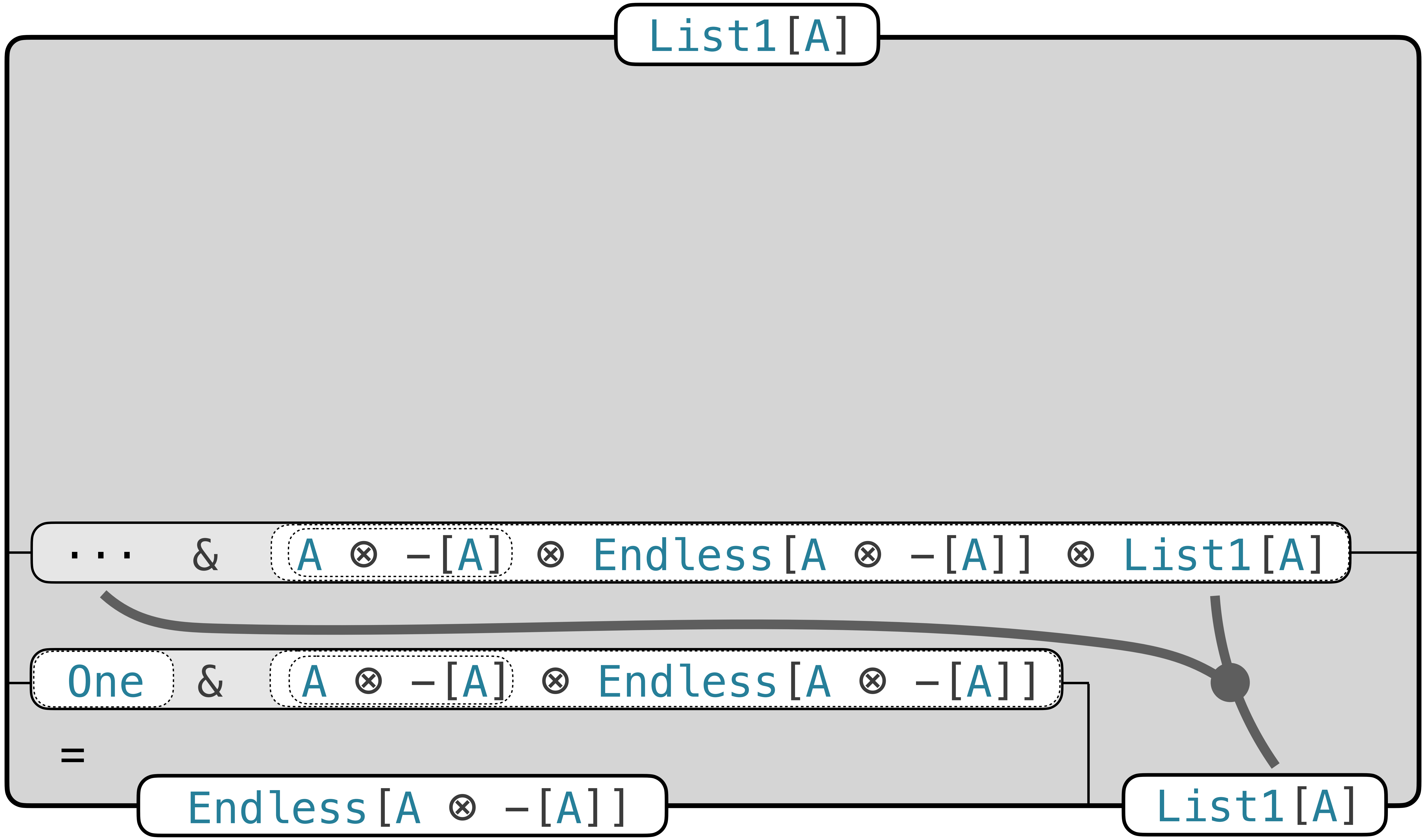
# Endless.pool

Present a *limited* supply of elements as an *endless* supply of *borrowed* elements.



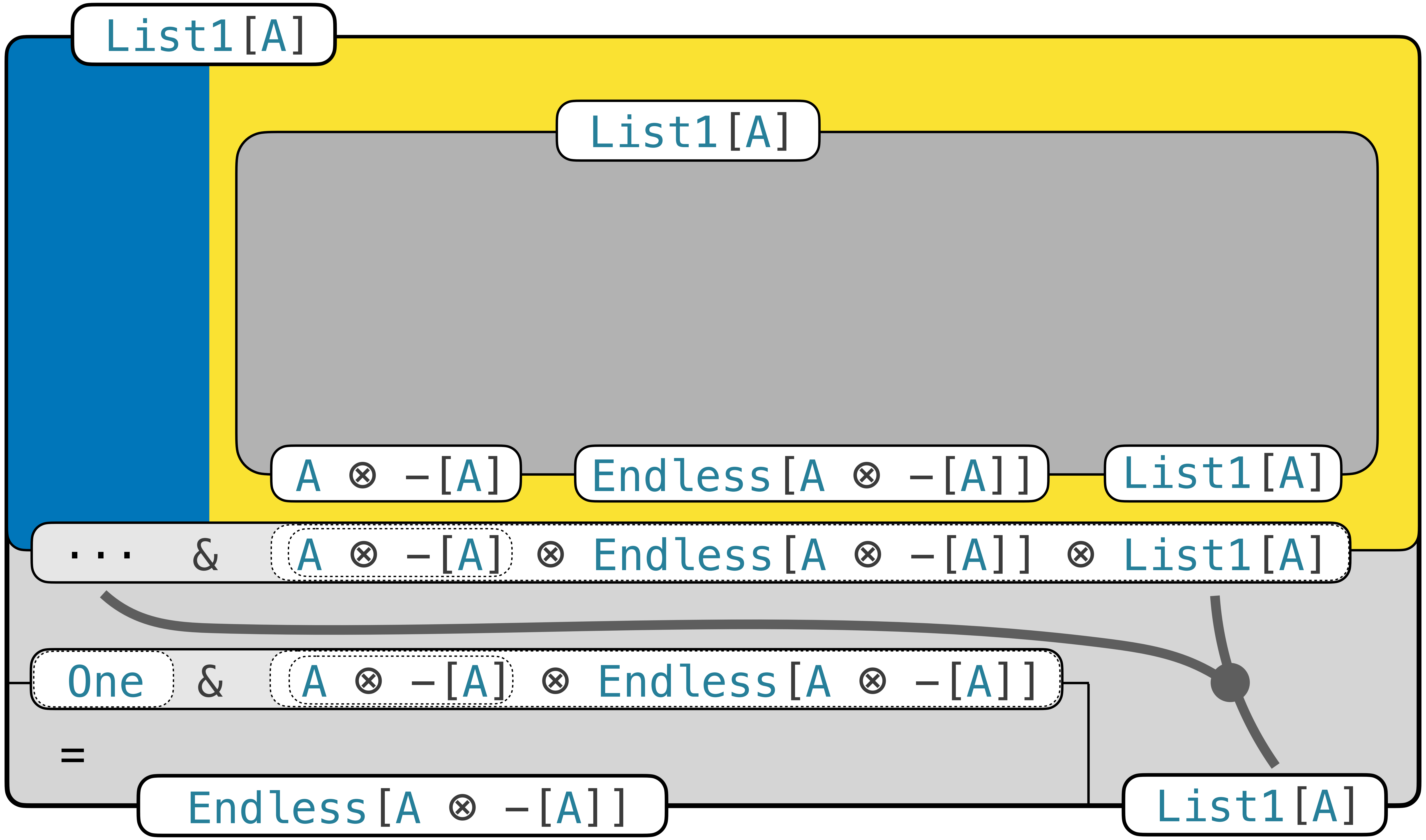
# Endless.pool

Present a *limited* supply of elements as an *endless* supply of *borrowed* elements.



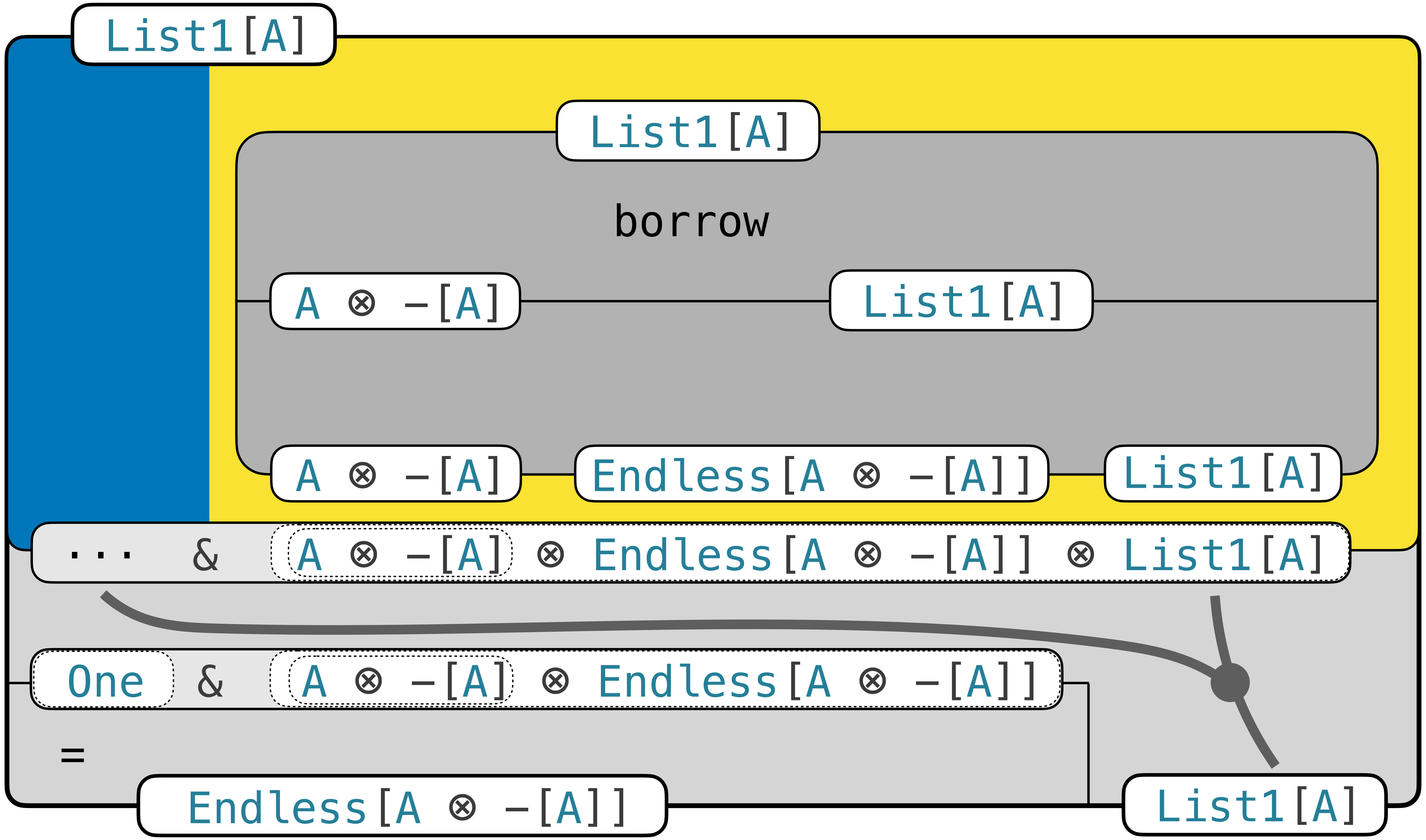
# Endless.pool

Present a *limited* supply of elements as an *endless* supply of *borrowed* elements.



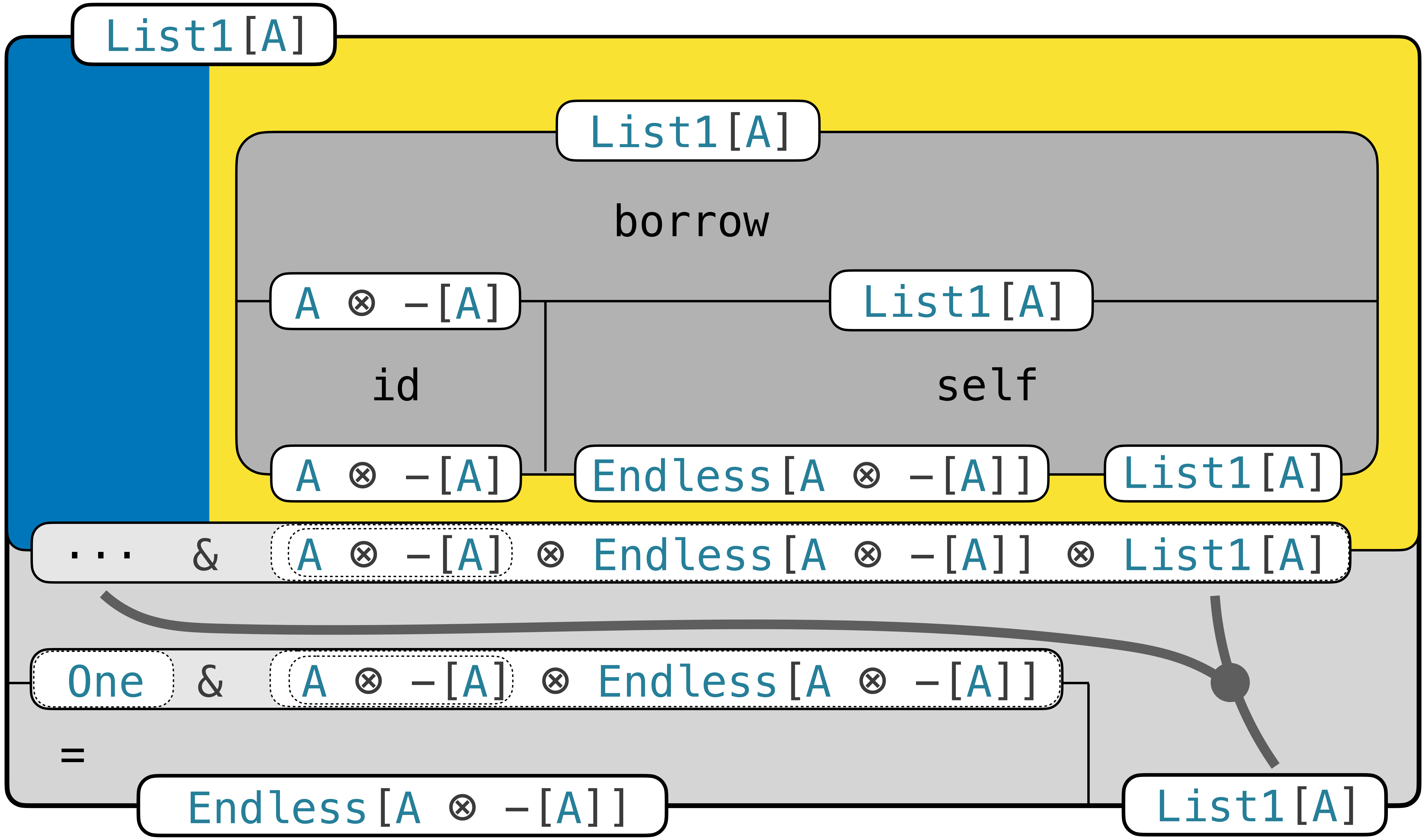
# Endless.pool

Present a *limited* supply of elements as an *endless* supply of *borrowed* elements.



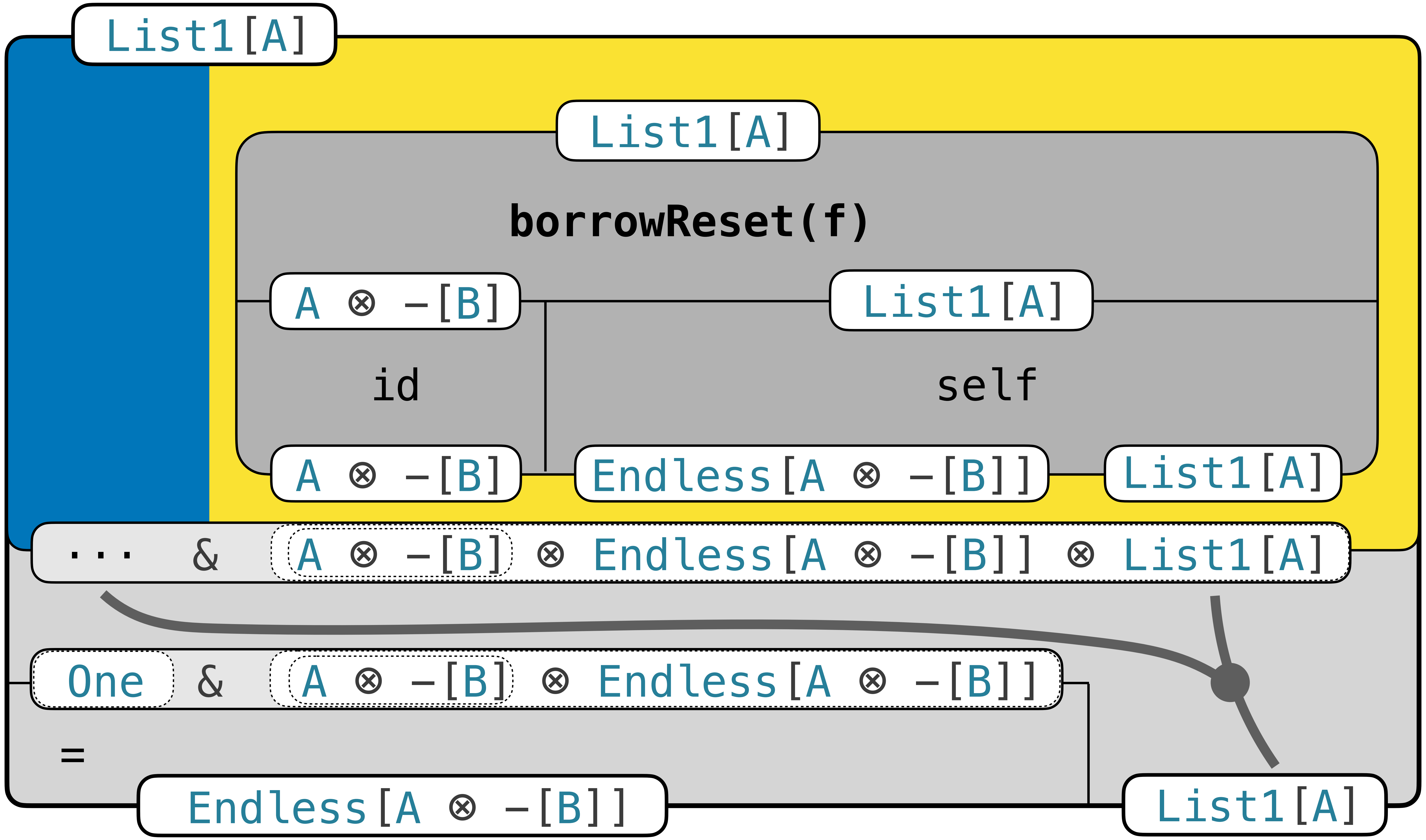
# Endless.pool

Present a *limited* supply of elements as an *endless* supply of *borrowed* elements.



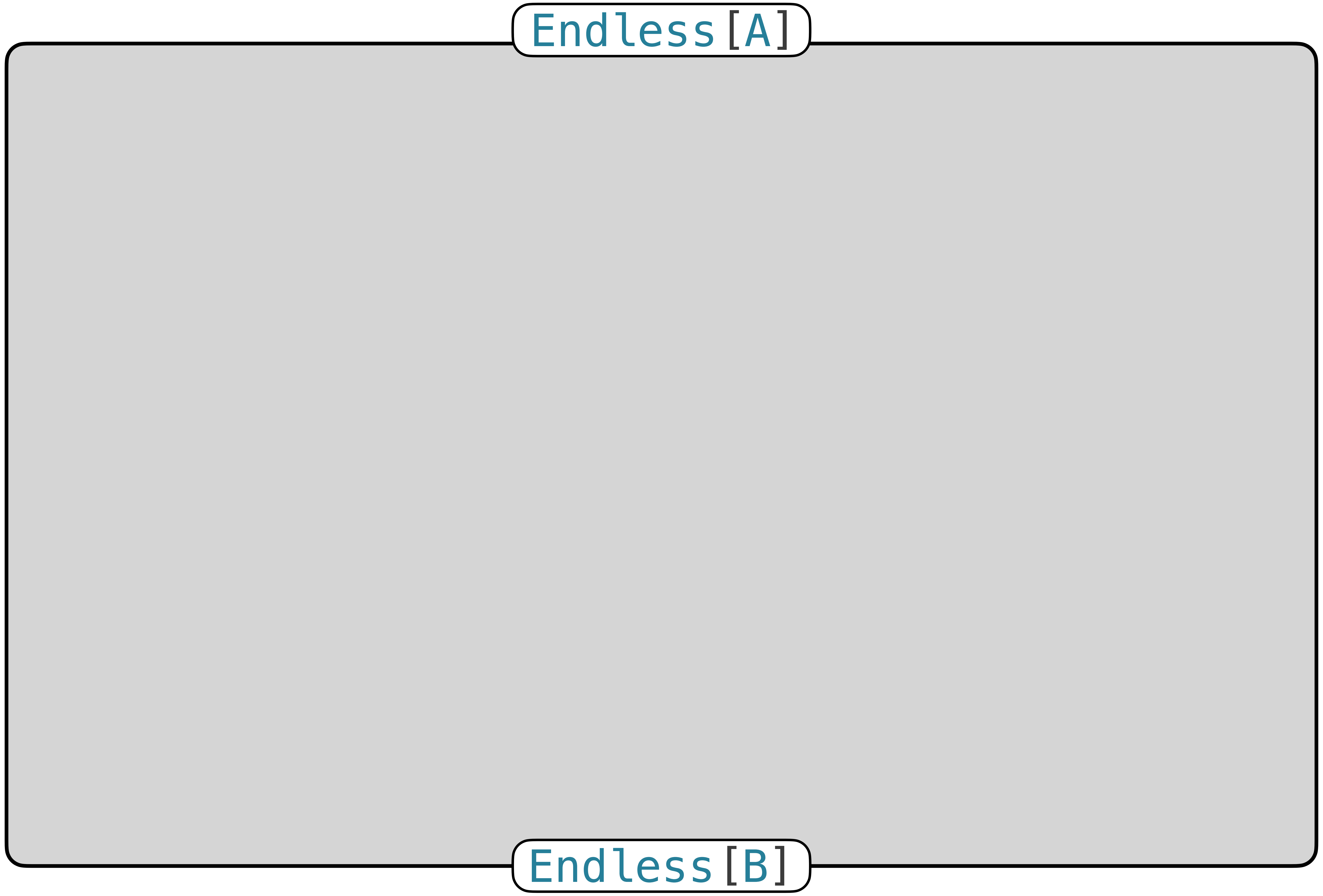
# Endless.poolReset(f)

Present a *limited* supply of elements as an *endless* supply of *borrowed* elements.



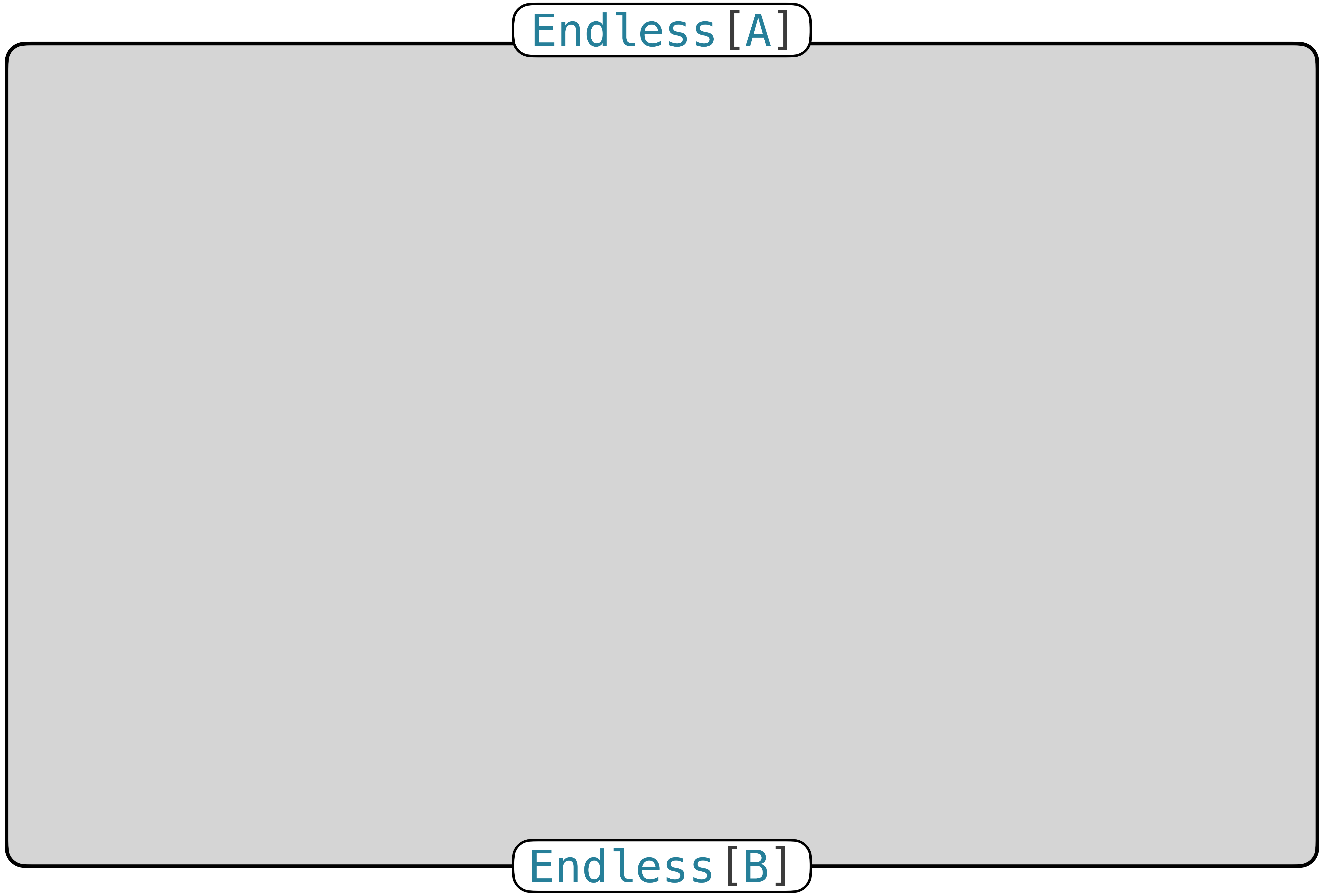
# Endless.mapSequentially(f)

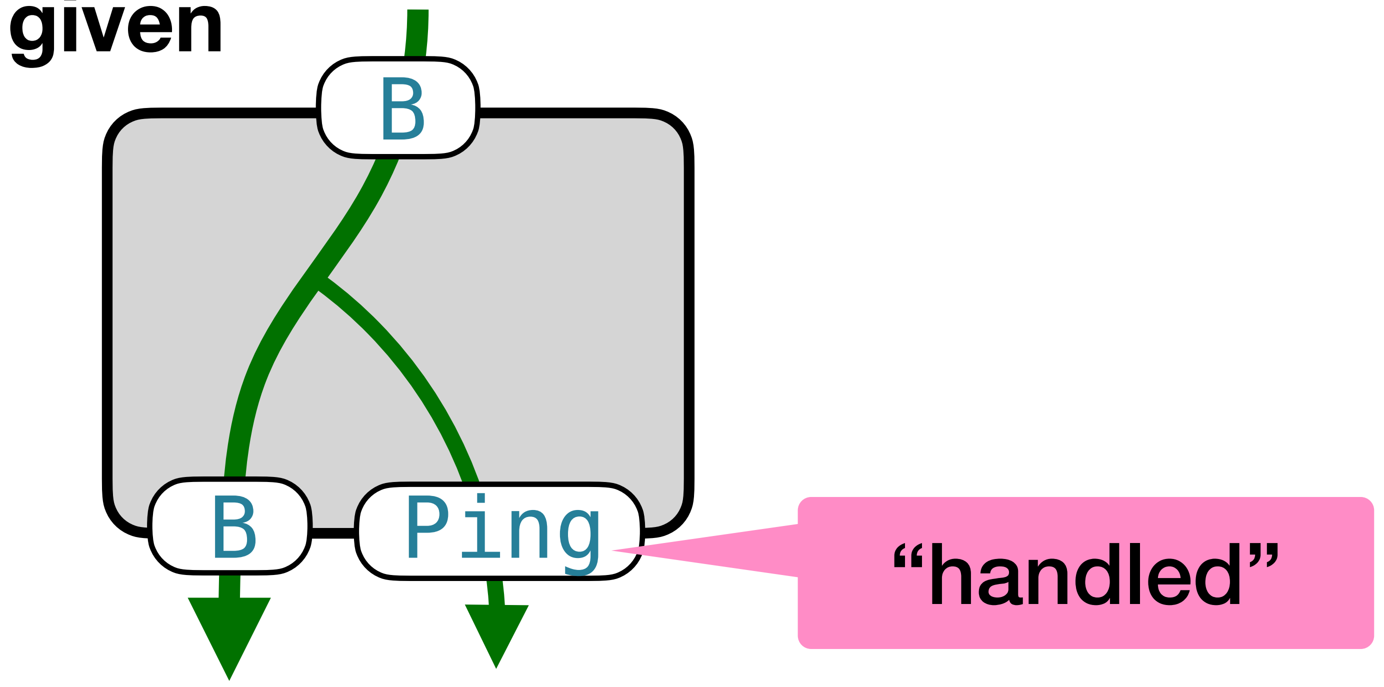
- Delay pulling from upstream until previous element has been *"handled"*



# Endless.mapSequentially(f)

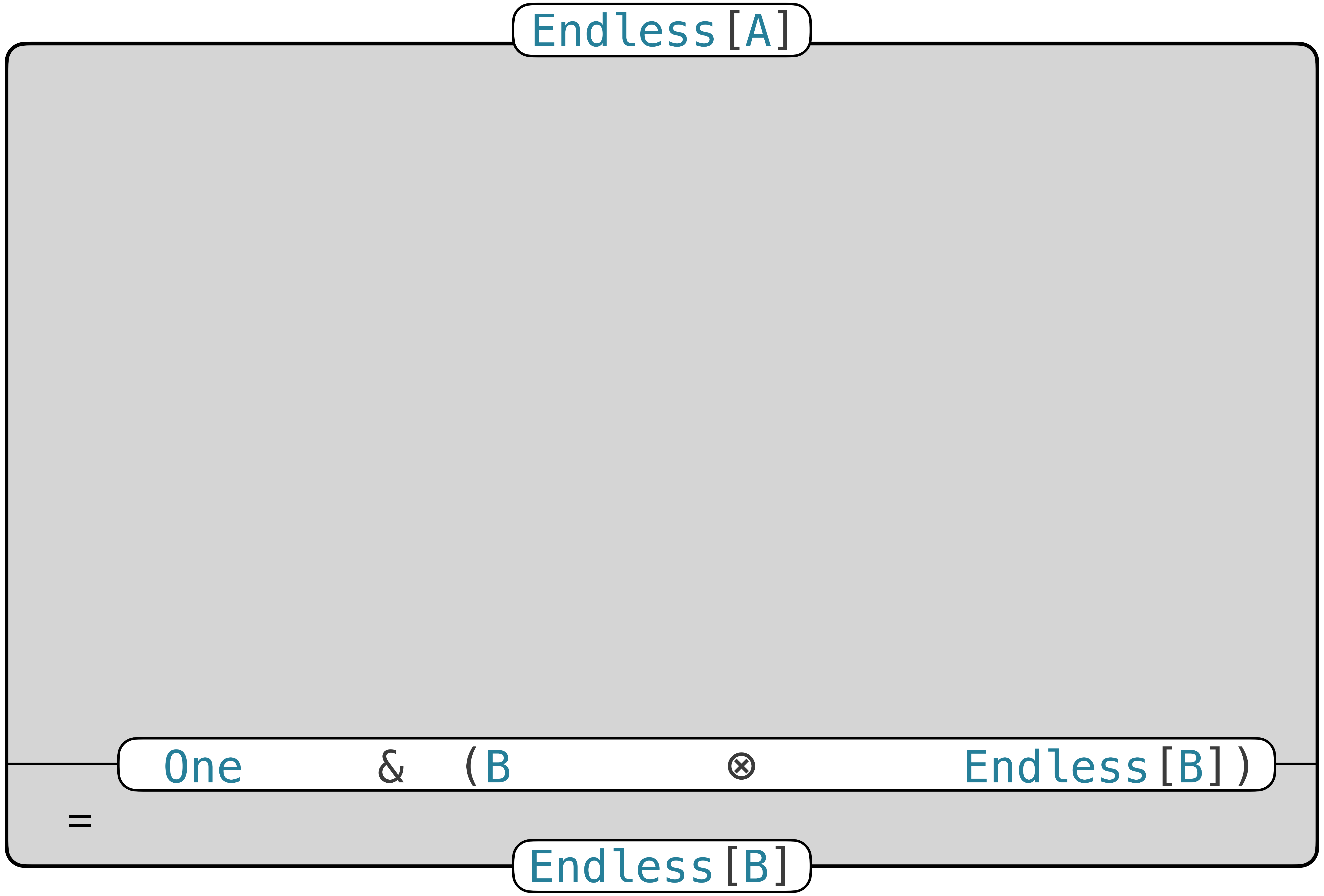
- Delay pulling from upstream until previous element has been *"handled"*

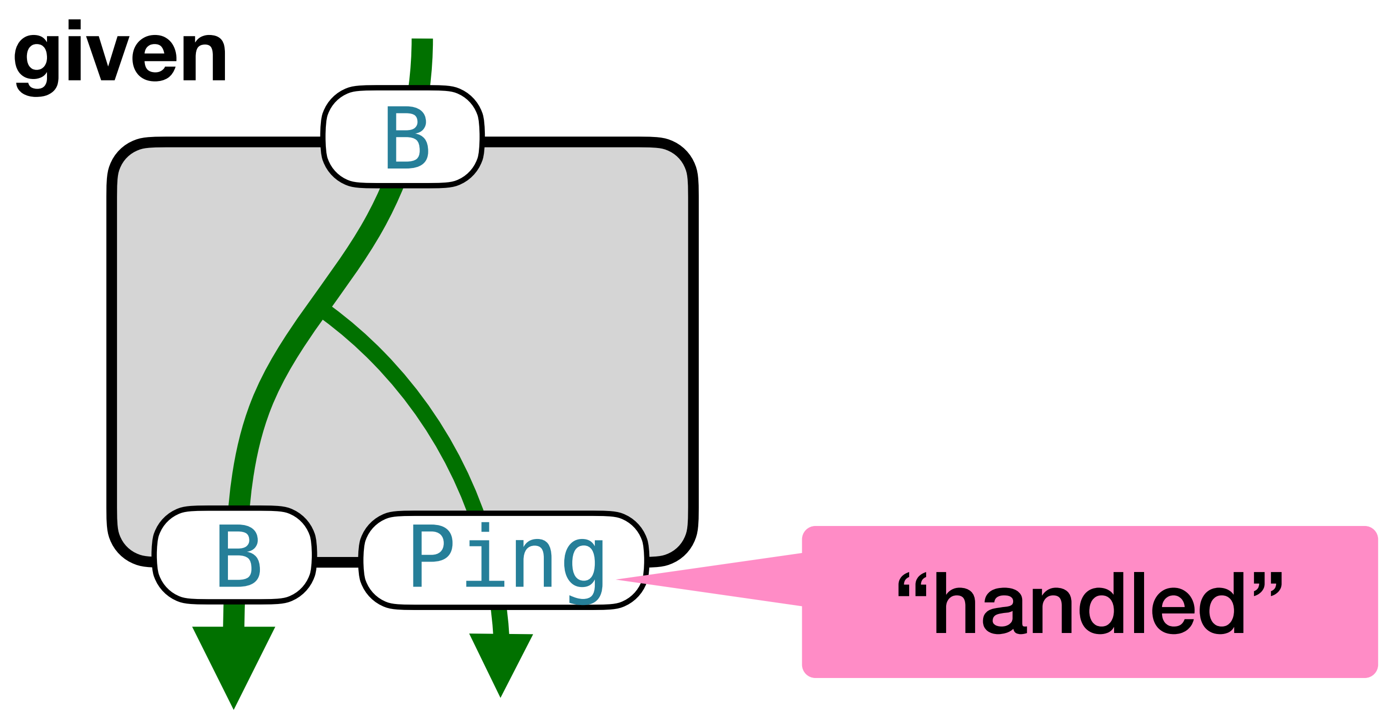


# Endless.mapSequentially(f)

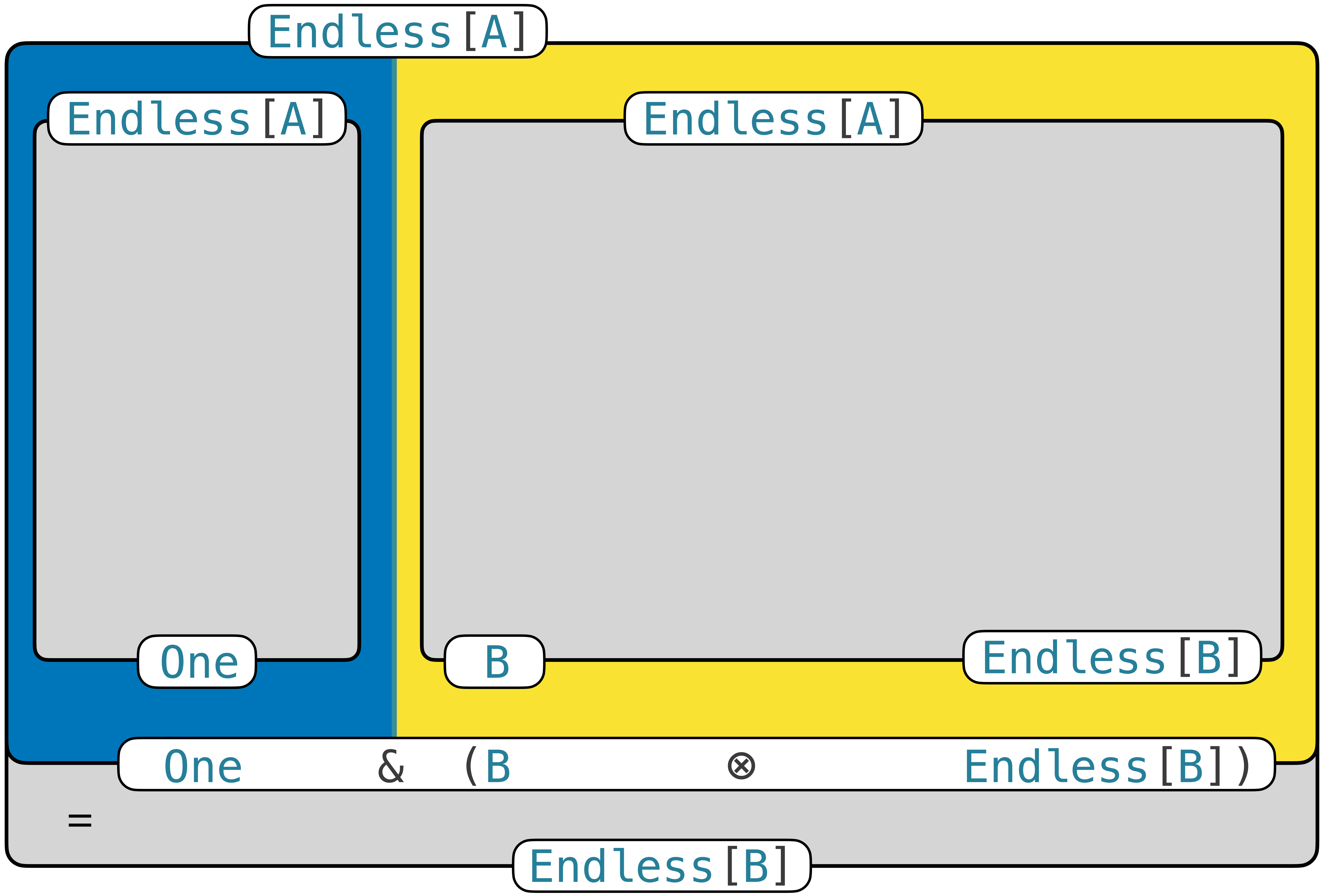
- Delay pulling from upstream until previous element has been *"handled"*

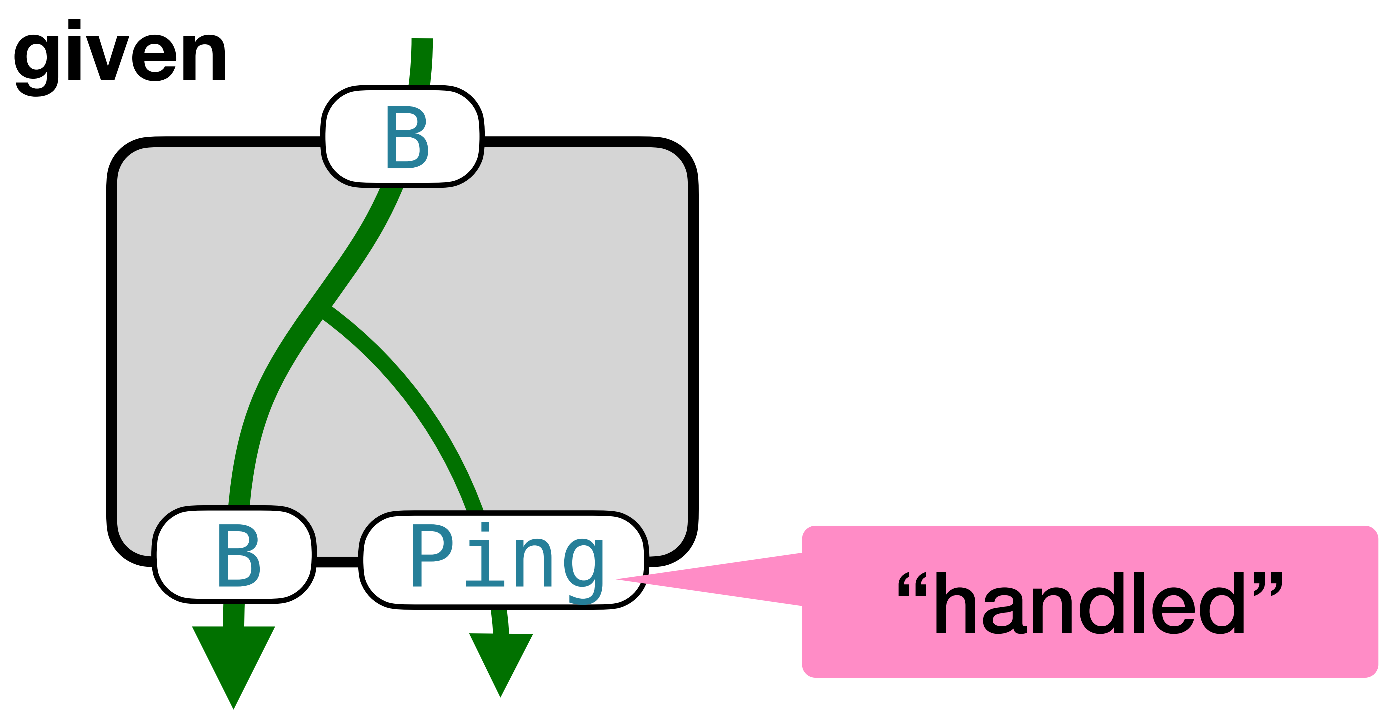


# Endless.mapSequentially(f)

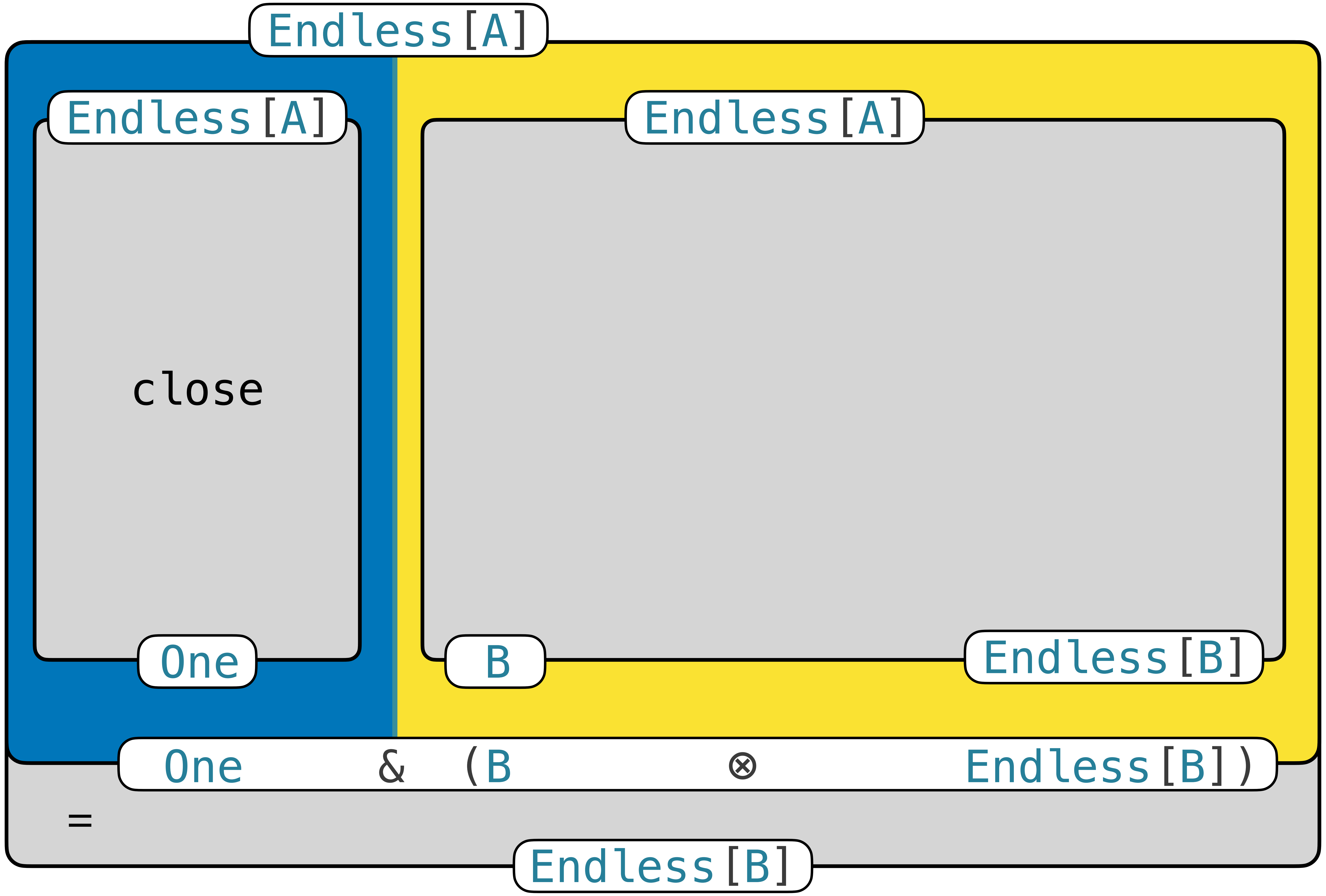
- Delay pulling from upstream until previous element has been *“handled”*

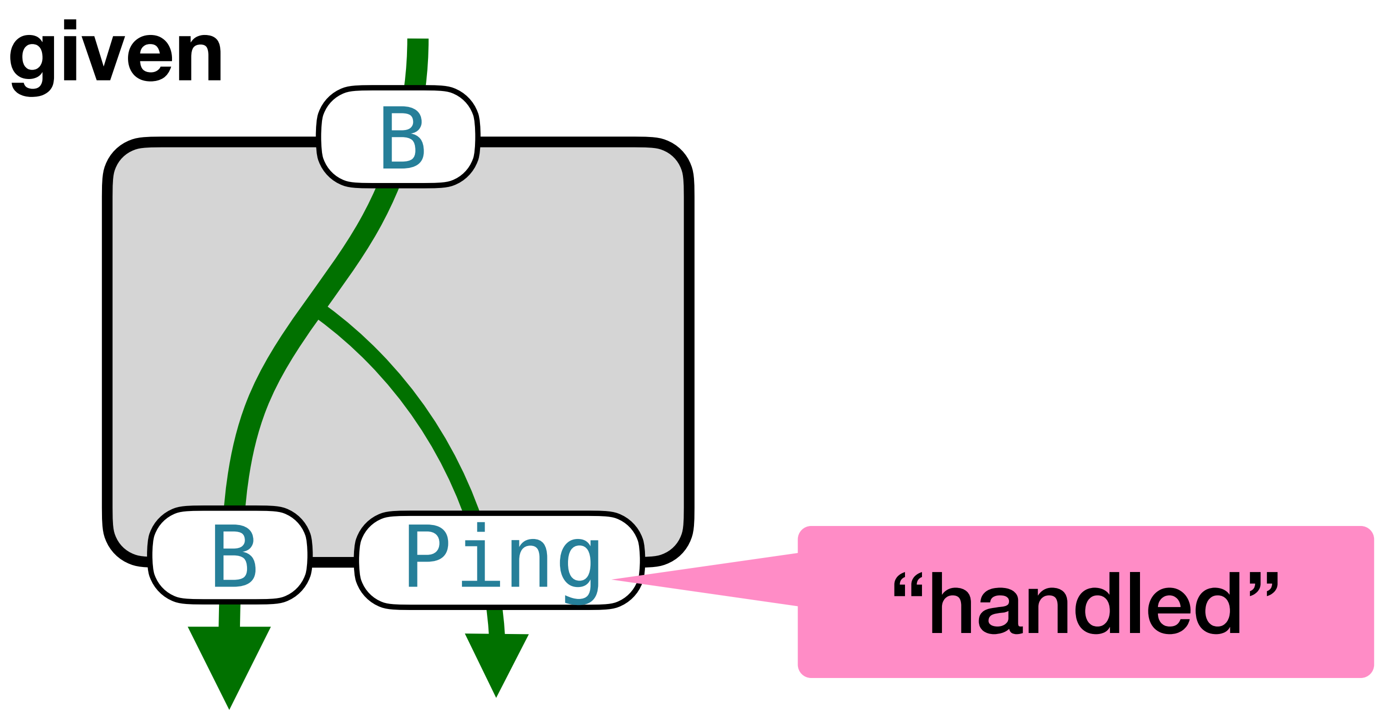


# Endless.mapSequentially(f)

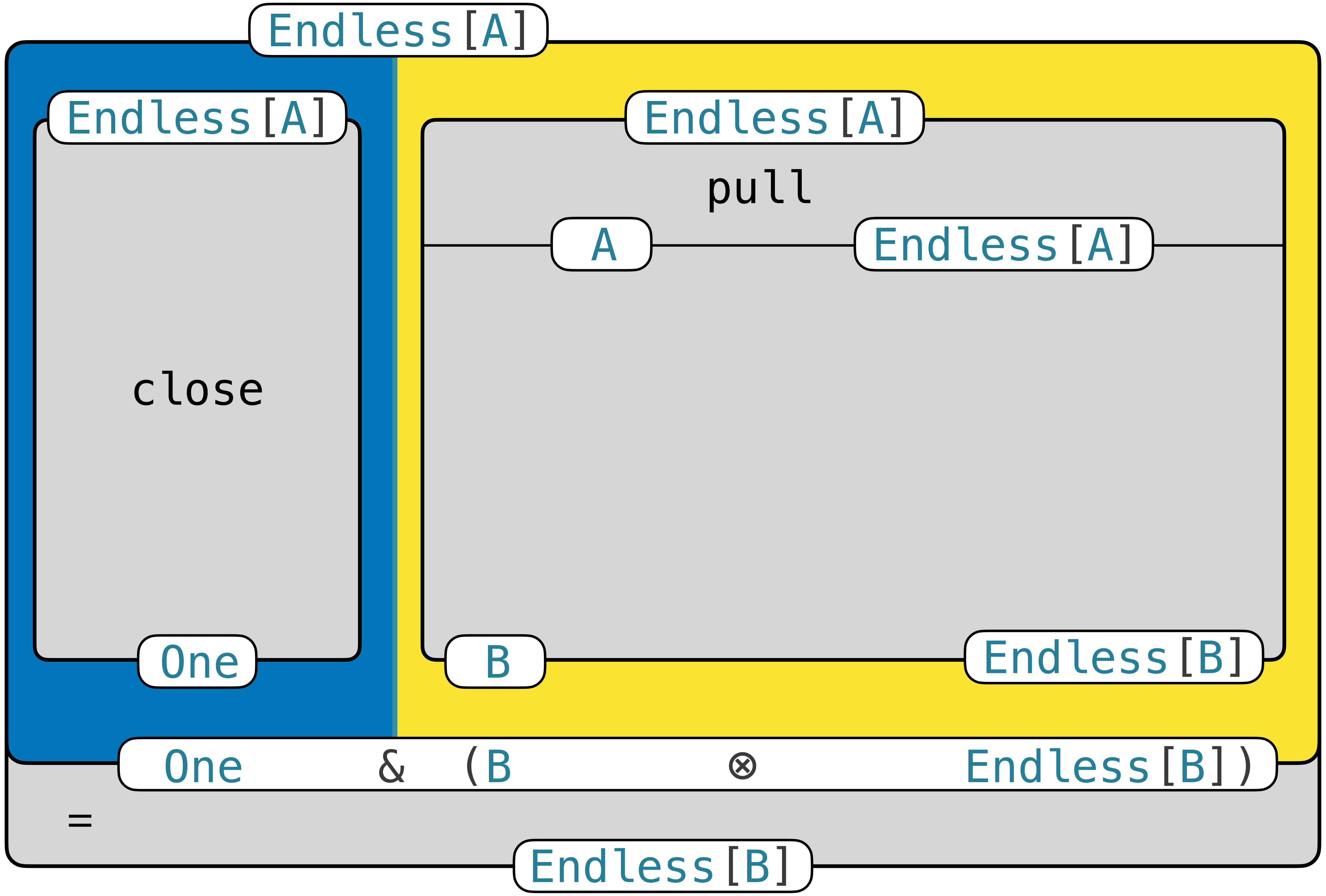
- Delay pulling from upstream until previous element has been *"handled"*

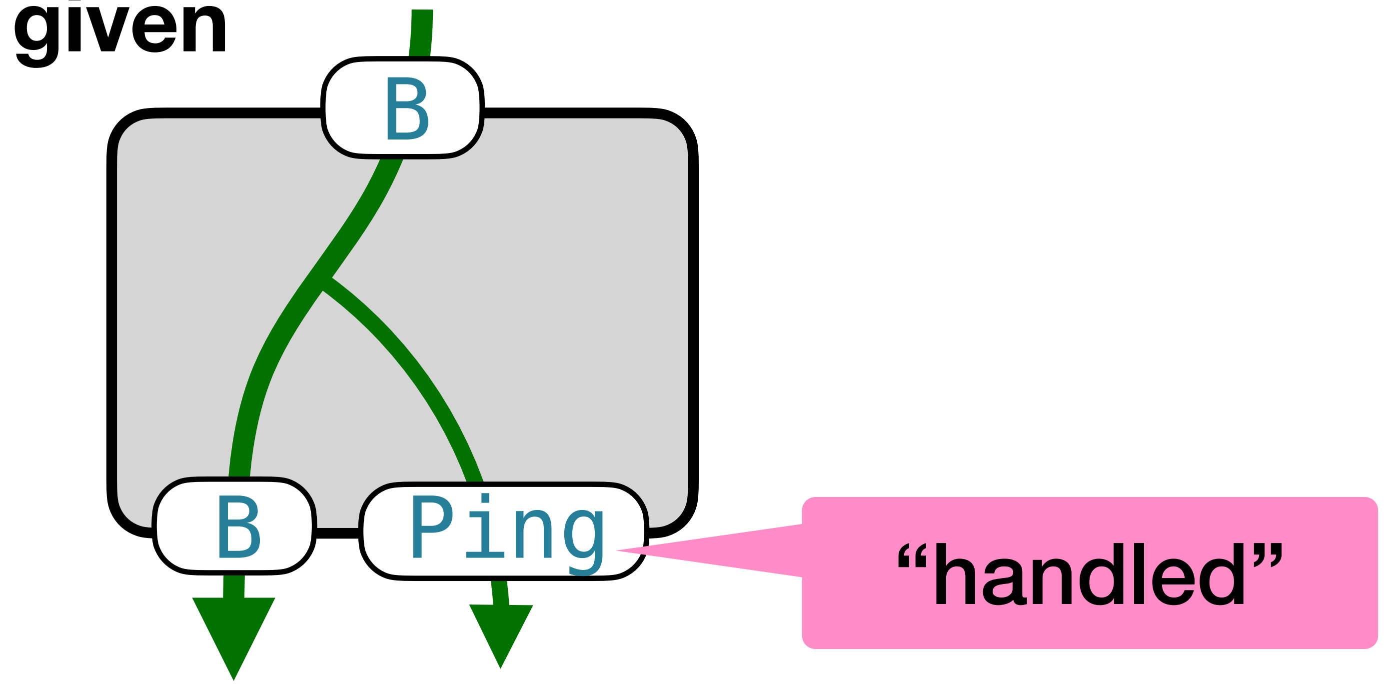


# Endless.mapSequentially(f)

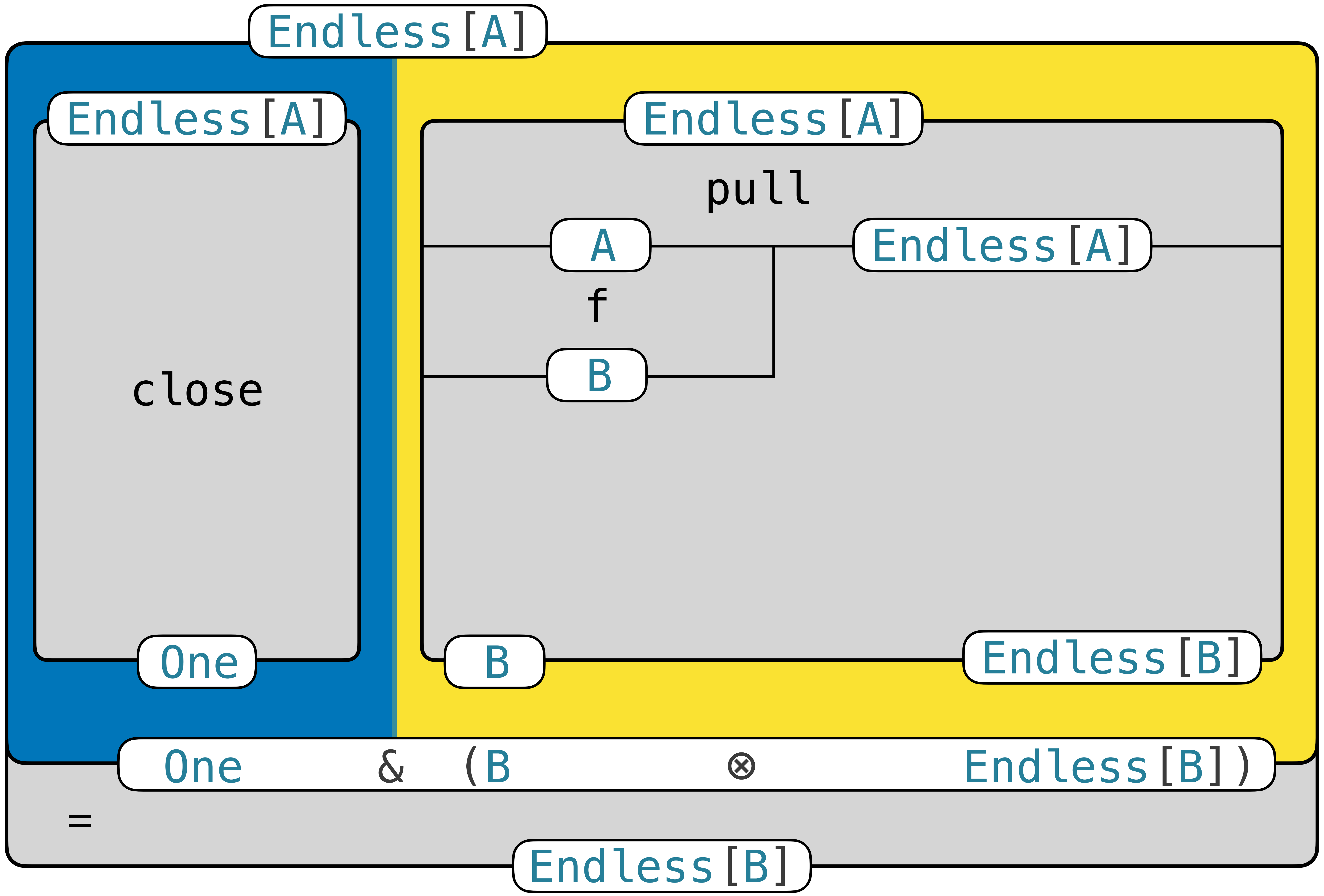
- Delay pulling from upstream until previous element has been *"handled"*

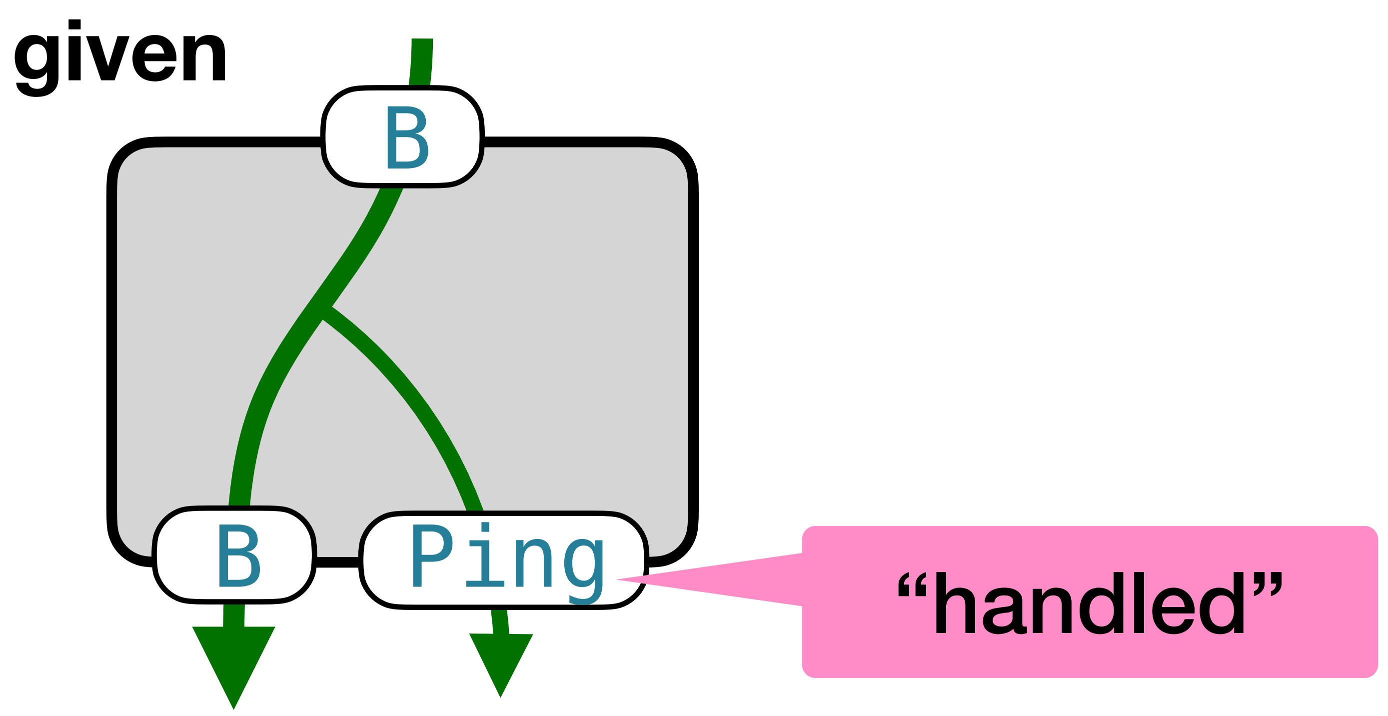


# Endless.mapSequentially(f)

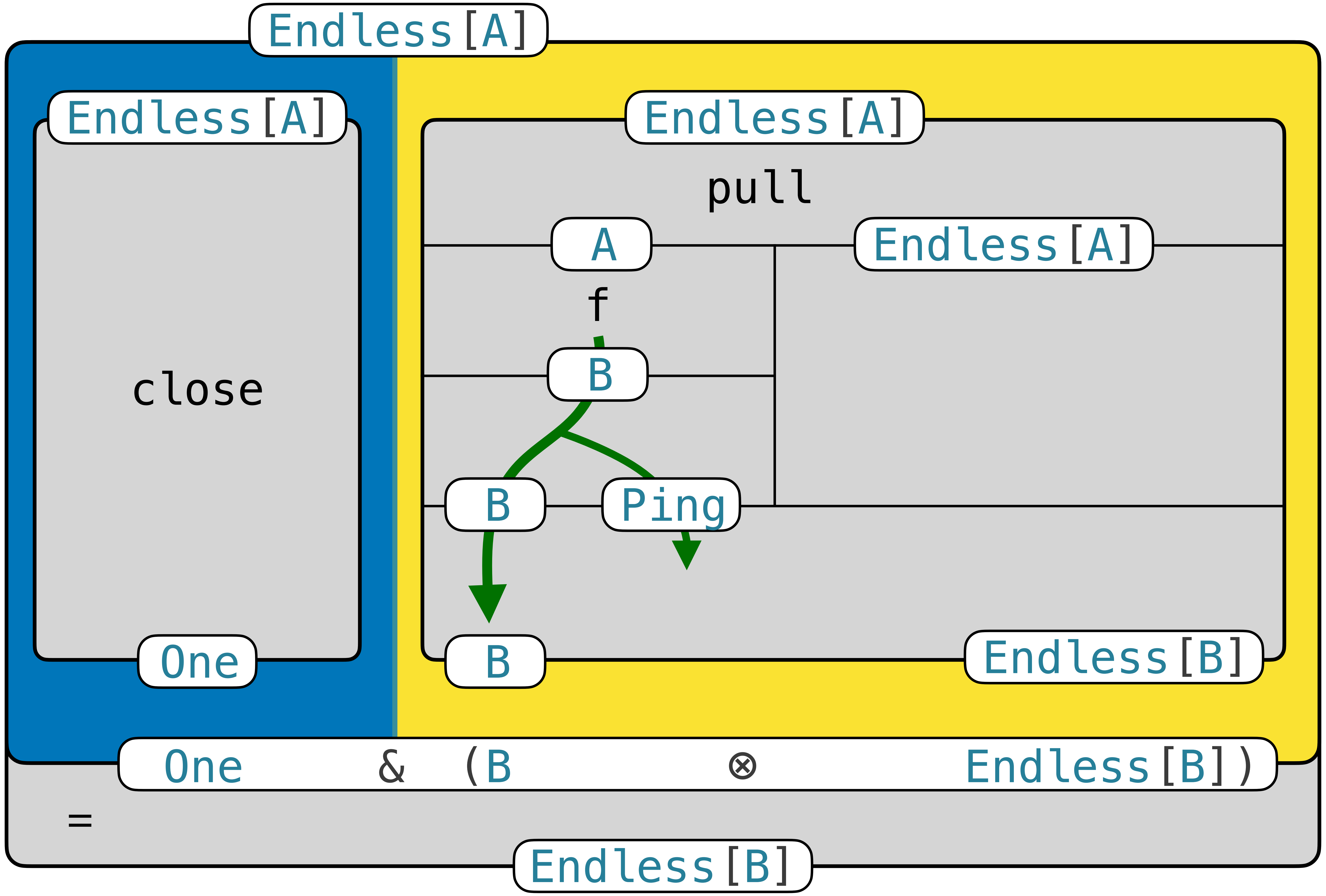
- Delay pulling from upstream until previous element has been *"handled"*

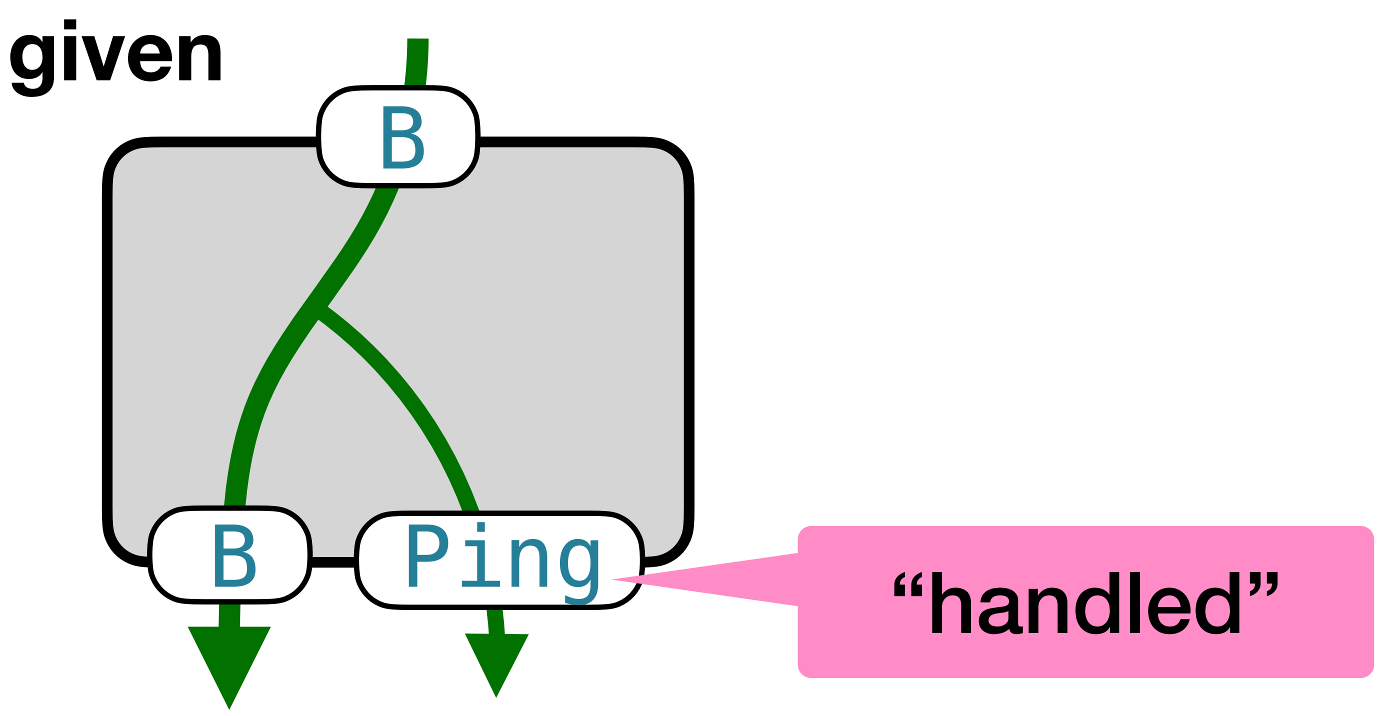


# Endless.mapSequentially(f)

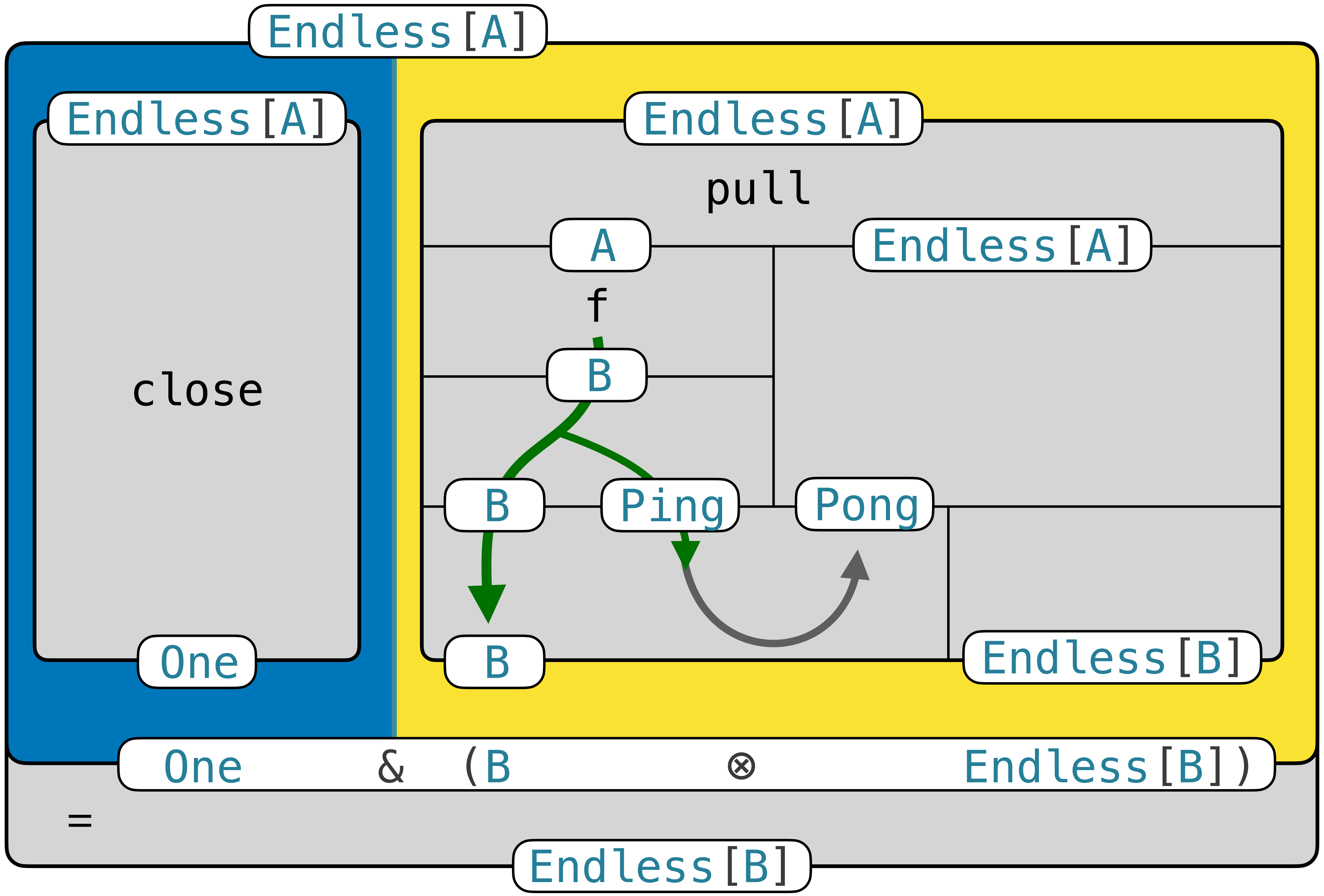
- Delay pulling from upstream until previous element has been *"handled"*

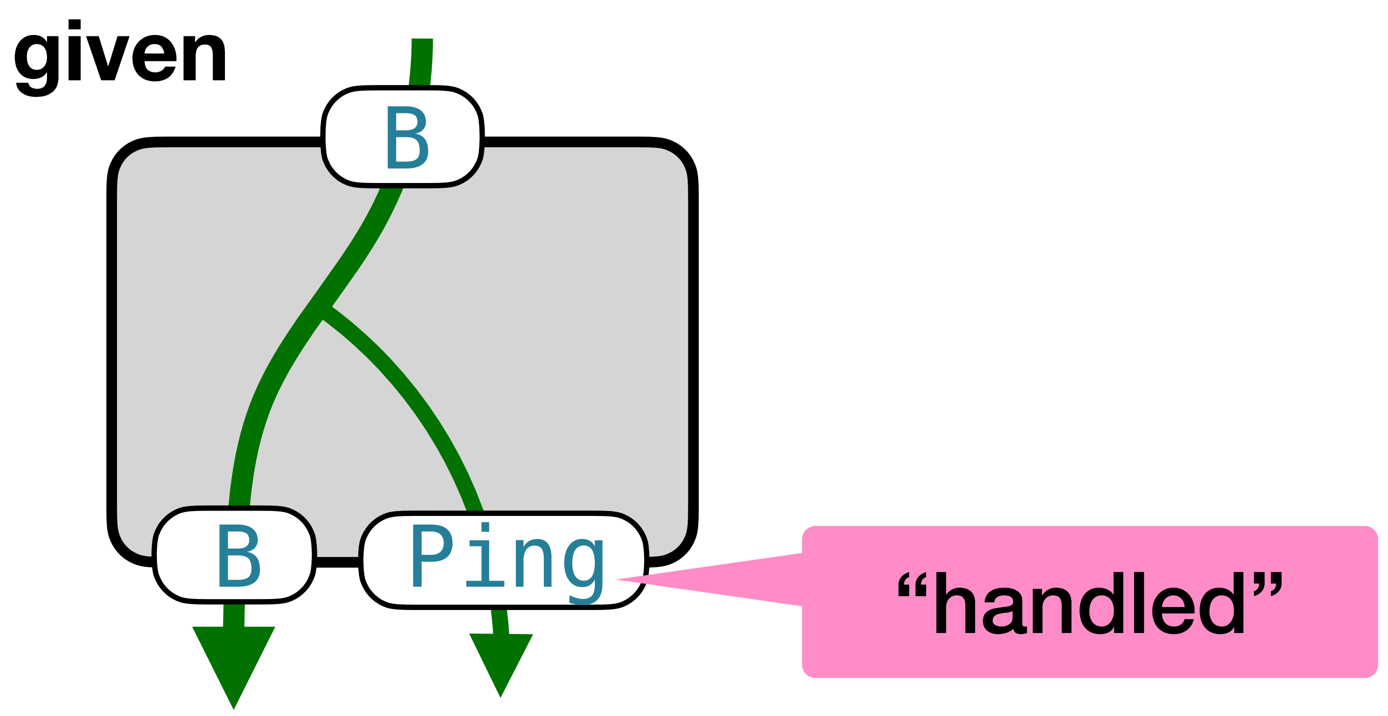


# Endless.mapSequentially(f)

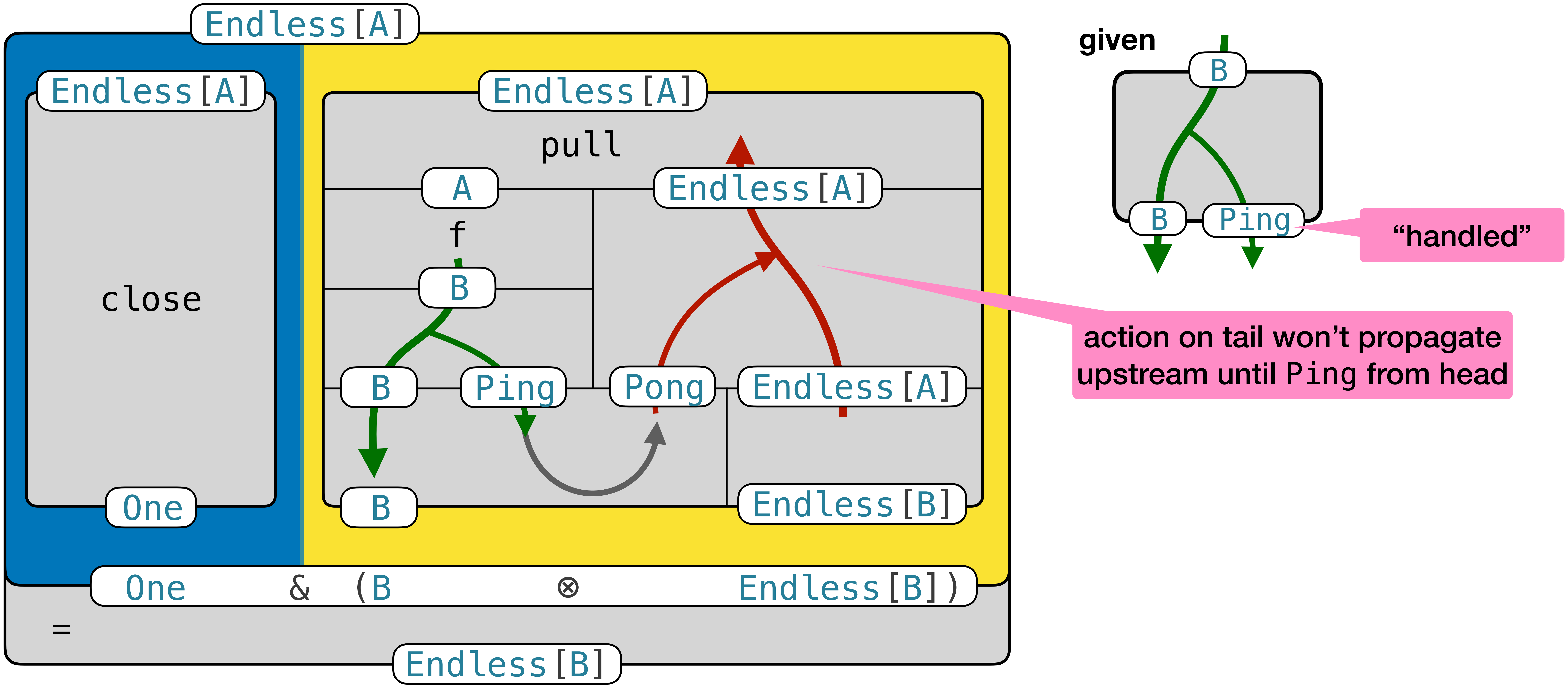
- Delay pulling from upstream until previous element has been *"handled"*



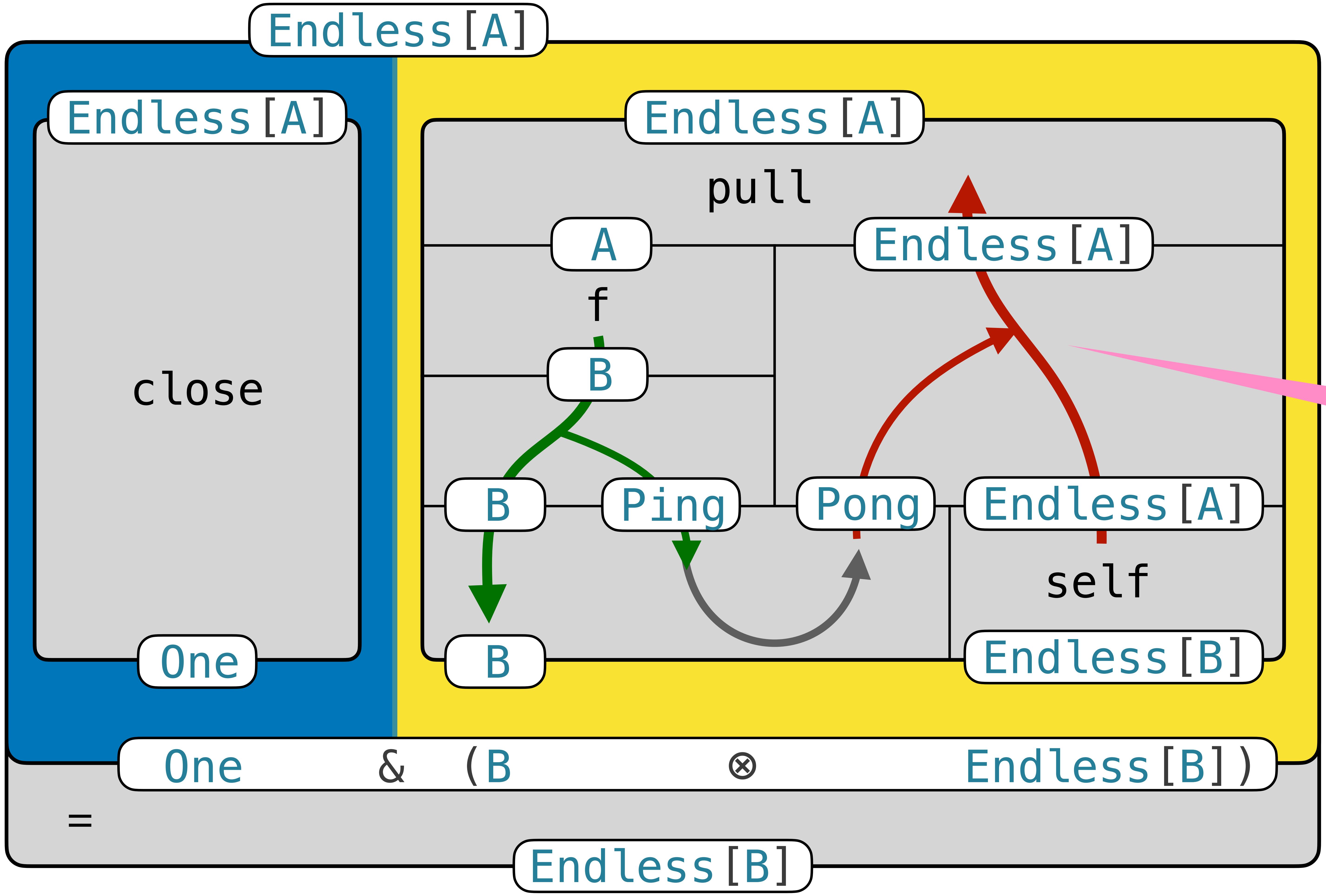
# Endless.mapSequentially(f)

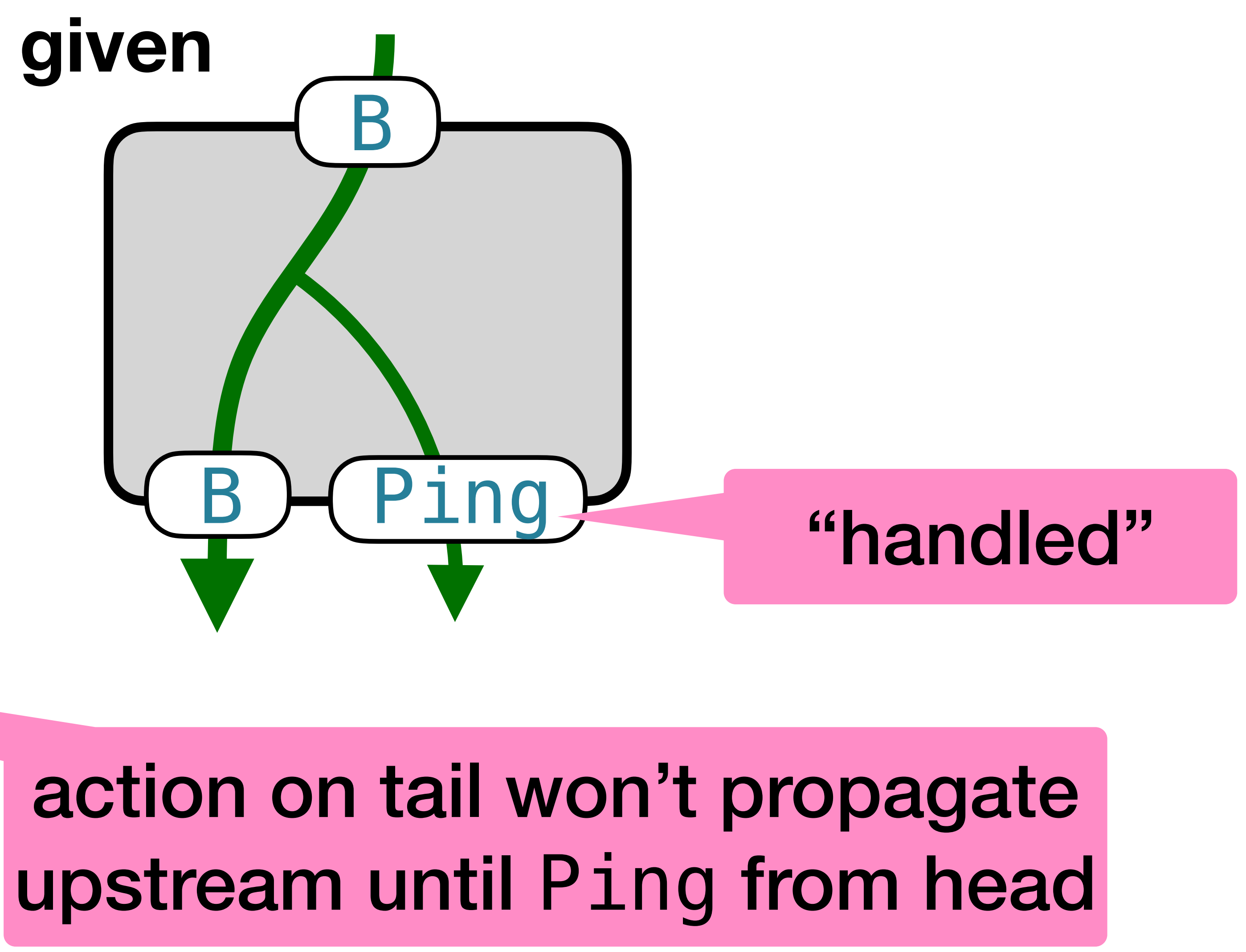
- Delay pulling from upstream until previous element has been *"handled"*



# Endless.mapSequentially(f)

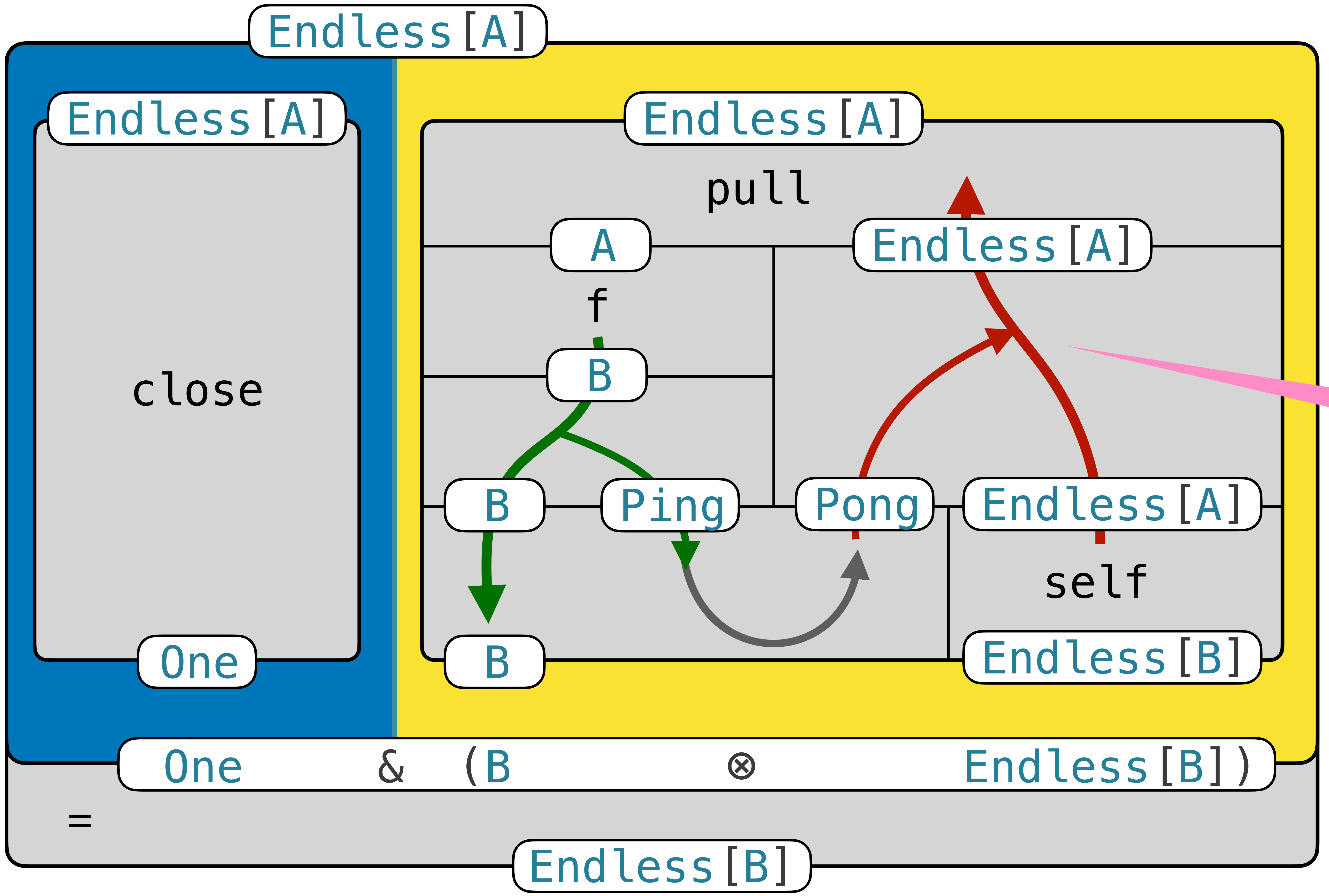
- Delay pulling from upstream until previous element has been *"handled"*

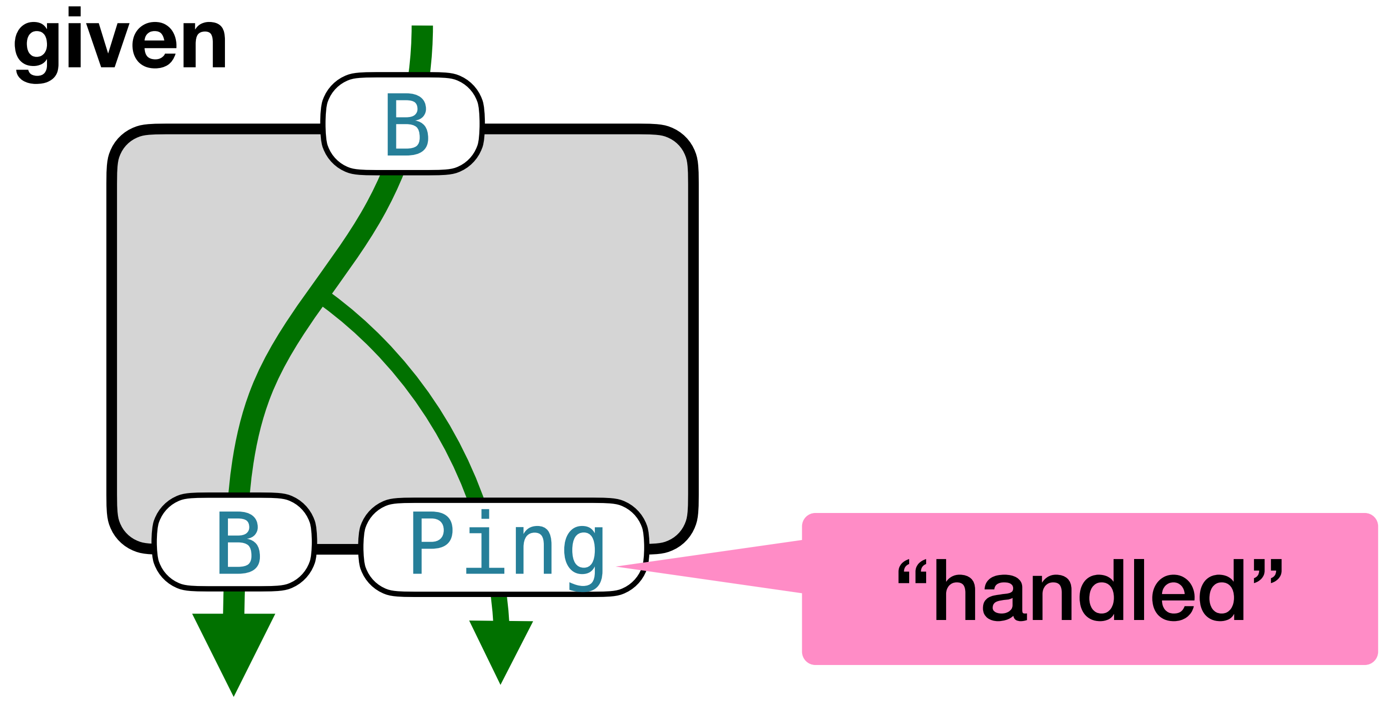


# Endless.mapSequentially(f)

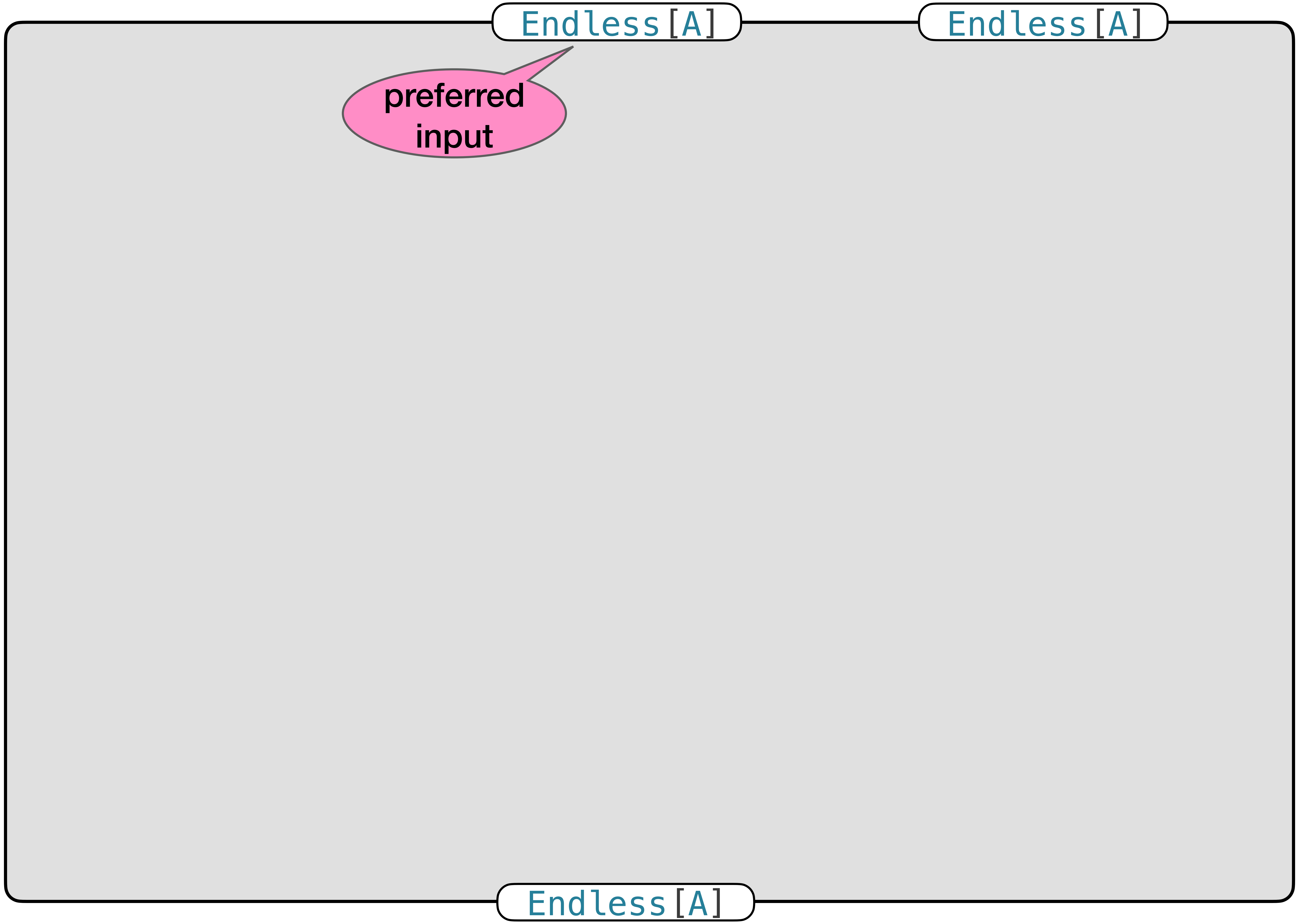
- Delay pulling from upstream until previous element has been *"handled"*



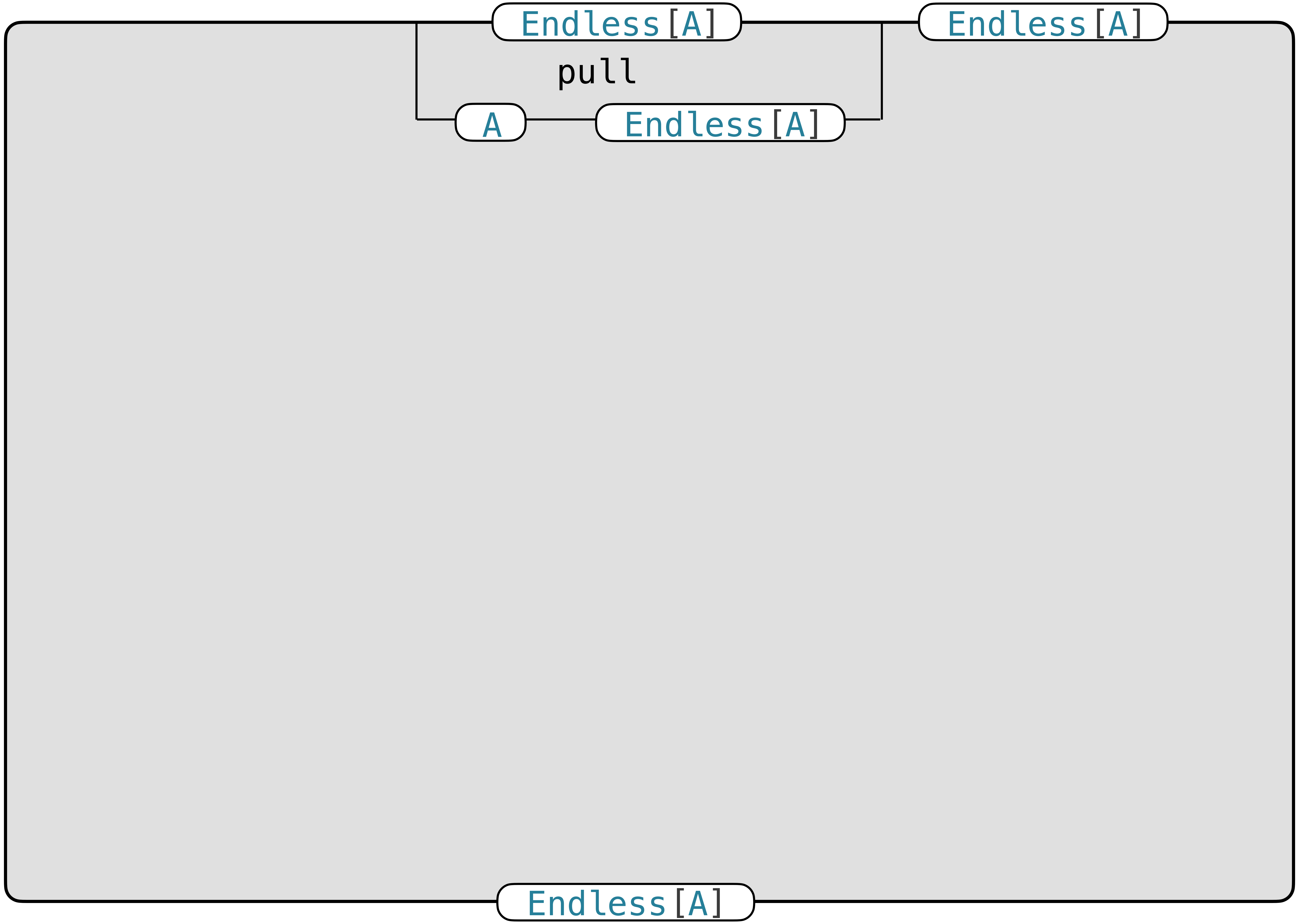




**Sequencing takes effort**

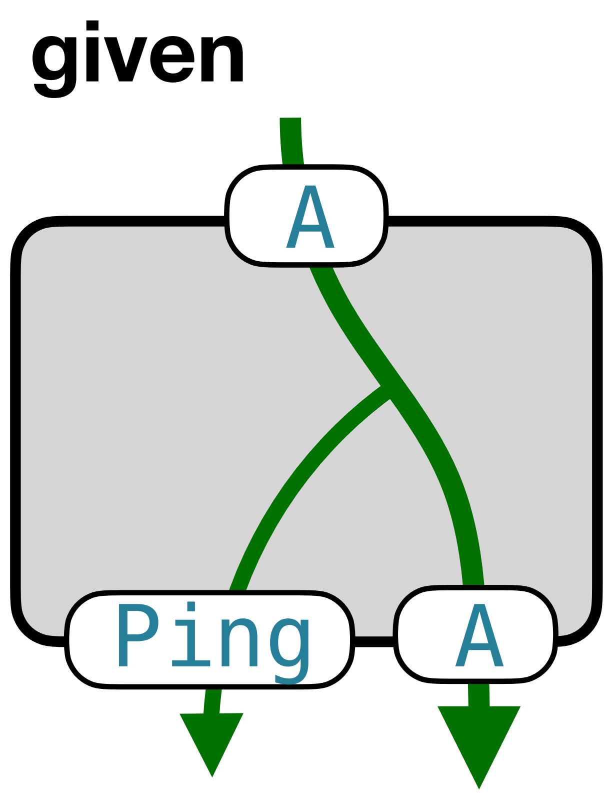
# Endless.mergePreferred

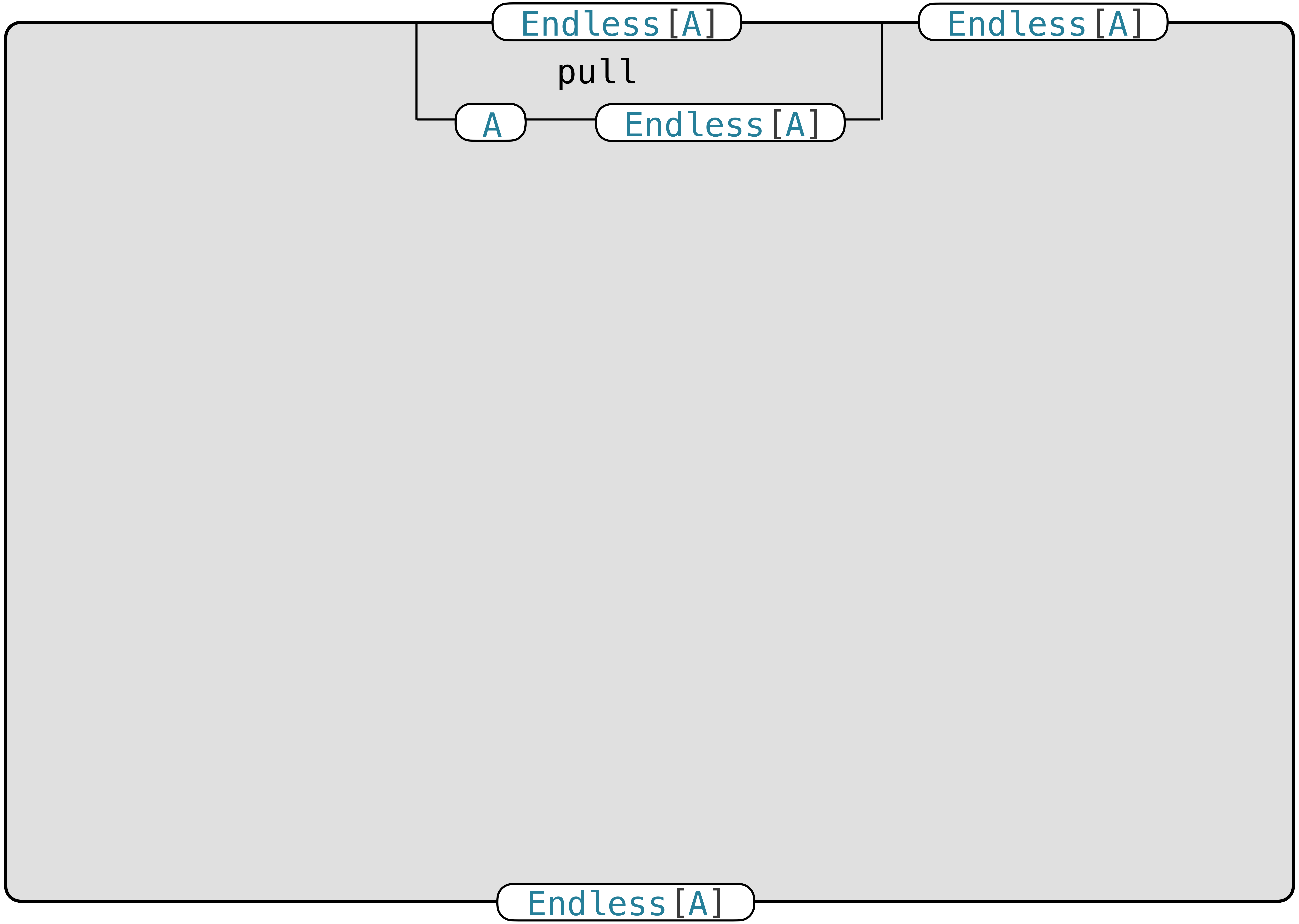


# Endless.mergePreferred

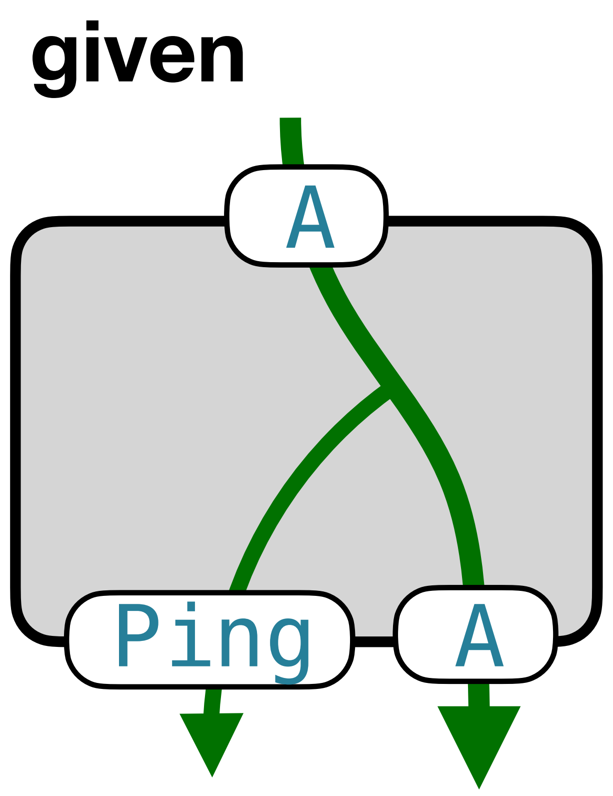


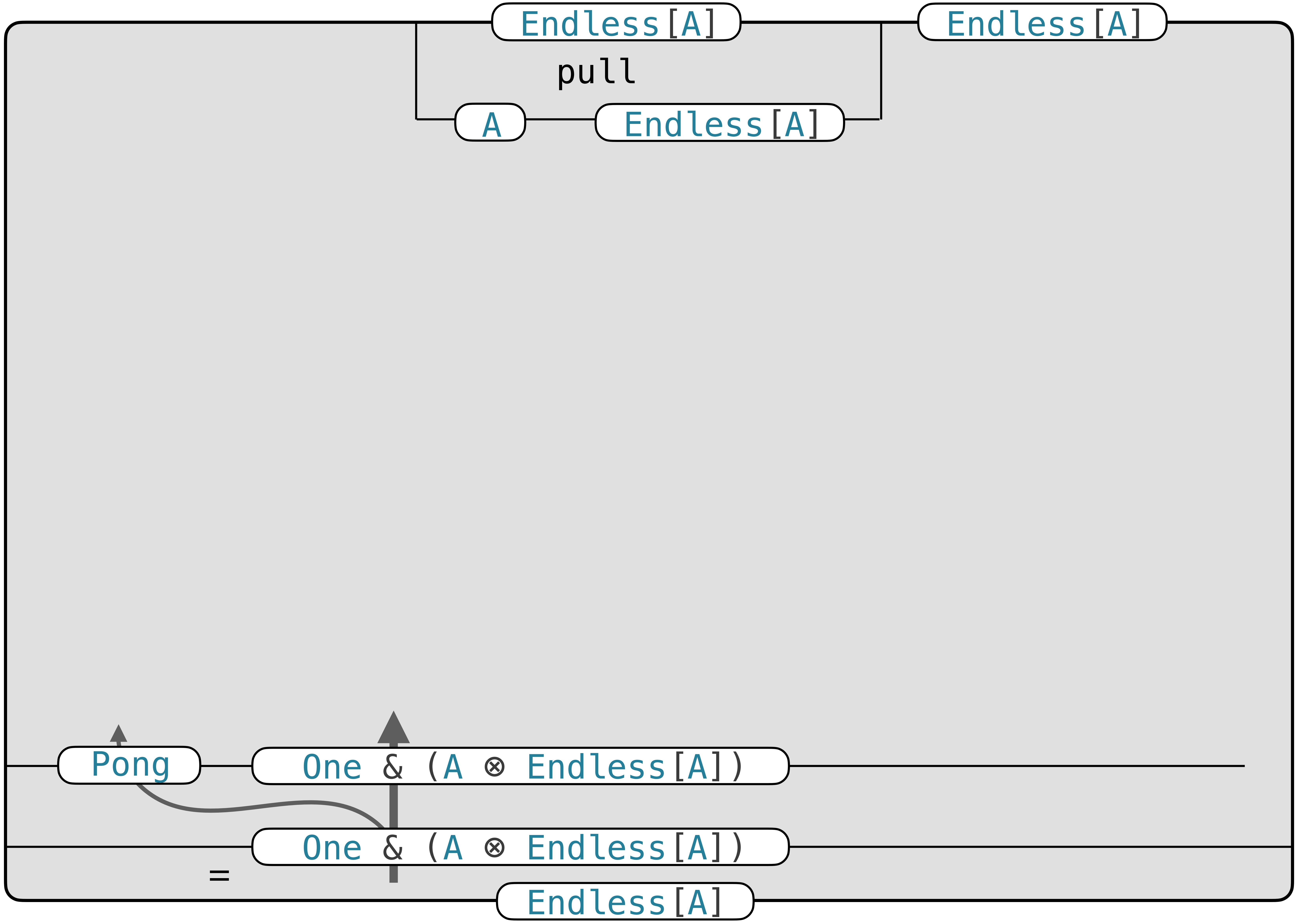
# Endless.mergePreferred



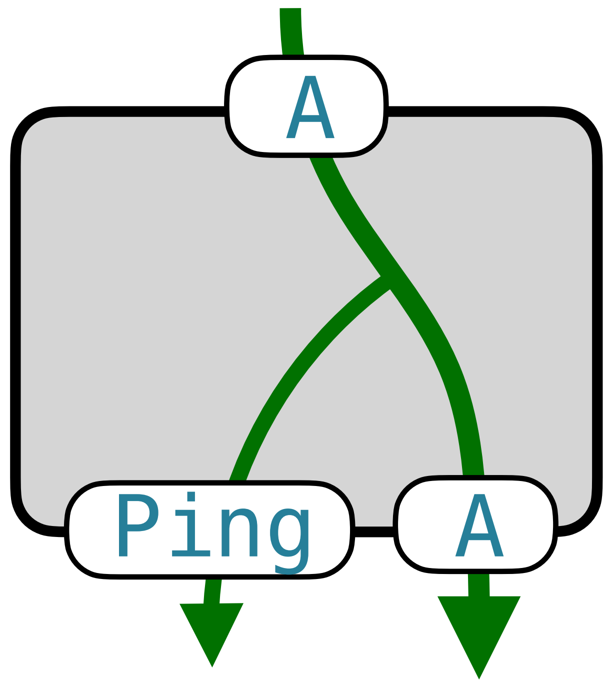
# Endless.mergePreferred

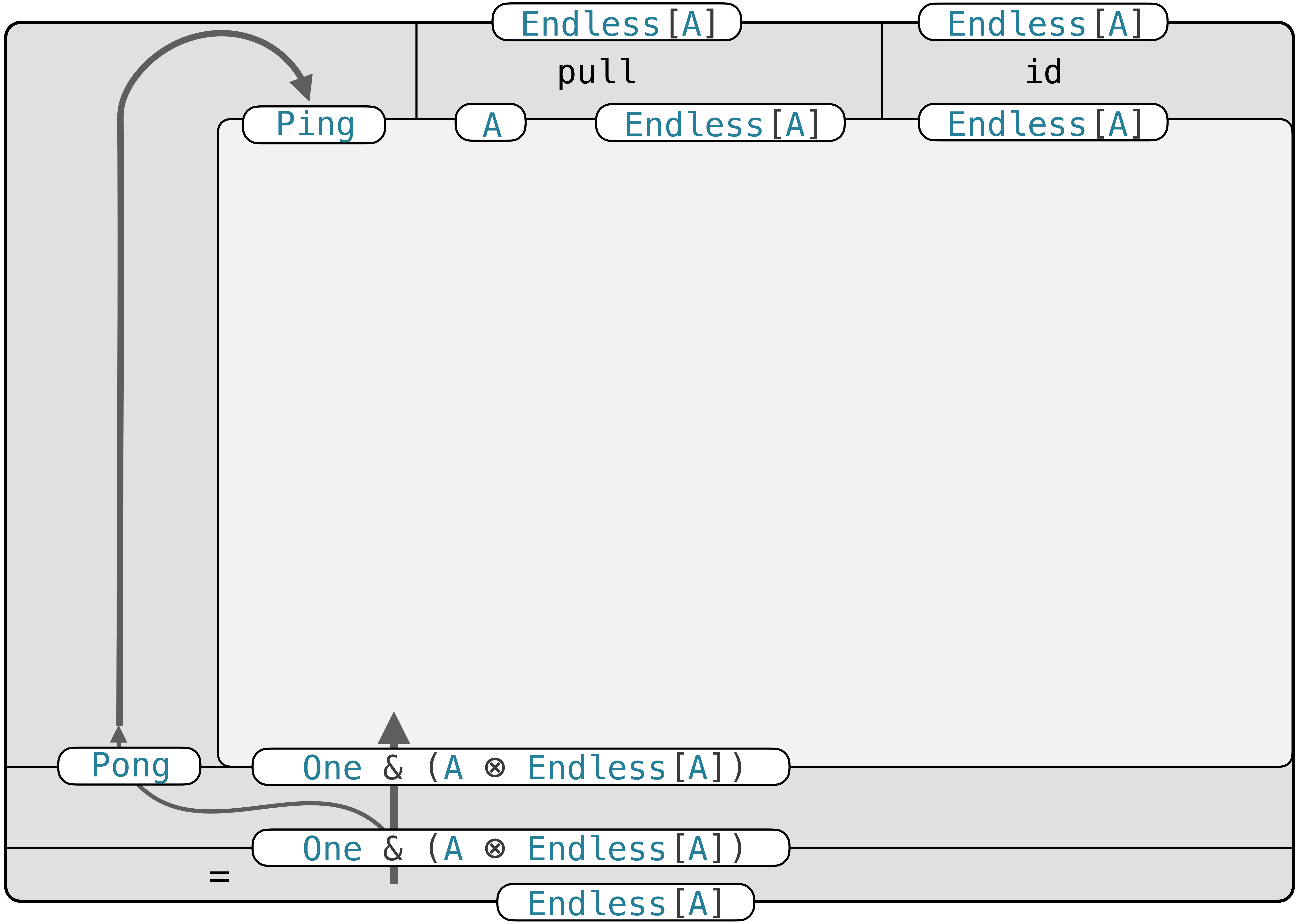


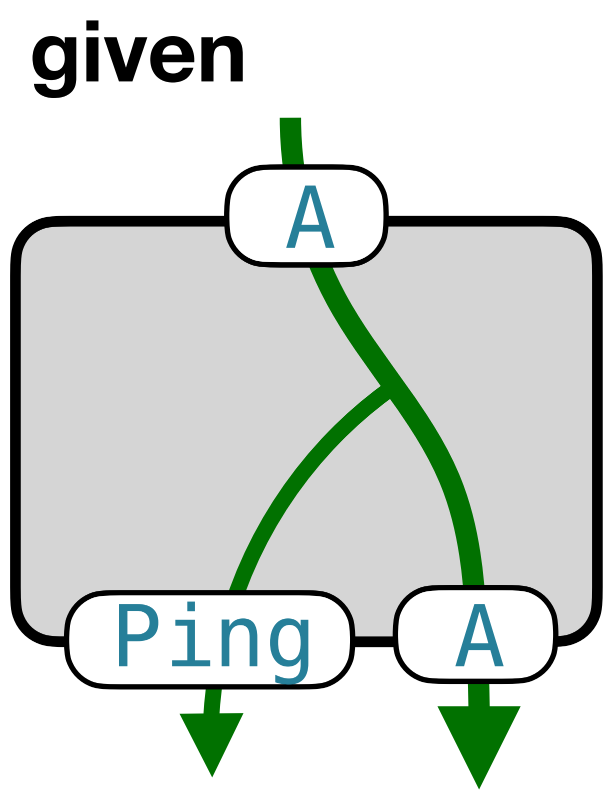
# Endless.mergePreferred

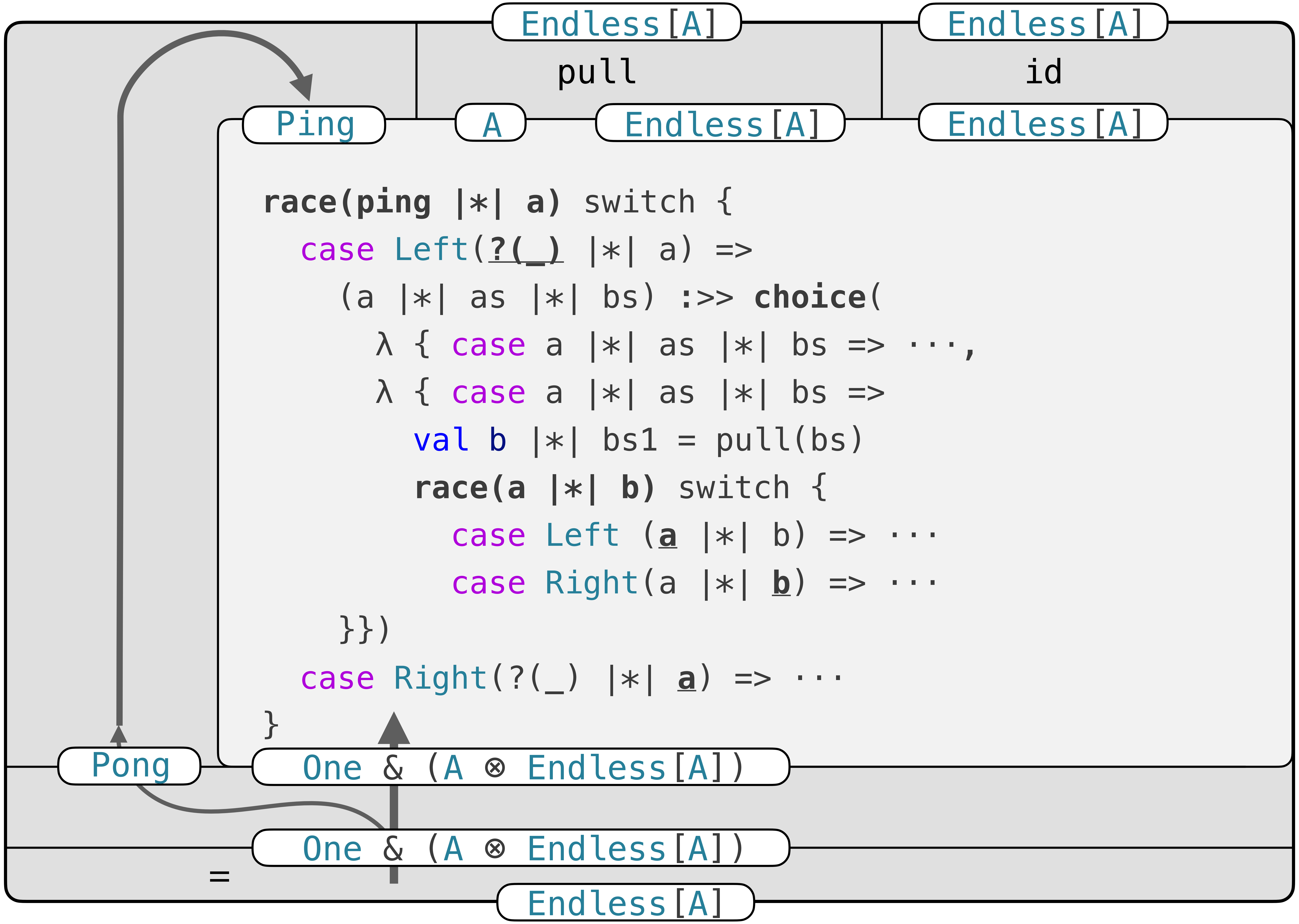
given



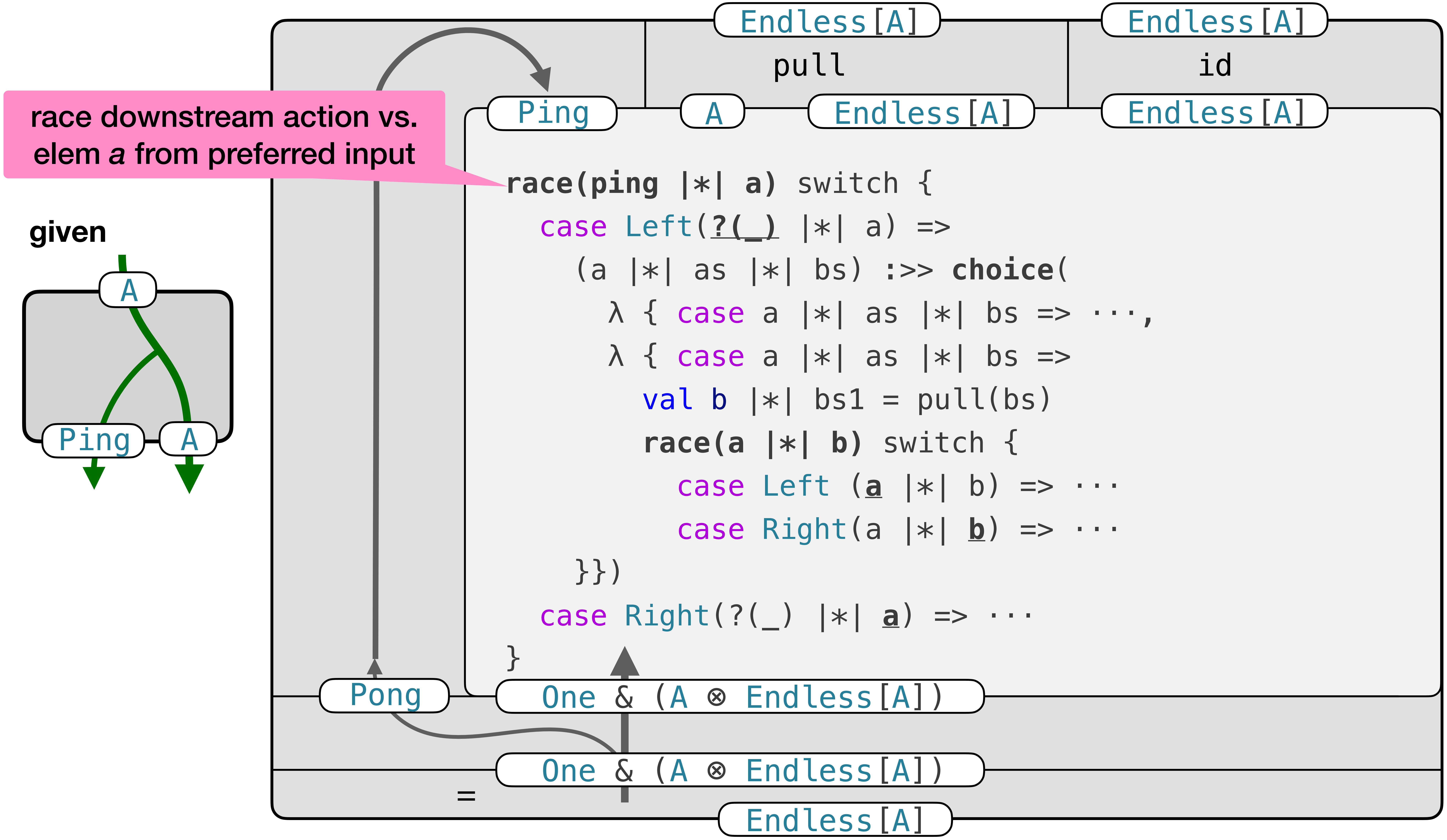
# Endless.mergePreferred



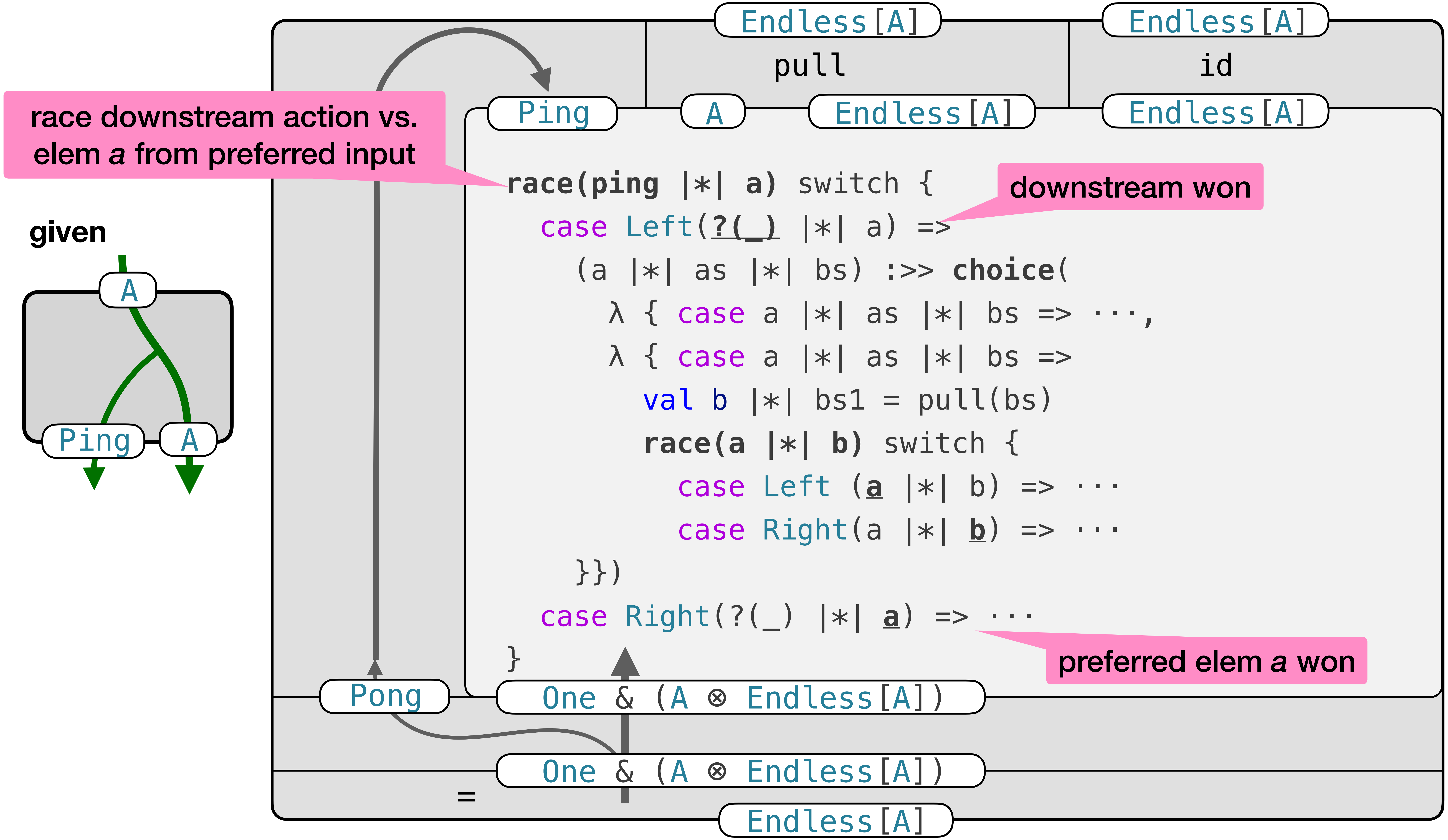




```
race(ping |*| a) switch {
  case Left(?(_) |*| a) =>
    (a |*| as |*| bs) :>> choice(
      λ { case a |*| as |*| bs => ···,
      λ { case a |*| as |*| bs =>
        val b |*| bs1 = pull(bs)
        race(a |*| b) switch {
          case Left (a |*| b) => ···
          case Right(a |*| b) => ···
    }})
  case Right(?(_) |*| a) => ···
}
```

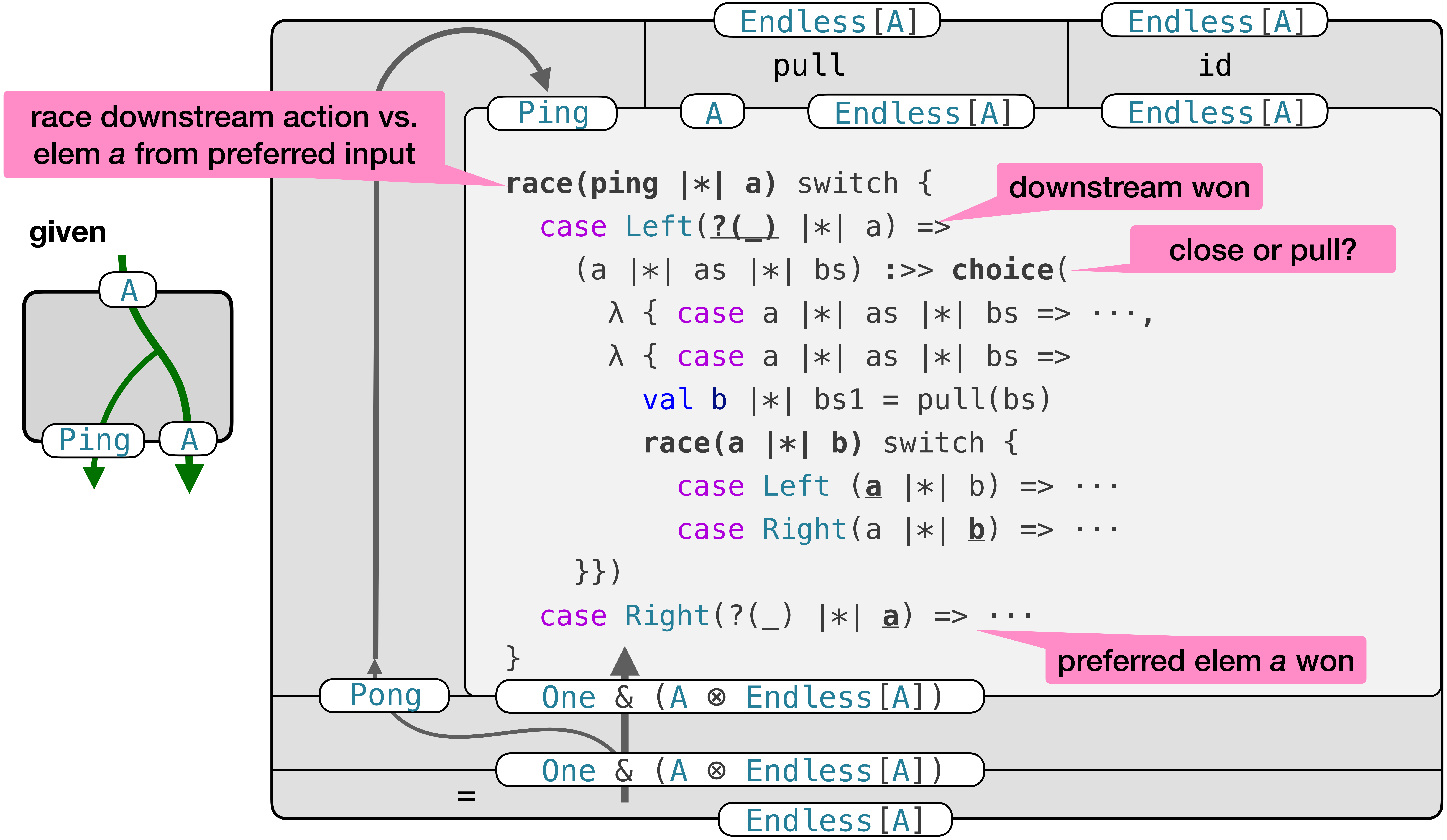
# Endless.mergePreferred



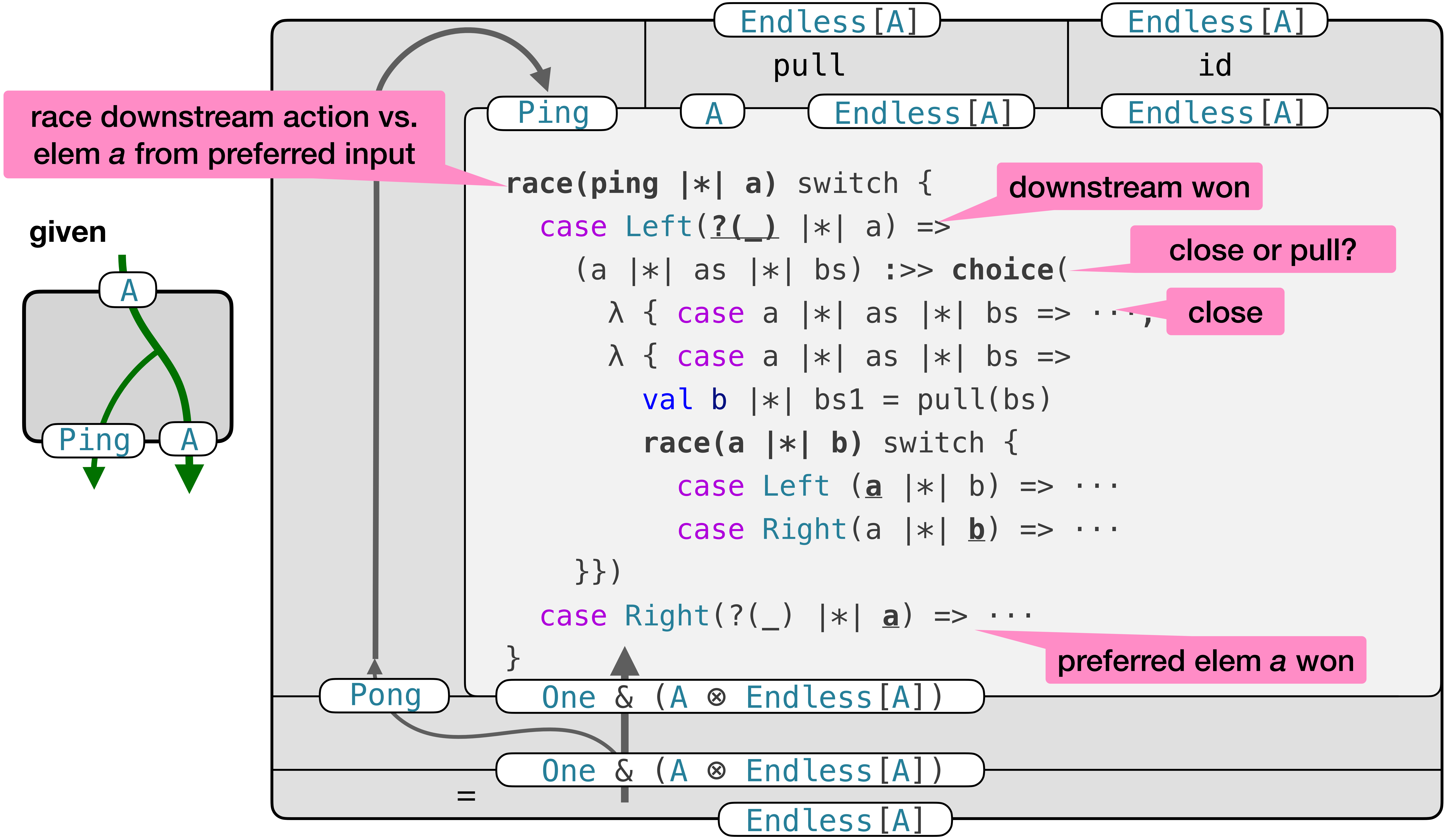
# Endless.mergePreferred



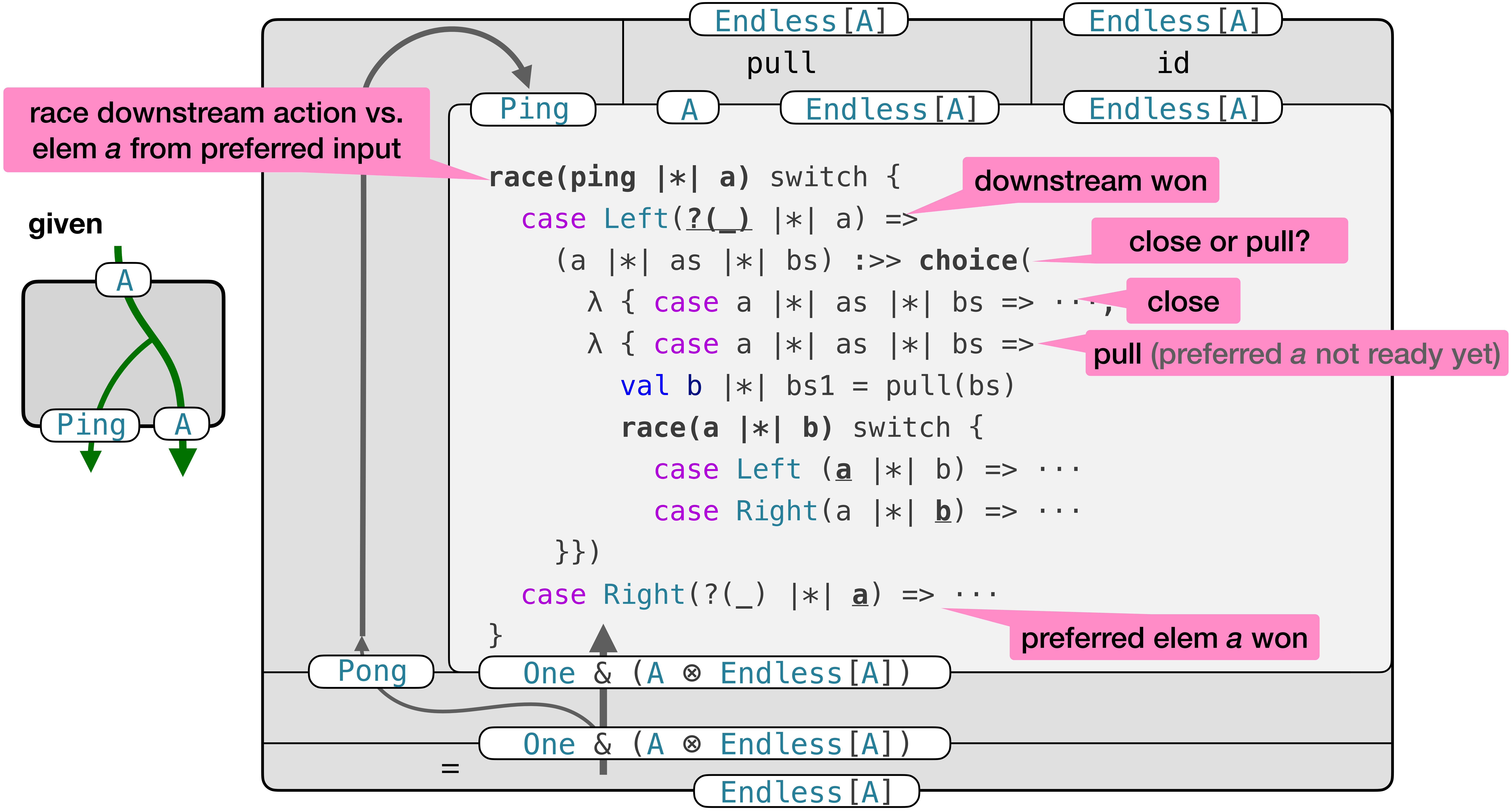
# Endless.mergePreferred



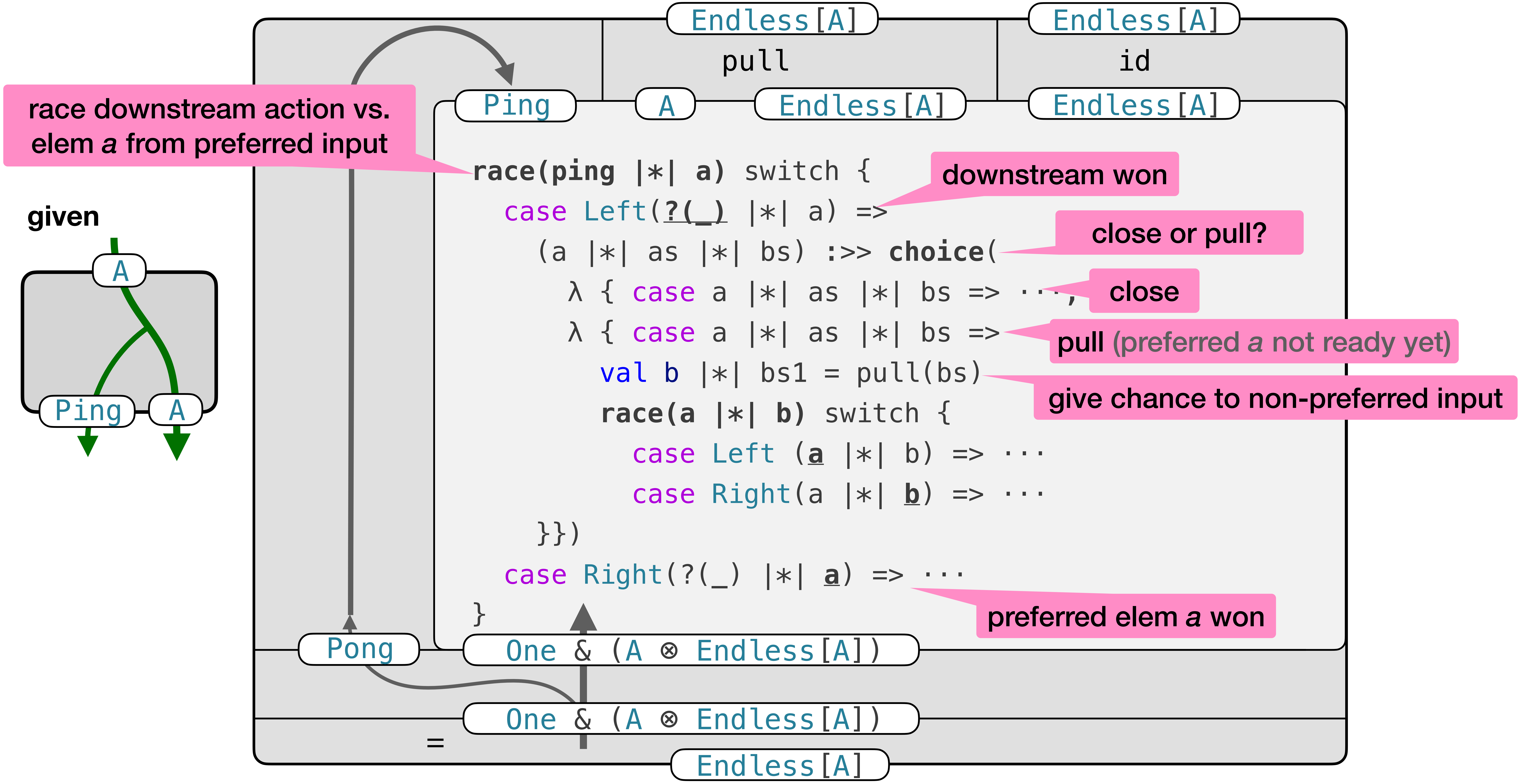
# Endless.mergePreferred



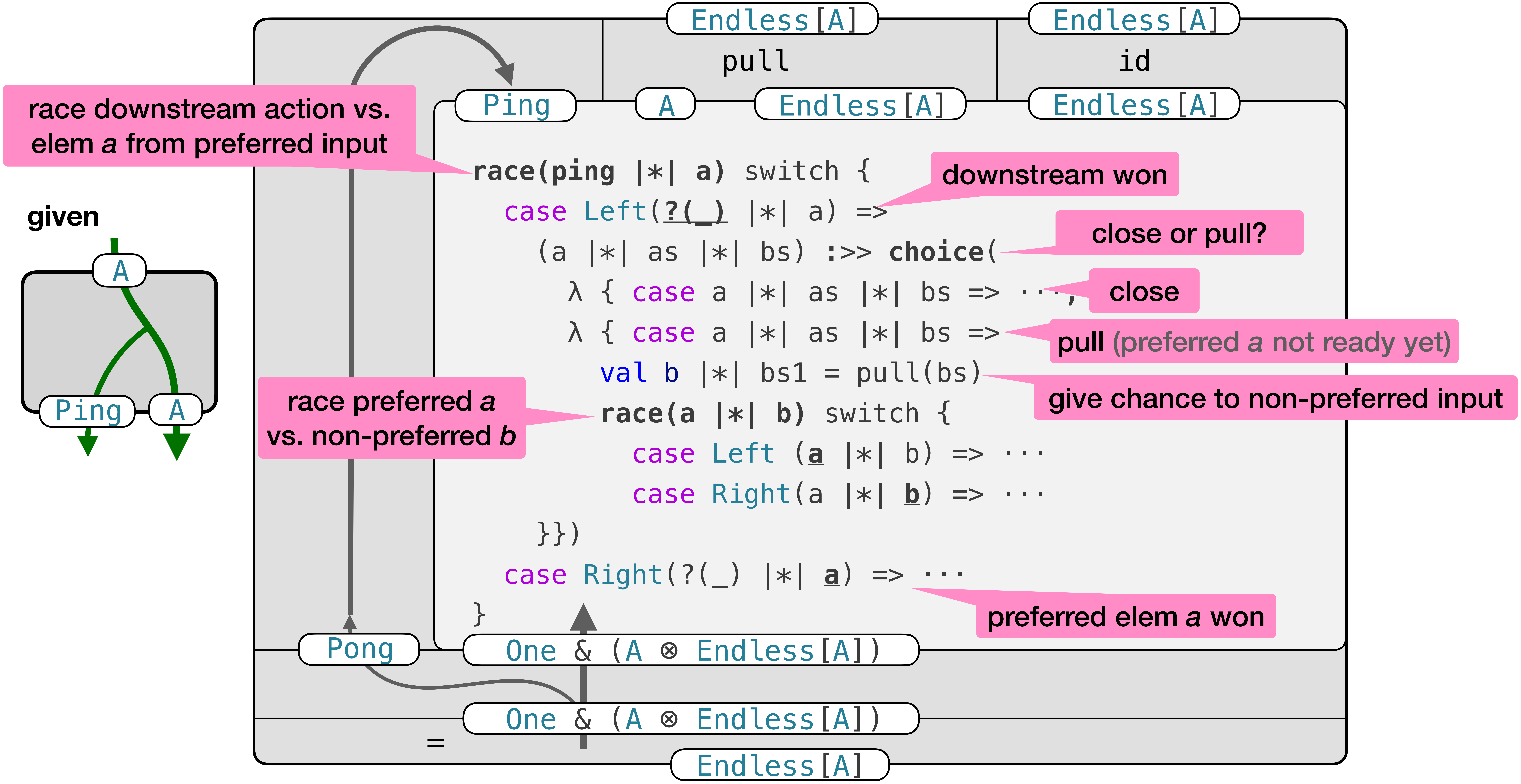
# Endless.mergePreferred



# Endless.mergePreferred



# Endless.mergePreferred



# Endless.mergePreferred

# The Santa Claus Problem

https://santaclausproblem.cs.unlv.edu/

# The Santa Claus Problem

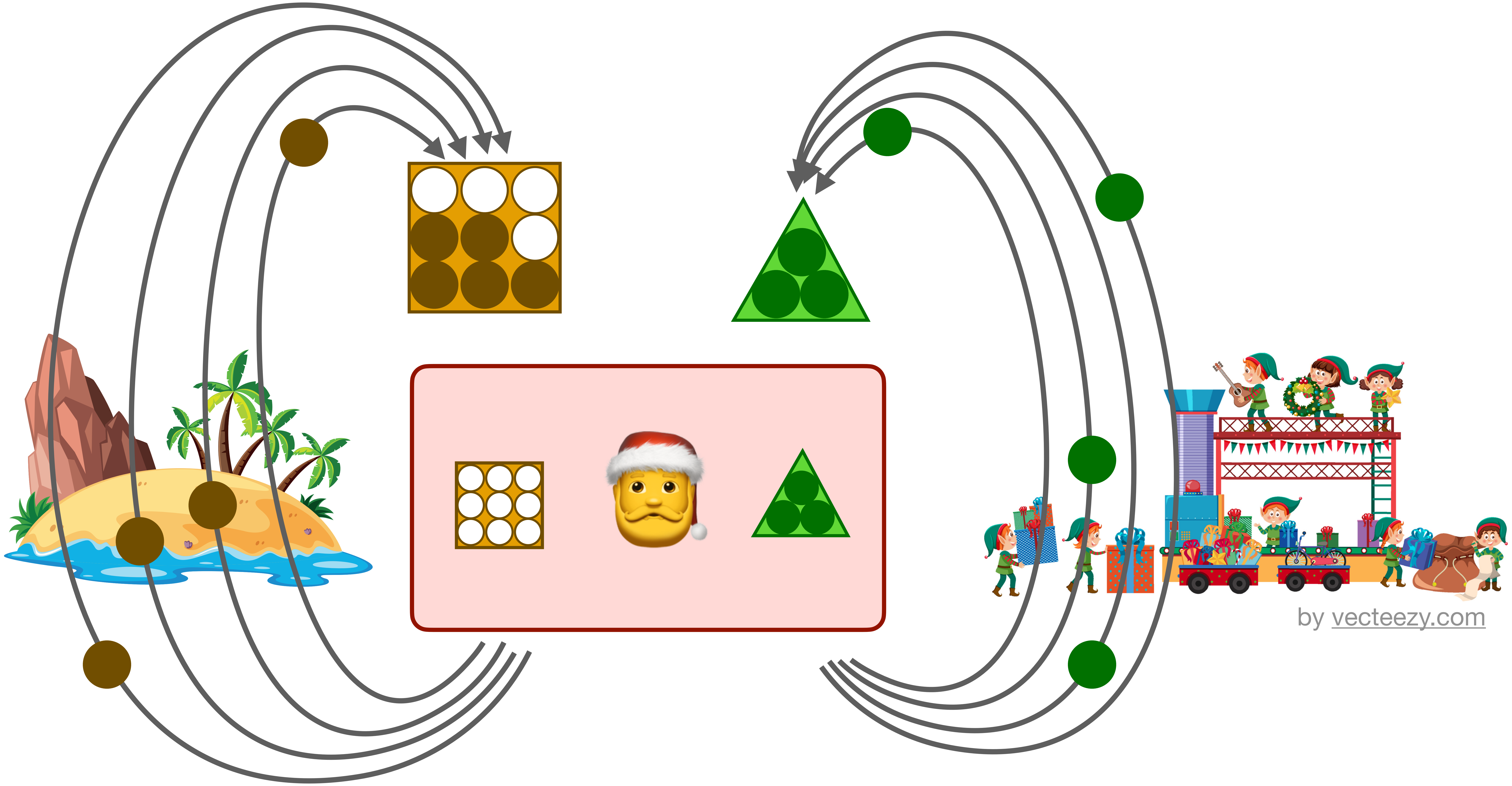
Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeer being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys. One elf's problem is never serious enough to wake up Santa (otherwise, he may *never* get any sleep), so, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer don't want to leave the tropics, and therefore they stay there until the last possible moment. They might not even come back, but since Santa is footing the bill for their year in paradise ... This could also explain the quickness in their delivering of presents, since the reindeer can't wait to get back to where it is warm.) The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

Trono, J.A. (1994). A new exercise in concurrency. ACM SIGCSE Bull., 26, 8-10.

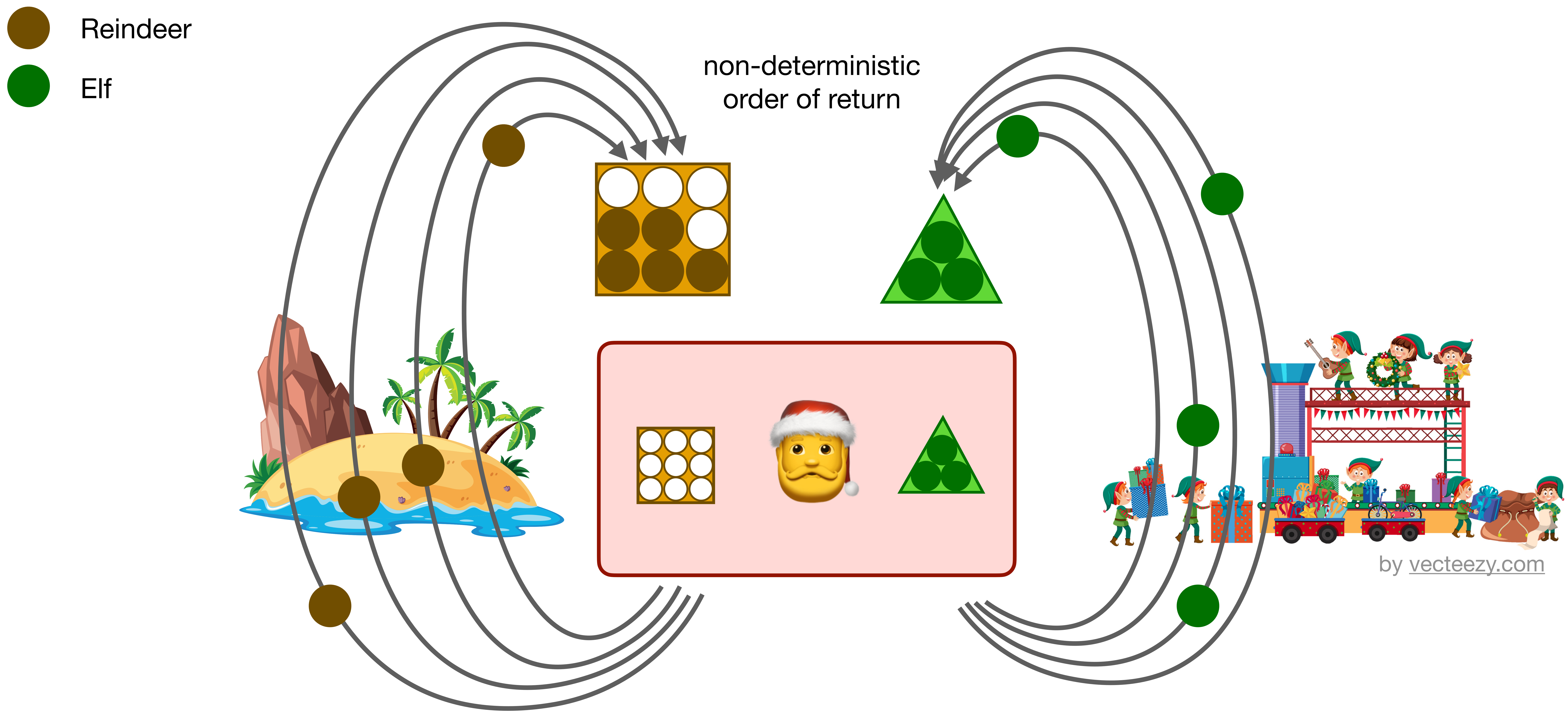
# The Santa Claus Problem

Reindeer

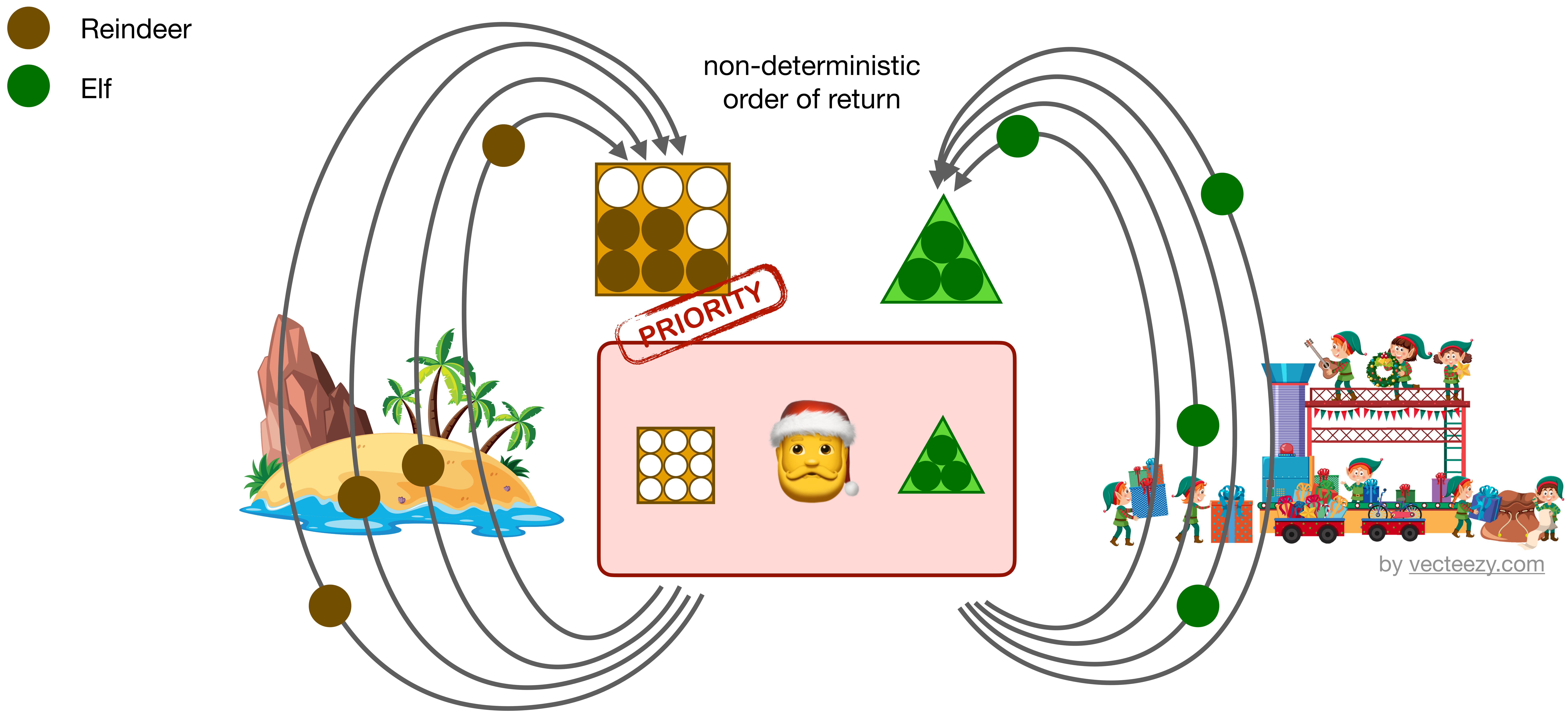
Elf

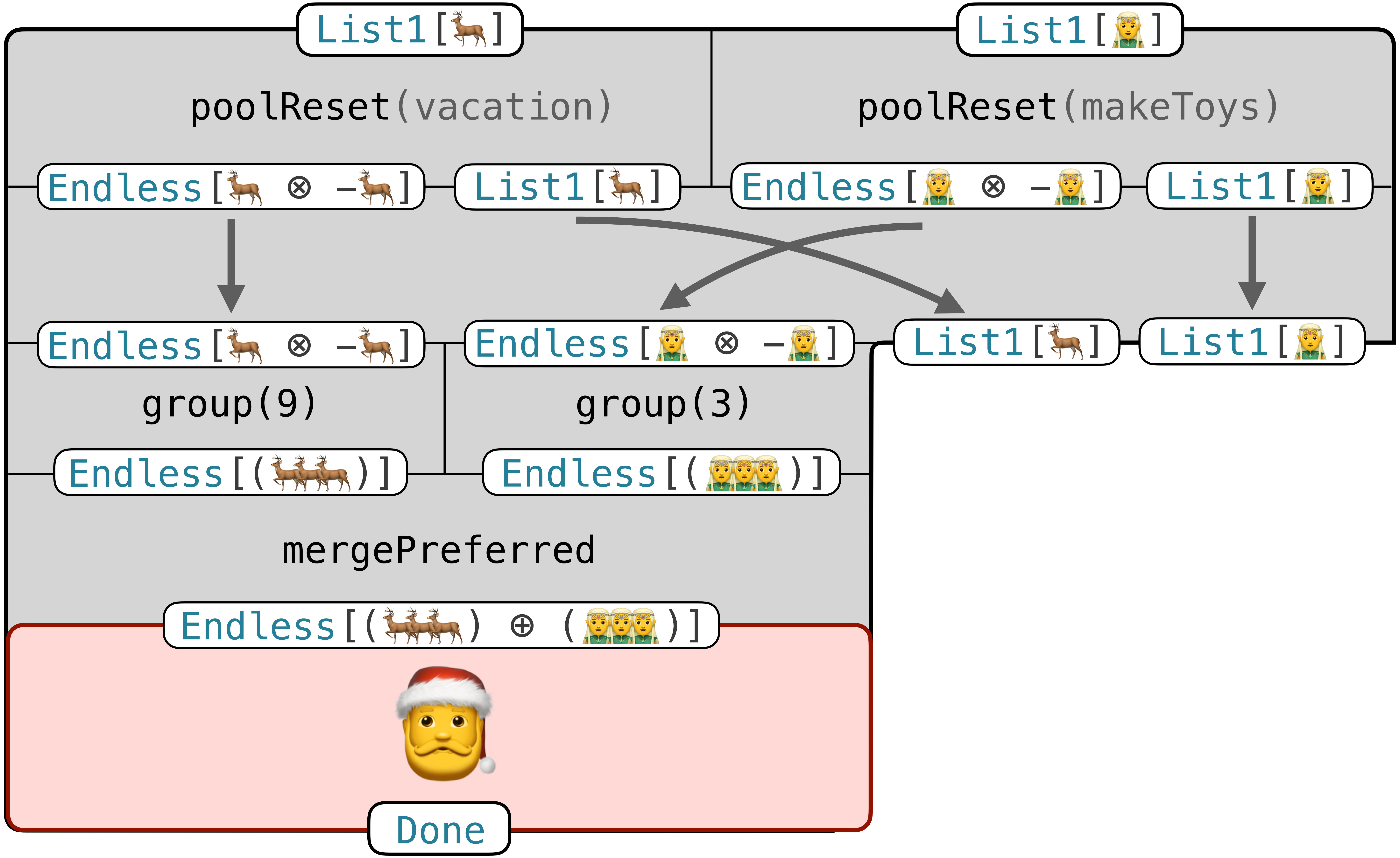


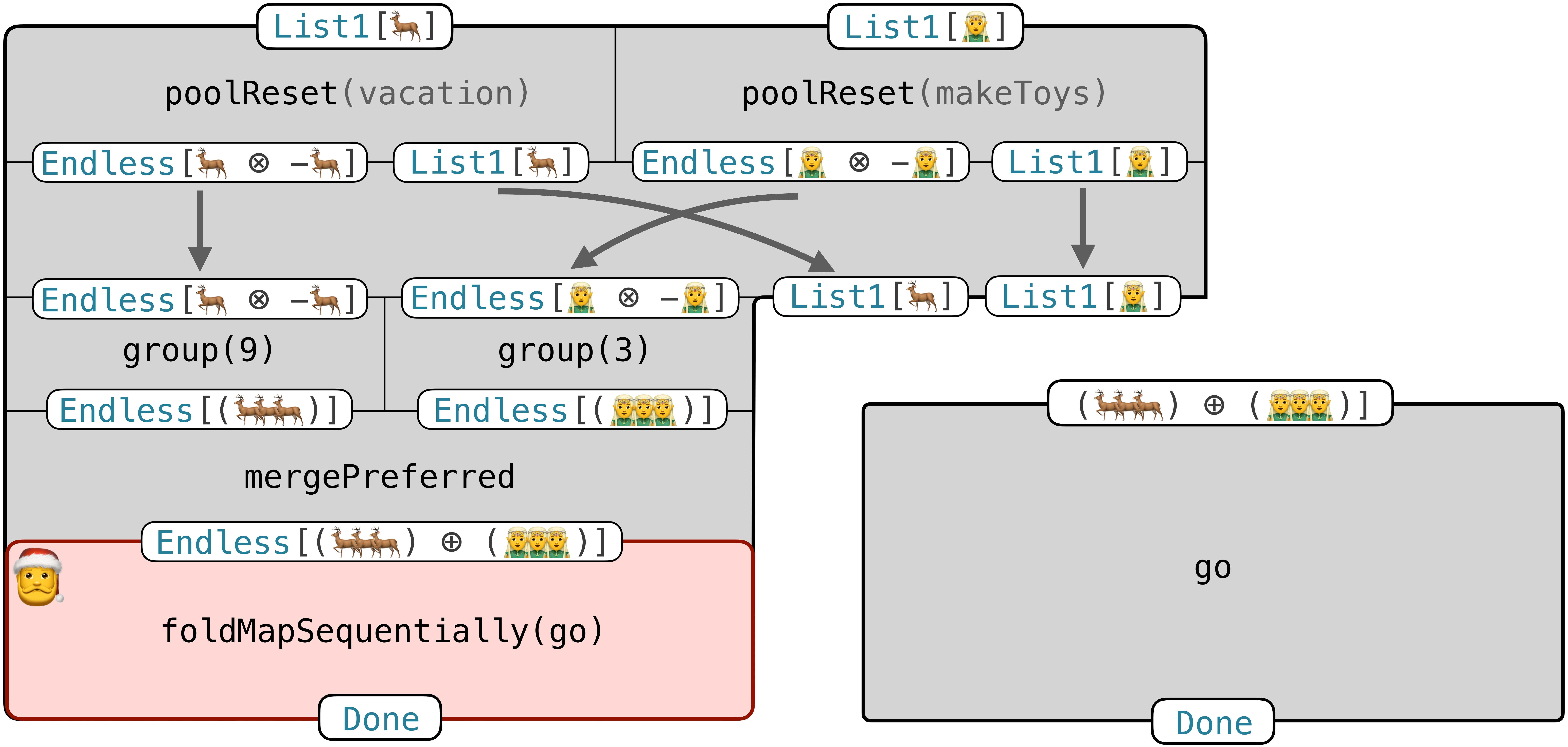
# The Santa Claus Problem



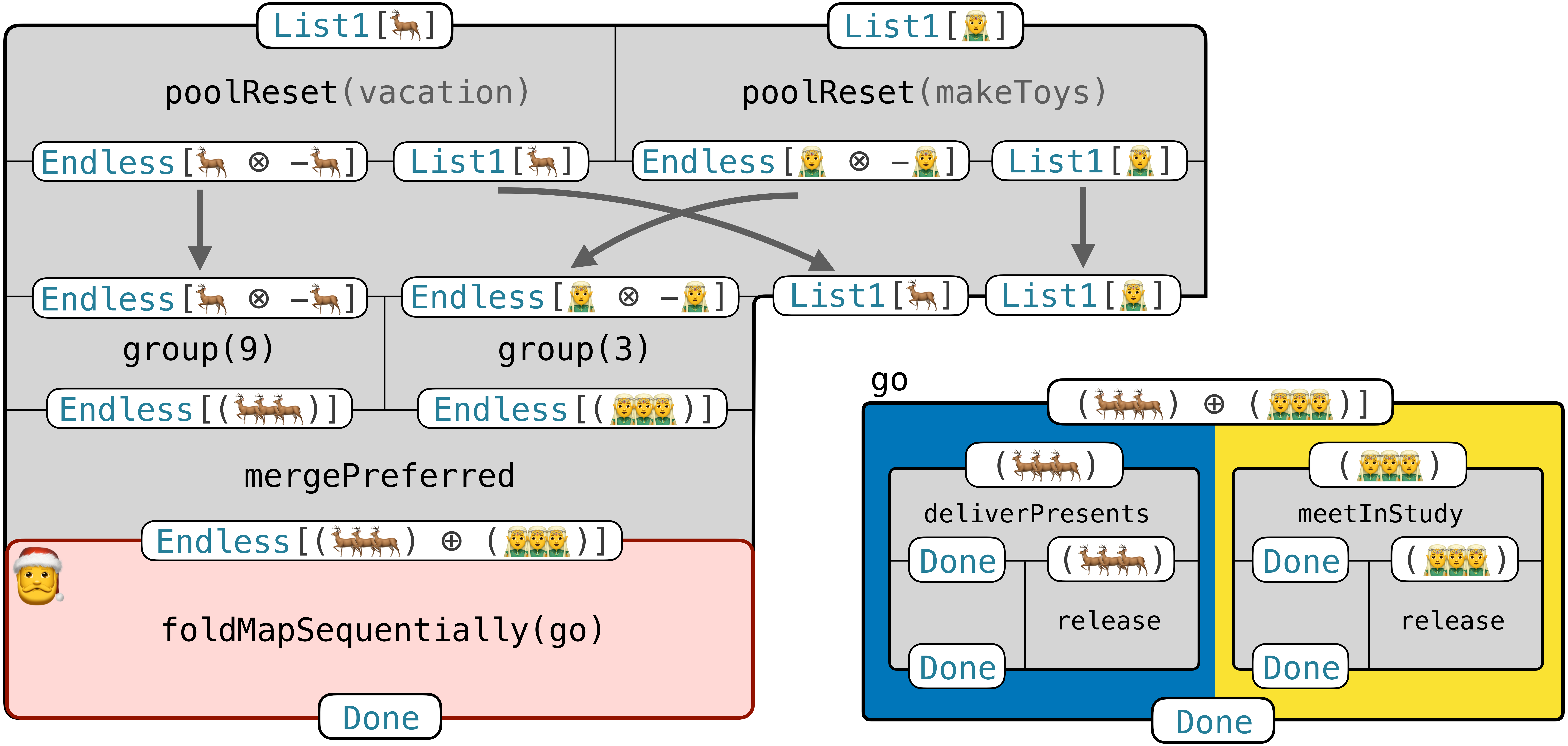
# The Santa Claus Problem



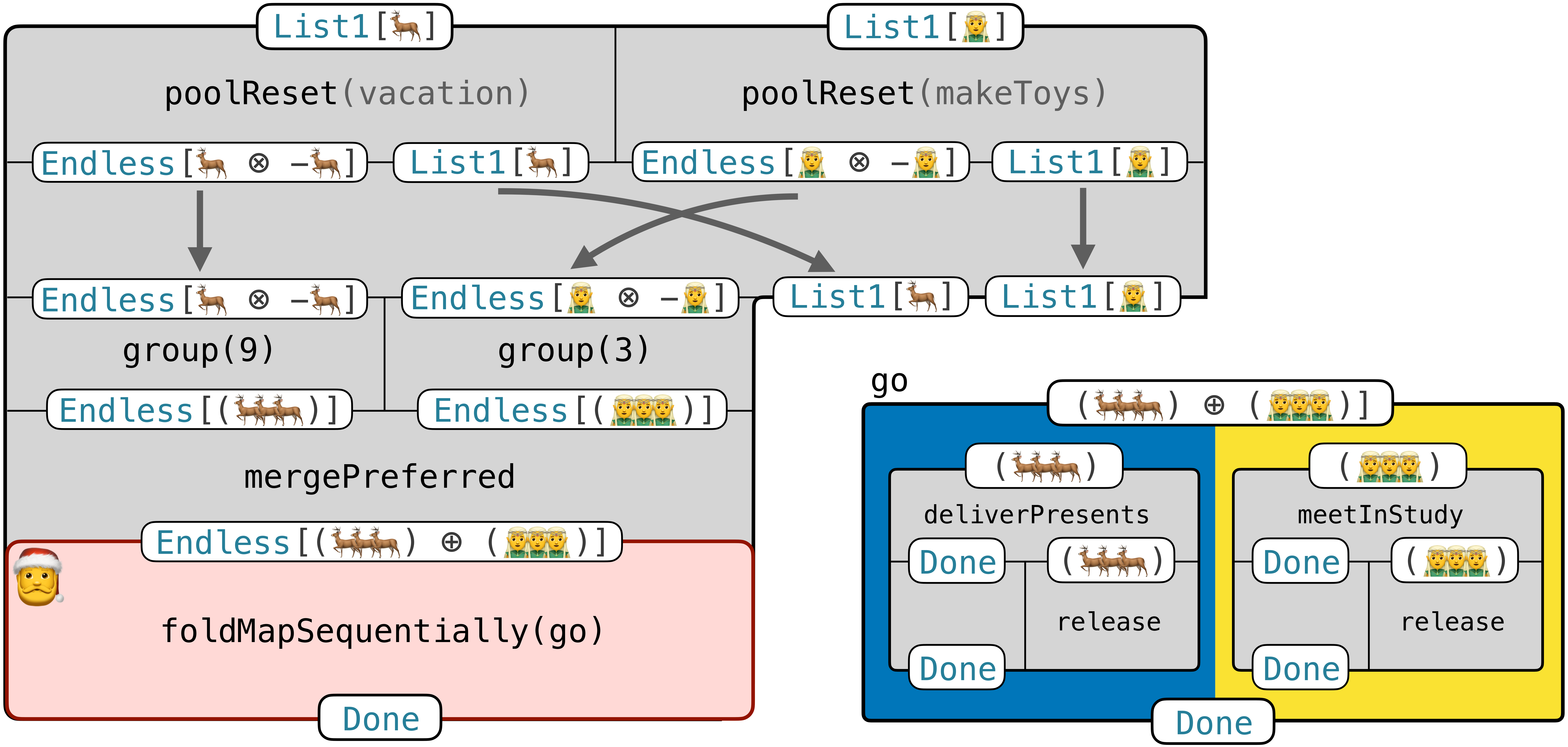

List1[🦌]
List1[🧝]
poolReset(vacation)
poolReset(makeToys)
Endless[🦌 ⊗ −🦌]
List1[🦌]
Endless[🧝 ⊗ −🧝]
List1[🧝]
Endless[🦌 ⊗ −🦌]
Endless[🧝 ⊗ −🧝]
List1[🦌]
List1[🧝]
group(9)
group(3)
Endless[(🦌🦌🦌)]
Endless[(🧝🧝🧝)]
mergePreferred
Endless[(🦌🦌🦌) ⊕ (🧝🧝🧝)]
Done



List1[🦌]
List1[🧝]
poolReset(vacation)
poolReset(makeToys)
Endless[🦌 ⊗ –🦌]
List1[🦌]
Endless[🧝 ⊗ –🧝]
List1[🧝]
Endless[🦌 ⊗ –🦌]
Endless[🧝 ⊗ –🧝]
List1[🦌]
List1[🧝]
group(9)
group(3)
Endless[(🦌🦌🦌)]
Endless[(🧝🧝🧝)]
mergePreferred
Endless[(🦌🦌🦌) ⊕ (🧝🧝🧝)]
foldMapSequentially(go)
Done
(🦌🦌🦌) ⊕ (🧝🧝🧝)]
go
Done

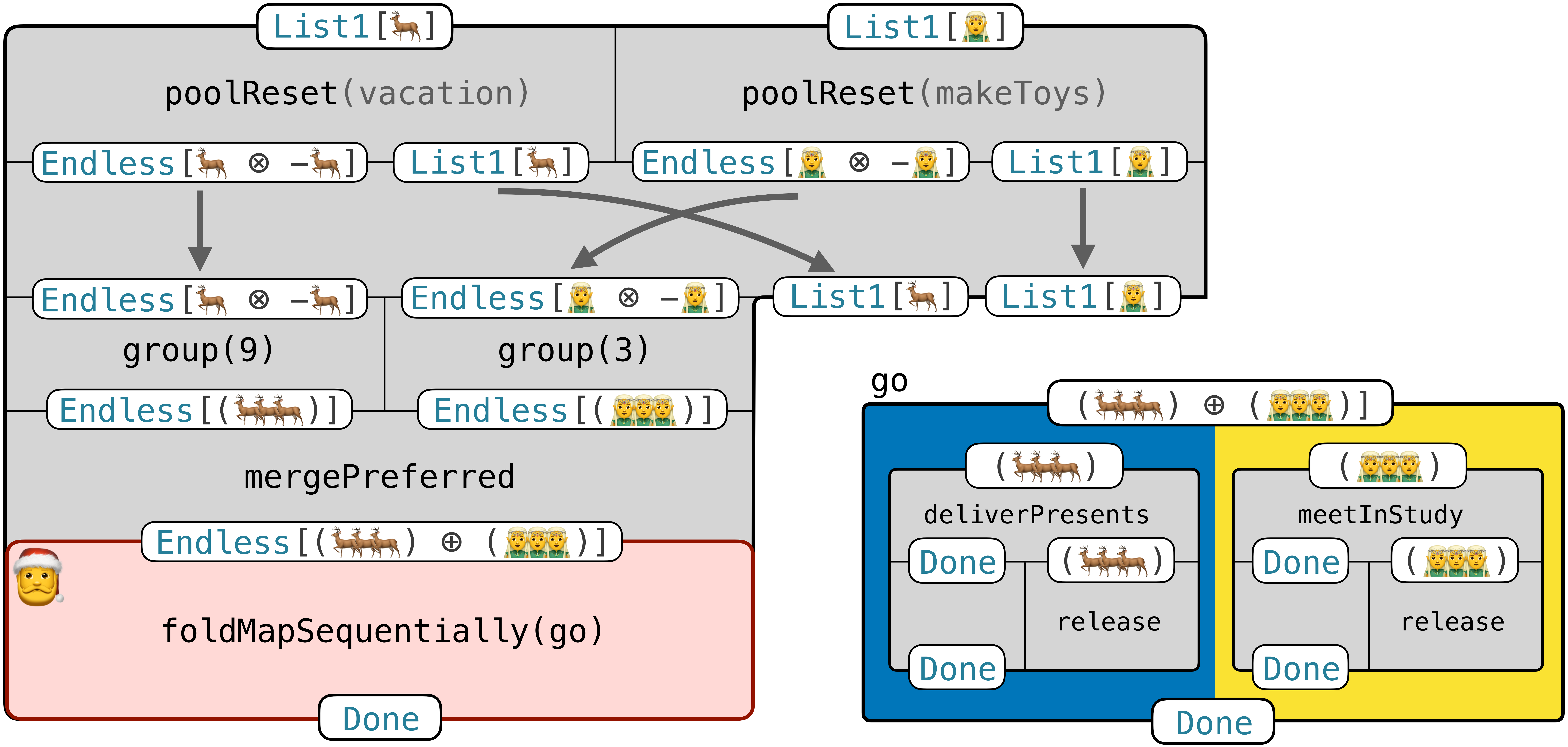


List1[🦌]
List1[🧝]
poolReset(vacation)
poolReset(makeToys)
Endless[🦌 ⊗ –🦌]
List1[🦌]
Endless[🧝 ⊗ –🧝]
List1[🧝]
Endless[🦌 ⊗ –🦌]
Endless[🧝 ⊗ –🧝]
List1[🦌]
List1[🧝]
group(9)
group(3)
Endless[(🦌🦌🦌)]
Endless[(🧝🧝🧝)]
mergePreferred
Endless[(🦌🦌🦌) ⊕ (🧝🧝🧝)]
foldMapSequentially(go)
Done
go
(🦌🦌🦌) ⊕ (🧝🧝🧝)]
(🦌🦌🦌)
deliverPresents
Done
(🦌🦌🦌)
release
Done
(🧝🧝🧝)
meetInStudy
Done
(🧝🧝🧝)
release
Done
Done



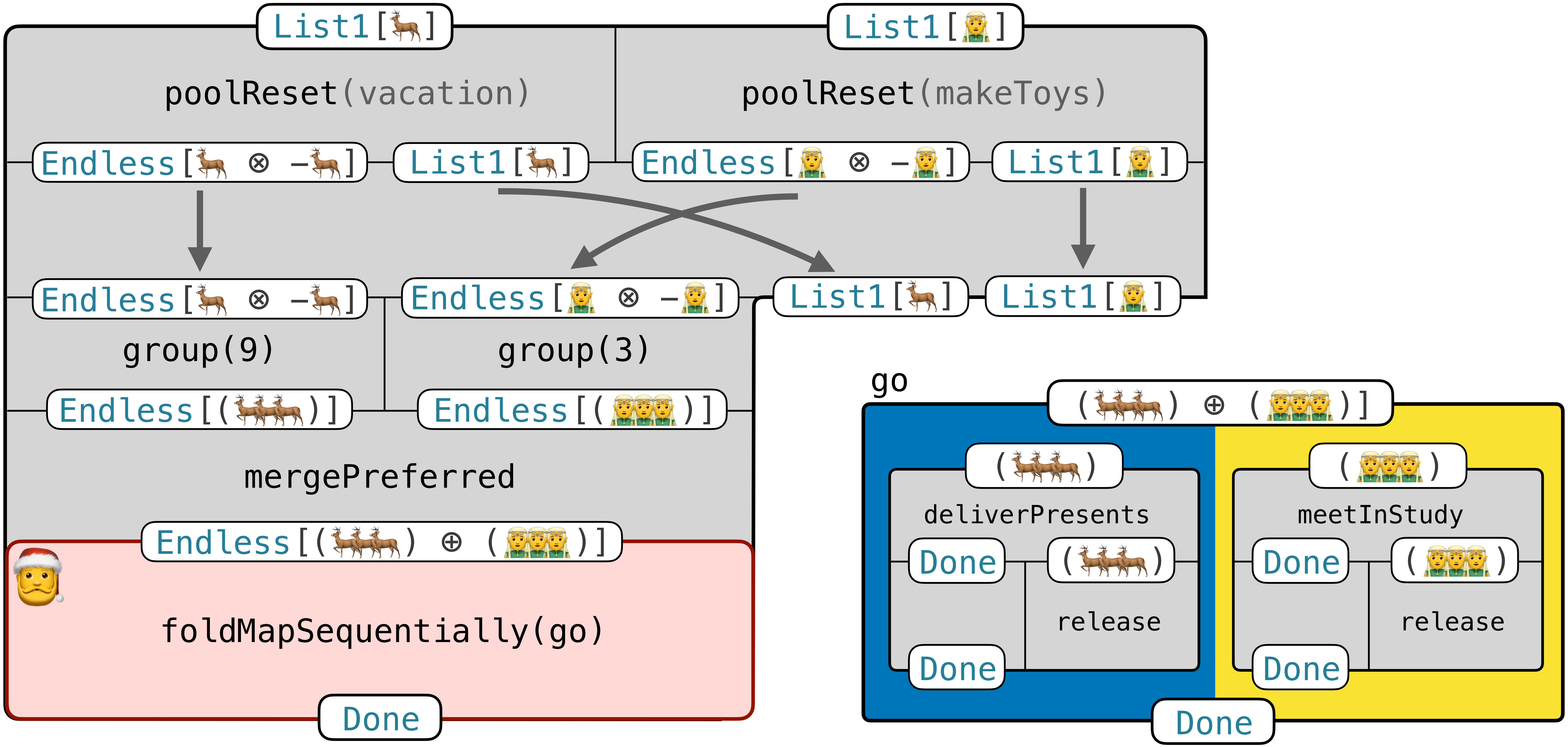
List1[🦌]
List1[🧝]
poolReset(vacation)
poolReset(makeToys)
Endless[🦌 ⊗ –🦌]
List1[🦌]
Endless[🧝 ⊗ –🧝]
List1[🧝]
Endless[🦌 ⊗ –🦌]
Endless[🧝 ⊗ –🧝]
List1[🦌]
List1[🧝]
group(9)
group(3)
Endless[(🦌🦌🦌)]
Endless[(🧝🧝🧝)]
mergePreferred
Endless[(🦌🦌🦌) ⊕ (🧝🧝🧝)]
foldMapSequentially(go)
Done
go
(🦌🦌🦌) ⊕ (🧝🧝🧝)]
(🦌🦌🦌)
deliverPresents
Done
(🦌🦌🦌)
release
Done
(🧝🧝🧝)
meetInStudy
Done
(🧝🧝🧝)
release
Done
Done


No threads ✔




No threads ✔

No side-effects ✔




No threads ✔

No side-effects ✔

Type-driven ✔

# Santa: Recap

| | |
|---|---|
| concurrent operation of 🦌's and 🧝's | implicit |
| non-deterministic order of return | insert into a sorted list |
| group forming | pull *k* elements from a sorted stream |
| priority of 🦌🦌🦌 | `mergePreferred`<br>(with nested `races`) |
| mutual exclusion<br>of delivering 🎁 and studying | `foldMapSequentially(f)`<br>(critical section defined by `f`) |

# Clash of Paradigms

- ⚡ concurrency seamless, **sequencing effortful**
  - ⬆️ need for explicit sequencing sometimes uncovers missing causal link
- ⚡ obligation to **consume everything** can be annoying
  - ⬆️ prevents many resource leaks
- ⚡ **explicit** case analysis of **non-determinism**
  - ⬆️ easier to check correctness

# Conclusion

# Conclusion

- possible to **compose** concurrent programs **like** pure **functions**

# Conclusion

- possible to **compose** concurrent programs **like** pure **functions**
- **type-driven** development applicable to **concurrency**

# Conclusion

- possible to **compose** concurrent programs **like** pure **functions**
- **type-driven** development applicable to **concurrency**

## It’s time to

# Conclusion

- possible to **compose** concurrent programs **like** pure **functions**
- **type-driven** development applicable to **concurrency**

## It’s time to

- **liberate concurrent** programming **from** the **sequential** paradigm of threads

# Conclusion

- possible to **compose** concurrent programs **like** pure **functions**
- **type-driven** development applicable to **concurrency**

## It's time to

- **liberate concurrent** programming **from** the **sequential** paradigm of threads
- **liberate functional** concurrency **from** reliance on **side effects**

# Conclusion

- possible to **compose** concurrent programs **like** pure **functions**
- **type-driven** development applicable to **concurrency**

## It's time to

- **liberate concurrent** programming **from** the **sequential** paradigm of threads
- **liberate functional** concurrency **from** reliance on **side effects**

## Let's make it happen!

github.com/TomasMikula/libretto/

# Questions?

github.com/TomasMikula/libretto/