Custom Stream Operators Made Safe And Simple with Libretto

SCALAR

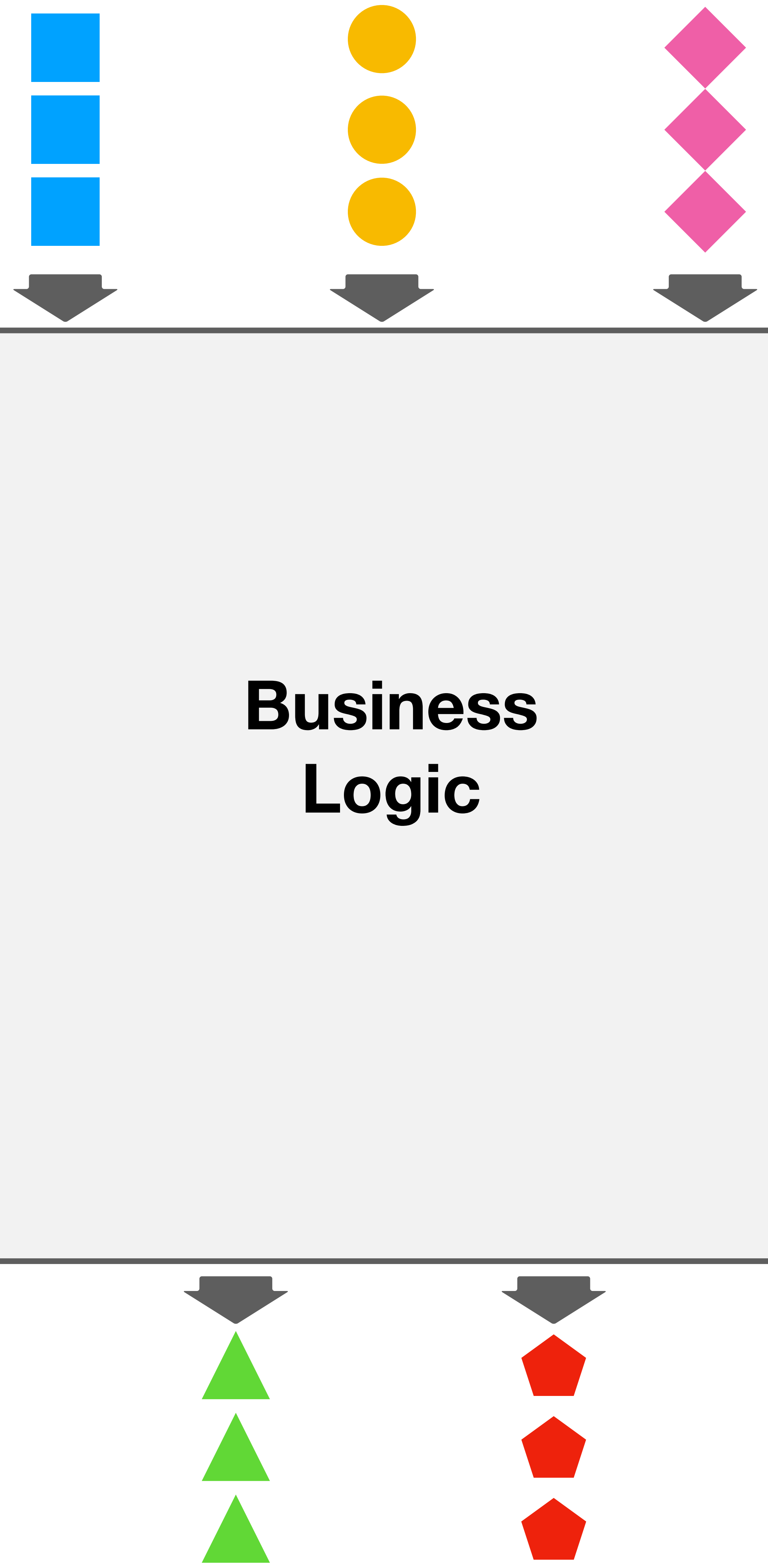
Scala Conference in Central Europe

Tomas Mikula

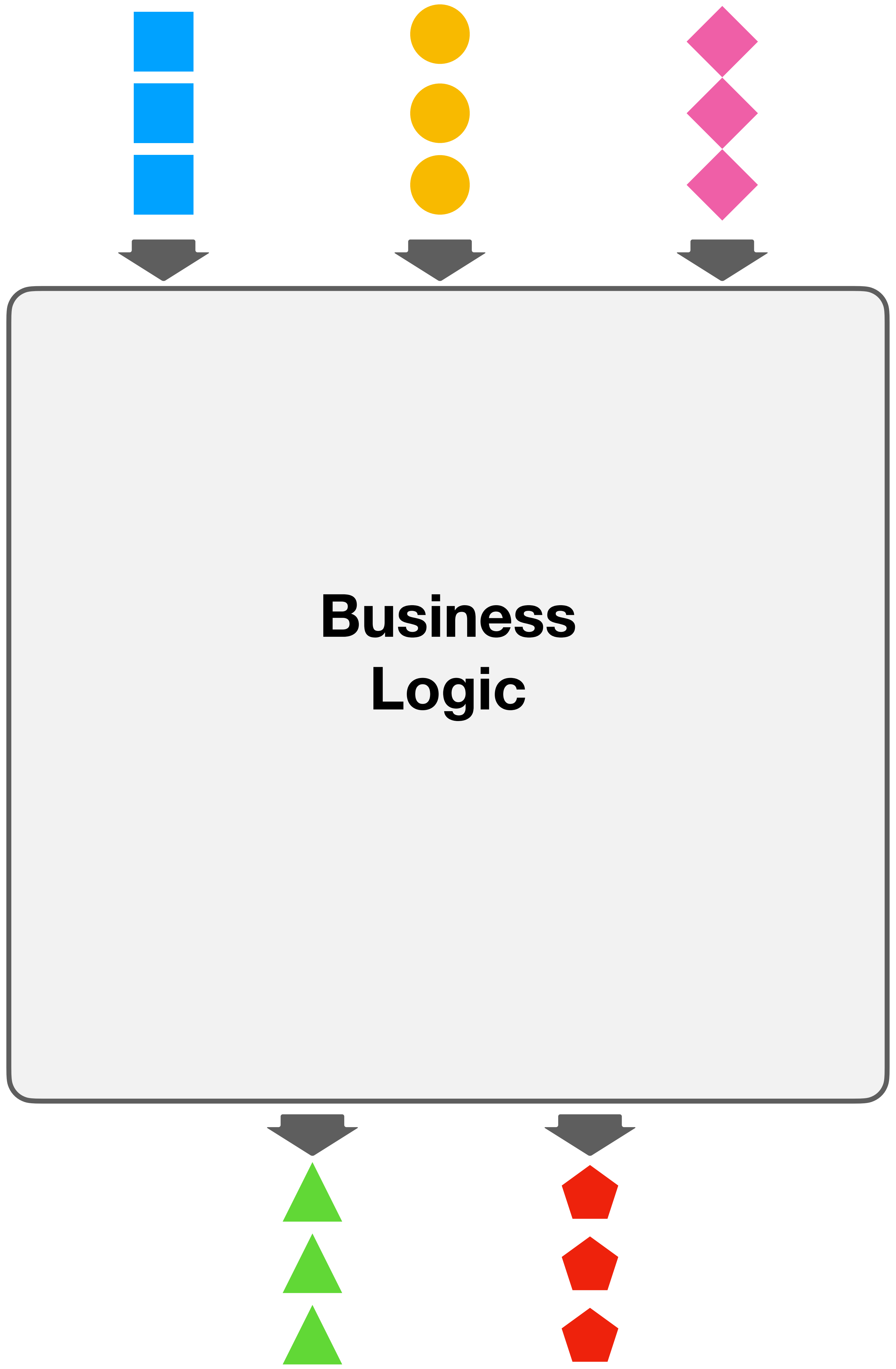
Mar 24, 2023

Custom Stream Operators Made Safe and Simple

Custom Stream Operators Made Safe and Simple



Custom Stream Operators Made Safe and Simple

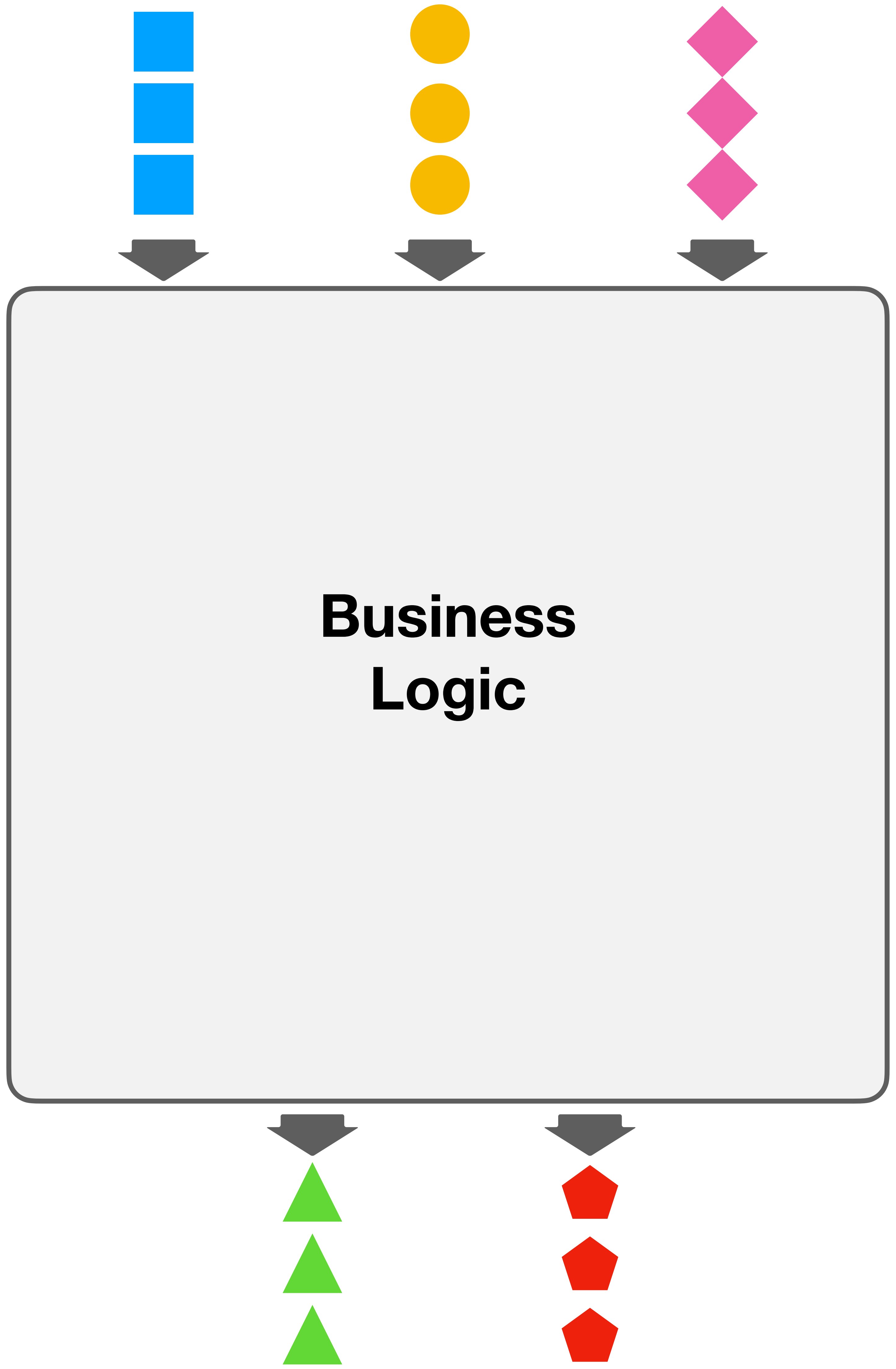


Safe

high rejection rate of wrong programs

(hard to shoot ourselves in the foot)

Custom Stream Operators Made Safe and Simple



Safe

high rejection rate of wrong programs

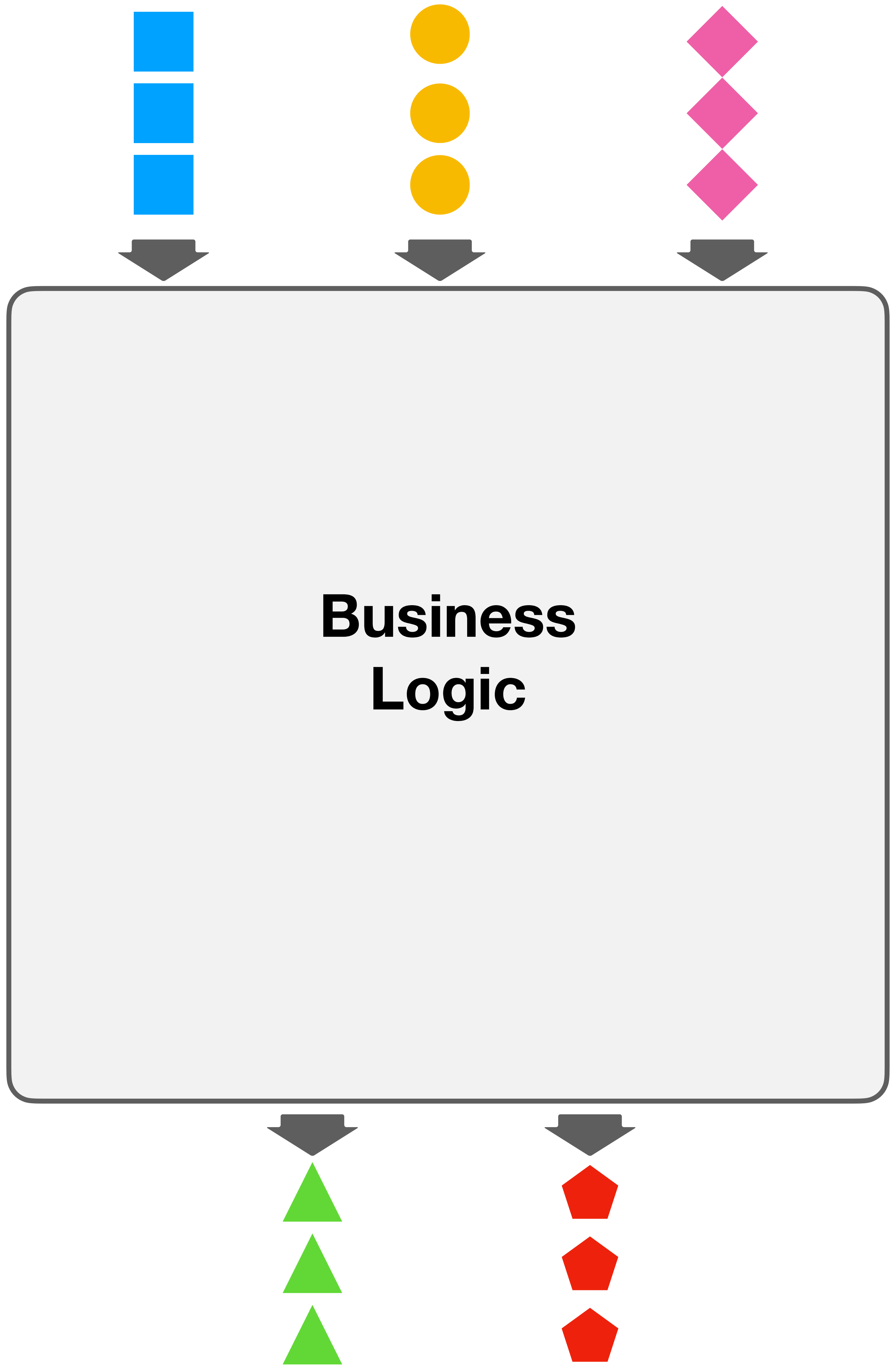
(hard to shoot ourselves in the foot)

Simple

low accidental complexity

(stay focused on business logic)

Custom Stream Operators Made Safe and Simple



Safe

high rejection rate of wrong programs

(hard to shoot ourselves in the foot)

Simple

low accidental complexity

(stay focused on business logic)

Stream

a sequence of *elements* produced and consumed gradually

Stream

a sequence of *elements* produced and consumed gradually

Control Flow	
proactive	reactive (not to be confused with “Reactive Streams”)

Stream

a sequence of *elements* produced and consumed gradually

Control Flow	
proactive	**reactive** (not to be confused with “Reactive Streams”)
Reactive Streams `Publisher` Akka `Source`	

Stream

a sequence of *elements* produced and consumed gradually

Control Flow	
proactive	reactive (not to be confused with “Reactive Streams”)
Reactive Streams `Publisher` Akka `Source`	`fs2.Stream` `zio.stream.ZStream`

Stream

a sequence of *elements* produced and consumed gradually

Control Flow	
proactive	**reactive** (not to be confused with “Reactive Streams”)

Reactive Streams `Publisher`
Akka `Source`
	`fs2.Stream`
	`zio.stream.ZStream`
	`libretto.stream.Source`

Stream

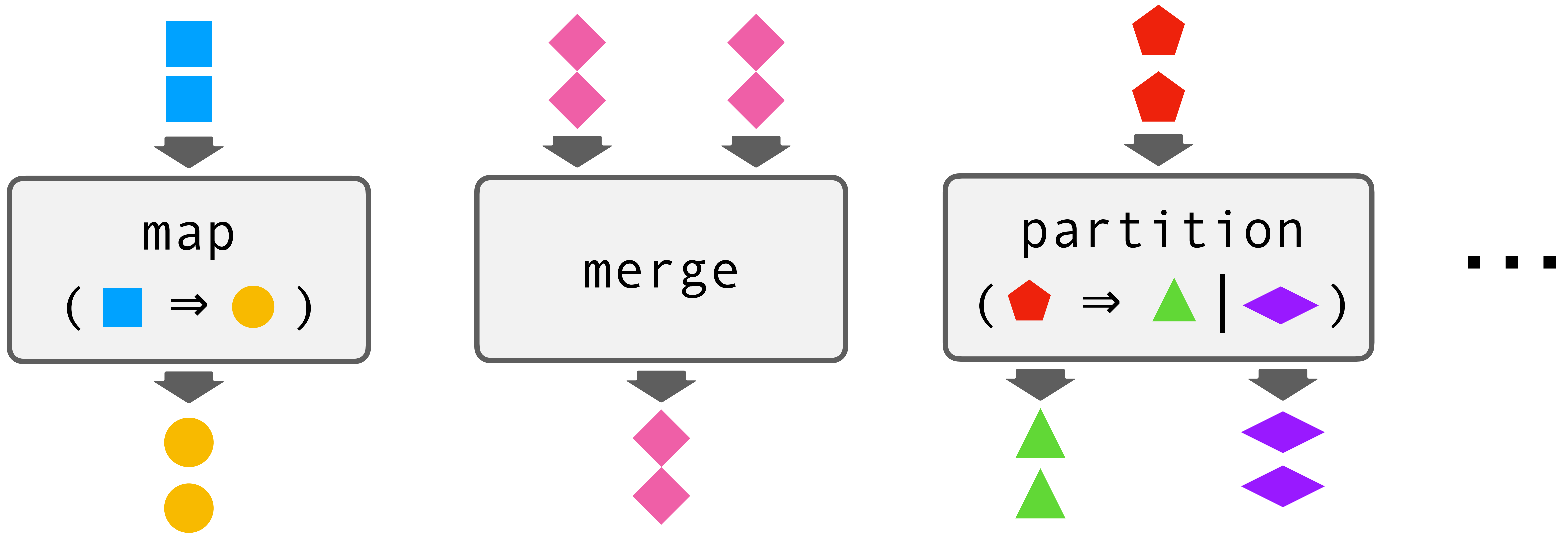
a sequence of *elements* produced and consumed gradually

		Control Flow	
		proactive	reactive (not to be confused with “Reactive Streams”)

Payload Flow	producer	Reactive Streams `Publisher`
		Akka `Source`
			`fs2.Stream`
			`zio.stream.ZStream`
			`libretto.stream.Source`

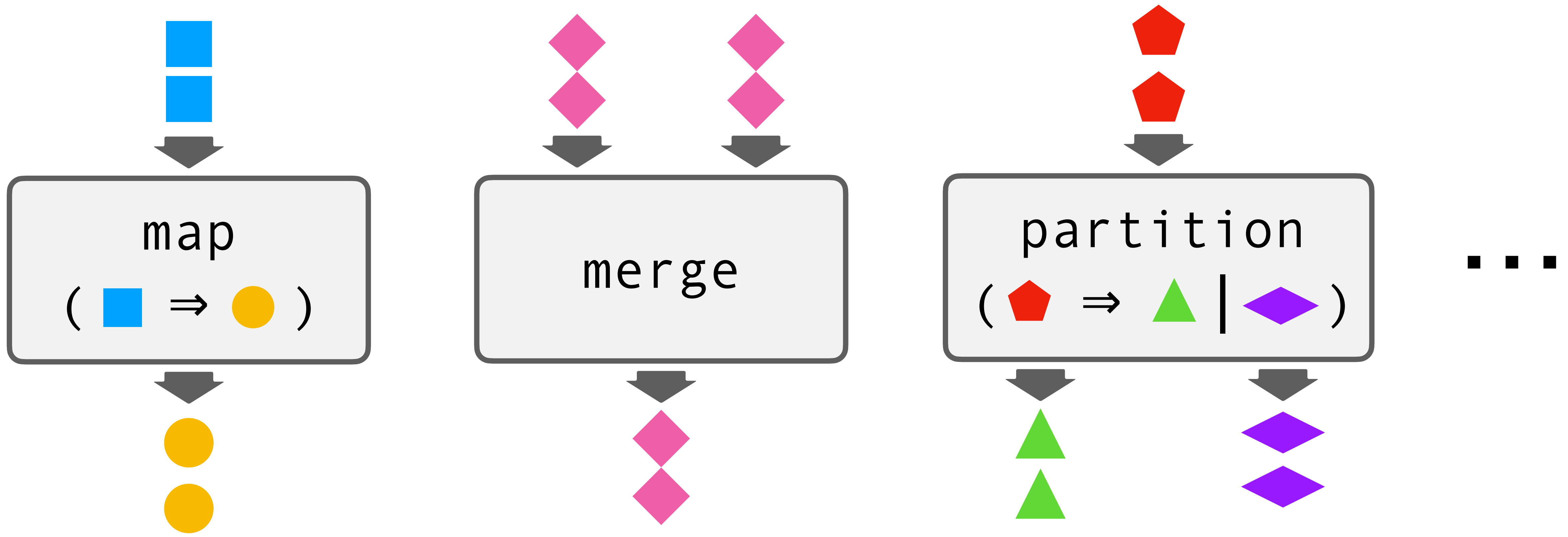
	consumer		

Libraries come with batteries included



- nice to work with
- *“declarative concurrency”*
- can go a long way
- ideally, never need anything custom

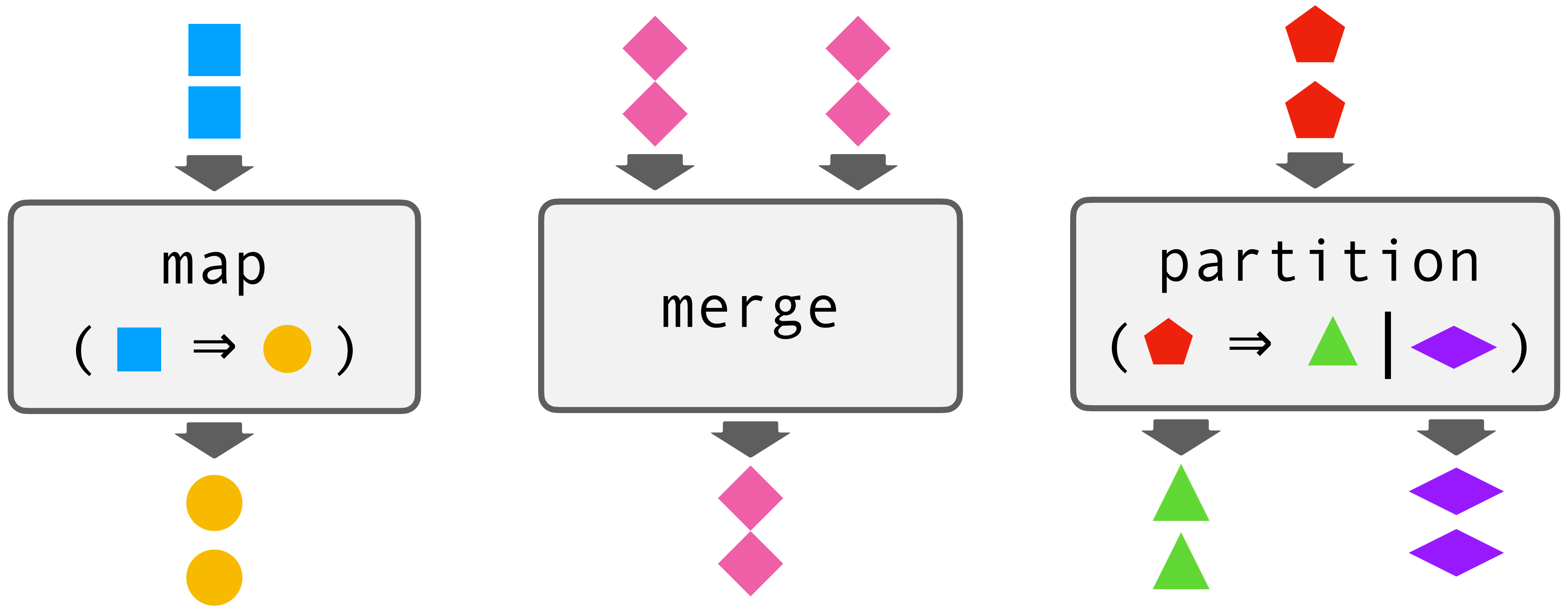
Libraries come with batteries included



- nice to work with
- *“declarative concurrency”*
- can go a long way
- ideally, never need anything custom

but what if we need something *custom?*

Libraries come with batteries included



map (■ ⇒ ●)

merge

partition (⬟ ⇒ ▲ | ◆)

...

- nice to work with
- *“declarative concurrency”*
- can go a long way
- ideally, never need anything custom

but what if we need something *custom?*

promises

queues

mutable variables

interruptions

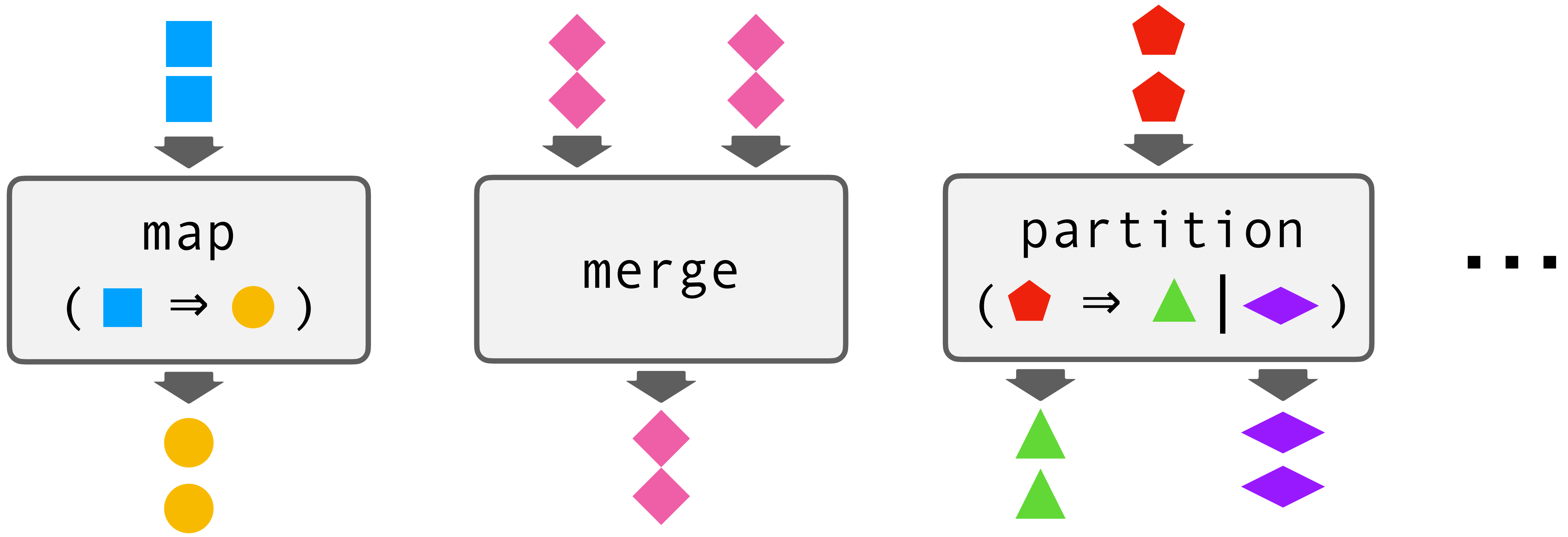
fibers

illegal state

scopes

locks

Libraries come with batteries included

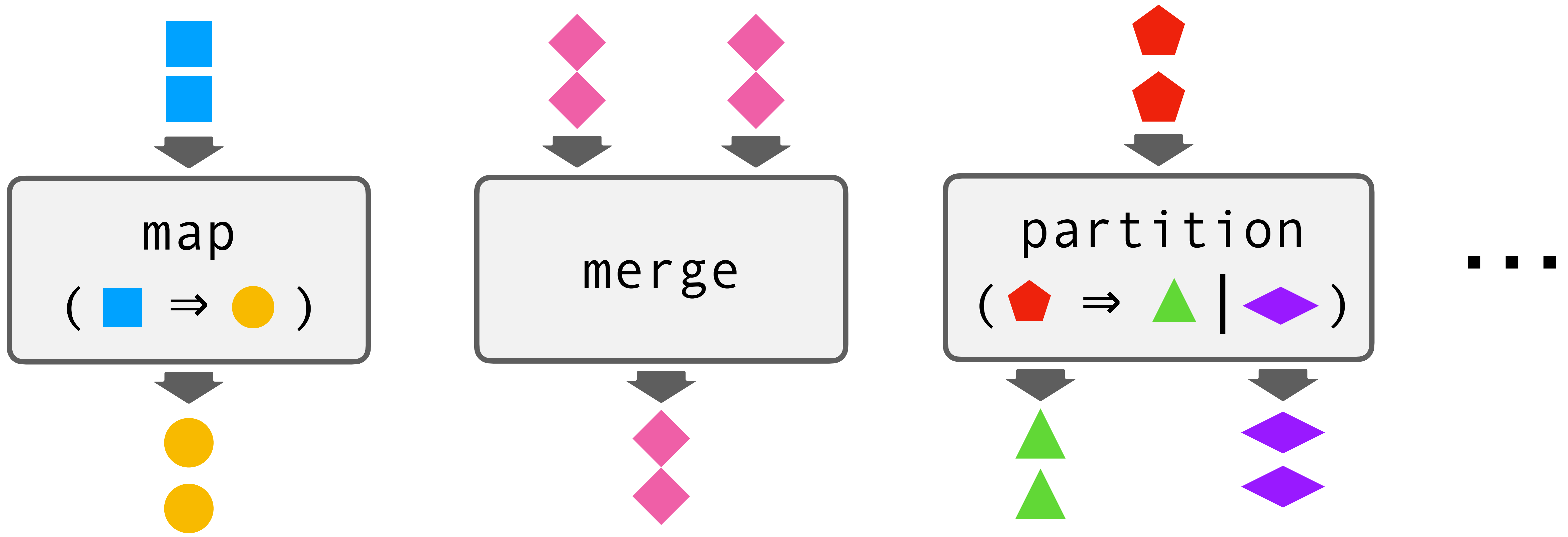


- nice to work with
- *“declarative concurrency”*
- can go a long way
- ideally, never need anything custom

but what if we need something ***custom?***

promises i e s
queues
b s
n s
u
mutable variables
interruptions
i
illegal state
scopes
o
fibers
l
g
c
locks

Libraries come with batteries included



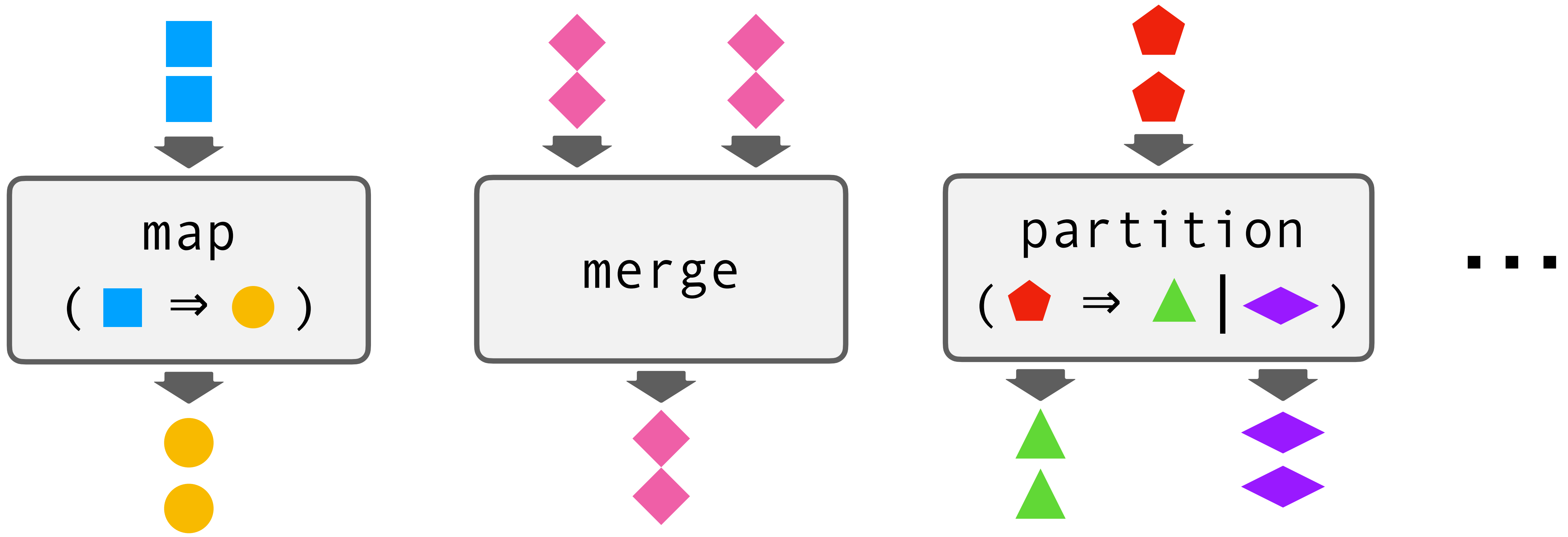
- nice to work with
- *"declarative concurrency"*
- can go a long way
- ideally, never need anything custom

but what if we need something ***custom?***

- All promises completed? Exactly once?

promises b s i queues e s u mutable variables n s interruptions i illegal state scopes fibers l o g locks c

Libraries come with batteries included



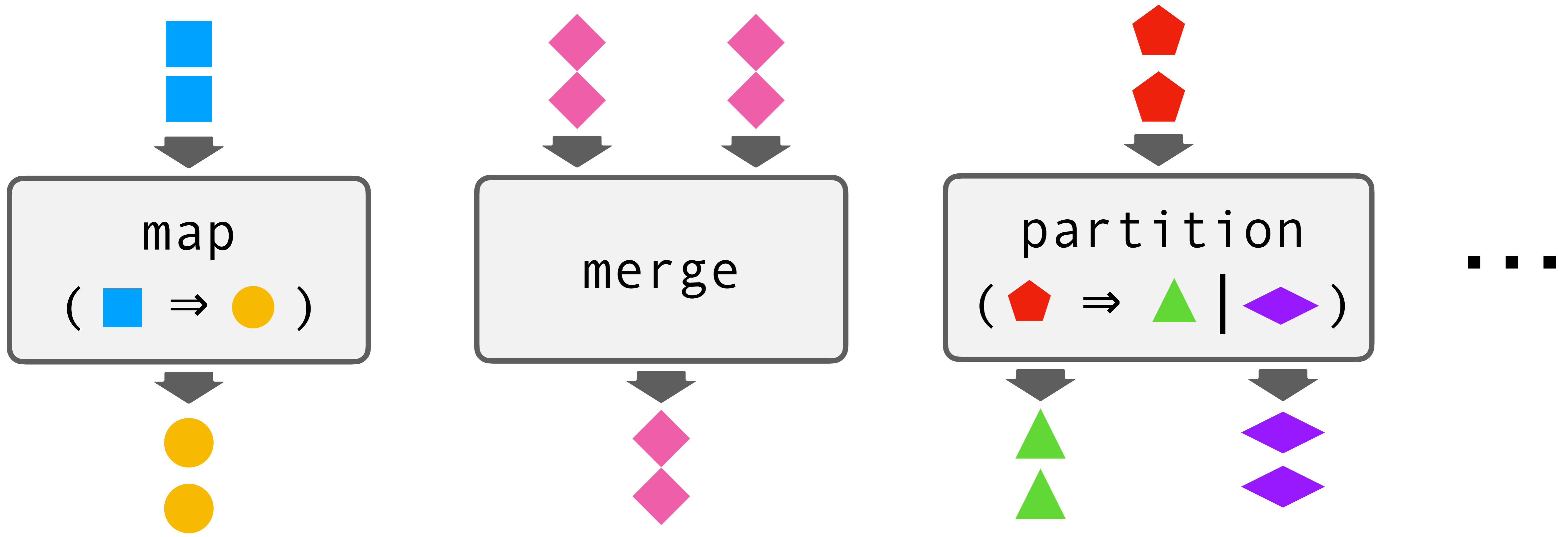
- nice to work with
- *"declarative concurrency"*
- can go a long way
- ideally, never need anything custom

but what if we need something *custom?*

promises e s
i queues
b s
n s
u mutable variables interruptions
i
o illegal state scopes
fibers
l g c
locks

- All promises completed? Exactly once?
- Are we not losing elements?

Libraries come with batteries included



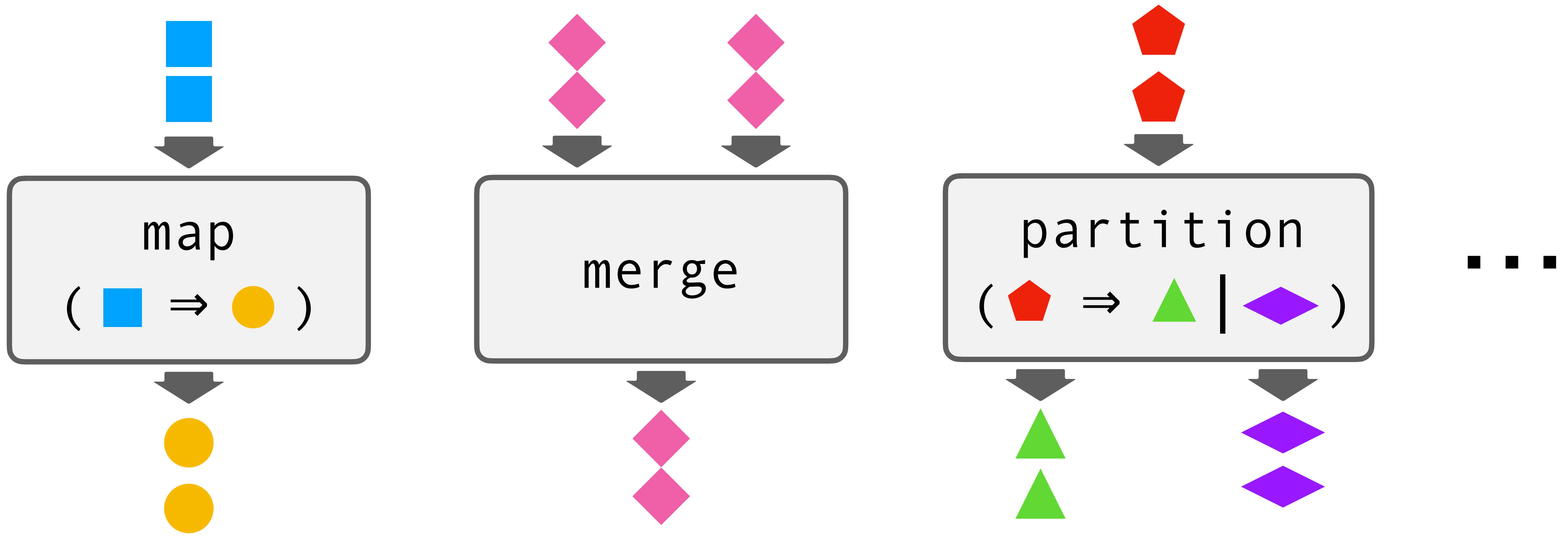
- nice to work with
- *“declarative concurrency”*
- can go a long way
- ideally, never need anything custom

but what if we need something *custom?*

promises i e s
b s queues
u n s
mutable interruptions
variables
i
fibers o illegal scopes
l g state c
locks

- All promises completed? Exactly once?
- Are we not losing elements?
- Is this state really unreachable?

Libraries come with batteries included



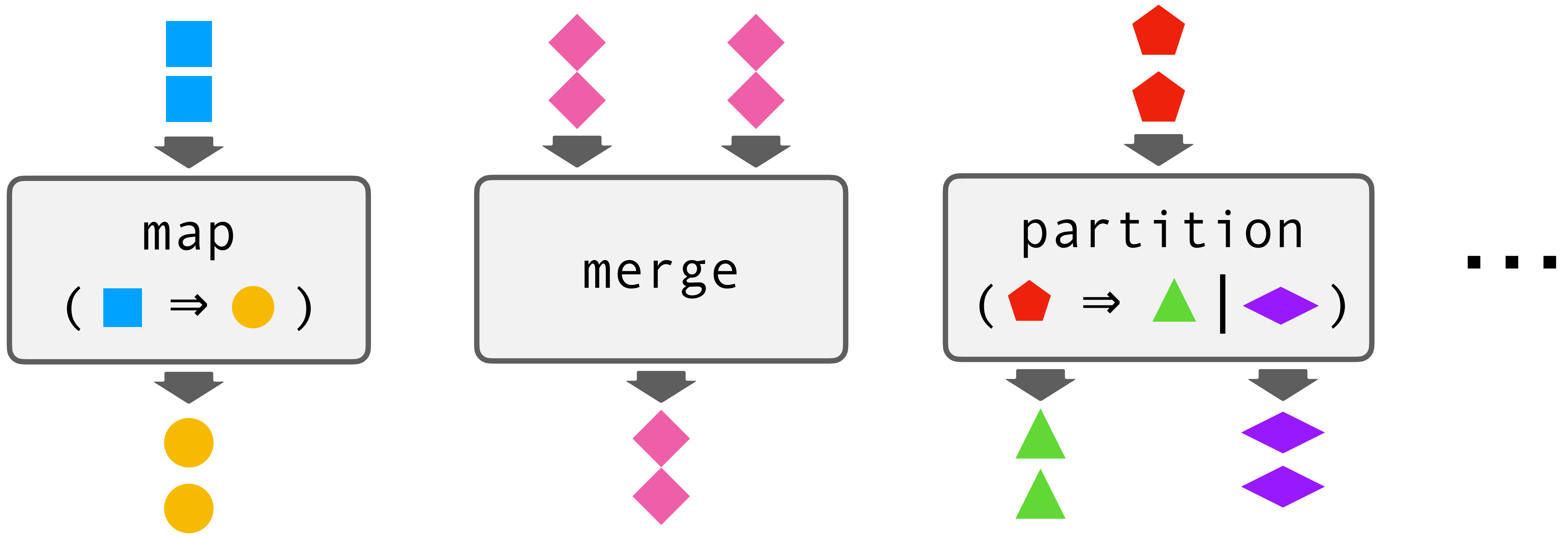
- nice to work with
- *“declarative concurrency”*
- can go a long way
- ideally, never need anything custom

but what if we need something *custom?*

promises e s
i
queues
b s
s
n
u
mutable interruptions
variables
i
o illegal scopes
fibers
g state
l
c
locks

- All promises completed? Exactly once?
- Are we not losing elements?
- Is this state really unreachable?
- Are we not pulling from a closed queue?

Libraries come with batteries included



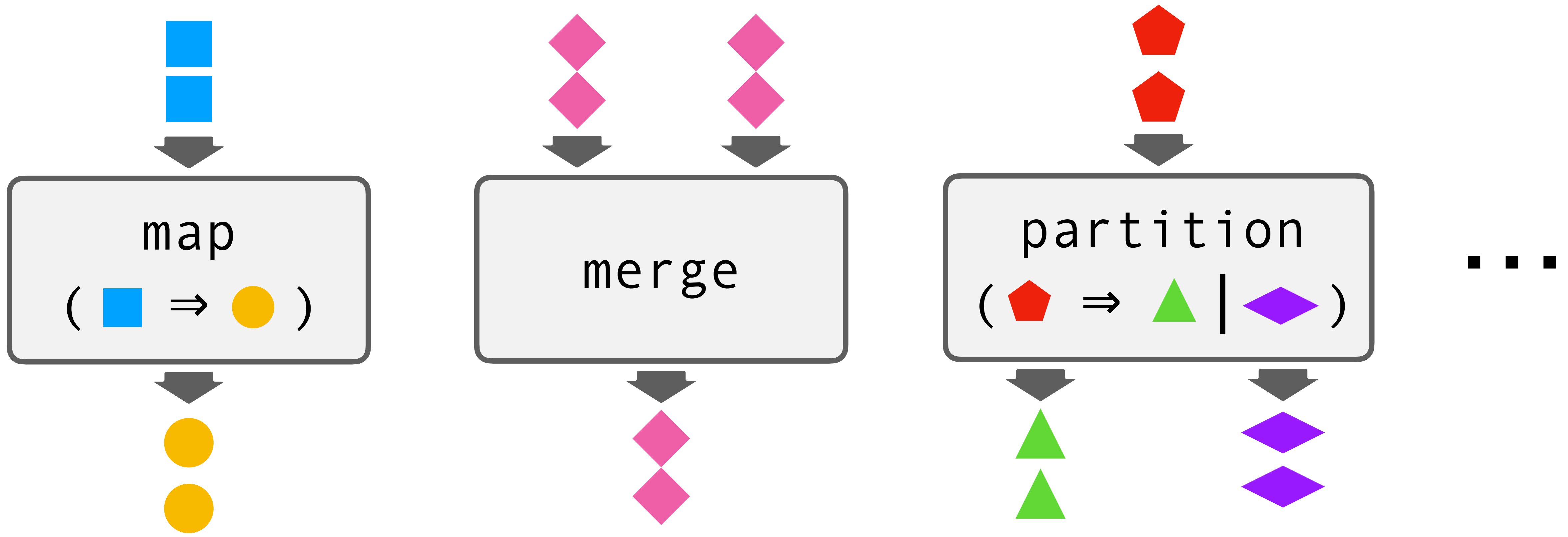
- nice to work with
- *“declarative concurrency”*
- can go a long way
- ideally, never need anything custom

but what if we need something *custom?*

promises e s
i queues
b s
s
n
u mutable variables interruptions
i
illegal state scopes o
fibers g
l c
locks

- All promises completed? Exactly once?
- Are we not losing elements?
- Is this state really unreachable?
- Are we not pulling from a closed queue?
- Are var updates noticed by the other side?

Libraries come with batteries included



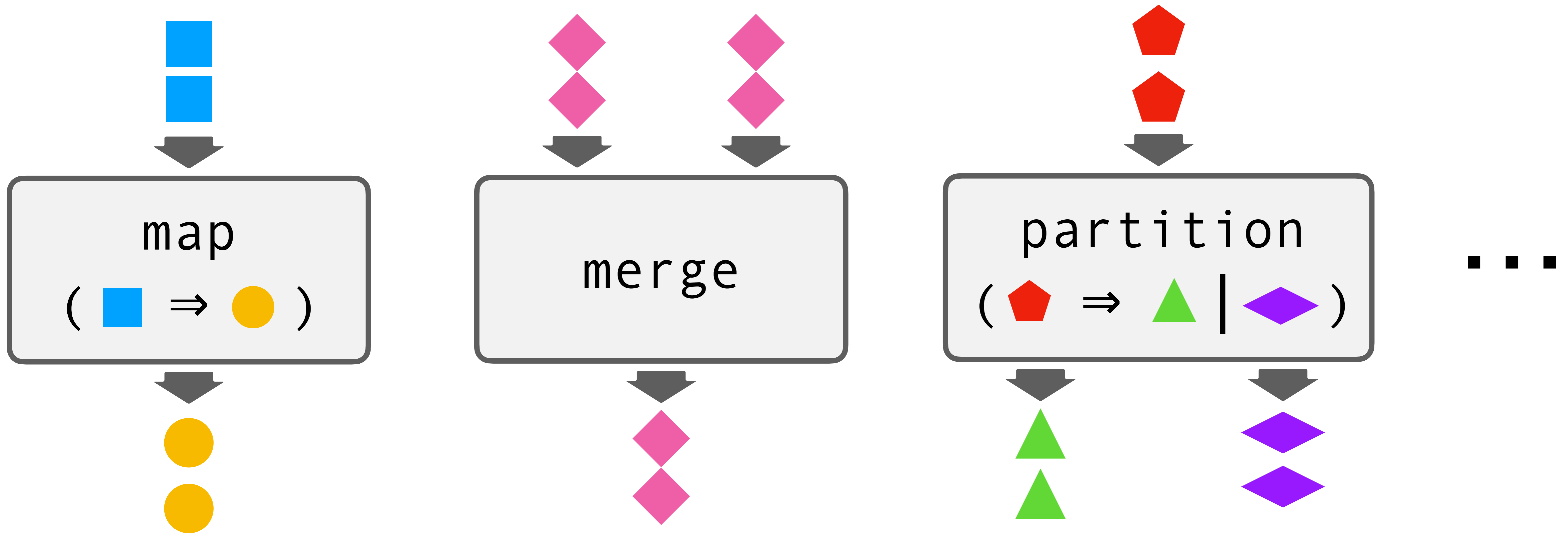
- nice to work with
- *"declarative concurrency"*
- can go a long way
- ideally, never need anything custom

but what if we need something *custom?*

promises i e s
b s queues
u mutable variables n s interruptions
i
fibers o illegal state scopes
l g c
locks

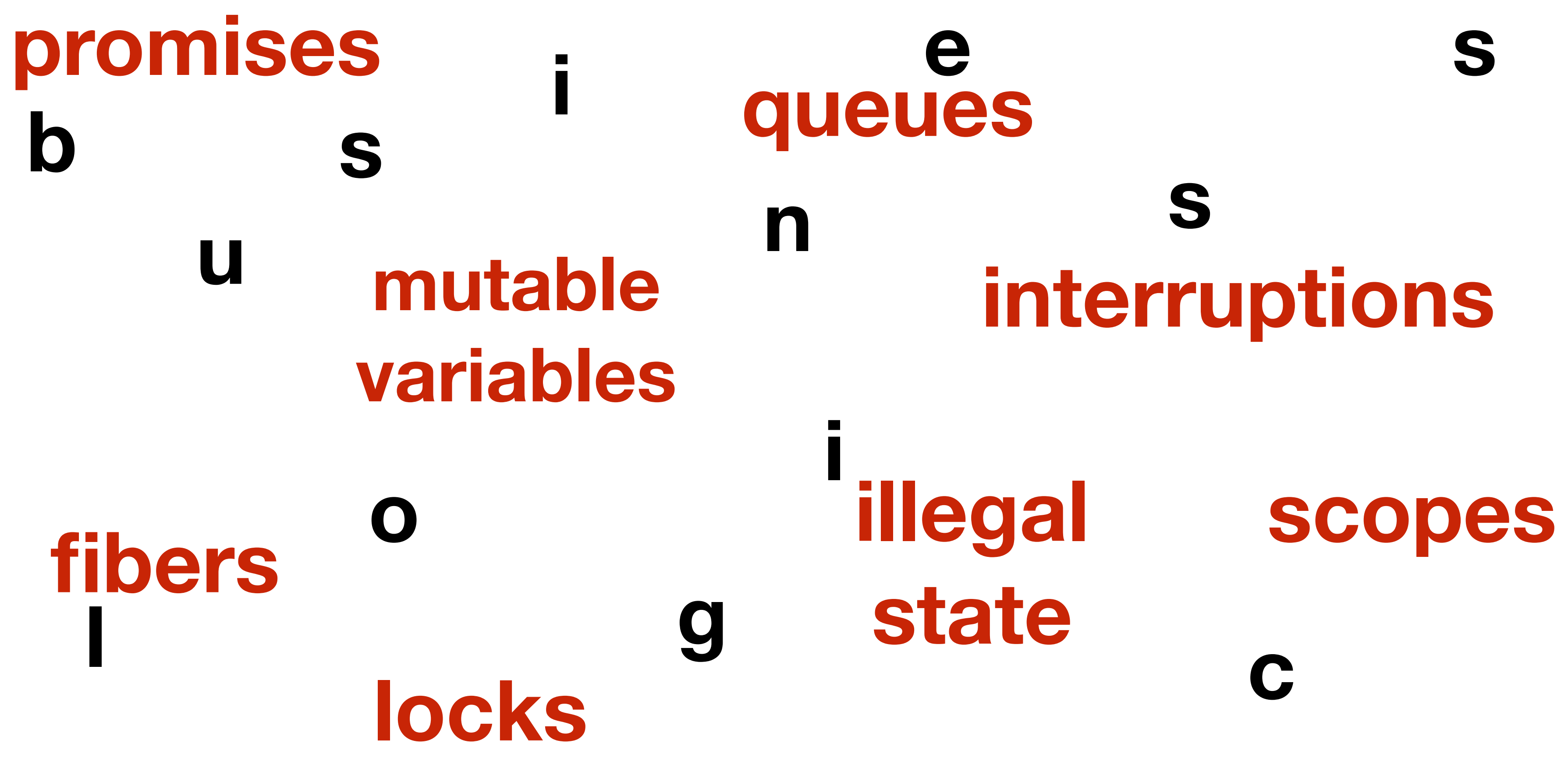
- All promises completed? Exactly once?
- Are we not losing elements?
- Is this state really unreachable?
- Are we not pulling from a closed queue?
- Are var updates noticed by the other side?
- What if the fiber gets cancelled?

Libraries come with batteries included



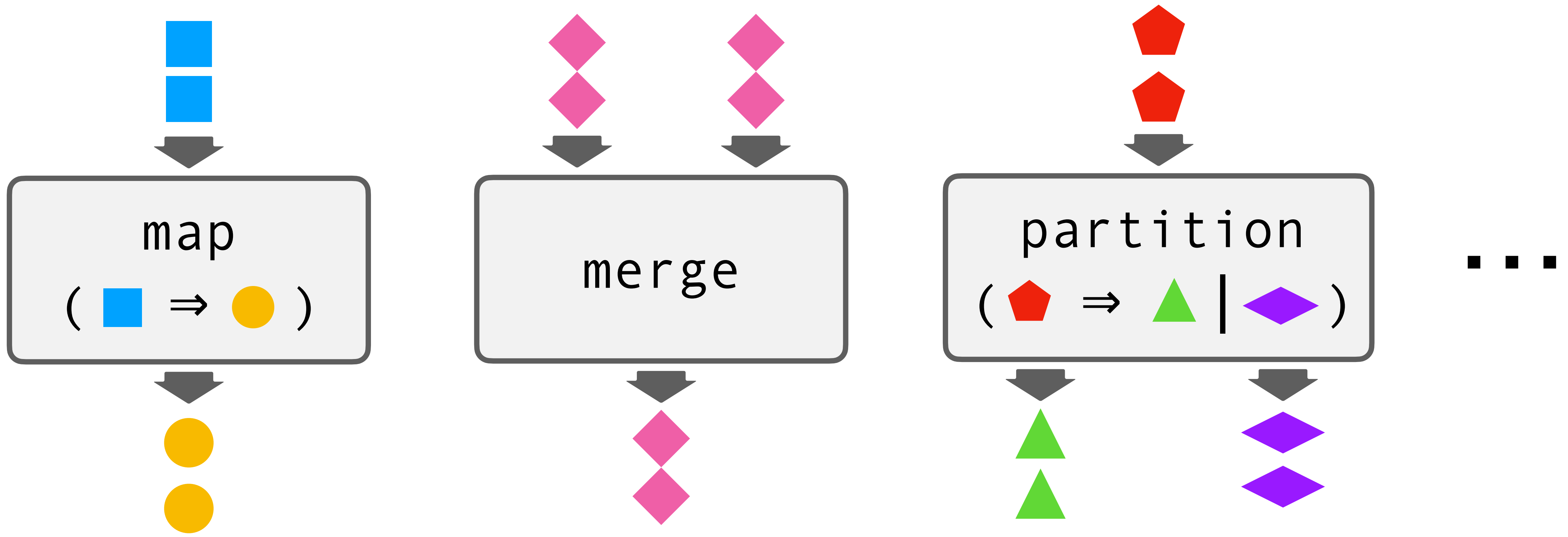
- nice to work with
- *“declarative concurrency”*
- can go a long way
- ideally, never need anything custom

but what if we need something *custom?*



- All promises completed? Exactly once?
- Are we not losing elements?
- Is this state really unreachable?
- Are we not pulling from a closed queue?
- Are var updates noticed by the other side?
- What if the fiber gets cancelled?
- Is this resource still alive?

Libraries come with batteries included



- nice to work with
- *"declarative concurrency"*
- can go a long way
- ideally, never need anything custom

but what if we need something ***custom?***

promises e s

b s i queues

u mutable variables n s interruptions

i illegal state scopes

fibers o

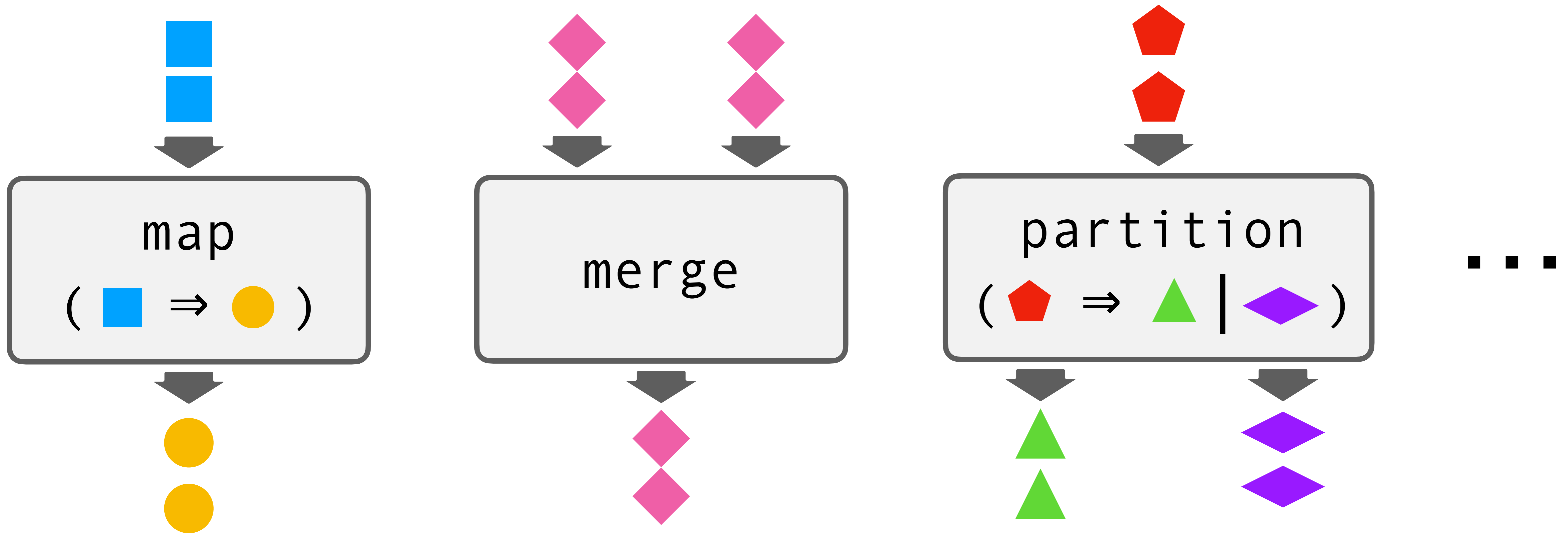
l g c

locks

- All promises completed? Exactly once?
- Are we not losing elements?
- Is this state really unreachable?
- Are we not pulling from a closed queue?
- Are var updates noticed by the other side?
- What if the fiber gets cancelled?
- Is this resource still alive?

...

Libraries come with batteries included



- nice to work with
- *“declarative concurrency”*
- can go a long way
- ideally, never need anything custom

but what if we need something ***custom?***

promises i e queues s

b s

s

n

u mutable variables interruptions

i

o illegal state scopes

fibers g

l c

locks

- All promises completed? Exactly once?
- Are we not losing elements?
- Is this state really unreachable?
- Are we not pulling from a closed queue?
- Are var updates noticed by the other side?
- What if the fiber gets cancelled?
- Is this resource still alive?

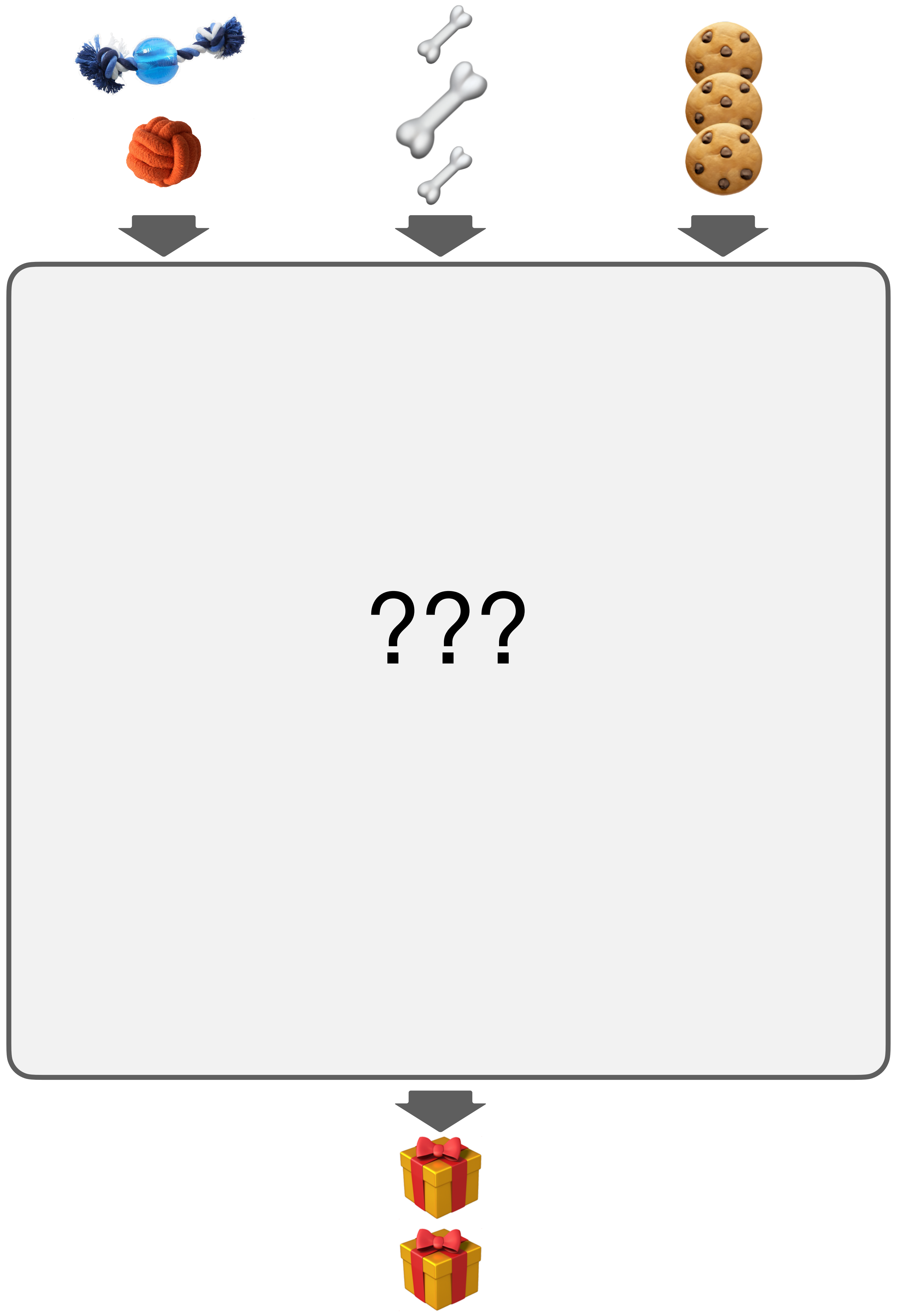
...

neither **Safe** nor **Simple**

The Libretto Way

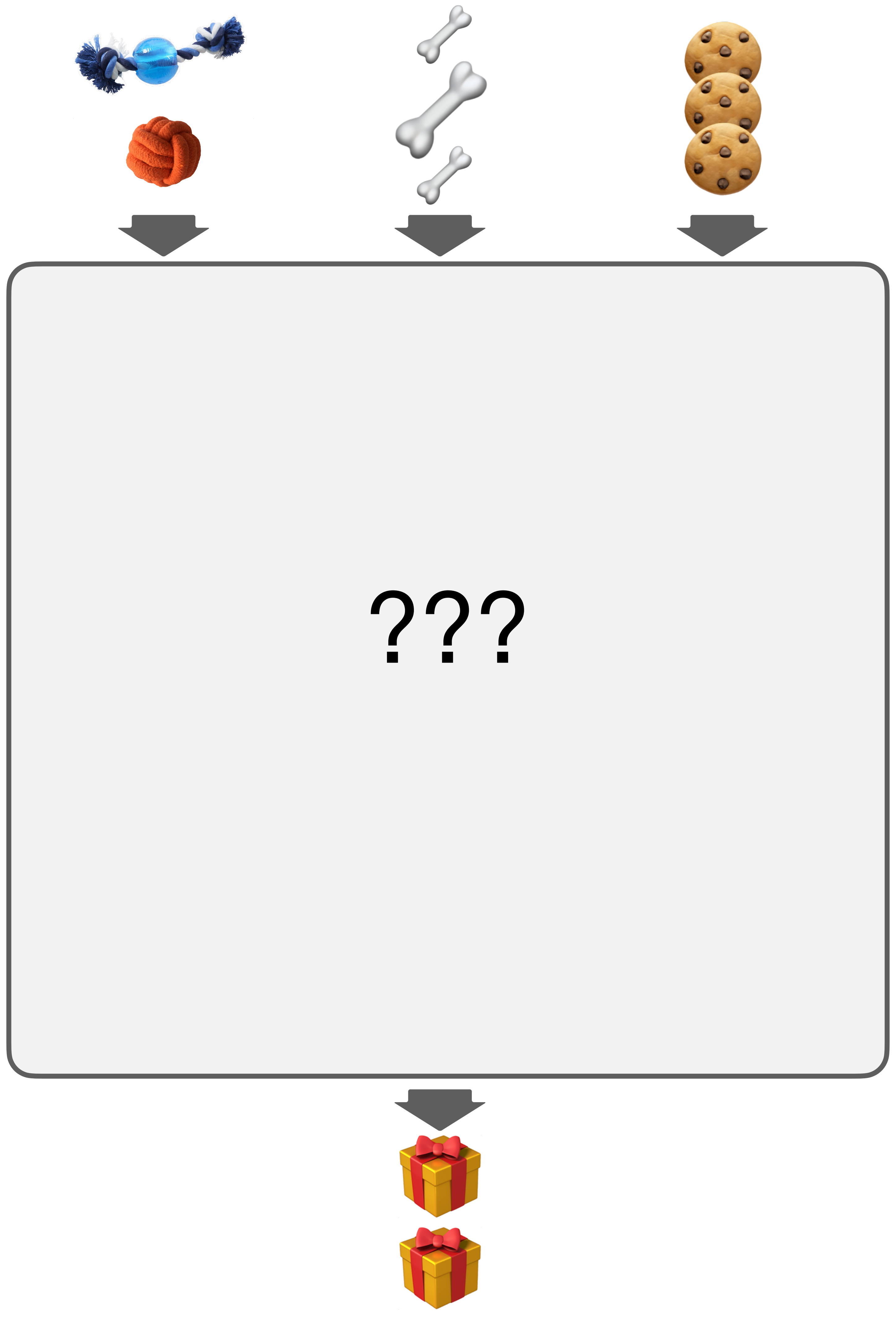
by example

Packaging Dog Presents



- **In:** toys, bones, biscuits
- **Out:** packages of either
  - 1 toy, 1 *large* bone, *3* biscuits
  - 1 toy, 1 *small* bone, *5* biscuits
- Halt when either:
  - no more downstream demand
  - any upstream runs out of items
- Discard at most 1 toy, 1 bone, 5 biscuits

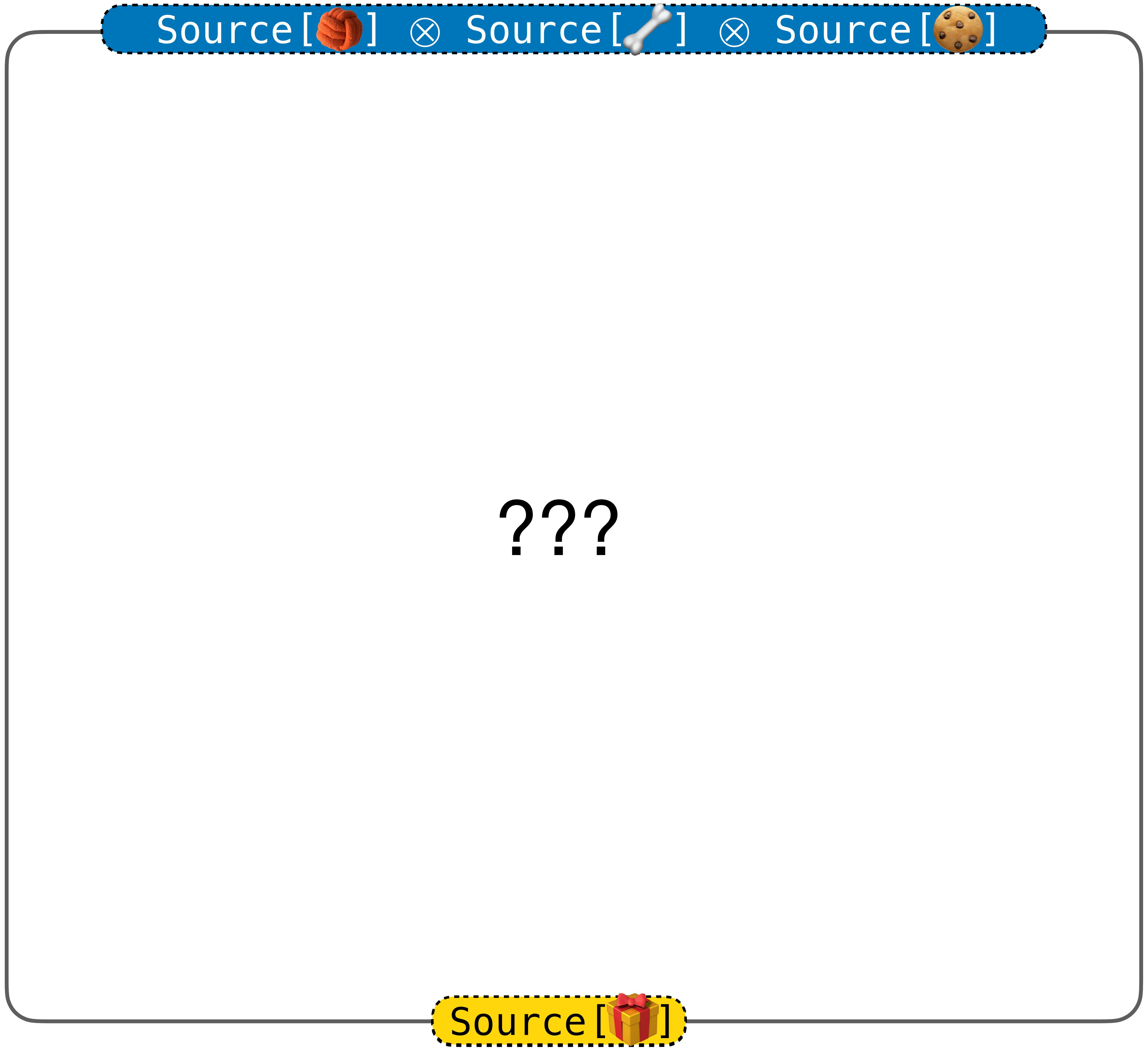
Packaging Dog Presents



- **In:** toys, bones, biscuits
- **Out:** packages of either
  - 1 toy, 1 *large* bone, *3* biscuits
  - 1 toy, 1 *small* bone, *5* biscuits
- Halt when either:
  - no more downstream demand
  - any upstream runs out of items
- Discard at most 1 toy, 1 bone, 5 biscuits

Pulling behavior depends on previously pulled values (size of the pulled bone).

Packaging Dog Presents



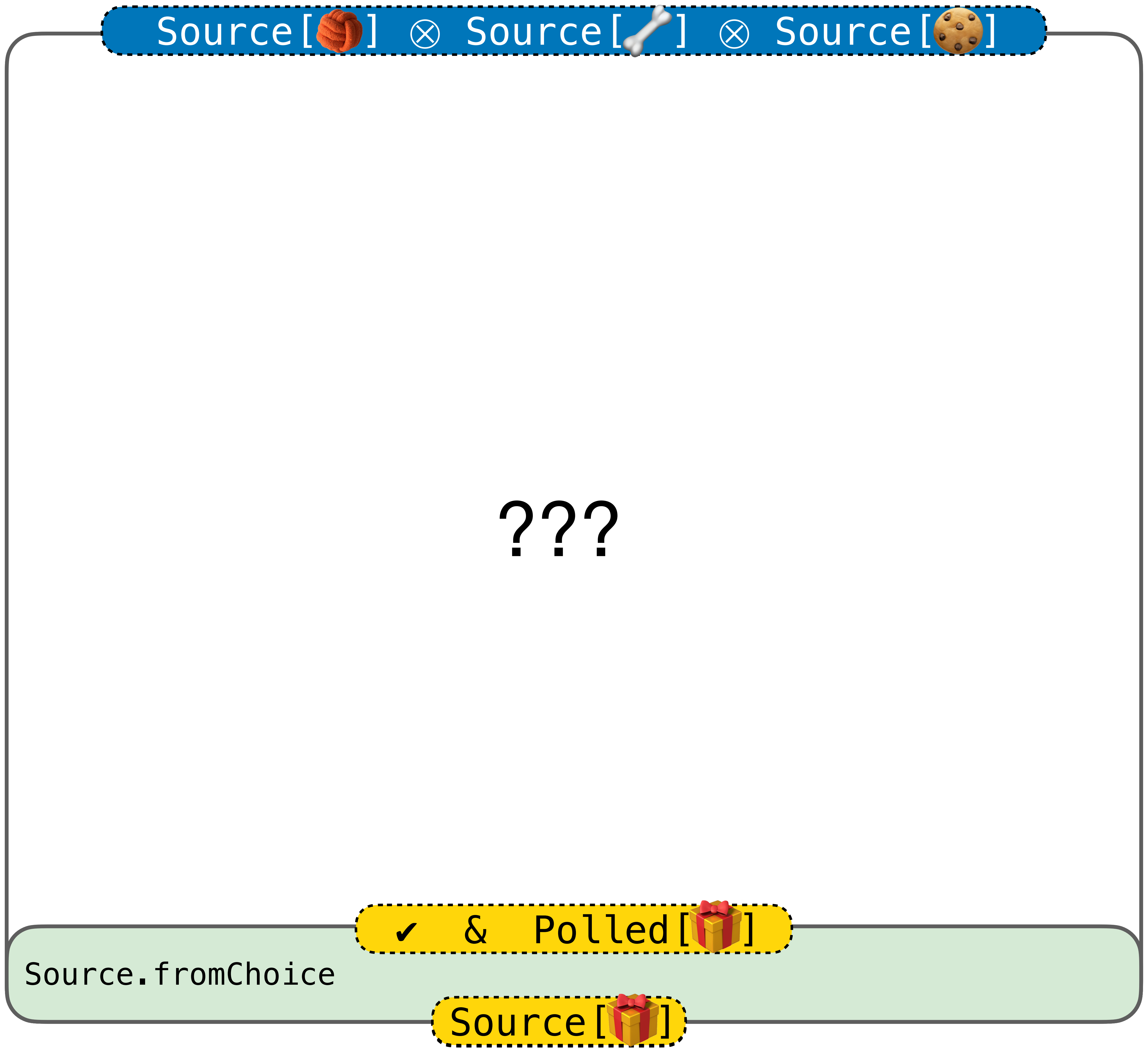
Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

???

Source[🎁]

???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

???

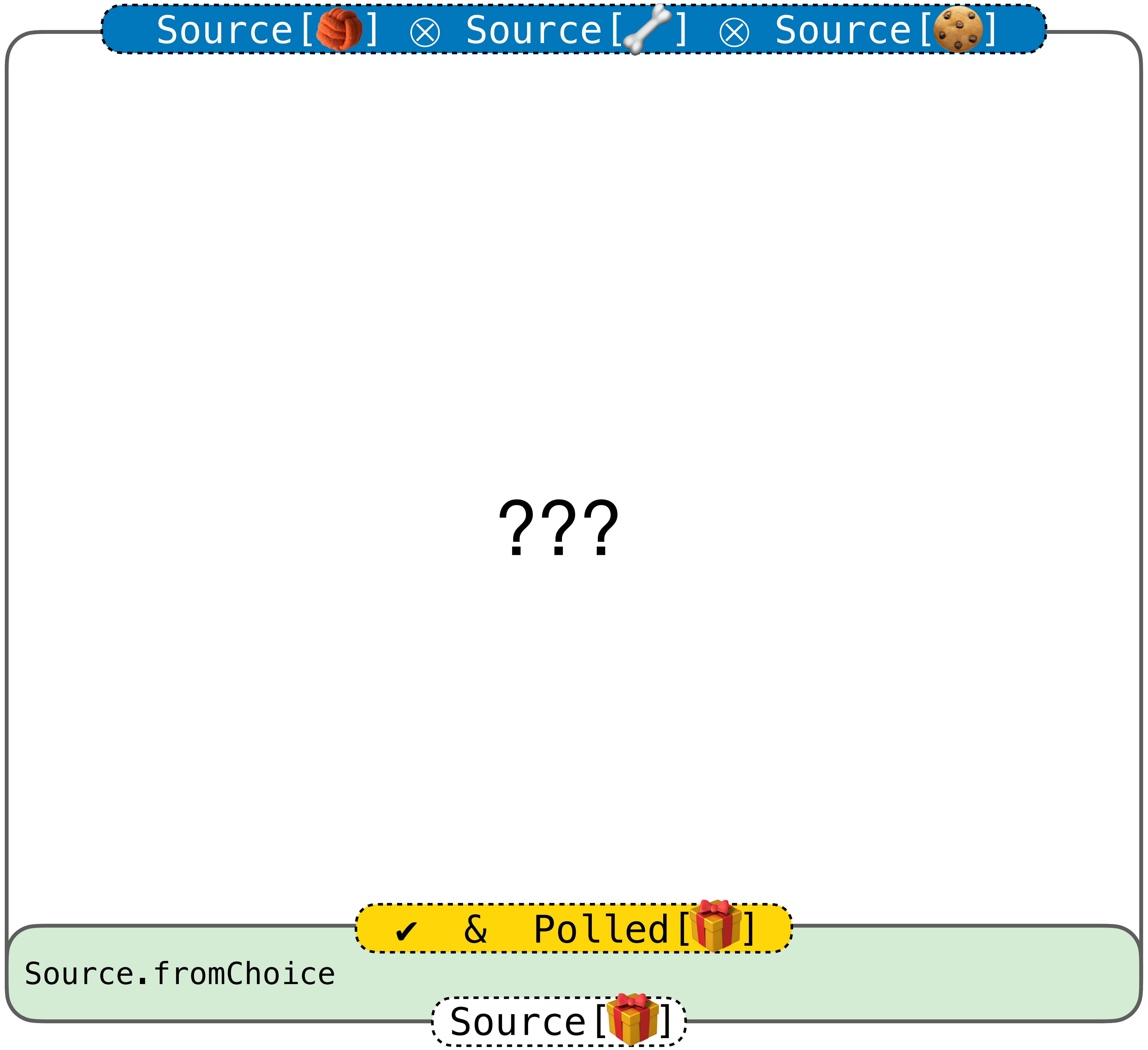
✔ & Polled[🎁]

Source.fromChoice

Source[🎁]

???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair
&	consumer choice
✔	Done signal
`Polled[A]`	requested next elem

Packaging Dog Presents



`Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]`

???

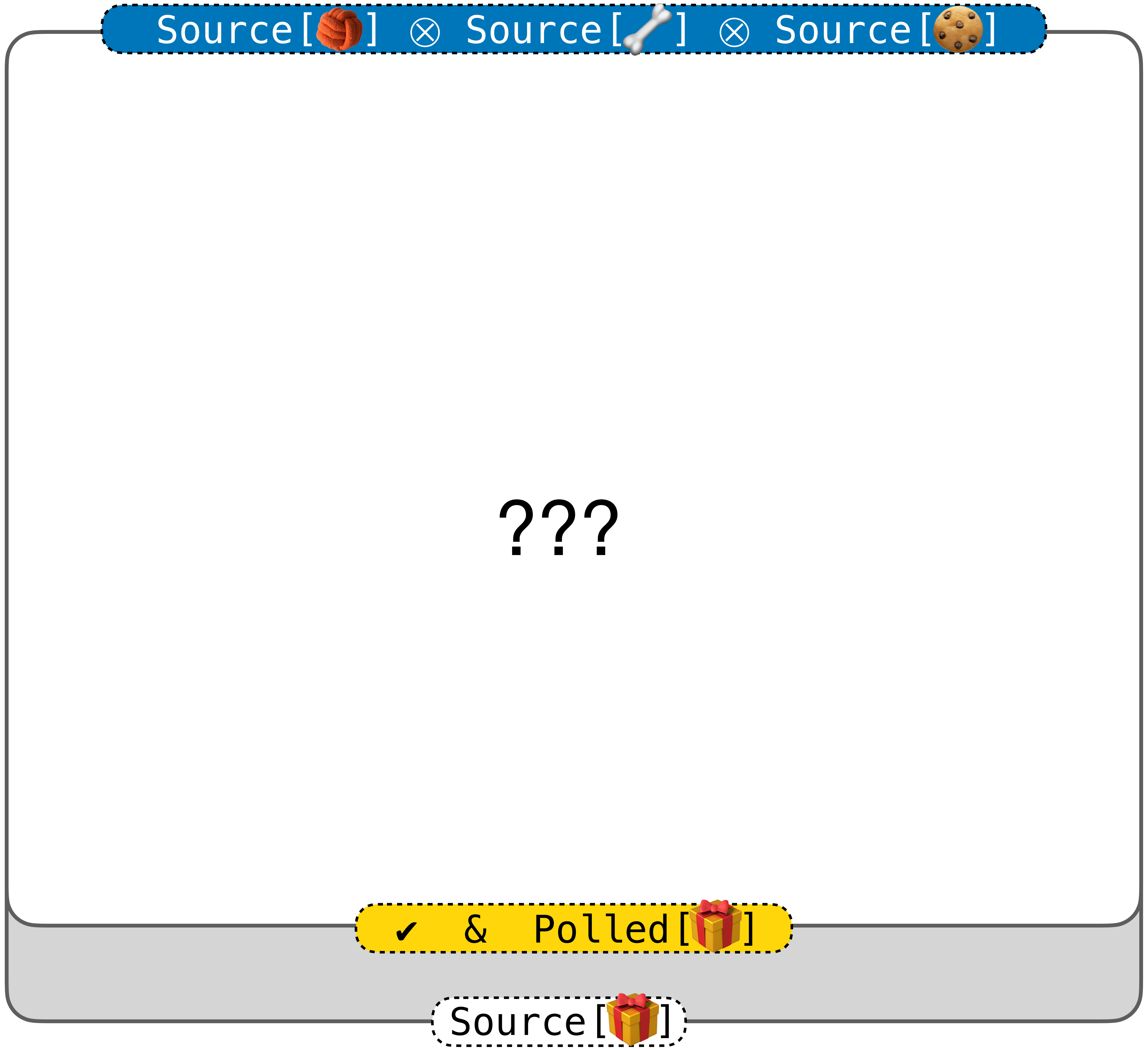
`✔ & Polled[🎁]`

`Source.fromChoice`

`Source[🎁]`

???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair
&	consumer choice
✔	Done signal
`Polled[A]`	requested next elem

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

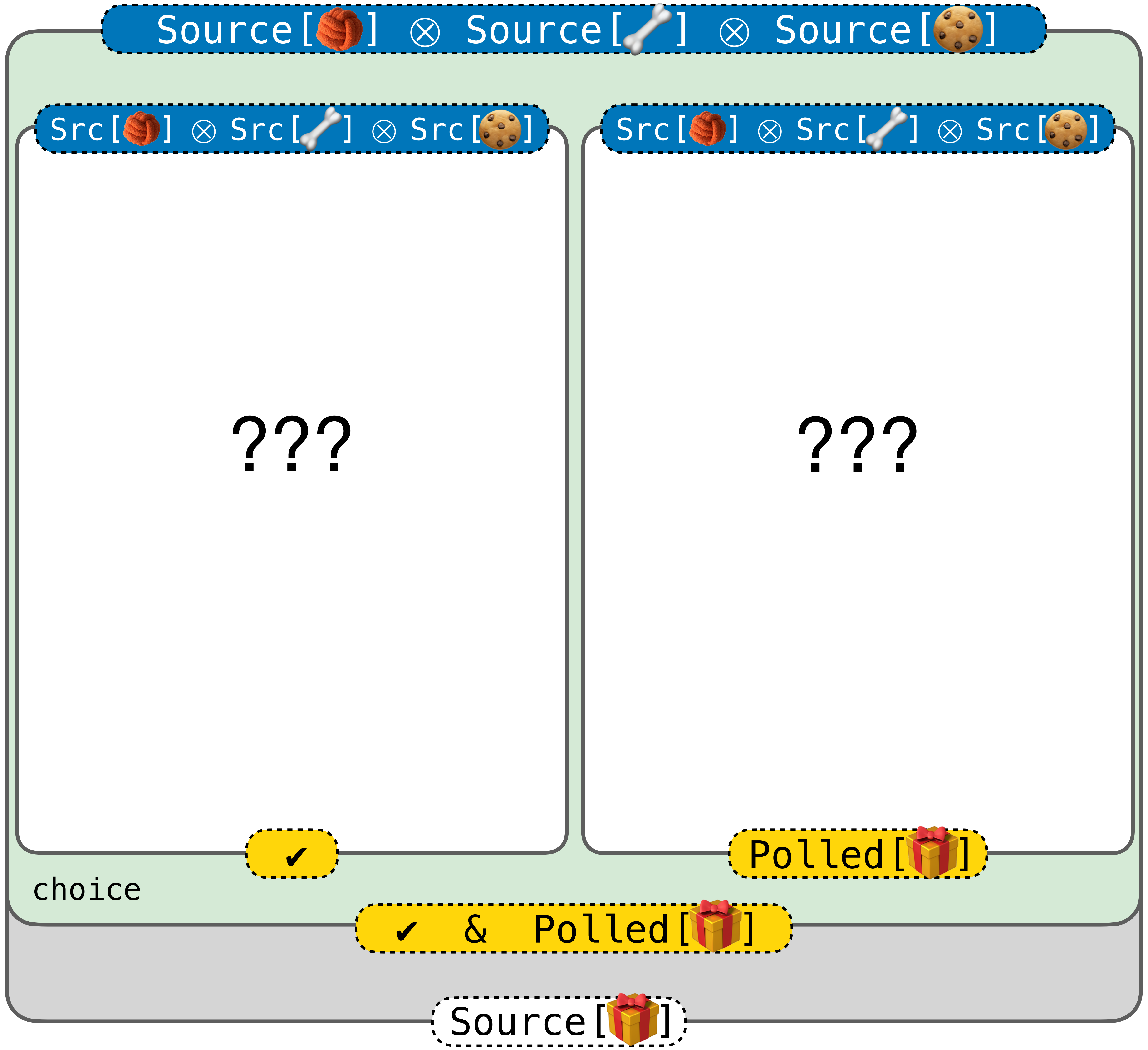
???

✔ & Polled[🎁]

Source[🎁]

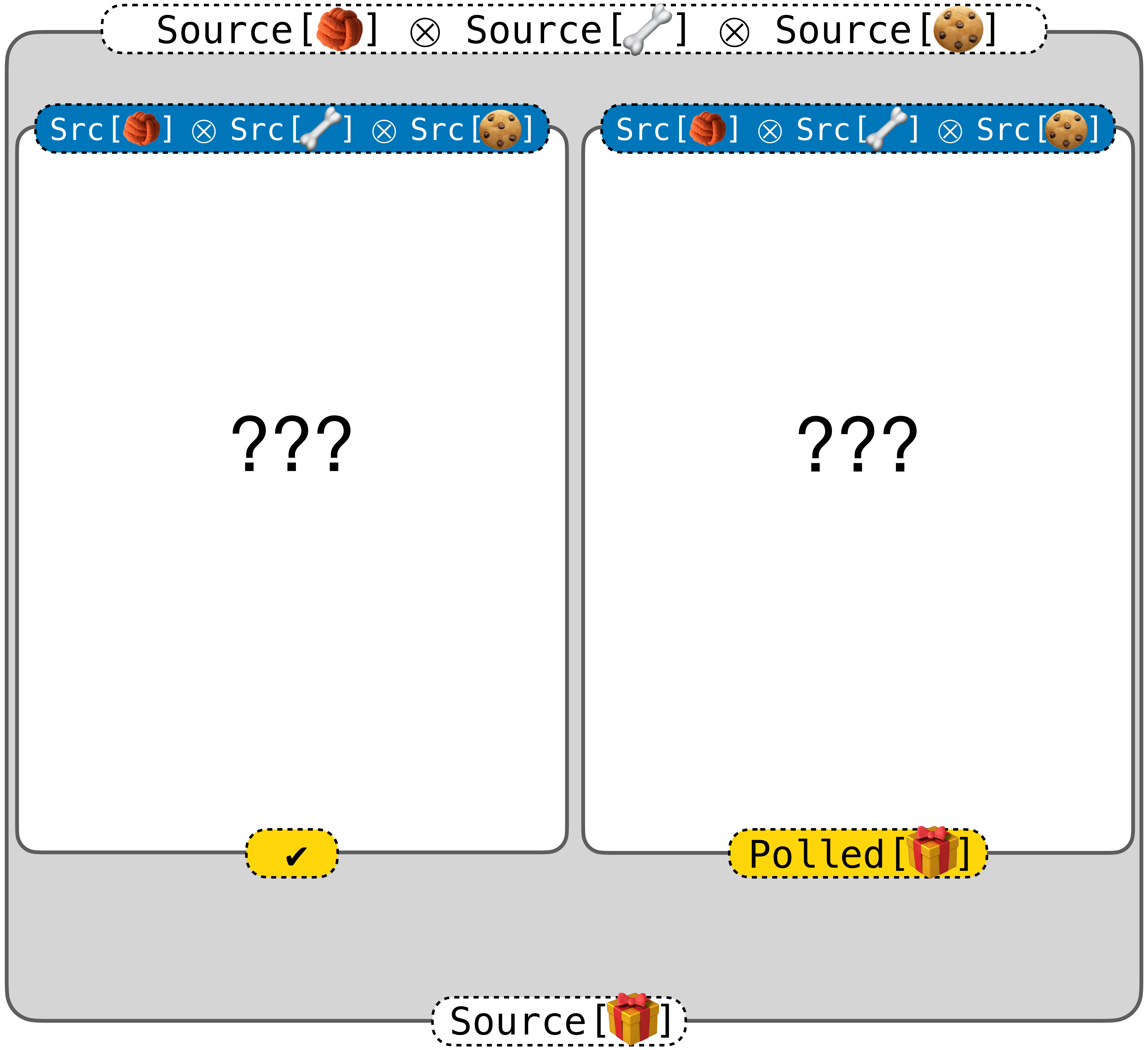
???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair
&	consumer choice
✔	Done signal
Polled[A]	requested next elem

Packaging Dog Presents



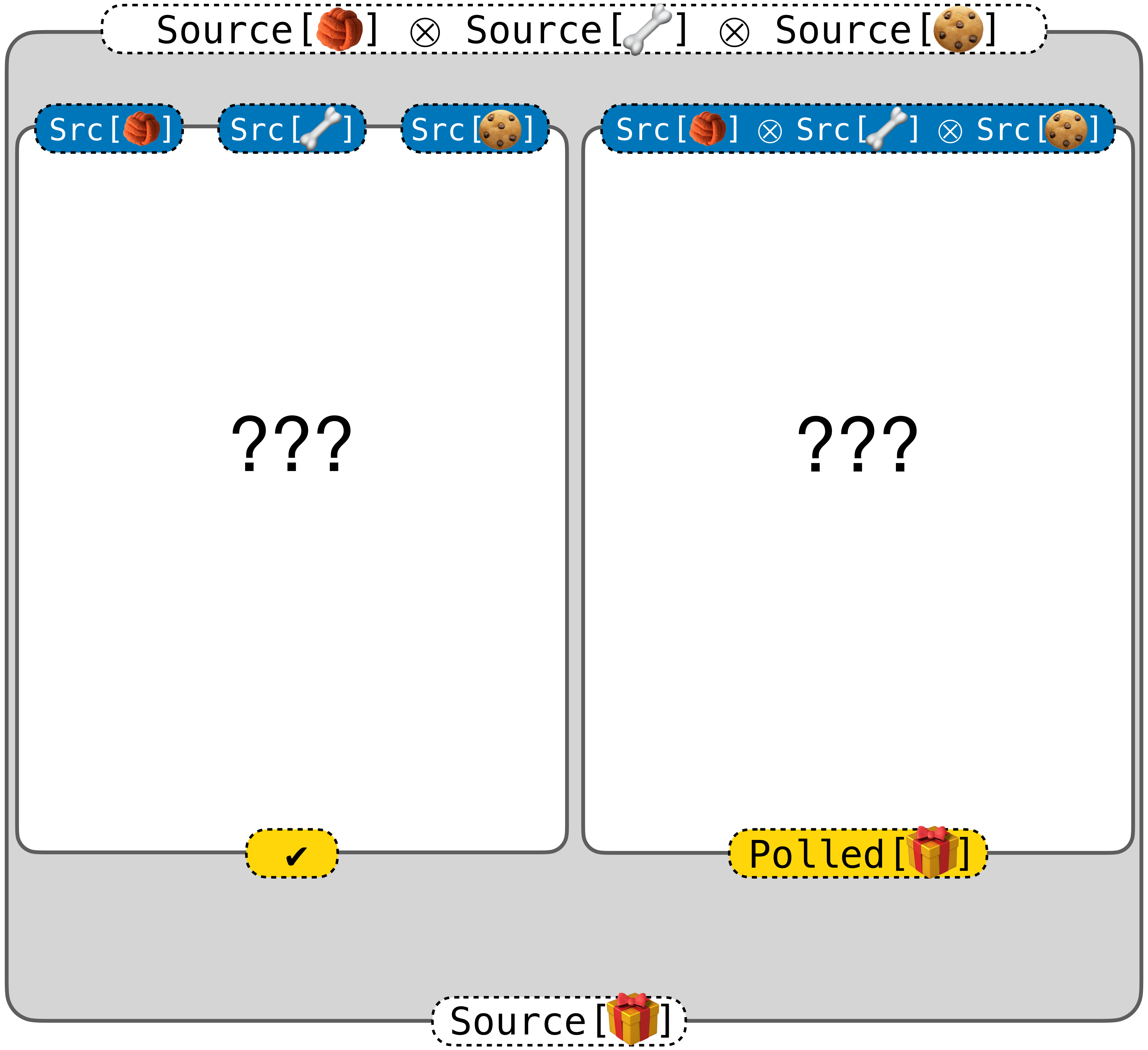
???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair
&	consumer choice
✔	Done signal
`Polled[A]`	requested next elem
`Src[A]`	abbr. `Source[A]`

Packaging Dog Presents



???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair
&	consumer choice
✔	Done signal
`Polled[A]`	requested next elem
`Src[A]`	abbr. `Source[A]`

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

Src[🧶] Src[🦴] Src[🍪]

???

✔

Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]

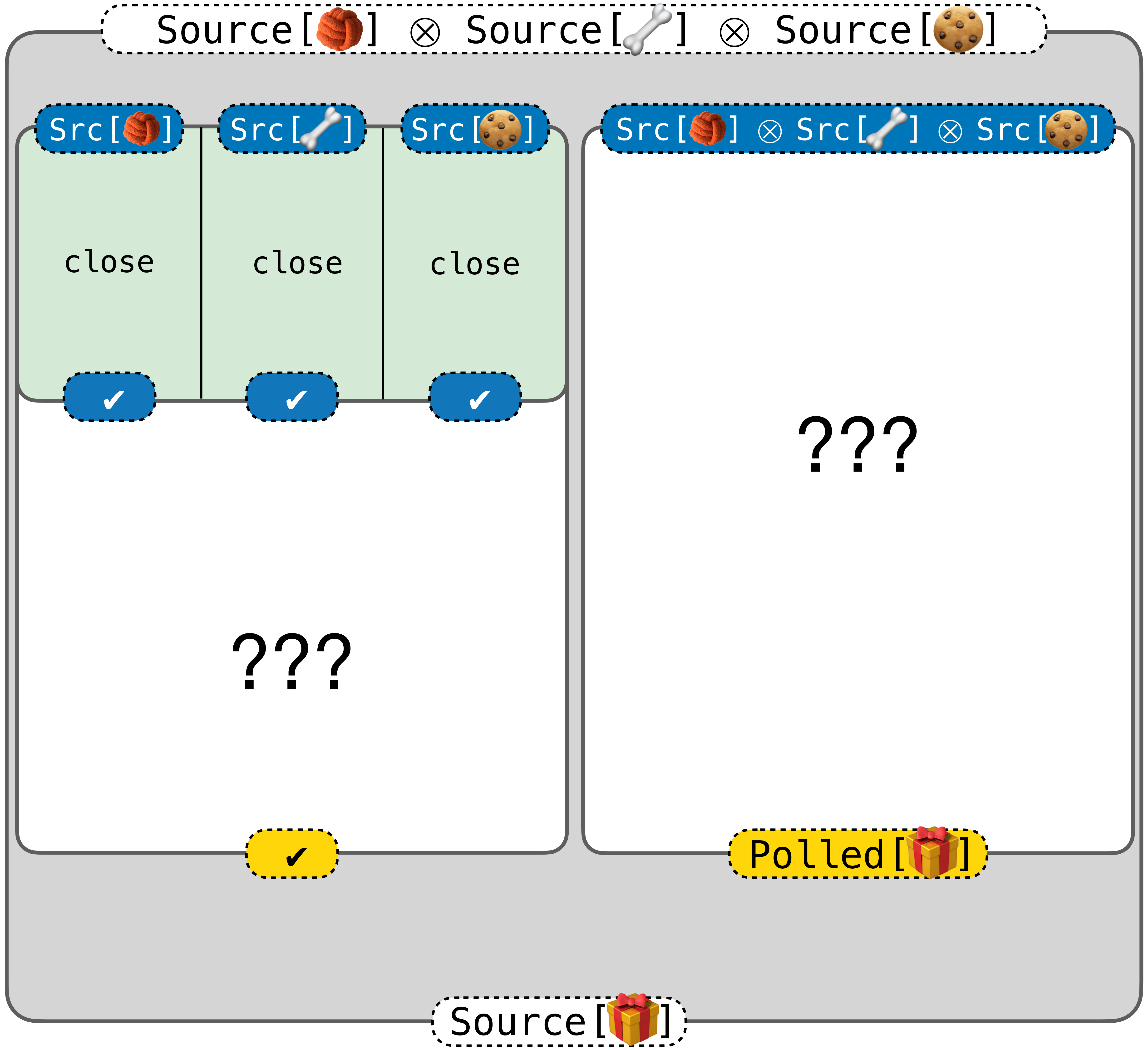
???

Polled[🎁]

Source[🎁]

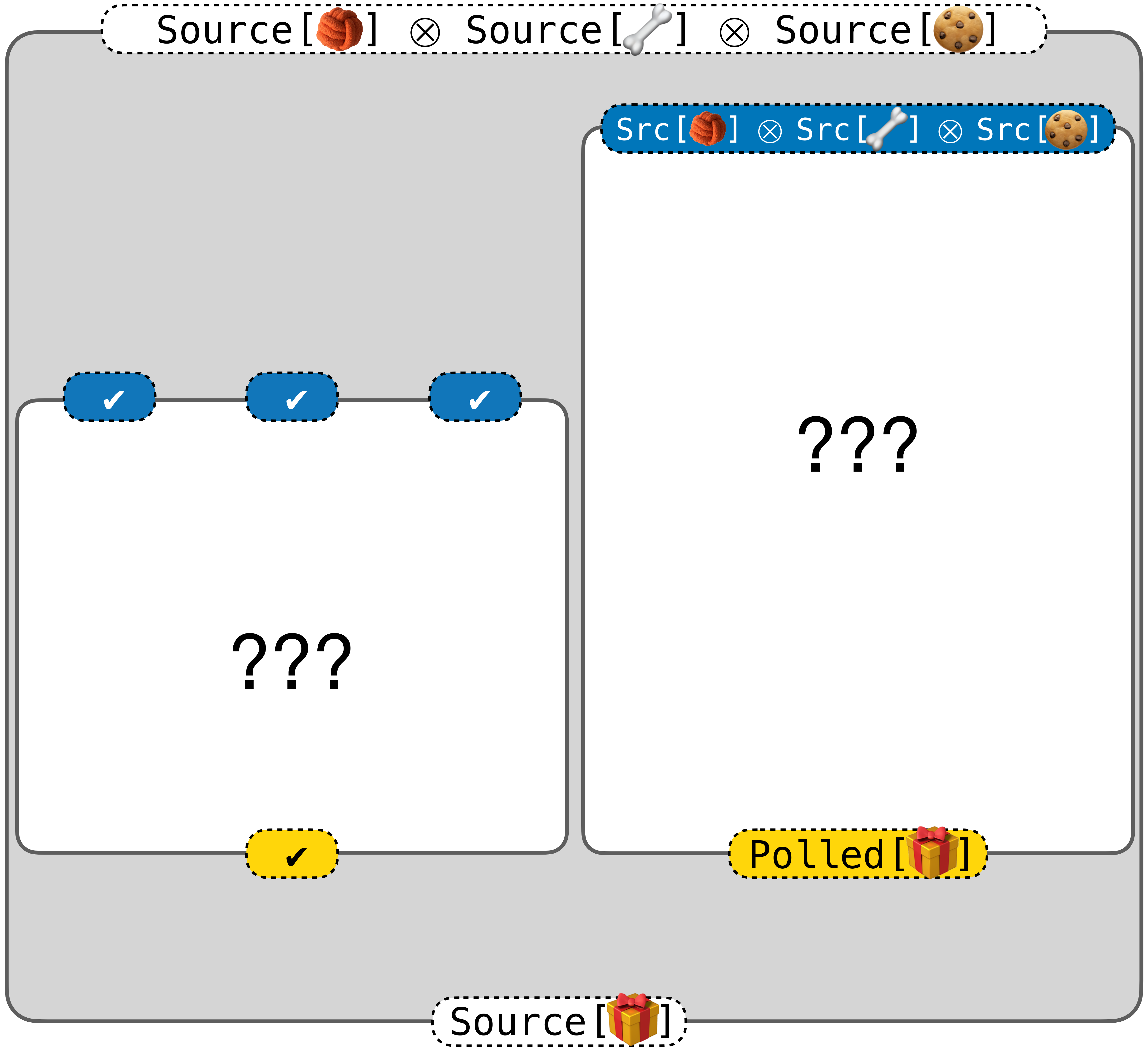
???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair
&	consumer choice
✔	Done signal
`Polled[A]`	requested next elem
`Src[A]`	abbr. `Source[A]`

Packaging Dog Presents



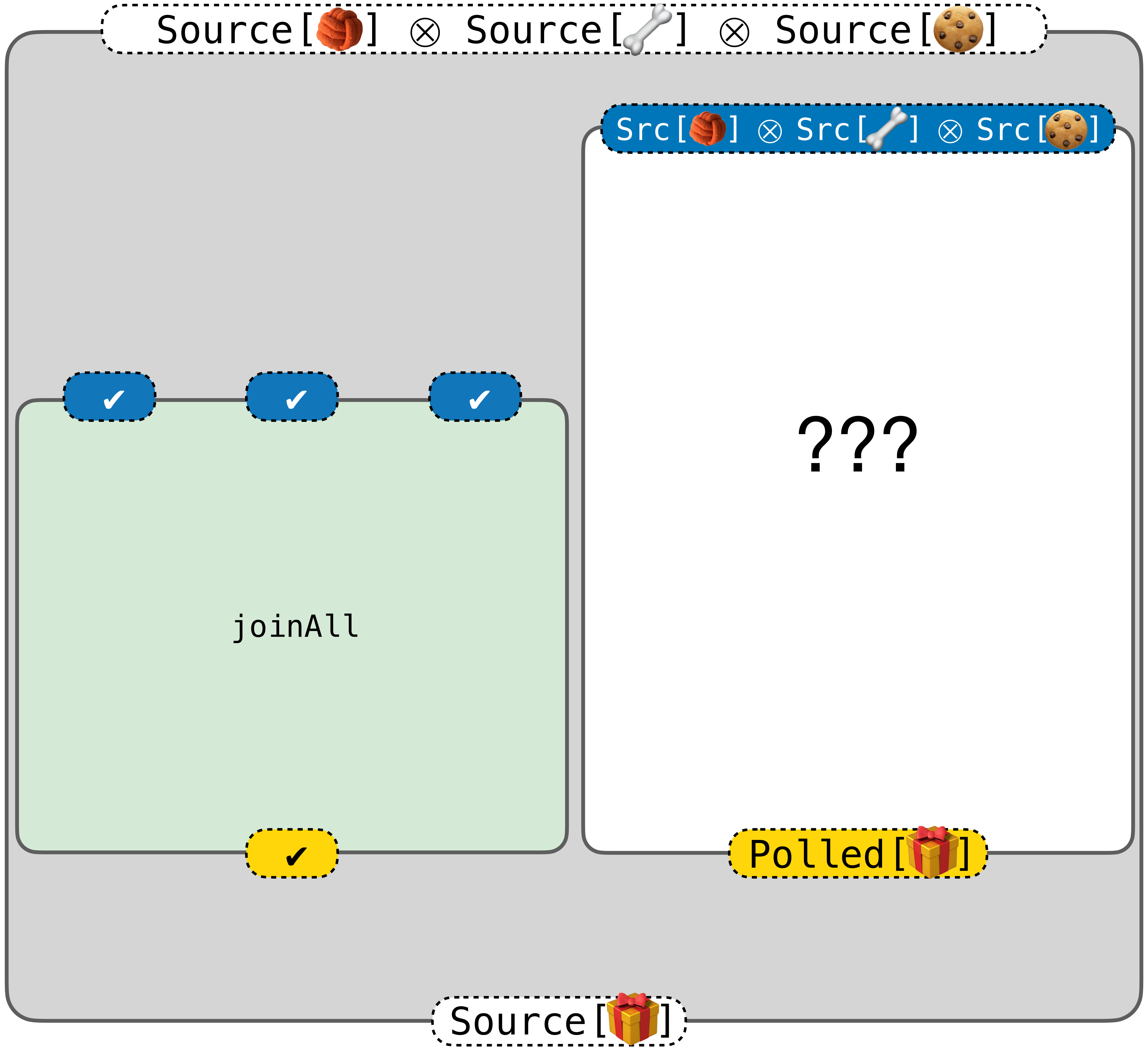
???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair
&	consumer choice
✔	Done signal
Polled[A]	requested next elem
Src[A]	abbr. Source[A]

Packaging Dog Presents



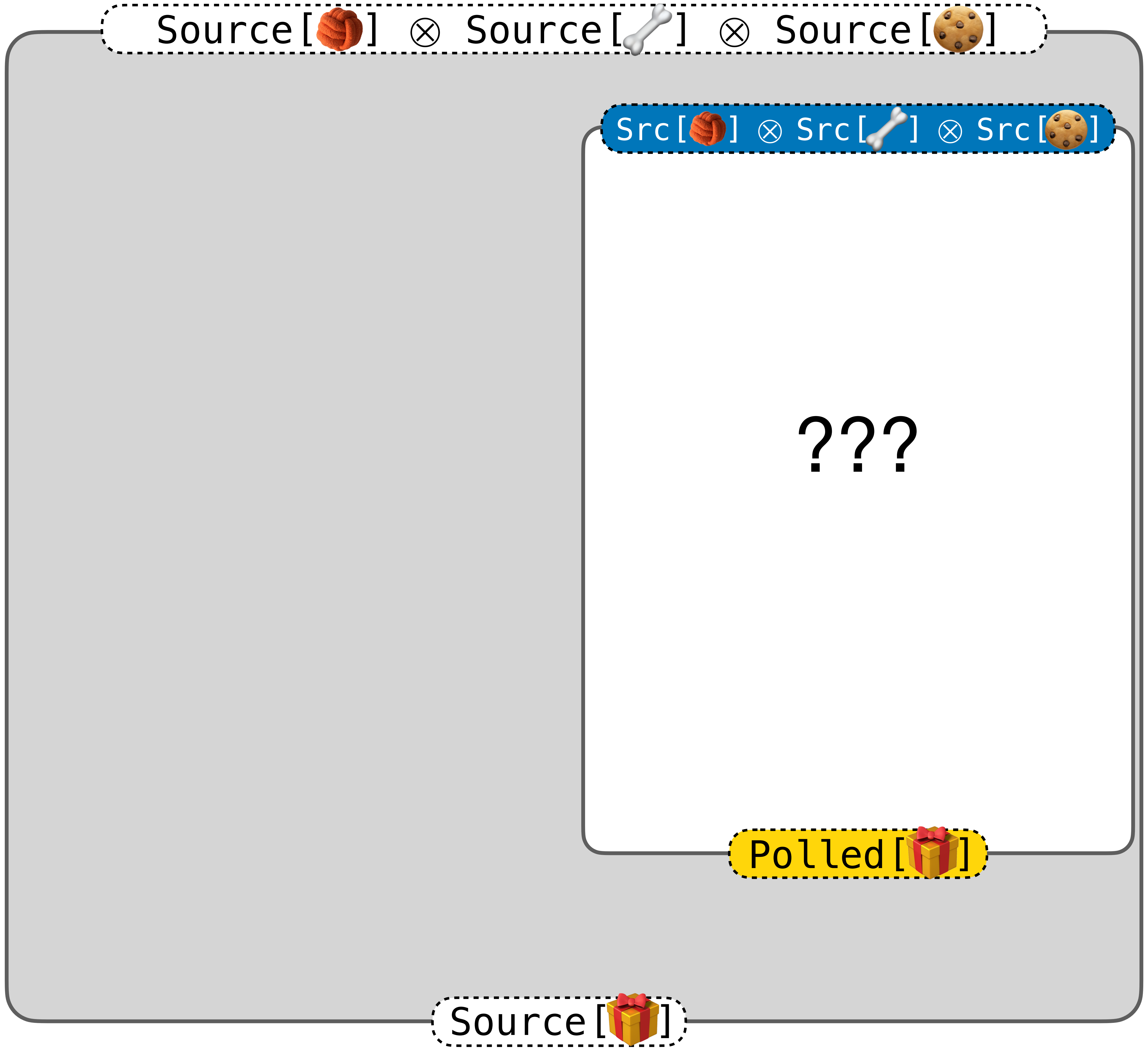
???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair
&	consumer choice
✓	Done signal
Polled[A]	requested next elem
Src[A]	abbr. Source[A]

Packaging Dog Presents



???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair
&	consumer choice
✓	Done signal
`Polled[A]`	requested next elem
`Src[A]`	abbr. `Source[A]`

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]

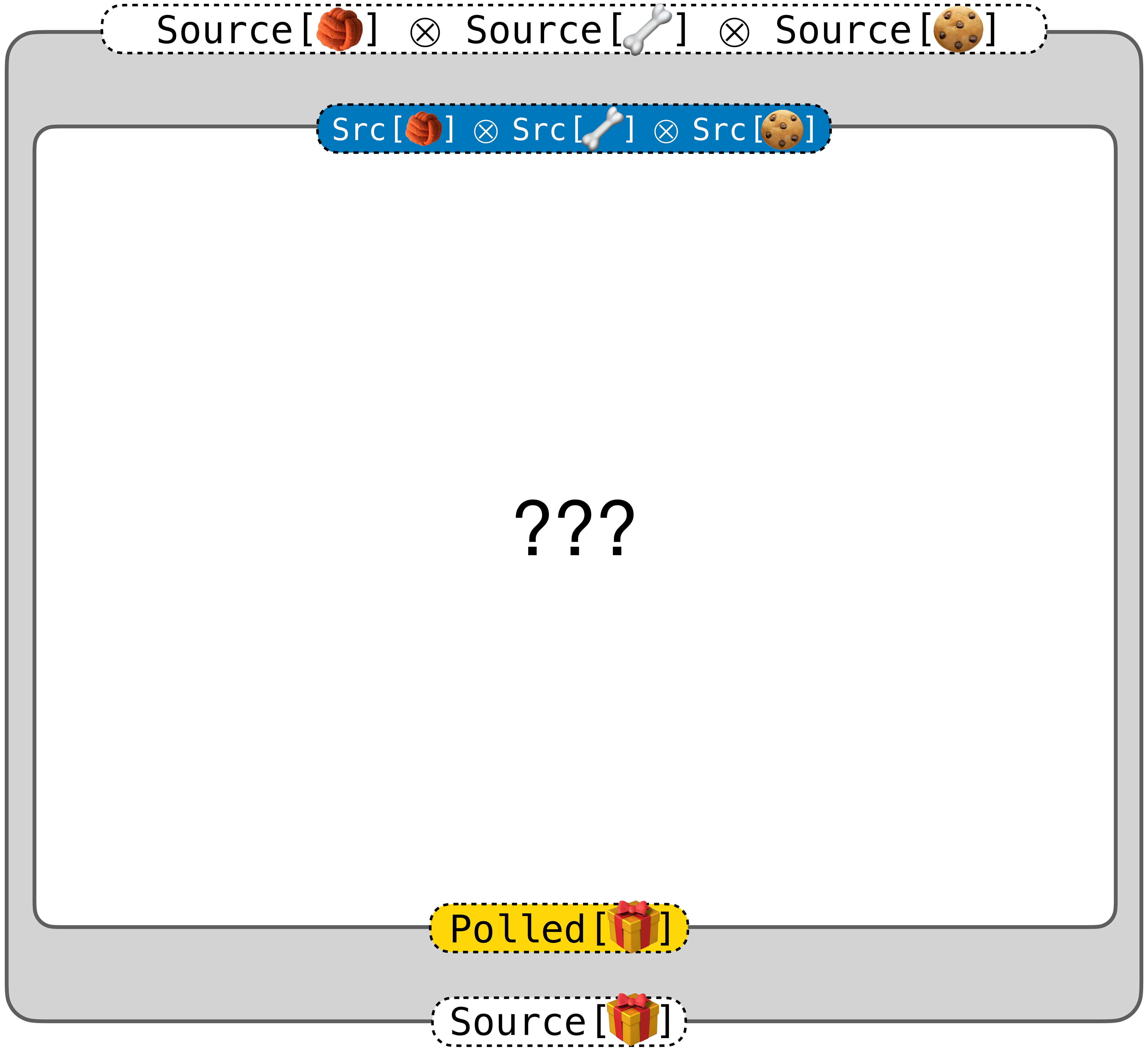
???

Polled[🎁]

Source[🎁]

???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair
&	consumer choice
✔	Done signal
`Polled[A]`	requested next elem
`Src[A]`	abbr. `Source[A]`

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]

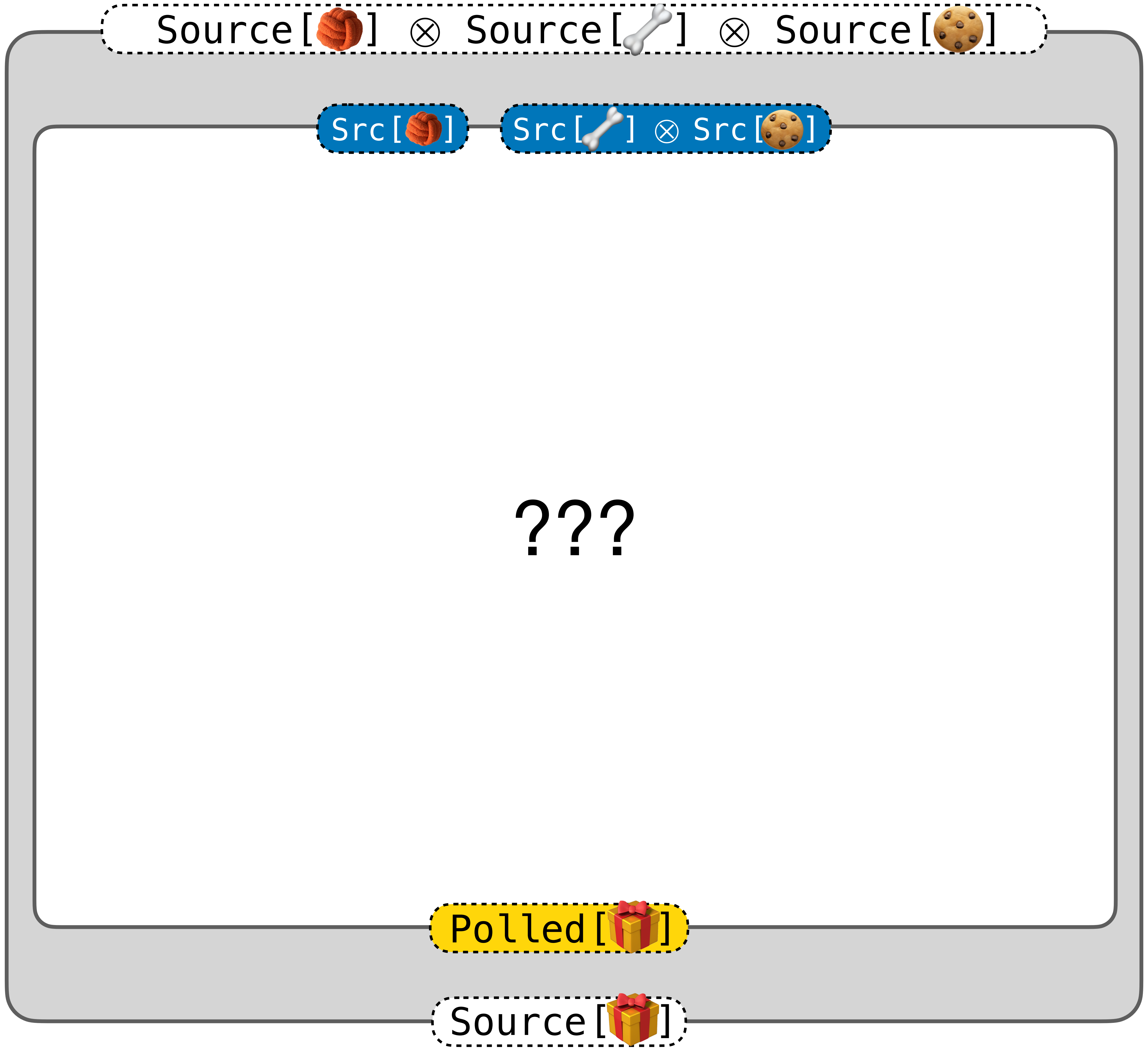
???

Polled[🎁]

Source[🎁]

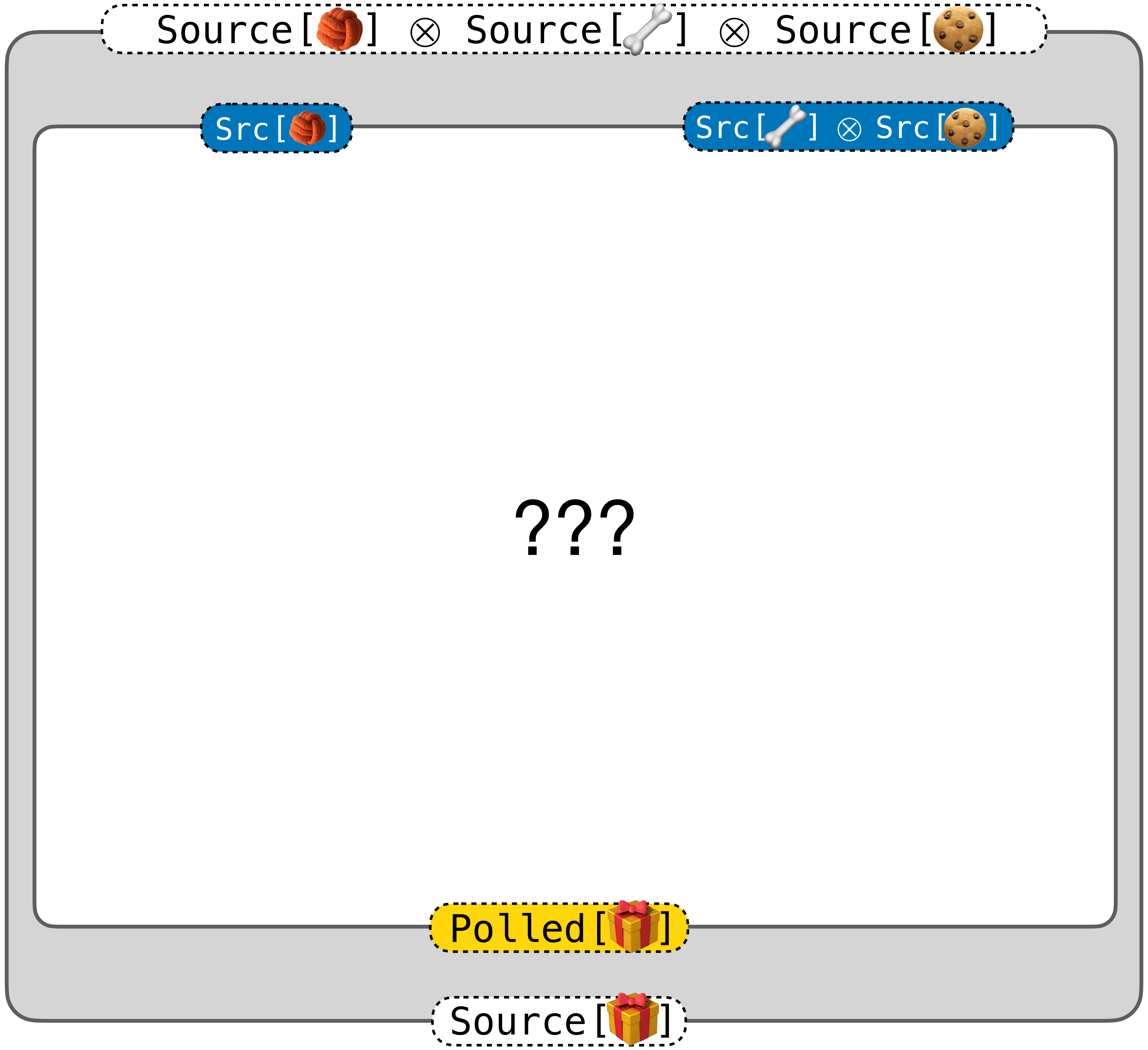
???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair
&	consumer choice
	.
✔	Done signal
Polled[A]	requested next elem
Src[A]	abbr. Source[A]

Packaging Dog Presents



???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair
&	consumer choice
	.
✔	Done signal
Polled[A]	requested next elem
Src[A]	abbr. Source[A]

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

Src[🧶]

Src[🦴] ⊗ Src[🍪]

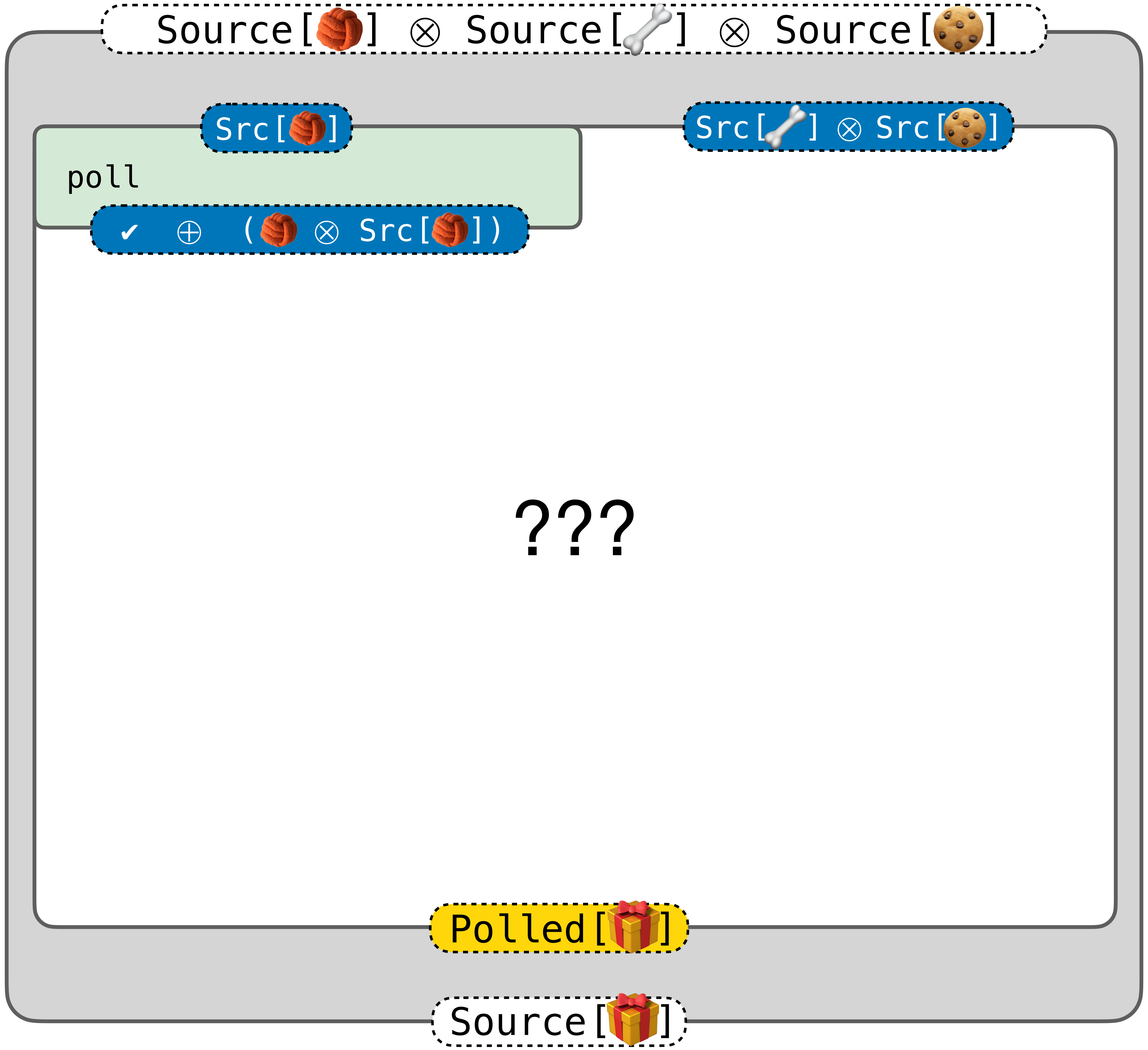
???

Polled[🎁]

Source[🎁]

???	hole to be filled
🟦	to be consumed
🟨	to be produced
⊗	concurrent pair
&	consumer choice
✔	Done signal
Polled[A]	requested next elem
Src[A]	abbr. Source[A]

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

Src[🧶]

Src[🦴] ⊗ Src[🍪]

poll

✔ ⊕ (🧶 ⊗ Src[🧶])

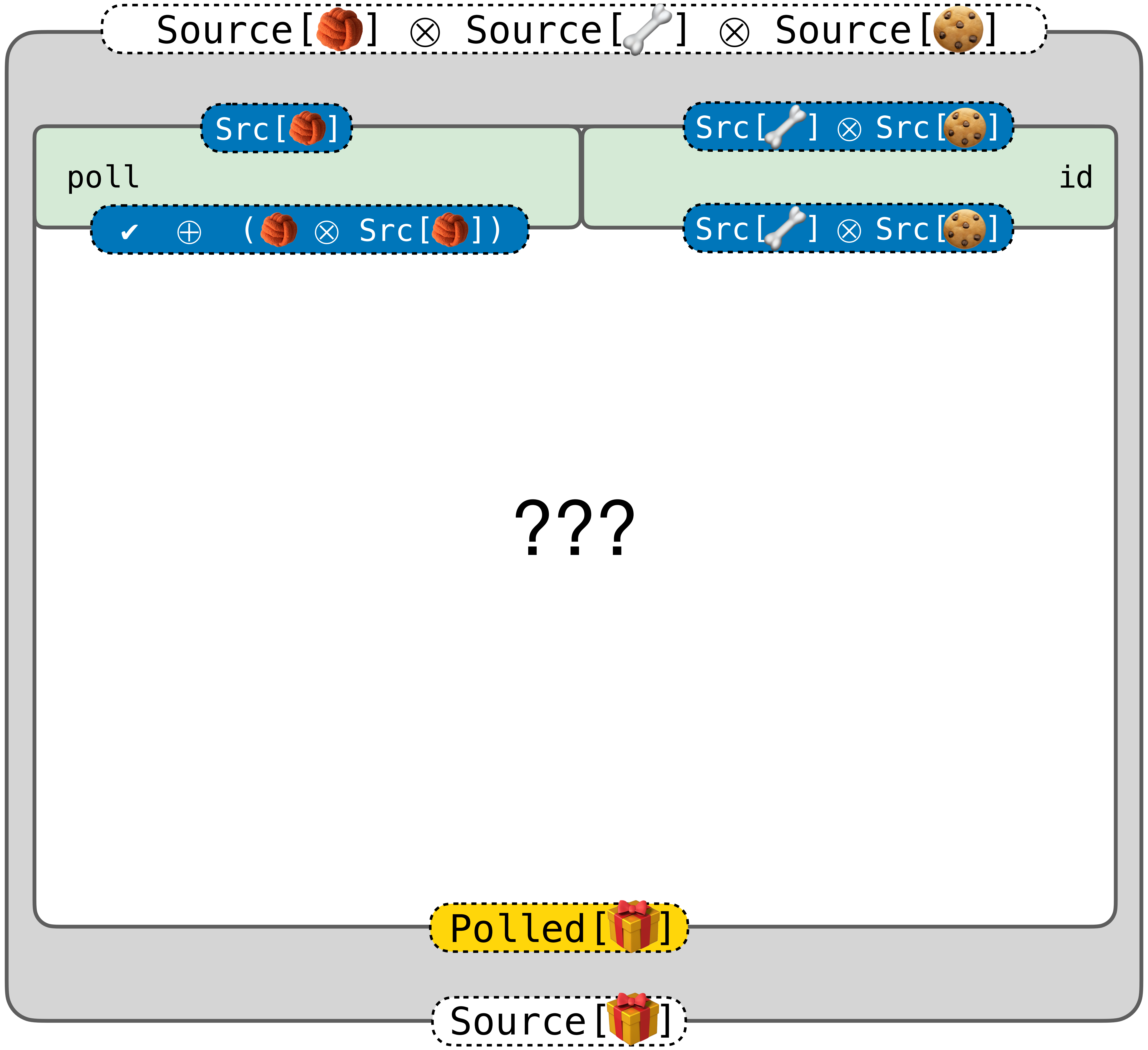
???

Polled[🎁]

Source[🎁]

???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
Polled[A]	requested next elem	✔ ⊕ (A ⊗ Source[A])
Src[A]	abbr. Source[A]	

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

Src[🧶]

poll

✔ ⊕ (🧶 ⊗ Src[🧶])

Src[🦴] ⊗ Src[🍪]

id

Src[🦴] ⊗ Src[🍪]

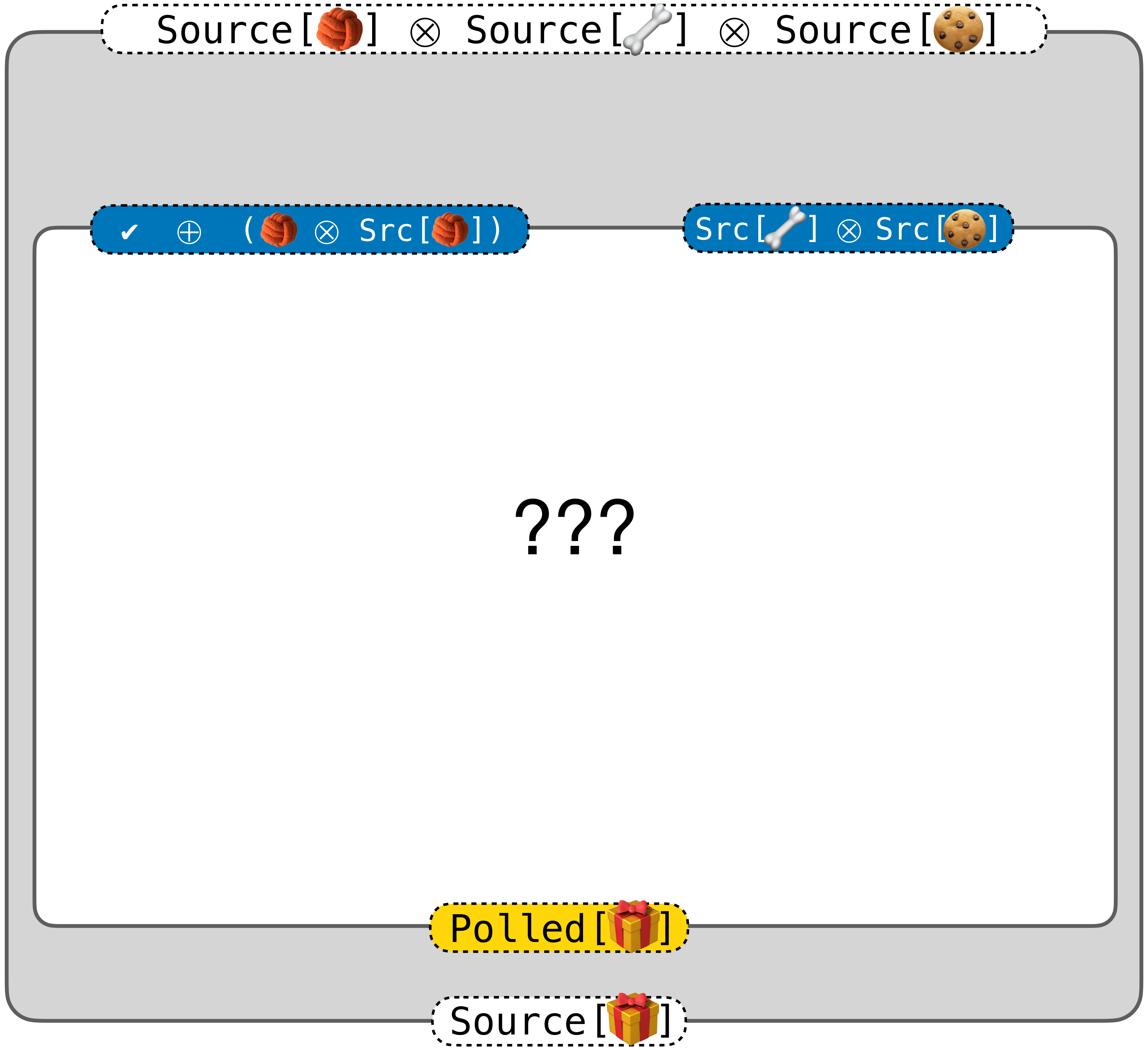
???

Polled[🎁]

Source[🎁]

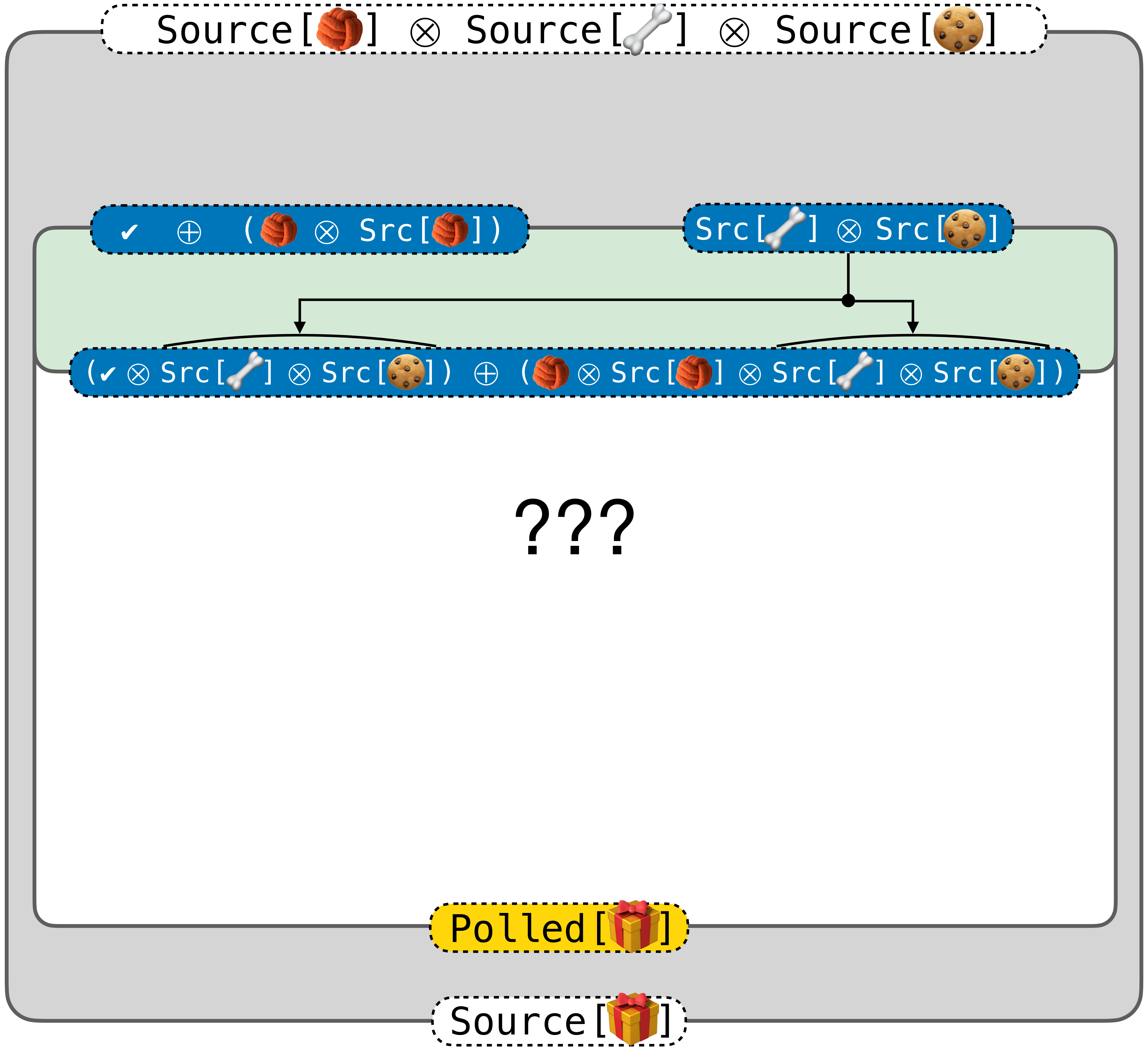
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	✔ ⊕ (A ⊗ Source[A])
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents



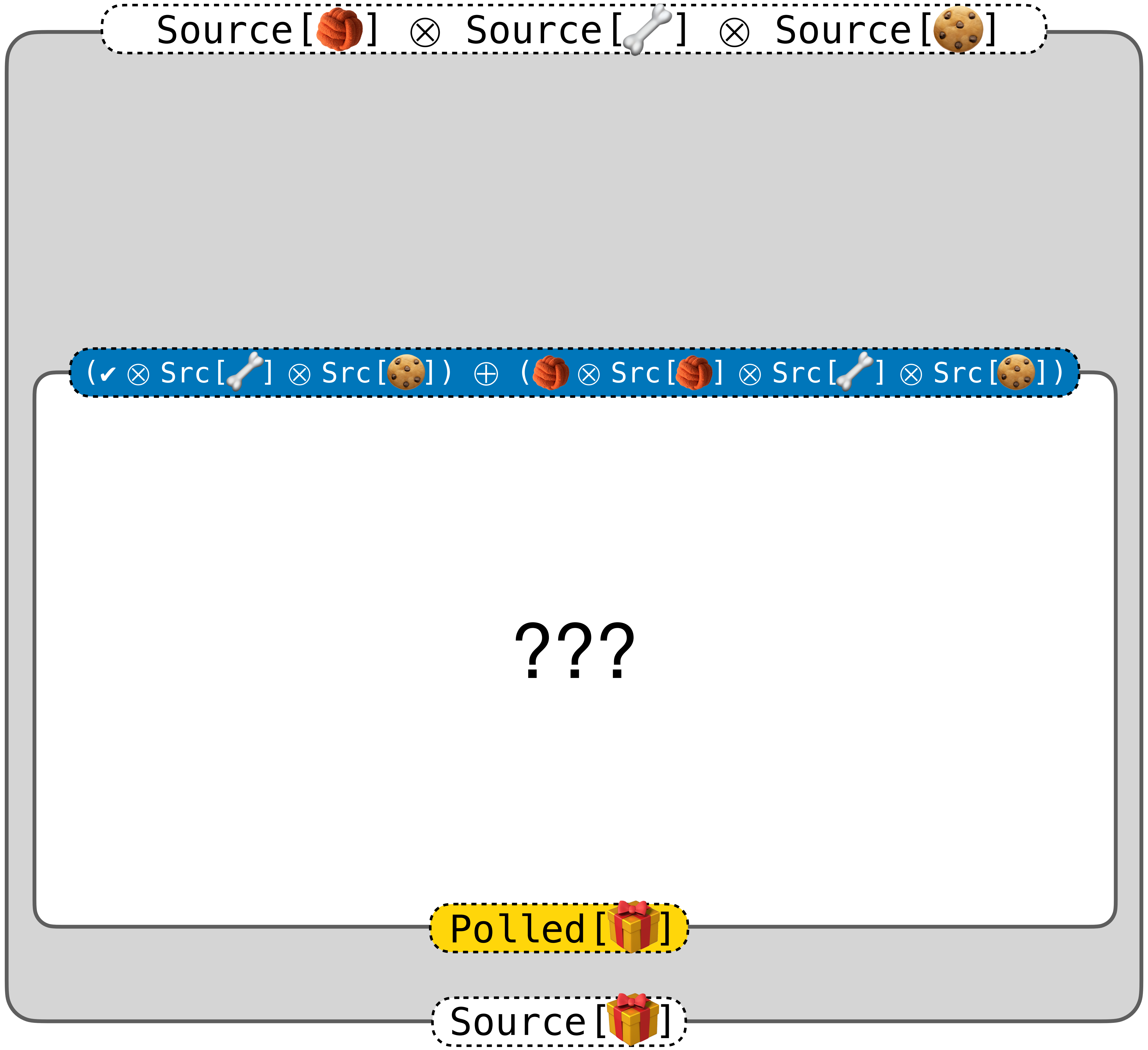
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
Polled[A]	requested next elem	✔ ⊕ (A ⊗ Source[A])
Src[A]	abbr. Source[A]	

Packaging Dog Presents


Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]
✔ ⊕ (🧶 ⊗ Src[🧶])
Src[🦴] ⊗ Src[🍪]
(✔ ⊗ Src[🦴] ⊗ Src[🍪]) ⊕ (🧶 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪])
???
Polled[🎁]
Source[🎁]

???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents



`Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]`

`(✔ ⊗ Src[🦴] ⊗ Src[🍪]) ⊕ (🧶 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪])`

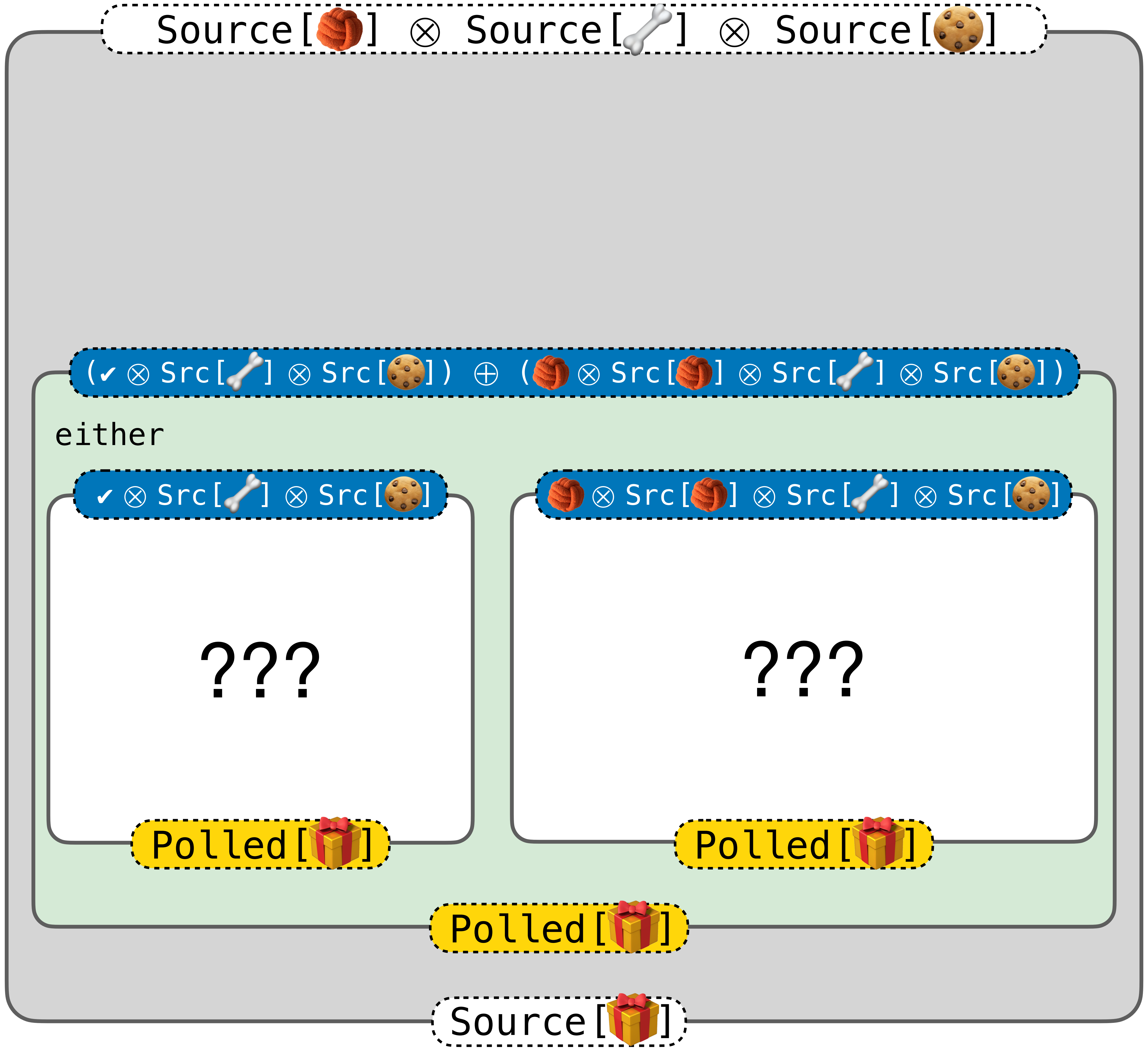
???

`Polled[🎁]`

`Source[🎁]`

???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	`Done` signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

(✔ ⊗ Src[🦴] ⊗ Src[🍪]) ⊕ (🧶 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪])

either

✔ ⊗ Src[🦴] ⊗ Src[🍪]

🧶 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]

???

???

Polled[🎁]

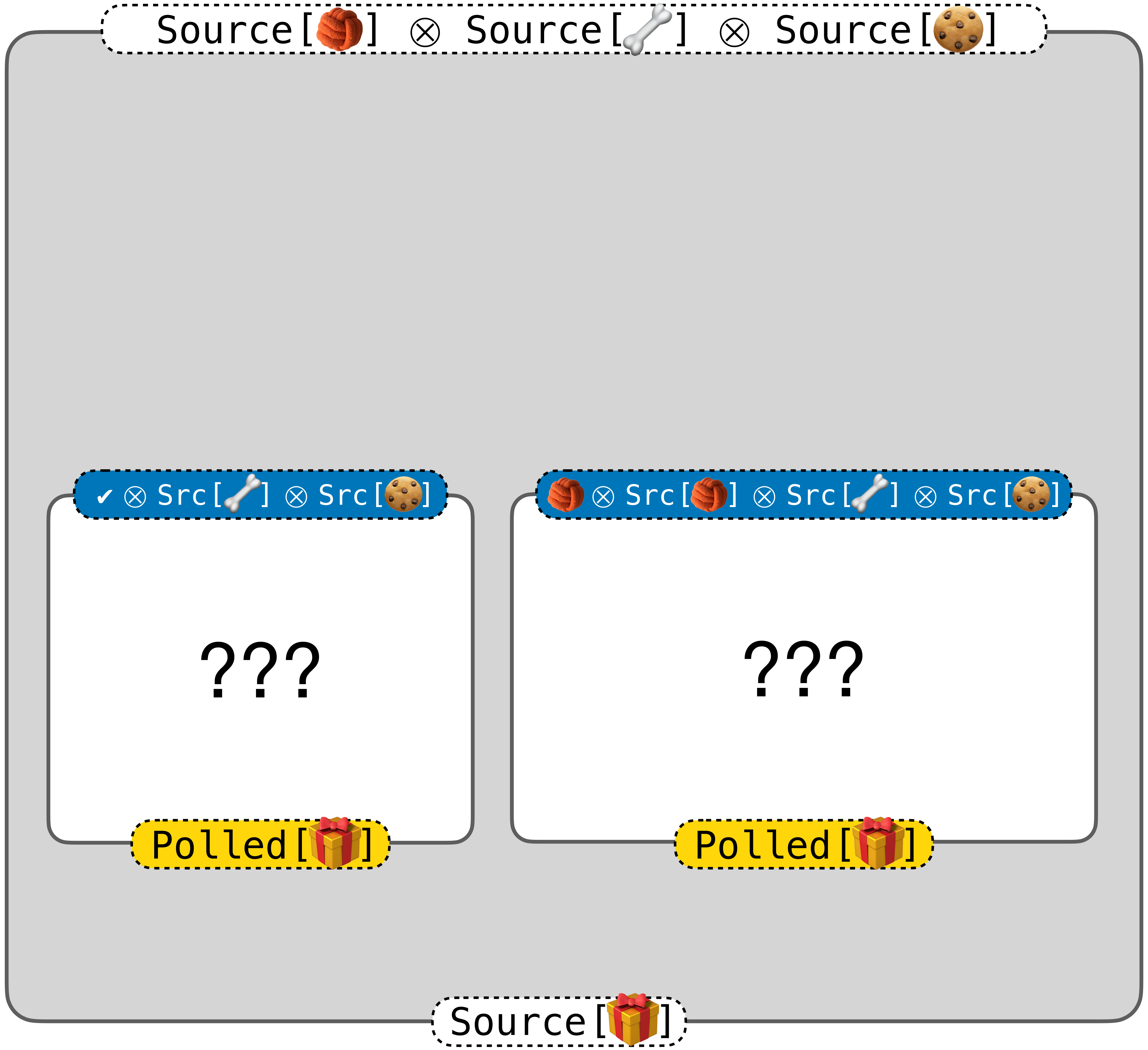
Polled[🎁]

Polled[🎁]

Source[🎁]

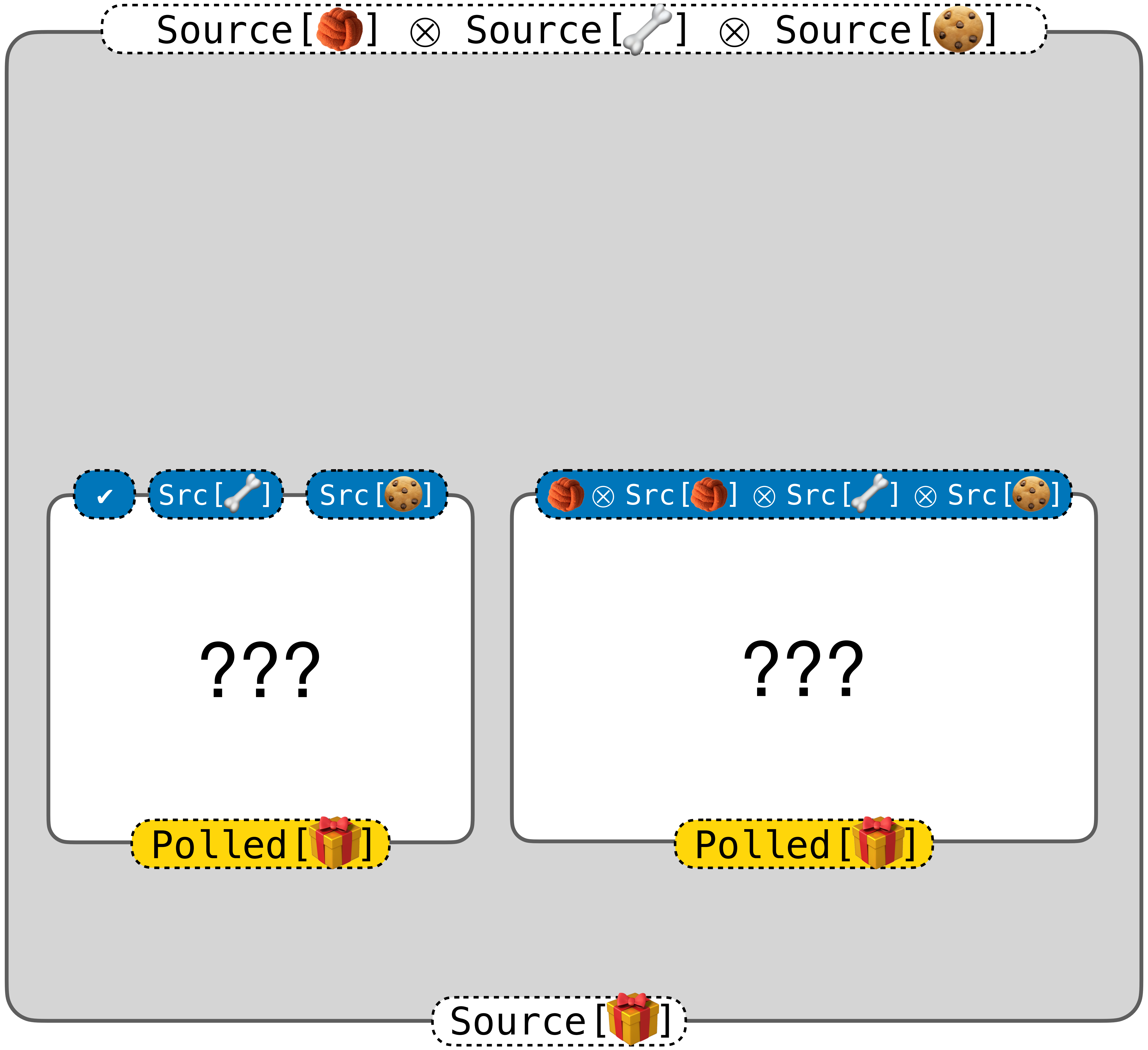
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents


Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]
✔ ⊗ Src[🦴] ⊗ Src[🍪]
???
Polled[🎁]
🧶 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]
???
Polled[🎁]
Source[🎁]

???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

✔ Src[🦴] Src[🍪]

???

Polled[🎁]

🧶 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]

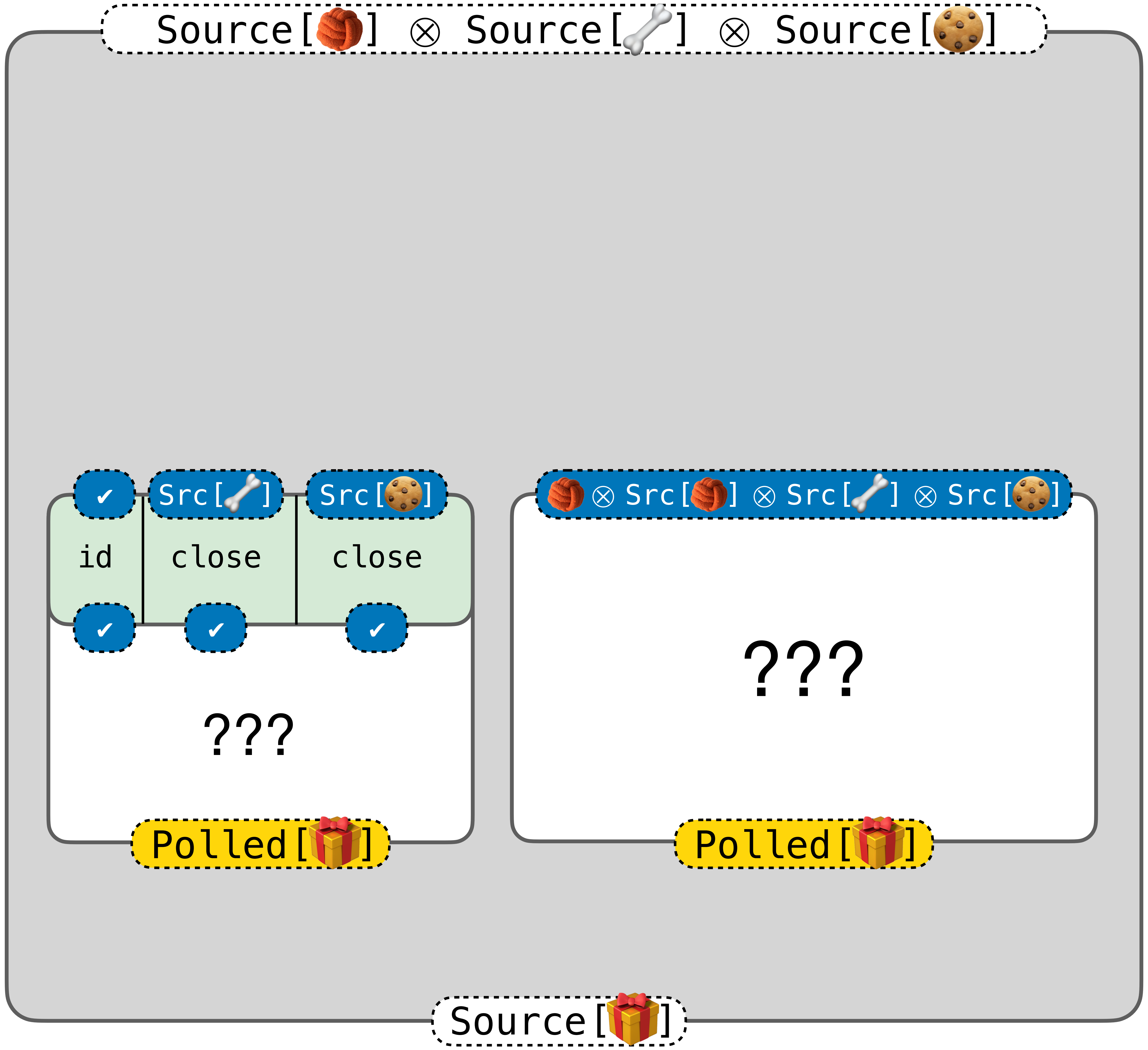
???

Polled[🎁]

Source[🎁]

???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
Polled[A]	requested next elem	✔ ⊕ (A ⊗ Source[A])
Src[A]	abbr. Source[A]	

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

✓ Src[🦴] Src[🍪]

id close close

✓ ✓ ✓

???

Polled[🎁]

🧶 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]

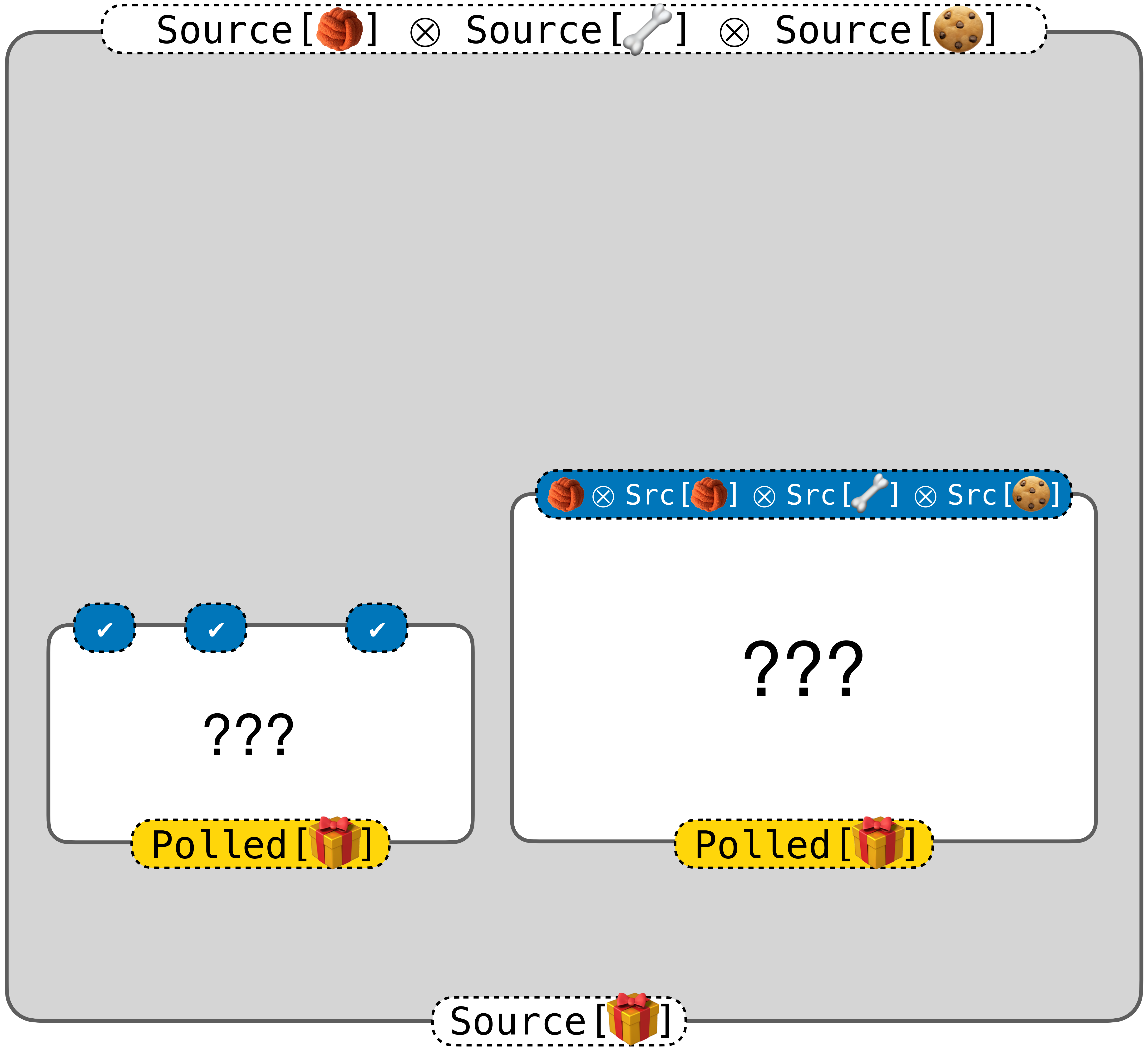
???

Polled[🎁]

Source[🎁]

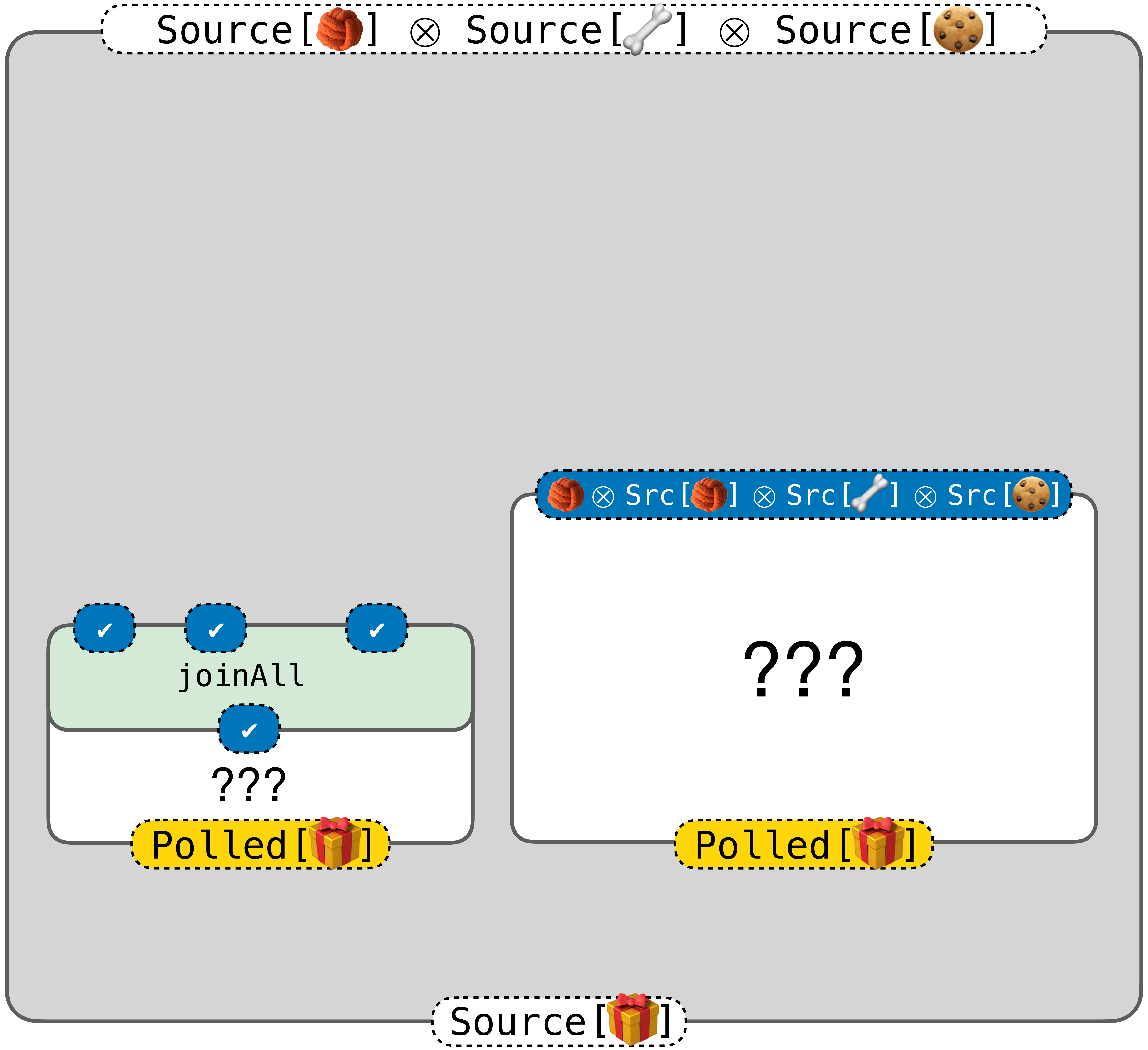
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✓	Done signal	
`Polled[A]`	requested next elem	`✓ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents


Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]
✓
✓
✓
???
Polled[🎁]
🧶 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]
???
Polled[🎁]
Source[🎁]

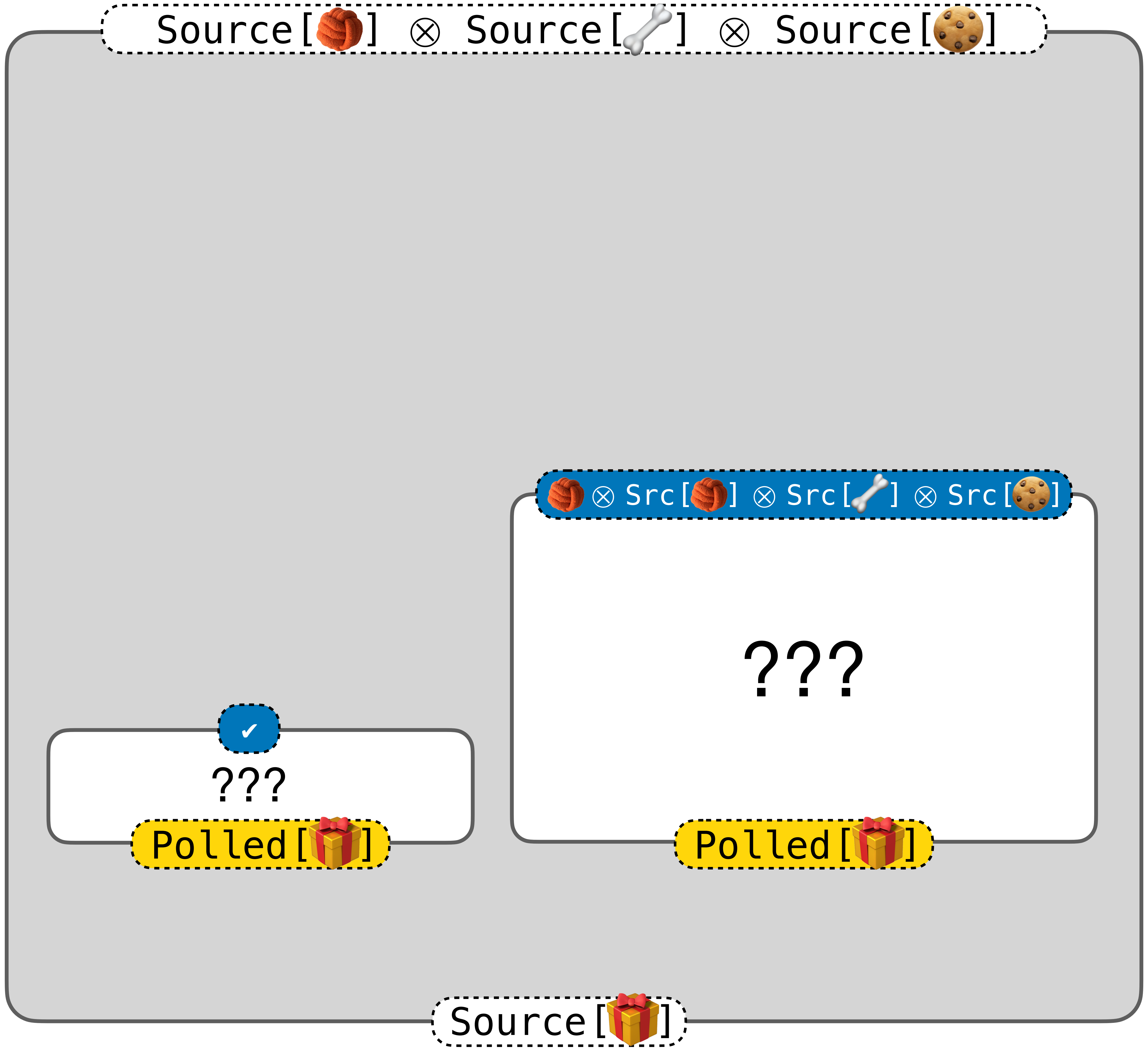
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents


Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]
✔
✔
✔
joinAll
✔
???
Polled[🎁]
🧶 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]
???
Polled[🎁]
Source[🎁]

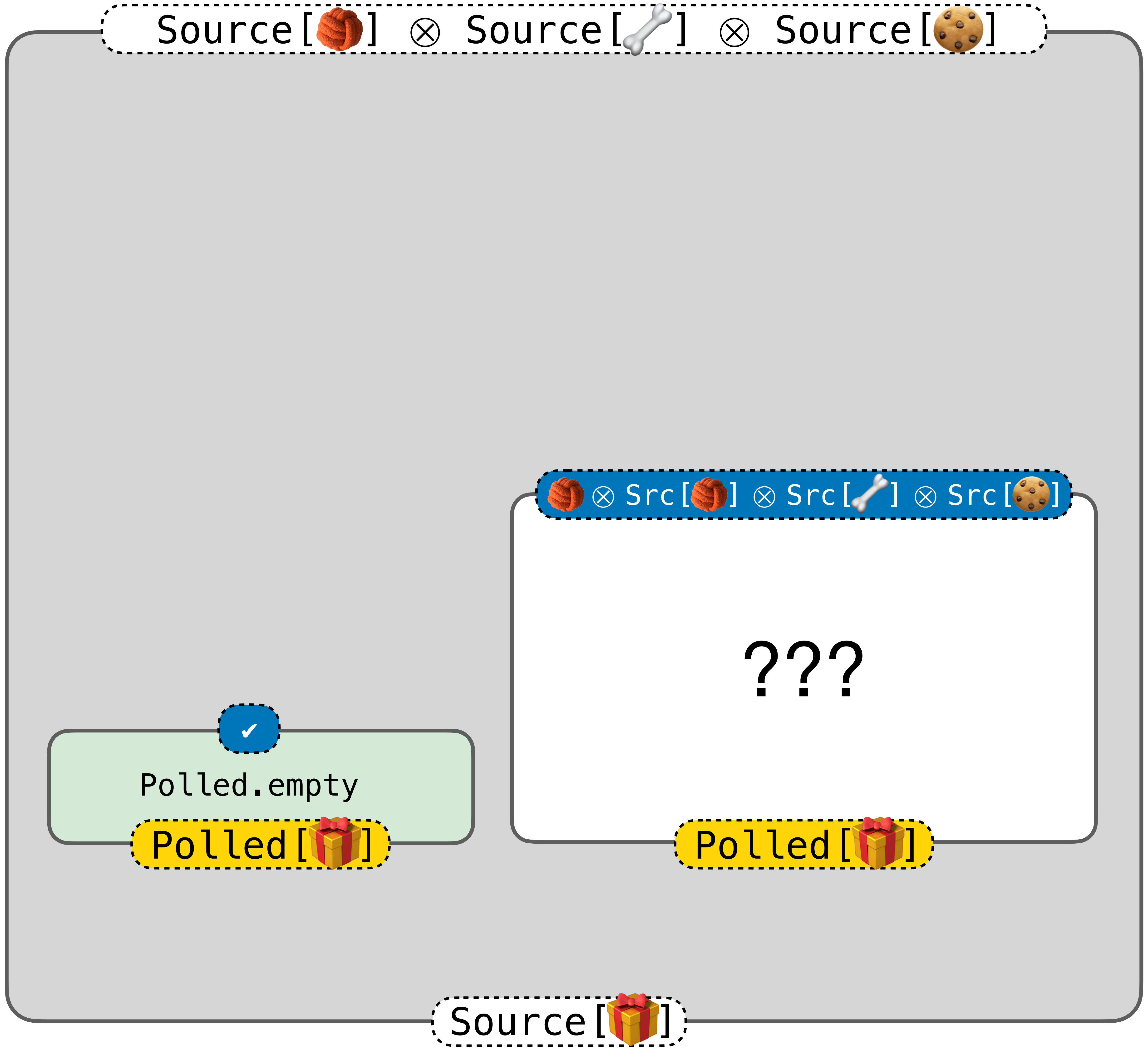
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents


Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]
🧶 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]
???
✔
???
Polled[🎁]
Polled[🎁]
Source[🎁]

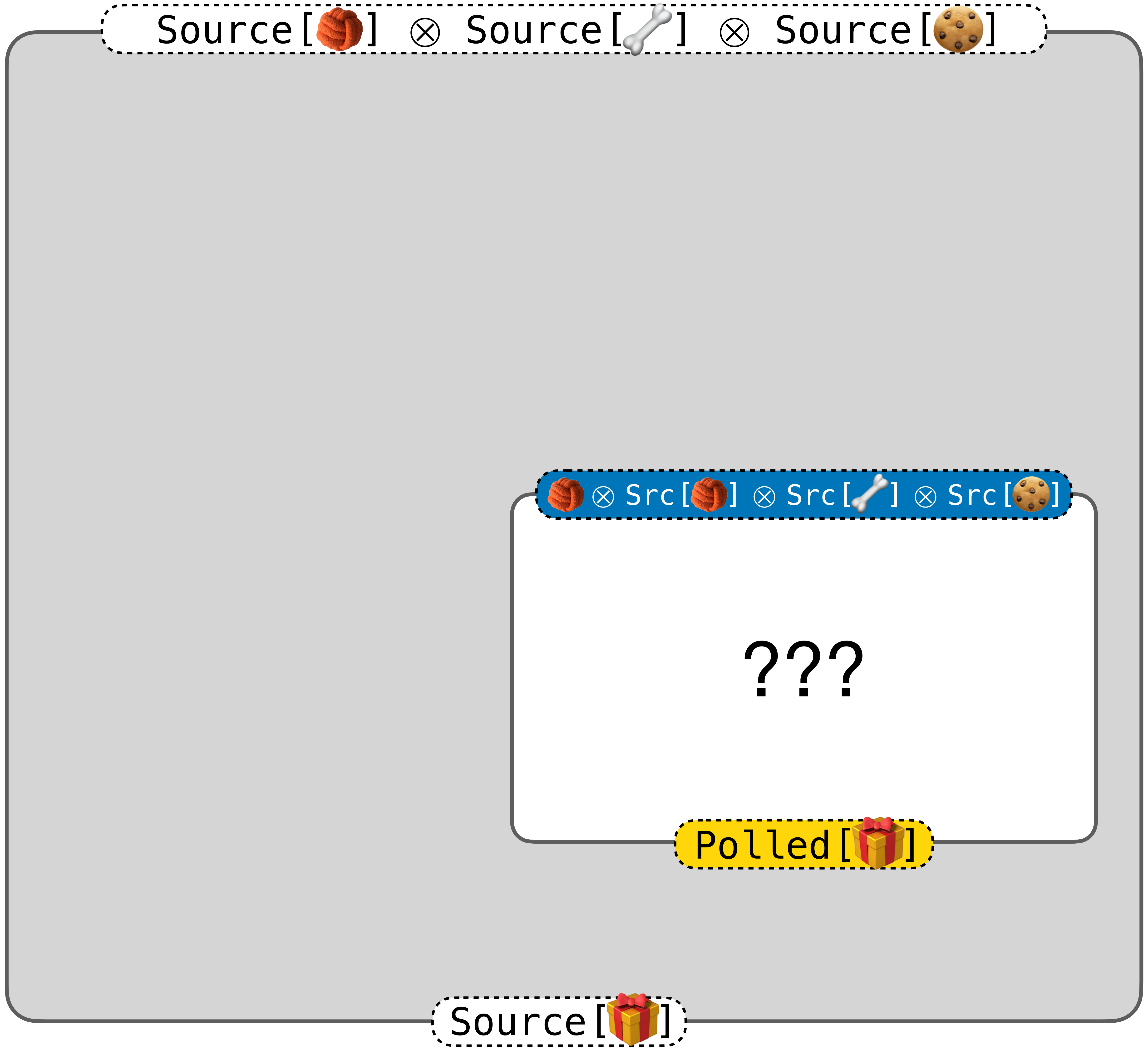
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
Polled[A]	requested next elem	✔ ⊕ (A ⊗ Source[A])
Src[A]	abbr. Source[A]	

Packaging Dog Presents


Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]
🧶 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]
???
Polled[🎁]
✓
Polled.empty
Polled[🎁]
Source[🎁]

???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	✔ ⊕ (A ⊗ Source[A])
`Src[A]`	abbr. Source[A]	

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

🧶 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]

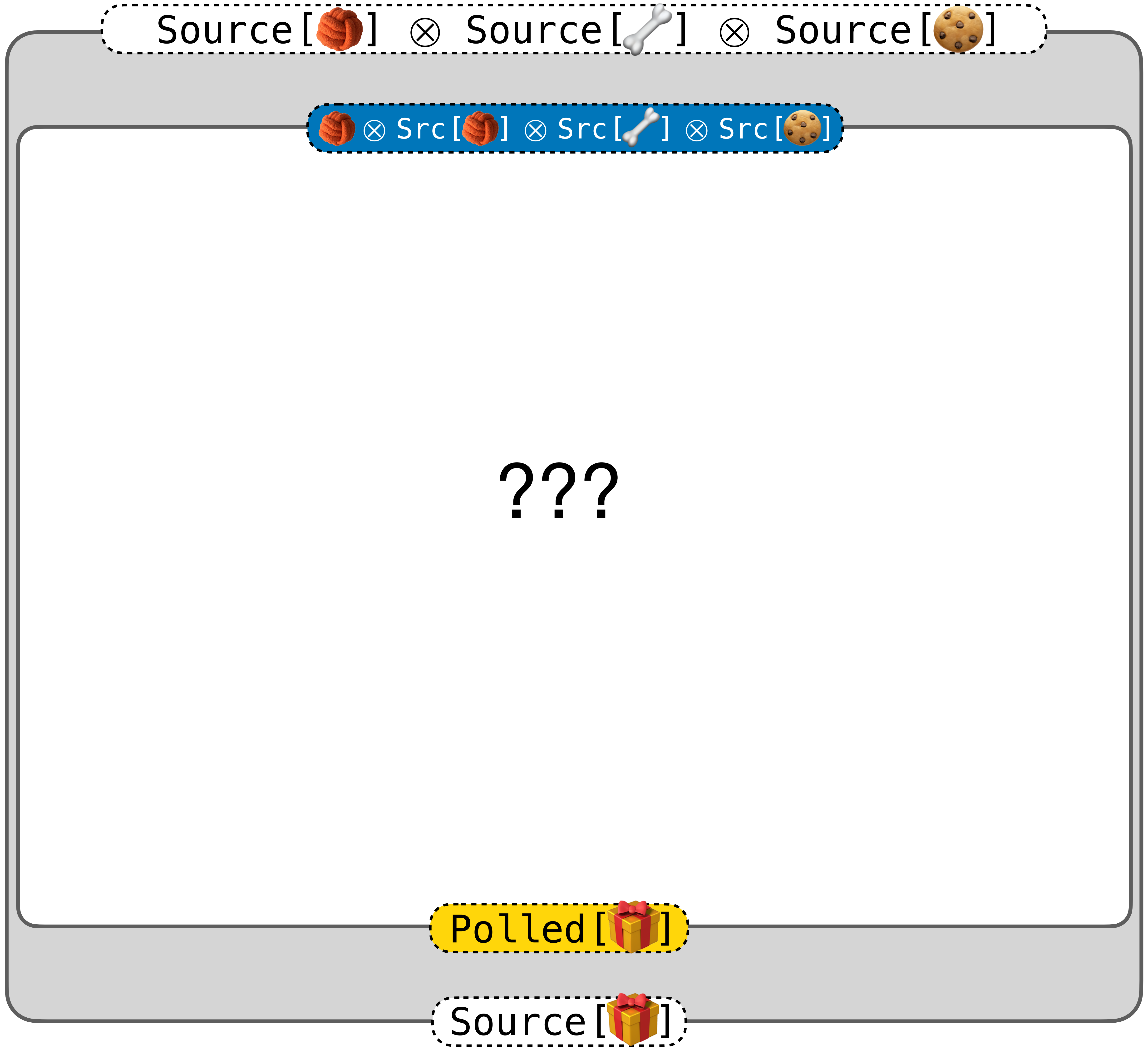
???

Polled[🎁]

Source[🎁]

???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	

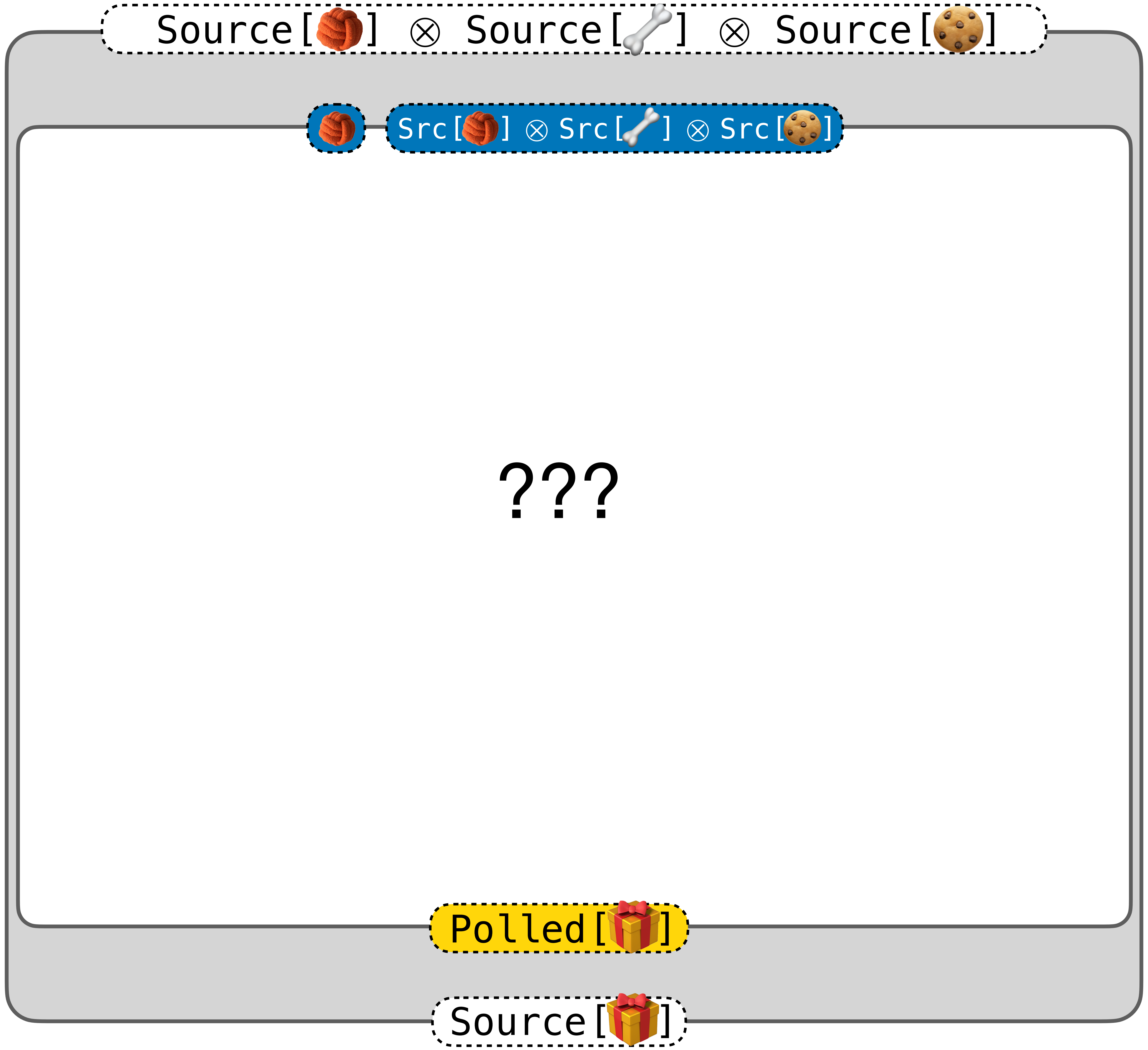
Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]
🧶 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]
???
Polled[🎁]
Source[🎁]

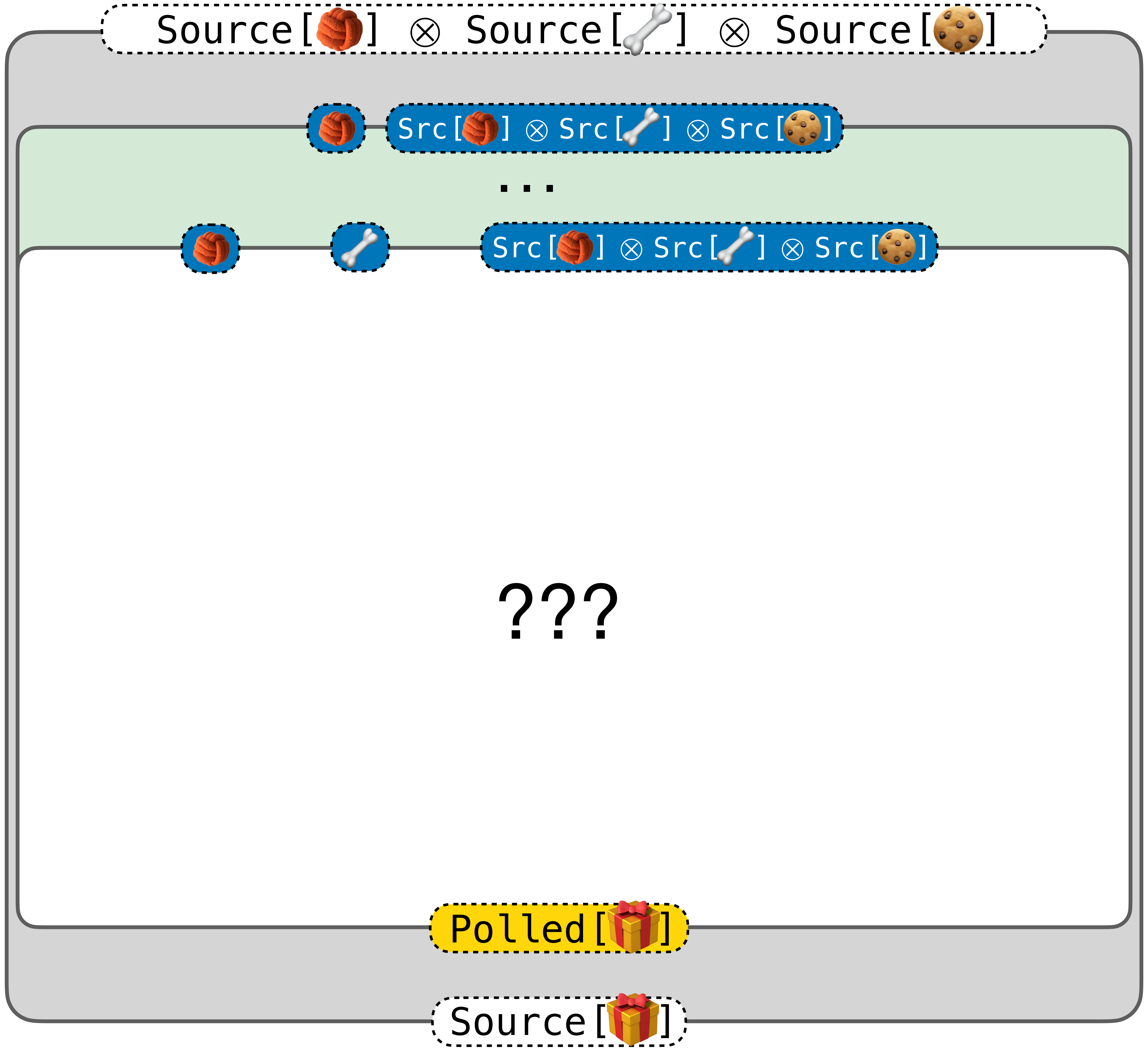
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	✔ ⊕ (A ⊗ Source[A])
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents



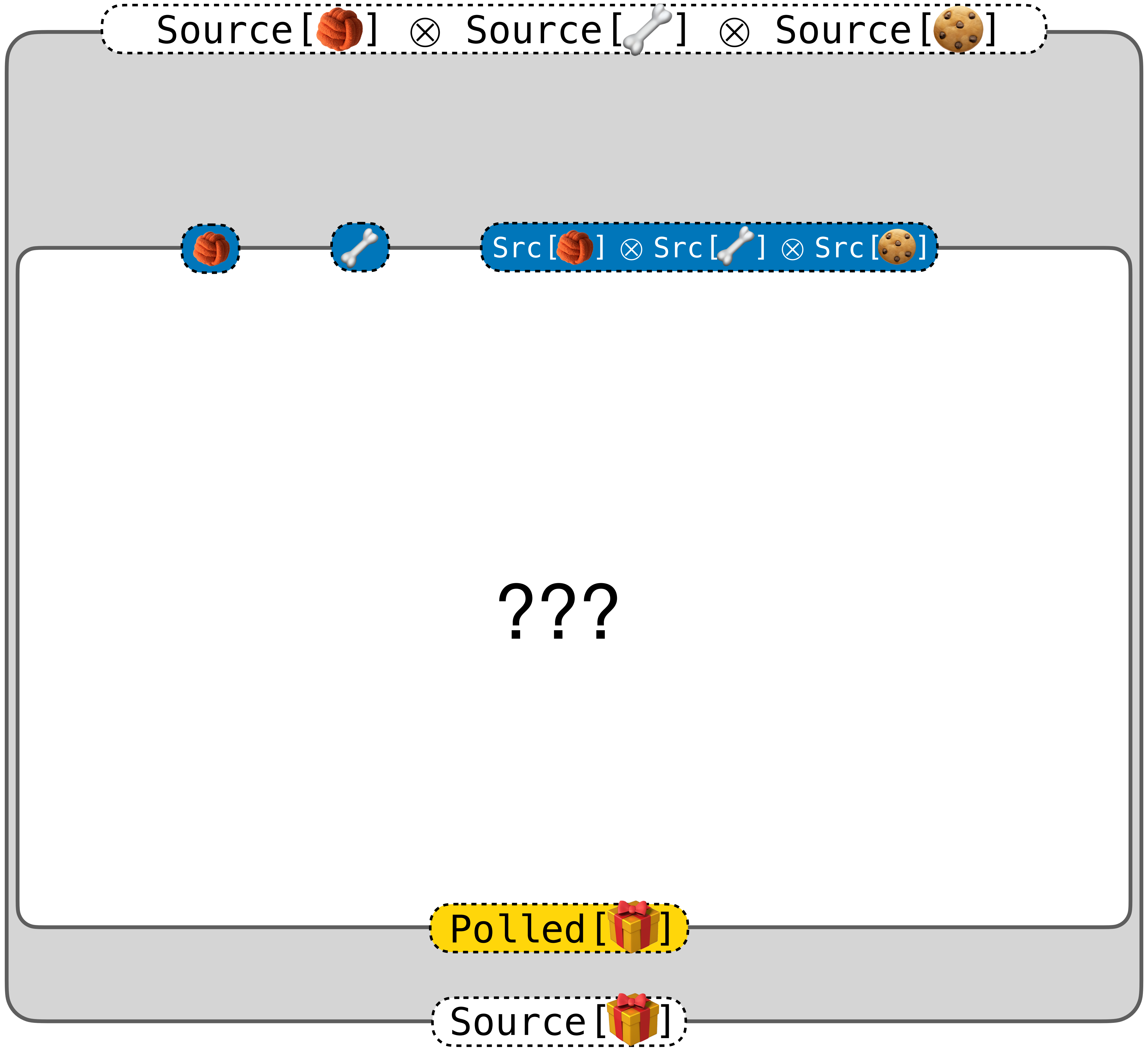
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents



???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

🧶

🦴

Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]

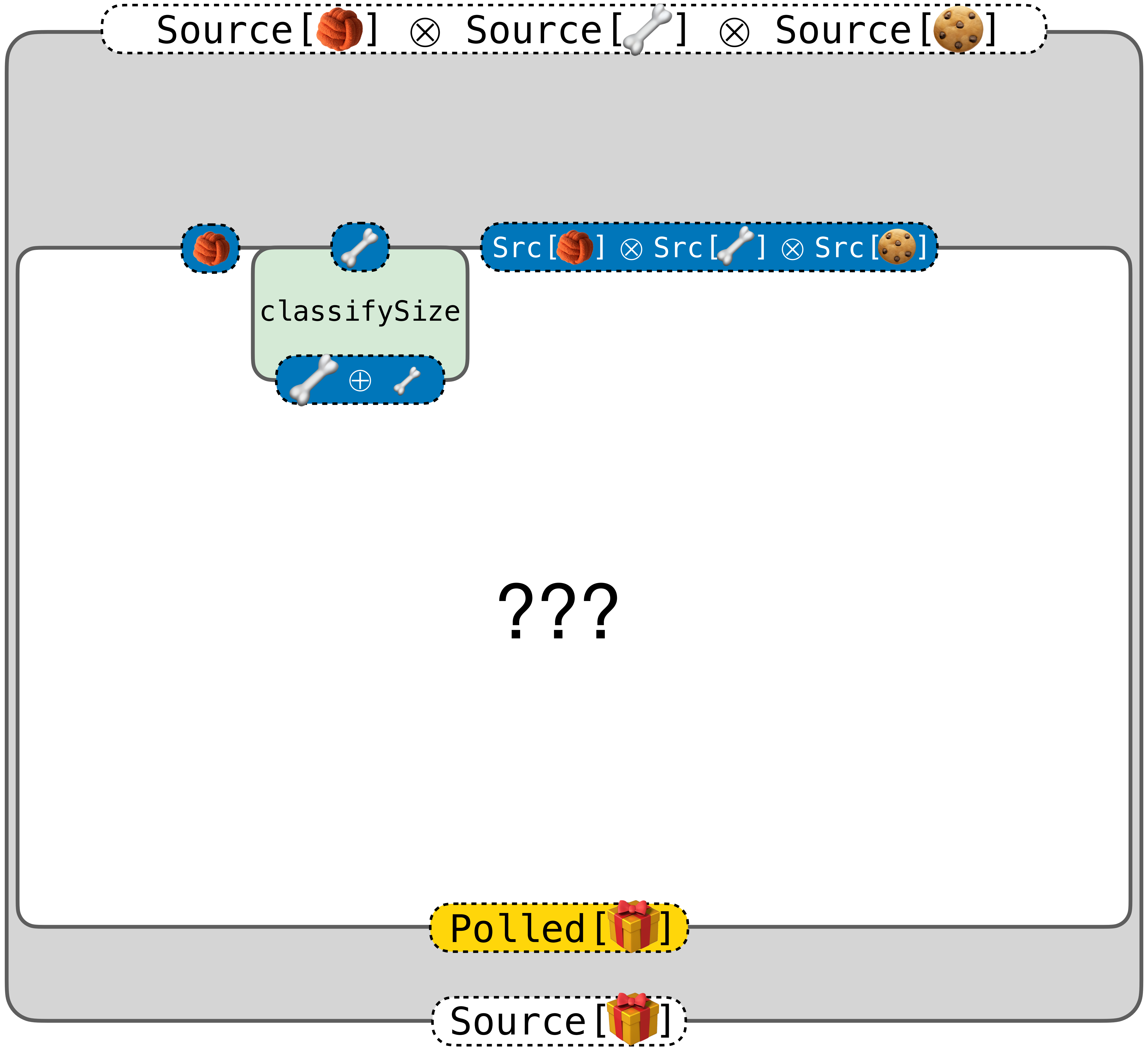
???

Polled[🎁]

Source[🎁]

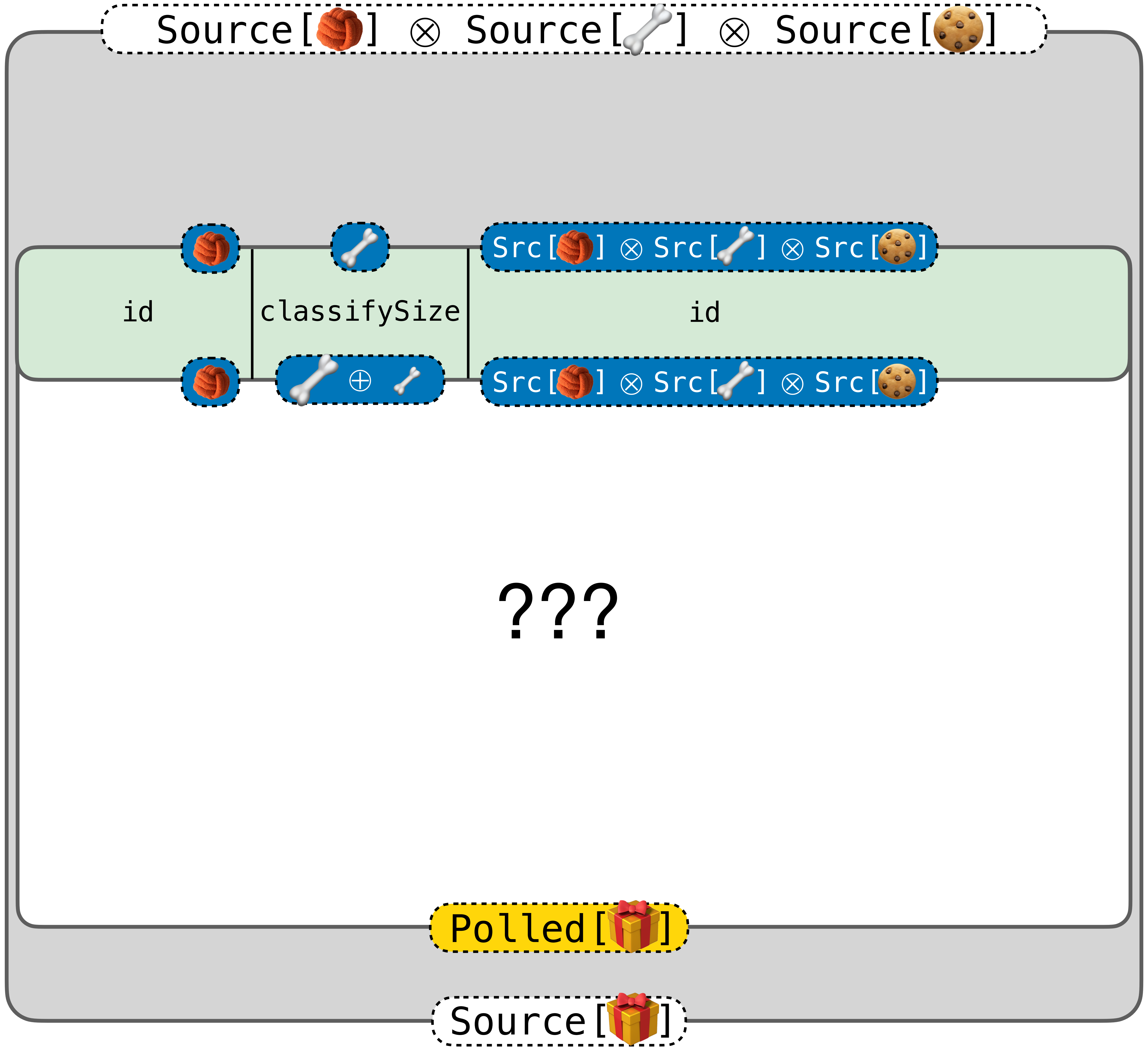
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents



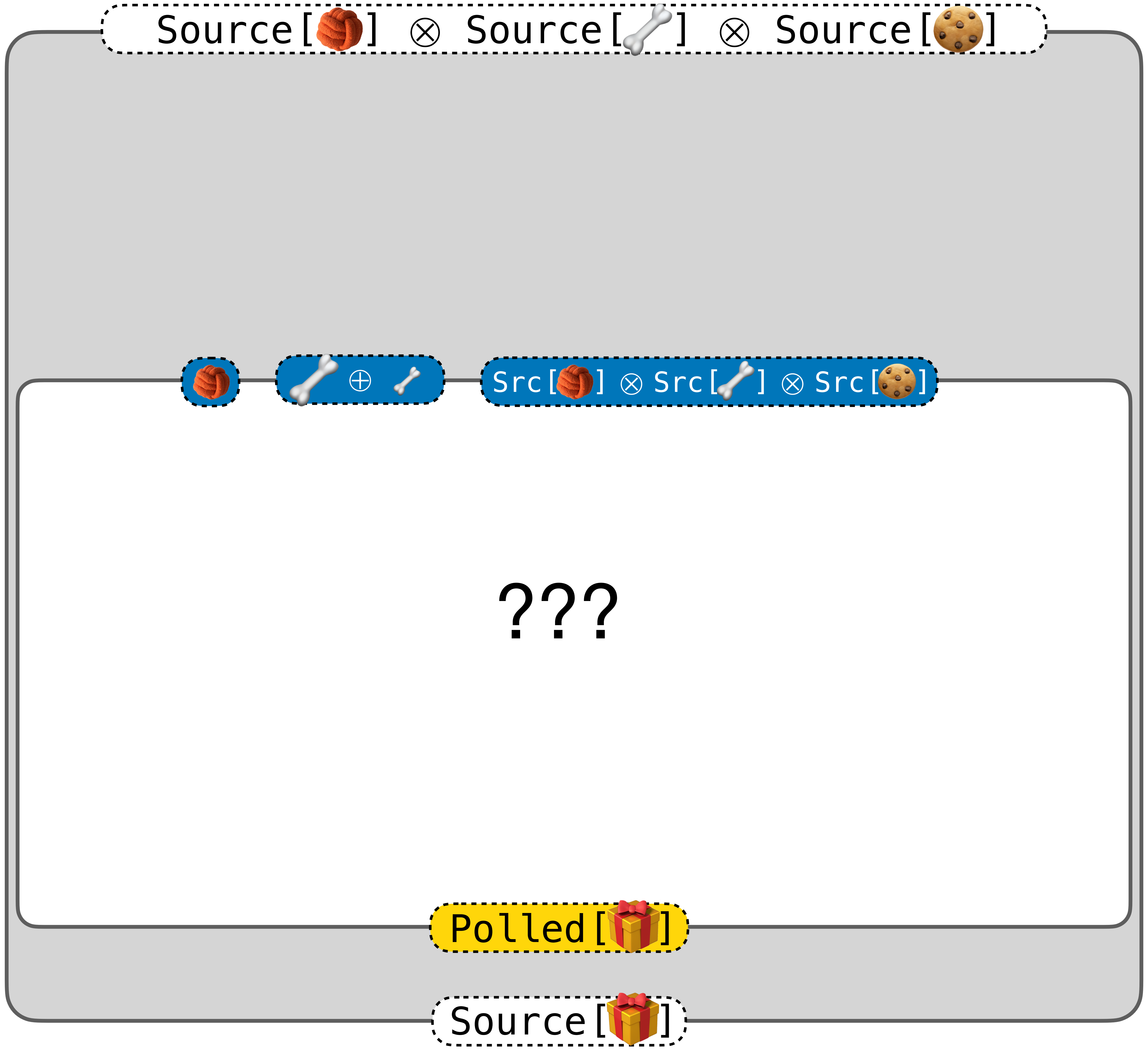
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents



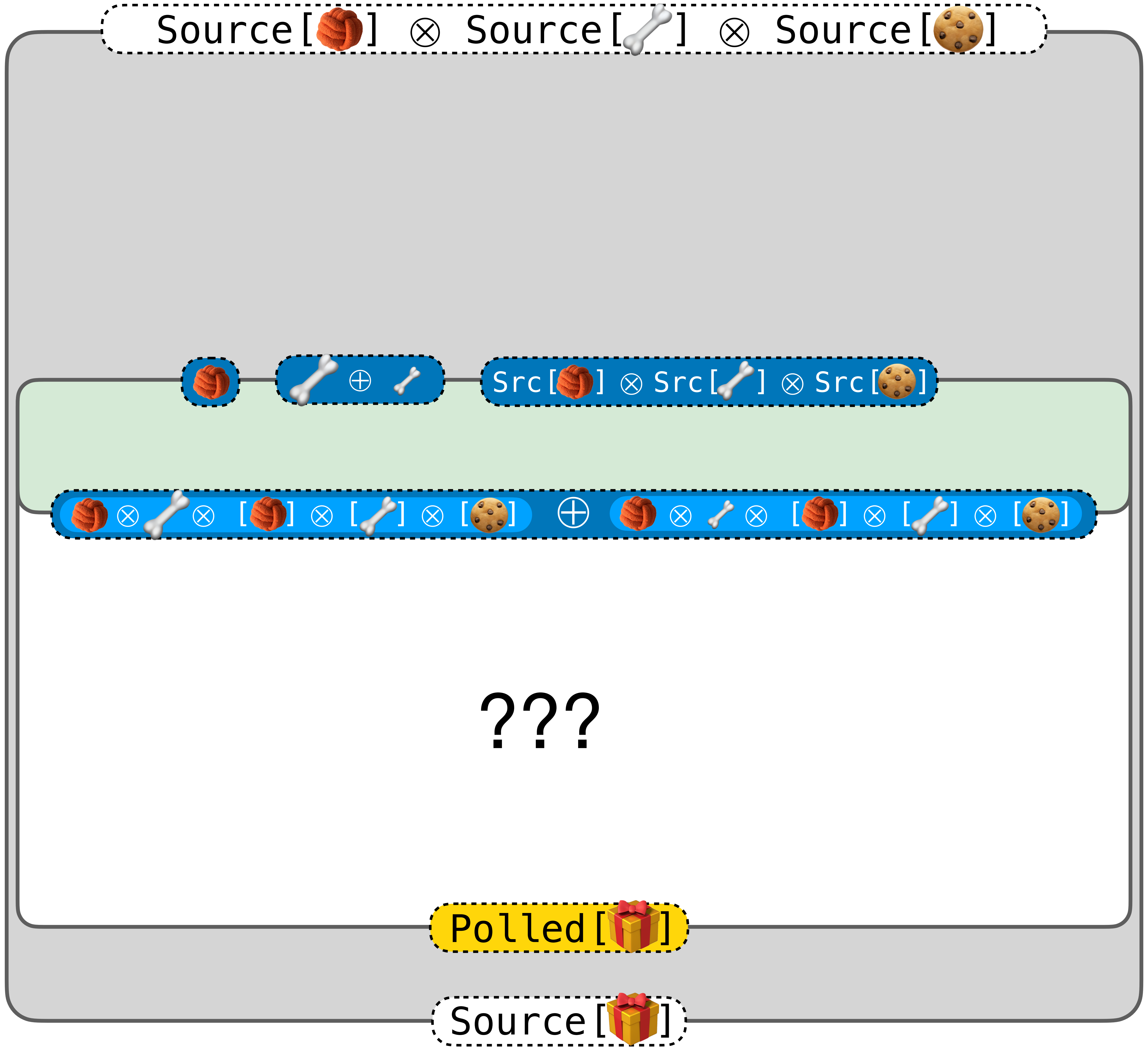
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	

Packaging Dog Presents



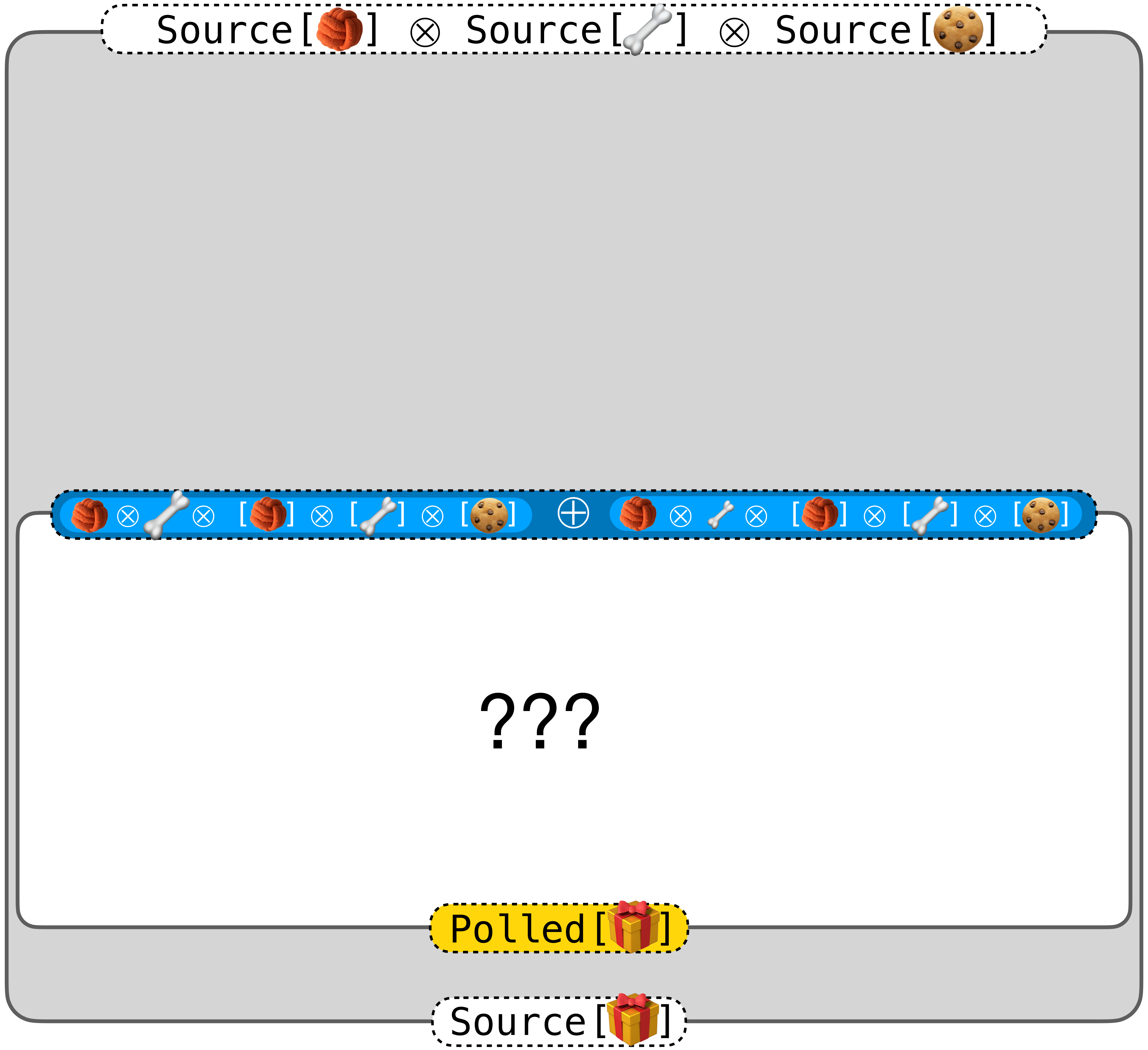
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
Polled[A]	requested next elem	✔ ⊕ (A ⊗ Source[A])
Src[A]	abbr. Source[A]	

Packaging Dog Presents



???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
Polled[A]	requested next elem	✔ ⊕ (A ⊗ Source[A])
Src[A]	abbr. Source[A]	
[A]	abbr. Source[A]	

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

🧶 ⊗ 🦴 ⊗ [🧶] ⊗ [🦴] ⊗ [🍪] ⊕ 🧶 ⊗ 🦴 ⊗ [🧶] ⊗ [🦴] ⊗ [🍪]

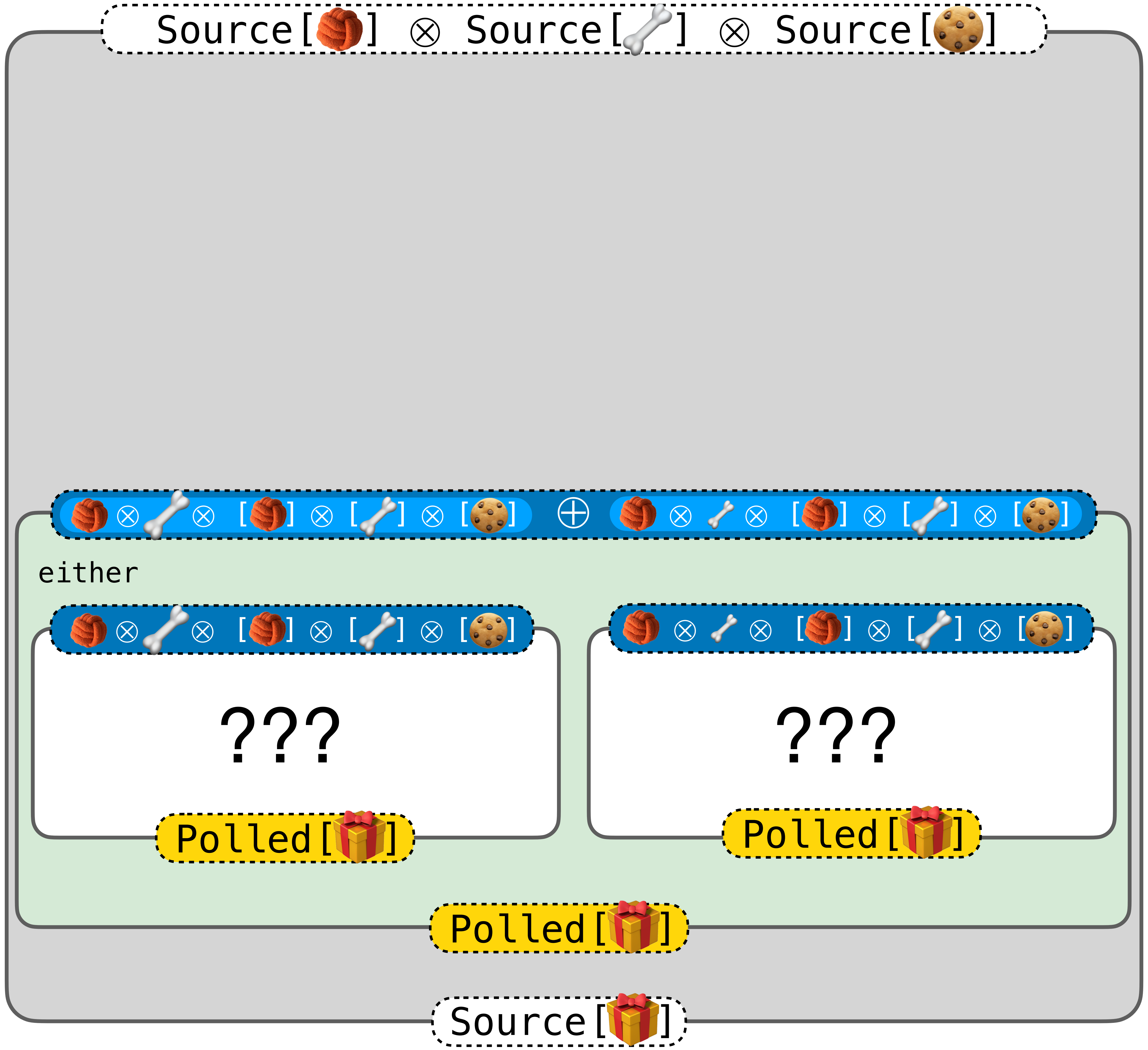
???

Polled[🎁]

Source[🎁]

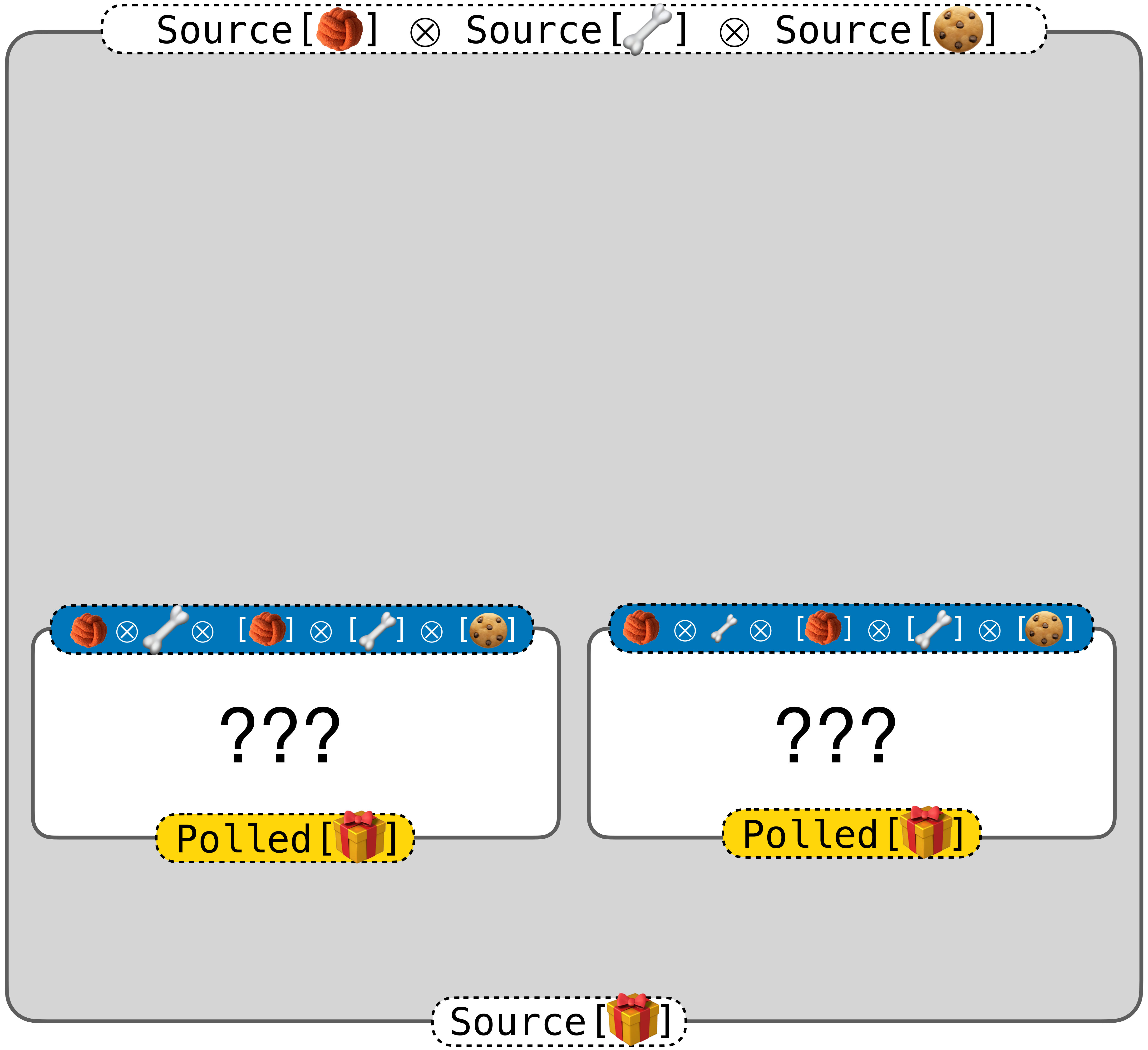
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	
`[A]`	abbr. `Source[A]`	

Packaging Dog Presents



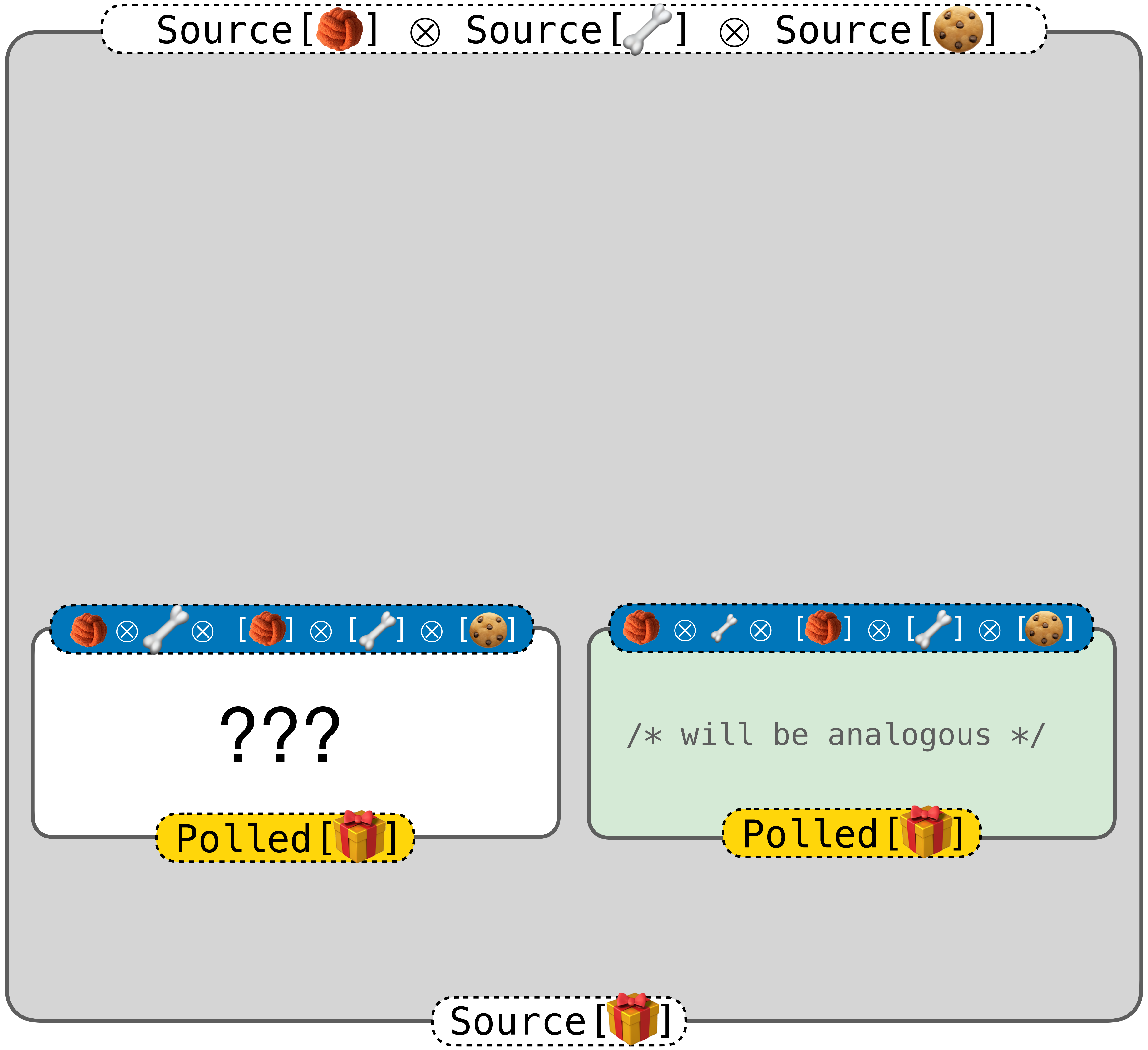
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	
`[A]`	abbr. `Source[A]`	

Packaging Dog Presents



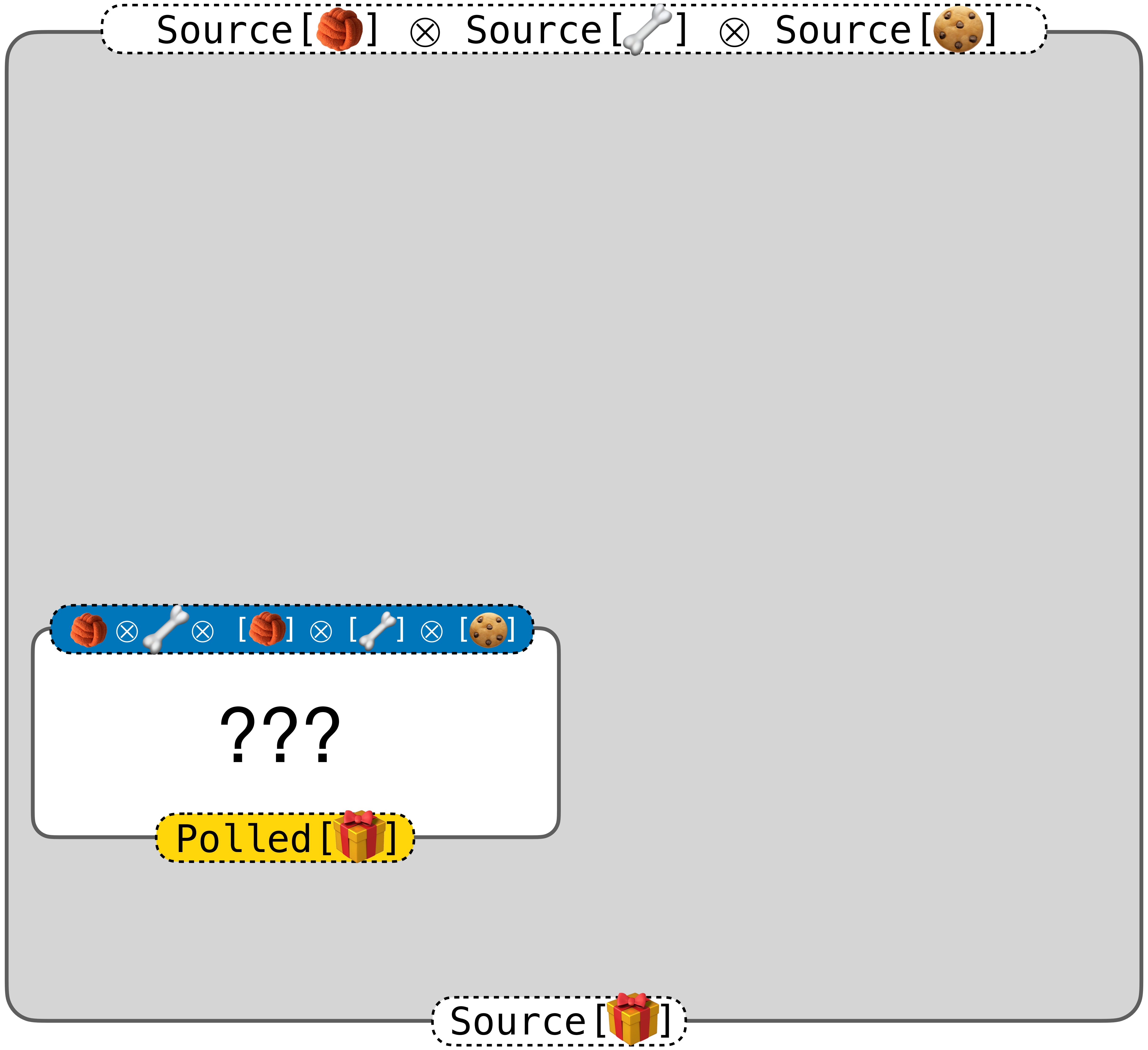
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
Polled[A]	requested next elem	✔ ⊕ (A ⊗ Source[A])
Src[A]	abbr. Source[A]	
[A]	abbr. Source[A]	

Packaging Dog Presents



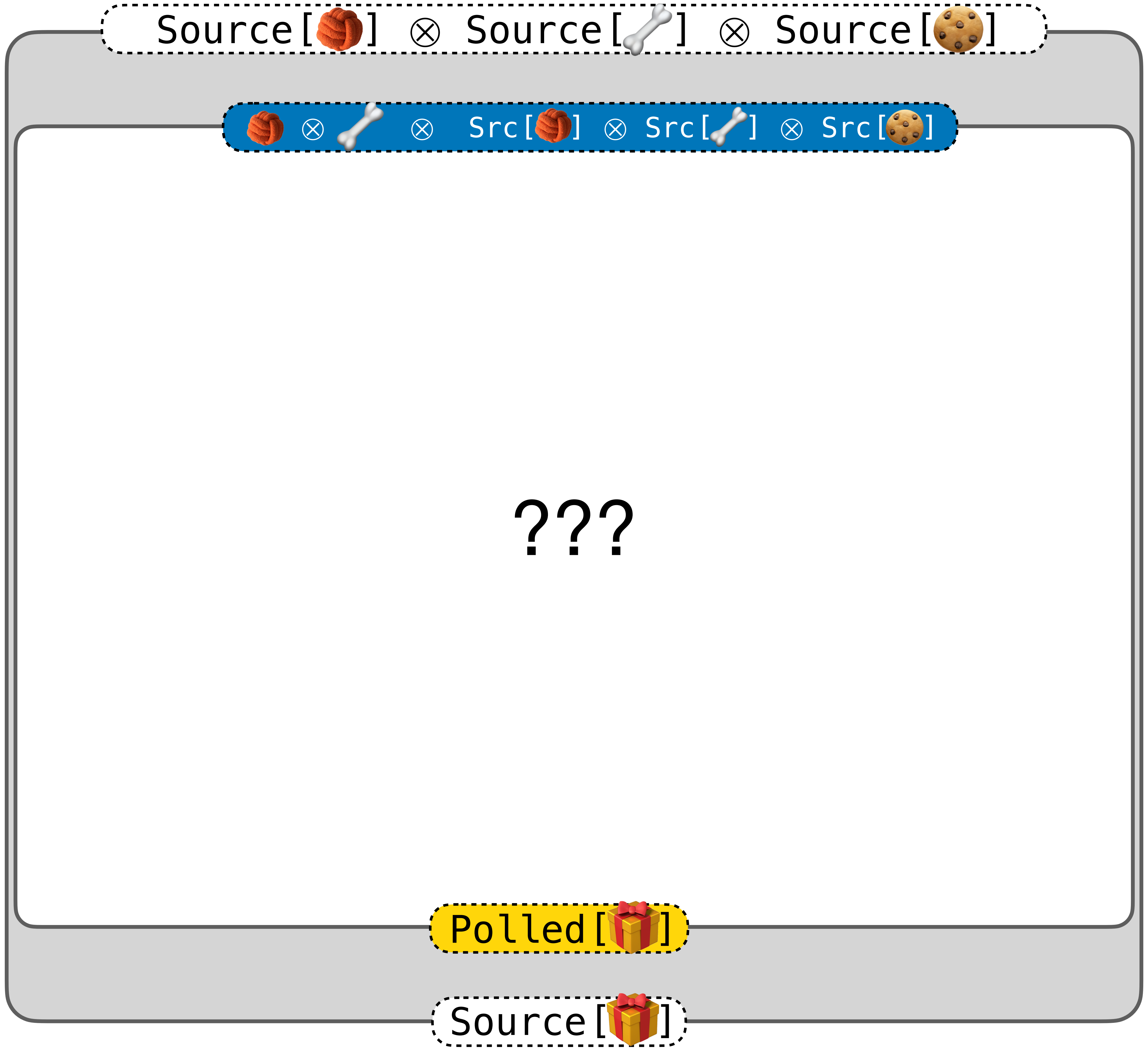
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	
`[A]`	abbr. `Source[A]`	

Packaging Dog Presents



???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
Polled[A]	requested next elem	✔ ⊕ (A ⊗ Source[A])
Src[A]	abbr. Source[A]	
[A]	abbr. Source[A]	

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

🧶 ⊗ 🦴 ⊗ Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]

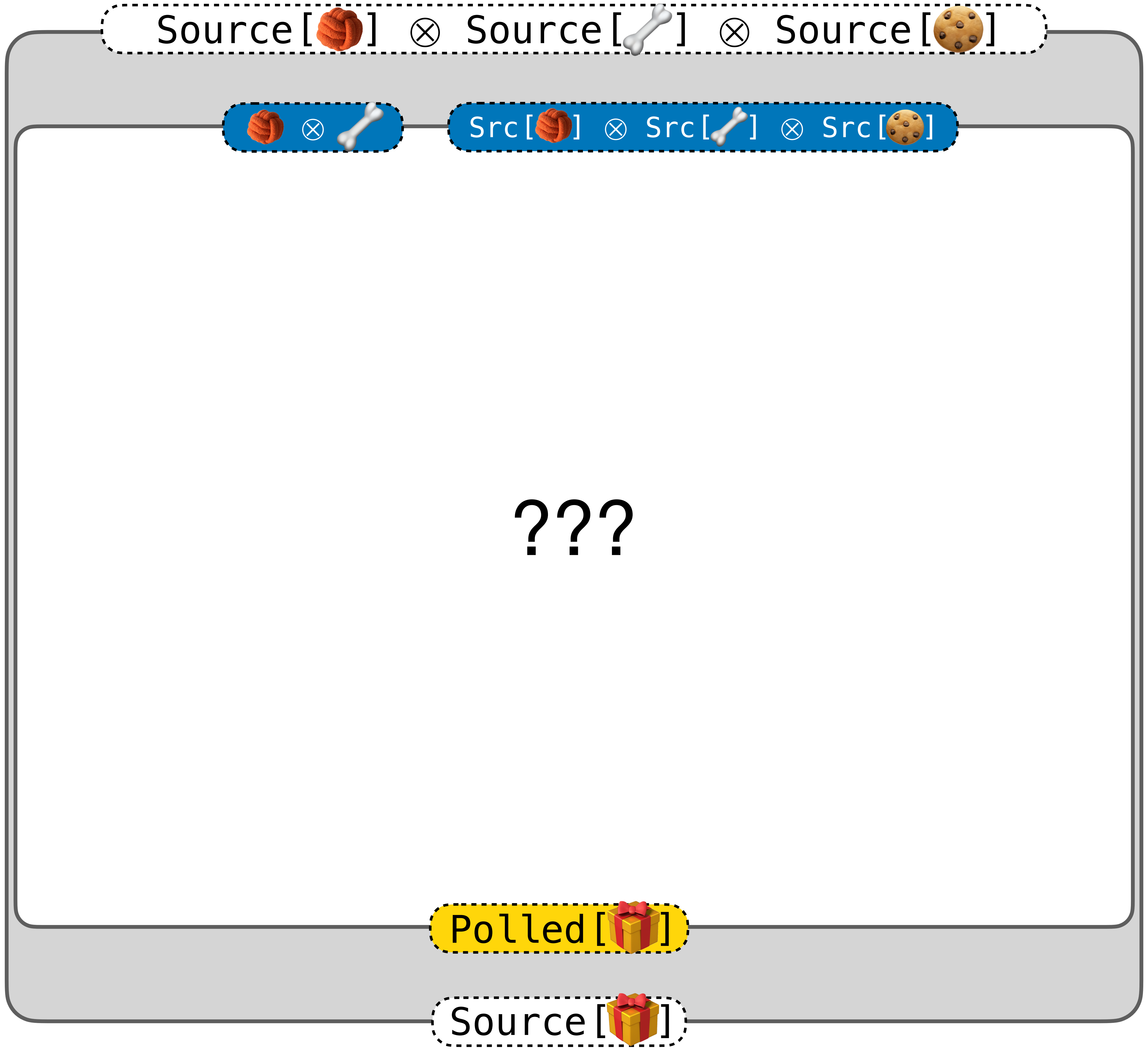
???

Polled[🎁]

Source[🎁]

???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
Polled[A]	requested next elem	✔ ⊕ (A ⊗ Source[A])
Src[A]	abbr. Source[A]	
[A]	abbr. Source[A]	

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

🧶 ⊗ 🦴

Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]

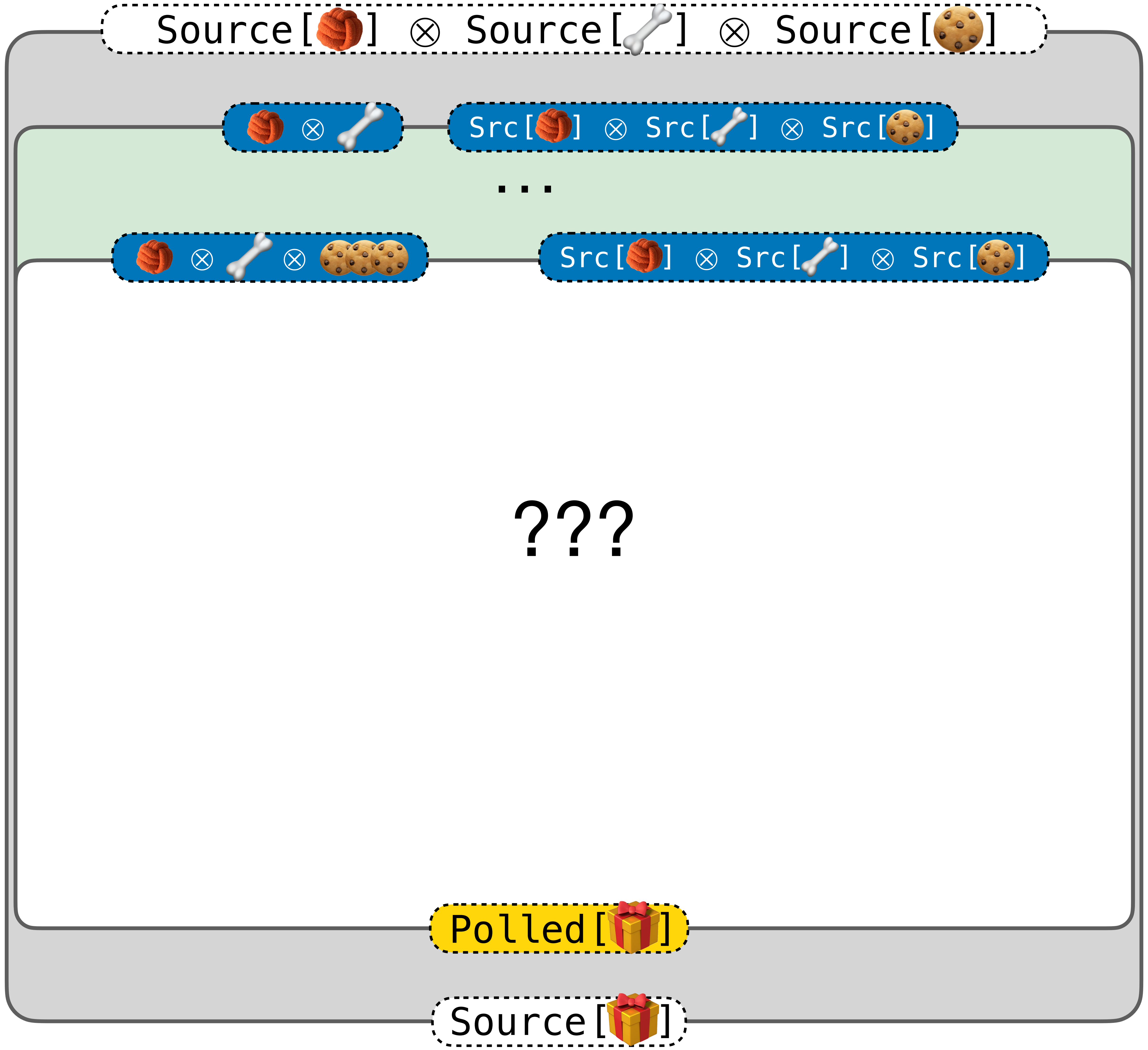
???

Polled[🎁]

Source[🎁]

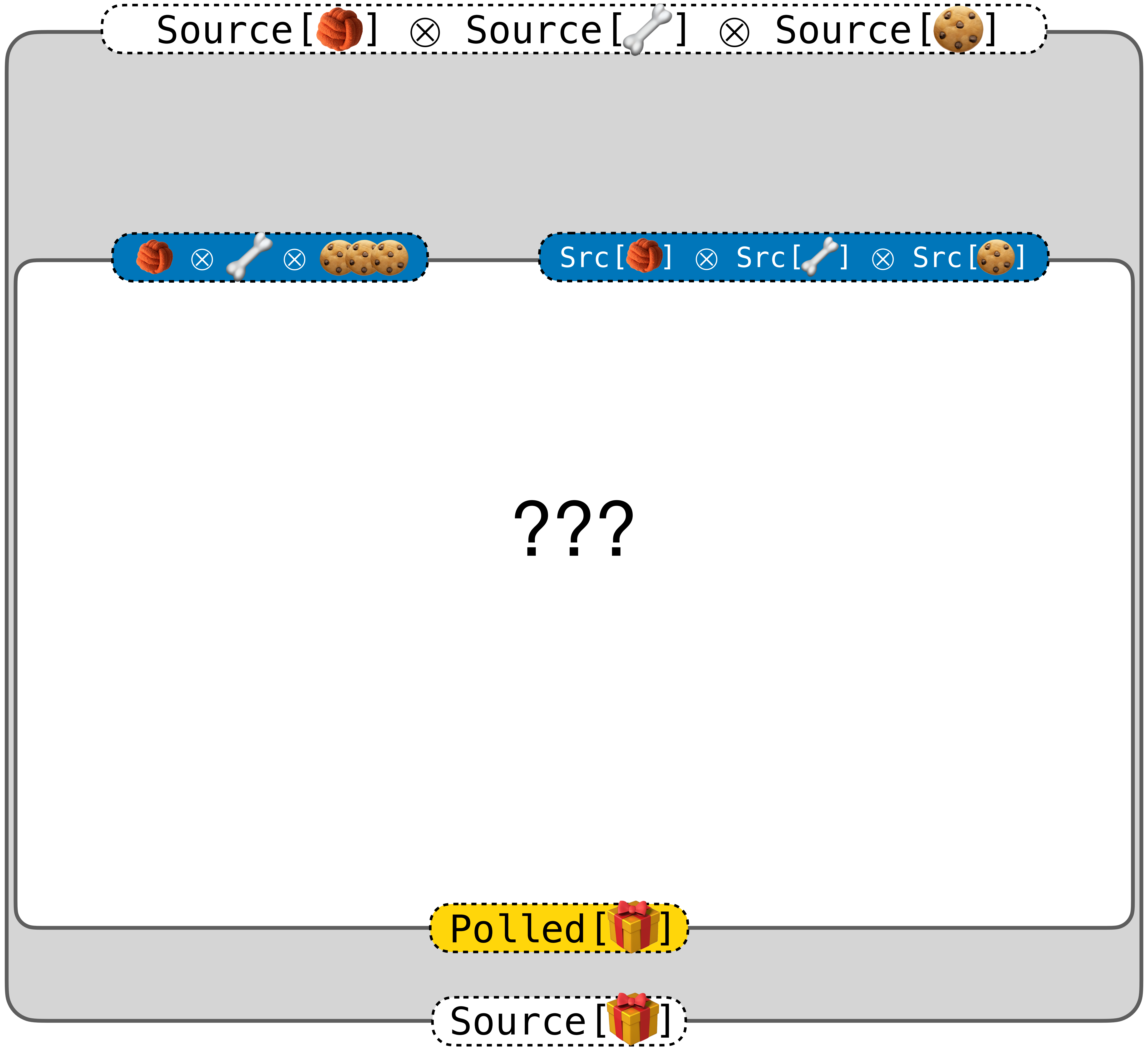
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
Polled[A]	requested next elem	✔ ⊕ (A ⊗ Source[A])
Src[A]	abbr. Source[A]	
[A]	abbr. Source[A]	

Packaging Dog Presents



???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	
`[A]`	abbr. `Source[A]`	

Packaging Dog Presents



Source[🧶] ⊗ Source[🦴] ⊗ Source[🍪]

🧶 ⊗ 🦴 ⊗ 🍪🍪🍪

Src[🧶] ⊗ Src[🦴] ⊗ Src[🍪]

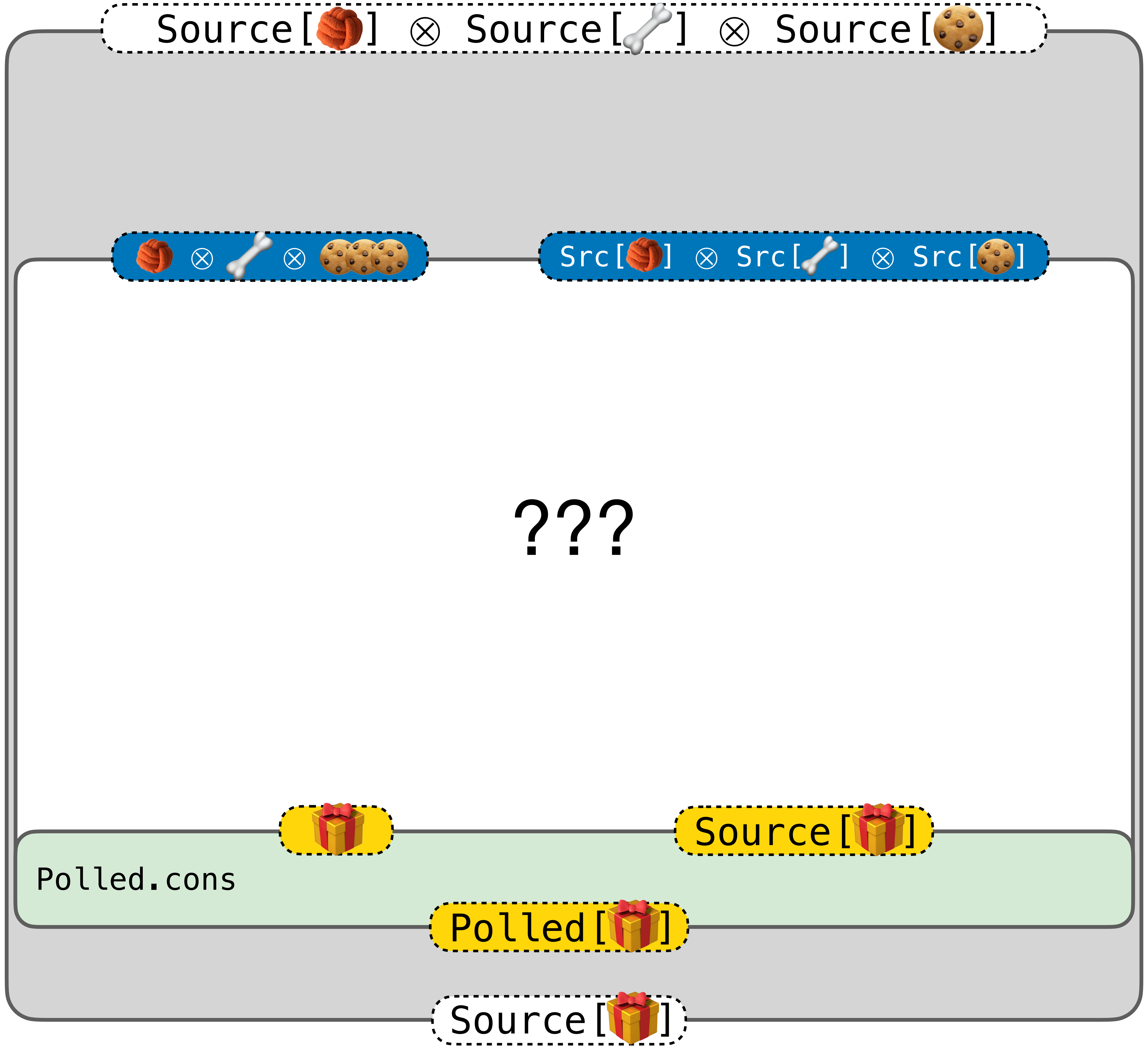
???

Polled[🎁]

Source[🎁]

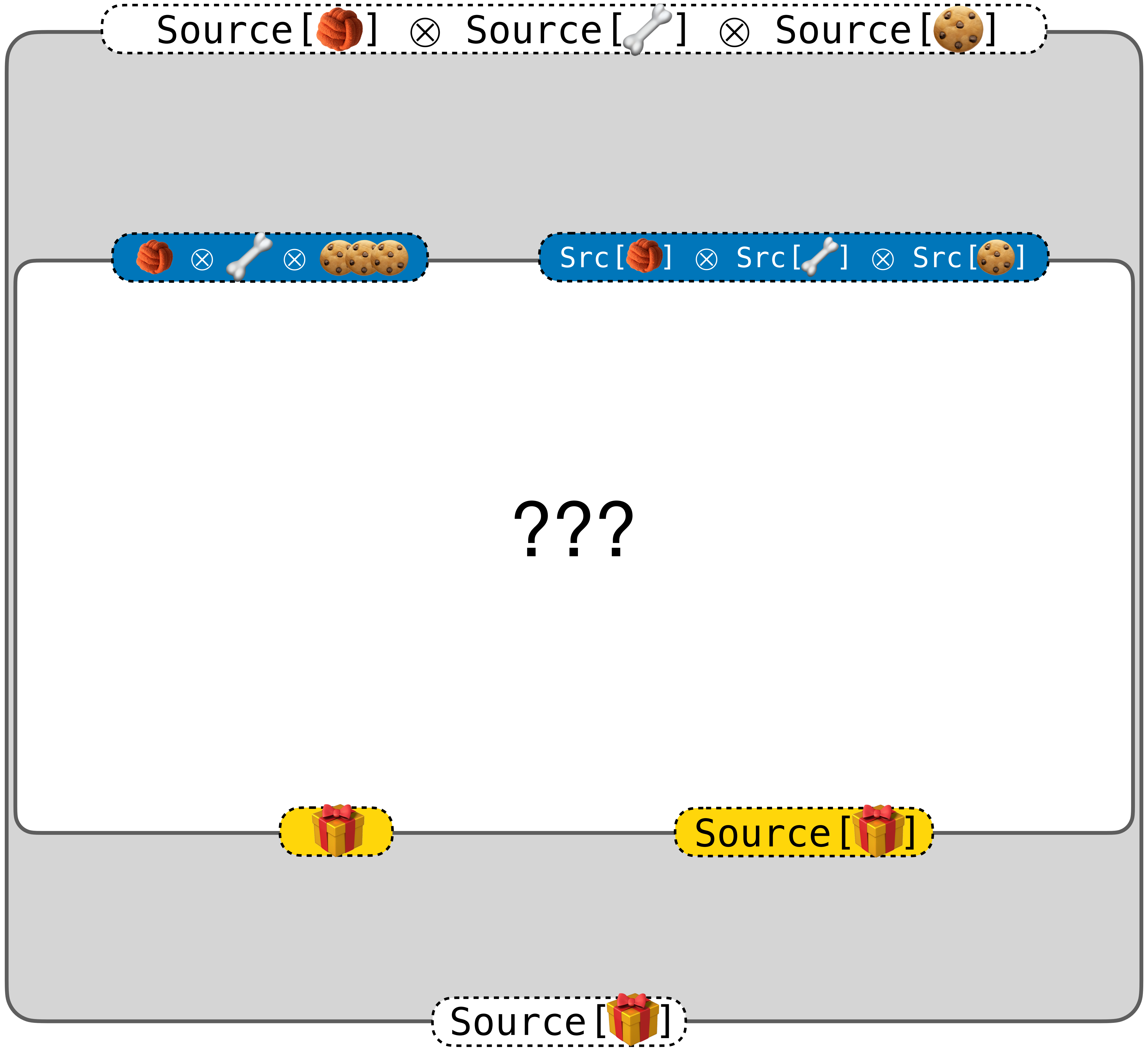
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	
`[A]`	abbr. `Source[A]`	

Packaging Dog Presents



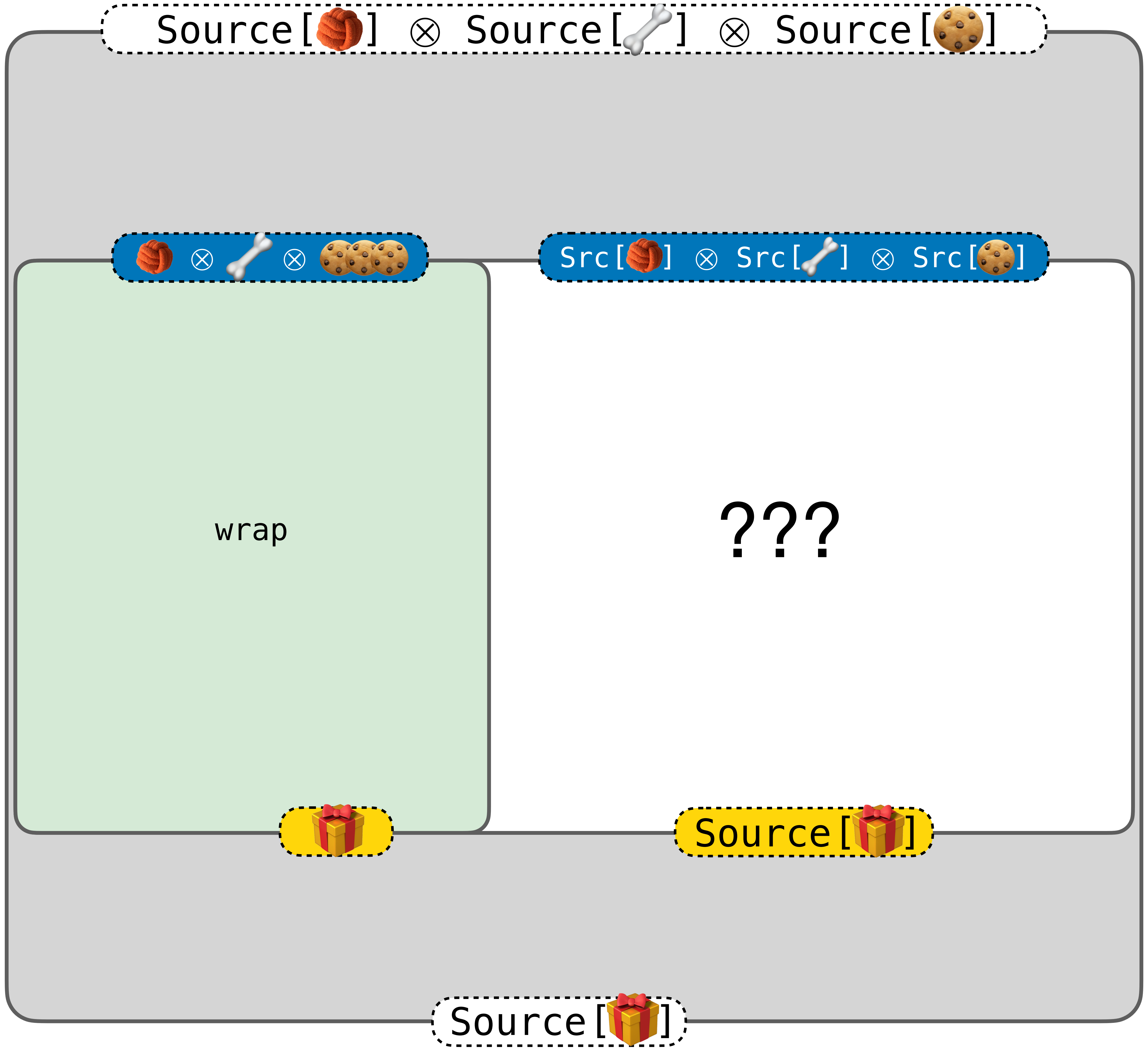
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	
`[A]`	abbr. `Source[A]`	

Packaging Dog Presents



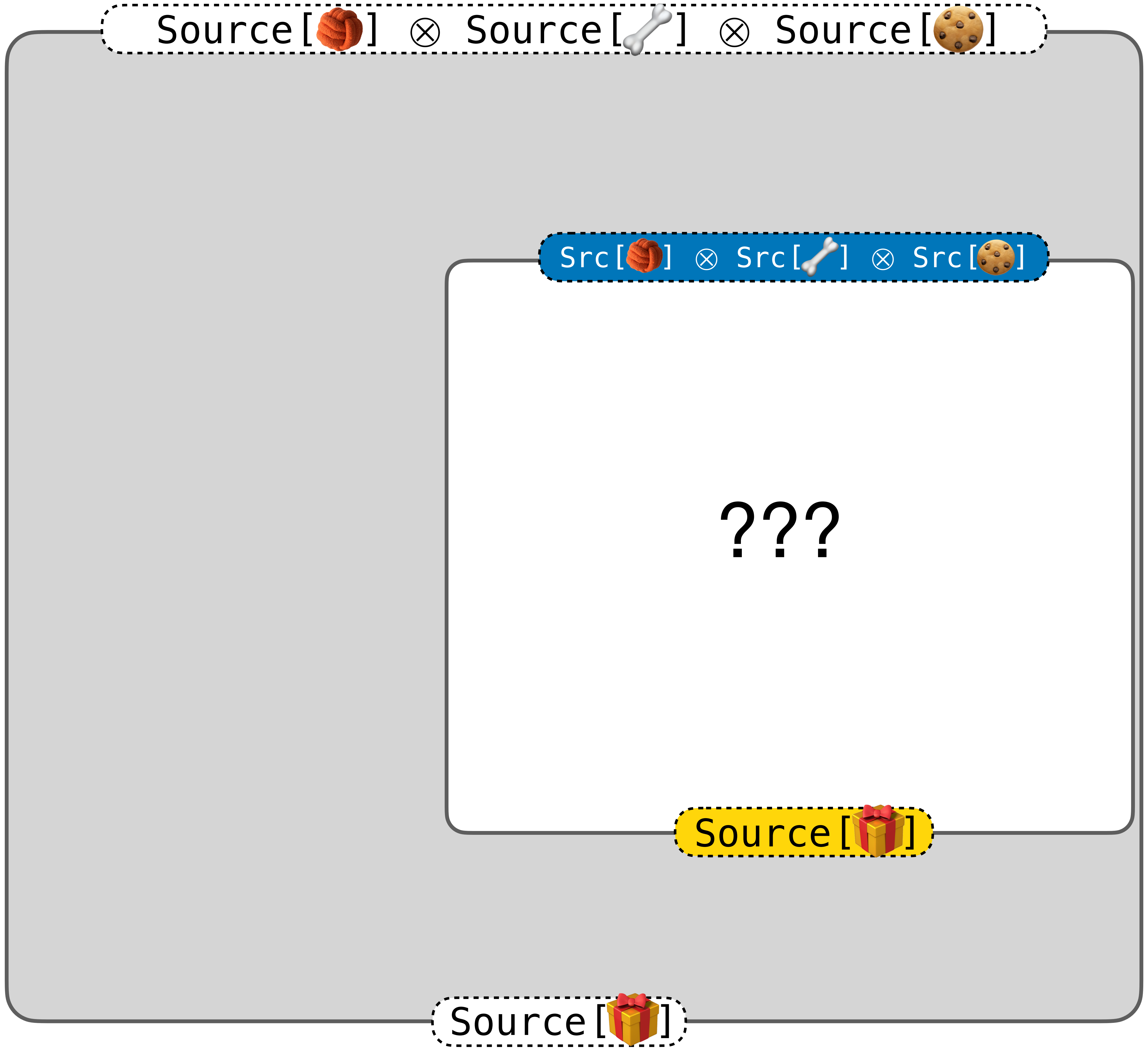
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	
`[A]`	abbr. `Source[A]`	

Packaging Dog Presents



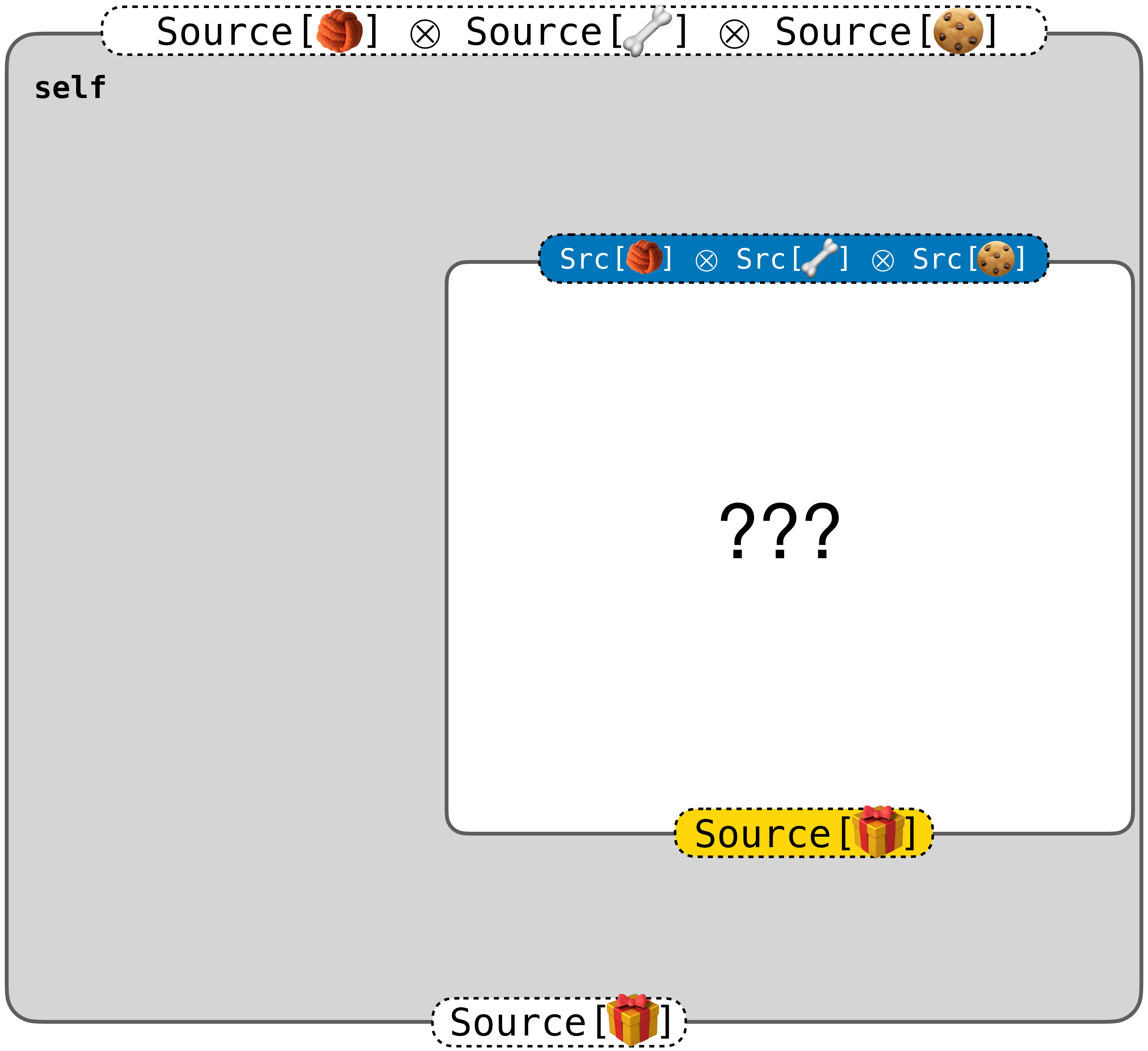
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
Polled[A]	requested next elem	✔ ⊕ (A ⊗ Source[A])
Src[A]	abbr. Source[A]	
[A]	abbr. Source[A]	

Packaging Dog Presents



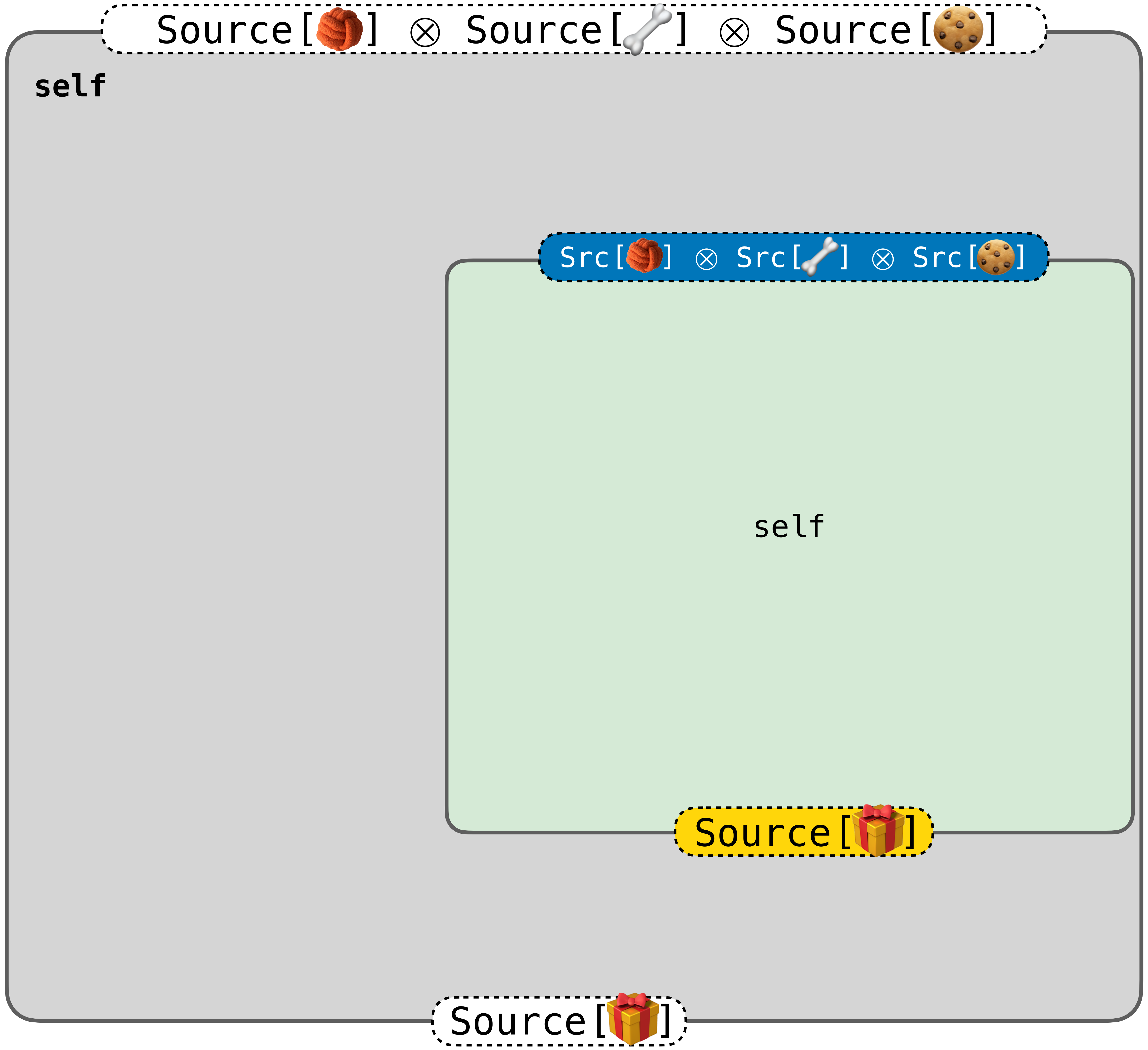
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	
`[A]`	abbr. `Source[A]`	

Packaging Dog Presents



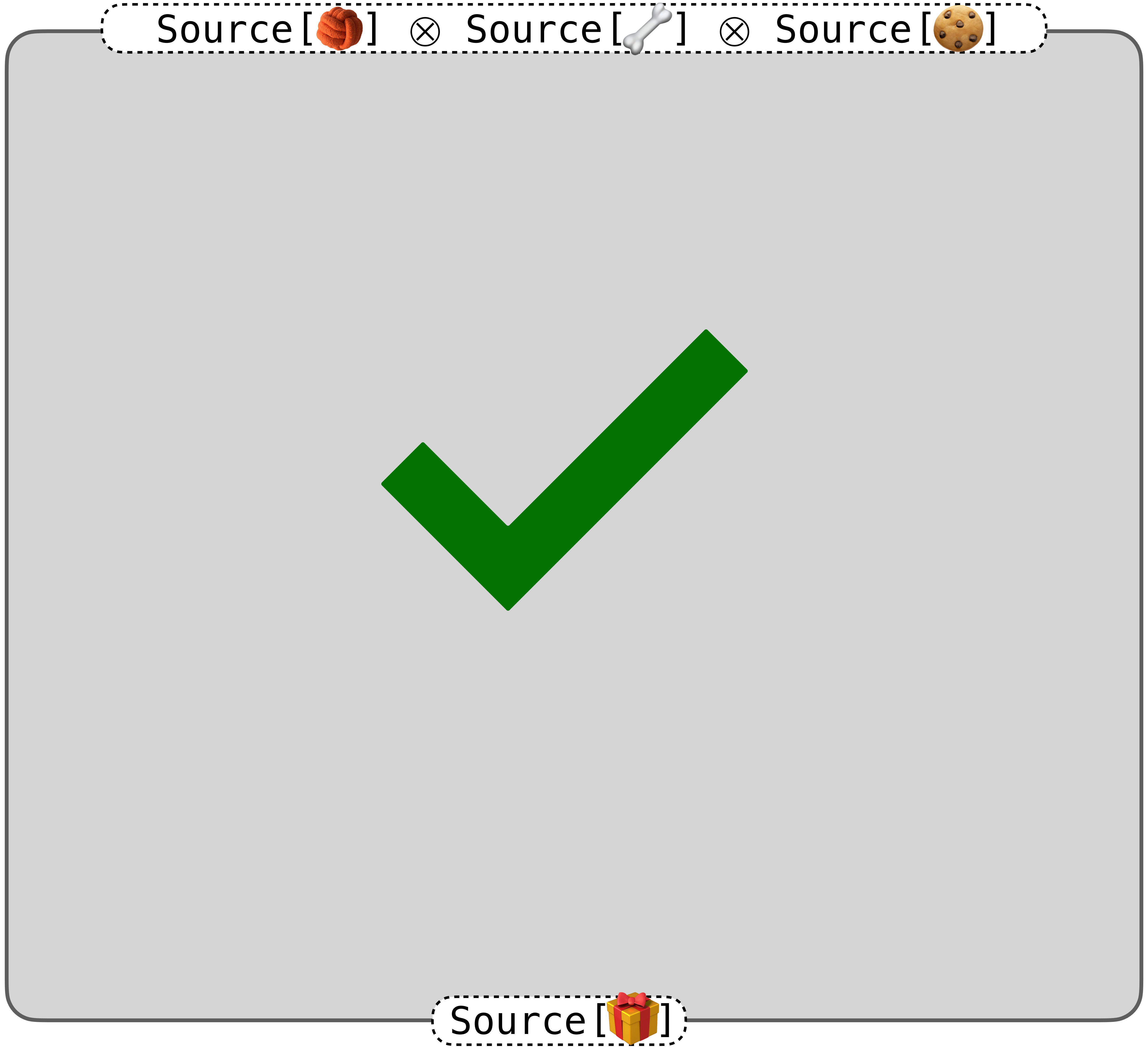
???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	`Done` signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	
`[A]`	abbr. `Source[A]`	

Packaging Dog Presents



???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	✔ ⊕ (A ⊗ Source[A])
`Src[A]`	abbr. `Source[A]`	
`[A]`	abbr. `Source[A]`	

Packaging Dog Presents



???	hole to be filled	
🟦	to be consumed	
🟨	to be produced	
⊗	concurrent pair	
&	consumer choice	
⊕	producer choice	
✔	Done signal	
`Polled[A]`	requested next elem	`✔ ⊕ (A ⊗ Source[A])`
`Src[A]`	abbr. `Source[A]`	
`[A]`	abbr. `Source[A]`	

Packaging Dog Presents

```
def packagingLine: (Source[Toy] |*| Source[Bone] |*| Source[Biscuit]) -o Source[Present] =
  ???
```

Packaging Dog Presents

```
def packagingLine: (Source[Toy] |*| Source[Bone] |*| Source[Biscuit]) -o Source[Present] =
  ???
```

Packaging Dog Presents

```
def packagingLine: (Source[Toy] |*| Source[Bone] |*| Source[Biscuit]) -⚬ Source[Present] =
  rec { self =>
    ???
  }
```

Packaging Dog Presents

```
def packagingLine: (Source[Toy] |*| Source[Bone] |*| Source[Biscuit]) -o Source[Present] =
  rec { self =>
    Source.from(
      onClose =
        λ { case (toys |*| bones |*| biscuits) =>
          ??? : $[✓]
        },

      onPoll =
        λ { case (toys |*| bones |*| biscuits) =>
          ??? : $[Polled[Present]]
        },
    )
  }
```

Packaging Dog Presents

```
def packagingLine: (Source[Toy] |*| Source[Bone] |*| Source[Biscuit]) -o Source[Present] =
  rec { self =>
    Source.from(
      onClose =
        λ { case (toys |*| bones |*| biscuits) =>
          ??? : $[✔]
        },

      onPoll =
        λ { case (toys |*| bones |*| biscuits) =>
          ??? : $[Polled[Present]]
        },
    )
  }
```

Packaging Dog Presents

```
def packagingLine: (Source[Toy] |*| Source[Bone] |*| Source[Biscuit]) -⚬ Source[Present] =
 rec { self =>
  Source.from(
   onClose =
    λ { case (toys |*| bones |*| biscuits) =>
     joinAll(close(toys), close(bones), close(biscuits))
    },

   onPoll =
    λ { case (toys |*| bones |*| biscuits) =>
     ??? : $[Polled[Present]]
    },
  )
 }
```

Packaging Dog Presents

```
λ { case (toys |*| bones |*| biscuits) =>
  ??? : $[Polled[Present]]
}
```

Packaging Dog Presents

```
λ { case (toys |*| bones |*| biscuits) =>
  poll(toys) switch {
    case Left( ✔ ) => // no toys left, still have bones and biscuits
      ??? : $[Polled[Present]]

    case Right(toy |*| toys) => // got a toy, still have bones and biscuits
      ??? : $[Polled[Present]]
  }
}
```

Packaging Dog Presents

```
λ { case (toys |*| bones |*| biscuits) =>
  poll(toys) switch {
    case Left( ✔ ) => // no toys left
      Polled.empty(joinAll( ✔ , close(bones), close(biscuits)))

    case Right(toy |*| toys) => // got a toy, still have bones and biscuits
      ??? : $[Polled[Present]]
  }
}
```

Packaging Dog Presents

```
λ { case (toys |*| bones |*| biscuits) =>
 poll(toys) switch {
  case Left( ✔ ) => // no toys left
   Polled.empty(joinAll( ✔ , close(bones), close(biscuits)))

  case Right(toy |*| toys) => // got a toy, still have biscuits
   poll(bones) switch {
    case Left( ✔ ) => // no bones left
     Polled.empty(joinAll( ✔ , neglect(toy), close(toys), close(biscuits)))
    case Right(bone |*| bones) => // got a bone, still have toy, toys, biscuits
     ??? : $[Polled[Present]]
   }
 }
}
```

Packaging Dog Presents

```
case Right(bone |*| bones) => // got a bone, still have toy, toys, biscuits
  ??? : $[Polled[Present]]
```

Packaging Dog Presents

```
case Right(bone |*| bones) => // got a bone
  Bone.classifySize(bone) switch {
    case Left(largeBone) => // got a large bone
      pullThreeBiscuits(biscuits) switch {
        case Left( ✔ ) => // not enough biscuits
          Polled.empty(joinAll(✔, neglect(toy), neglect(largeBone), close(toys), close(bones)))
        case Right(biscuit3 |*| biscuits) => // got three biscuits
          Polled.cons(
            wrap(toy, largeBone, biscuit3) |*|
            self(toys |*| bones |*| biscuits)
          )
      }
    case Right(smallBone) => // got a small bone
      // analogous
```

Packaging Dog Presents

```
case Right(bone |*| bones) => // got a bone
  Bone.classifySize(bone) switch {
    case Left(largeBone) => // got a large bone
      pullThreeBiscuits(biscuits) switch {
        case Left( ✔ ) => // not enough biscuits
          Polled.empty(joinAll(✔, neglect(toy), neglect(largeBone), close(toys), close(bones)))
        case Right(biscuit3 |*| biscuits) => // got three biscuits
          Polled.cons(
            wrap(toy, largeBone, biscuit3) |*|
            self(toys |*| bones |*| biscuits)
          )
      }
    case Right(smallBone) => // got a small bone
      // analogous
```

✔

Packaging Dog Presents

```
case Right(bone |*| bones) => // got a bone
 Bone.classifySize(bone) switch {
  case Left(largeBone) => // got a large bone
   pullThreeBiscuits(biscuits) switch {
    case Left( ✔ ) => // not enough biscuits
     Polled.empty(joinAll(✔, neglect(toy),          close(toys), close(bones)))
    case Right(biscuit3 |*| biscuits) => // got three biscuits
     Polled.cons(
      wrap(toy, largeBone, biscuit3) |*|
      self(toys |*| bones |*| biscuits)
     )
   }
  case Right(smallBone) => // got a small bone
   // analogous
```

Unused variable `largeBone`

Packaging Dog Presents

```
case Right(bone |*| bones) => // got a bone
  Bone.classifySize(bone) switch {
    case Left(largeBone) => // got a large bone
      pullThreeBiscuits(biscuits) switch {
        case Left( ✔ ) => // not enough biscuits
          Polled.empty(joinAll(✔, neglect(toy),        close(toys), close(toys), close(bones)))
        case Right(biscuit3 |*| biscuits) => // got three biscuits
          Polled.cons(
            wrap(toy, largeBone, biscuit3) |*|
            self(toys |*| bones |*| biscuits)
          )
      }
    case Right(smallBone) => // got a small bone
      // analogous
```

Unused variable `largeBone`

Overused variable `toys`

Packaging Dog Presents

```
case Right(bone |*| bones) => // got a bone
  Bone.classifySize(bone) switch {
    case Left(largeBone) => // got a large bone
      pullThreeBiscuits(biscuits) switch {
        case Left( ✔ ) => // not enough biscuits
          Polled.empty(joinAll(✔, neglect(toy),        close(toys), close(toys), close(bones)))
        case Right(biscuit3 |*| biscuits) => // got three biscuits
          Polled.cons(
            wrap(toy, largeBone, biscuit3) |*|
            self(toys |*| bones |*| biscuits)
          )
      }
    case Right(smallBone) => // got a small bone
      // analogous
```

Unused variable `largeBone`

Overused variable `toys`

Not properly wired ⇒ **unrepresentable**

- exception from the surrounding λ
- *assembly-time* error

Packaging Dog Presents

```
case Right(bone |*| bones) => // got a bone
  Bone.classifySize(bone) switch {
    case Left(largeBone) => // got a large bone
      pullThreeBiscuits(biscuits) switch {
        case Left( ✔ ) => // not enough biscuits
          Polled.empty(joinAll(✔, neglect(toy),        close(toys), close(toys), close(bones)))
        case Right(biscuit3 |*| biscuits) => // got three biscuits
          Polled.cons(
            wrap(toy, largeBone, biscuit3) |*|
            self(toys |*| bones |*| biscuits)
          )
      }
    case Right(smallBone) => // got a small bone
      // analogous
```

Unused variable `largeBone`

Overused variable `toys`

Not properly wired ⇒ **unrepresentable**

- exception from the surrounding λ
- *assembly-time* error

`test("packagingLine") { packagingLine }`

Packaging Dog Presents: Alternatives

Packaging Dog Presents: Alternatives

FS2’s `Stream.pull`

ZIO’s `ZStream.toPull`

Packaging Dog Presents: Alternatives

FS2’s `Stream.pull`

ZIO’s `ZStream.toPull`

- much less safe
- slightly more accidental complexity

Integrating with ZIO Streams

Libretto

```
def packagingLine: (Source[Toy] |*| Source[Bone] |*| Source[Biscuit]) -⚬ Source[Present]
```

Integrating with ZIO Streams

Libretto

```
def packagingLine: (Source[Toy] |*| Source[Bone] |*| Source[Biscuit]) -⚬ Source[Present]
```

ZIO

```
def go(
  toys: UStream[Toy],
  bones: UStream[Bone],
  biscuits: UStream[Biscuit],
): ZIO[Scope, Nothing, UStream[Present]] =
```

Integrating with ZIO Streams

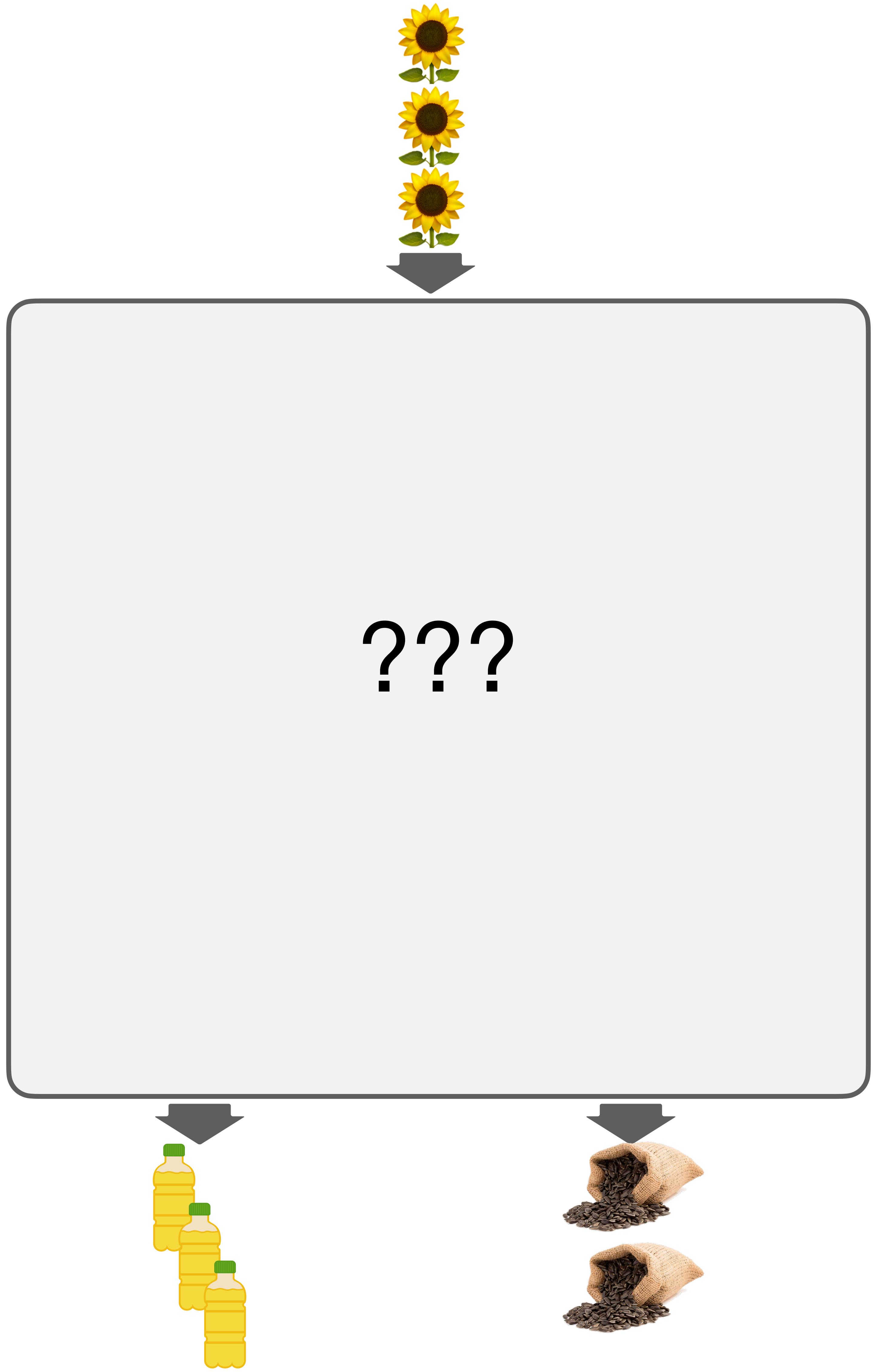
Libretto

```
def packagingLine: (Source[Toy] |*| Source[Bone] |*| Source[Biscuit]) -⚬ Source[Present]
```

ZIO

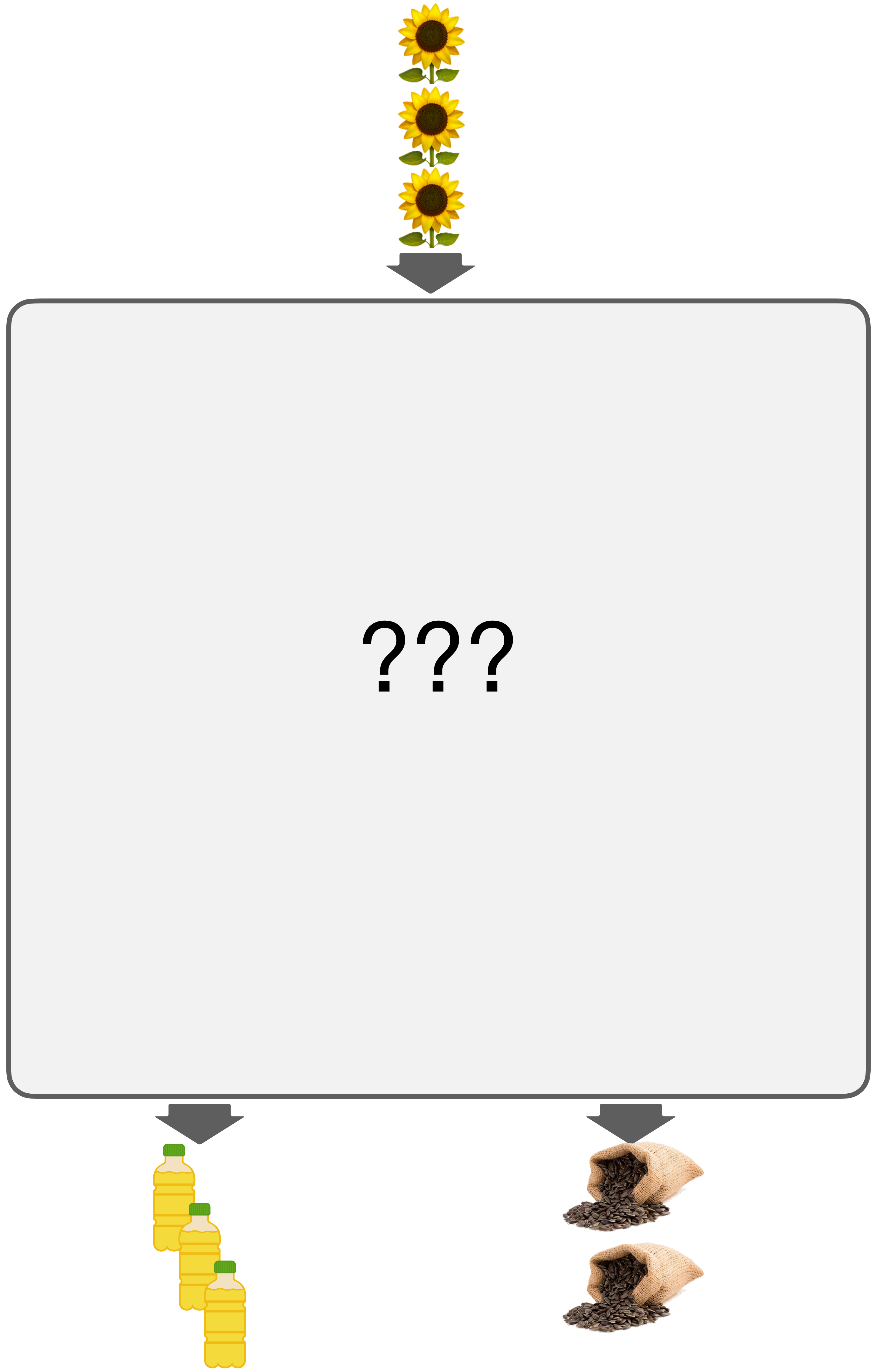
```
def go(
  toys: UStream[Toy],
  bones: UStream[Bone],
  biscuits: UStream[Biscuit],
): ZIO[Scope, Nothing, UStream[Present]] =
  (toys.asSource |*| bones.asSource |*| biscuits.asSource)
    .through_(packagingLine)
    .map(_.zstream)
```

Sunflower Processing Facility



- **In:** sunflowers
- **Out:** oil bottles 🧴, packs of seeds 🫘
- 5 🌻 for 🧴, 3 🌻 for 🫘
- Start on whichever item demanded first
- Halt when either:
  - both downstreams close
  - run out of sunflowers
- Waste at most 4 sunflowers

Sunflower Processing Facility



- **In:** sunflowers
- **Out:** oil bottles 🧴, packs of seeds 🌰
- 5 🌻 for 🧴, 3 🌻 for 🌰
- Start on whichever item demanded first
- Halt when either:
  - both downstreams close
  - run out of sunflowers
- Waste at most 4 sunflowers

Behavior depends on which downstream acts first (racing).

Sunflower Processing Facility: Idea

Sunflower Processing Facility: Idea

- feed the input source into a **queue**

Sunflower Processing Facility: Idea

- feed the input source into a **queue**
- start each consumer in a **fiber** and let them compete in pulling from queue

Sunflower Processing Facility: Idea

- feed the input source into a **queue**
- start each consumer in a **fiber** and let them compete in pulling from queue
- obtain a **lock** to pull the respective number of sunflowers (3 or 5)

Sunflower Processing Facility: Idea

- feed the input source into a **queue**
- start each consumer in a **fiber** and let them compete in pulling from queue
- obtain a **lock** to pull the respective number of sunflowers (3 or 5)
- notify the upstream when both consumer close using a **CountdownLatch**

Sunflower Processing Facility: Bad Idea

- feed the input source into e
- start each consumer in a **fiber** ompete in pulling from queue
- obtain a **lock** to pull the respec f sunflowers (3 or 5)
- notify the upstream when nsumer ing a **CountdownLatch**

Sunflower Processing Facility

```
def sunflowerProcessor: Source[Sunflower] -⚬ (Source[SeedsPack] |*| Source[OilBottle]) =
  rec { self =>
    λ { sunflowers =>
      producing { case seedsOut |*| oilOut => // give names to the outputs
        ???
      }
    }
  }
```

Sunflower Processing Facility

```
def sunflowerProcessor: Source[Sunflower] -o (Source[SeedsPack] |*| Source[OilBottle]) =
  rec { self =>
    λ { sunflowers =>
      producing { case seedsOut |*| oilOut => // give names to the outputs
        ???
      }
    }
  }
```

Sunflower Processing Facility

```
def sunflowerProcessor: Source[Sunflower] -⚬ (Source[SeedsPack] |*| Source[OilBottle]) =
 rec { self =>
  λ { sunflowers =>
   producing { case seedsOut |*| oilOut => // give names to the outputs
    ???
   }
  }
 }
```

Sunflower Processing Facility

```
def sunflowerProcessor: Source[Sunflower] -o (Source[SeedsPack] |*| Source[OilBottle]) =
  rec { self =>
    λ { sunflowers =>
      producing { case seedsOut |*| oilOut => // give names to the outputs
        // race the outputs by which one acts (i.e. pulls or closes) first
        (selectBy(notifyAction, notifyAction) >>: (seedsOut |*| oilOut)) switch {
          case Left(seedsOut |*| oilOut) => // seed output acted first
            ???
          case Right(seedsOut |*| oilOut) => // oil output acted first
            ???
        }
      }
    }
  }
```

Sunflower Processing Facility

```
case Left(seedsOut |*| oilOut) => // seed output acted first, still have sunflowers
  ???
```

Sunflower Processing Facility

```
case Left(seedsOut |*| oilOut) => // seed output acted first
  (fromChoice >>: seedsOut) switch {
    case Left( ✔ ) => // seed output closing, still have sunflowers, oilOut
      ???
    case Right(pullingSeeds) => // seed output pulling, still have sunflowers, oilOut
      ???
  }
```

Sunflower Processing Facility

```
case Left(seedsOut |*| oilOut) => // seed output acted first
  (fromChoice >>: seedsOut) switch {
    case Left( ✔ ) => // seed output closing, still have sunflowers, oilOut
      ???
    case Right(pullingSeeds) => // seed output pulling, still have sunflowers, oilOut

        pull3(sunflowers) switch {
          case Right(sunflower3 |*| sunflowers) =>


            ???


          case Left(   ✔  ) => // no more sunflowers
            ???
        }
  }
```

Sunflower Processing Facility

```
case Left(seedsOut |*| oilOut) => // seed output acted first
  (fromChoice >>: seedsOut) switch {
    case Left( ✔ ) => // seed output closing, still have sunflowers, oilOut
      ???
    case Right(pullingSeeds) => // seed output pulling, still have sunflowers, oilOut

        pull3(sunflowers) switch {
          case Right(sunflower3 |*| sunflowers) =>
            val seedsPack = roastSeedsAndPack(sunflower3)
            val seedsPacks |*| oilBottles = self(sunflowers)
            ???
          case Left(  ✔  ) => // no more sunflowers
            ???
        }
  }
```

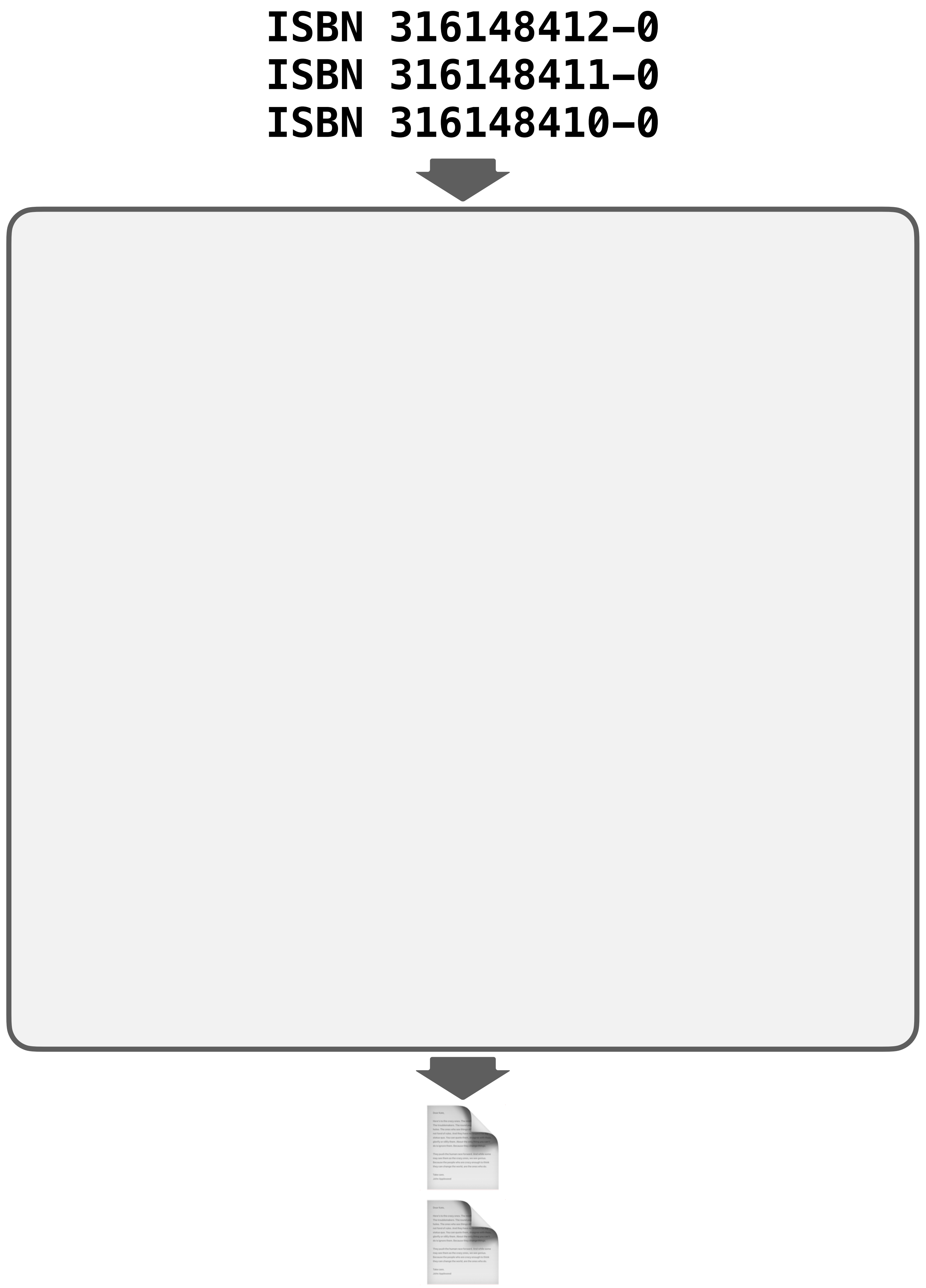
Sunflower Processing Facility

```
case Left(seedsOut |*| oilOut) => // seed output acted first
  (fromChoice >>: seedsOut) switch {
    case Left( ✔ ) => // seed output closing, still have sunflowers, oilOut
      ???
    case Right(pullingSeeds) => // seed output pulling, still have sunflowers, oilOut
      (pullingSeeds |*| oilOut) :=
        pull3(sunflowers) switch {
          case Right(sunflower3 |*| sunflowers) =>
            val seedsPack = roastSeedsAndPack(sunflower3)
            val seedsPacks |*| oilBottles = self(sunflowers)
            Polled.cons(seedsPack |*| seedsPacks) |*| oilBottles
          case Left(+( ✔ )) => // no more sunflowers
            Polled.empty( ✔ ) |*| Source.empty( ✔ )
        }
  }
```

Sunflower Processing Facility

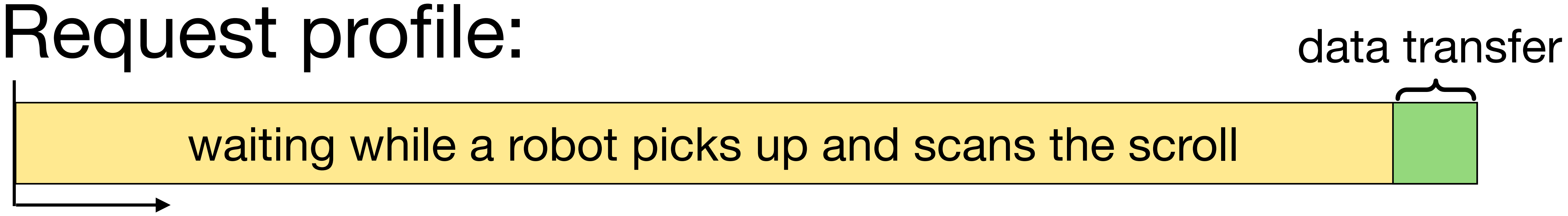
```
case Left(seedsOut |*| oilOut) => // seed output acted first
  (fromChoice >>: seedsOut) switch {
    case Left( ✔ ) => // seed output closing, still have sunflowers, oilOut
      ???
    case Right(pullingSeeds) => // seed output pulling, still have sunflowers, oilOut
      (pullingSeeds |*| oilOut) :=
        pull3(sunflowers) switch {
          case Right(sunflower3 |*| sunflowers) =>
            val seedsPack = roastSeedsAndPack(sunflower3)
            val seedsPacks |*| oilBottles = self(sunflowers)
            Polled.cons(seedsPack |*| seedsPacks) |*| oilBottles
          case Left(+( ✔ )) => // no more sunflowers
            Polled.empty( ✔ ) |*| Source.empty( ✔ )
        }
  }
```

Digital Library of Alexandria



```
ISBN 316148412-0
ISBN 316148411-0
ISBN 316148410-0
```

- **In:** scroll IDs (ISBNs)
- **Out:** pages of all given scrolls, in order
- Use provided API to request a scroll by its ID
  - returns a stream of scanned pages
- Fair use policy: max *k* concurrent connections
- Request profile:



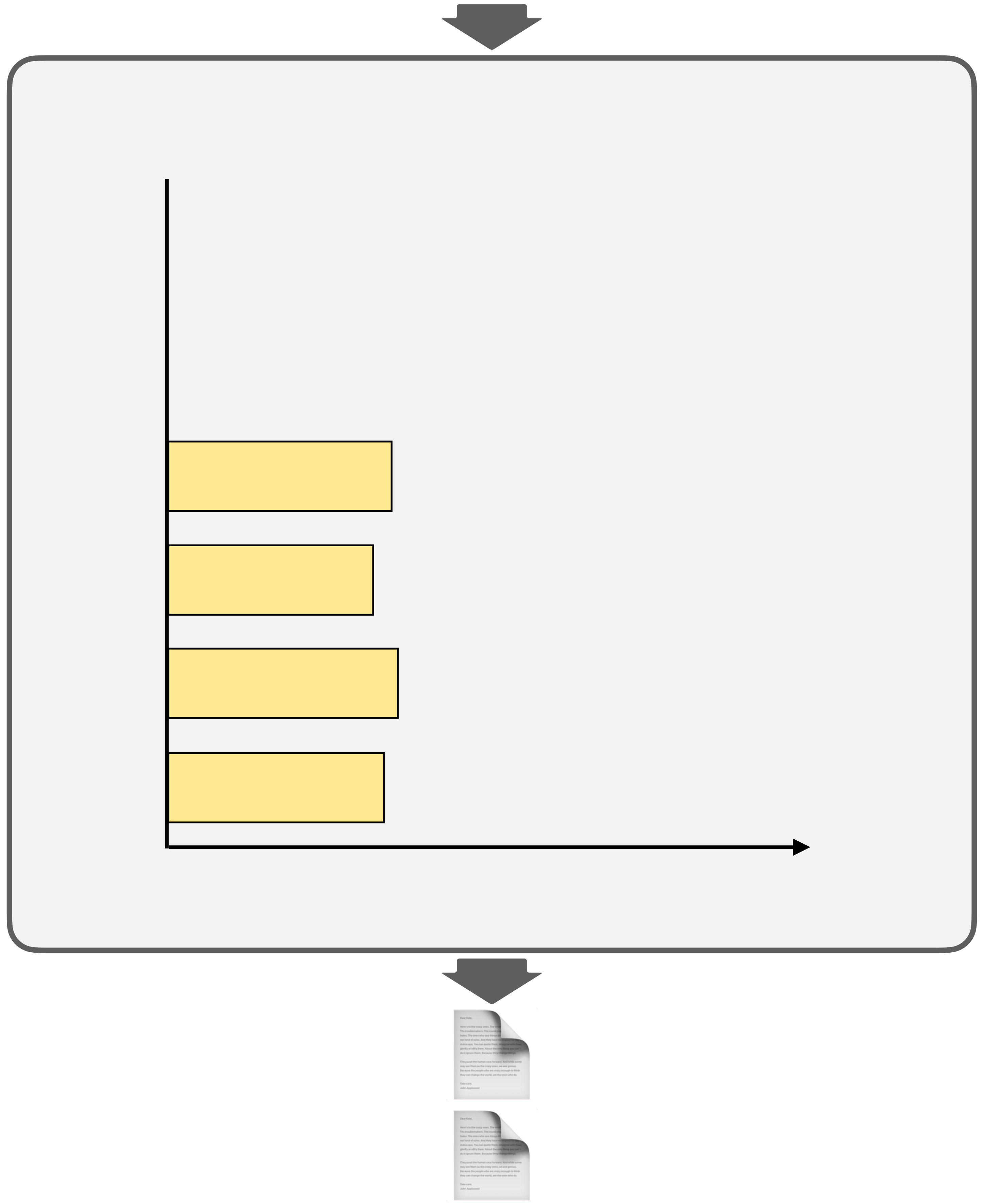
data transfer

waiting while a robot picks up and scans the scroll

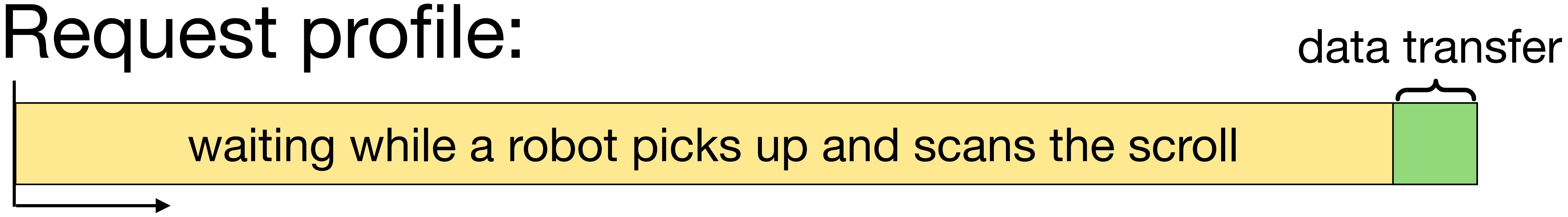
- Use all *k* connections to prepare documents, transfer data sequentially

Digital Library of Alexandria

```
ISBN 316148412-0
ISBN 316148411-0
ISBN 316148410-0
```



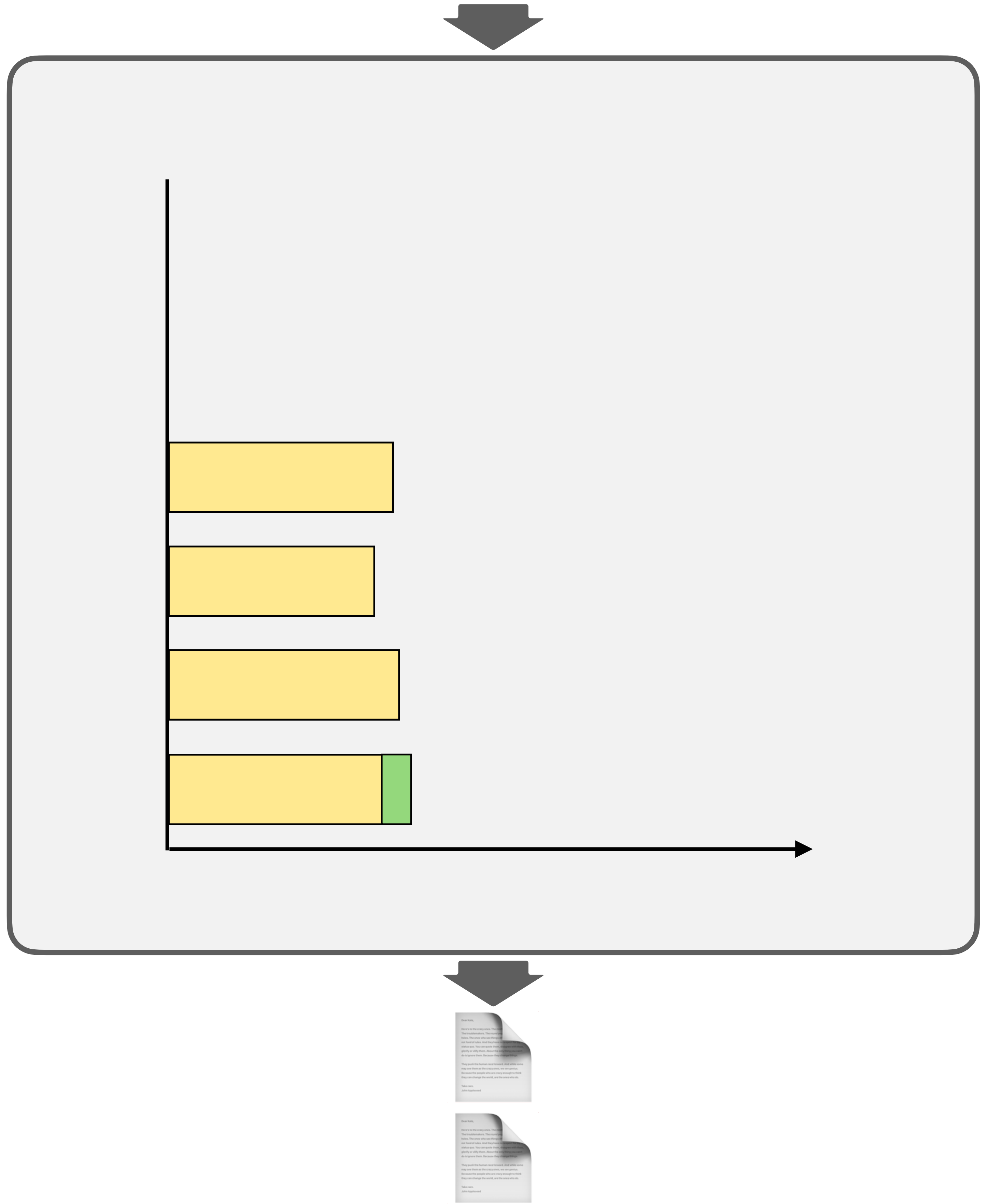
- **In:** scroll IDs (ISBNs)
- **Out:** pages of all given scrolls, in order
- Use provided API to request a scroll by its ID
    - returns a stream of scanned pages
- Fair use policy: max *k* concurrent connections
- Request profile:



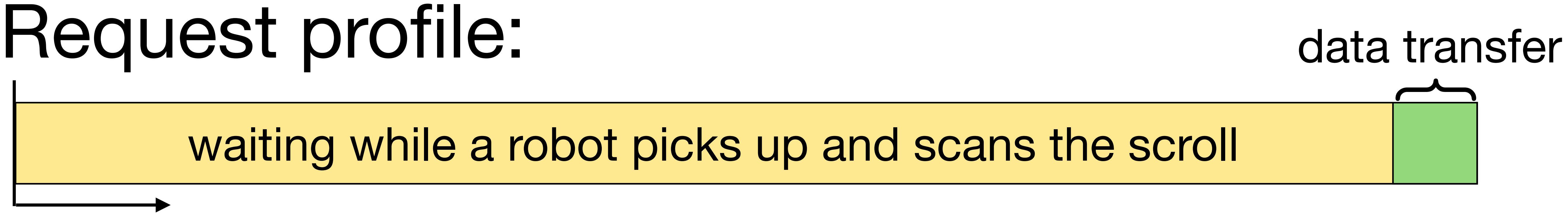
- Use all *k* connections to prepare documents, transfer data sequentially

Digital Library of Alexandria

ISBN 316148412-0
ISBN 316148411-0
ISBN 316148410-0



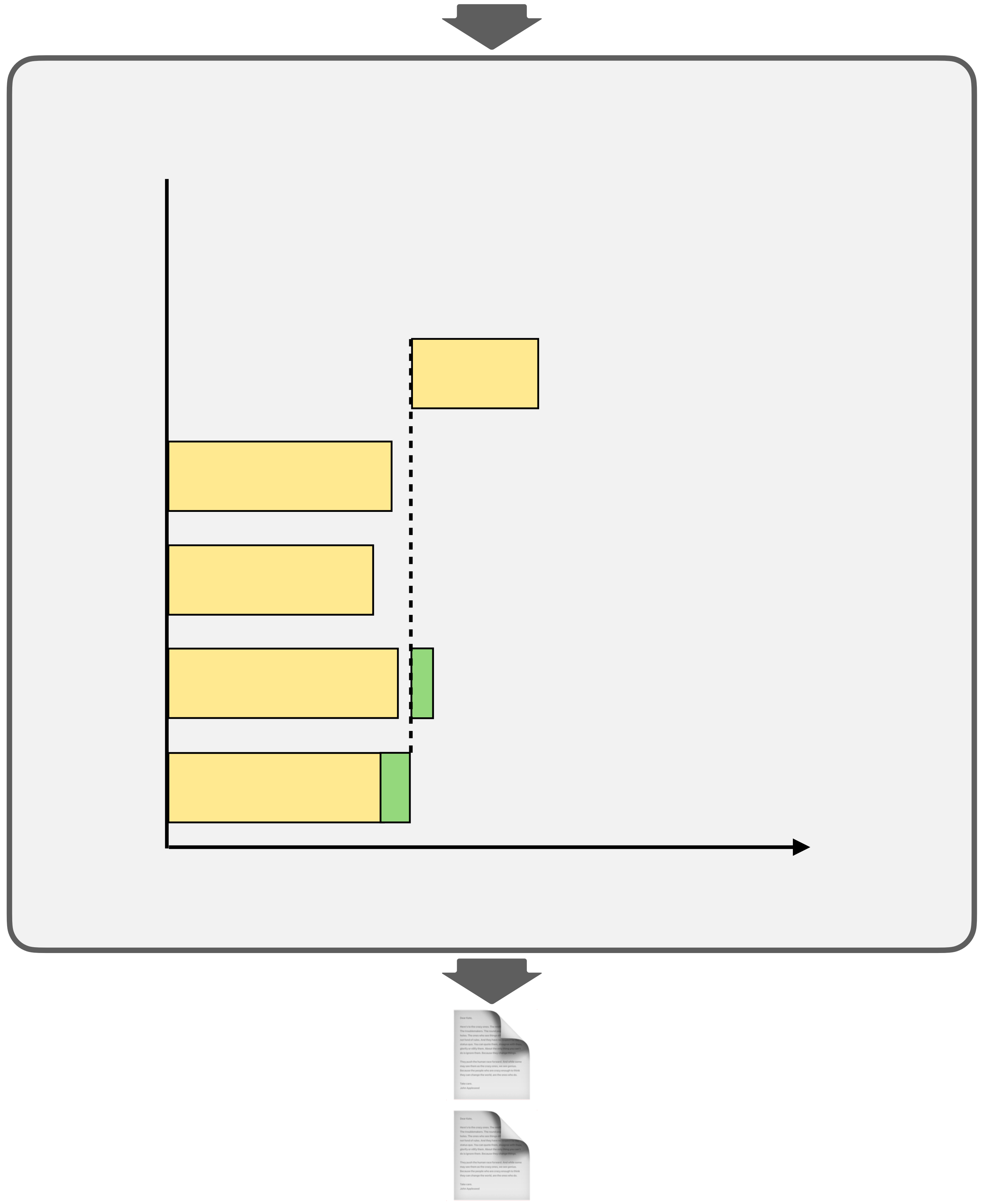
- **In:** scroll IDs (ISBNs)
- **Out:** pages of all given scrolls, in order
- Use provided API to request a scroll by its ID
  - returns a stream of scanned pages
- Fair use policy: max *k* concurrent connections
- Request profile:



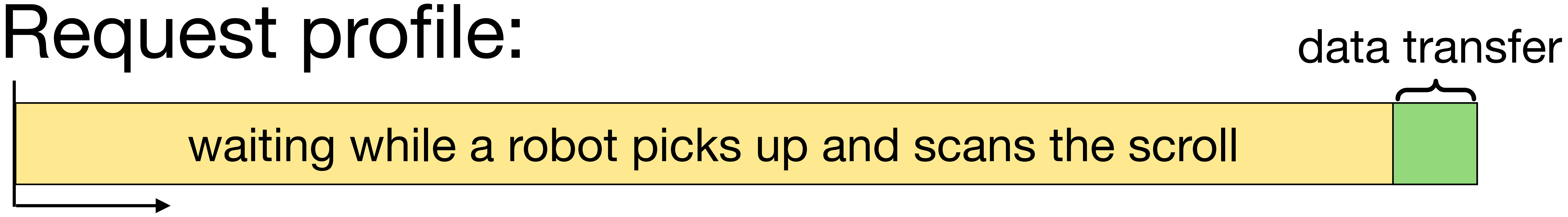
- Use all *k* connections to prepare documents, transfer data sequentially

Digital Library of Alexandria

```
ISBN 316148412-0
ISBN 316148411-0
ISBN 316148410-0
```

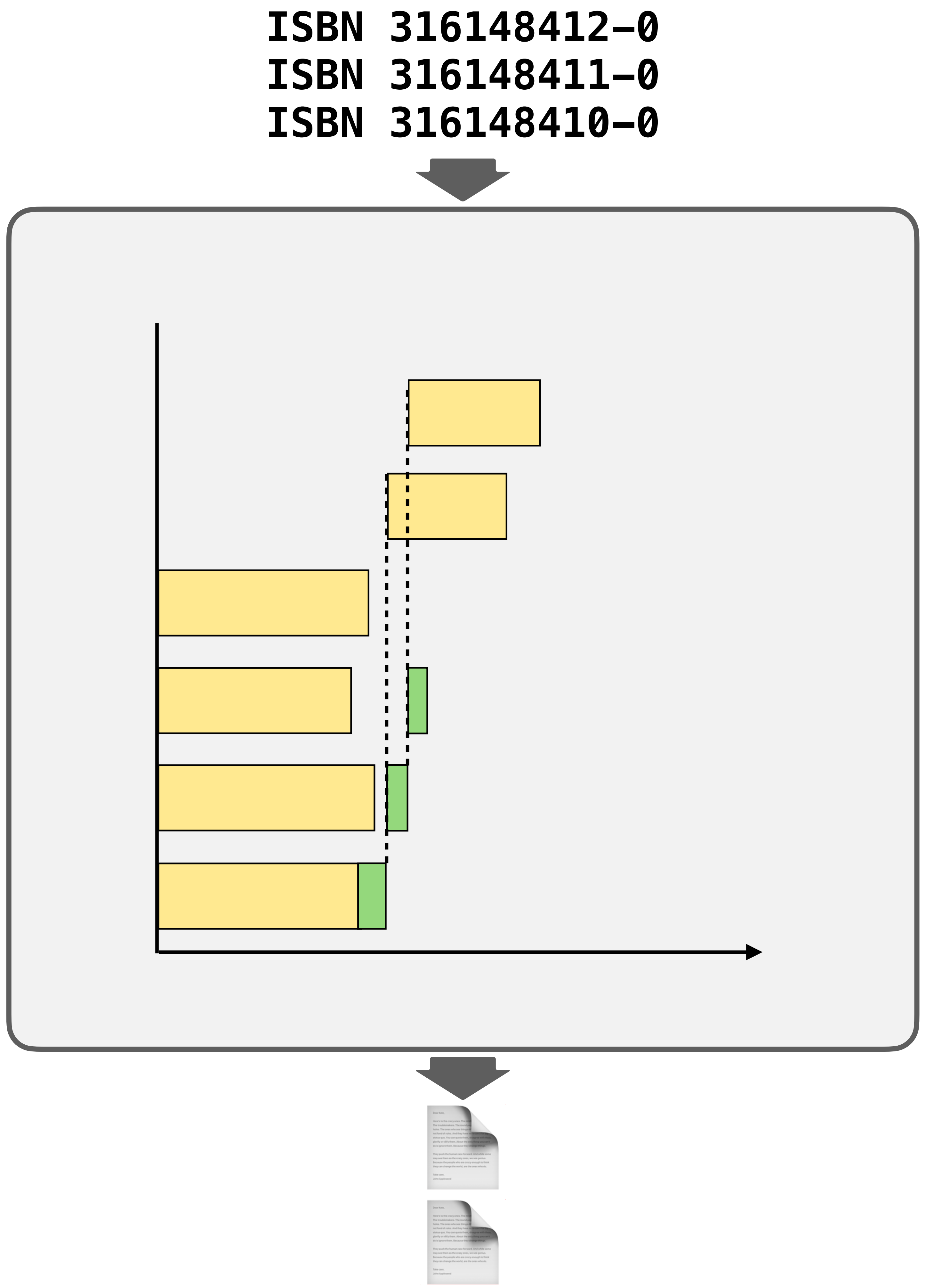


- **In:** scroll IDs (ISBNs)
- **Out:** pages of all given scrolls, in order
- Use provided API to request a scroll by its ID
  - returns a stream of scanned pages
- Fair use policy: max *k* concurrent connections
- Request profile:

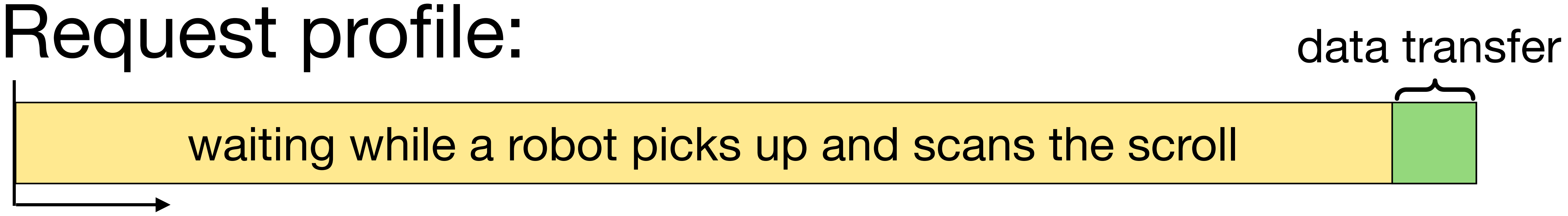


- Use all *k* connections to prepare documents, transfer data sequentially

Digital Library of Alexandria

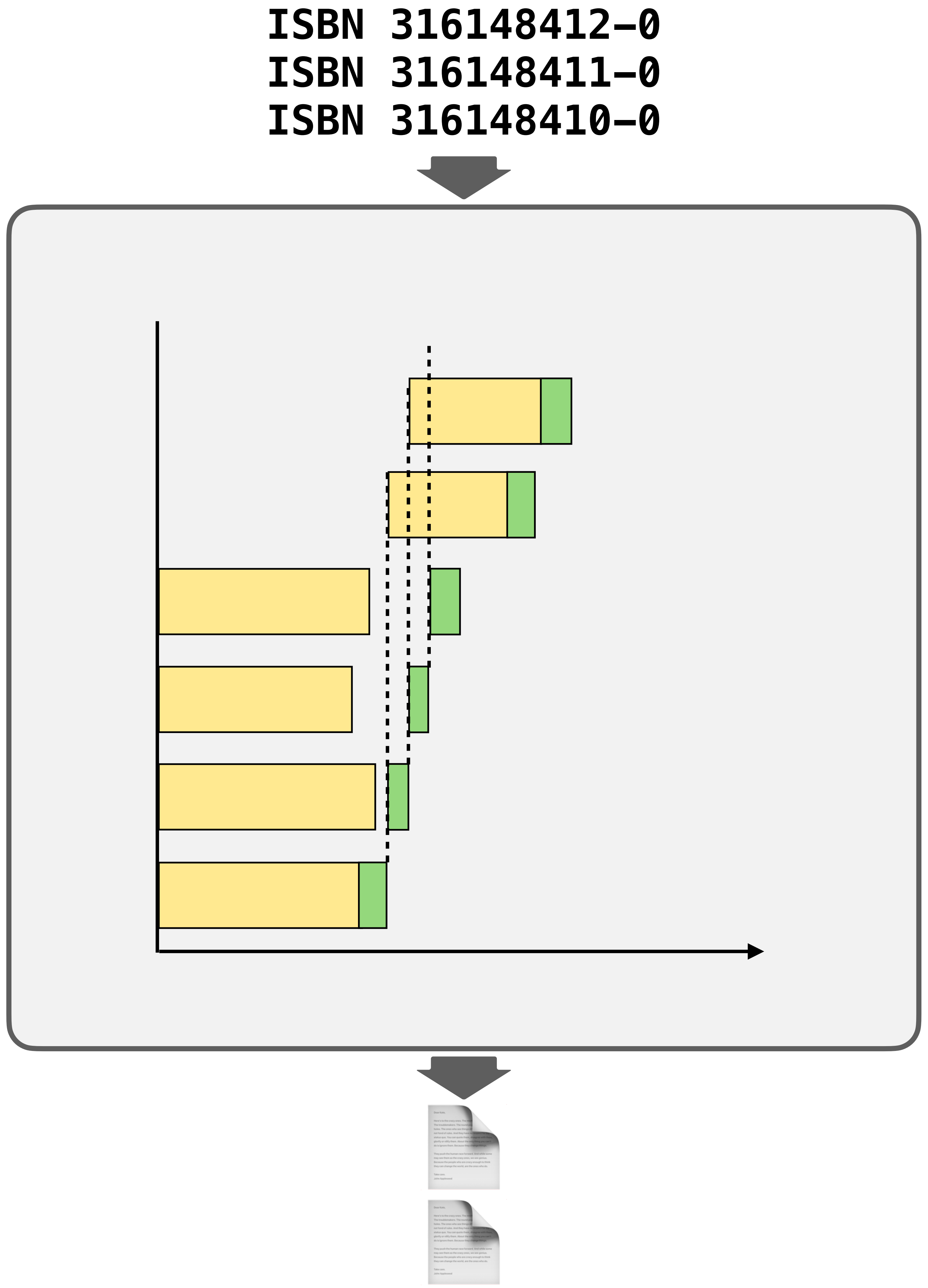


- **In:** scroll IDs (ISBNs)
- **Out:** pages of all given scrolls, in order
- Use provided API to request a scroll by its ID
  - returns a stream of scanned pages
- Fair use policy: max *k* concurrent connections
- Request profile:

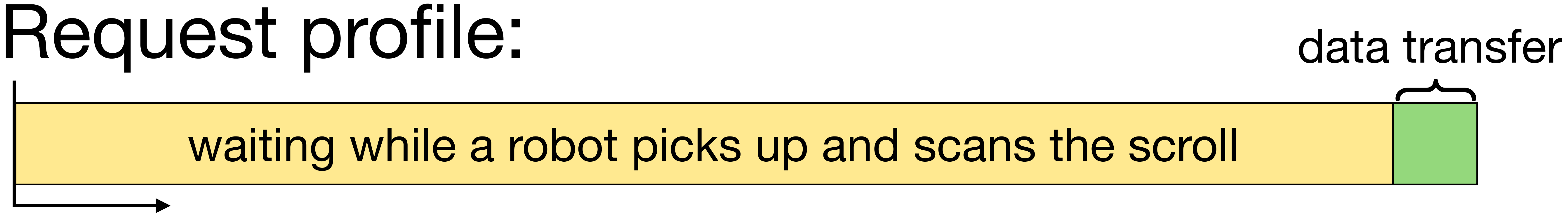


- Use all *k* connections to prepare documents, transfer data sequentially

Digital Library of Alexandria

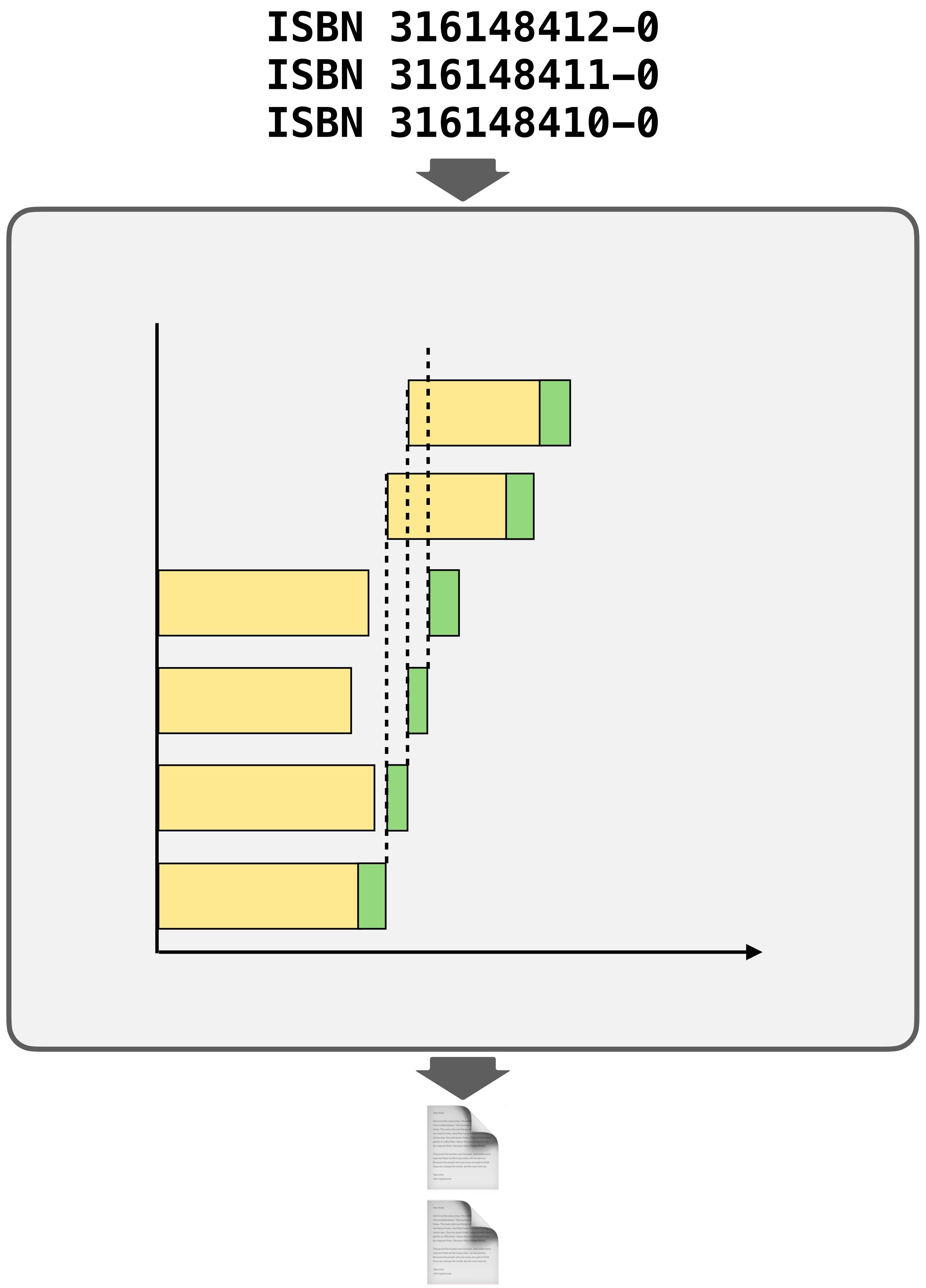


- **In:** scroll IDs (ISBNs)
- **Out:** pages of all given scrolls, in order
- Use provided API to request a scroll by its ID
  - returns a stream of scanned pages
- Fair use policy: max *k* concurrent connections
- Request profile:

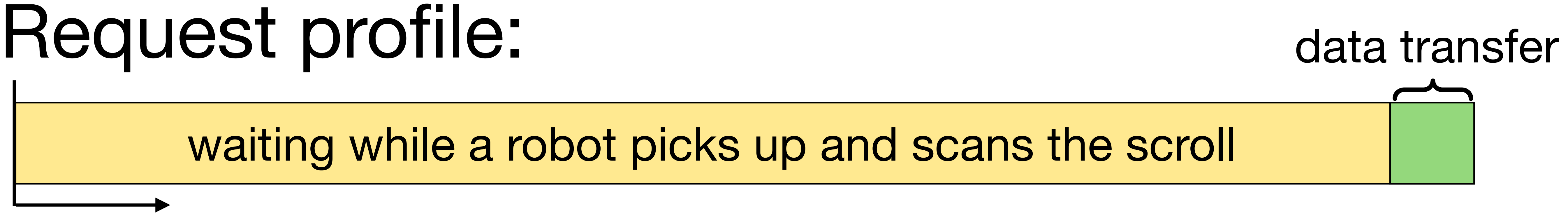


- Use all *k* connections to prepare documents, transfer data sequentially

Digital Library of Alexandria



- **In:** scroll IDs (ISBNs)
- **Out:** pages of all given scrolls, in order
- Use provided API to request a scroll by its ID
    - returns a stream of scanned pages
- Fair use policy: max *k* concurrent connections
- Request profile:



- Use all *k* connections to prepare documents, transfer data sequentially

Non-trivial resource lifetimes
(overlapping, but not nested)

Digital Library of Alexandria

Digital Library of Alexandria

```
// Provided.
// Opens a connection that is closed when the resulting Source is closed.
def fetchScroll: (Connector |*| ISBN) -o Source[📄]
```

Digital Library of Alexandria

```
// Provided.
// Opens a connection that is closed when the resulting Source is closed.
def fetchScroll: (Connector |*| ISBN) -o Source[📄]

def downloadAll(k: Int): (Connector |*| Source[ISBN]) -o Source[📄] =
```

Digital Library of Alexandria

```
// Provided.
// Opens a connection that is closed when the resulting Source is closed.
def fetchScroll: (Connector |*| ISBN) -o Source[📄]

def downloadAll(k: Int): (Connector |*| Source[ISBN]) -o Source[📄] =
  mapWith(fetchScroll) // Source[Source[📄]]
```

Digital Library of Alexandria

```
// Provided.
// Opens a connection that is closed when the resulting Source is closed.
def fetchScroll: (Connector |*| ISBN) -o Source[📄]

def downloadAll(k: Int): (Connector |*| Source[ISBN]) -o Source[📄] =
  mapWith(fetchScroll) // Source[Source[📄]]
    > prefetch(k - 1)(discardPrefetched = Source.close)
```

Digital Library of Alexandria

```
// Provided.
// Opens a connection that is closed when the resulting Source is closed.
def fetchScroll: (Connector |*| ISBN) -o Source[📄]

def downloadAll(k: Int): (Connector |*| Source[ISBN]) -o Source[📄] =
  mapWith(fetchScroll) // Source[Source[📄]]
    > prefetch(k - 1)(discardPrefetched = Source.close)
    > flatten
```

Digital Library of Alexandria

```
// Provided.
// Opens a connection that is closed when the resulting Source is closed.
def fetchScroll: (Connector |*| ISBN) -o Source[📄]

def downloadAll(k: Int): (Connector |*| Source[ISBN]) -o Source[📄] =
  mapWith(fetchScroll) // Source[Source[📄]]
    > prefetch(k - 1)(discardPrefetched = Source.close)
    > flatten
```

Correct
Resource Safe

Digital Library of Alexandria

```
def fetchScroll: (Connector |*| ISBN) -o Source[📄]

def downloadAll(k: Int): (Connector |*| Source[ISBN]) -o Source[📄] =
  mapWith(fetchScroll) // Source[Source[📄]]
    > prefetch(k - 1)(discardPrefetched = Source.close)
    > flatten
```

Correct
Resource Safe

Does not work in libs where `Source` / `Stream` is a “blueprint”

Digital Library of Alexandria

```
def fetchScroll: (Connector |*| ISBN) -o Source[📄]

def downloadAll(k: Int): (Connector |*| Source[ISBN]) -o Source[📄] =

  mapWith(fetchScroll) // Source[Source[📄]]

    > prefetch(k - 1)(discardPrefetched = Source.close)

    > flatten
```

Correct
Resource Safe

Does not work in libs where `Source` / `Stream` is a “blueprint”

```
Stream[Stream[📄]] .prefetch(n) .flatten
```

- prefetches blueprints, does not start doc preparation

Digital Library of Alexandria

```
def fetchScroll: (Connector |*| ISBN) -o Source[📄]

def downloadAll(k: Int): (Connector |*| Source[ISBN]) -o Source[📄] =

  mapWith(fetchScroll) // Source[Source[📄]]
    > prefetch(k - 1)(discardPrefetched = Source.close)
    > flatten
```

Correct
Resource Safe

Does not work in libs where `Source` / `Stream` is a “blueprint”

```
Stream[Stream[📄]] .prefetch(n) .flatten
```

- prefetches blueprints, does not start doc preparation

```
Stream[Stream[📄]] .flatten .prefetch(n)
```

- prefetches n pages of concatenation, instead of preparing n documents

Digital Library of Alexandria

```
def fetchScroll: (Connector |*| ISBN) -o Source[📄]

def downloadAll(k: Int): (Connector |*| Source[ISBN]) -o Source[📄] =

  mapWith(fetchScroll) // Source[Source[📄]]
    > prefetch(k - 1)(discardPrefetched = Source.close)
    > flatten
```

Correct
Resource Safe

Does not work in libs where `Source` / `Stream` is a “blueprint”

```
Stream[Stream[📄]] .prefetch(n) .flatten
```

- prefetches blueprints, does not start doc preparation

```
Stream[Stream[📄]] .flatten .prefetch(n)
```

- prefetches n pages of concatenation, instead of preparing n documents

```
Stream[ISBN] .mapAsync(n)(ISBN => IO[Stream[📄]])
```

- if `IO` action starts doc prep in background, who closes connection if `Stream` never consumed?

Digital Library of Alexandria

```
def fetchScroll: (Connector |*| ISBN) -⚬ Source[📄]

def downloadAll(k: Int): (Connector |*| Source[ISBN]) -⚬ Source[📄] =
  mapWith(fetchScroll) > prefetch(k - 1)(discardPrefetched = Source.close) > flatten
```

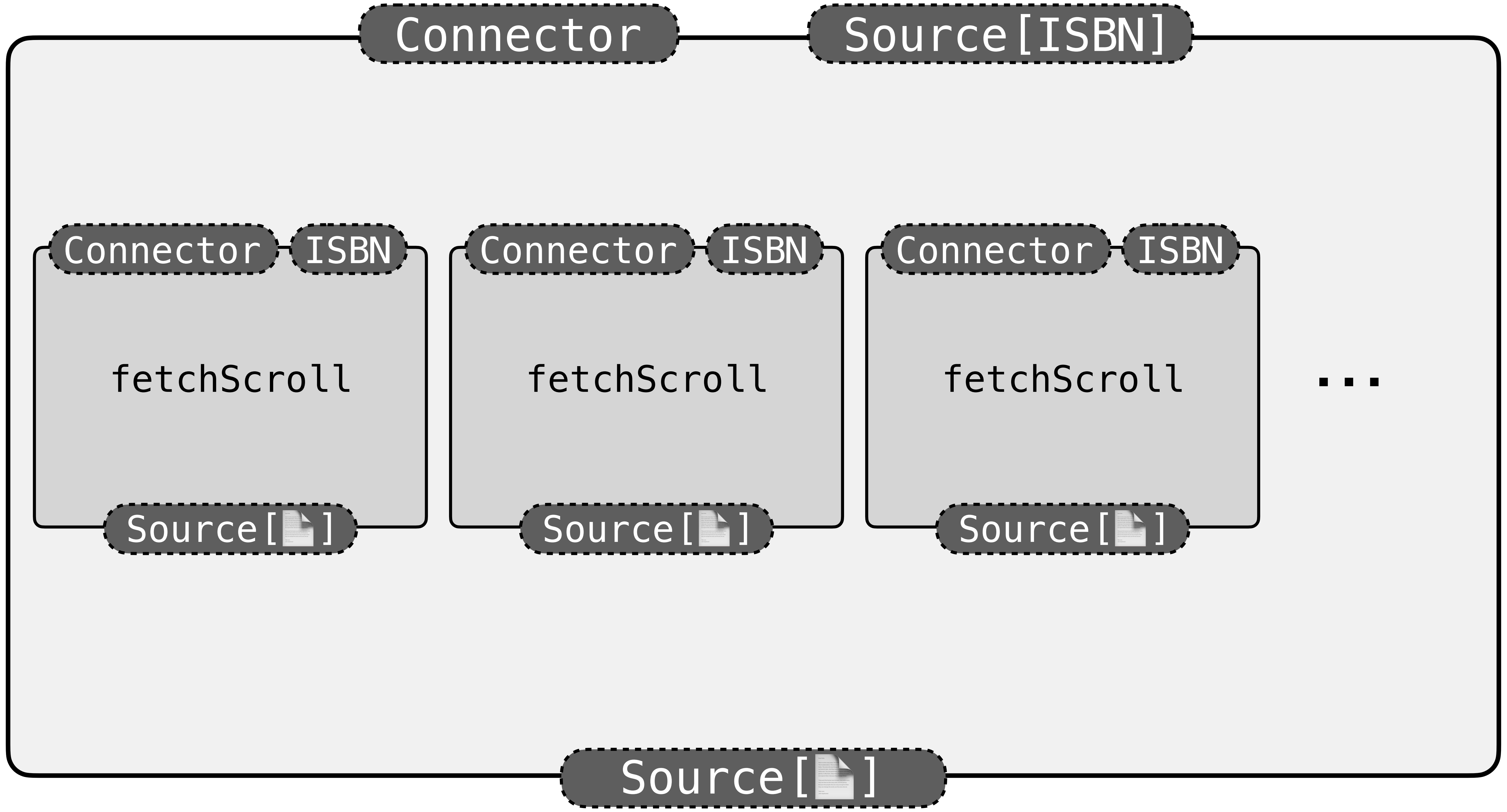
Why does it work in Libretto?

`Source[📄]`

- not a blueprint
- phantom type
- *interface* of interaction (`poll`, `close`)
- *running* process on each sides
- `A -⚬ Source[📄]` is the blueprint

Resources

- not tied to inflexible (nested) scopes
- release guaranteed by **linearity**



Summary

Summary

Declarative or **Expressive**? Pick two!

Summary

Declarative or **Expressive**? Pick two!

Stream operators in Libretto are

safer and **simpler** than the alternatives.

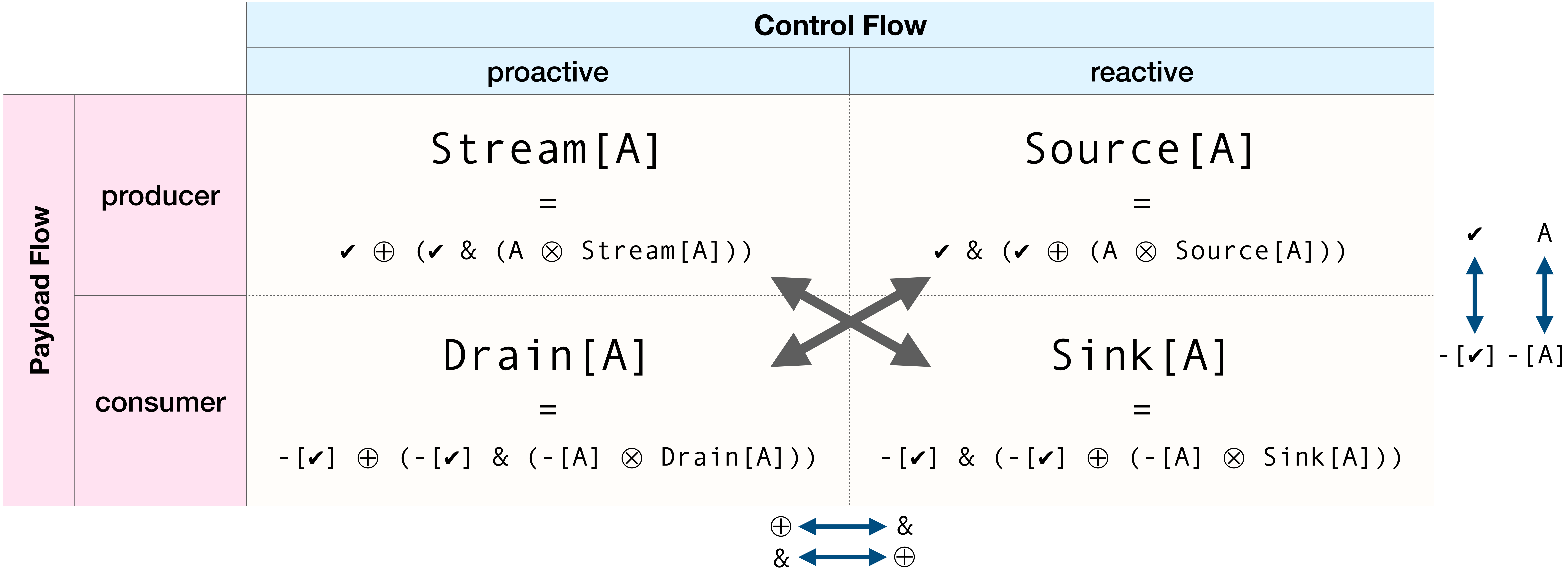
Summary

Declarative or **Expressive**? Pick two!

Stream operators in Libretto are
safer and **simpler** than the alternatives.

(I might be biased, feel free to challenge.)

Streams in Libretto



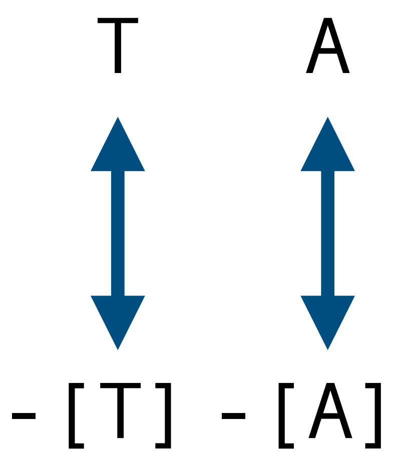
		Control Flow	
		proactive	reactive
Payload Flow	producer	Stream[A] = ✔ ⊕ (✔ & (A ⊗ Stream[A]))	Source[A] = ✔ & (✔ ⊕ (A ⊗ Source[A]))
	consumer	Drain[A] = -[✔] ⊕ (-[✔] & (-[A] ⊗ Drain[A]))	Sink[A] = -[✔] & (-[✔] ⊕ (-[A] ⊗ Sink[A]))

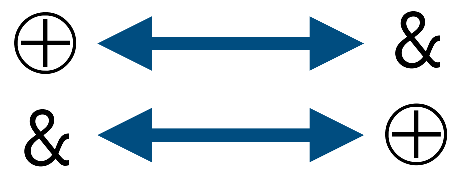
-[Stream[A]] ~ Sink[A]

-[Source[A]] ~ Drain[A]

Bonus: Streams with Custom Terminator

		Control Flow	
		proactive	**reactive**
Payload Flow	**producer**	StreamT[T,A] = T ⊕ (T & (A ⊗ StreamT[T,A]))	SourceT[T,A] = T & (T ⊕ (A ⊗ SourceT[T,A]))
	consumer	DrainT[T,A] = -[T] ⊕ (-[T] & (-[A] ⊗ DrainT[T,A]))	SinkT[A] = -[T] & (-[T] ⊕ (-[A] ⊗ SinkT[T,A]))





Example: API of a TV streaming service

```
Tv = ✔ & (ChannelName =o SourceT[Tv,VideoFrame])
```

- ensures consuming at most 1 channel at a time

Gateway drug to session types

Thank you!

github.com/TomasMikula/libretto/

(includes runnable version of each shown example)