ECOOP 2014

Rely-Guarantee Protocols

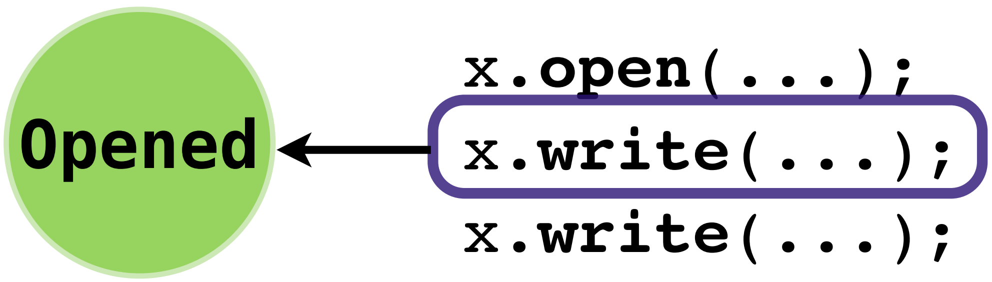
Filipe Militão[1,2] Jonathan Aldrich[1] Luís Caires[2]

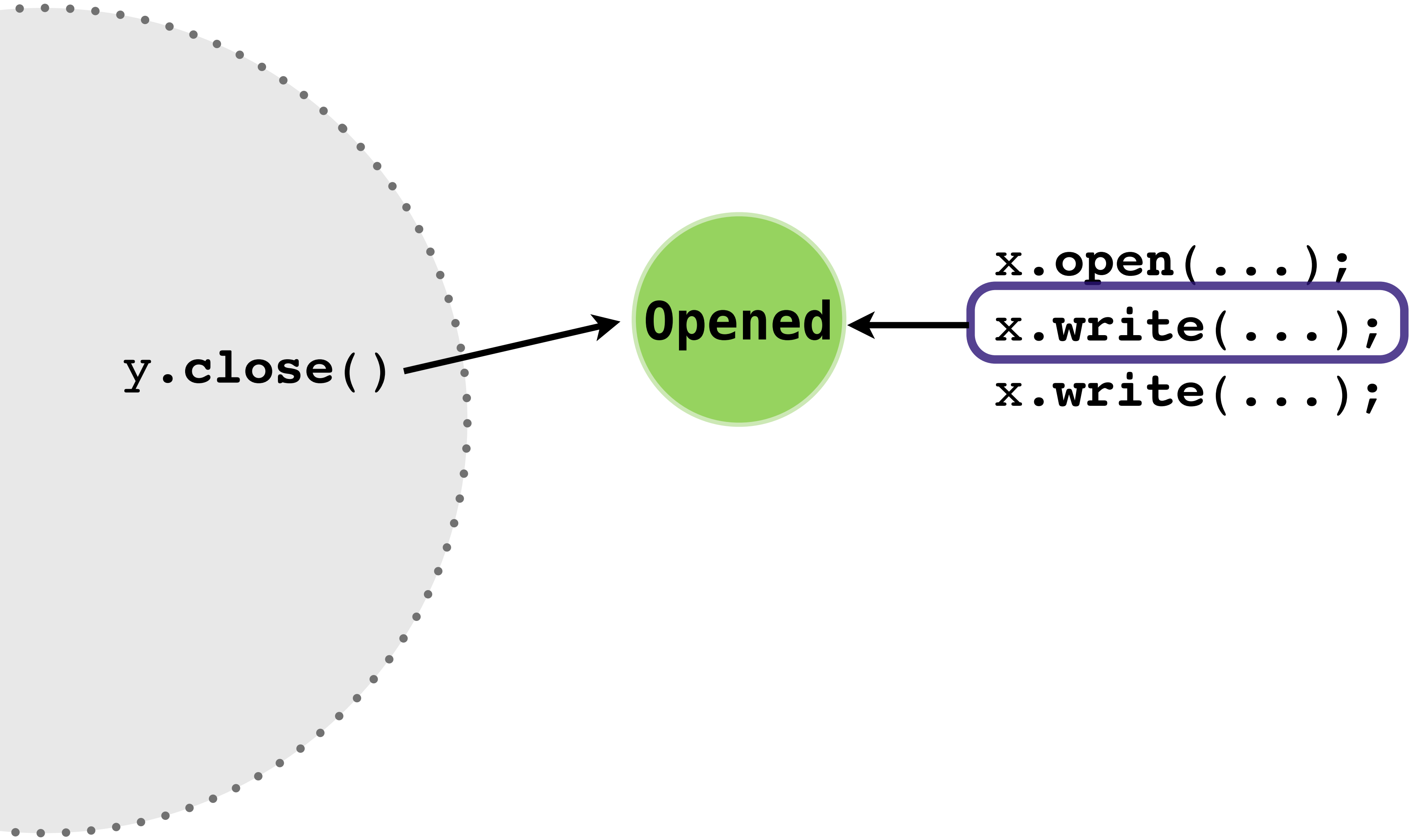
[1] Carnegie Mellon University, Pittsburgh, USA
[2] Universidade Nova de Lisboa, Lisboa, Portugal

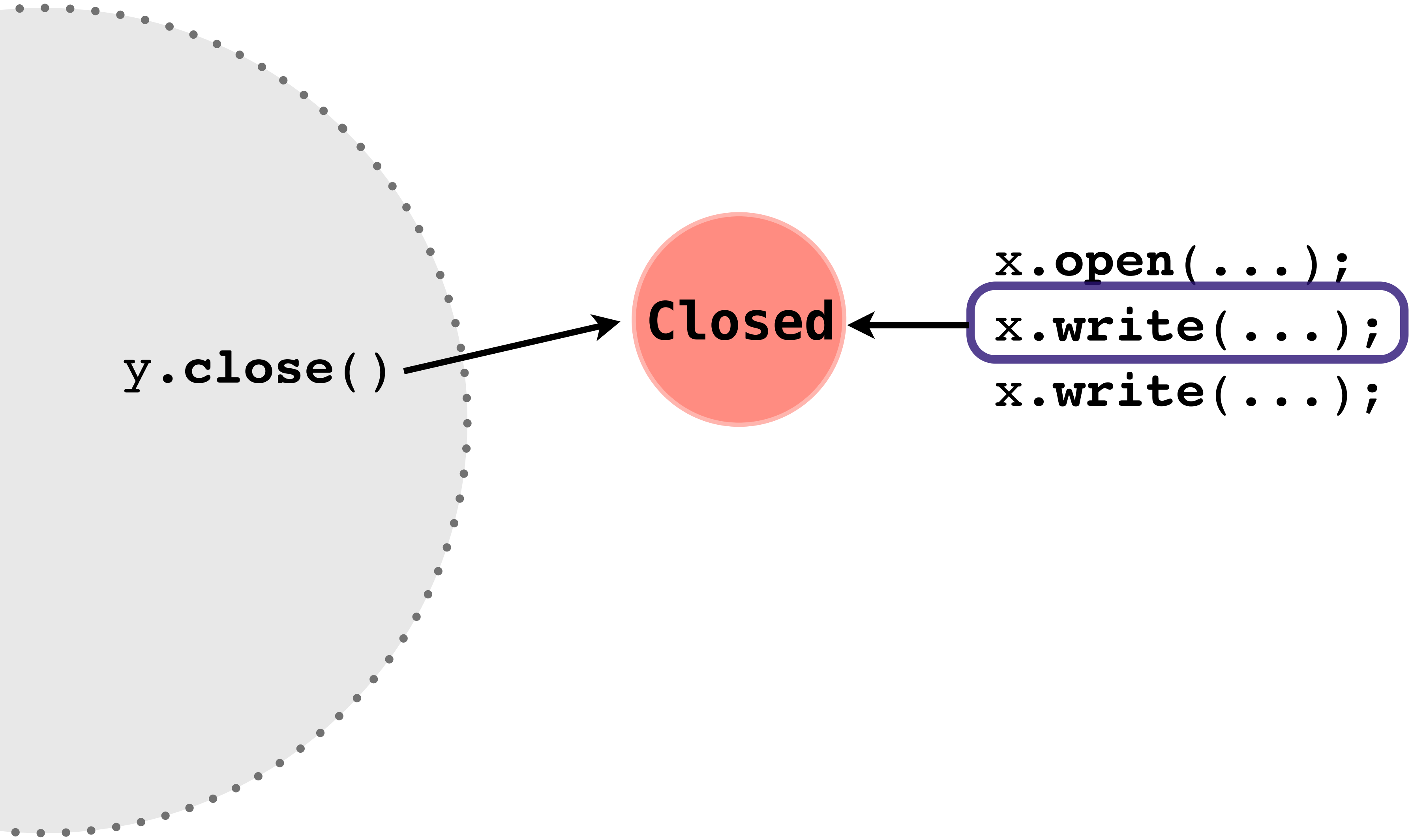
Motivation

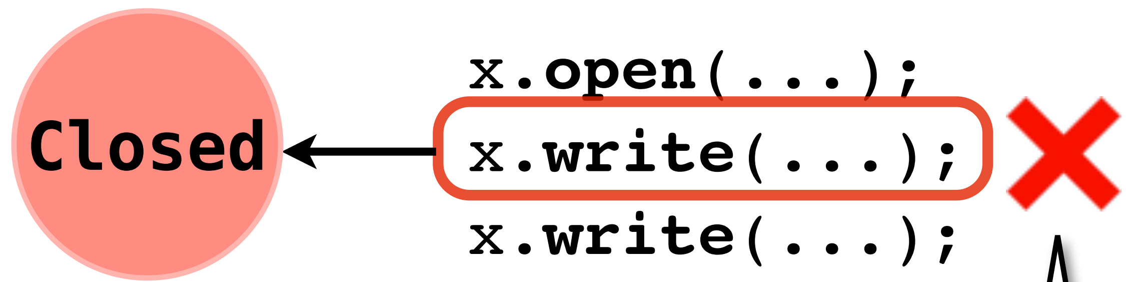
- Mutable state can be useful in certain cases.
- Precisely tracking the properties of mutable state avoids a class of state-related errors.
- However, aliasing makes tracking such properties challenging.

```
x.open(...);
x.write(...);
x.write(...);
```


Opened
x.open(...);
x.write(...);
x.write(...);


y.close()
Opened
x.open(...);
x.write(...);
x.write(...);


y.close()
Closed
x.open(...);
x.write(...);
x.write(...);



The assumption that `x` was pointing to an `Opened` file can be invalidated due to the **interference** caused by `y`.

Contribution

A novel *interference*-control mechanism, **Rely-Guarantee Protocols**, to statically handle interference in the use of mutable state that is shared by aliases through statically disconnected variables.

Language

- Polymorphic λ-calculus with mutable references (and immutable records, tagged sums, ...).
- Technically, we use a variant of **L³** adapted for usability and extended with new constructs, and our sharing mechanism.

[*Ahmed, Fluet, and Morrisett.* **L³: A linear language with locations**. Fundam. Inform. 2007.]

State as a Linear Resource

ref A

State as a Linear Resource

ref A

reference with contents of type A

State as a Linear Resource

ref A

reference with contents of type A

$$\underbrace{\qquad\qquad\qquad}_{\textbf{ref}\ p \qquad \textbf{rw}\ p\ A}$$

ref p **rw** p A

State as a Linear Resource

ref A

reference with contents of type A

ref p **rw** p A

duplicable **ref**erence to location p

State as a Linear Resource

ref A

reference with contents of type A

ref p
ref p
ref p
ref p
ref p
ref p
ref p

rw p A

duplicable **ref**erence to location p

State as a Linear Resource

ref A

reference with contents of type A

ref p
ref p
ref p
ref p
ref p
ref p
ref p

duplicable **ref**erence to location p

rw p A

linear ("unique") **r**ead+**w**rite **capability** of location p with contents of type A

State as a Linear Resource

ref A

reference with contents of type A

duplicable **ref**erence to location *p*

ref *p*
ref *p*
ref *p*
ref *p*
ref *p*
ref *p*
ref *p*

rw *p* A

linear ("unique") **r**ead+**w**rite **capability** of location *p* with contents of type A

p links ref to capability

$\mathtt{x} : \mathbf{ref}\ p$ $\qquad$ $\mathbf{rw}\ p\ \mathbf{string}$

```
let y = x in
  x := 1;
  delete y;
  x := false
end
```

```
let y = x in
```

y : **ref** *p*   x : **ref** *p*   **rw** *p* **string**

```
  x := 1;
  delete y;
  x := false
end
```

```
let y = x in
  x := 1;
```

y : **ref** *p*   x : **ref** *p*   **rw** *p* **int**

```
  delete y;
  x := false
end
```

```
let y = x in
   x := 1;
   delete y;
```

$\boxed{y : \textbf{ref}\ p \quad x : \textbf{ref}\ p}$

```
   x := false
end
```

```
let y = x in
  x := 1;
  delete y;
```

y : **ref** p x : **ref** p

```
  x := false
end
```

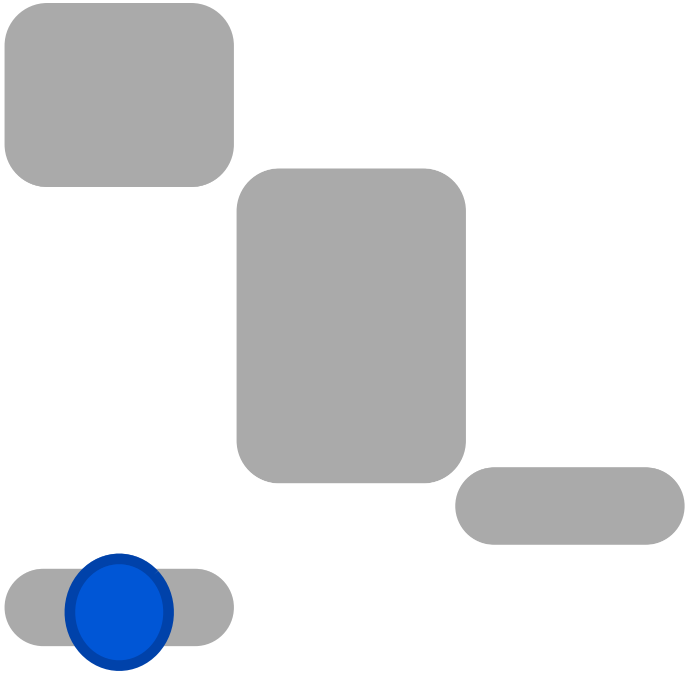
Type Error: Missing capability to location p.

Why sharing?

Capabilities are *linear* (a.k.a. “unique”)!

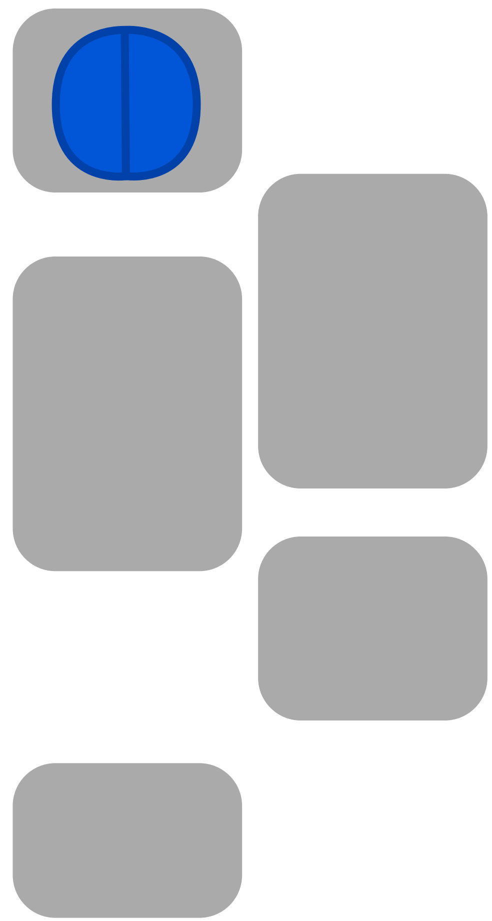
Why sharing?

Capabilities are *linear* (a.k.a. "unique")!



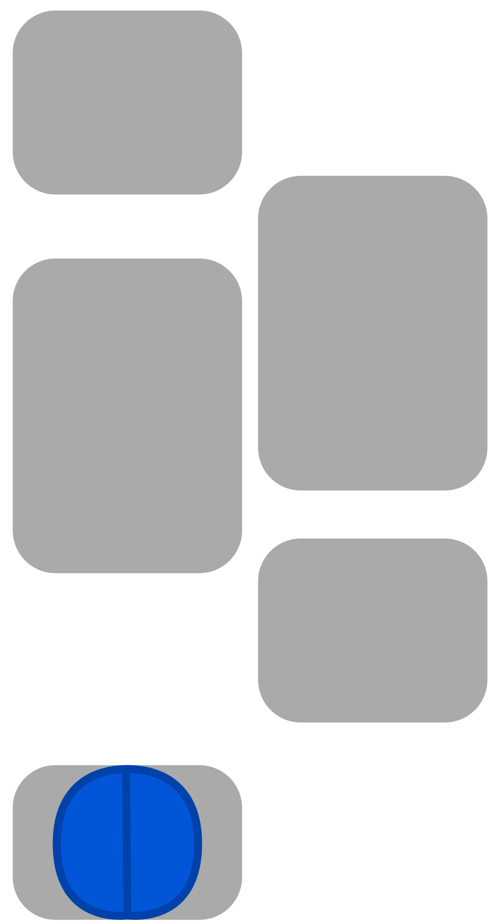
Sharing

A capability is split into **rely-guarantee protocols** to safely coordinate access to the shared state.



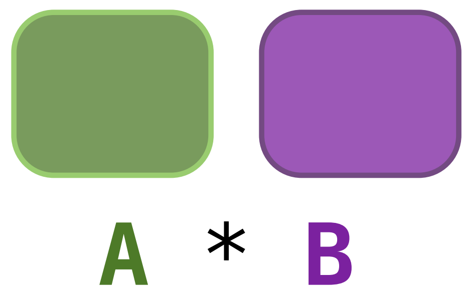
Sharing

A capability is split into **rely-guarantee protocols** to safely coordinate access to the shared state.



Disjoint

- Linearity ensured disjointness.

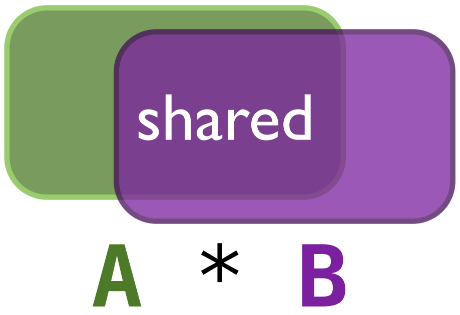


- Sharing causes *fictional* disjointness.

[Dinsdale-Young, *et al.* **Concurrent Abstract Predicates.** (ECOOP'10), and other works].

Fictionally **Disjoint**

- Linearity ensured disjointness.



- Sharing causes *fictional* disjointness.

[Dinsdale-Young, *et al.* **Concurrent Abstract Predicates.** (ECOOP'10), and other works].

Problems of Sharing

1. Account for **interference** (*public* changes).

   Consider all possible interleaved uses of aliases and how they may change the shared state.

2. Handle **private** changes.

   Making sure other aliases do not see any intermediate or inconsistent states of the shared state (which may appear due to type changing assignments like **`int`** to **`string`**, etc.).

Alias Interleaving

```
x := 1;
doSomething();
!x // what do we get?
```

Alias Interleaving

```
fun().1

fun().x := false

fun().delete x
```

```
x := 1;
doSomething();
!x // what do we get?
```

`doSomething` interleave *zero or more* aliases to the same state as referenced by **`x`**.

Alias Interleaving

```
x := 1;
doSomething();
!x // what do we get?
```

If **`doSomething`** did change the same state as aliased by **`x`** (i.e. interfered), what change occurred?

Handling Interference

- One solution is to ensure that each alias obeys an initially held invariant, *invariant-based sharing*.

$$\boxed{\texttt{I}} \Rightarrow \boxed{\texttt{I}}$$

- Instead, we adapt the *spirit* of rely-guarantee reasoning to a state-centric model by generalizing the specification of shared state interactions.

$$\boxed{\texttt{R}} \Rightarrow \boxed{\texttt{G}}$$

By individually constraining the actions of each alias, we can make stronger (as in more precise) assumptions how interference may change the shared state.

Handling Interference

- One solution is to ensure that each alias obeys an initially held invariant, *invariant-based sharing*.

$$\mathtt{I} \Rightarrow \mathtt{I}$$

- Instead, we adapt the *spirit* of rely-guarantee reasoning to a state-centric model by generalizing the specification of shared state interactions.

Relied state

$$\mathtt{R} \Rightarrow \mathtt{G}$$

By individually constraining the actions of each alias, we can make stronger (as in more precise) assumptions how interference may change the shared state.

Handling Interference

- One solution is to ensure that each alias obeys an initially held invariant, *invariant-based sharing*.

$$\mathtt{I} \Rightarrow \mathtt{I}$$

- Instead, we adapt the *spirit* of rely-guarantee reasoning to a state-centric model by generalizing the specification of shared state interactions.

Relied state $\mathtt{R} \Rightarrow \mathtt{G}$ **G**uaranteed state

By individually constraining the actions of each alias, we can make stronger (as in more precise) assumptions how interference may change the shared state.

Modeling Interference

A *Rely-Guarantee Protocol* models the shared state interaction from the alias' own view/perspective:

- The alias' actions are constrained to fit within what the protocol specifies/allows.
- Interference is observed through new state(s) that *may* appear when inspecting the shared state. Thus, the protocol may specify actions over states that can only be produced by other aliases.

Rely-Guarantee Protocols

- An *interference*-control mechanism.
- I will focus on presenting the following:
  1. Protocol Specification ("public changes")
  2. Protocol Use ("private changes")
  3. Protocol Conformance ("alias interleaving")

(see the paper for more technical details)

Types

$$
\begin{array}{lll}
A ::= & !A & \text{(pure/persistent)} \\
\mid & A \multimap A & \text{(linear function)} \\
\mid & A :: A & \text{(stacking)} \\
\mid & A * A & \text{(separation)} \\
\mid & [\overline{\mathtt{f} : A}] & \text{(record)} \\
\mid & X & \text{(type variable)} \\
\mid & \forall X.A & \text{(universal type quantification)} \\
\mid & \exists X.A & \text{(existential type quantification)} \\
\mid & \forall t.A & \text{(universal location quantification)} \\
\mid & \exists t.A & \text{(existential location quantification)} \\
\mid & \mathbf{ref}\ p & \text{(reference type)} \\
\mid & \mathbf{rec}\ X.A & \text{(recursive type)} \\
\mid & \sum_i \mathtt{l}_i \# A_i & \text{(tagged sum)} \\
\mid & A \oplus A & \text{(alternative)} \\
\mid & A \,\&\, A & \text{(intersection)} \\
\mid & \mathbf{rw}\ p\ A & \text{(read-write capability to } p\text{)} \\
\mid & \mathsf{none} & \text{(empty capability)} \\
\mid & A \Rightarrow A & \text{(rely)} \\
\mid & A ; A & \text{(guarantee)}
\end{array}
$$

Types

$$
\begin{array}{rlll}
A ::= & \boxed{!A \quad \text{(pure/persistent)}} & \mid & \mathbf{ref}\ p \quad \text{(reference type)} \\
\mid & A \multimap A \quad \text{(linear function)} & \mid & \mathbf{rec}\ X.A \quad \text{(recursive type)} \\
\mid & A :: A \quad \text{(stacking)} & \mid & \sum_i \mathtt{l}_i\#A_i \quad \text{(tagged sum)} \\
\mid & A * A \quad \text{(separation)} & \mid & A \oplus A \quad \text{(alternative)} \\
\mid & [\overline{\mathtt{f} : A}] \quad \text{(record)} & \mid & A \,\&\, A \quad \text{(intersection)} \\
\mid & X \quad \text{(type variable)} & \mid & \mathbf{rw}\ p\ A \quad \text{(read-write capability to } p\text{)} \\
\mid & \forall X.A \quad \text{(universal type quantification)} & \mid & \mathbf{none} \quad \text{(empty capability)} \\
\mid & \exists X.A \quad \text{(existential type quantification)} & \mid & A \Rightarrow A \quad \text{(rely)} \\
\mid & \forall t.A \quad \text{(universal location quantification)} & \mid & A; A \quad \text{(guarantee)} \\
\mid & \exists t.A \quad \text{(existential location quantification)} & &
\end{array}
$$

Types

$$
\begin{array}{rlll}
A ::= & !A & \text{(pure/persistent)} \\
\mid & A \multimap A & \text{(linear function)} \\
\mid & A :: A & \text{(stacking)} \\
\mid & A * A & \text{(separation)} \\
\mid & [\overline{\mathtt{f} : A}] & \text{(record)} \\
\mid & X & \text{(type variable)} \\
\mid & \forall X.A & \text{(universal type quantification)} \\
\mid & \exists X.A & \text{(existential type quantification)} \\
\mid & \forall t.A & \text{(universal location quantification)} \\
\mid & \exists t.A & \text{(existential location quantification)} \\
\mid & \mathbf{ref}\ p & \text{(reference type)} \\
\mid & \mathbf{rec}\ X.A & \text{(recursive type)} \\
\mid & \sum_i \mathtt{l}_i \# A_i & \text{(tagged sum)} \\
\mid & A \oplus A & \text{(alternative)} \\
\mid & A \,\&\, A & \text{(intersection)} \\
\mid & \mathbf{rw}\ p\ A & \text{(read-write capability to } p\text{)} \\
\mid & \mathbf{none} & \text{(empty capability)} \\
\mid & A \Rightarrow A & \text{(rely)} \\
\mid & A ; A & \text{(guarantee)}
\end{array}
$$

Types

$$
\begin{array}{rlll}
A ::= & !A & \text{(pure/persistent)} \\
\mid & A \multimap A & \text{(linear function)} \\
\mid & A :: A & \text{(stacking)} \\
\mid & A * A & \text{(separation)} \\
\mid & \boxed{[\overline{\mathtt{f} : A}] \ \text{(record)}} & \\
\mid & X & \text{(type variable)} \\
\mid & \forall X.A & \text{(universal type quantification)} \\
\mid & \exists X.A & \text{(existential type quantification)} \\
\mid & \forall t.A & \text{(universal location quantification)} \\
\mid & \exists t.A & \text{(existential location quantification)} \\
\mid & \mathbf{ref}\ p & \text{(reference type)} \\
\mid & \mathbf{rec}\ X.A & \text{(recursive type)} \\
\mid & \sum_i \mathtt{l}_i \# A_i & \text{(tagged sum)} \\
\mid & A \oplus A & \text{(alternative)} \\
\mid & A \,\&\, A & \text{(intersection)} \\
\mid & \mathbf{rw}\ p\ A & \text{(read-write capability to } p\text{)} \\
\mid & \mathbf{none} & \text{(empty capability)} \\
\mid & A \Rightarrow A & \text{(rely)} \\
\mid & A ; A & \text{(guarantee)}
\end{array}
$$

Types

$$
\begin{array}{llll}
A ::= & !A & \text{(pure/persistent)} \\
\mid & A \multimap A & \text{(linear function)} \\
\mid & A :: A & \text{(stacking)} \\
\mid & A * A & \text{(separation)} \\
\mid & [\overline{\mathtt{f} : A}] & \text{(record)} \\
\mid & X & \text{(type variable)} \\
\mid & \forall X.A & \text{(universal type quantification)} \\
\mid & \exists X.A & \text{(existential type quantification)} \\
\mid & \forall t.A & \text{(universal location quantification)} \\
\mid & \exists t.A & \text{(existential location quantification)} \\
\mid & \mathbf{ref}\ p & \text{(reference type)} \\
\mid & \mathbf{rec}\ X.A & \text{(recursive type)} \\
\mid & \sum_i \boxed{\mathtt{l}_i\#A_i} & \text{(tagged sum)} \\
\mid & A \oplus A & \text{(alternative)} \\
\mid & A \,\&\, A & \text{(intersection)} \\
\mid & \mathbf{rw}\ p\ A & \text{(read-write capability to } p\text{)} \\
\mid & \mathbf{none} & \text{(empty capability)} \\
\mid & A \Rightarrow A & \text{(rely)} \\
\mid & A; A & \text{(guarantee)}
\end{array}
$$

Types

$$
\begin{array}{rlll}
A ::= & !A & \text{(pure/persistent)} \\
| & A \multimap A & \text{(linear function)} \\
| & A :: A & \text{(stacking)} \\
| & A * A & \text{(separation)} \\
| & [\overline{\mathtt{f} : A}] & \text{(record)} \\
| & X & \text{(type variable)} \\
| & \forall X.A & \text{(universal type quantification)} \\
| & \exists X.A & \text{(existential type quantification)} \\
| & \forall t.A & \text{(universal location quantification)} \\
| & \exists t.A & \text{(existential location quantification)} \\
| & \mathbf{ref}\ p & \text{(reference type)} \\
| & \mathbf{rec}\ X.A & \text{(recursive type)} \\
| & \sum_i \mathtt{l}_i\#A_i & \text{(tagged sum)} \\
| & A \oplus A & \text{(alternative)} \\
| & A \,\&\, A & \text{(intersection)} \\
| & \mathbf{rw}\ p\ A & \text{(read-write capability to } p\text{)} \\
| & \mathsf{none} & \text{(empty capability)} \\
| & A \Rightarrow A & \text{(rely)} \\
| & A; A & \text{(guarantee)}
\end{array}
$$

1. Protocol Specification

$$\begin{aligned} P \ ::= \ & \mathbf{rec}\, X.P \mid X \mid P \oplus P \mid P \,\&\, P \\ & \mid A \Rightarrow P \mid A; P \mid \textsf{none} \end{aligned}$$

1. Protocol Specification

$$P ::= \boxed{\mathbf{rec}\ X.P \mid X} \mid P \oplus P \mid P \,\&\, P$$
$$\mid A \Rightarrow P \mid A; P \mid \textsf{none}$$

1. Protocol Specification

$$P ::= \mathbf{rec}\, X.P \mid X \mid \boxed{P \oplus P} \mid P \,\&\, P \mid A \Rightarrow P \mid A; P \mid \textsf{none}$$

1. Protocol Specification

$$P ::= \mathbf{rec}\, X.P \mid X \mid P \oplus P \mid \boxed{P \,\&\, P} \mid A \Rightarrow P \mid A; P \mid \textsf{none}$$

1. Protocol Specification

$$P \ ::= \ \textbf{rec}\, X.P \mid X \mid P \oplus P \mid P \,\&\, P$$
$$\mid \boxed{A \Rightarrow P} \mid A; P \mid \textbf{none}$$

The shared state satisfies **A**, and requires the alias to obey the guarantee **P**.

1. Protocol Specification

$$P \ ::= \ \mathbf{rec}\ X.P \mid X \mid P \oplus P \mid P \,\&\, P$$
$$\mid A \Rightarrow P \mid \boxed{A; P} \mid \textbf{none}$$

Requires the client to establish (guarantee) that the shared state satisfies **`A`** before continuing the use of the protocol as **`P`**.

1. Protocol Specification

$$P ::= \mathbf{rec}\, X.P \mid X \mid P \oplus P \mid P \,\&\, P$$
$$\mid A \Rightarrow P \mid A; P \mid \boxed{\textsf{none}}$$

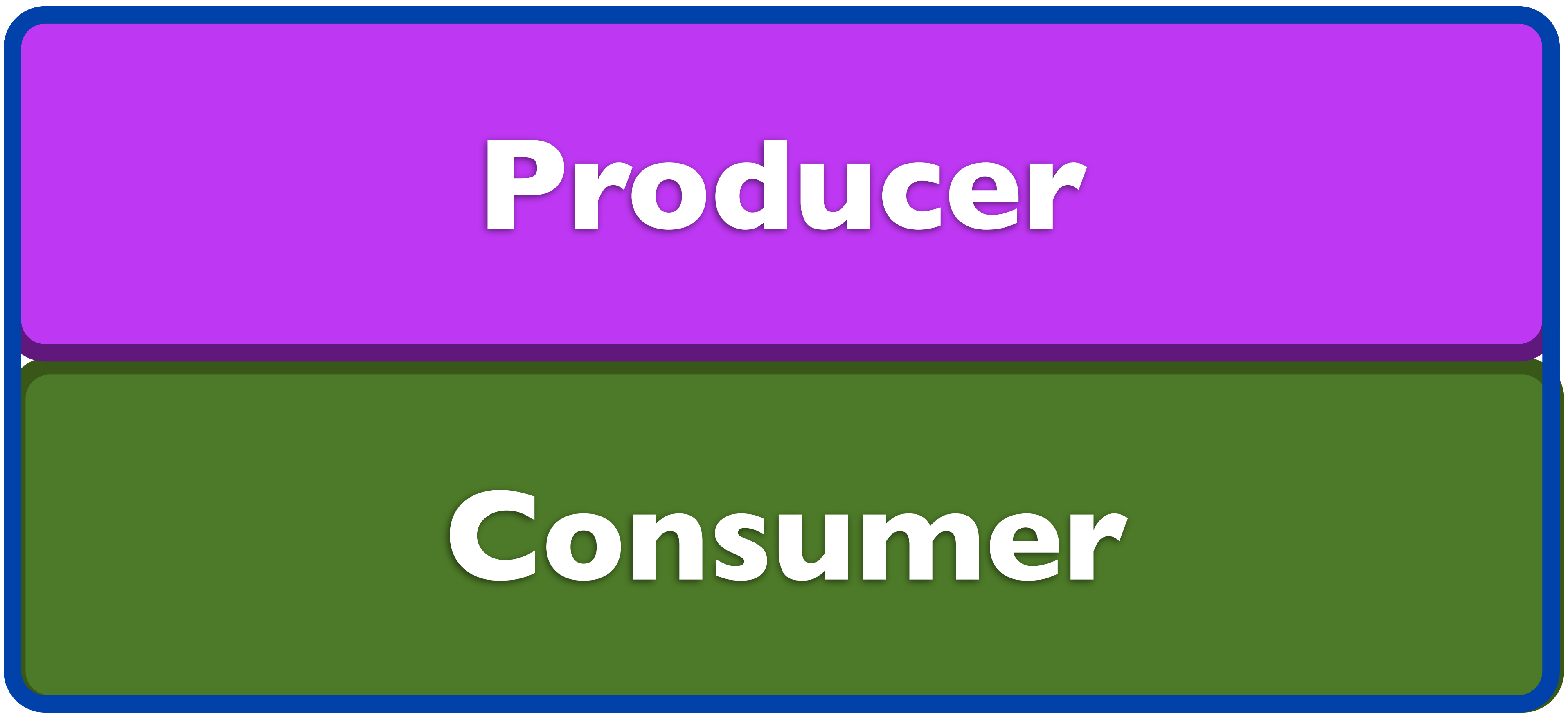
Shared Pipe

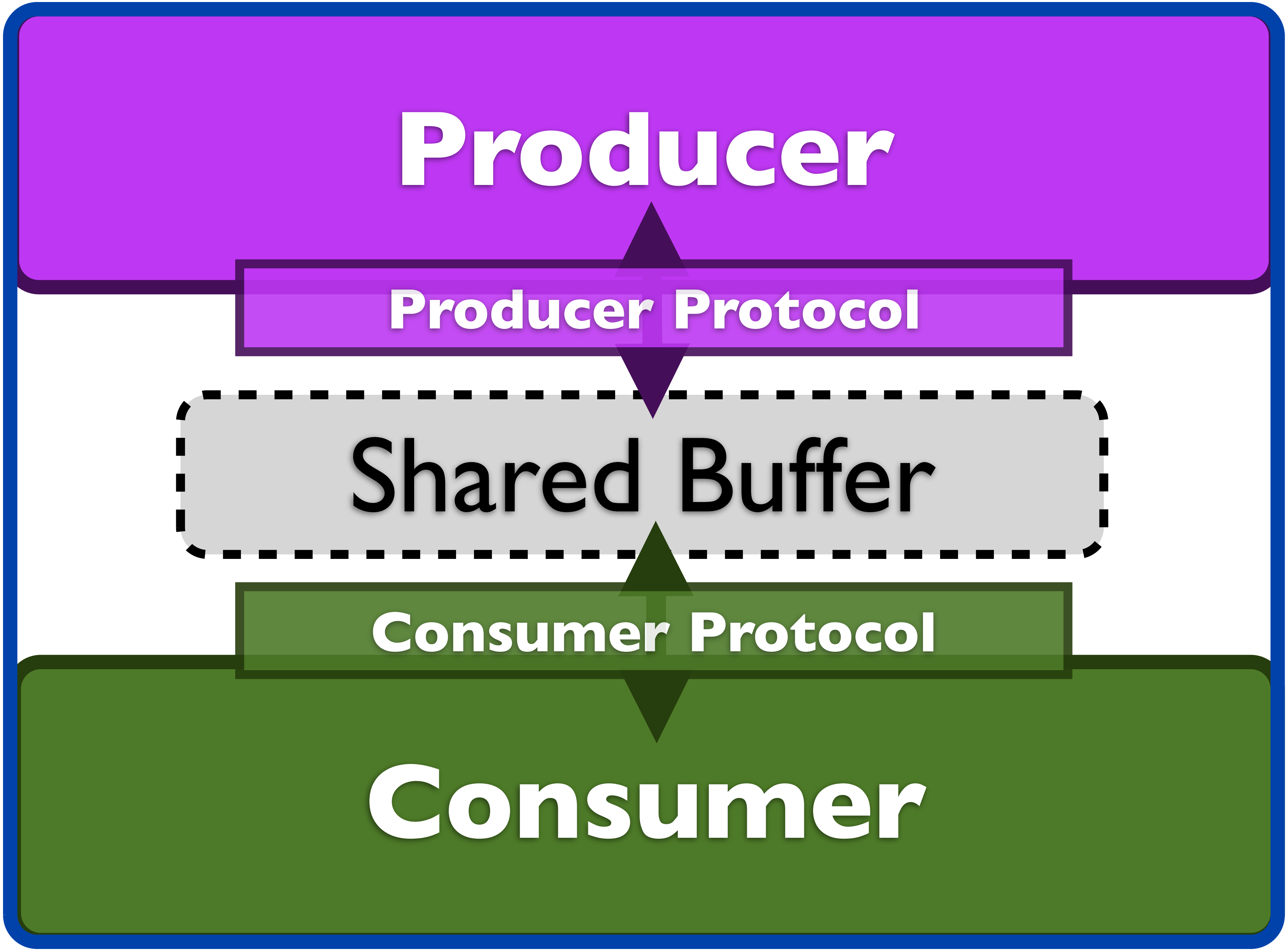
Shared by two aliases interacting via a common buffer, here modeled as a *singly linked list*.

1. The **`Producer`** alias may **`put`** new elements in or **`close`** the pipe.
2. The **`Consumer`** alias may only **`tryTake`** elements from the buffer.

   The result of **`tryTake`** is one of the following states: either there was some **`Result`**, or **`NoResult`**, or the pipe is fully **`Depleted`**.

Pipe


Producer
Consumer

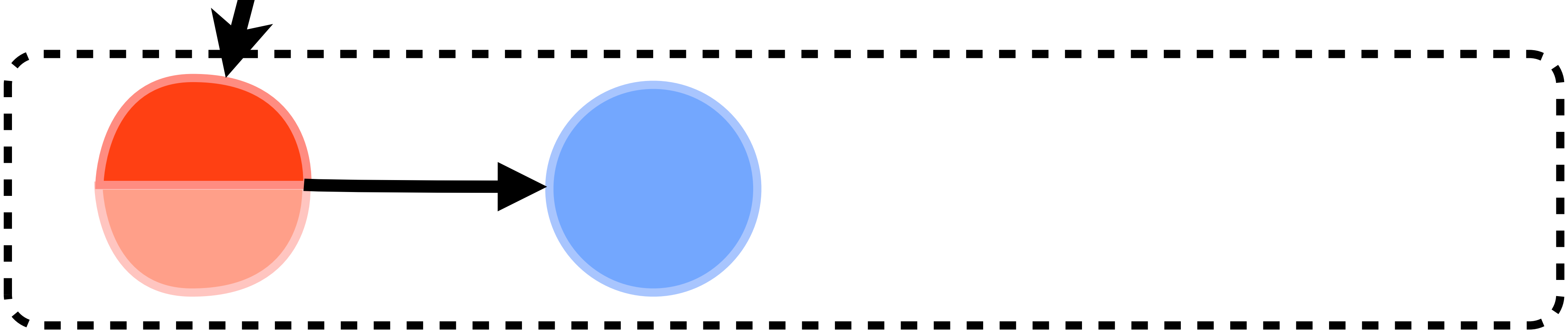

Producer
Producer Protocol
Shared Buffer
Consumer Protocol
Consumer

Producer

$\texttt{tail} : \textbf{Empty} \Rightarrow (\textbf{Filled} \oplus \textbf{Closed}) \; ; \textbf{none}$

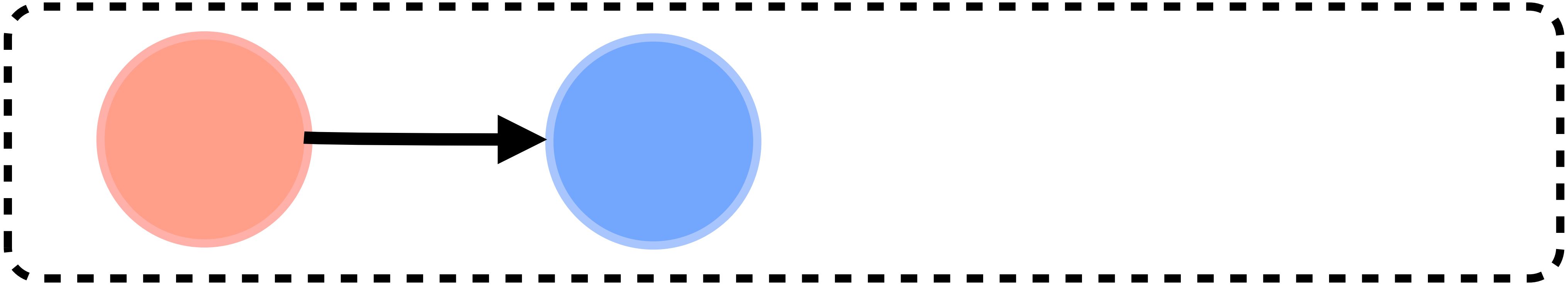
Producer

`tail` : **Empty** ⇒ (**Filled** ⊕ **Closed**) ; **none**



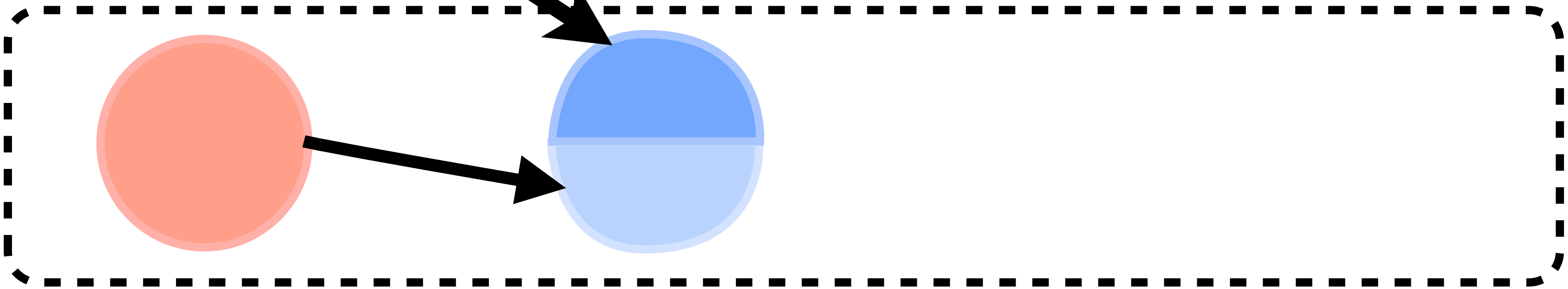
Producer

`tail : none`



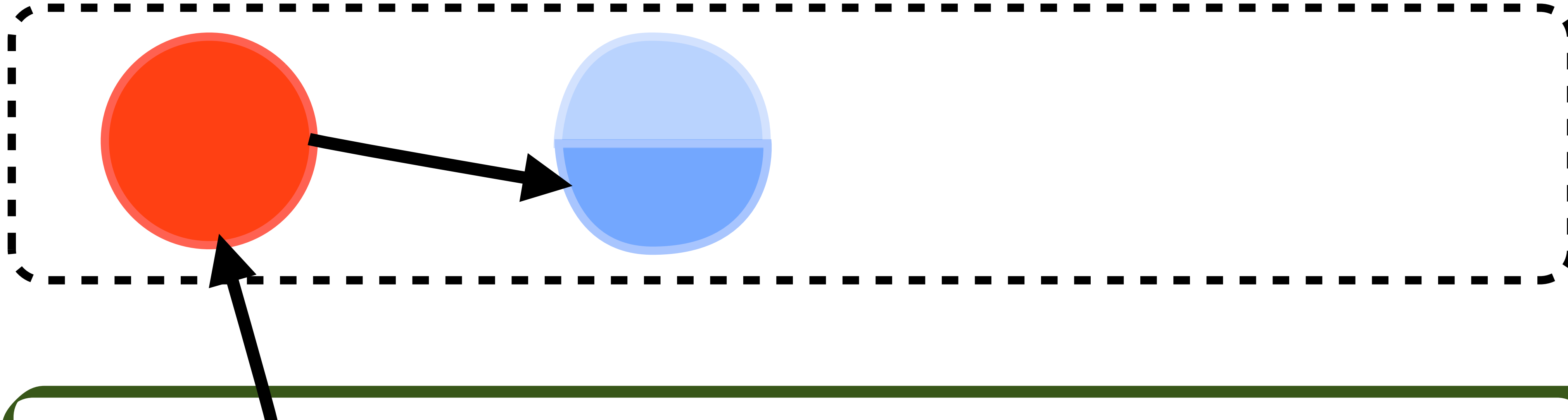
Producer

$\texttt{tail} : \textbf{Empty} \Rightarrow (\, \textbf{Filled} \oplus \textbf{Closed} \,) \,;\, \textbf{none}$



Producer

`tail` : **Empty** $\Rightarrow$ (**Filled** $\oplus$ **Closed**) ; **none**



`head` : **rec** X.((**Empty** $\Rightarrow$ **Empty** ; X)

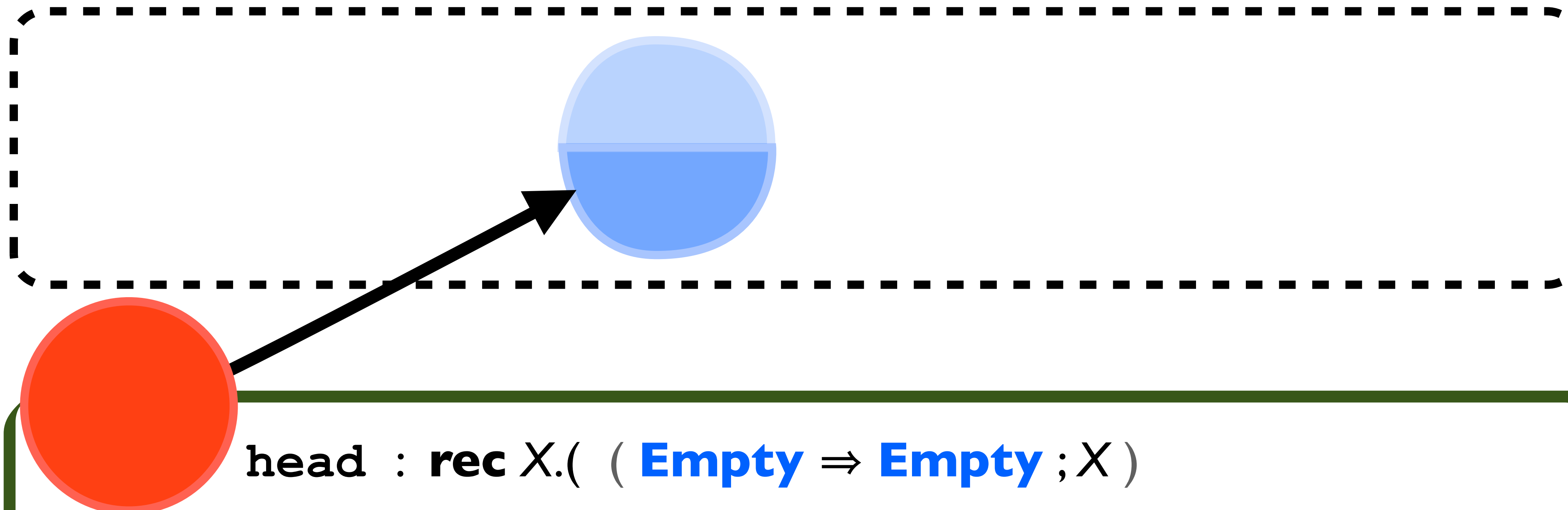
$\oplus$ (**Filled** $\Rightarrow$ **none** ; **none**)

$\oplus$ (**Closed** $\Rightarrow$ **none** ; **none**))

Consumer

Producer

`tail` : **Empty** $\Rightarrow$ (**Filled** $\oplus$ **Closed**) ; **none**



`head` : **rec** X.((**Empty** $\Rightarrow$ **Empty** ; X)

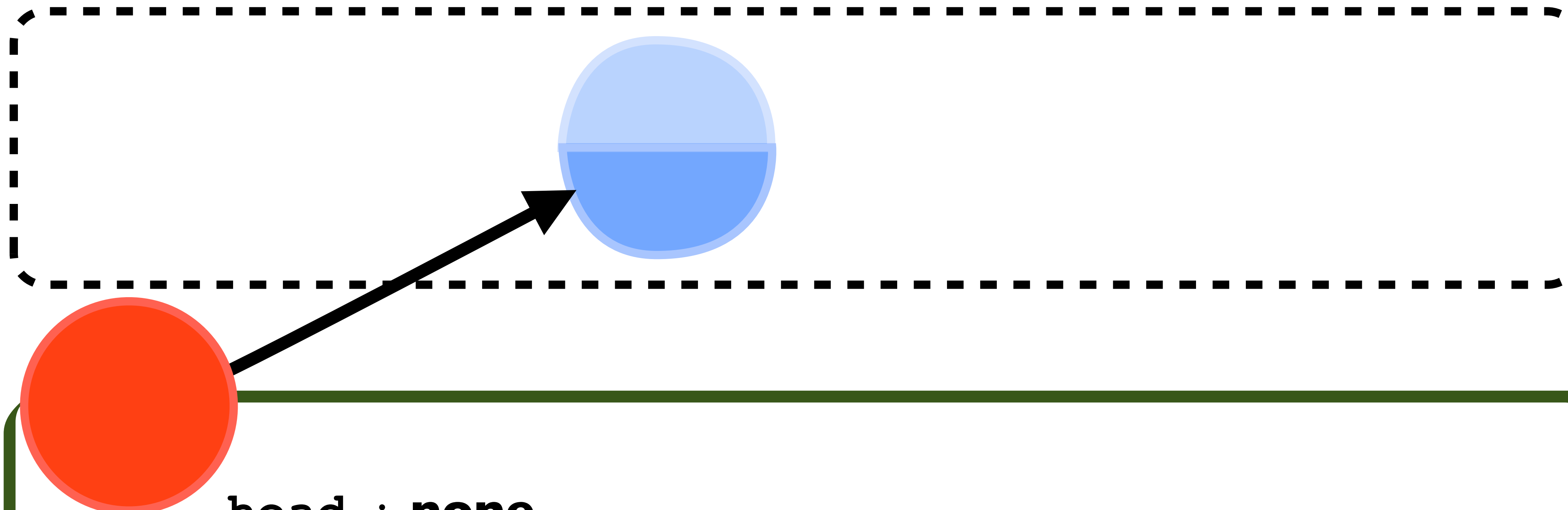
$\oplus$ (**Filled** $\Rightarrow$ **none** ; **none**)

$\oplus$ (**Closed** $\Rightarrow$ **none** ; **none**))

Consumer

Producer

$\texttt{tail} : \textsf{Empty} \Rightarrow (\textsf{Filled} \oplus \textsf{Closed})\ ; \textsf{none}$

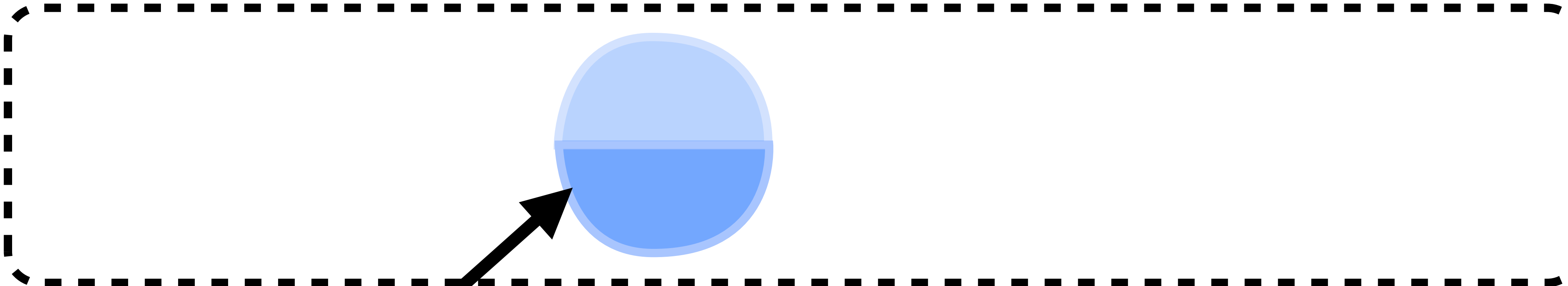


$\texttt{head} : \textsf{none}$

Consumer

Producer

$\mathtt{tail} : \mathbf{Empty} \Rightarrow (\ \mathbf{Filled} \oplus \mathbf{Closed}\ ) ; \mathbf{none}$



$$\begin{aligned}\mathtt{head} : \mathbf{rec}\ X.(\ &(\ \mathbf{Empty} \Rightarrow \mathbf{Empty} ; X\ ) \\ &\oplus (\ \mathbf{Filled} \Rightarrow \mathbf{none} ; \mathbf{none}\ ) \\ &\oplus (\ \mathbf{Closed} \Rightarrow \mathbf{none} ; \mathbf{none}\ )\ )\end{aligned}$$

Consumer

`Producer` **Protocol**:

$\texttt{tail} : \textbf{Empty} \Rightarrow (\textbf{Filled} \oplus \textbf{Closed}) ; \textbf{none}$

`Consumer` **Protocol**:

$$\begin{aligned}\texttt{head} : \textbf{rec}\, X.(\quad & (\textbf{Empty} \Rightarrow \textbf{Empty} ; X) \\ \oplus\; & (\textbf{Filled} \Rightarrow \textbf{none} ; \textbf{none}) \\ \oplus\; & (\textbf{Closed} \Rightarrow \textbf{none} ; \textbf{none}) \quad)\end{aligned}$$

Producer Protocol:

```
T[t] = rw t Empty#[] ⇒
  ( (rw t Node#[...]) ⊕ (rw t Closed#[]) );
      none
```

Consumer Protocol:

```
H[t] = rec X.(
    ( rw t Empty#[] ⇒ rw t Empty#[] ; X )
  ⊕ ( rw t Node#[...] ⇒ none ; none )
  ⊕ ( rw t Closed#[] ⇒ none ; none ) )
```

Pipe Typestate

```
∃P.∃C.( ![
      put : !( ( !int :: P ) ⊸ ( ![] :: P ) ) ,
    close : !( ( ![] :: P ) ⊸ ![] ) ,
  tryTake : !( ( ![] :: C ) ⊸ Depleted#![] +
      NoResult#(![] :: C) + Result#(!int :: C) )
 ] :: ( C * P ) )
```

Pipe Typestate

```
∃P.∃C.( ![
      put : !( ( !int :: P ) ⊸ ( ![] :: P ) ) ,
    close : !( ( ![] :: P ) ⊸ ![] ) ,
  tryTake : !( ( ![] :: C ) ⊸ Depleted#![] +
      NoResult#(![] :: C) + Result#(!int :: C) )
 ] :: ( C * P ) )
```

Pipe Typestate

```
∃P.∃C.( ![
      put : !( ( !int :: P ) ⊸ ( ![] :: P ) ) ,
    close : !( ( ![] :: P ) ⊸ ![] ) ,
  tryTake : !( ( ![] :: C ) ⊸ Depleted#![] +
      NoResult#(![] :: C) + Result#(!int :: C) )
 ] :: ( C * P ) )
```

rw p ∃p.(ref p :: T[p])

Pipe Typestate

```
∃P.∃C.( ![
      put : !( ( !int :: P ) ⊸ ( ![] :: P ) ) ,
    close : !( ( ![] :: P ) ⊸ ![] ) ,
  tryTake : !( ( ![] :: C ) ⊸ Depleted#![] +
      NoResult#(![] :: C) + Result#(!int :: C) )
] :: ( C * P ) )
```

```
rw p ∃p.(ref p :: T[p])
```

```
rw c ∃p.(ref p :: H[p])
```

Problems of Sharing

1. Account for **interference** (*public* changes).

   Consider all possible interleaved uses of aliases and how they may change the shared state.

2. Handle **private** changes.

   Making sure other aliases do not see any intermediate or inconsistent states of the shared state (which may appear due to type changing assignments like **`int`** to **`string`**, etc.).

Syntax

$$
\begin{array}{lll}
v ::= \rho & \text{(address)} \\
\quad \mid\ x & \text{(variable)} \\
\quad \mid\ \mathsf{fun}(x : A).e & \text{(function)} \\
\quad \mid\ \langle t \rangle\, e & \text{(location abstraction)} \\
\quad \mid\ \langle X \rangle\, e & \text{(type abstraction)} \\
\quad \mid\ \langle p, v \rangle & \text{(pack location)} \\
\quad \mid\ \langle A, v \rangle & \text{(pack type)} \\
\quad \mid\ \{\overline{\mathtt{f} = v}\} & \text{(record)} \\
\quad \mid\ \mathtt{l}\#v & \text{(tagged value)} \\
\\
e ::= v & \text{(value)} \\
\quad \mid\ v[p] & \text{(location application)} \\
\quad \mid\ v[A] & \text{(type application)} \\
\quad \mid\ v.\mathtt{f} & \text{(field)} \\
\quad \mid\ v\ v & \text{(application)} \\
\quad \mid\ \mathsf{let}\ x = e\ \mathsf{in}\ e\ \mathsf{end} & \text{(let)} \\
\quad \mid\ \mathsf{open}\ \langle t, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end} & \text{(open location)} \\
\quad \mid\ \mathsf{open}\ \langle X, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end} & \text{(open type)} \\
\quad \mid\ \mathsf{new}\ v & \text{(cell creation)} \\
\quad \mid\ \mathsf{delete}\ v & \text{(cell deletion)} \\
\quad \mid\ !v & \text{(dereference)} \\
\quad \mid\ v := v & \text{(assign)} \\
\quad \mid\ \mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{l}\#x \rightarrow e}\ \mathsf{end} & \text{(case)} \\
\quad \mid\ \mathsf{share}\ A_0\ \mathsf{as}\ A_1 \parallel A_2 & \text{(share)} \\
\quad \mid\ \mathsf{focus}\ \overline{A} & \text{(focus)} \\
\quad \mid\ \mathsf{defocus} & \text{(defocus)}
\end{array}
$$

Syntax

$v ::= \rho$	(address)	$\mid v.\mathtt{f}$	(field)
$\mid x$	(variable)	$\mid v\ v$	(application)
$\mid \mathsf{fun}(x : A).e$	(function)	$\mid \mathsf{let}\ x = e\ \mathsf{in}\ e\ \mathsf{end}$	(let)
$\mid \langle t \rangle\, e$	(location abstraction)	$\mid \mathsf{open}\ \langle t, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end}$	(open location)
$\mid \langle X \rangle\, e$	(type abstraction)	$\mid \mathsf{open}\ \langle X, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end}$	(open type)
$\mid \langle p, v \rangle$	(pack location)	$\mid \mathsf{new}\ v$	(cell creation)
$\mid \langle A, v \rangle$	(pack type)	$\mid \mathsf{delete}\ v$	(cell deletion)
$\mid \{\overline{\mathtt{f} = v}\}$	(record)	$\mid\ !v$	(dereference)
$\mid \mathtt{l}\#v$	(tagged value)	$\mid v := v$	(assign)
		$\mid \mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{l}\#x \rightarrow e}\ \mathsf{end}$	(case)
$e ::= v$	(value)	$\mid \mathsf{share}\ A_0\ \mathsf{as}\ A_1 \parallel A_2$	(share)
$\mid v[p]$	(location application)	$\mid \mathsf{focus}\ \overline{A}$	(focus)
$\mid v[A]$	(type application)	$\mid \mathsf{defocus}$	(defocus)

)	$\mid\ v.1$	(field)
e)	$\mid\ v\ v$	(application
n)	$\mid$ let $x = e$ in e end	(let)
n abstraction)	$\mid$ open $\langle t, x \rangle = v$ in e end	(open locati
straction)	$\mid$ open $\langle X, x \rangle = v$ in e end	(open type)
cation)	$\mid$ new v	(cell creatio
pe)	$\mid$ delete v	(cell deletio
	$\mid\ !v$	(dereferenc
value)	$\mid\ v := v$	(assign)
	$\mid$ case v of $\overline{1\#x \to e}$ end	(case)
	$\mid$ share A_0 as $A_1 \mid\mid A_2$	(share)
n application)	$\mid$ focus $\overline{A}$	(focus)
plication)	$\mid$ defocus	(defocus)

2. Protocol Use

- Protocols are used through **`focus`** and **`defocus`** constructs.
- They serve two purposes:

a) **Hide *private* changes** from the other aliases of that shared state.

b) **Advance the step** of the protocol, by obeying the constraints on *public* changes.

Focus / Defocus

`focus` **Empty**

. . .

`defocus`

Focus / Defocus

Empty ⇒ Filled ; Next

`focus` **Empty**

. . .

`defocus`

Focus / Defocus

$\boxed{\textbf{Empty}} \Rightarrow$ **Filled** ; **Next**

`focus` **Empty**

...

`defocus`

Focus / Defocus

focused state

Empty ⇒ Filled ; Next

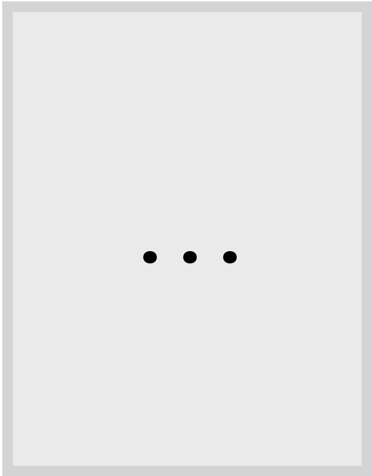
defocus-guarantee

```
focus Empty
  . . .
defocus
```

Focus / Defocus

Empty ⇒ Filled ; Next

`focus` **Empty**



`defocus`

Focus / Defocus

Empty ⇒ Filled ; Next

`focus` **Empty**

Empty , Filled ; Next

. . .

`defocus`

Focus / Defocus

Empty ⇒ Filled ; Next

`focus` Empty

Empty , Filled ; Next

PartiallyFilled , Filled ; Next

`defocus`

Focus / Defocus

Empty ⇒ Filled ; Next

`focus` Empty

Empty , Filled ; Next

. . .

Filled , Filled ; Next

`defocus`

Focus / Defocus

Empty ⇒ Filled ; Next

`focus` Empty

Empty , Filled ; Next

. . .

Filled , Filled ; Next

`defocus`

Focus / Defocus

Empty ⇒ Filled ; Next

`focus` Empty

Empty , Filled ; Next

. . .

Filled , Filled ; Next

`defocus`

Focus / Defocus

Empty ⇒ Filled ; Next

`focus` Empty

Empty , Filled ; Next

. . .

Filled , Filled ; Next

`defocus`

Next

Focus / Defocus

$\textbf{Empty} \Rightarrow \textbf{Filled} ; \textbf{Next} \quad , \quad \Delta$

`focus` **Empty**

private changes

$\textbf{Empty}, \quad \textbf{Filled} ; \textbf{Next} \triangleright \Delta$

. . .

$\textbf{Filled}, \quad \textbf{Filled}; \textbf{Next} \triangleright \Delta$

`defocus`

$\textbf{Next} \quad , \quad \Delta$

Focus / Defocus

Empty $\Rightarrow$ **Filled** ; **Next** , $\boxed{\Delta}$

`focus` **Empty**

private changes

Empty, **Filled** ; **Next** $\boxed{\triangleright\ \Delta}$

. . .

Filled, **Filled**; **Next** $\boxed{\triangleright\ \Delta}$

`defocus`

Next , $\boxed{\Delta}$

Focus / Defocus

$\text{Empty} \Rightarrow \text{Filled} \,;\, \text{Next} \;, \boxed{\Delta}$

`focus` Empty

private changes

$\text{Empty}, \;\; \text{Filled} \,;\, \text{Next} \boxed{\triangleright\, \Delta}$

hides any state that may allow reentrant accesses to focused state

. . .

$\text{Filled}, \;\; \text{Filled}; \text{Next} \boxed{\triangleright\, \Delta}$

`defocus`

$\text{Next} \;, \boxed{\Delta}$

Problems of Sharing

1. Account for **interference** (*public* changes).

   Consider all possible interleaved uses of aliases and how they may change the shared state.

2. Handle **private** changes.

   Making sure other aliases do not see any intermediate or inconsistent states of the shared state (which may appear due to type changing assignments like `int` to `string`, etc.).

3. Protocol Conformance

- Protocols are introduced explicitly, in pairs, through the **share** construct:

  ```
  share A as B || C
  ```

  "type **`A`** (either a capability or an existing protocol) can be safely *split* in types **`B`** and **`C`** (two protocols)"

- Arbitrary aliasing is possible by continuing to split an existing protocol.

Checking `share`

- We must check that a protocol is aware of all possible states that may appear due to the "interleaving" of other aliases of that shared state.
- Checking a split is built from two components:

a) a *stepping relation*, that "simulates" a single use of **focus-defocus** (i.e. a step of the protocol).

b) a *protocol conformance definition* that ensures the protocol considers all possible alias interleaving.

Protocol Conformance Example

```
share E as
  rec X.( E ⇒ E;X ⊕ N ⇒ none ⊕ C ⇒ none )
  || E ⇒ ( N ⊕ C )
```

Protocol Conformance Example

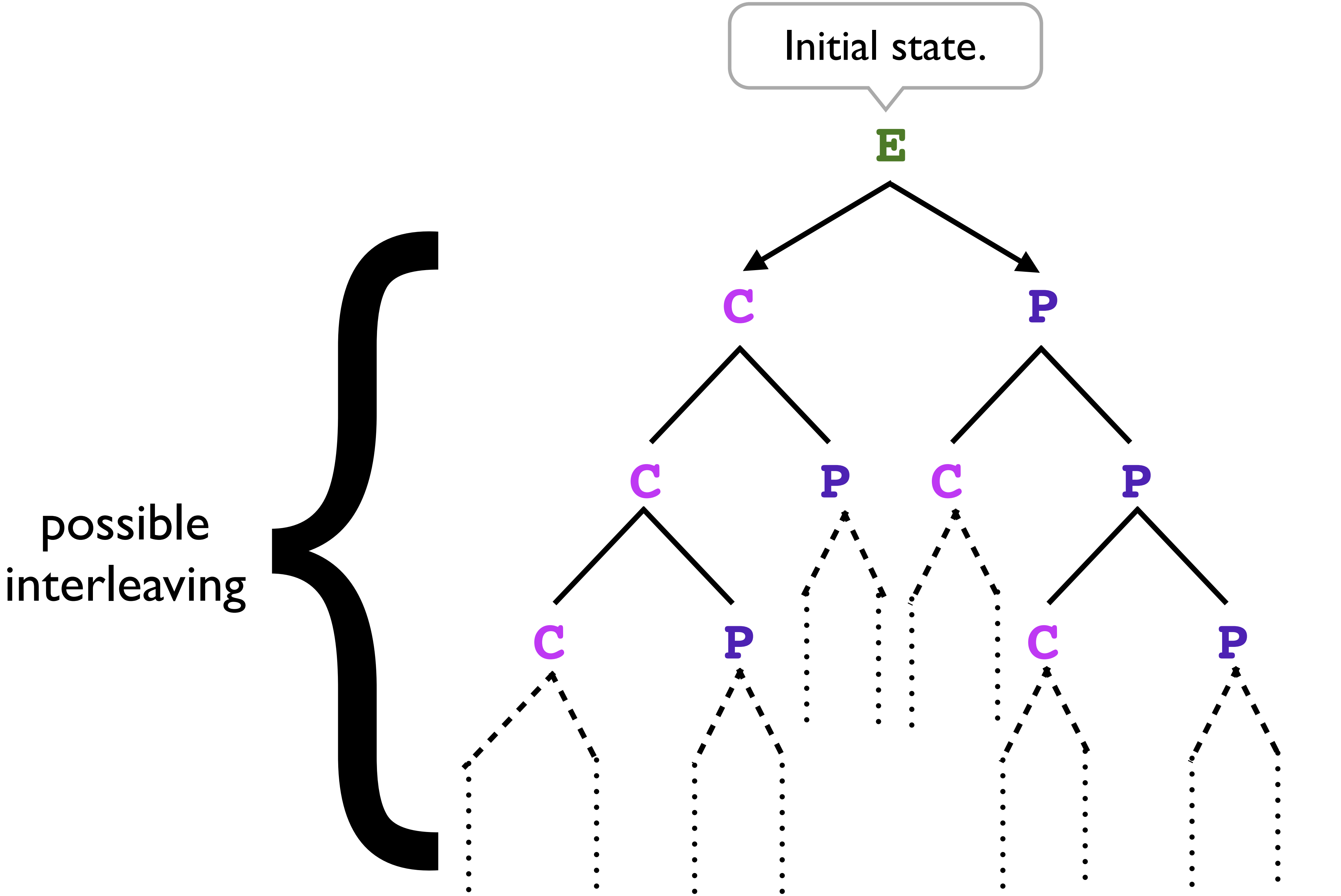
`share E as`

Consumer

$$\boxed{\texttt{rec X.(E} \Rightarrow \texttt{E;X} \oplus \texttt{N} \Rightarrow \texttt{none} \oplus \texttt{C} \Rightarrow \texttt{none)}}$$

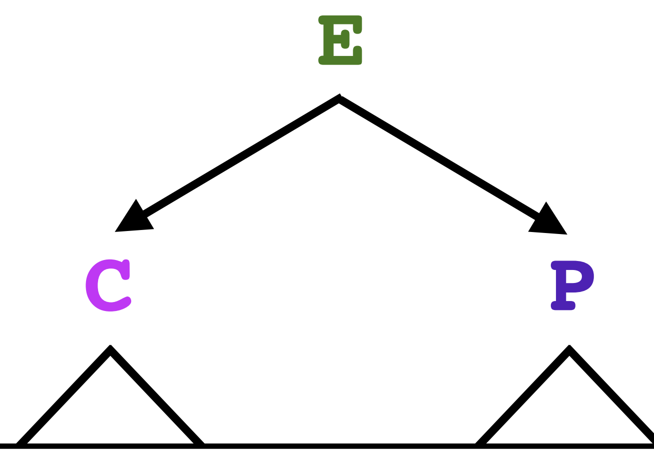
$$\texttt{||} \; \boxed{\texttt{E} \Rightarrow \texttt{(N} \oplus \texttt{C)}}$$

Producer



Initial state.
E
C
P
C
P
C
P
C
P
C
P
possible interleaving

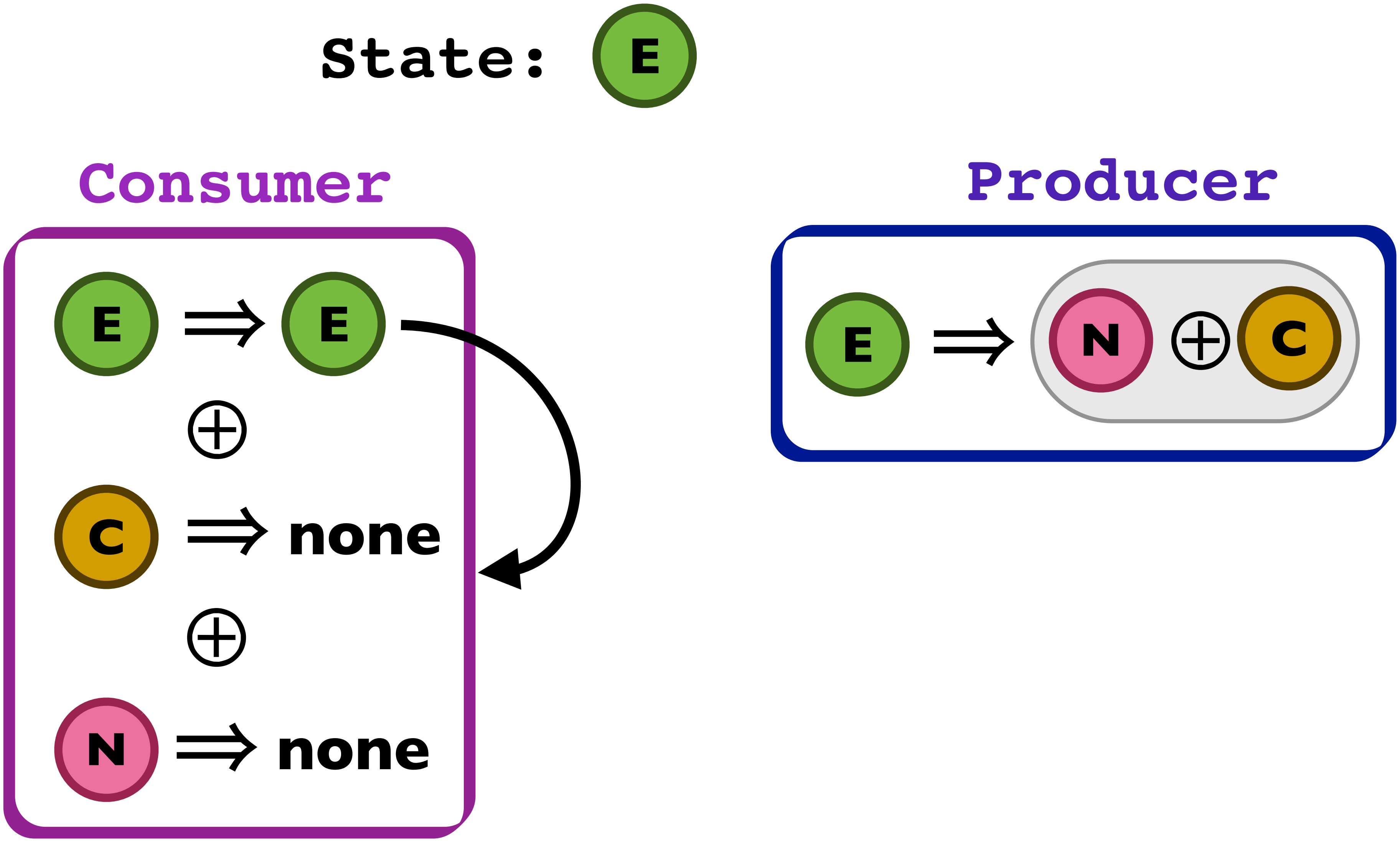
Initial state.



possible interleaving

However, our protocols can only list a **finite number of distinct states**, and each protocol lists a **finite number of distinct protocol steps**.

This will ensure that there is finite number of distinct *configurations*, each representing one possible alias interleaving in the use of the state that is being shared by the protocols.



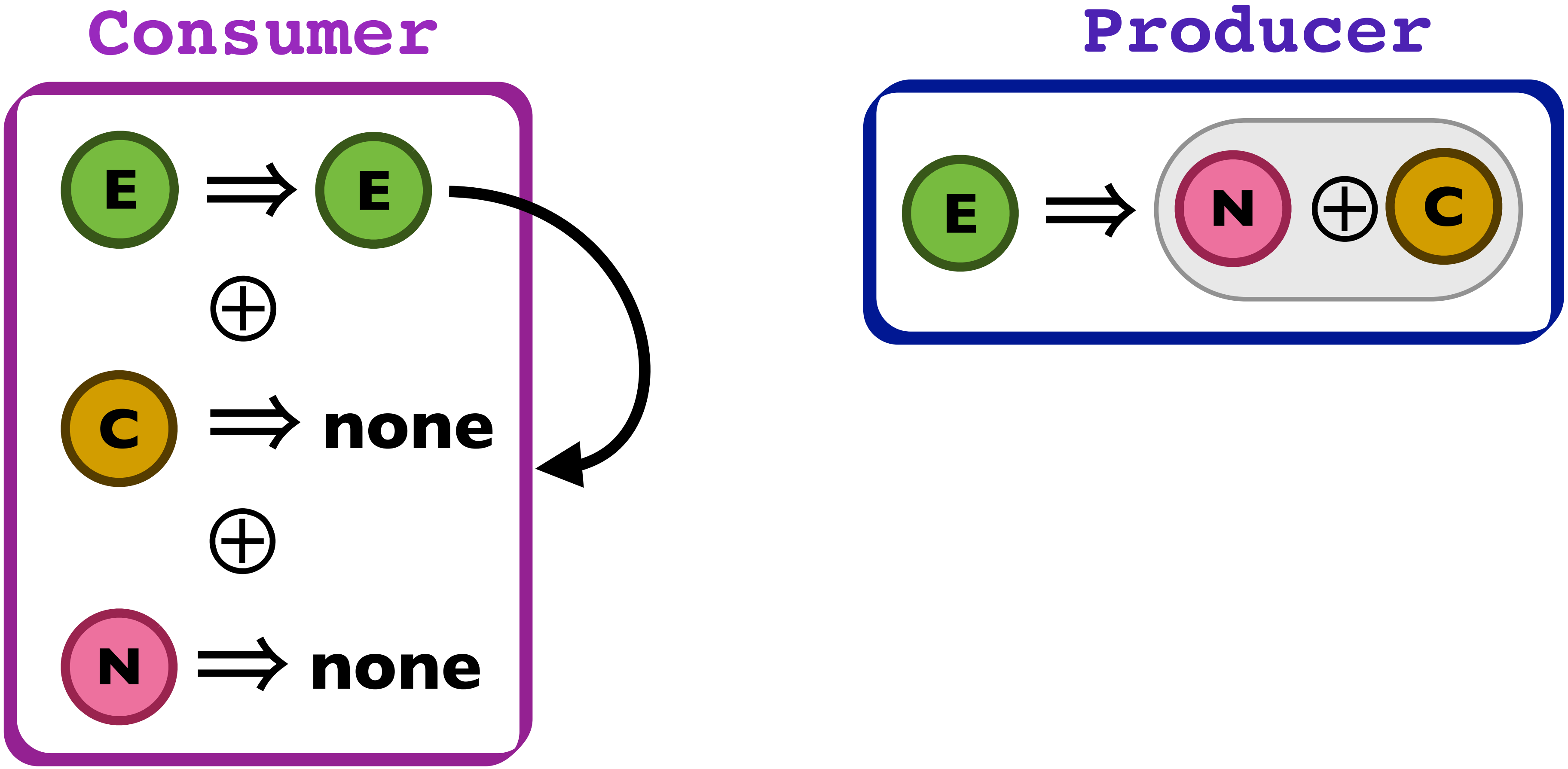
Configurations:

$$\langle \mathsf{E} \;\Rightarrow\; \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\, ;\, X\,) \parallel \mathsf{E} \Rightarrow (\, \mathsf{N} \oplus \mathsf{C}\,)\rangle$$

$$\langle \mathsf{N} \oplus \mathsf{C} \;\Rightarrow\; \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\, ;\, X\,) \parallel \textbf{none}\rangle$$

$$\langle \textbf{none} \;\Rightarrow\; \textbf{none} \parallel \textbf{none}\rangle$$

State:



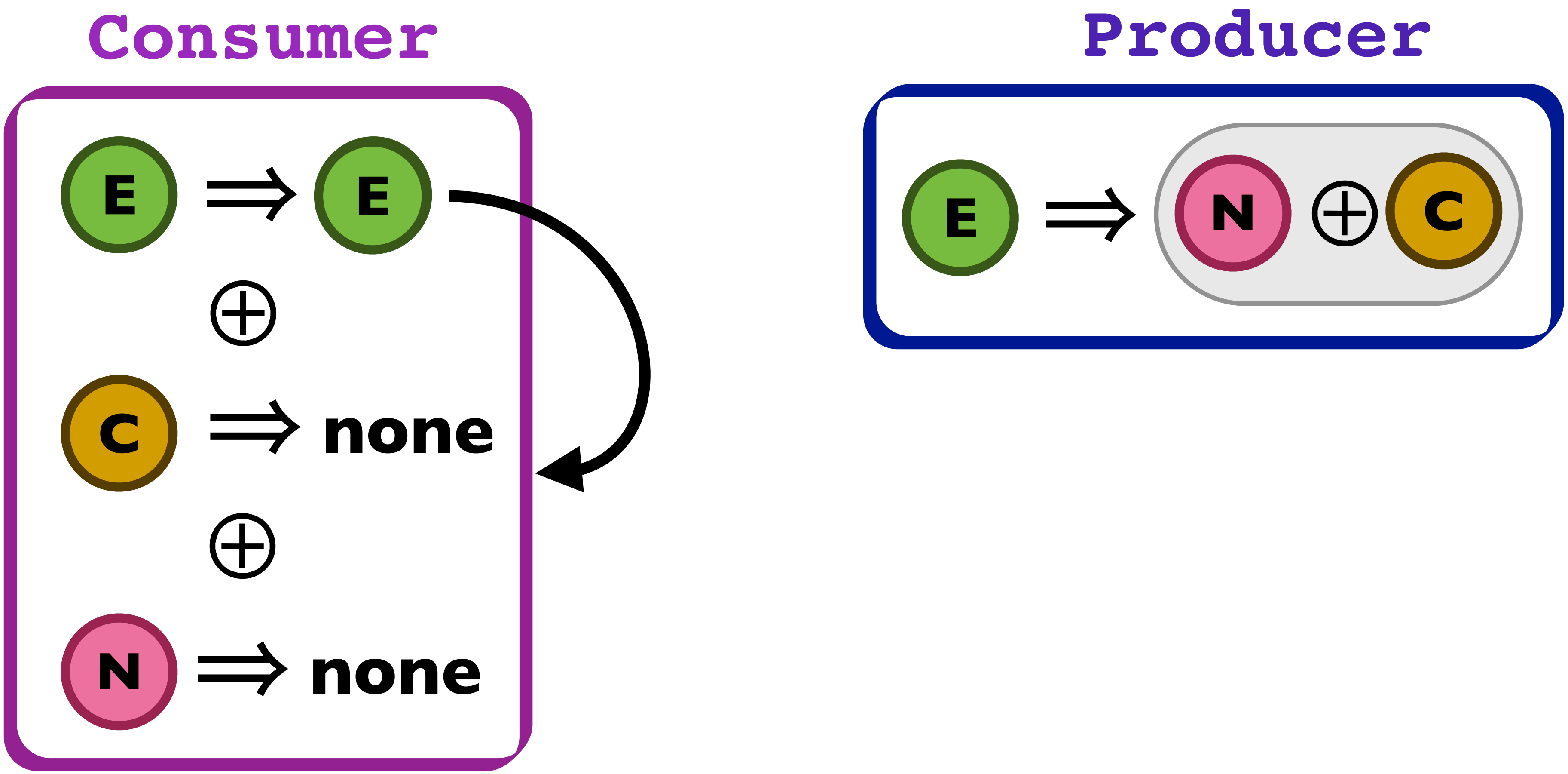
Configurations:

$\langle \mathsf{E} \Rightarrow \boxed{\mathbf{rec}\, X.(\mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\, ;\, X)} \parallel \mathsf{E} \Rightarrow (\mathsf{N} \oplus \mathsf{C}) \rangle$

$\langle \mathsf{N} \oplus \mathsf{C} \Rightarrow \mathbf{rec}\, X.(\mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\, ;\, X) \parallel \textbf{none} \rangle$

$\langle \textbf{none} \Rightarrow \textbf{none} \parallel \textbf{none} \rangle$

State:

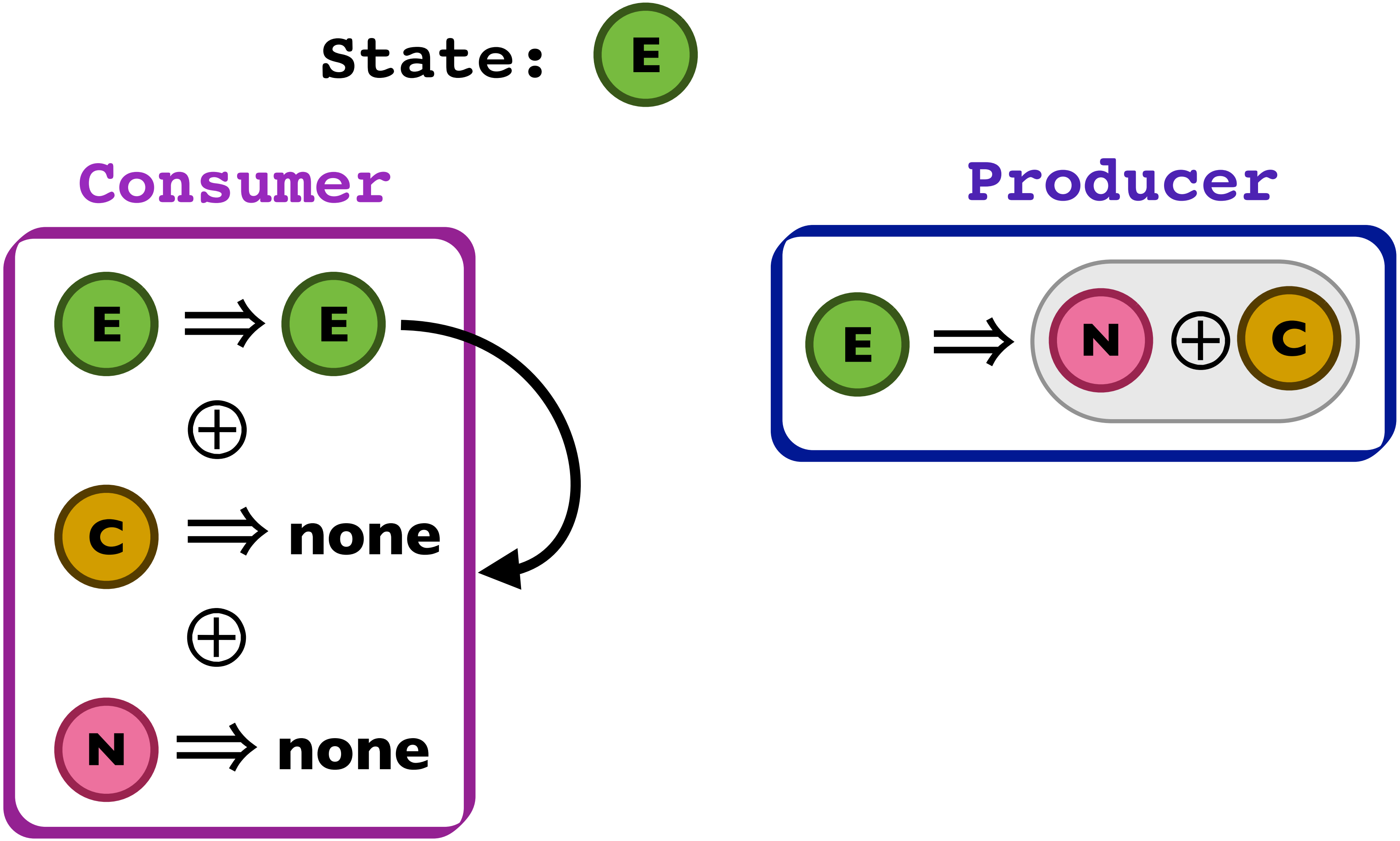


Configurations:

$$\langle \mathsf{E} \;\Rrightarrow\; \mathbf{rec}\, X.(\,\mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \boxed{\mathsf{E} \Rightarrow \mathsf{E}\, ;\, X}\,) \parallel \mathsf{E} \Rightarrow (\,\mathsf{N} \oplus \mathsf{C}\,)\rangle$$

$$\langle \mathsf{N} \oplus \mathsf{C} \;\Rrightarrow\; \mathbf{rec}\, X.(\,\mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\, ;\, X\,) \parallel \textbf{none}\rangle$$

$$\langle \textbf{none} \;\Rrightarrow\; \textbf{none} \parallel \textbf{none}\rangle$$

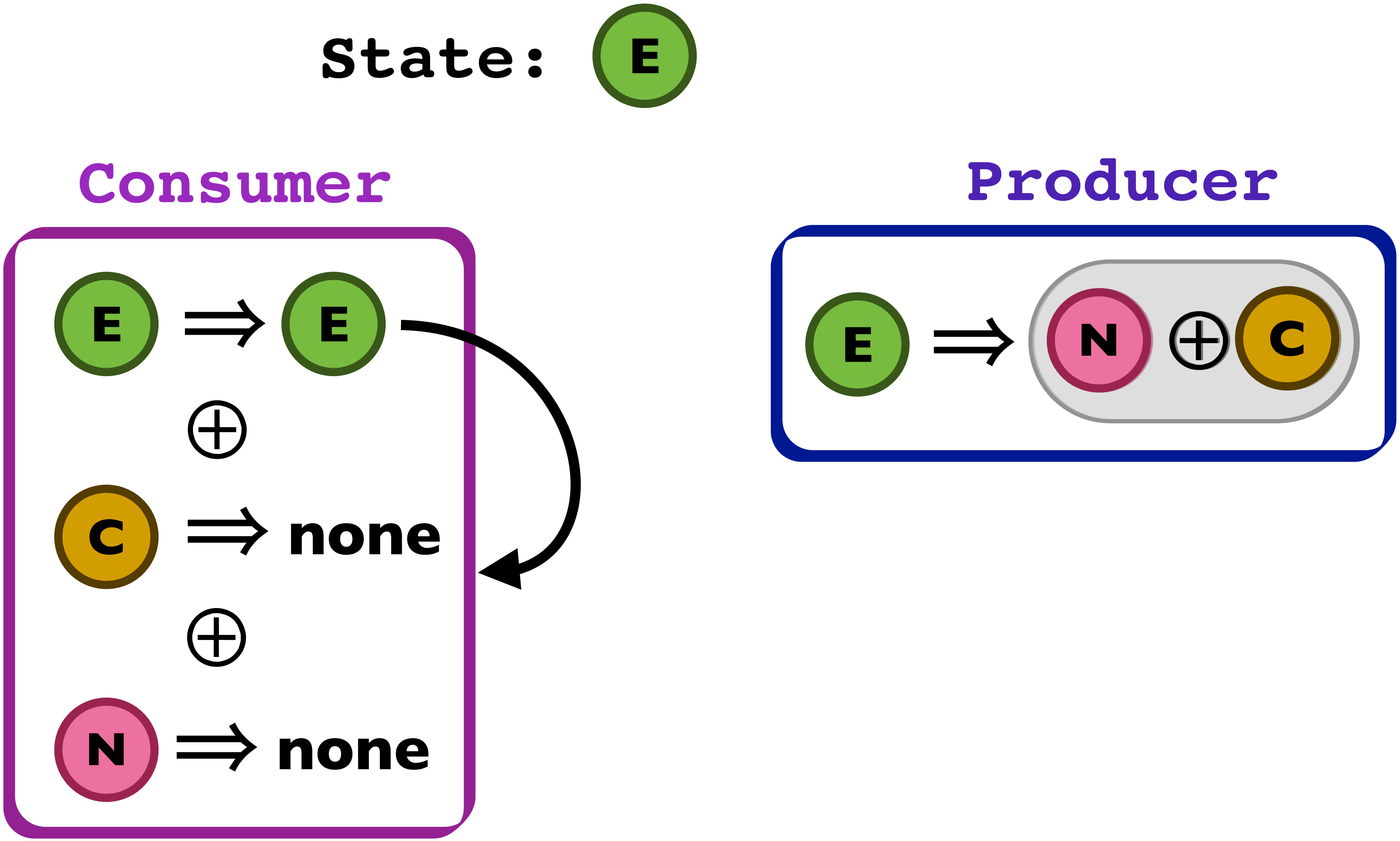


Configurations:

$$\langle \mathsf{E} \;\Rightarrow\; \mathbf{rec}\, X.(\,\mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\,;\, X\,) \parallel \mathsf{E} \Rightarrow (\,\mathsf{N} \oplus \mathsf{C}\,)\rangle$$

$$\langle \mathsf{N} \oplus \mathsf{C} \;\Rightarrow\; \mathbf{rec}\, X.(\,\mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\,;\, X\,) \parallel \textbf{none}\rangle$$

$$\langle \textbf{none} \;\Rightarrow\; \textbf{none} \parallel \textbf{none}\rangle$$



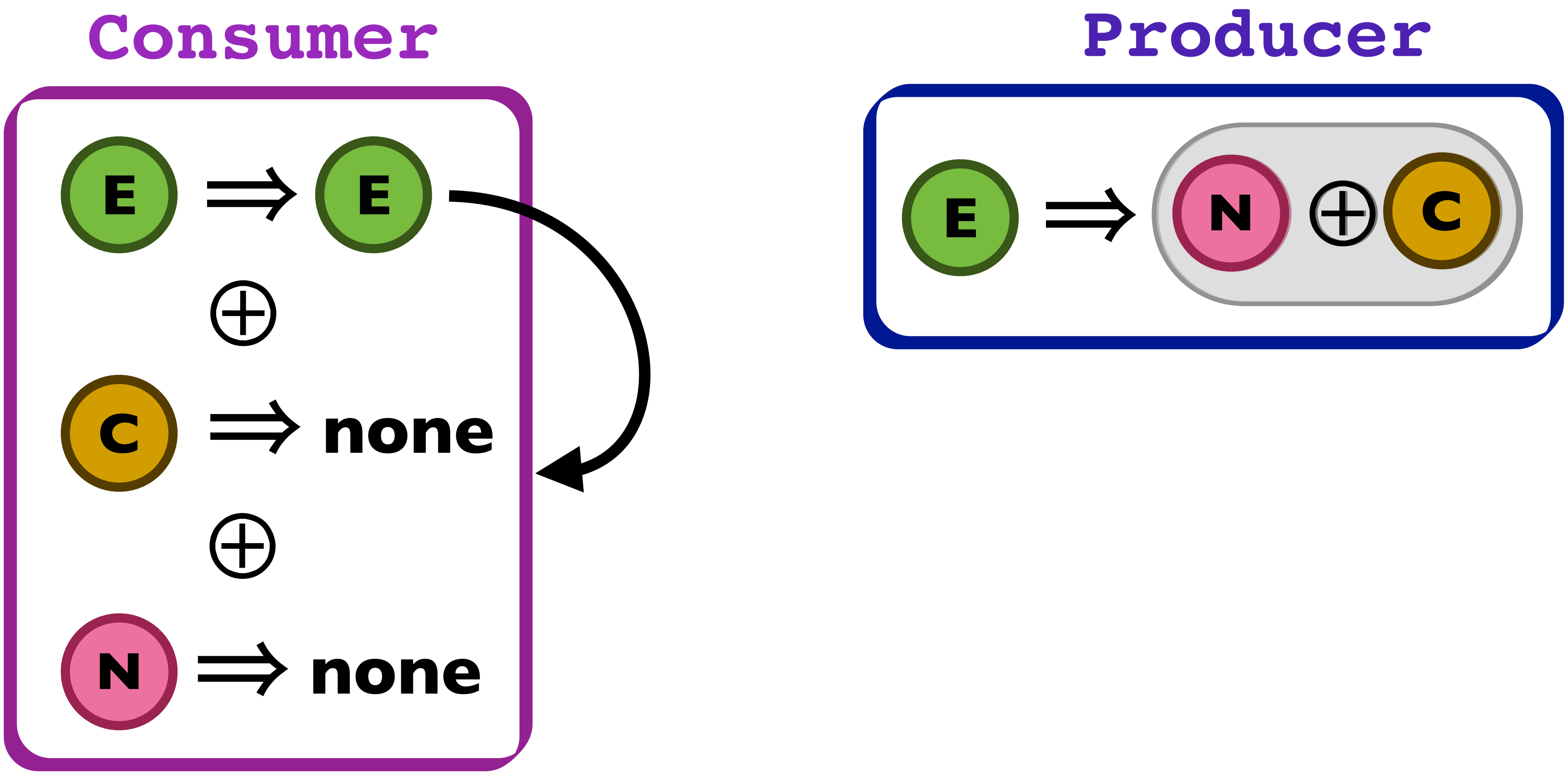
Configurations:

$\langle \mathsf{E} \quad \Rrightarrow \quad \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E} \,;\, X\,) \parallel \mathsf{E} \Rightarrow (\,\mathsf{N} \oplus \mathsf{C}\,)\rangle$

$\langle \mathsf{N} \oplus \mathsf{C} \quad \Rrightarrow \quad \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E} \,;\, X\,) \parallel \textbf{none}\rangle$

$\langle \textbf{none} \quad \Rrightarrow \quad \textbf{none} \parallel \textbf{none}\rangle$

State: E



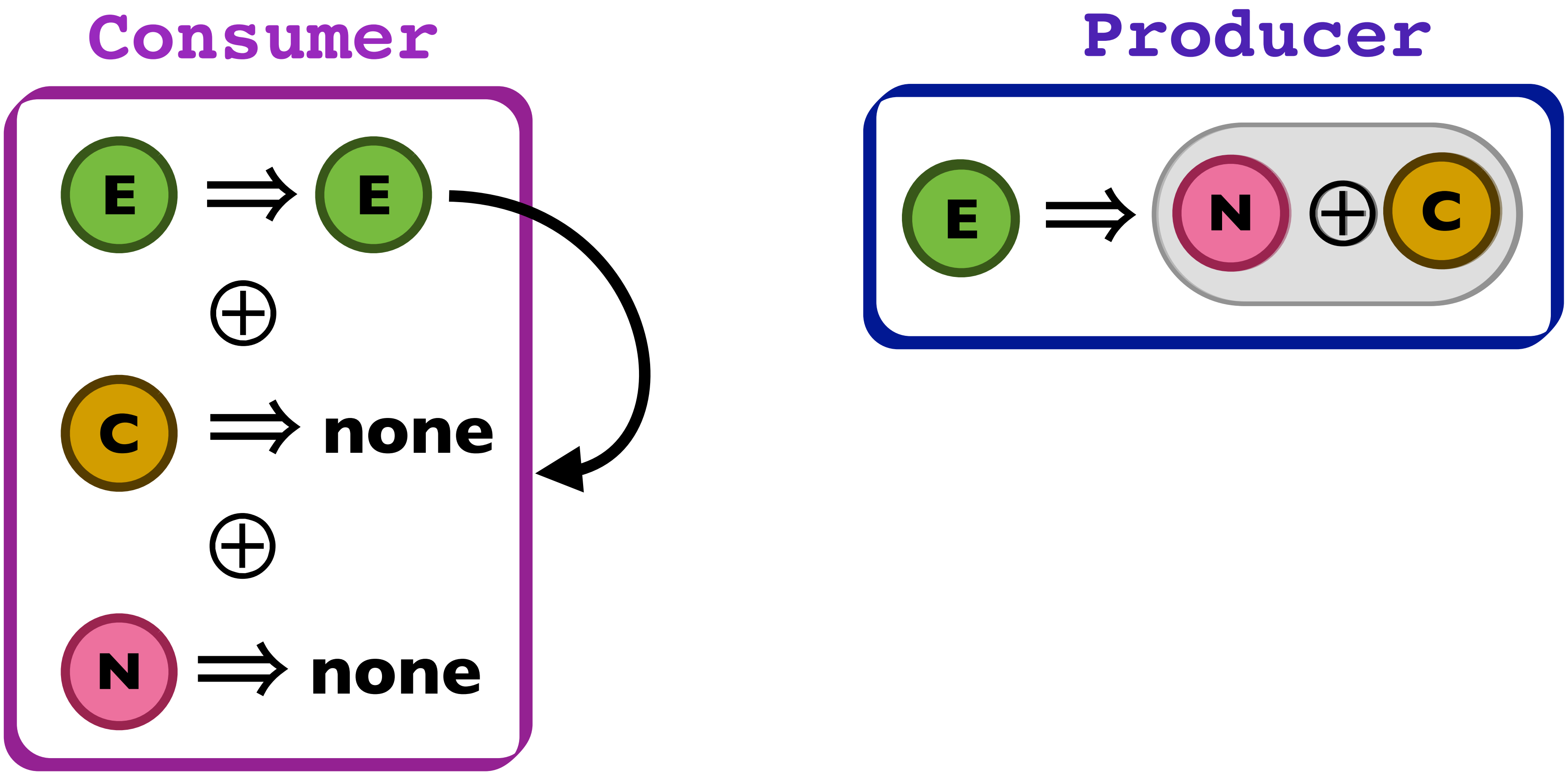
Configurations:

$$\langle \mathsf{E} \Rrightarrow \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E} \,;\, X\,) \parallel \boxed{\mathsf{E} \Rightarrow (\, \mathsf{N} \oplus \mathsf{C}\,)} \rangle$$

$$\langle \mathsf{N} \oplus \mathsf{C} \Rrightarrow \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E} \,;\, X\,) \parallel \textbf{none} \rangle$$

$$\langle \textbf{none} \Rrightarrow \textbf{none} \parallel \textbf{none} \rangle$$

State:



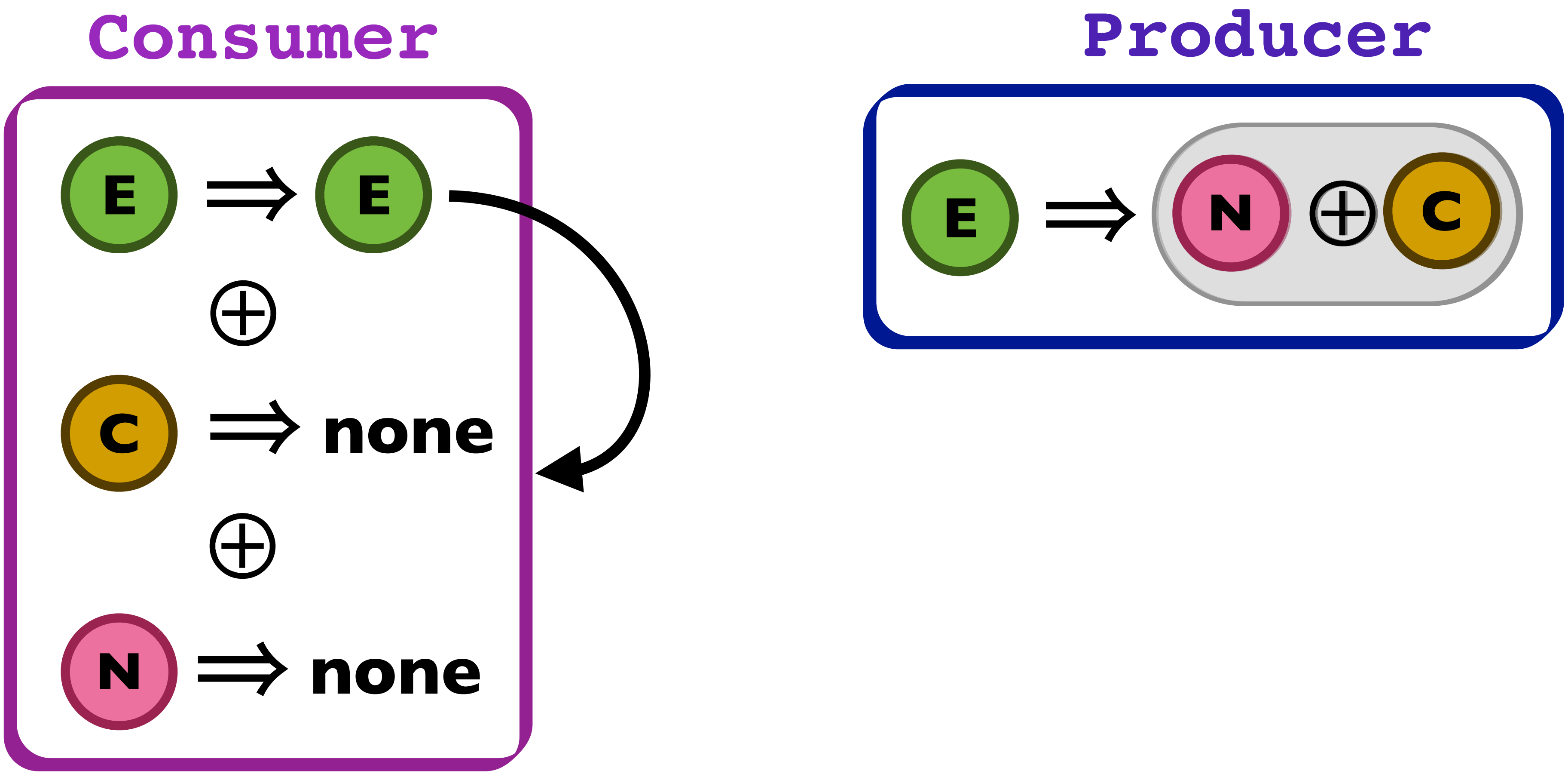
Configurations:

$\langle \mathsf{E} \;\Rightarrow\; \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \mathbf{none} \oplus \mathsf{C} \Rightarrow \mathbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E} \,;\, X\,) \parallel \boxed{\mathsf{E} \Rightarrow (\,\mathsf{N} \oplus \mathsf{C}\,)} \rangle$

$\langle \mathsf{N} \oplus \mathsf{C} \;\Rightarrow\; \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \mathbf{none} \oplus \mathsf{C} \Rightarrow \mathbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E} \,;\, X\,) \parallel \mathbf{none} \rangle$

$\langle \mathbf{none} \;\Rightarrow\; \mathbf{none} \parallel \mathbf{none} \rangle$

State:

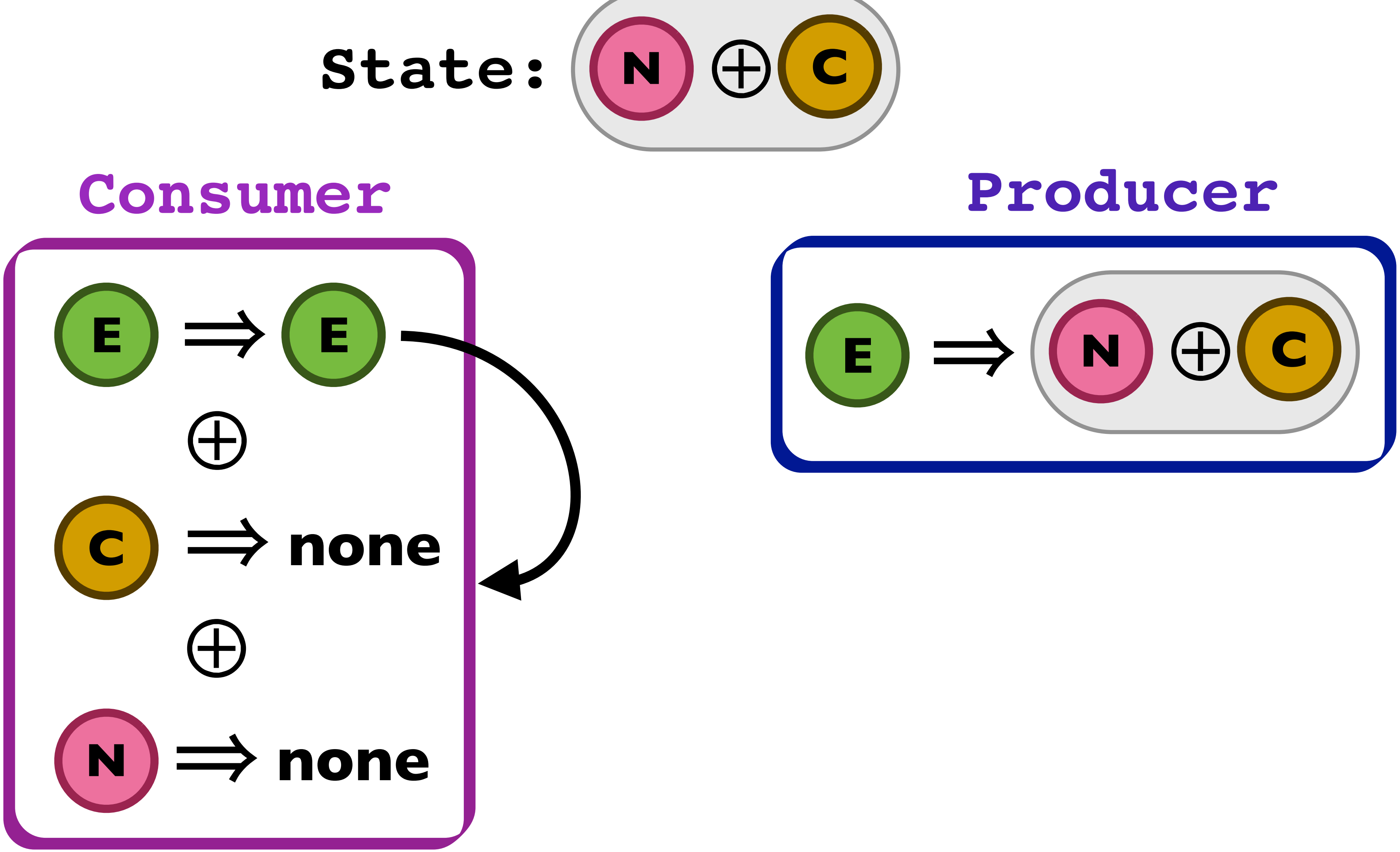


Configurations:

$\langle \mathsf{E} \Rrightarrow \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\, ;\, X\,) \parallel \boxed{\mathsf{E} \Rightarrow (\, \mathsf{N} \oplus \mathsf{C}\,)} \rangle$

$\langle \mathsf{N} \oplus \mathsf{C} \Rrightarrow \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\, ;\, X\,) \parallel \textbf{none} \rangle$

$\langle \textbf{none} \Rrightarrow \textbf{none} \parallel \textbf{none} \rangle$

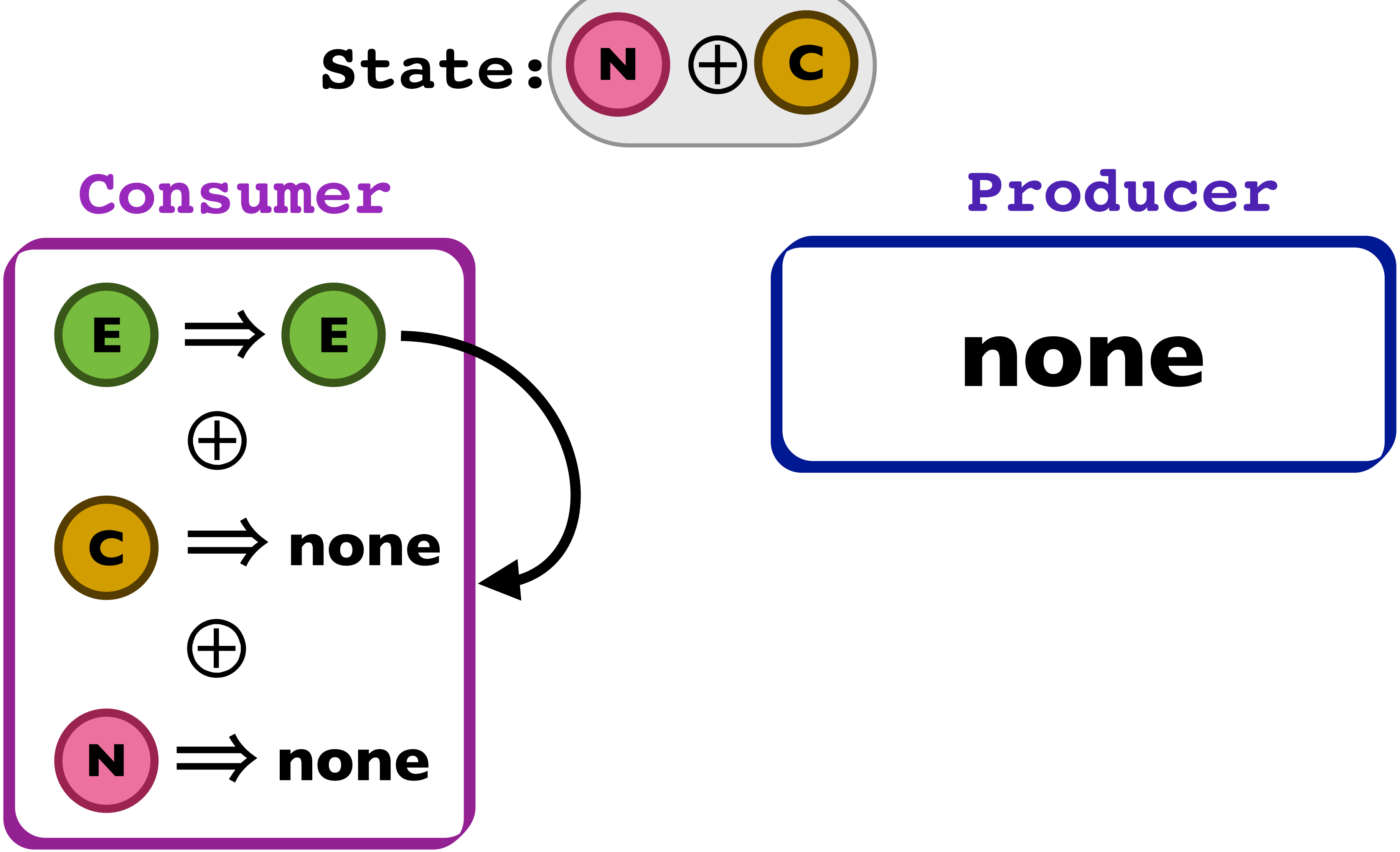


Configurations:

$\langle \mathsf{E} \;\Rrightarrow\; \mathbf{rec}\, X.(\,\mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\,;\, X\,) \parallel \mathsf{E} \Rightarrow (\,\mathsf{N} \oplus \mathsf{C}\,)\rangle$

$\langle \mathsf{N} \oplus \mathsf{C} \;\Rrightarrow\; \mathbf{rec}\, X.(\,\mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\,;\, X\,) \parallel \textbf{none}\rangle$

$\langle \textbf{none} \;\Rrightarrow\; \textbf{none} \parallel \textbf{none}\rangle$

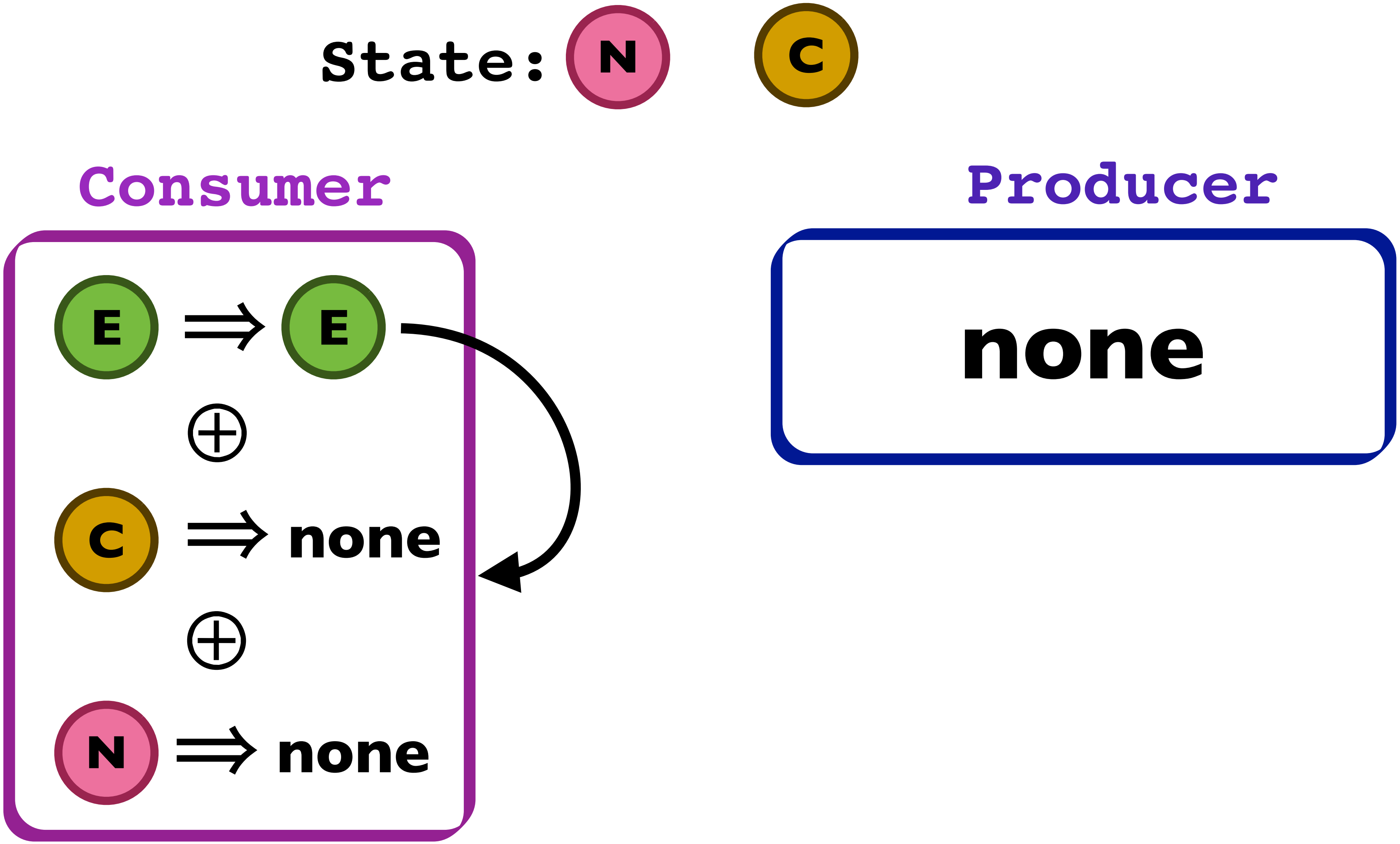


Configurations:

$\langle \mathsf{E} \;\Rrightarrow\; \mathbf{rec}\, X.(\,\mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\,;\, X\,) \parallel \mathsf{E} \Rightarrow (\,\mathsf{N} \oplus \mathsf{C}\,)\rangle$

$\langle \mathsf{N} \oplus \mathsf{C} \;\Rrightarrow\; \mathbf{rec}\, X.(\,\mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\,;\, X\,) \parallel \textbf{none}\rangle$

$\langle \textbf{none} \;\Rrightarrow\; \textbf{none} \parallel \textbf{none}\rangle$



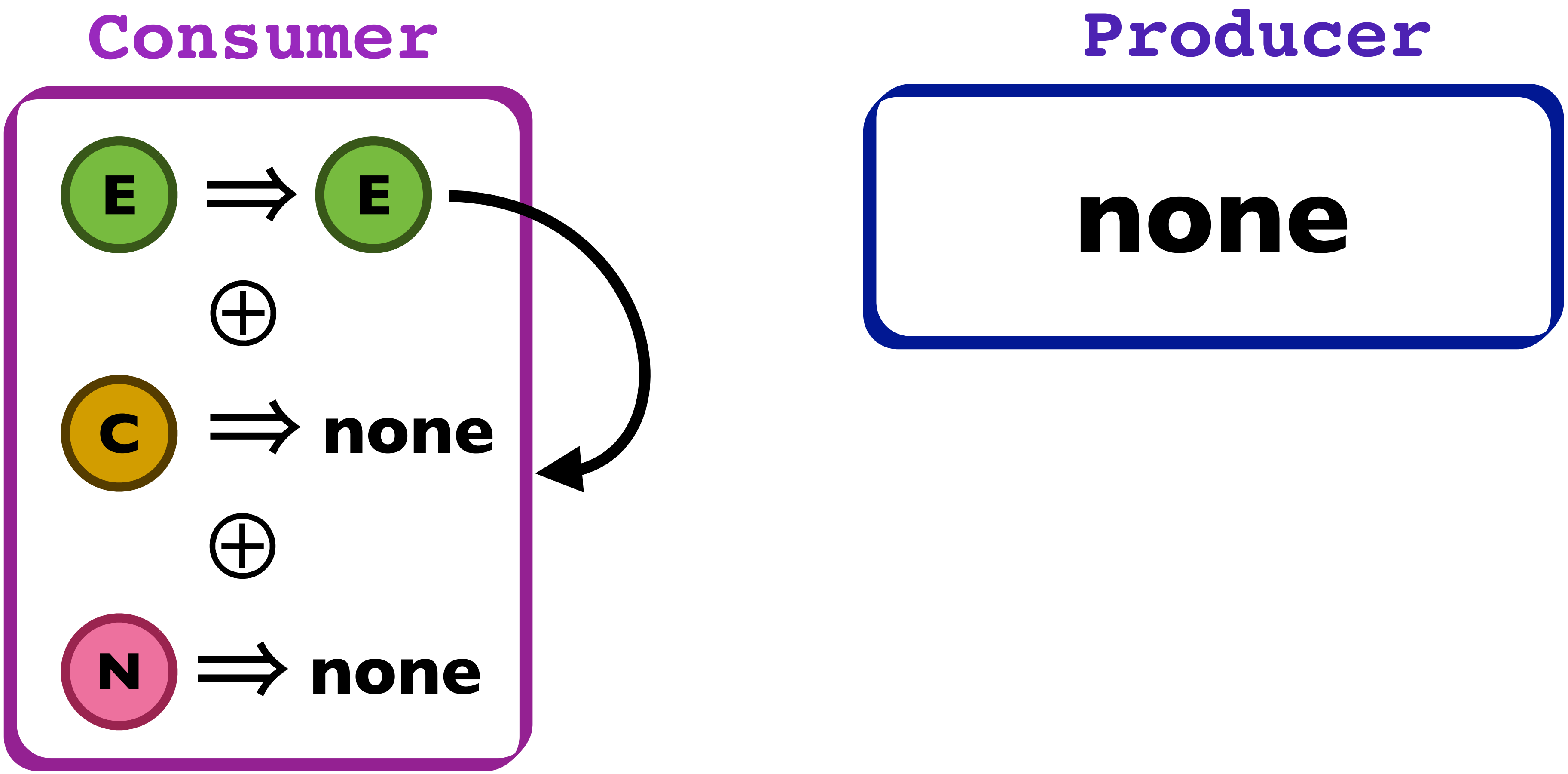
Configurations:

$\langle \mathsf{E} \;\Rightarrow\; \mathbf{rec}\, X.(\,\mathsf{N} \Rightarrow \mathbf{none} \oplus \mathsf{C} \Rightarrow \mathbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E} \,;\, X\,) \parallel \mathsf{E} \Rightarrow (\,\mathsf{N} \oplus \mathsf{C}\,)\rangle$

$\langle \mathsf{N} \oplus \mathsf{C} \;\Rightarrow\; \mathbf{rec}\, X.(\,\mathsf{N} \Rightarrow \mathbf{none} \oplus \mathsf{C} \Rightarrow \mathbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E} \,;\, X\,) \parallel \mathbf{none}\rangle$

$\langle \mathbf{none} \;\Rightarrow\; \mathbf{none} \parallel \mathbf{none}\rangle$

State: none



Configurations:

$\langle \mathsf{E} \;\Rrightarrow\; \mathbf{rec}\, X.(\,\mathsf{N} \Rightarrow \mathsf{none} \oplus \mathsf{C} \Rightarrow \mathsf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\,;\, X\,) \parallel \mathsf{E} \Rightarrow (\,\mathsf{N} \oplus \mathsf{C}\,)\rangle$

$\langle \mathsf{N} \oplus \mathsf{C} \;\Rrightarrow\; \mathbf{rec}\, X.(\,\mathsf{N} \Rightarrow \mathsf{none} \oplus \mathsf{C} \Rightarrow \mathsf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\,;\, X\,) \parallel \mathsf{none}\rangle$

$\langle \mathsf{none} \;\Rrightarrow\; \mathsf{none} \parallel \mathsf{none}\rangle$

State: **none**

Consumer

none

Producer

none

Configurations:

$\langle \mathsf{E} \Rrightarrow \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\,;\, X\,) \parallel \mathsf{E} \Rightarrow (\, \mathsf{N} \oplus \mathsf{C}\,)\rangle$

$\langle \mathsf{N} \oplus \mathsf{C} \Rrightarrow \mathbf{rec}\, X.(\, \mathsf{N} \Rightarrow \textbf{none} \oplus \mathsf{C} \Rightarrow \textbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\,;\, X\,) \parallel \textbf{none}\rangle$

$\langle \textbf{none} \Rrightarrow \textbf{none} \parallel \textbf{none}\rangle$

Related Work

Krishnaswami, Turon, Dreyer, Garg. **Superficially Substructural Types**. ICFP 2012.

Dinsdale-Young, Birkedal, Gardner, Parkinson, Yang. **Views: compositional reasoning for concurrent programs**. POPL 2013.

- Powerful generalization of *split* and *merge* operations (using commutative monoids) that enables expressive and precise descriptions of sharing.

Gordon, Ernst, Grossman. **Rely-Guarantee References for Refinement Types over Aliased Mutable Data**. PLDI 2013.

- References extended with predicate (for expressing local knowledge), rely and guarantee relations to handle sharing of state.

(Paper includes additional Related Work.)

Related Work

Krishnaswami, Turon, Dreyer, Garg. **Superficially Substructural Types**. ICFP 2012.

Dinsdale-Young, Birkedal, Gardner, Parkinson, Yang. **Views: compositional reasoning for concurrent programs**. POPL 2013.

- Po
  m

Gord
Typ

- Re
  gu

- We are limited to finite state representations, i.e. *typestates*.

\+ Protocols can express *changes over time* ("temporal sharing"), without requiring the use of auxiliary variables to distinguish steps.

\+ Sharing is a typing artifact and is not tied to a module.

\+ Can be type checked without manual intervention.

Summary

- **Contribution:** novel interference-control mechanism, *Rely-Guarantee Protocols*, to control sharing of state mutable by statically disconnected variables.
- **Topics Covered:** (more details in the paper)
    1. Protocol Specification ("public changes")
    2. Protocol Use ("private changes")
    3. Protocol Conformance ("alias interleaving")

  Experimental Prototype Implementation:

  http://**deaf-parrot**.googlecode.com