Rely-Guarantee Protocols
for Safe Interference over Shared Memory

Thesis Defense

Filipe Militão

December 15, 2015.

Co-advised by *Jonathan Aldrich* (CMU) and *Luís Caires* (UNL).

Software Defects

Our over reliance on software aggravates the impact of software defects (“*bugs*”) on society:

1962 - Mariner 1 Spacecraft (incorrect guidance signal transmission)
1985 - Therac-25 radiation therapy machine (race condition)
1991 - Patriot Missile Error (inaccurate tracking)
1996 - Ariane 5 Flight 501 (overflow in 16-bit register)
1998 - NASA’s Mars Climate Orbiter (imperial to metrics conversion)
2006 - Heathrow Terminal 5 Opening (baggage handling system)
2007 - Microsoft Excel 2007 (wrong result on 850*77.1)
2009 - Toyota vehicles (sudden unintended acceleration)

...

Thus, there is a clear and increasing need for mechanisms to improve software reliability.

Mutable State (I)

Mutable state can be useful in certain cases such as to improve efficiency, clarity, or even to enable greater extensibility.

We propose a new type-based technique to detect, at compile-time, a class of errors related to the unsafe use of *shared mutable state*.

Mutable State (II)

We aim to precisely detect and avoid *unsafe interference* in the use of mutable state that is shared through aliasing.

By precisely tracking the properties of mutable state we can avoid a class of state-related run-time errors and eliminates the need for some defensive run-time tests.

```
File file = new File( “fillet” );
file.write( “foo” );
file.close();
file.write( “bar” );
```

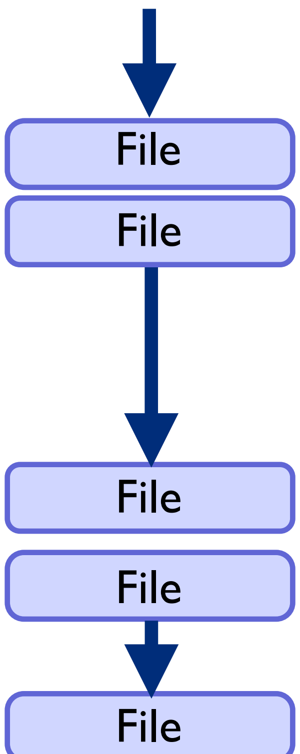
```
File file = new File( "fillet" );
file.write( "foo" );
file.close();
file.write( "bar" );
```

Error: *runtime* exception!

```
File file = new File( “fillet” );
file.write( “foo” );
file.close();
file.write( “bar” );
```

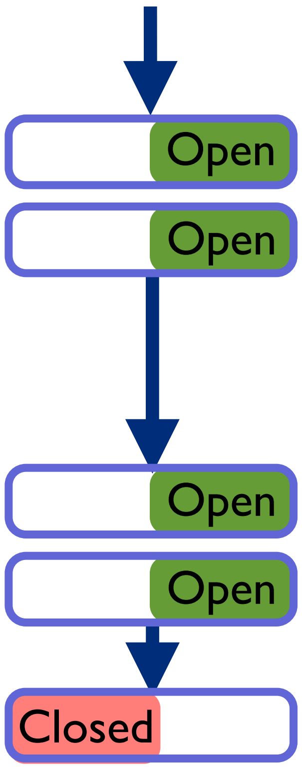
```
File file = new File( “fillet” );
file.write( “foo” );
file.close();
file.write( “bar” );
```

```
class File {
    FileDescriptor fd;
    File( string filename ){
     fd = OS.createFile( filename );
    }
    void write( string s ){
     if( fd == null )
      throw Exception(“invalid file descriptor”);
     fd.write( s );
    }
    void close(){
     fd = null;
    }
}
```




```
class File {
    FileDescriptor fd;
    File( string filename ){
     fd = OS.createFile( filename );
    }
    void write( string s ){
     if( fd == null )
      throw Exception("invalid file descriptor");
     fd.write( s );
    }
    void close(){
     fd = null;
    }
}
```

The "flat" abstraction does not *precisely* express the **changing properties of** `File`'s **internal state** (*`fd`*).



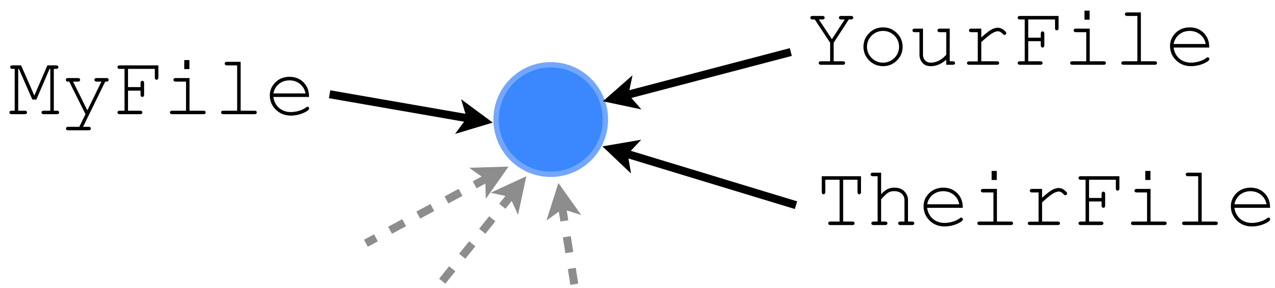

```
class File {
    FileDescriptor fd;
    File( string filename ){
     fd = OS.createFile( filename );
    }
    void write( string s ){
     if( fd == null )
      throw Exception("invalid file descriptor");
     fd.write( s );
    }
    void close(){
     fd = null;
    }
}
```

Unnecessary because we statically guarantee to only call `write` when Open.

Types capture the precise, changing, **type of the state**.

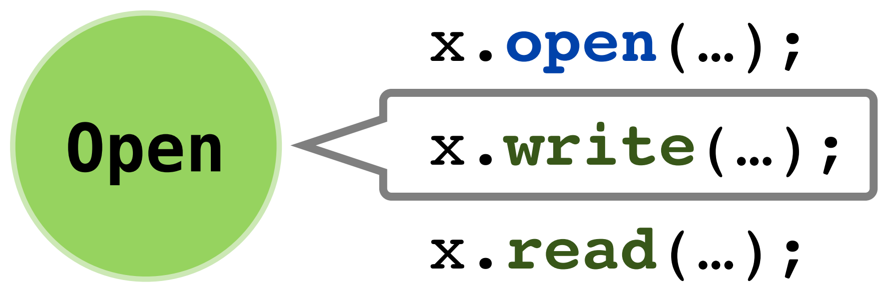
Challenge: Interference

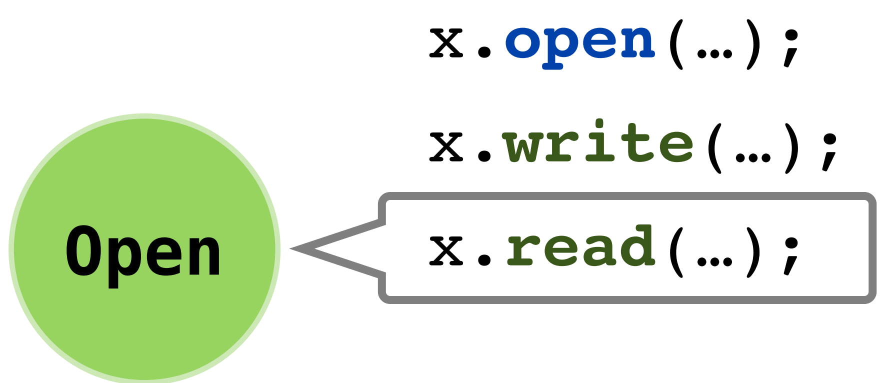
A*liasing* allows different names to refer to the same mutable state.

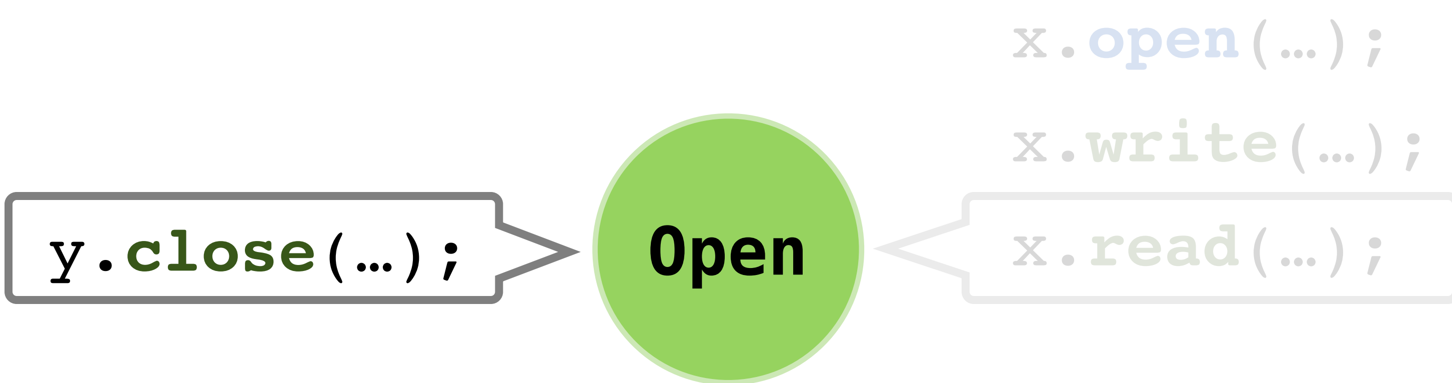


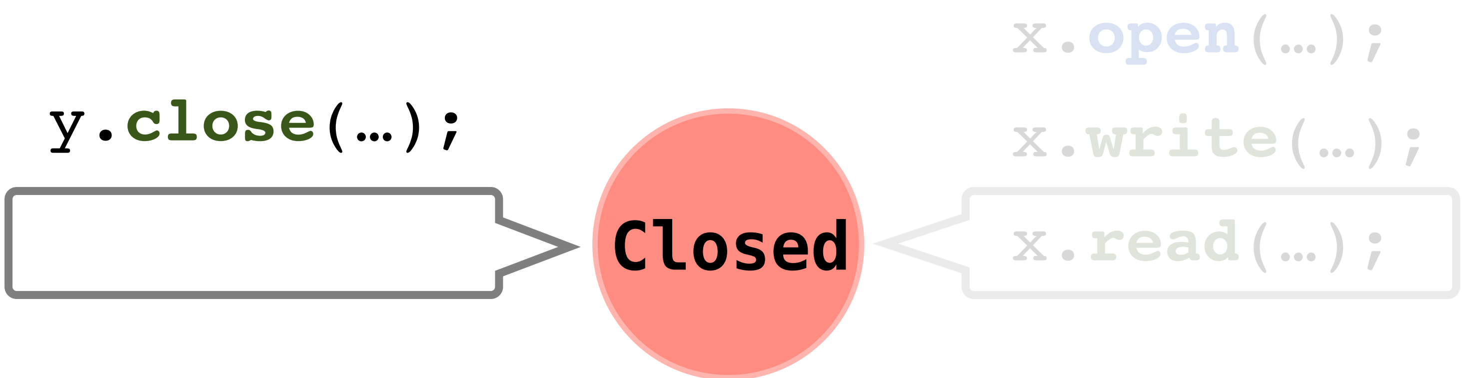
However, uncontrolled aliasing can lead to interference-related errors, where the actions of different aliases break each other's (precise) assumptions on the contents of the shared state.

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


Open
x.open(…);
x.write(…);
x.read(…);


x.open(…);
x.write(…);
Open
x.read(…);


y.close(…);
Open
x.open(…);
x.write(…);
x.read(…);


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

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

Closed

Broken local assumption: **interference**!

Thesis Statement

Rely-Guarantee Protocols provide a modular, composable, expressive, and automatically verifiable mechanism to control the interference resulting from the interaction of non-local aliases that share access to mutable state.

Overview

Aliasing Memory Locations ⇨ Typestate Abstraction ⇨ Sharing with *Rely-Guarantee Protocols*

Language

- Polymorphic λ-calculus with mutable references (and immutable records, tagged sums, ...).
- We use a variant of **L³** [*Ahmed, Fluet, and Morrisett.* **L³: A linear language with locations**. Fundam. Inform. 2007.] adapted for usability and extended with new constructs, and our sharing mechanism.

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

ref p **rw** p A

State as a Linear Resource

ref A

reference with contents of type A

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

duplicable **ref**erence to location p

State as a Linear Resource

ref *A*

reference with contents of type *A*

ref *p* **rw** *p* *A*

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

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

State as a Linear Resource

ref A

reference with contents of type A

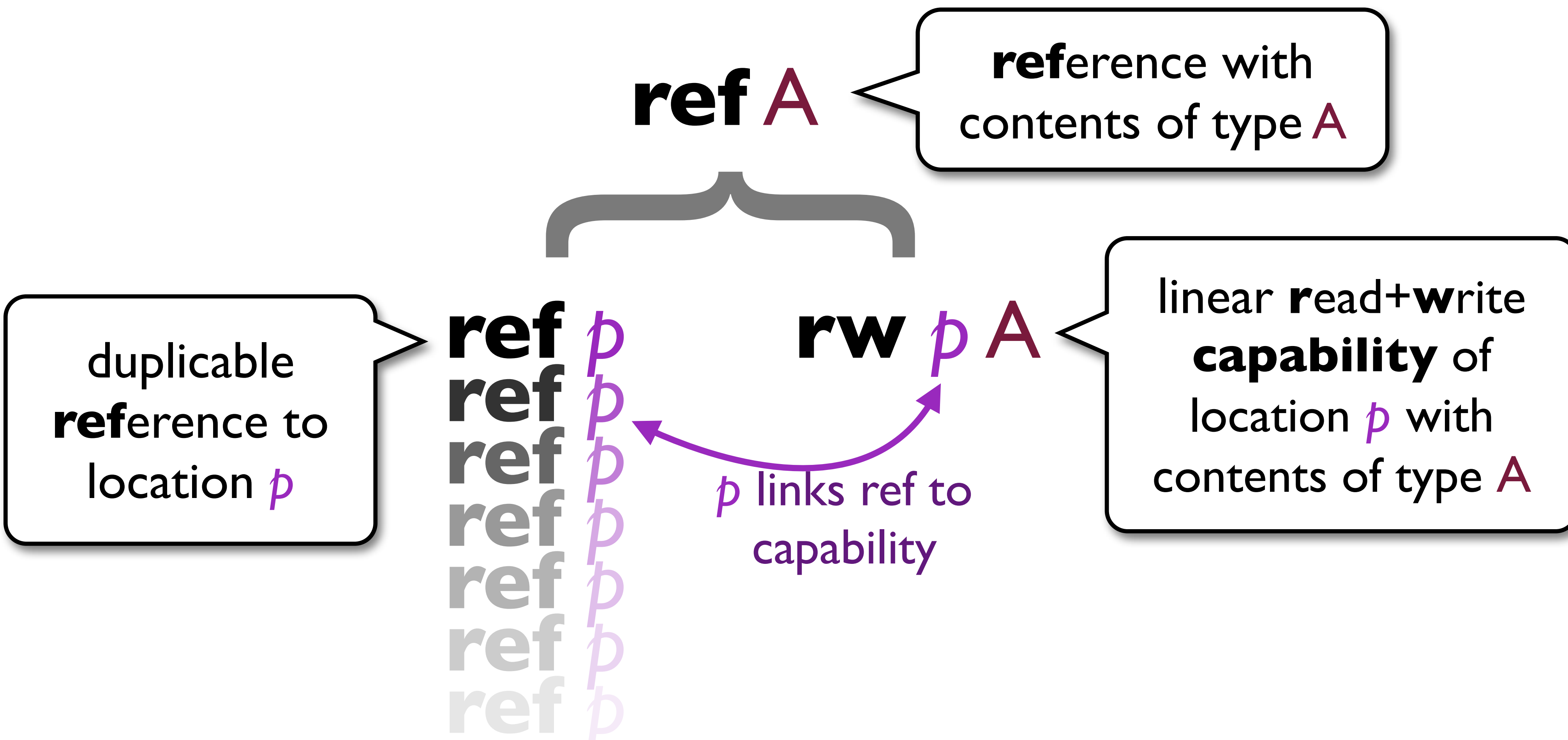
ref p
ref p
ref p
ref p
ref p
ref p
ref p

duplicable **ref**erence to location p

rw p A

linear **r**ead+**w**rite **capability** of location p with contents of type A

State as a Linear Resource



x : **ref** ***p*** **rw** ***p*** **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;
```

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

```
  x := false
end
```

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

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

Type Error: Missing capability to location *p*.

How to use capabilities in **functions**?

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

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

```
  x := false
end
```

Type Error: Missing capability to location *p*.

```
left:ref l
```

```
fun( i : int :: rw l [] ).
      left := i
```

```
fun( i : int :: rw l [] ).
left:ref l   i : int :: rw l []
left := i
```

```
fun( i : int :: rw l [] ).
left:ref l   i : int      rw l []
     left := i
```

```
fun( i : int :: rw l [] ).
        left := i
left:ref l    i : int        rw l int
```

```
fun( i : int :: rw l [] ).
        left := i
```

$$(\texttt{int :: rw l []}) \multimap (\texttt{[] :: rw l int})$$

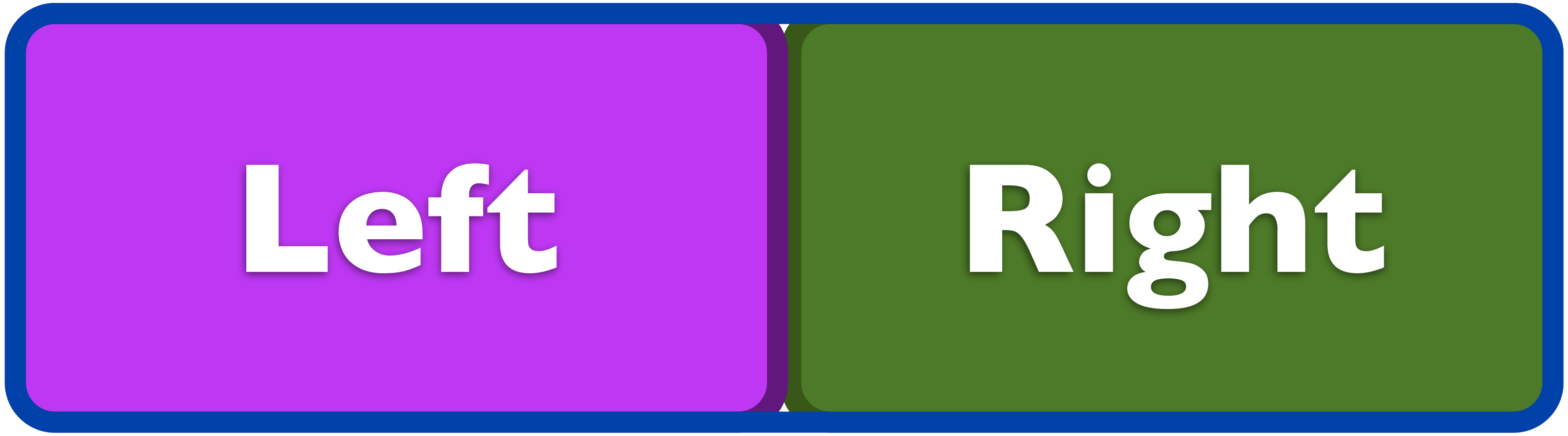
How to build **typestate abstractions**?

```
fun( i : int :: rw l [] ).
      left := i
```

```
(int :: rw l []) ⊸ ([] :: rw l int)
```

Pair Typestate

Pair Typestate



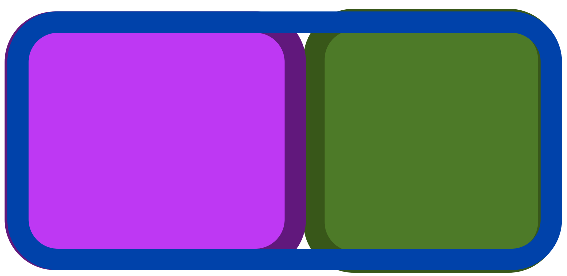
Pair Typestate

- ***Initialize*** each component separately.

- ***Sum*** both components, and ***destroy*** the pair.

Pair Typestate

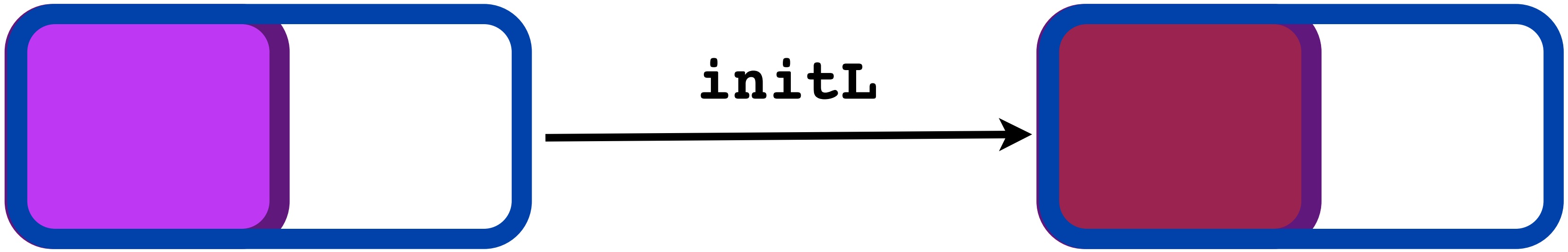
- ***Initialize*** each component separately.



- ***Sum*** both components, and ***destroy*** the pair.

Pair Typestate

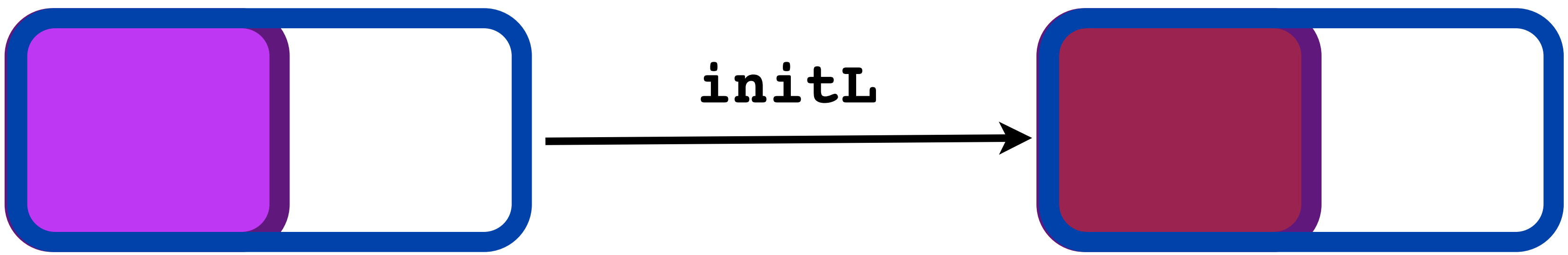
- ***Initialize*** each component separately.



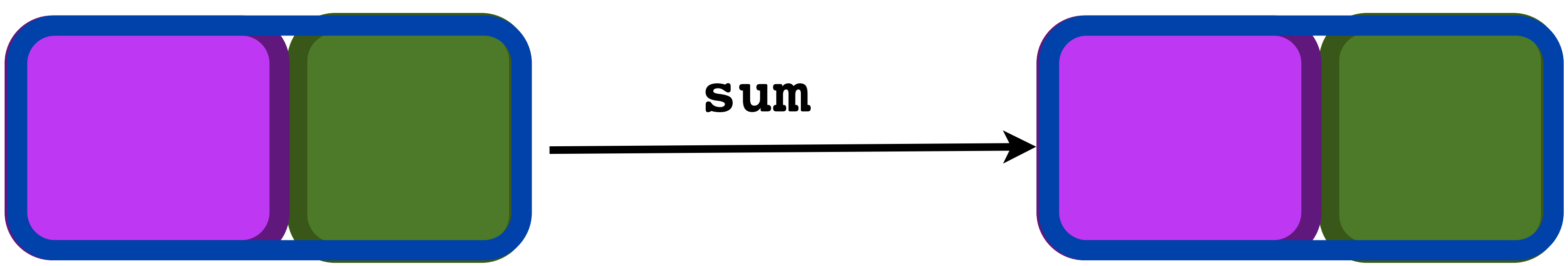
- ***Sum*** both components, and ***destroy*** the pair.

Pair Typestate

- ***Initialize*** each component separately.



- ***Sum*** both components, and ***destroy*** the pair.



```
initL : ( int :: rw 1 [] ) ⊸ ( [] :: rw 1 int )
```

```
[   initL : ( int :: rw l [] ) ⊸ ( [] :: rw l int ),
    initR : ( int :: rw r [] ) ⊸ ( [] :: rw r int ),
      sum : ( [] :: ( rw l int * rw r int ) ) ⊸
                 ( int :: ( rw l int * rw r int ) ),
  destroy : ( [] :: ( rw l int * rw r int ) ) ⊸ [] )
]
```

```
[  initL : ( int :: rw l [] ) ⊸ ( [] :: rw l int ),
   initR : ( int :: rw r [] ) ⊸ ( [] :: rw r int ),
     sum : ( [] :: ( rw l int * rw r int ) ) ⊸
                 ( int :: ( rw l int * rw r int ) ),
 destroy : ( [] :: ( rw l int * rw r int ) ) ⊸ [] )
] :: ( rw l [] * rw r [] )
```

```
[  initL : ( int :: EL ) ⊸ ( [] :: rw l int ),
   initR : ( int :: rw r [] ) ⊸ ( [] :: rw r int ),
     sum : ( [] :: ( rw l int * rw r int ) ) ⊸
                  ( int :: ( rw l int * rw r int ) ),
 destroy : ( [] :: ( rw l int * rw r int ) ) ⊸ [] )
] :: ( EL * rw r [] )
```

```
[   initL : ( int :: EL ) ⊸ ( [] :: L ),
    initR : ( int :: rw r [] ) ⊸ ( [] :: rw r int ),
      sum : ( [] :: ( L * rw r int ) ) ⊸
                  ( int :: ( L * rw r int ) ),
  destroy : ( [] :: ( L * rw r int ) ) ⊸ [] )
] :: ( EL * rw r [] )
```

```
[  initL : ( int :: EL ) ⊸ ( [] :: L ),
   initR : ( int :: ER ) ⊸ ( [] :: rw r int ),
     sum : ( [] :: ( L * rw r int ) ) ⊸
                    ( int :: ( L * rw r int ) ),
 destroy : ( [] :: ( L * rw r int ) ) ⊸ [] )
] :: ( EL * ER )
```

```
[  initL : ( int :: EL ) ⊸ ( [] :: L ),
   initR : ( int :: ER ) ⊸ ( [] :: R ),
     sum : ( [] :: ( L * R ) ) ⊸
               ( int :: ( L * R ) ),
 destroy : ( [] :: ( L * R ) ) ⊸ [] )
] :: ( EL * ER )
```

```
∃EL.∃L.∃ER.∃R.( [
      initL : !( int :: EL      ⊸   [] :: L ),
      initR : !( int :: ER      ⊸   [] :: R ),
        sum : !(  [] :: L * R   ⊸  int :: L * R ),
    destroy : !(  [] :: L * R   ⊸   [] )
  ] :: EL * ER )
```

Pair Typestate

```
newPair :
 !( [] ⊸  ∃EL.∃L.∃ER.∃R.( ![
          initL : !( int :: EL      ⊸   [] :: L ),
          initR : !( int :: ER      ⊸   [] :: R ),
            sum : !(  [] :: L * R   ⊸  int :: L * R ),
        destroy : !(  [] :: L * R   ⊸   [] )
     ] :: EL * ER ) )
```

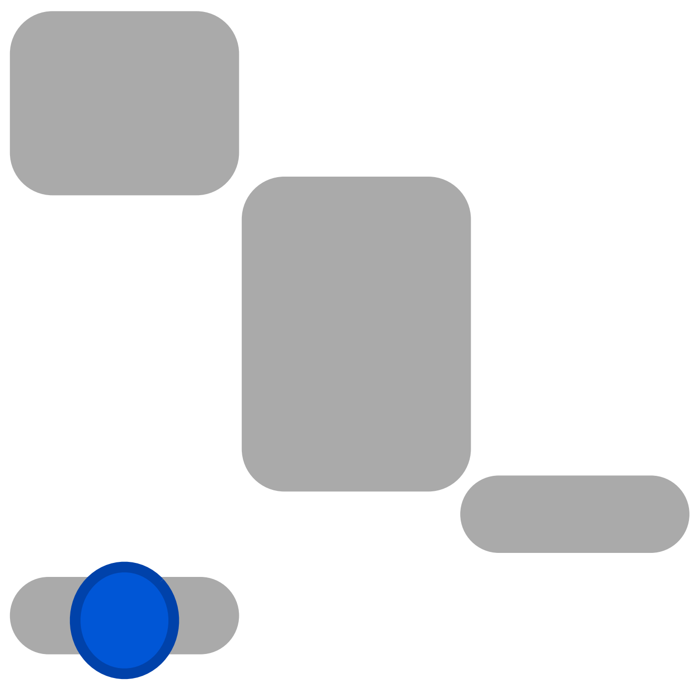
- Type expresses the changing properties of the object's state, **typestate** (*E*mpty*L*eft, *L*eft, *E*mpty*R*ight and *R*ight).
- The types states *EL*/*L* and *ER*/*R* correlate to separate (disjoint) internal state that can operate independently of the other.

So... Why sharing?

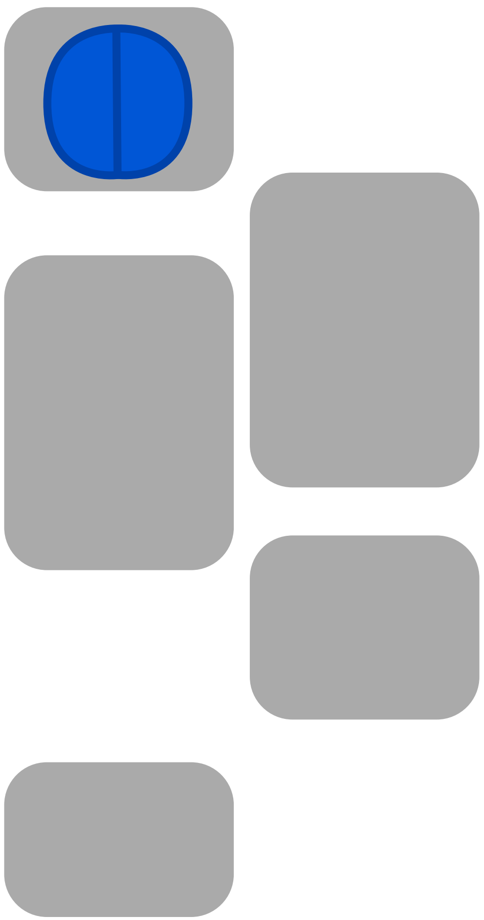
The capability is linear: it cannot be used *simultaneously* from two different parts of the program.

So... Why sharing?

The capability is linear: it cannot be used *simultaneously* from two different parts of the program.

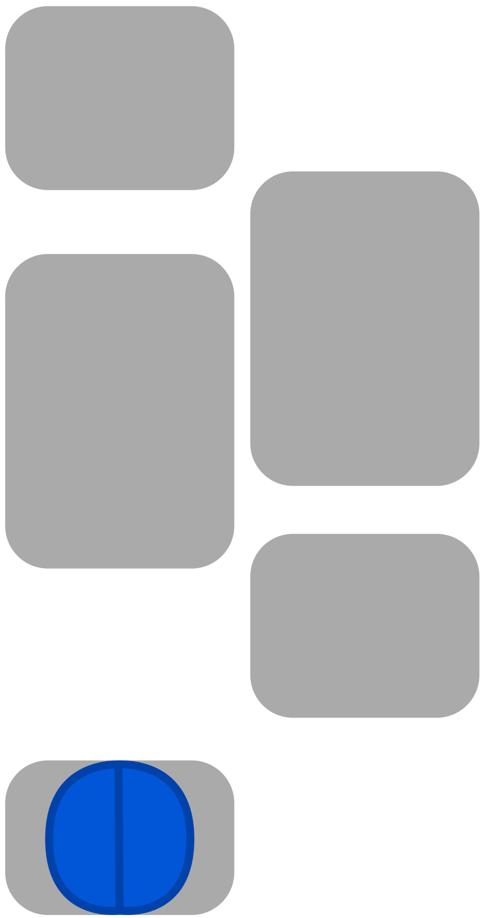


Sharing



- We want to use state *simultaneously*, beyond linearly.
- We need a typing mechanism to safely *coordinate* access to shared state and avoid unsafe interference.

Sharing

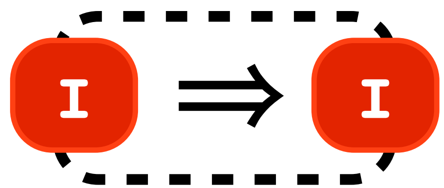


- We want to use state *simultaneously*, beyond linearly.

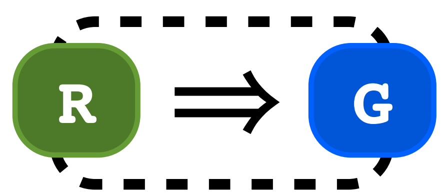
- We need a typing mechanism to safely *coordinate* access to shared state and avoid unsafe interference.

Sharing

- One solution is to have each alias preserve an initially held invariant, *invariant-based sharing*.

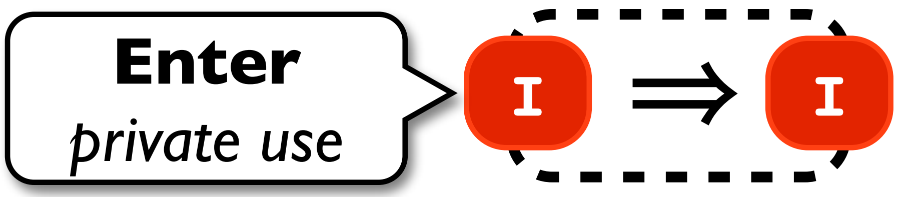


- Instead, we adapt the *spirit* of rely-guarantee reasoning, to enable more precise uses.

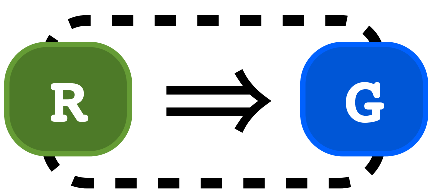


Sharing

- One solution is to have each alias preserve an initially held invariant, *invariant-based sharing*.

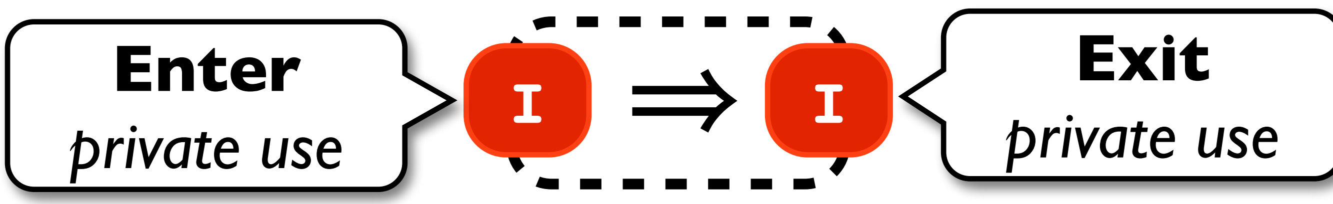


- Instead, we adapt the *spirit* of rely-guarantee reasoning, to enable more precise uses.

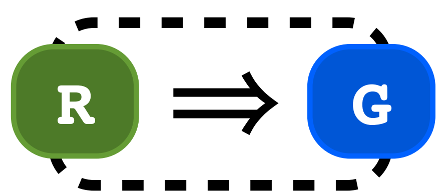


Sharing

- One solution is to have each alias preserve an initially held invariant, *invariant-based sharing*.

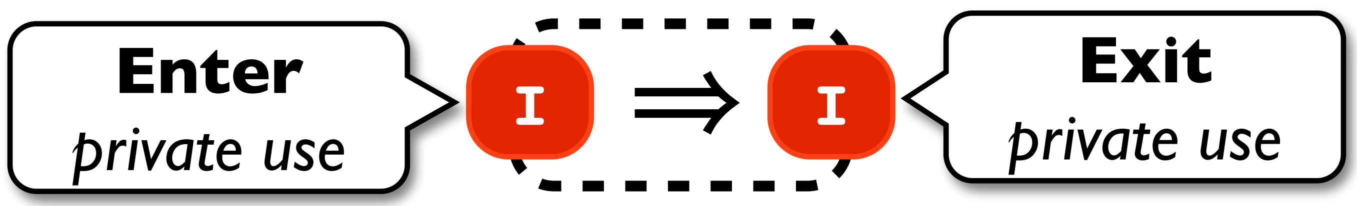


- Instead, we adapt the *spirit* of rely-guarantee reasoning, to enable more precise uses.



Sharing

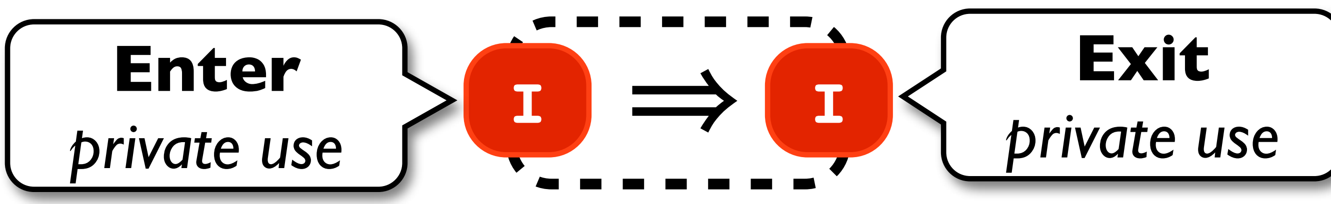
- One solution is to have each alias preserve an initially held invariant, *invariant-based sharing*.



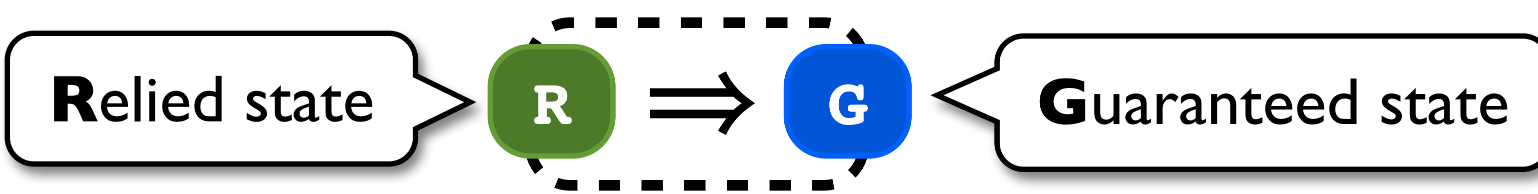
- Instead, we adapt the *spirit* of rely-guarantee reasoning, to enable more precise uses.

Sharing

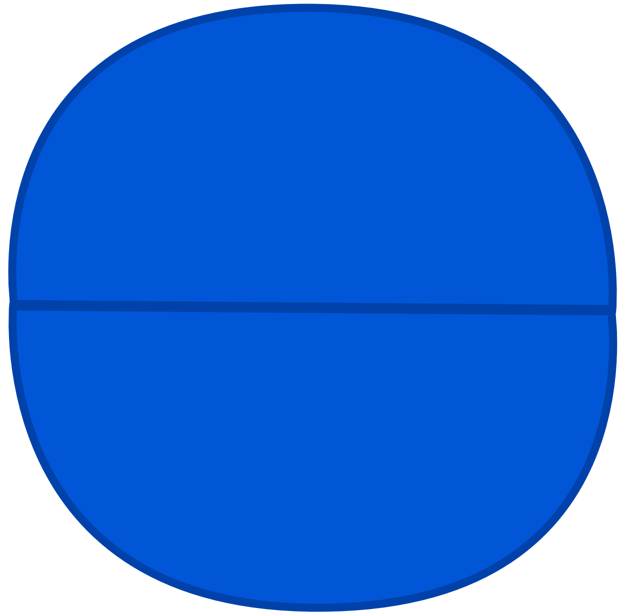
- One solution is to have each alias preserve an initially held invariant, *invariant-based sharing*.



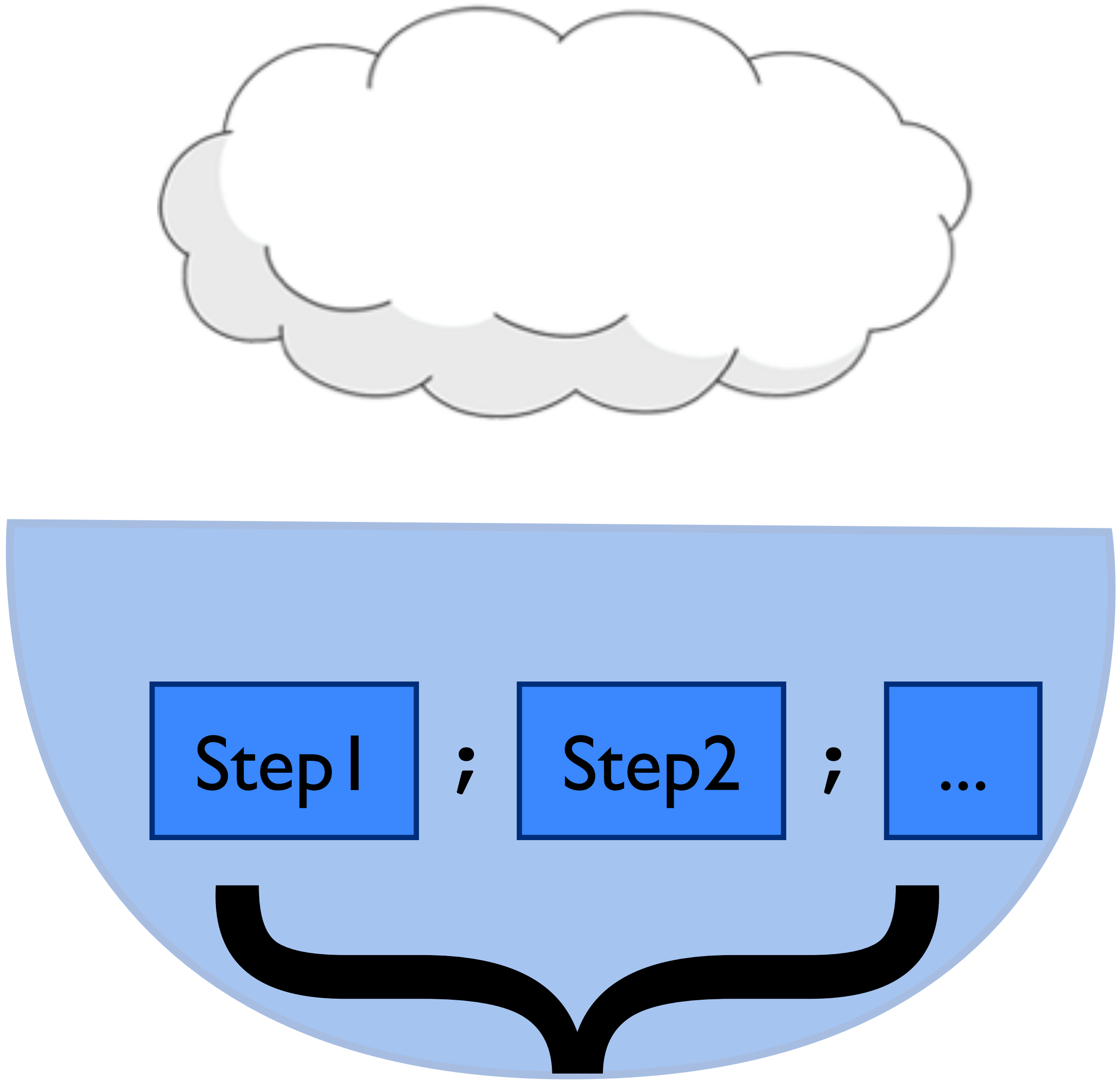
- Instead, we adapt the *spirit* of rely-guarantee reasoning, to enable more precise uses.



Shared cell

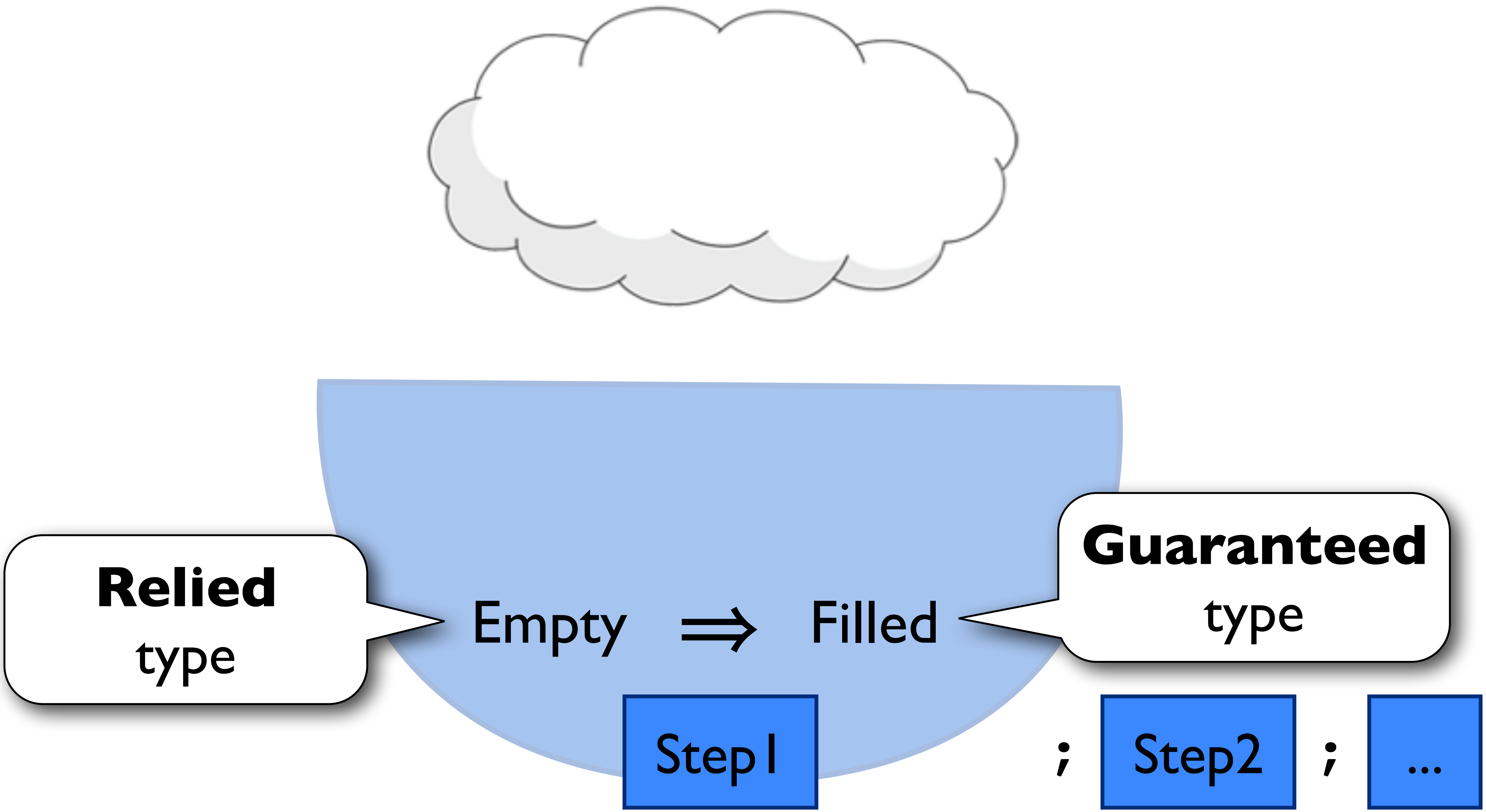


Alias' Local View

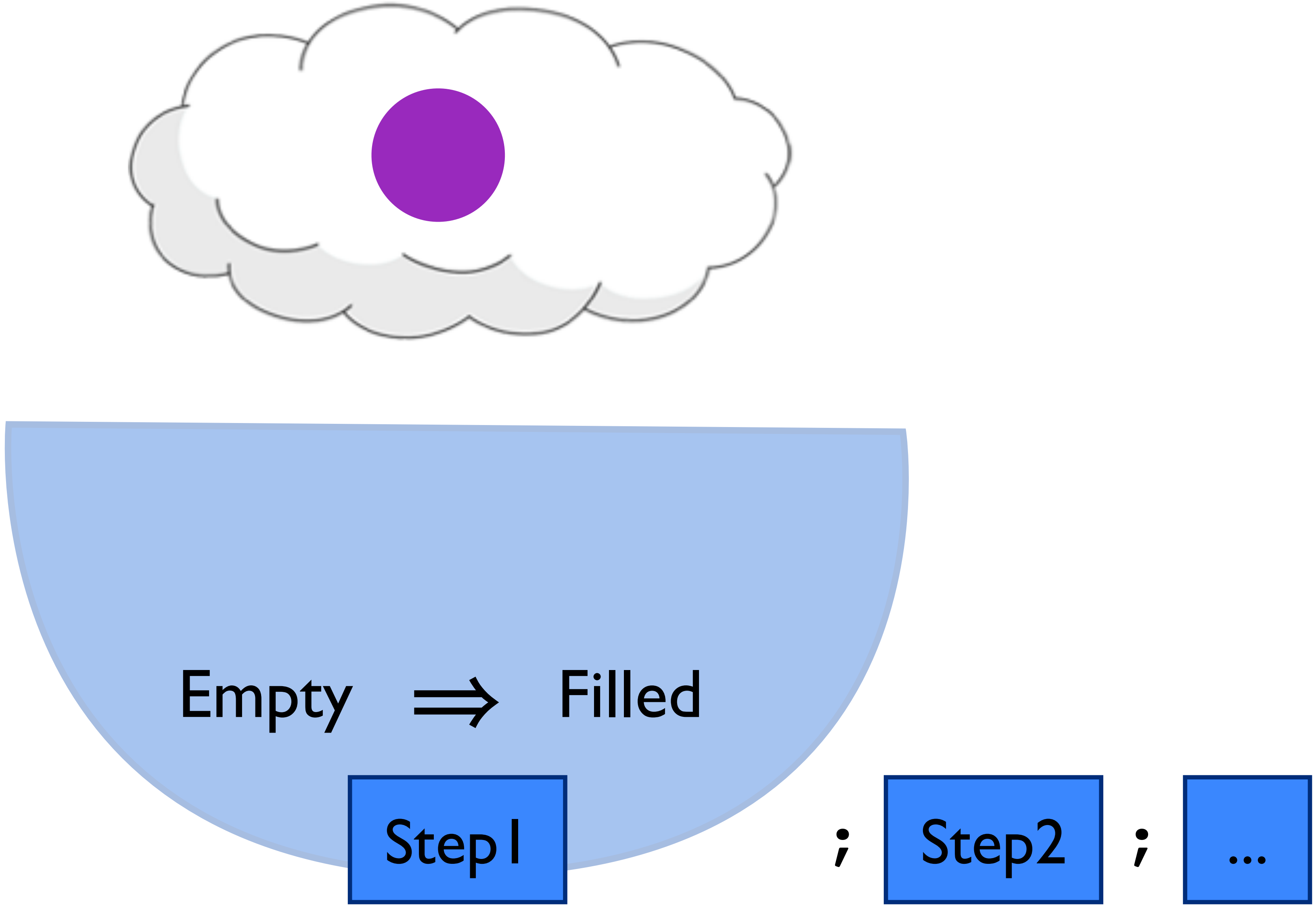


rely-guarantee protocol

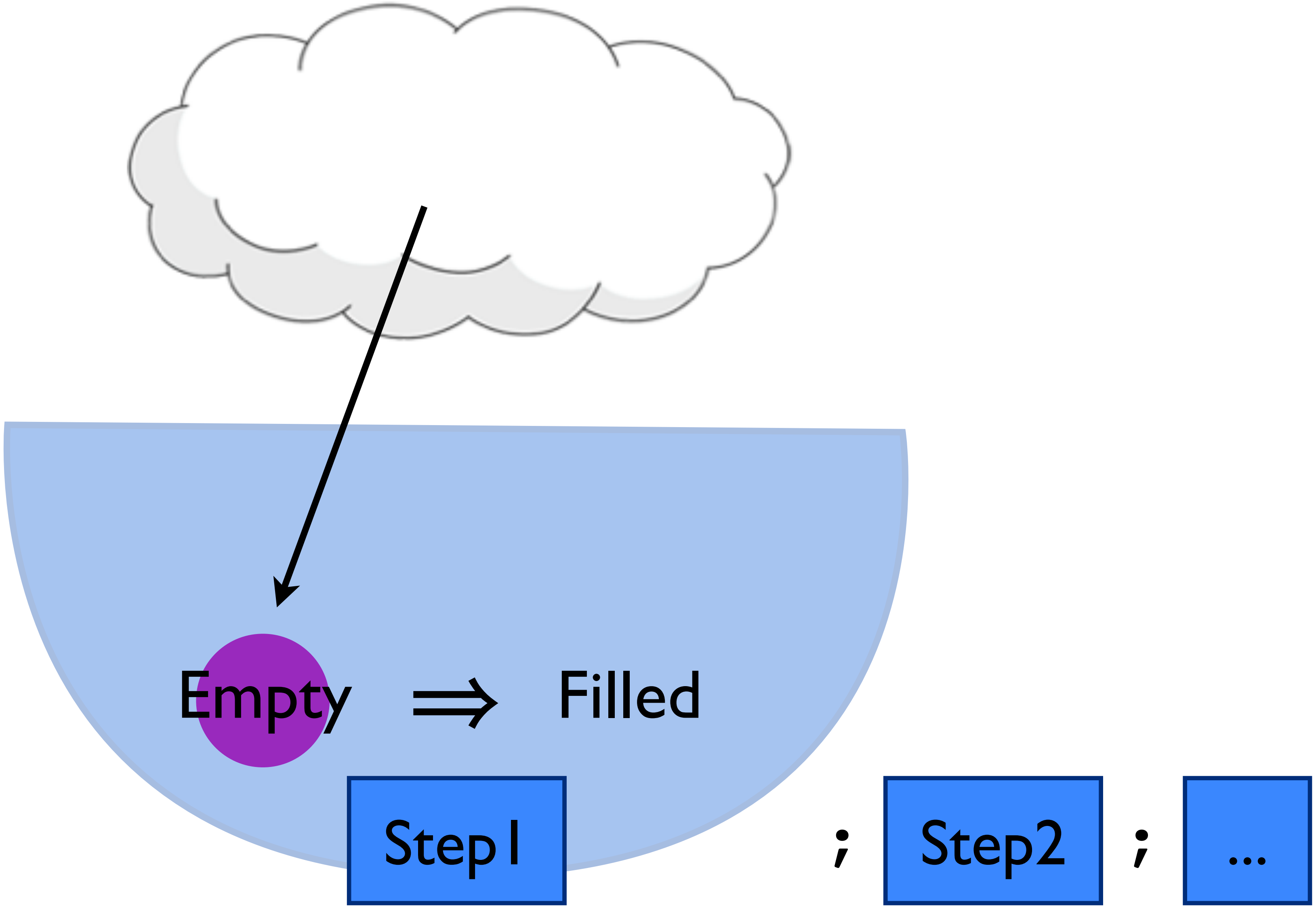
Alias' Local View



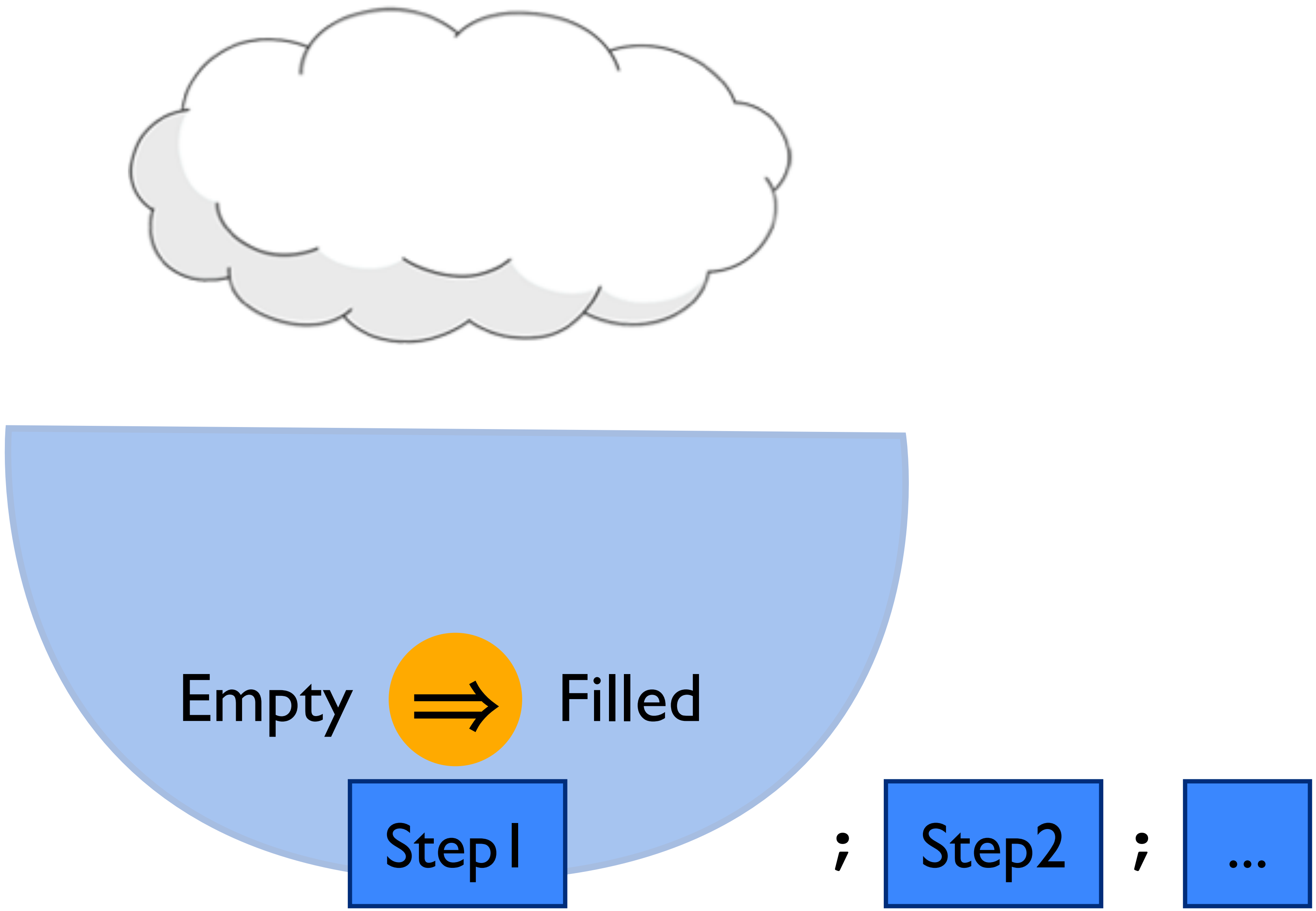
Alias' Local View



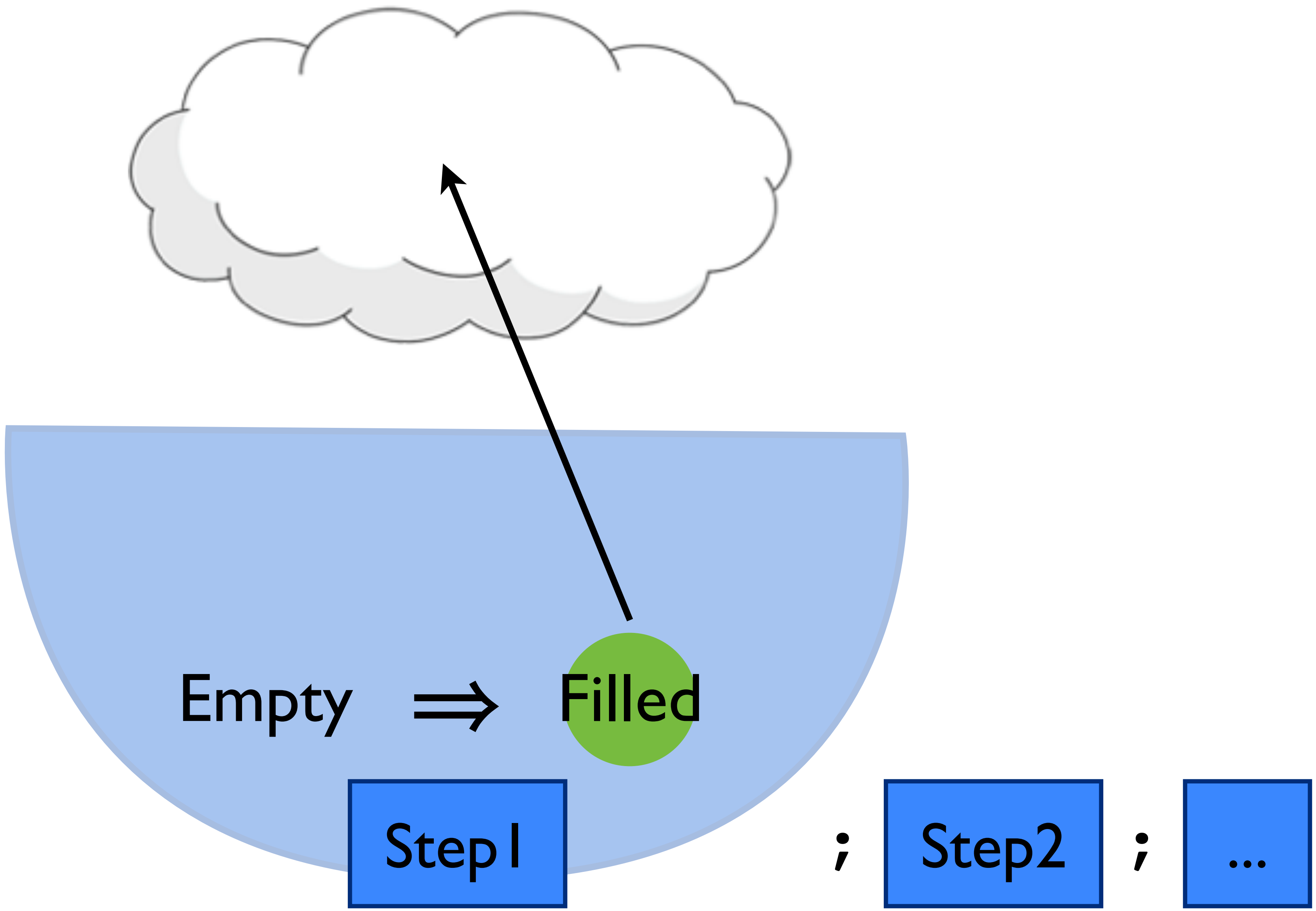
Alias' Local View



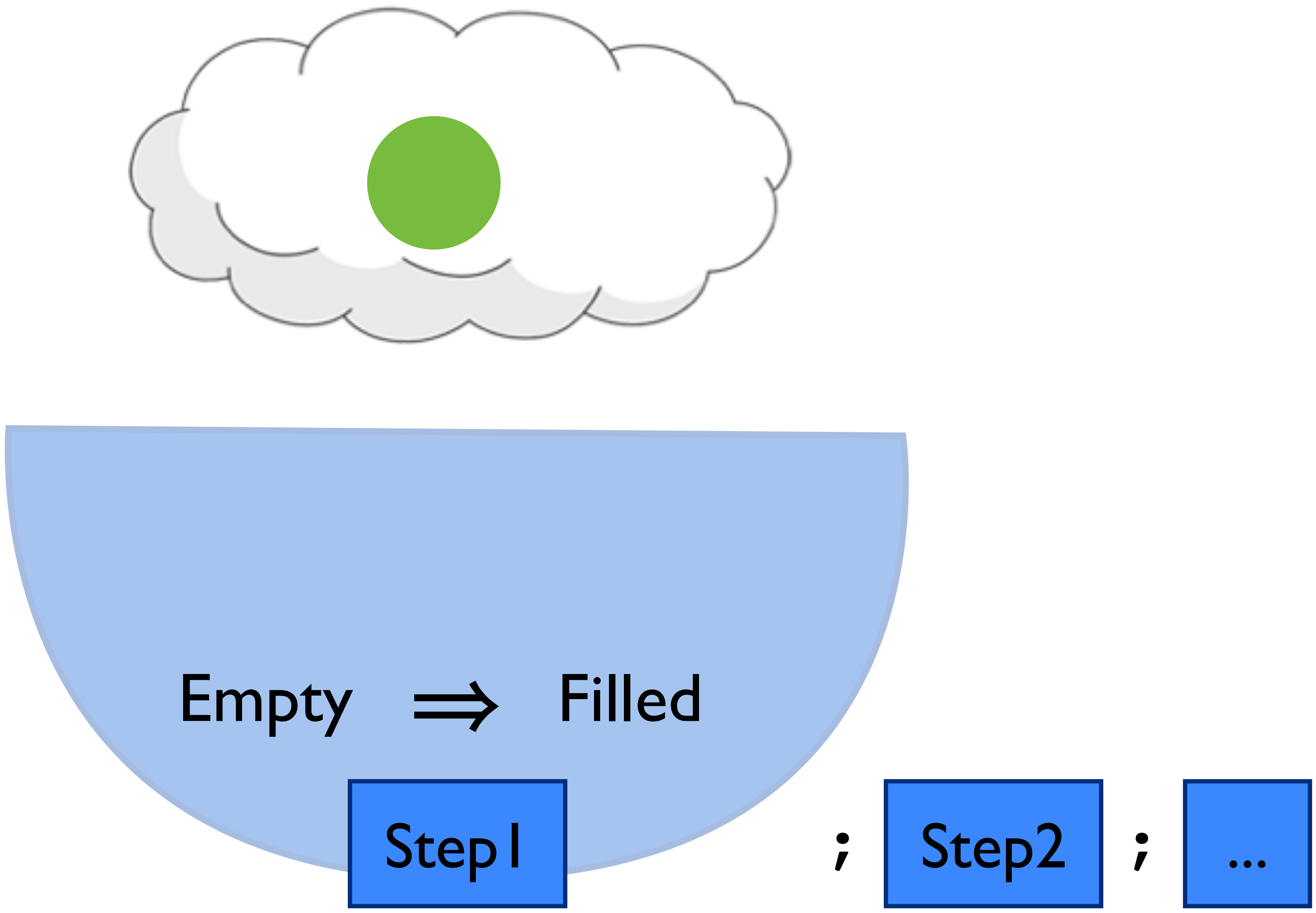
Alias' Local View



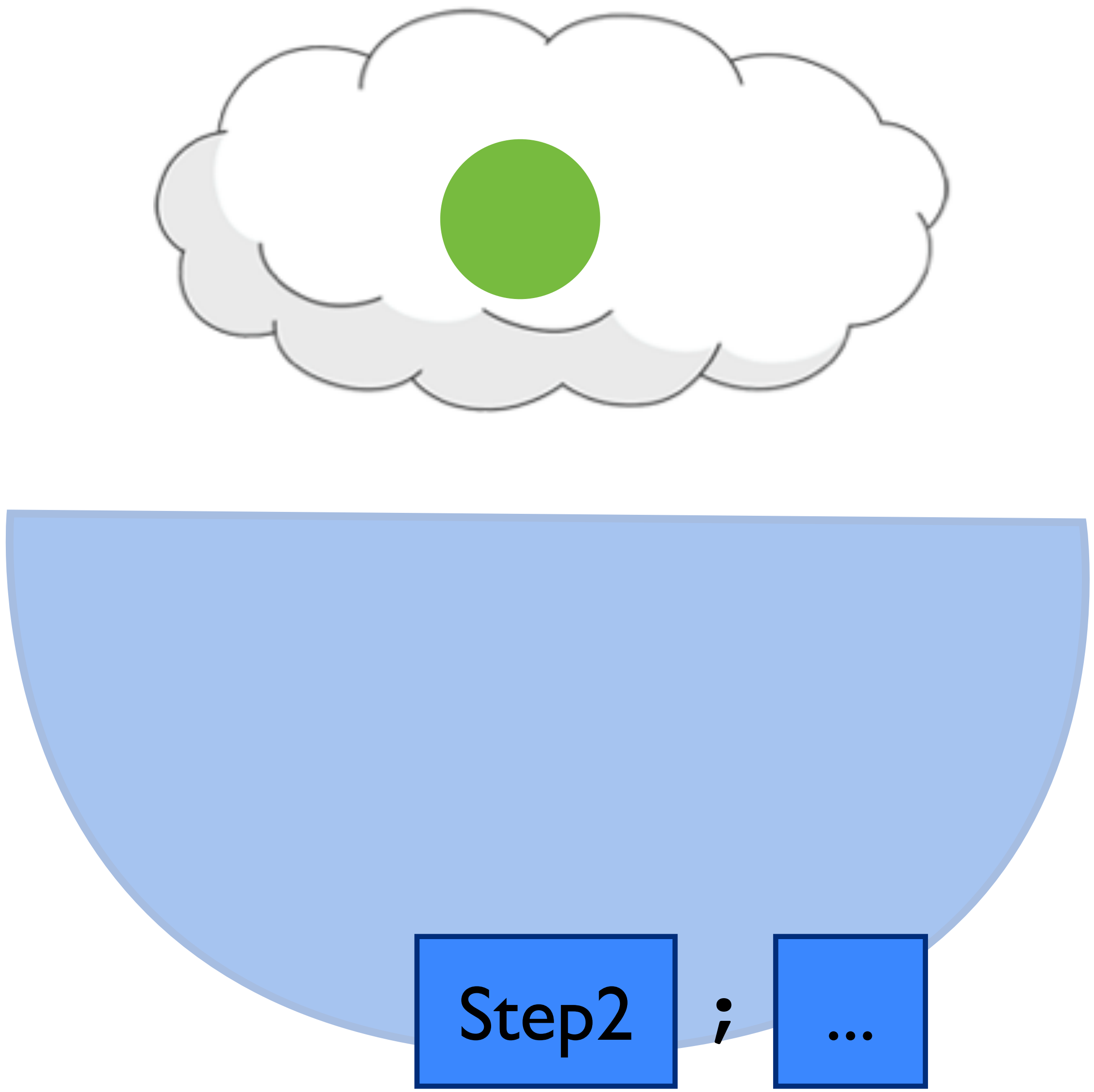
Alias' Local View



Alias' Local View

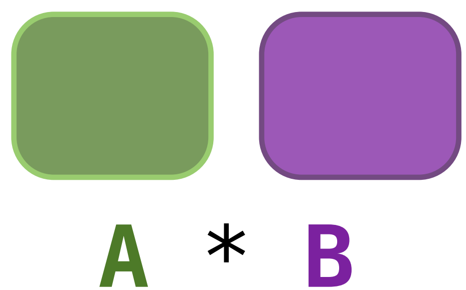


Alias' Local View



Disjoint

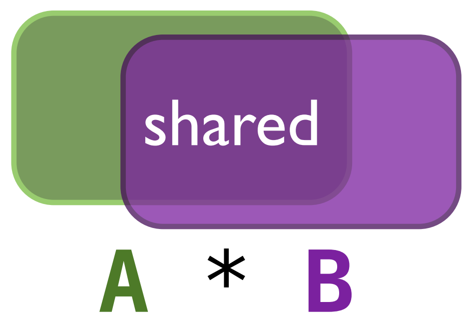
- Linearity ensured the state was disjoint:
  only one capability to some cell may exist.



- Sharing enables the same cell to be used through seemingly different name.
  Types that "look" disjoint, may in fact alias the same state - i.e. they are *fictionally* disjoint.

Fictionally Disjoint

- Linearity ensured the state was disjoint:
  only one capability to some cell may exist.



- Sharing enables the same cell to be used through seemingly different name.
  Types that "look" disjoint, may in fact alias the same state - i.e. they are *fictionally* disjoint.

Alias Interleaving

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

Alias Interleaving

```
doSomething :
   [] ⊸ []
```

```
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`, although fictionally disjoint, may actually interleave *zero or more* uses of aliases to the same state as referenced by `x`.

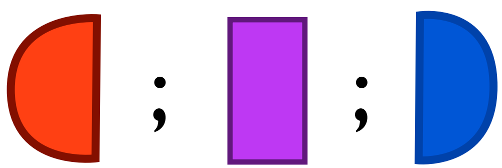
Alias Interleaving

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

Are the uses done in `doSomething` compatible with our local reasoning of how `x` changes?

Protocol Composition

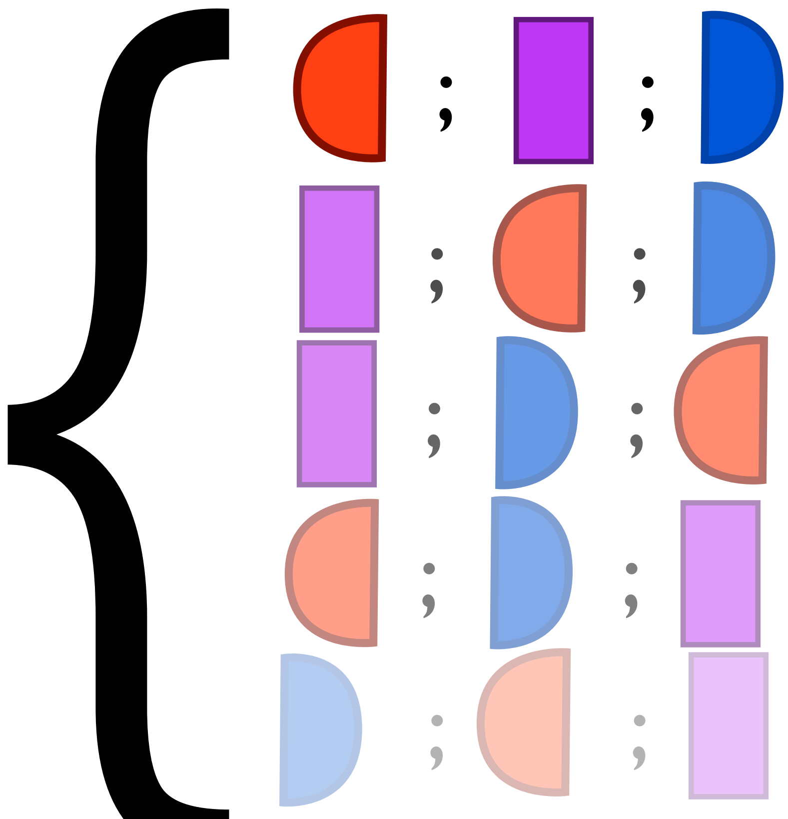
Checks if combining all local views creates a globally consistent, i.e. safe, use of the shared state mutable state independently of interleaving.



Protocol Composition

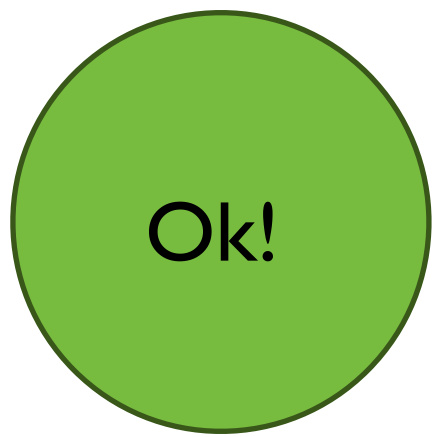
Checks if combining all local views creates a globally consistent, i.e. safe, use of the shared state mutable state independently of interleaving.

possible interleaving



Protocol Composition

Checks if combining all local views creates a globally consistent, i.e. safe, use of the shared state mutable state independently of interleaving.



Rely-Guarantee Protocols

- An *interference*-control mechanism, permission to mutate the shared state is conditioned on what actions the protocol allows.
- I will focus on presenting the following:
    1. Protocol Specification
    2. Protocol Use
    3. Protocol Composition

Rely-Guarantee Protocols

- An *interference*-control mechanism, permission to mutate the shared state is conditioned on what actions the protocol allows.
- I will focus on presenting the following:

1. Protocol Specification
2. Protocol Use
3. Protocol Composition

Protocol Specification

$$\begin{aligned} P \;::=\;& (\mathbf{rec}\, X(\overline{u}).P)[\overline{U_P}] \;\mid\; X[\overline{U_P}] \;\mid\; P \oplus P \\ &\mid\; P \mathbin{\&} P \;\mid\; R \Rightarrow P \;\mid\; R\,;\,P \;\mid\; \textsf{none} \end{aligned}$$

Protocol Specification

$$
\begin{aligned}
P \ ::= \ & \boxed{(\mathbf{rec}\ X(\overline{u}).P)[\overline{U_P}] \ \mid \ X[\overline{U_P}]} \mid P \oplus P \\
& \mid \ P \,\&\, P \ \mid \ R \Rightarrow P \ \mid \ R\,;\,P \ \mid \ \textsf{none}
\end{aligned}
$$

Protocol Specification

$$P \;::=\; (\mathbf{rec}\, X(\overline{u}).P)[\overline{U_P}] \;\mid\; X[\overline{U_P}] \;\mid\; \boxed{P \oplus P}$$
$$\mid\; P \,\&\, P \;\mid\; R \Rightarrow P \;\mid\; R\,;\,P \;\mid\; \textsf{none}$$

Protocol Specification

$$
\begin{aligned}
P \ ::= \ & (\mathbf{rec}\, X(\overline{u}).P)[\overline{U_P}] \ \mid \ X[\overline{U_P}] \ \mid \ P \oplus P \\
& \mid \ \boxed{P \,\&\, P} \ \mid \ R \Rightarrow P \ \mid \ R\,;\, P \ \mid \ \textsf{none}
\end{aligned}
$$

Protocol Specification

$$
\begin{aligned}
P \ ::= \ & (\mathbf{rec}\ X(\overline{u}).P)[\overline{U_P}] \mid X[\overline{U_P}] \mid P \oplus P \\
& \mid P \,\&\, P \mid \boxed{R \Rightarrow P} \mid R\,;\,P \mid \textsf{none}
\end{aligned}
$$

States that it is safe to assume that shared state satisfies **R**, and requires the alias to obey the guarantee **P**.

Protocol Specification

$$P ::= (\mathbf{rec}\ X(\overline{u}).P)[\overline{U_P}] \mid X[\overline{U_P}] \mid P \oplus P$$
$$\mid P \,\&\, P \mid R \Rightarrow P \mid \boxed{R\,;\,P} \mid \textbf{none}$$

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

Protocol Specification

$$P \;::=\; (\mathbf{rec}\, X(\overline{u}).P)[\overline{U_P}] \;\mid\; X[\overline{U_P}] \;\mid\; P \oplus P$$
$$\mid\; P \,\&\, P \;\mid\; R \Rightarrow P \;\mid\; R\,;\,P \;\mid\; \boxed{\textsf{none}}$$

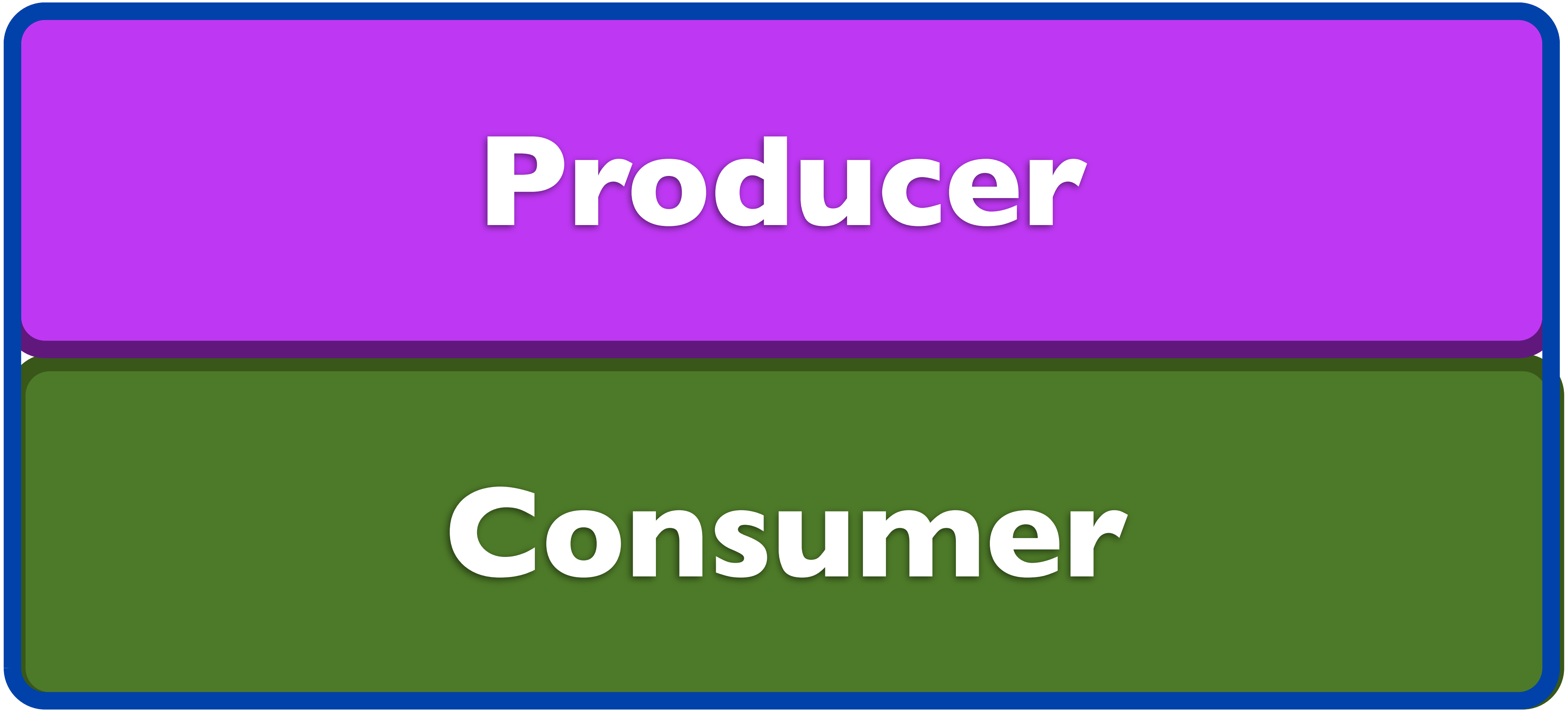
Shared Pipe

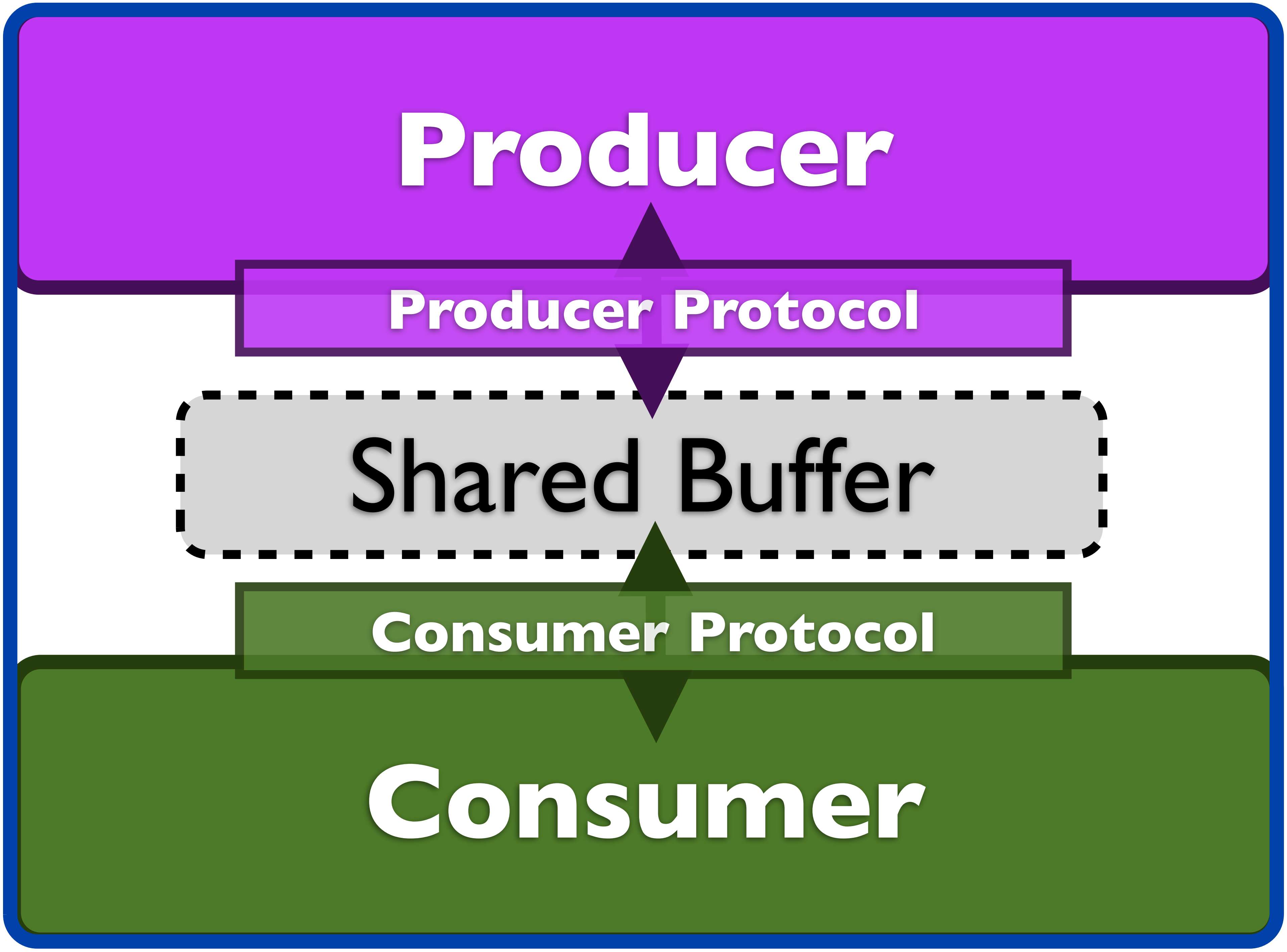
Shared by two aliases interacting via a common buffer, here modeled as a *singly linked list*.

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

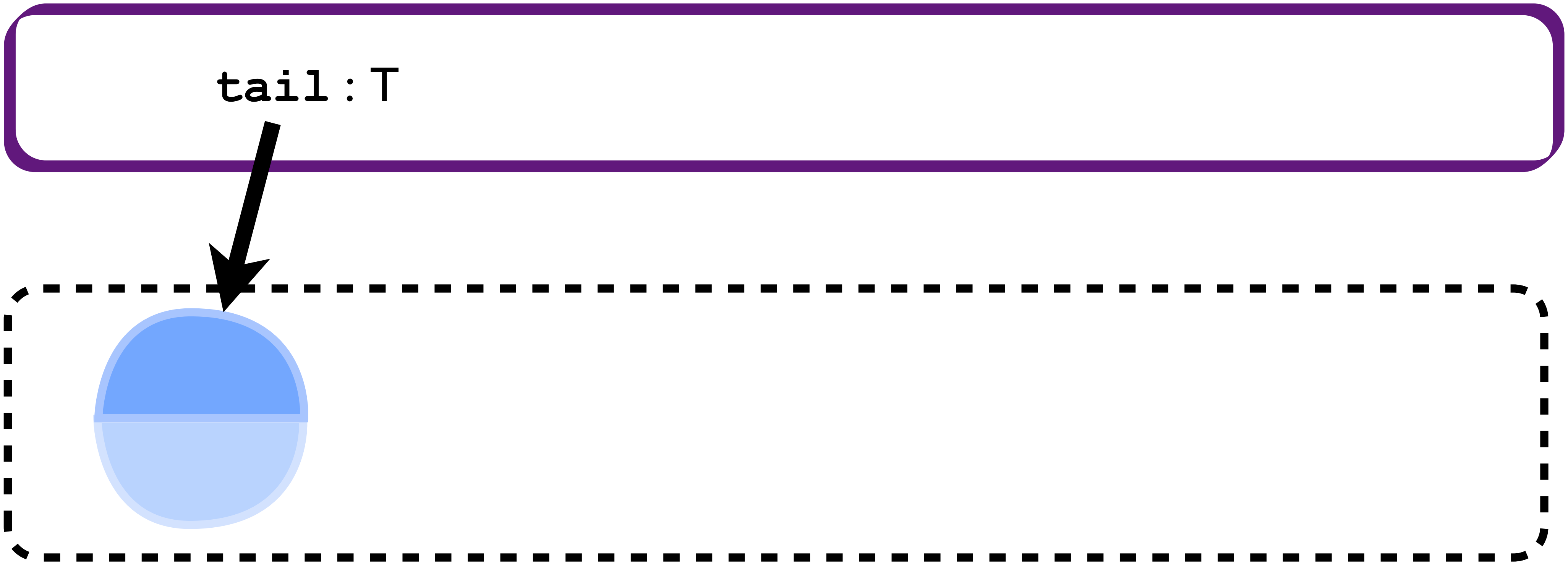
   The result of **`tryTake`** is one of the following: either there was some **`Result`**, or **`NoResult`**, or the pipe is fully **`Depleted`**.

Pipe


Producer
Consumer


Producer
Producer Protocol
Shared Buffer
Consumer Protocol
Consumer

Producer

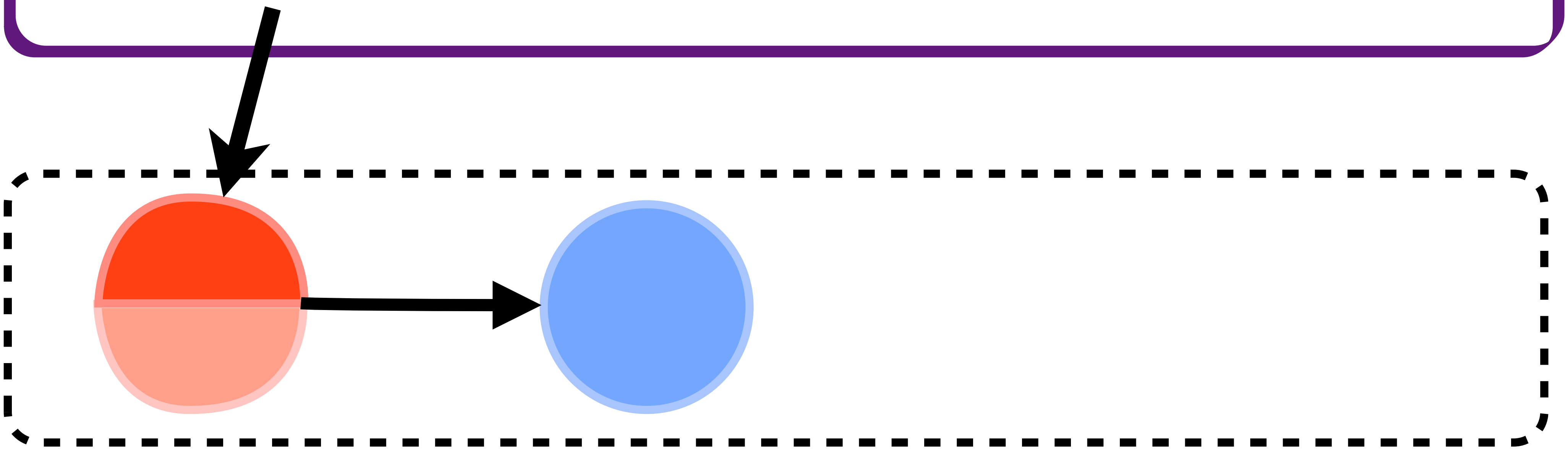


Producer

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

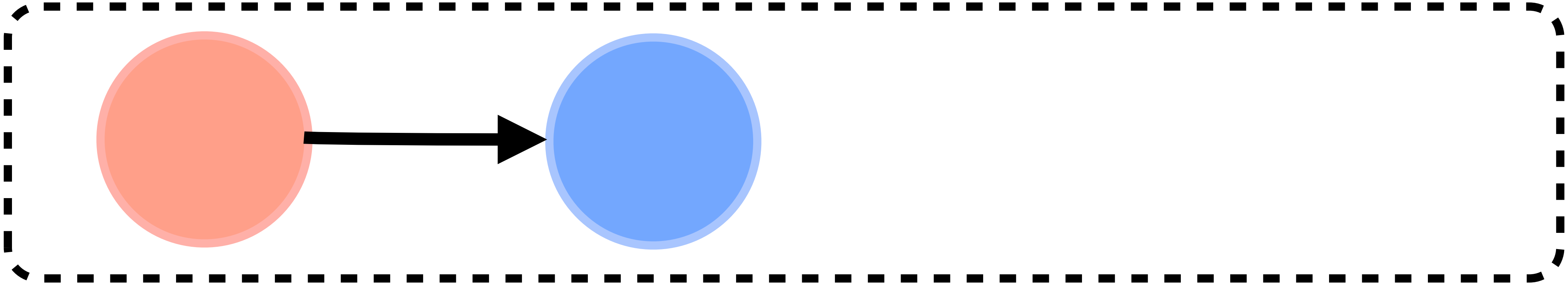
Producer

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



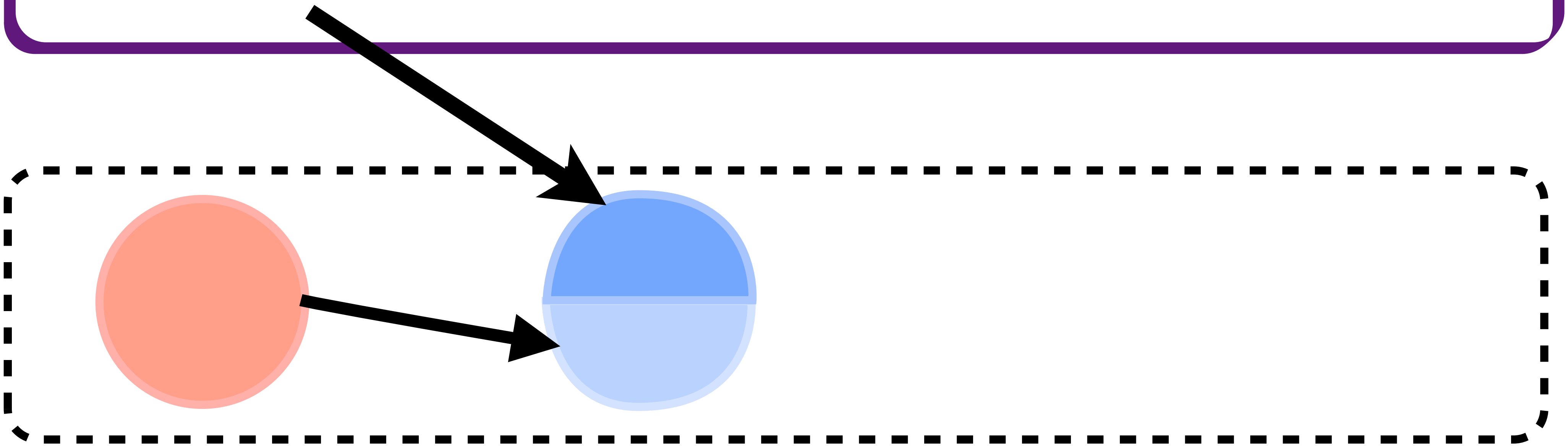
Producer

`tail` : **none**



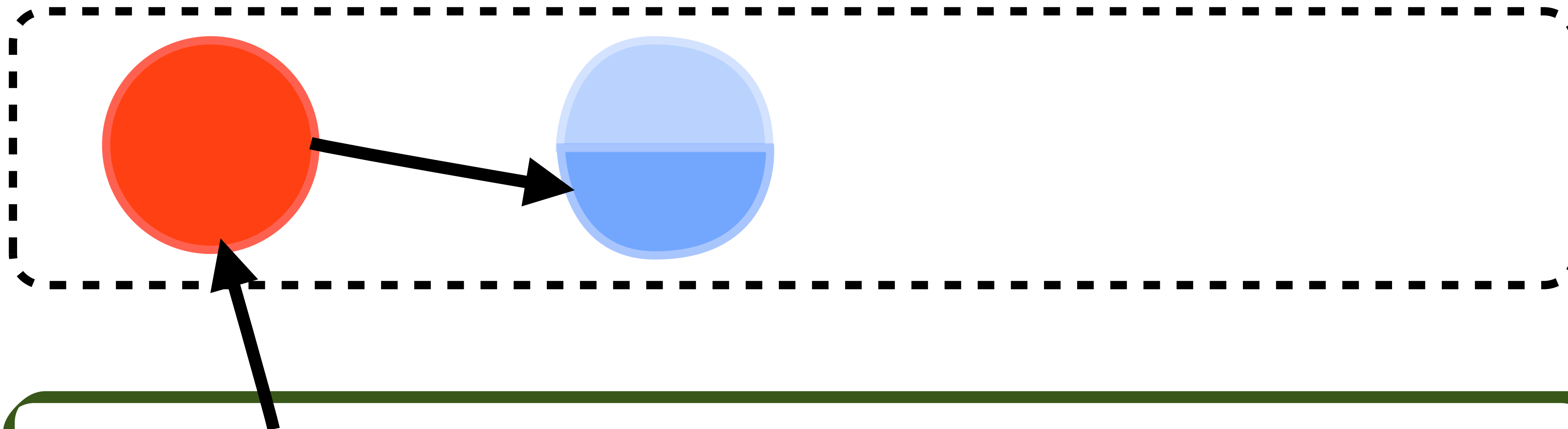
Producer

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



Producer

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

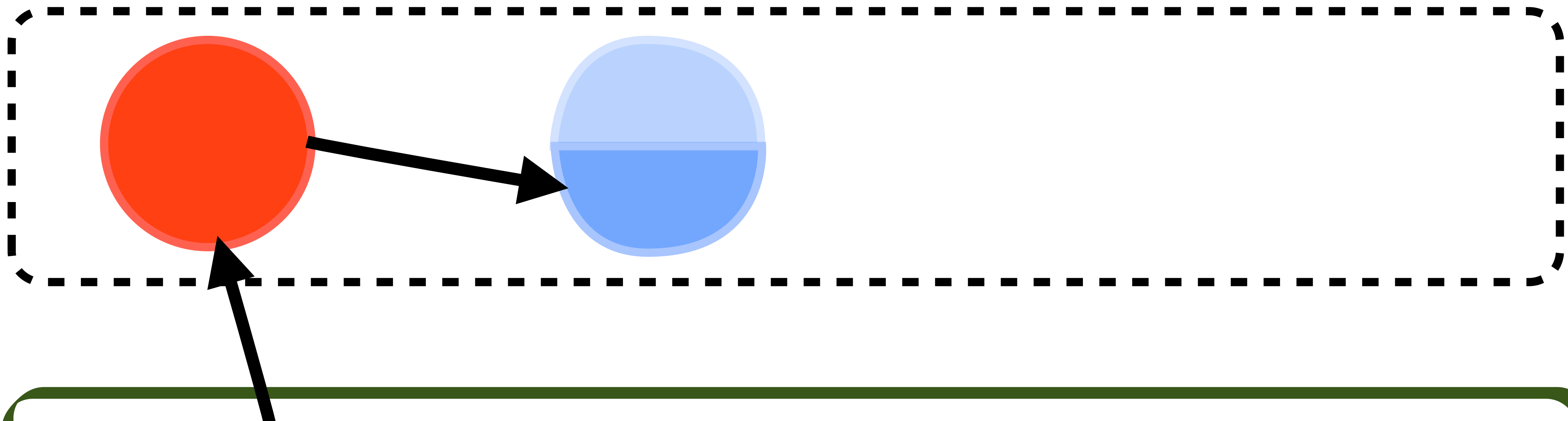


`head` : H

Consumer

Producer

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



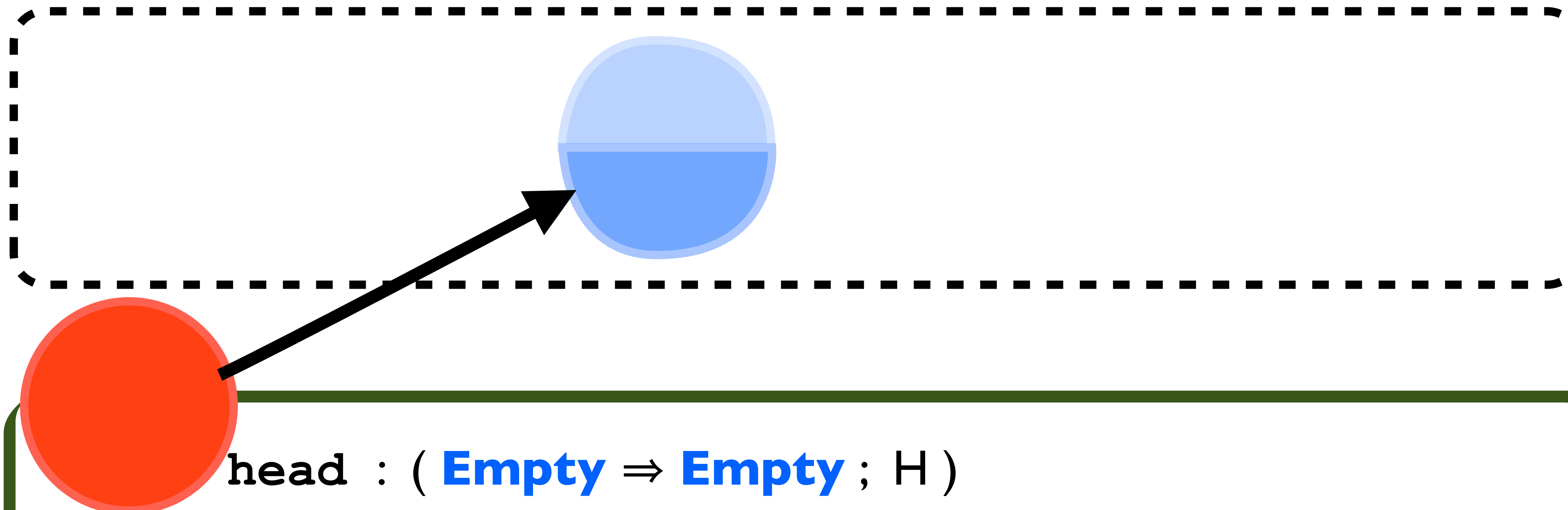
`head` : (**Empty** $\Rightarrow$ **Empty** ; H)

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

Consumer

Producer

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



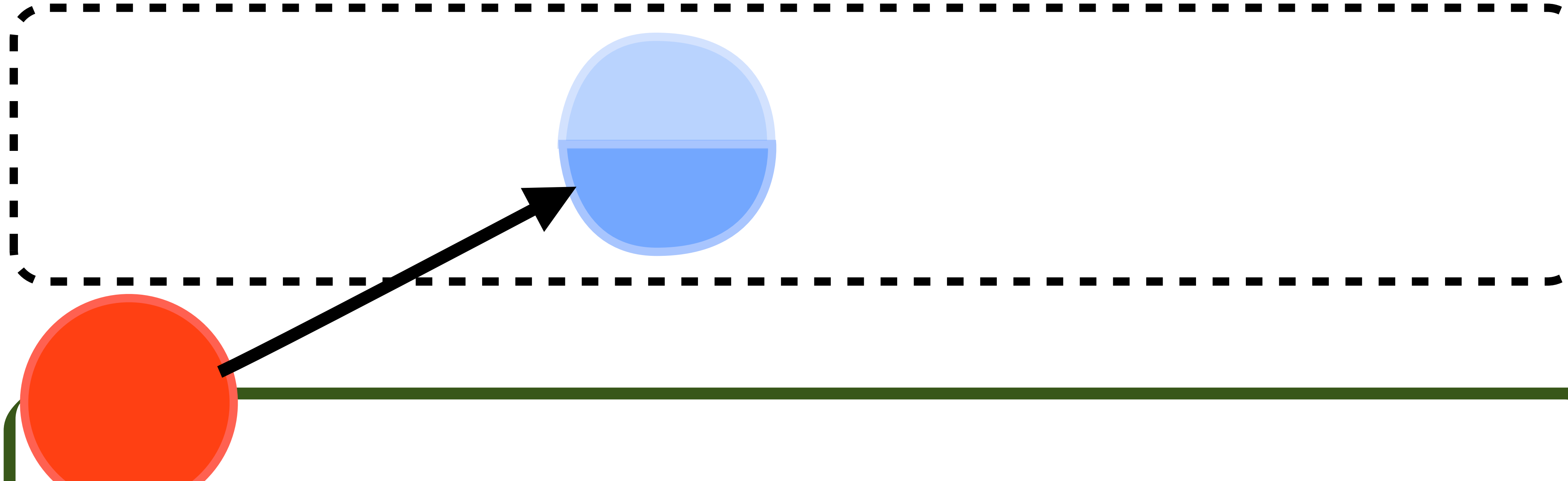
`head` : (**Empty** ⇒ **Empty** ; H)

⊕ (**Filled** ⇒ **none**) ⊕ (**Closed** ⇒ **none**)

Consumer

Producer

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

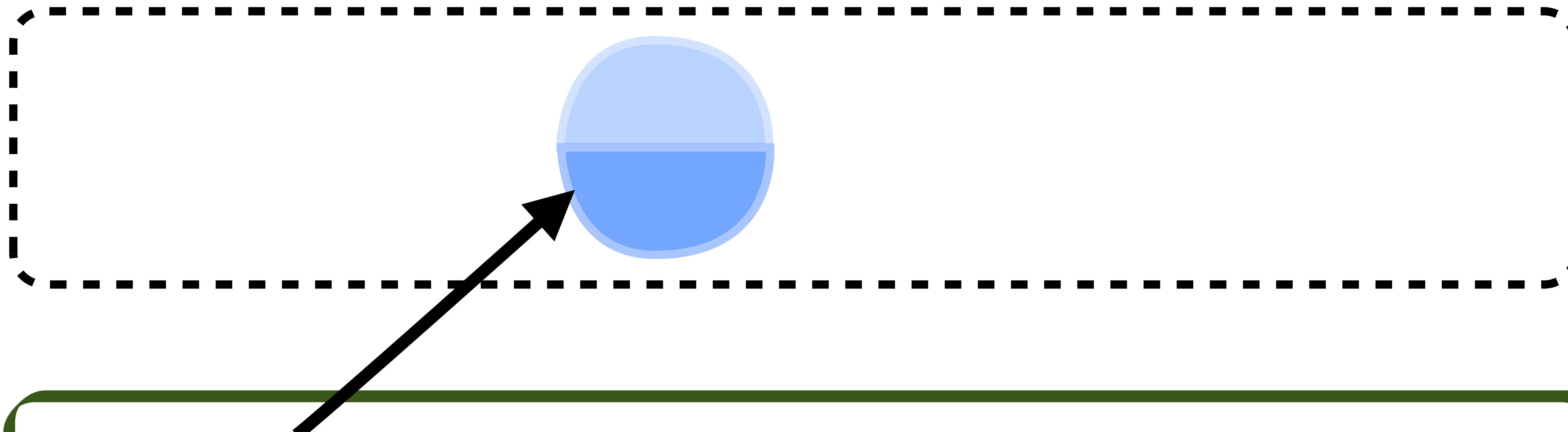


head : **none**

Consumer

Producer

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



`head` : (Empty ⇒ Empty ; H)

⊕ (Filled ⇒ none) ⊕ (Closed ⇒ none)

Consumer

Types

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

Types

$$
\begin{array}{lcll}
A & ::= & !A & \text{(pure type)} \\
 & | & A \multimap A & \text{(linear function)} \\
 & | & \forall X.A & \text{(universal type quantification)} \\
 & | & \exists X.A & \text{(existential type quantification)} \\
 & | & [\overline{\mathtt{f} : A}] & \text{(record)} \\
 & | & \sum_i \mathtt{t}_i\#A_i & \text{(tagged sum)} \\
 & | & \mathbf{ref}\ p & \text{(reference type)} \\
 & | & (\mathbf{rec}\ X(\overline{u}).A)[\overline{U}] & \text{(recursive type)} \\
 & | & X[\overline{U}] & \text{(type variable)} \\
 & | & A :: A & \text{(resource stacking)} \\
 & | & A * A & \text{(separation)} \\
 & | & \forall l.A & \text{(universal location quantification}
\end{array}
$$

Types

A	$::=$	$!A$	(pure type)
	$\mid$	$A \multimap A$	(linear function)
	$\mid$	$\forall X.A$	(universal type quantification)
	$\mid$	$\exists X.A$	(existential type quantification)
	$\mid$	$[\overline{\mathtt{f} : A}]$	(record)
	$\mid$	$\sum_i \mathtt{t}_i \# A_i$	(tagged sum)
	$\mid$	$\mathbf{ref}\ p$	(reference type)
	$\mid$	$(\mathbf{rec}\ X(\overline{u}).A)[\overline{U}]$	(recursive type)
	$\mid$	$X[\overline{U}]$	(type variable)
	$\mid$	$A :: A$	(resource stacking)
	$\mid$	$A * A$	(separation)
	$\mid$	$\forall l.A$	(universal location quantification

Types

$$
\begin{array}{lcll}
A & ::= & !A & \text{(pure type)} \\
 & \mid & A \multimap A & \text{(linear function)} \\
 & \mid & \forall X.A & \text{(universal type quantification)} \\
 & \mid & \exists X.A & \text{(existential type quantification)} \\
 & \mid & [\overline{\mathtt{f} : A}] & \text{(record)} \\
 & \mid & \sum_i \mathtt{t}_i \# A_i & \text{(tagged sum)} \\
 & \mid & \mathbf{ref}\ p & \text{(reference type)} \\
 & \mid & (\mathbf{rec}\ X(\overline{u}).A)[\overline{U}] & \text{(recursive type)} \\
 & \mid & X[\overline{U}] & \text{(type variable)} \\
 & \mid & A :: A & \text{(resource stacking)} \\
 & \mid & A * A & \text{(separation)} \\
 & \mid & \forall l.A & \text{(universal location quantification}
\end{array}
$$

$\mid$	$\mathbf{ref}\ p$	(reference type)
$\mid$	$(\mathbf{rec}\ X(\overline{u}).A)[\overline{U}]$	(recursive type)
$\mid$	$X[\overline{U}]$	(type variable)
$\mid$	$A :: A$	(resource stacking)
$\mid$	$A * A$	(separation)
$\mid$	$\forall l.A$	(universal location quantification
$\mid$	$\exists l.A$	(existential location quantificatio
$\mid$	$A \oplus A$	(alternative)
$\mid$	$A \,\&\, A$	(intersection)
$\mid$	$\mathbf{rw}\ p\ A$	(read-write capability to p)
$\mid$	**none**	(empty resource)
$\mid$	$A \Rightarrow A$	(rely type)
$\mid$	$A; A$	(guarantee type)

Producer (tail) **Protocol**:

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

Consumer (head) **Protocol**:

```
H[h] =
    ( rw h Empty#[] ⇒ rw h Empty#[] ; H[h] )
  ⊕ ( rw h Node#[…] ⇒ none ; none )
  ⊕ ( rw h 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])
```

Rely-Guarantee Protocols

- An *interference*-control mechanism, permission to mutate the shared state is conditioned on what actions the protocol allows.
- I will focus on presenting the following:
  1. Protocol Specification
  2. Protocol Use
  3. Protocol Composition

Rely-Guarantee Protocols

- An *interference*-control mechanism, permission to mutate the shared state is conditioned on what actions the protocol allows.
- I will focus on presenting the following:
  1. Protocol Specification
  2. Protocol Use
  3. Protocol Composition

Syntax

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

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

2. Protocol Use

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

a) **Hide *privates* 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 / Defocus

Empty ⇒ Filled ; Next

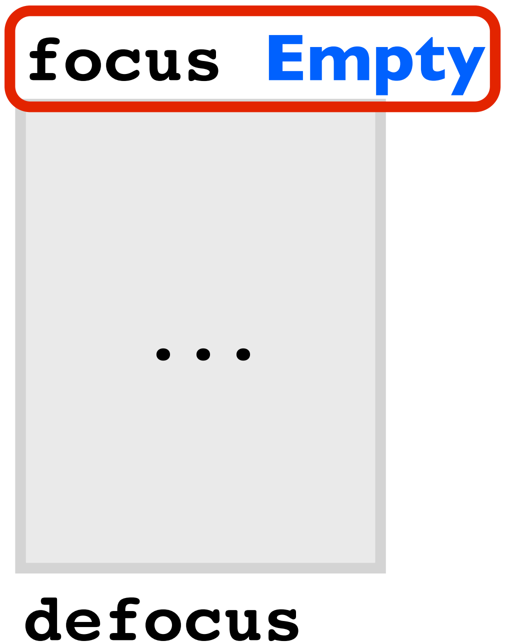
`focus` **Empty**

...

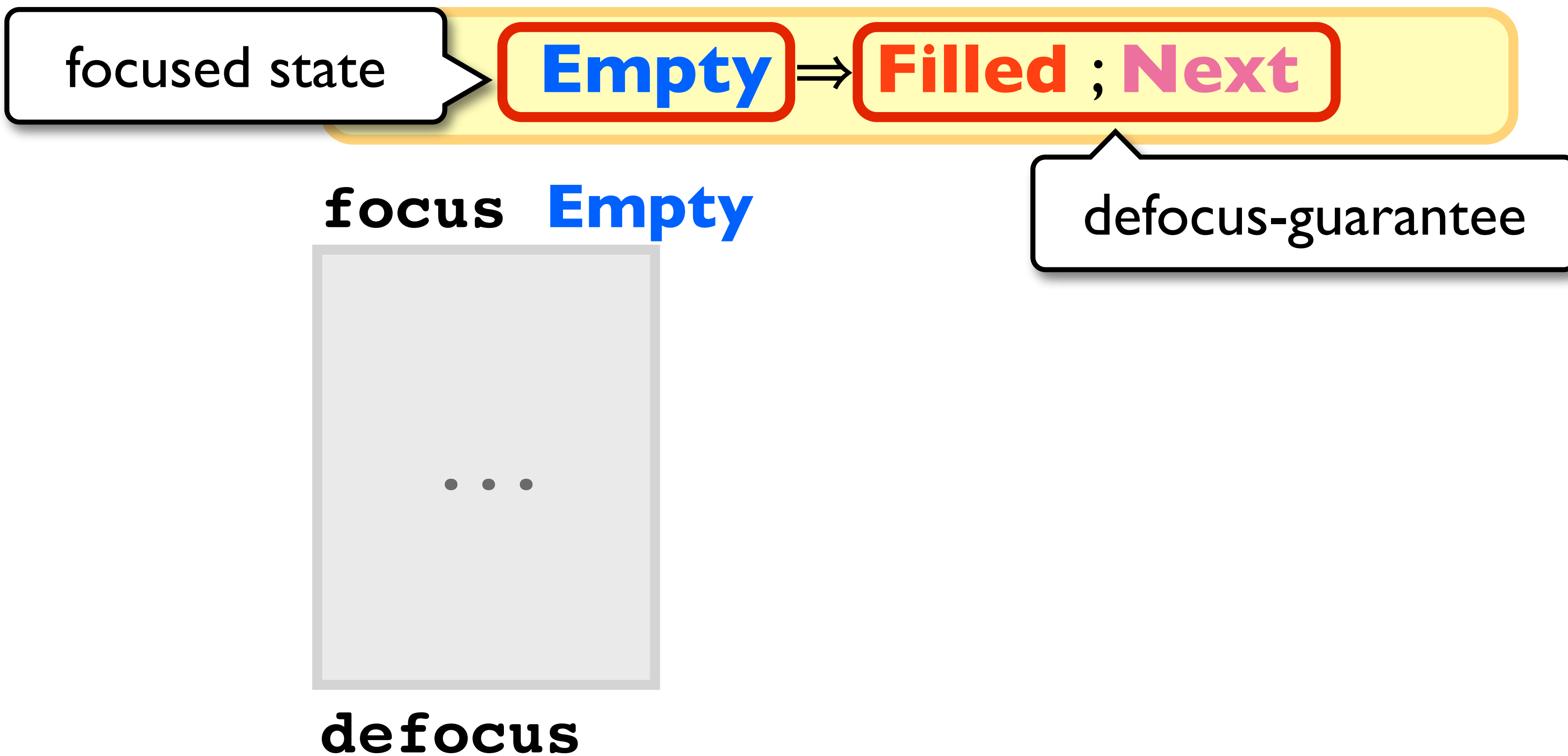
`defocus`

Focus / Defocus

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



Focus / 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

Rely-Guarantee Protocols

- An *interference*-control mechanism, permission to mutate the shared state is conditioned on what actions the protocol allows.
- I will focus on presenting the following:

1. Protocol Specification
2. Protocol Use
3. Protocol Composition

Rely-Guarantee Protocols

- An *interference*-control mechanism, permission to mutate the shared state is conditioned on what actions the protocol allows.
- I will focus on presenting the following:
    1. Protocol Specification
    2. Protocol Use
    3. $\boxed{\text{3. Protocol Composition}}$

3. Protocol Composition

- 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.
- **share** type checks only if the protocols compose safely (i.e. no unsafe interference is possible).

Checking `share`

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

a) simulating a single use of a protocols, a single **focus-defocus** block (i.e. a step of the protocol).

b) ensuring that each protocol considers all possible protocol interleaving.

Protocol Composition Example

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

Protocol Composition Example

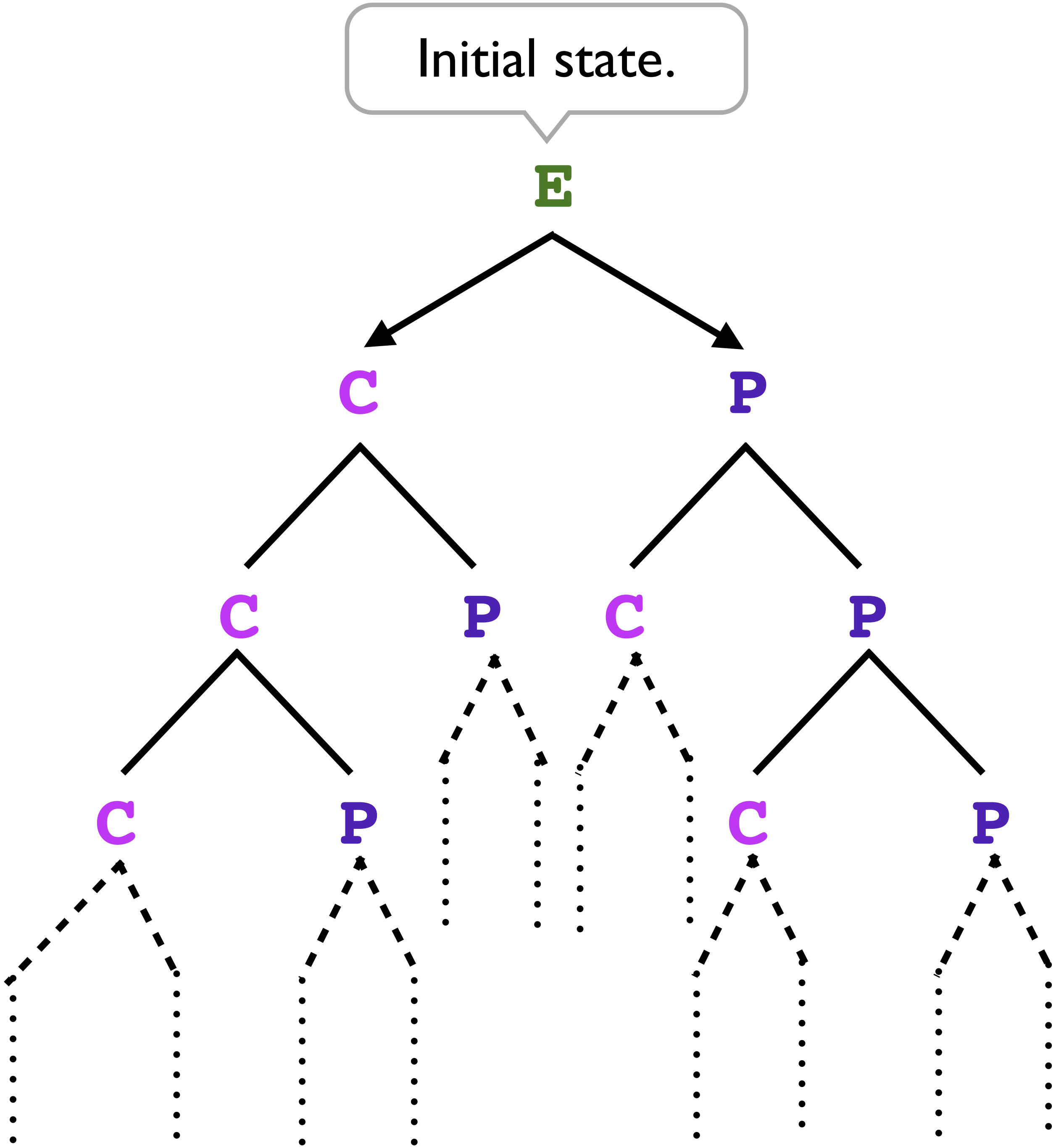
share $\mathtt{E}$ as

Consumer

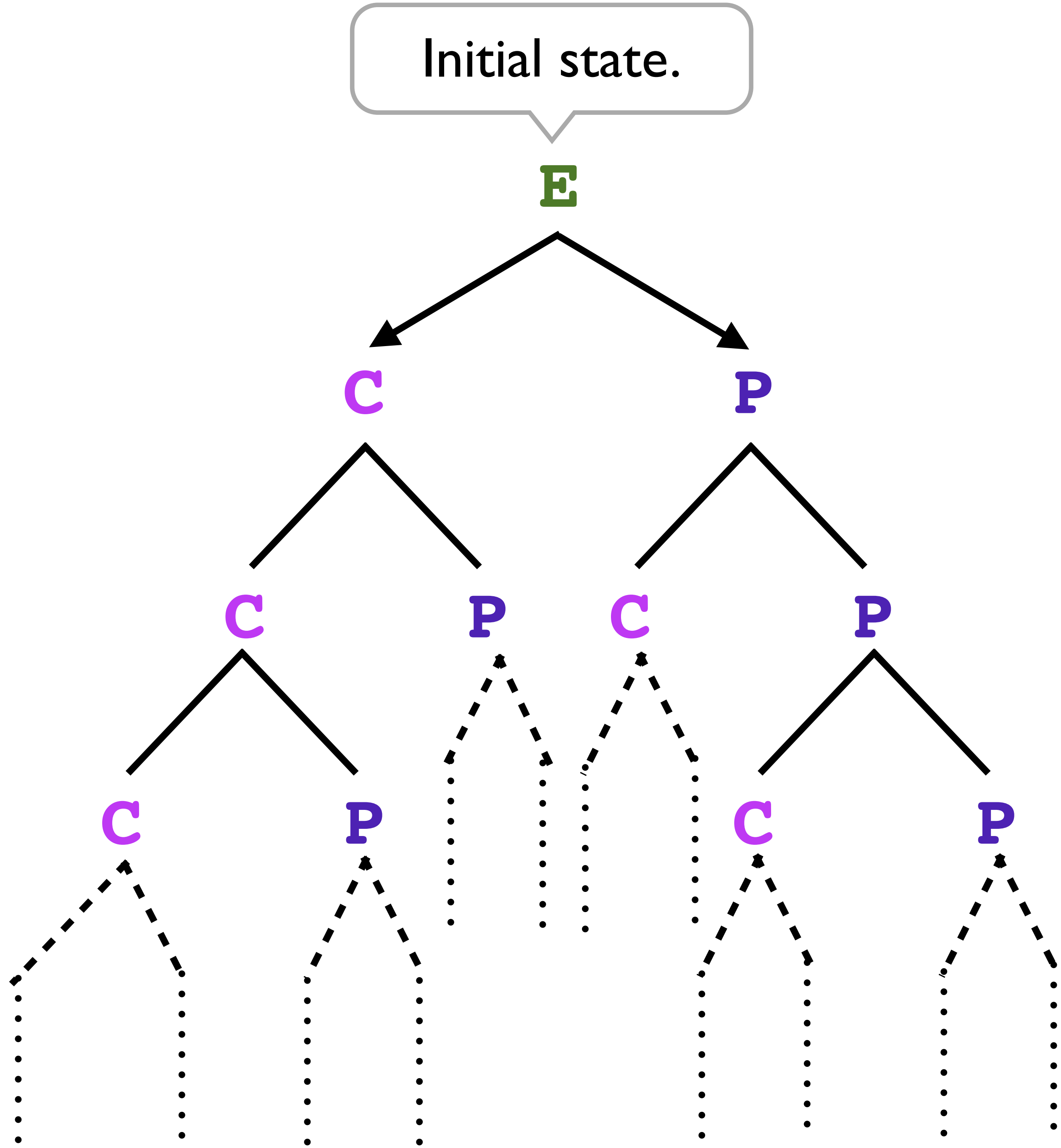
$\boxed{\mathtt{rec}\ \mathtt{X}.(\ \mathtt{E} \Rightarrow \mathtt{E};\mathtt{X} \oplus \mathtt{N} \Rightarrow \mathtt{none} \oplus \mathtt{C} \Rightarrow \mathtt{none}\ )}$

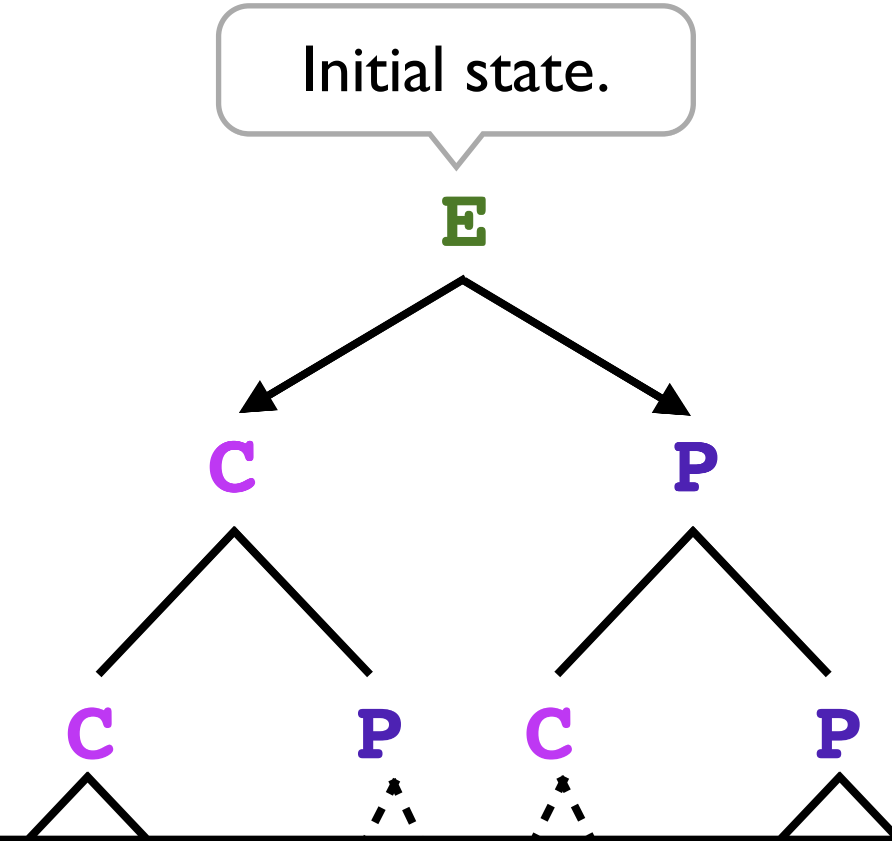
$\|\ \boxed{\mathtt{E} \Rightarrow (\ \mathtt{N} \oplus \mathtt{C}\ )}$

Producer


Initial state.
E
C
P
C
P
C
P
C
P
C
P

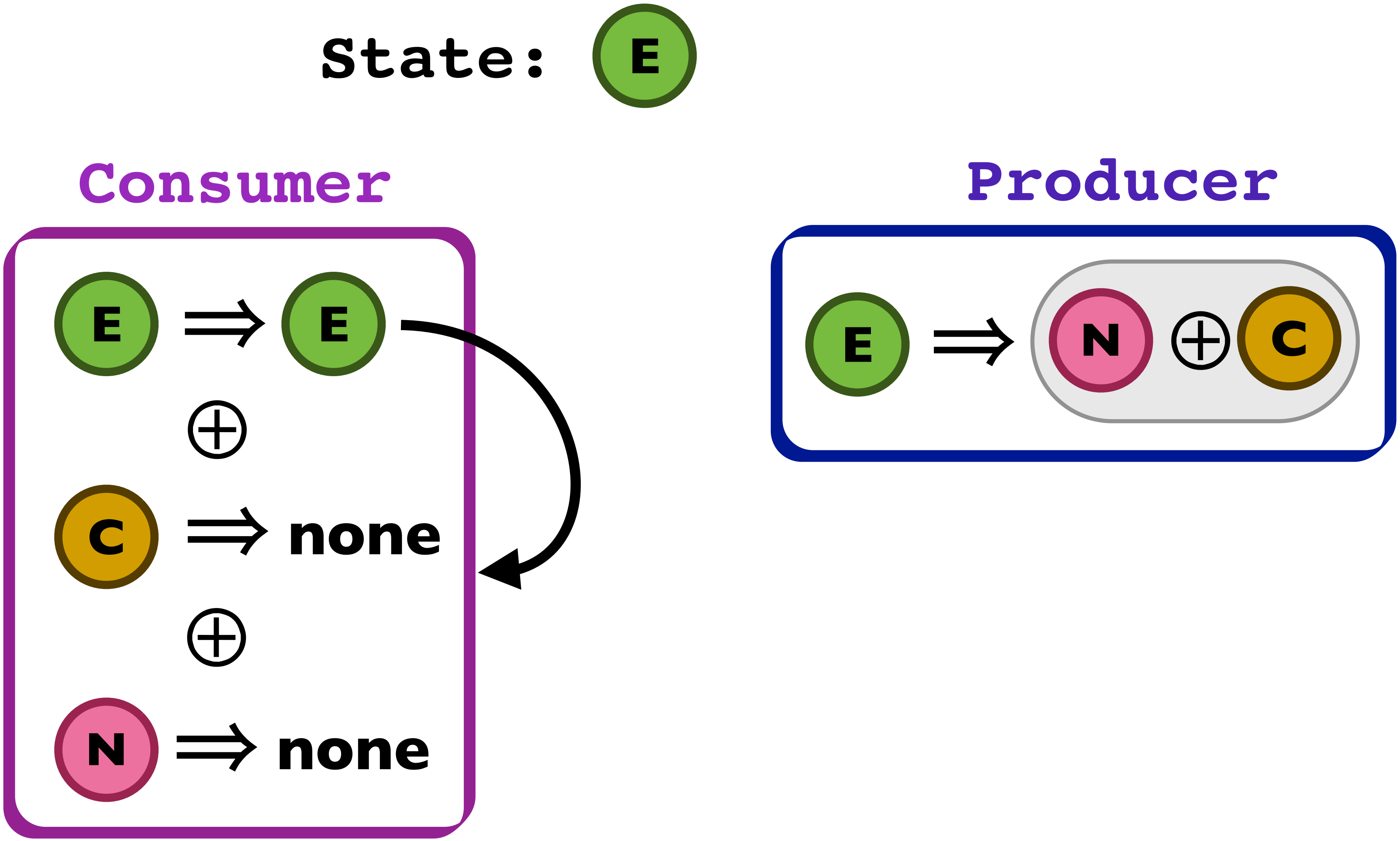
possible interleaving





possible interleaving

Up to this point protocols can only list a **finite number of distinct states**, and each protocol lists a **finite number of distinct protocol steps**. Thus, there is a finite number of different *configurations* representing the uses of the protocols.



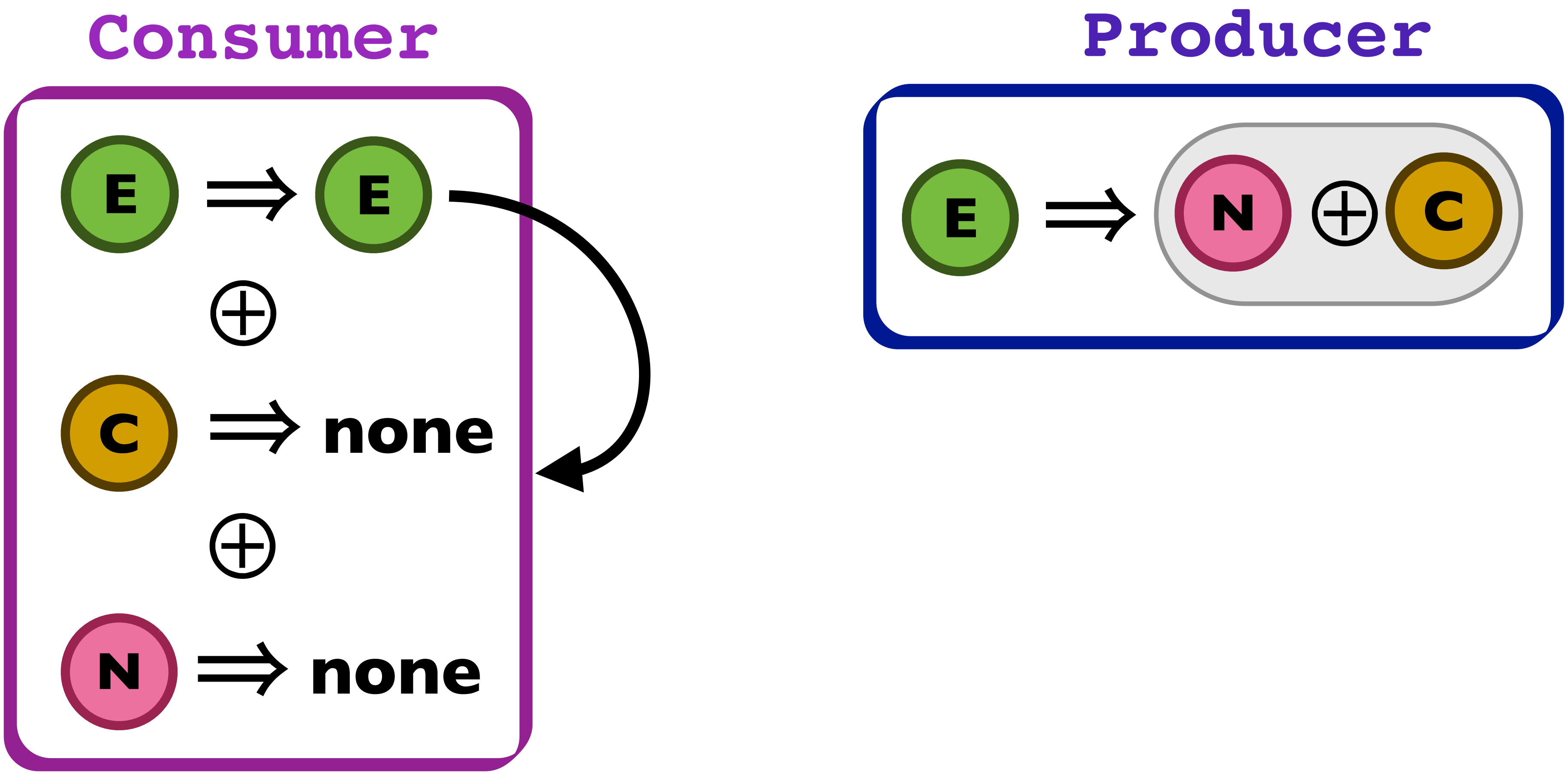
Configurations:

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

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

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

State:



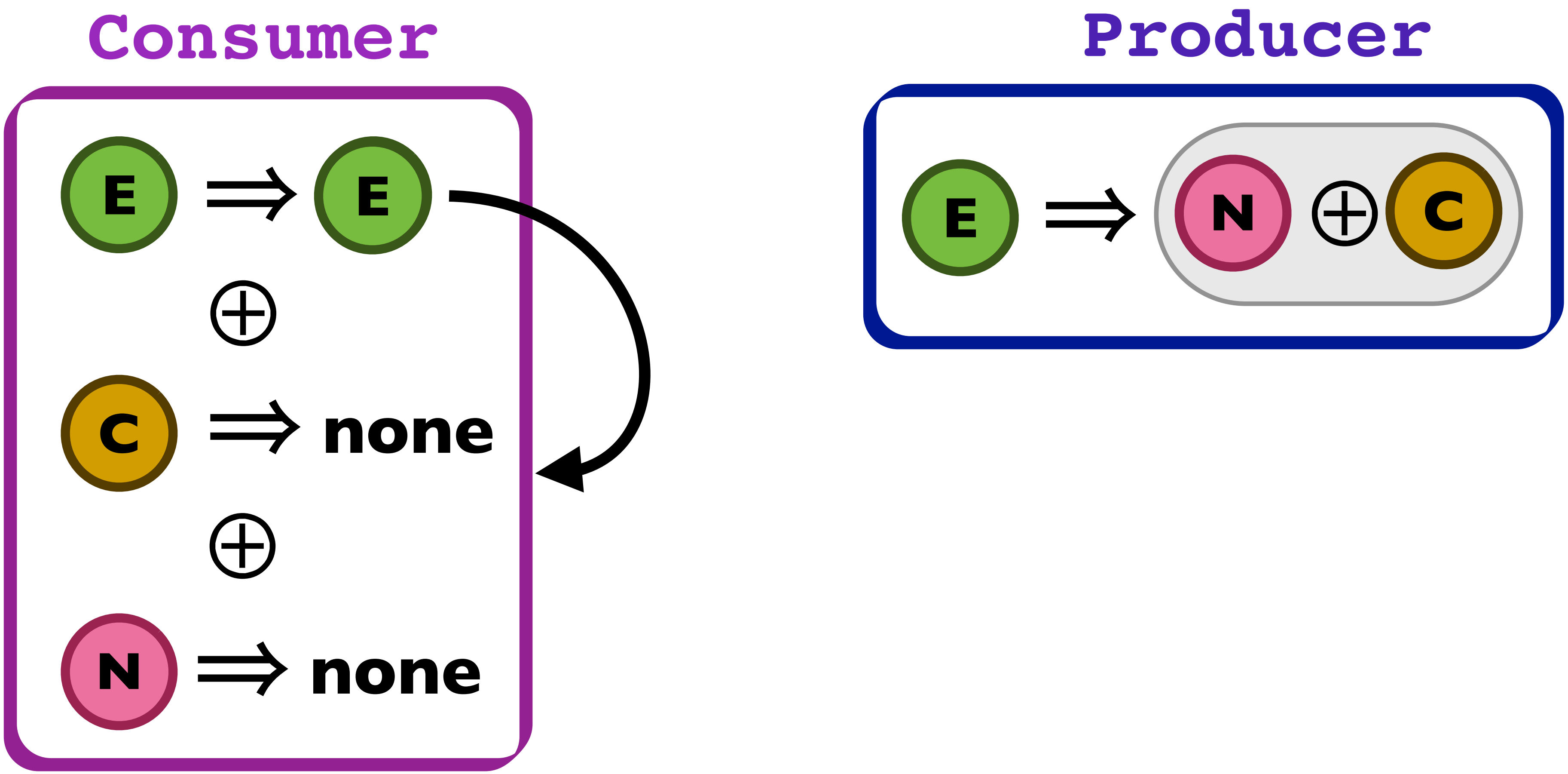
Configurations:

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

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

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

State:

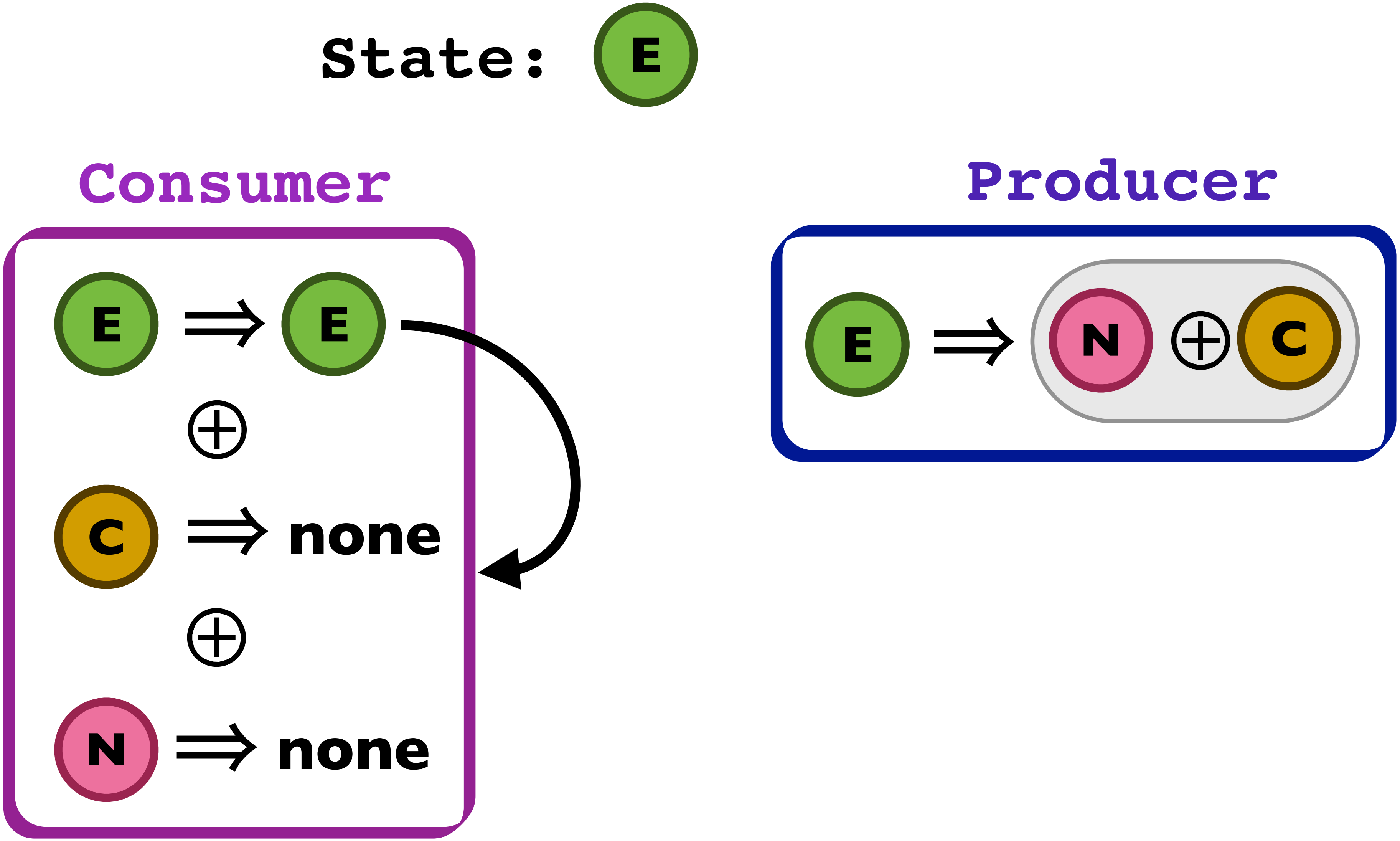


Configurations:

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

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

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

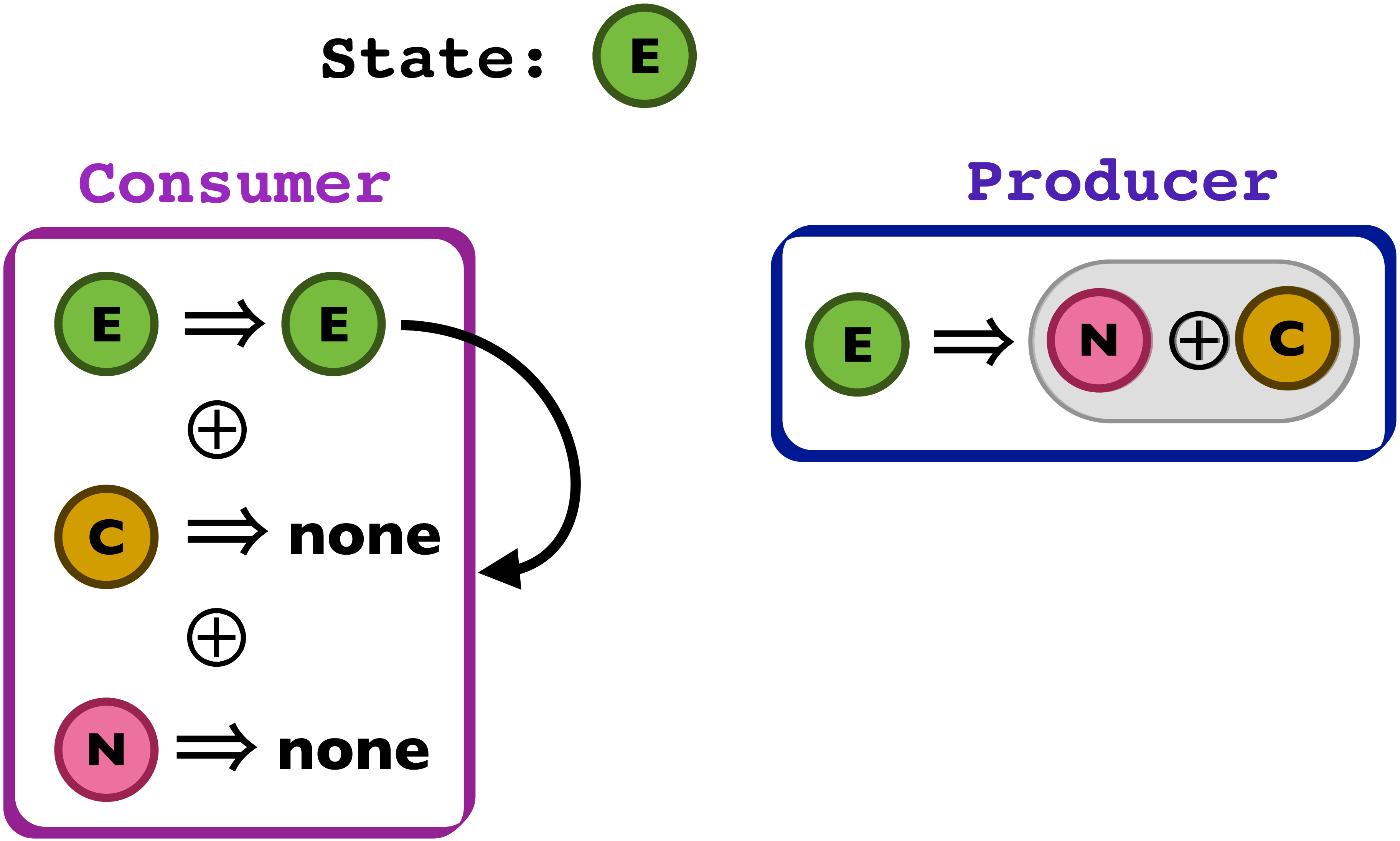


Configurations:

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

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

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

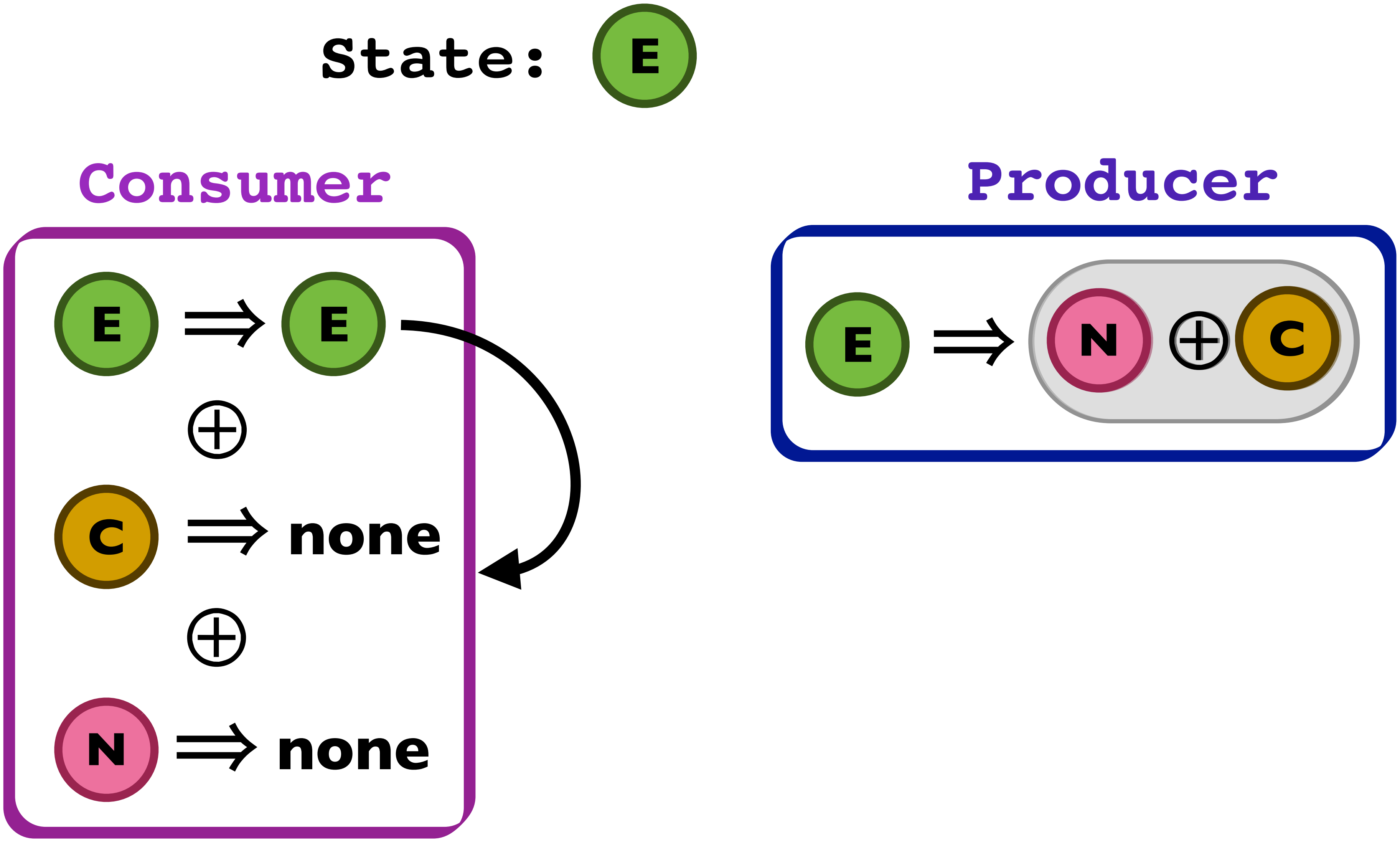


Configurations:

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

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

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



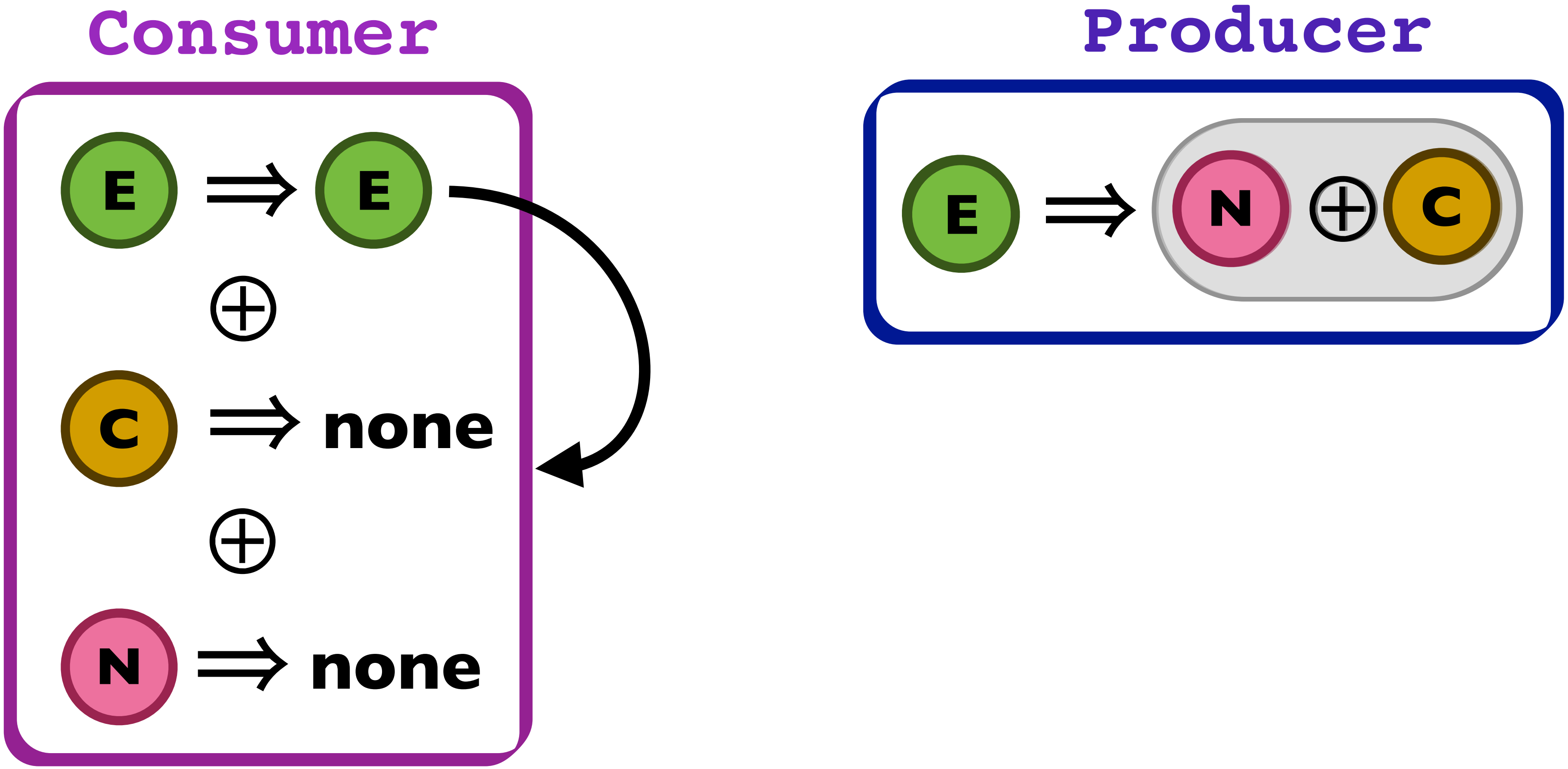
Configurations:

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

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

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

State:



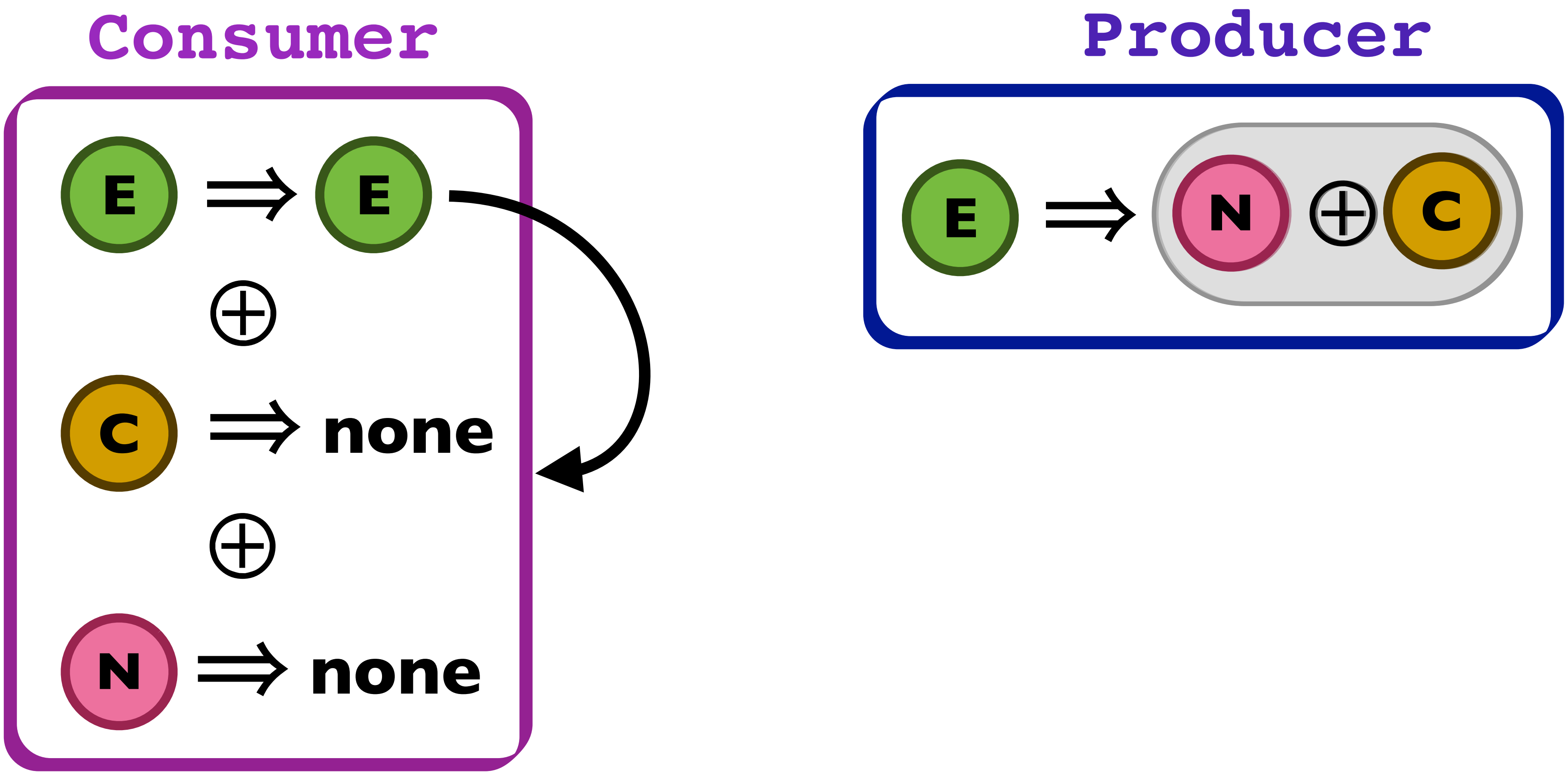
Configurations:

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

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

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

State:

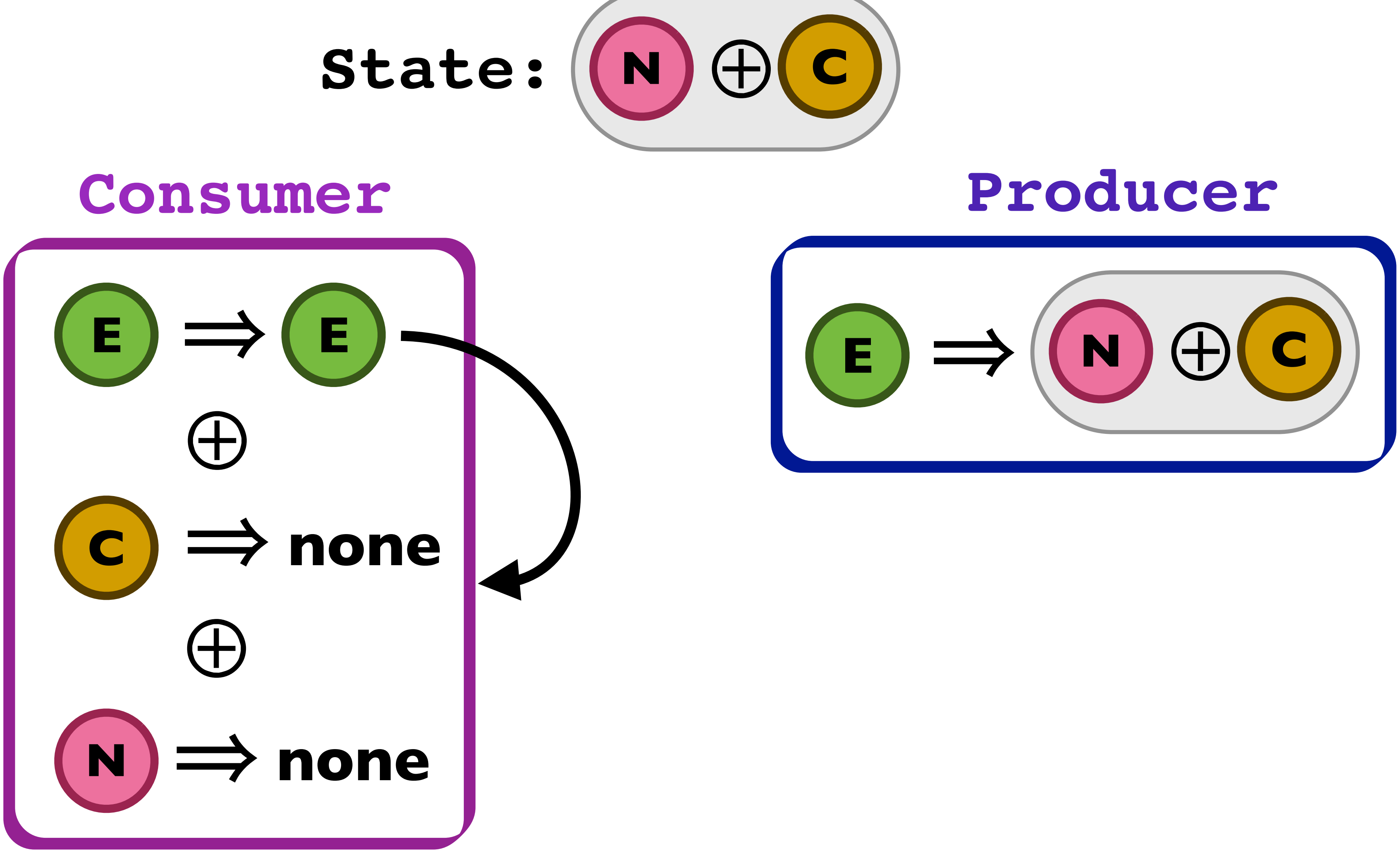


Configurations:

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

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

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

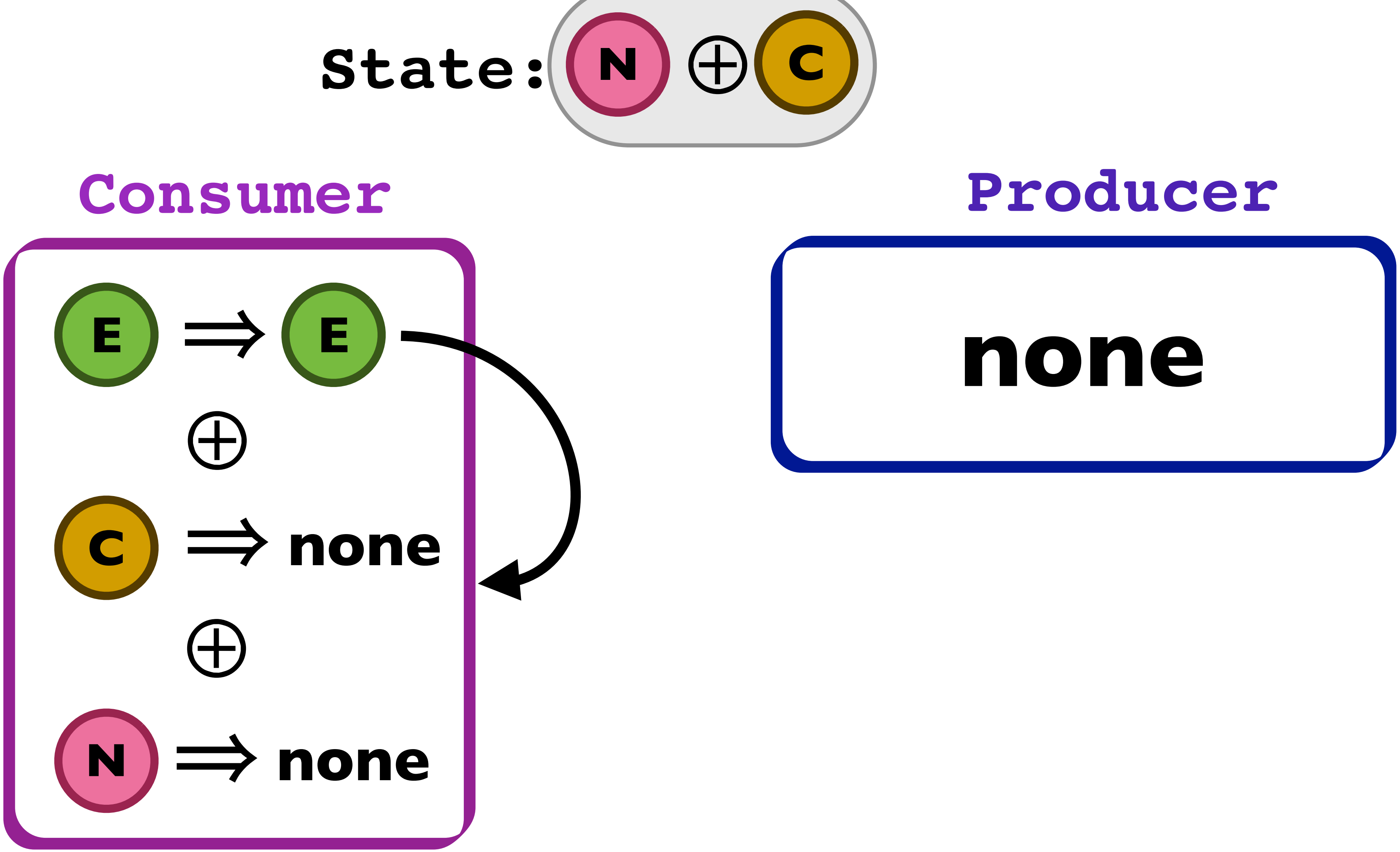


Configurations:

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

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

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

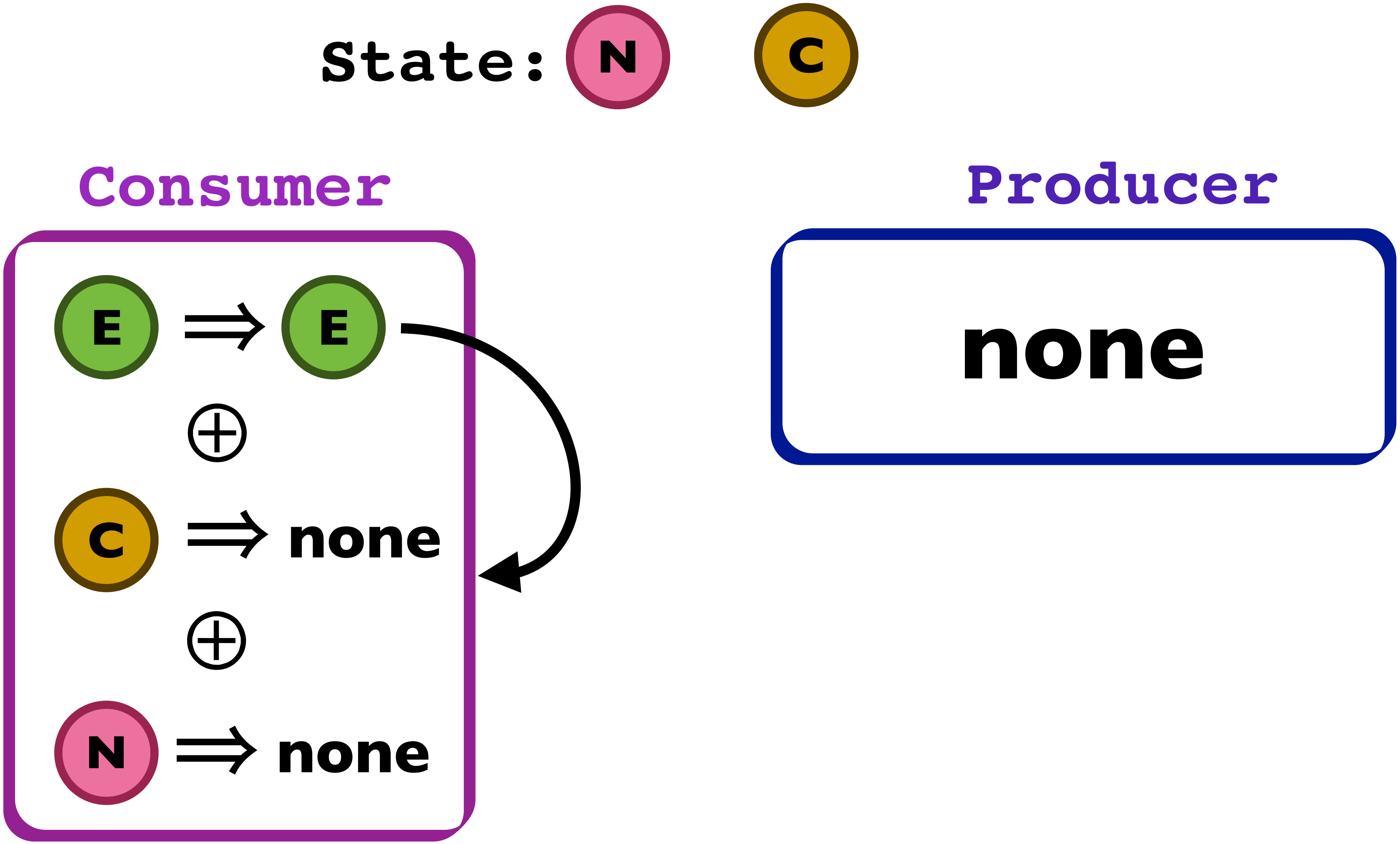


Configurations:

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

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

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



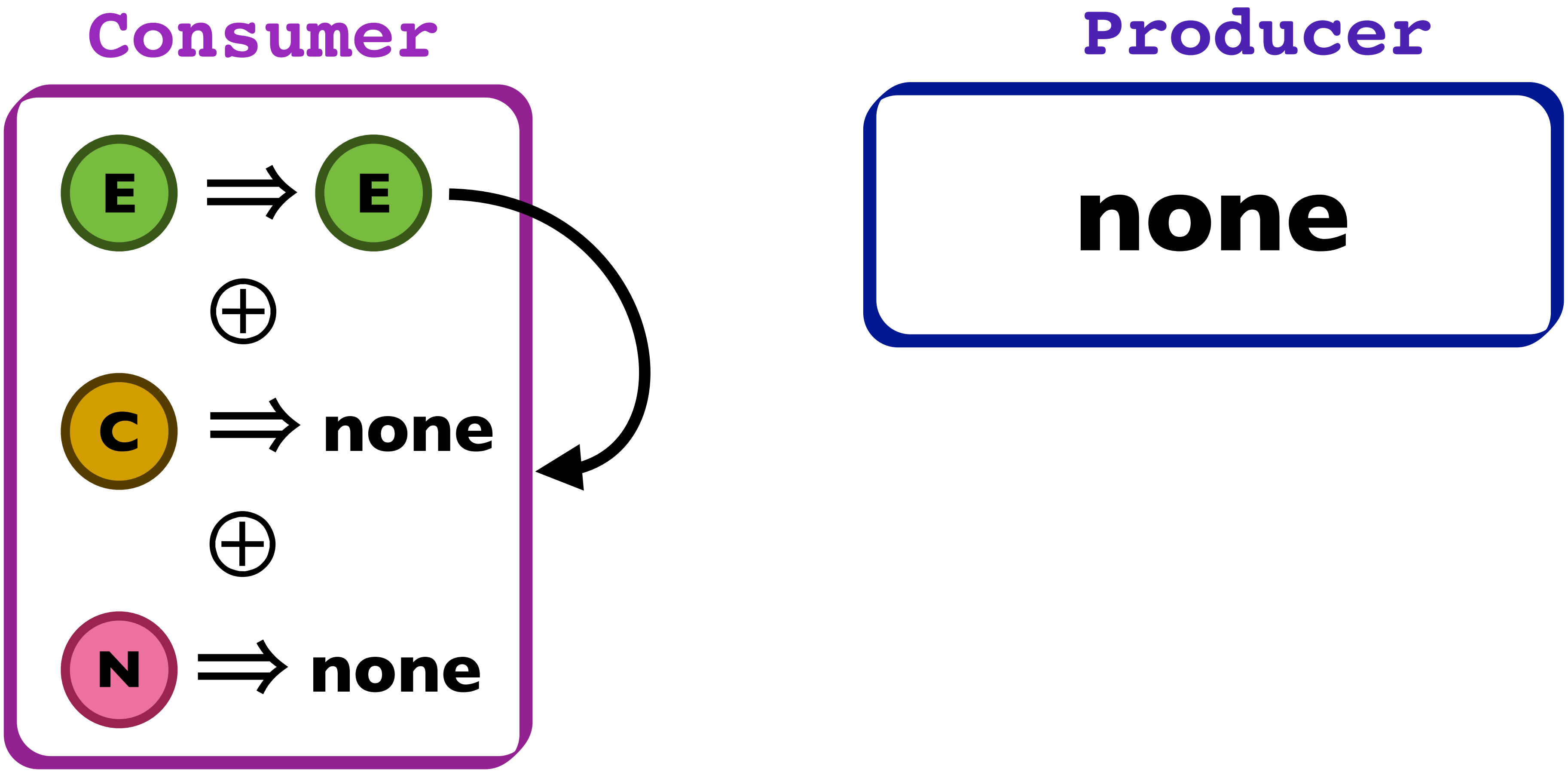
Configurations:

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

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

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

State: none



Configurations:

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

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

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

State: none

Consumer

none

Producer

none

Configurations:

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

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

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

Rely-Guarantee Protocols

- An *interference*-control mechanism, permission to mutate the shared state is conditioned on what actions the protocol allows.
- I will focus on presenting the following:

  1. Protocol Specification
  2. Protocol Use
  3. $\boxed{\text{Protocol Composition}}$

Protocols

- An *interference*-control mechanism, permission to mutate the shared state is conditioned on what actions the protocol allows.
- I will focus on presenting the following:
  1. Protocol Specification
  2. Protocol Use
  3. Protocol Composition

A few more details:

- Improved Locality of Protocol Specification
- Fork/Join Concurrency

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

H[h] =
   ( rw h Empty#[] ⇒ rw h Empty#[] ; H[h] )
  ⊕ ( rw h Node#[…] ⇒ none ; none )
  ⊕ ( rw h Closed#[] ⇒ none ; none )
```

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

From H's local ***perspective*** only the tag is important, not the tagged type. How can we model this?

```
H[h] =
  ( rw h Empty#[] ⇒ rw h Empty#[] ; H[h] )
 ⊕ ( rw h Node#[…] ⇒ none ; none )
 ⊕ ( rw h Closed#[] ⇒ none ; none )
```

Protocol Specification 2.0

$$
\begin{array}{rcl}
P, Q & ::= & (\mathbf{rec}\ X(\overline{u}).P)[\overline{U_P}] \\
& | & X[\overline{U_P}] \\
& | & P \oplus P \\
& | & P \;\&\; P \\
& | & R \Rightarrow P \\
& | & R\,;\ P \\
& | & \textbf{none} \\
& | & \exists l.P \\
& | & \forall l.P \\
& | & \exists X <: A.P \\
& | & \forall X <: A.P
\end{array}
$$

$$\mathtt{T} = \textbf{Empty}\#[\,] \Rightarrow \textbf{Full}\#\mathtt{int}\,;\mathtt{none}$$

$$\mathtt{H} = (\,\textbf{Empty}\#[\,] \Rightarrow \textbf{Empty}\#[\,]\,;\mathtt{H}\,)$$
$$\oplus\,(\,\textbf{Full}\#\mathtt{int} \Rightarrow \mathtt{none}\,;\mathtt{none}\,)$$
$$\oplus\,(\,\textbf{Close}\#[\,] \Rightarrow \mathtt{none}\,;\mathtt{none}\,)$$

$$\mathtt{T} = \textbf{Empty}\#[\,] \Rightarrow \textbf{Full}\#\mathtt{int} \,;\, \textbf{none}$$

$$\begin{aligned}\mathtt{H} = \exists X.(\, &\textbf{Empty}\#X \Rightarrow \textbf{Empty}\#X \,;\, \mathtt{H}\,)\\ &\oplus (\,\textbf{Full}\#\mathtt{int} \Rightarrow \textbf{none} \,;\, \textbf{none}\,)\\ &\oplus (\,\textbf{Close}\#[\,] \Rightarrow \textbf{none} \,;\, \textbf{none}\,)\end{aligned}$$

`H` no longer sees the content of **Empty**,
and `T` can now use that part of the state unilaterally.

$$
\begin{aligned}
\mathtt{T} = {} & (\,\mathbf{Empty}\#[\,] \Rightarrow \mathbf{Empty}\#[\,]\,;\,\mathtt{T}\,) \\
& \&\,(\mathbf{Empty}\#[\,] \Rightarrow \mathbf{Full}\#\mathtt{int}\,;\,\mathbf{none})
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{H} = \exists X.\,& (\,\mathbf{Empty}\#X \Rightarrow \mathbf{Empty}\#X\,;\,\mathtt{H}\,) \\
& \oplus (\,\mathbf{Full}\#\mathtt{int} \Rightarrow \mathbf{none}\,;\,\mathbf{none}\,) \\
& \oplus (\,\mathbf{Close}\#[\,] \Rightarrow \mathbf{none}\,;\,\mathbf{none}\,)
\end{aligned}
$$

T remembers the local information.

T is free to unilaterally change tagged type.

$$\mathtt{T[Y]} = (\textbf{Empty}\#\mathtt{Y} \Rightarrow \boxed{\forall Z.(\textbf{Empty}\#Z\,;\mathtt{T}[Z])})$$
$$\quad \& \,(\textbf{Empty}\#\mathtt{Y} \Rightarrow \textbf{Full}\#\mathtt{int}\,;\textbf{none})$$

$$\mathtt{H} = \exists X.(\textbf{Empty}\#X \Rightarrow \textbf{Empty}\#X\,;\mathtt{H})$$
$$\quad \oplus \,(\textbf{Full}\#\mathtt{int} \Rightarrow \textbf{none}\,;\textbf{none})$$
$$\quad \oplus \,(\textbf{Close}\#[] \Rightarrow \textbf{none}\,;\textbf{none})$$

Improved Locality

- Protocols can be more "generic" and more isolated (decoupled) from other protocols' actions, while remaining free from unsafe interference.
- Protocol re-splits can take advantage of this flexibility by means of more flexible *specializations*, such as via nested re-splits.

Protocol *Specialization*

$$\texttt{T[Y]} = (\textbf{Empty}\#\texttt{Y} \Rightarrow \forall Z.(\textbf{Empty}\#Z\,;\,\texttt{T[}Z\texttt{]}))$$
$$\&\,(\textbf{Empty}\#\texttt{Y} \Rightarrow \textbf{Full}\#\texttt{int}\,;\,\texttt{none})$$

re-split

$$(\,\textbf{Empty}\#(\textbf{Wait}\#[\,]) \Rightarrow \textbf{Empty}\#(\textbf{Ready}\#\texttt{int})\,;\,\texttt{none}\,)$$
$$\oplus\;(\,\textbf{Empty}\#(\textbf{Ready}\#\texttt{int}) \Rightarrow \textbf{Full}\#\texttt{int}\,;\,\texttt{none}\,)$$

Protocol *Specialization*

$$\mathtt{T}[\mathtt{Y}] = (\,\mathbf{Empty}\#\mathtt{Y} \Rightarrow \forall Z.(\mathbf{Empty}\#Z \,;\, \mathtt{T}[Z])\,)$$
$$\&\,(\,\mathbf{Empty}\#\mathtt{Y} \Rightarrow \mathbf{Full}\#\mathtt{int} \,;\, \mathtt{none}\,)$$

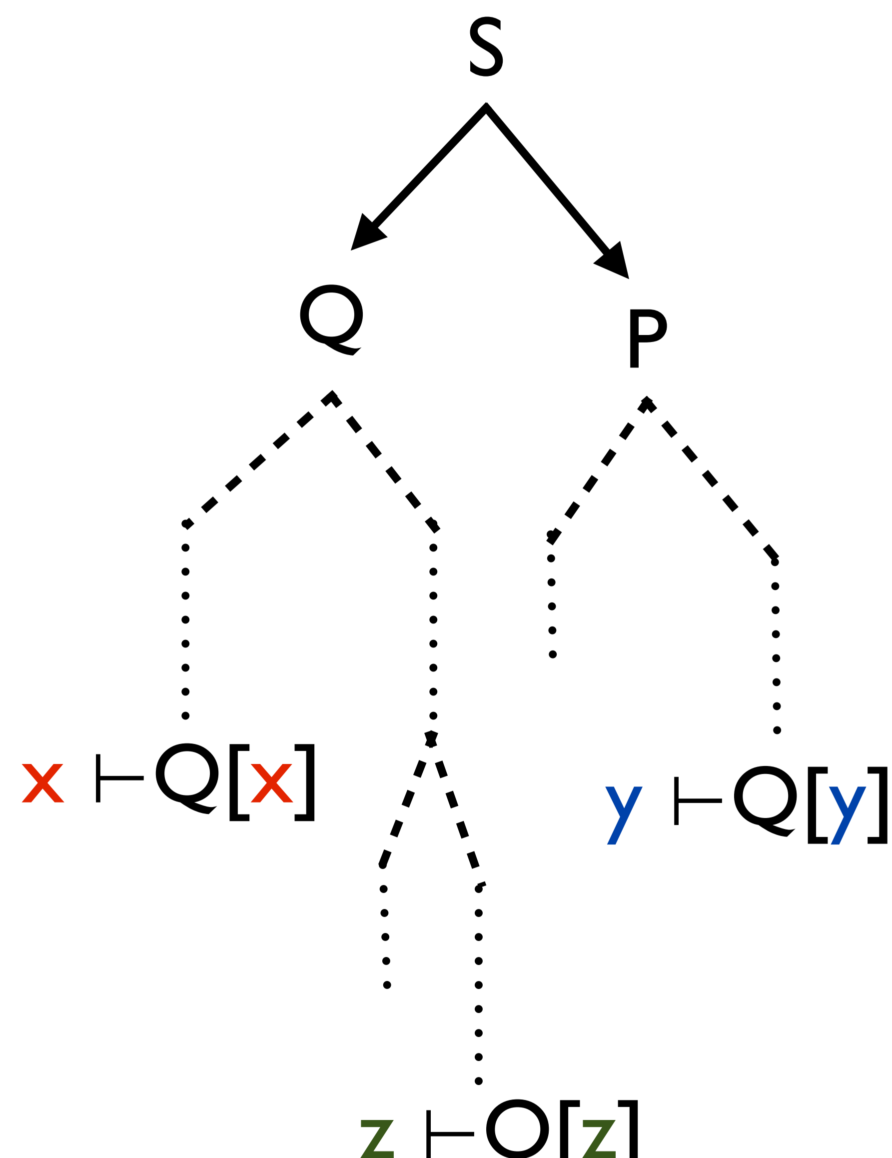
re-split

$$(\,\mathbf{Empty}\#\boxed{(\mathbf{Wait}\#[\,])} \Rightarrow \mathbf{Empty}\#\boxed{(\mathbf{Ready}\#\mathtt{int})} \,;\, \mathtt{none}\,)$$
$$\oplus\;(\,\mathbf{Empty}\#\boxed{(\mathbf{Ready}\#\mathtt{int})} \Rightarrow \mathbf{Full}\#\mathtt{int} \,;\, \mathtt{none}\,)$$

"Nesting", the specialization works *within* the original interference.

Protocol Composition 2.0

- Configurations are checked "symbolically", each representing a *class* of configurations, up to renaming and weakening.
- Well-formedness conditions on types ensures the number of different sub-terms of a type is bounded, guaranteeing decidability.



Concurrency

- Protocol Composition protects against all possible interleaving that may occur.
- Same static set of possible interleaving, even if concurrency may non-deterministically pick different interleaving at run-time.

  `focus/defocus` >> `lock/unlock`, adds `fork e`

- *Caveat*: no deadlock avoidance, nor termination/liveness guarantees.

Overview of Main Technical Results

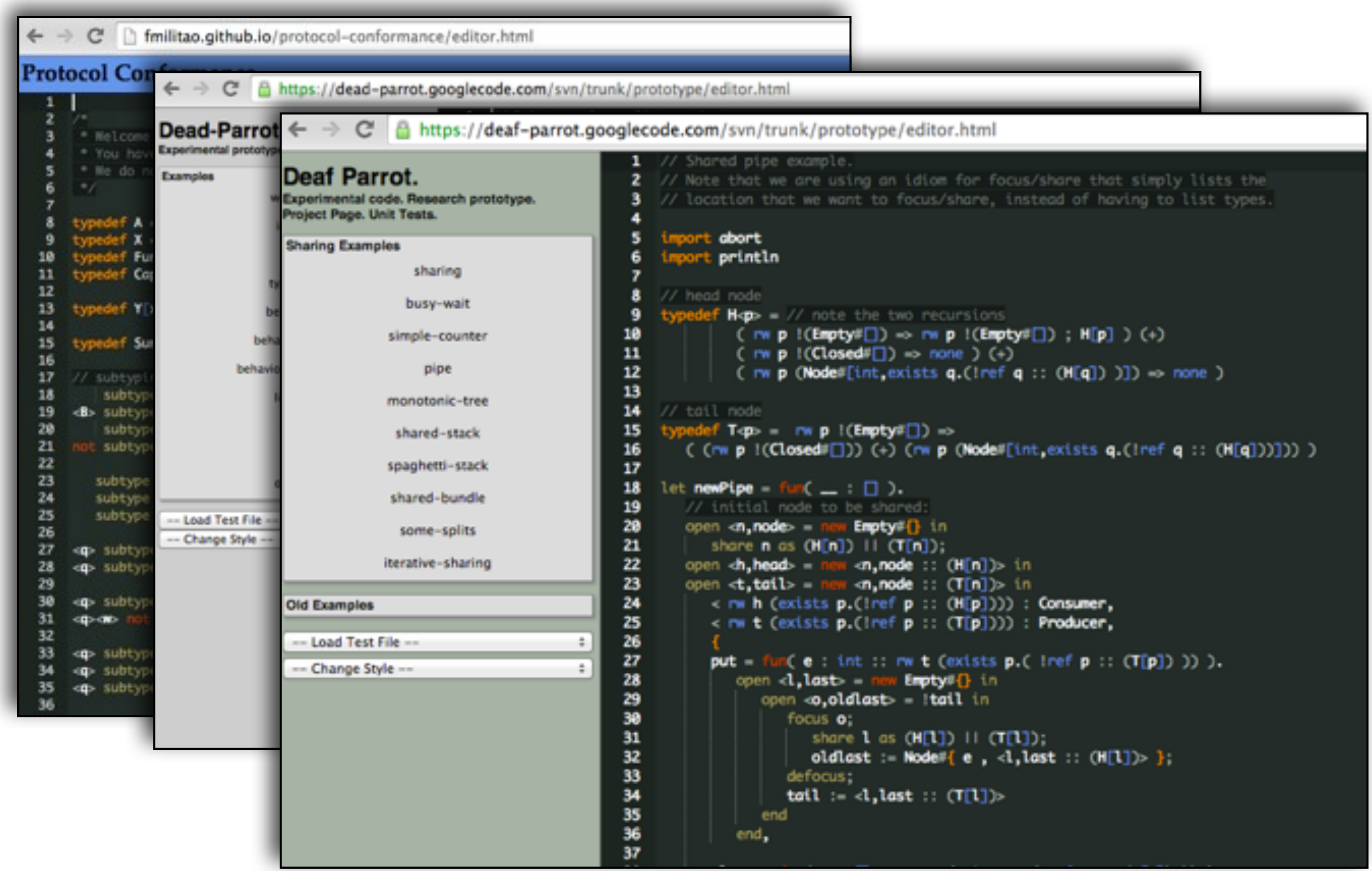
- Standard *Progress* and *Preservation* results.
- *Data race freedom* and *memory safety* (no dereference of null pointers, etc).
- Stepping of configurations preserves safe Protocol Composition.
- Protocol Composition is a *partial commutative monoid*.
- Protocol Composition is decidable.

Thesis also includes

- All technical details such as the axiomatic definition of Protocol Composition, algorithm, and formal system for sequential and concurrent settings.
- Additional examples, including:
  - Full pipe code example
  - Encoding (non-distributed) typeful message-passing concurrency (buyer-shipper-seller example)
- Soundness proof.
- Prototype implementations.

Prototypes

JavaScript-based implementations, run in the browser.



Summary

- Typestates using existential abstraction.

[Militão, Aldrich, Caires. **Substructural Typestates**. PLPV'14.]

- *Rely-Guarantee Protocols* for handling safe interference:
    1. Protocol Specification ("local view on public changes")
    2. Protocol Use ("hidden, private changes")
    3. Protocol Composition ("ensure safe alias interleaving")

[Militão, Aldrich, Caires. **Rely-Guarantee Protocols**. ECOOP'14.]

- Safe, decidable composition of protocols even when parts of a protocol are abstracted.
- Interference control over sequential and concurrent settings.