Carnegie Mellon
SCHOOL OF COMPUTER SCIENCE

Session-Typed Concurrent Contracts

HANNAH GOMMERSTADT

LIMIN JIA & FRANK PFENNING

Sorting Process

Sorting Process

← 3

Sorting Process

← 3

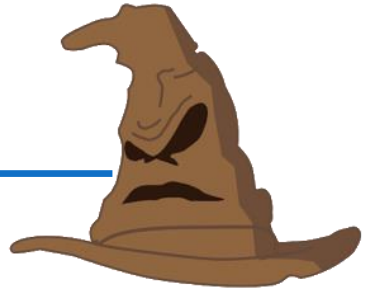
← 5

Sorting Process

← 3

← 5

← 4

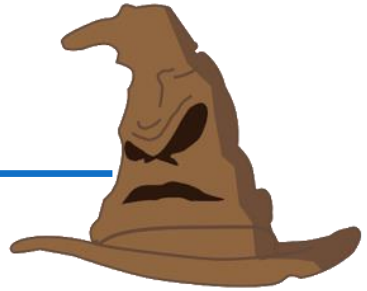


Sorting Process

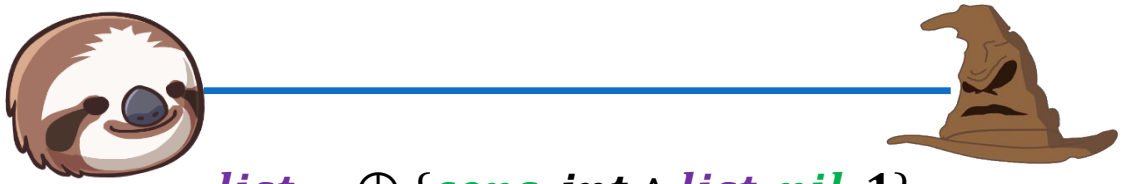
← 3

← 5

← 4

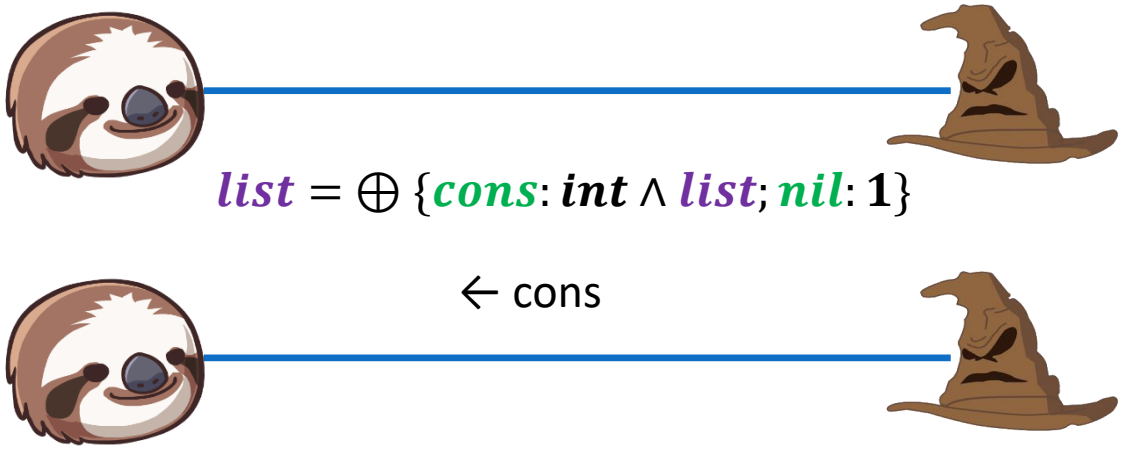


Session Types?!

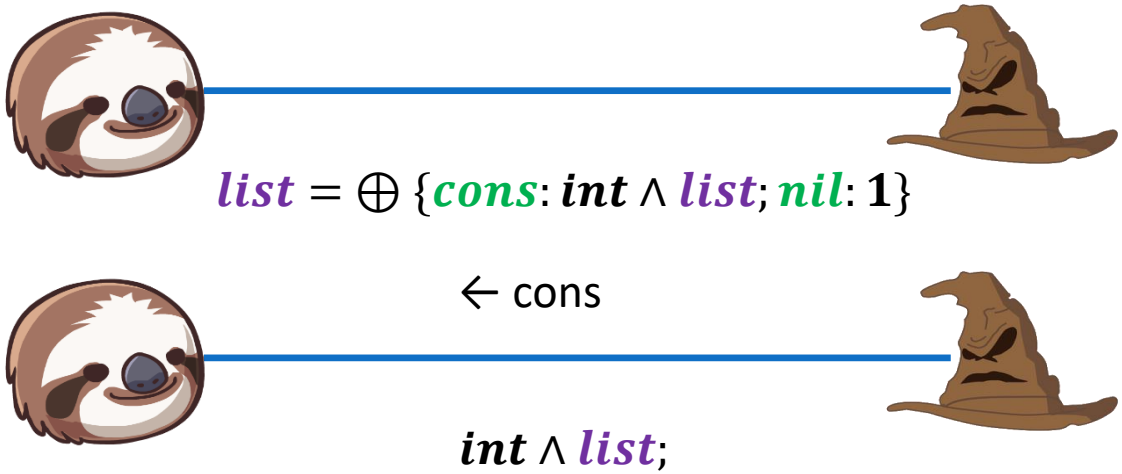


$$\mathit{list} = \oplus\{\mathit{cons}: \mathit{int} \wedge \mathit{list}; \mathit{nil}: \mathbf{1}\}$$

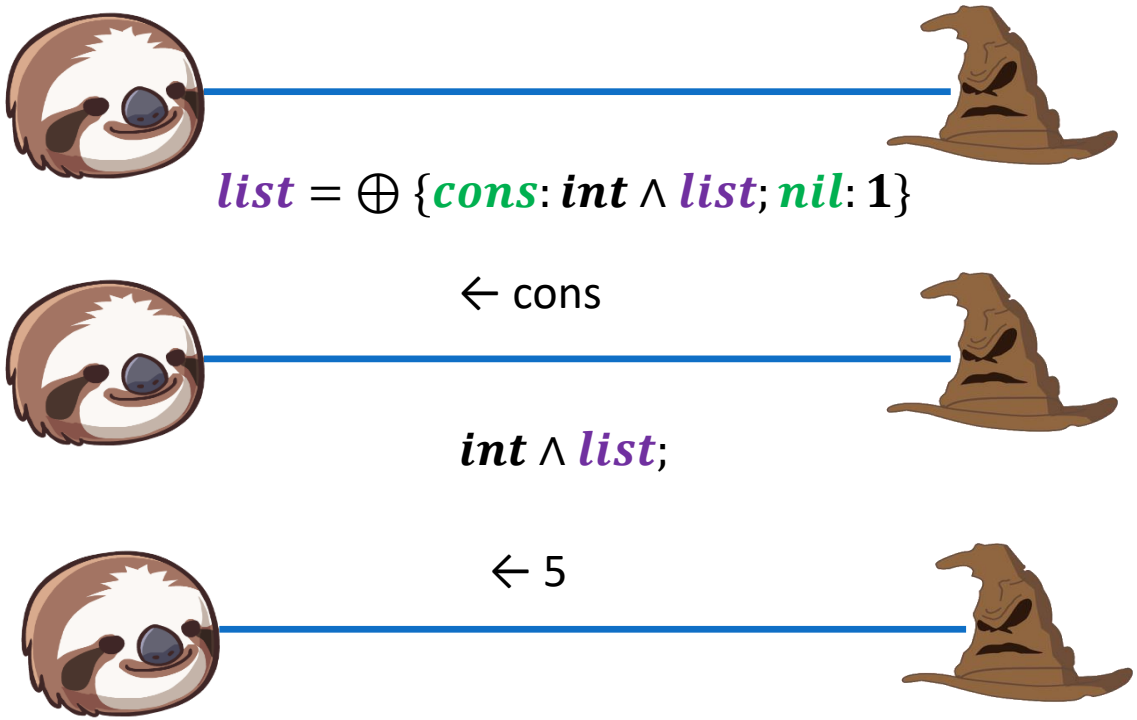
Session Types?!



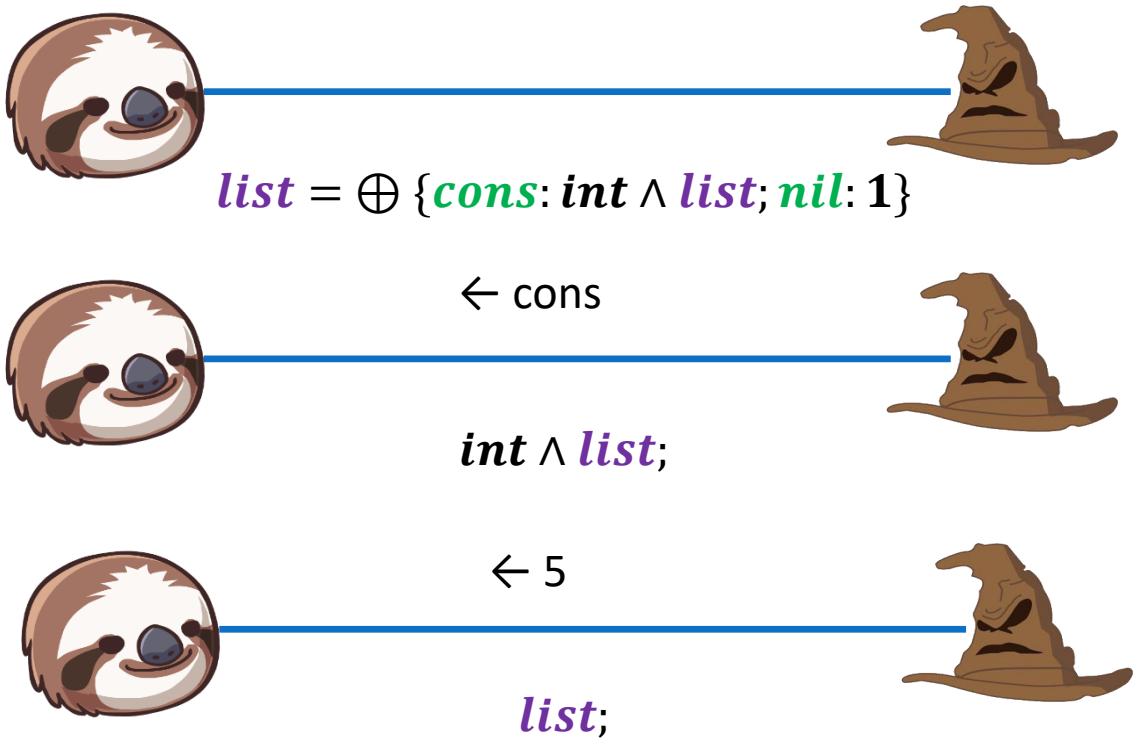
Session Types?!



Session Types?!



Session Types?!

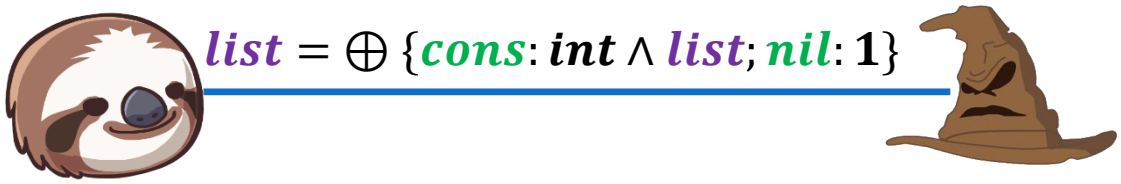


Session Types as Contracts

- Session types prescribe communication contracts between processes
- Dynamically typechecked because processes are untrusted
- Prior work (POPL 2016) on monitoring and blame assignment

Sorting Contract

Contract we have:



Contract we want:

All received integers are in ascending order

Challenge

How do we dynamically monitor contracts that cannot be expressed as session type?

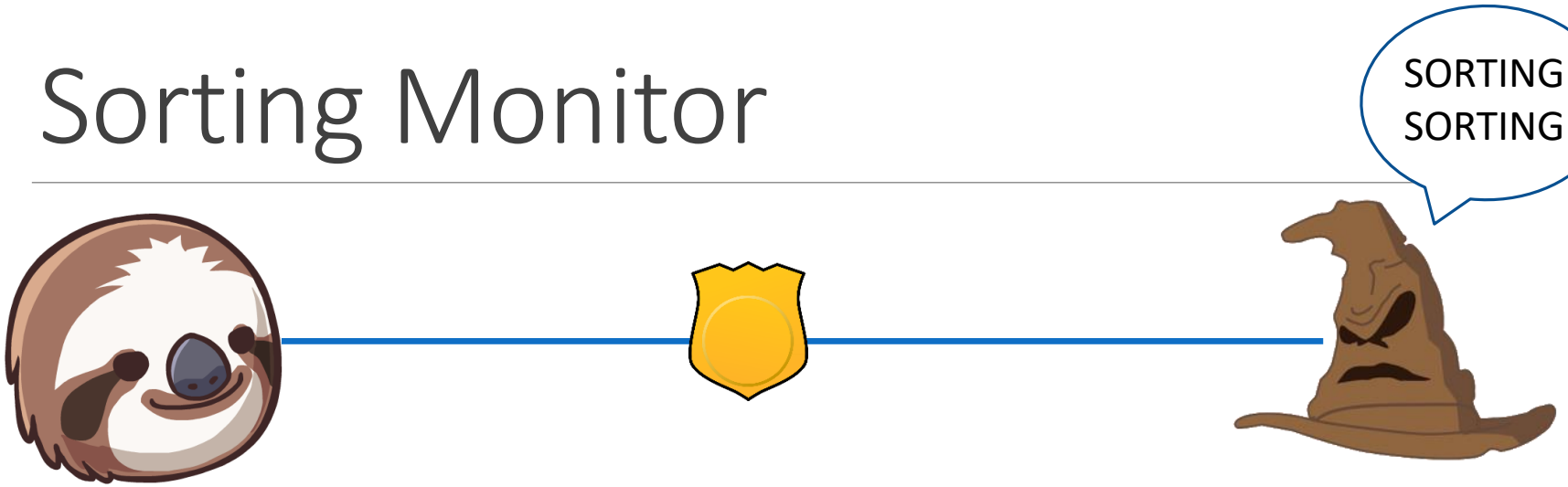
Contributions

- Partial-identity processes that monitor stateful contracts dynamically
- Monitor generation from type refinements
- Method for verifying that monitors are partial identities

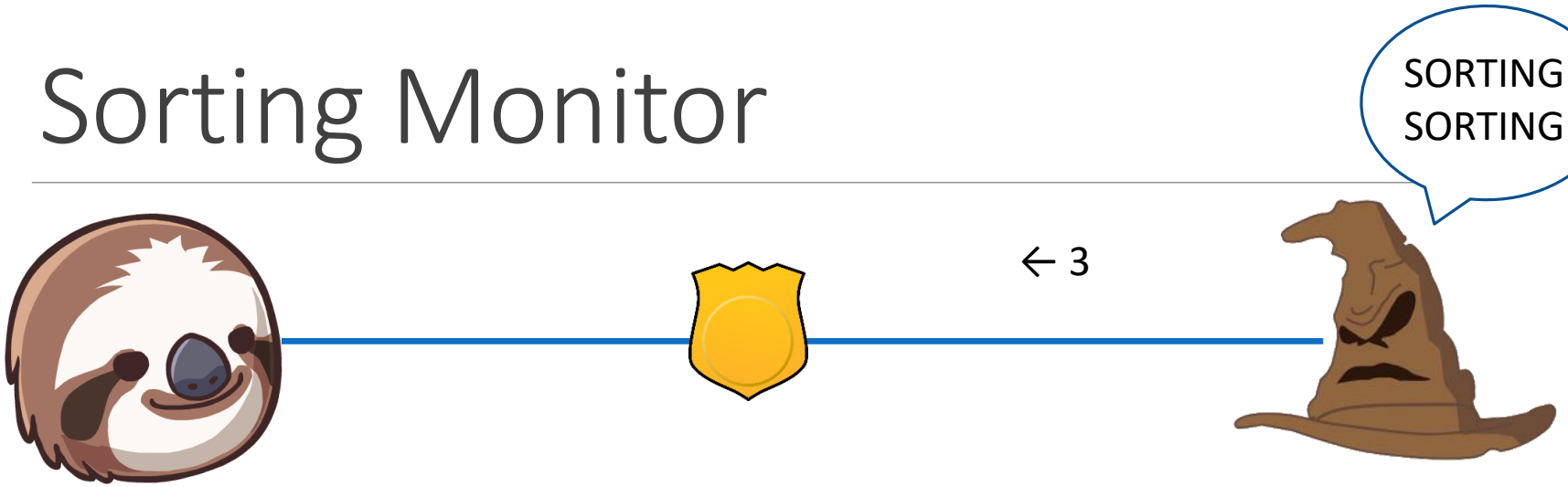
Partial Identity Monitors!

- Processes written in the same language as all other code
- Abort if contract is violated, otherwise are transparent
- Check properties by using internal state

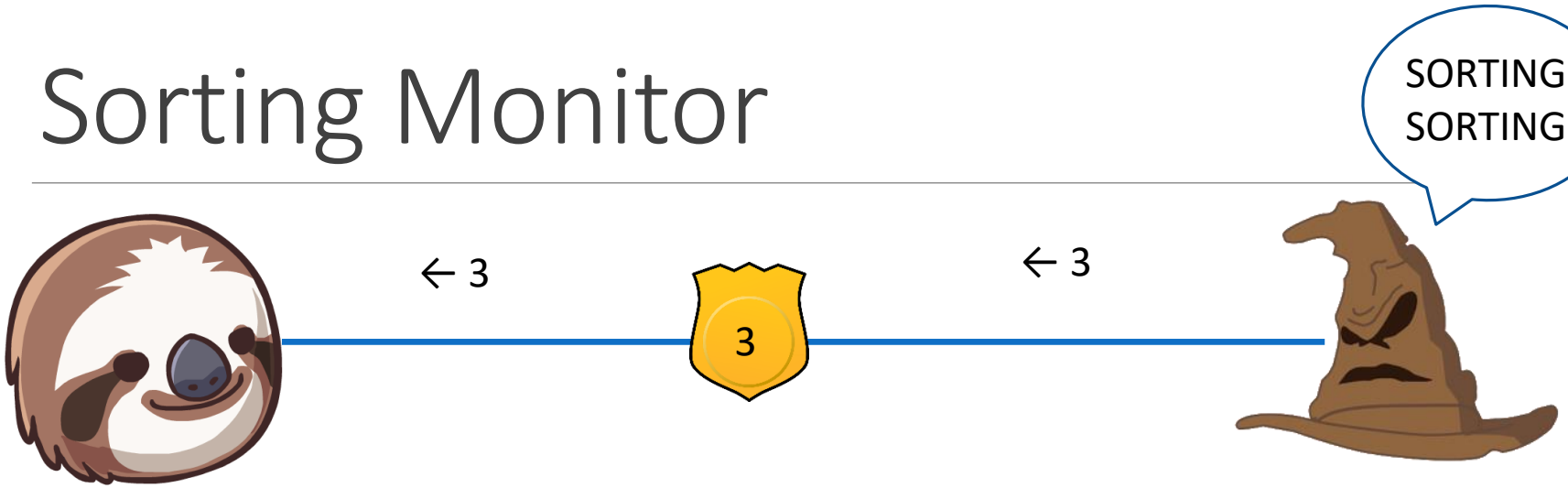
Sorting Monitor



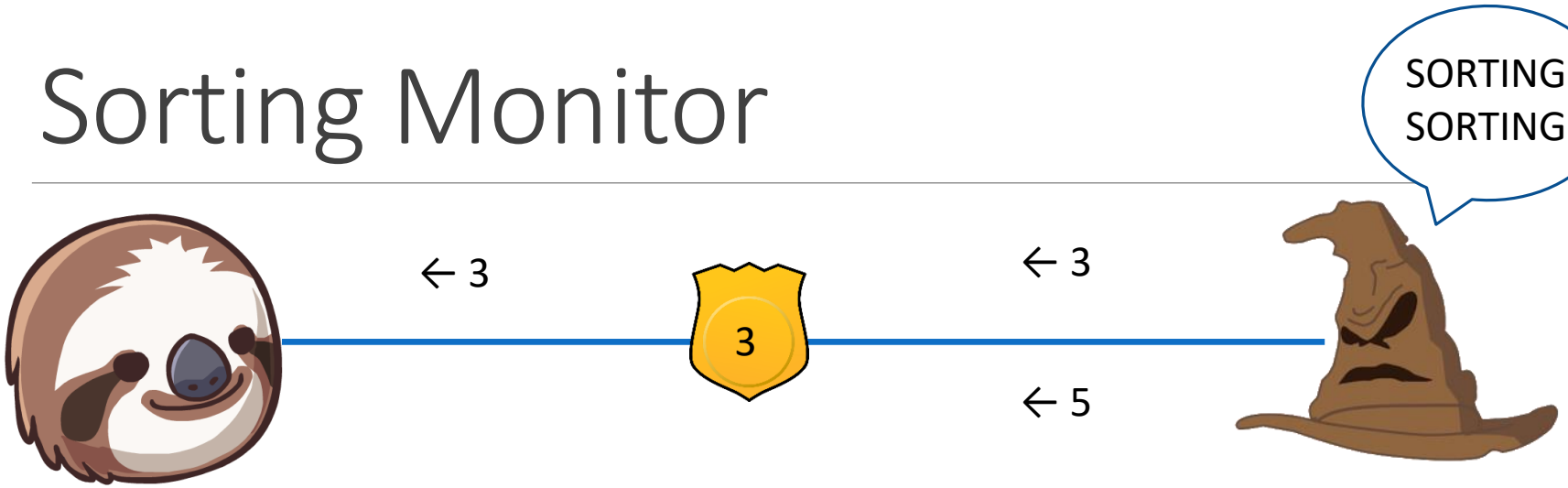
Sorting Monitor



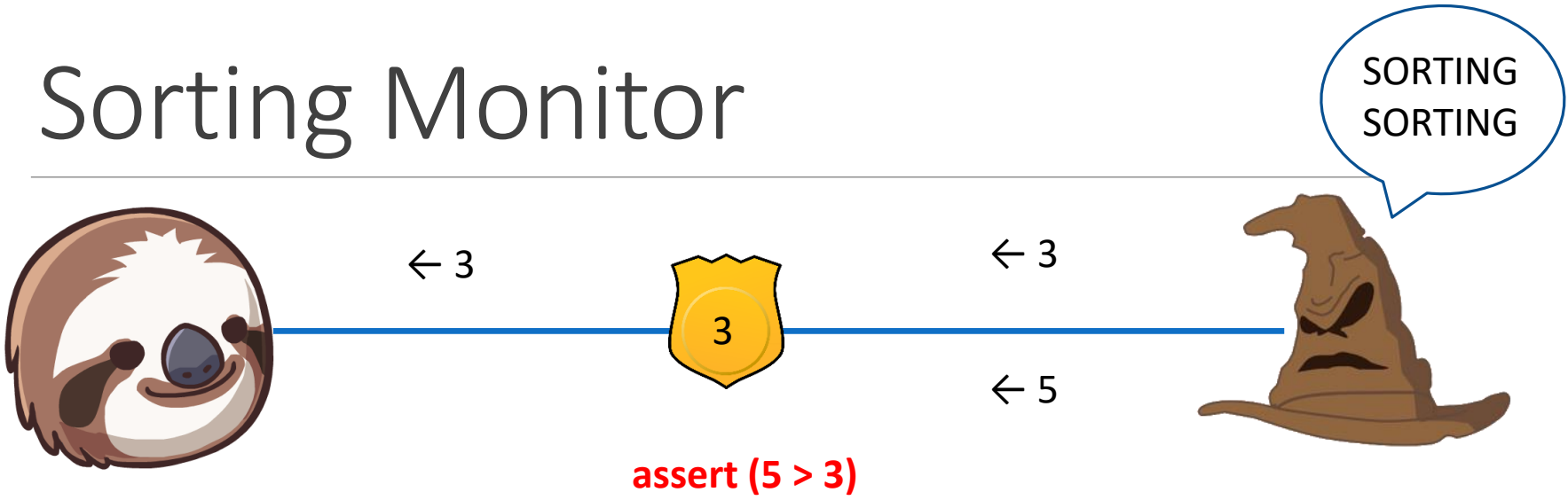
Sorting Monitor



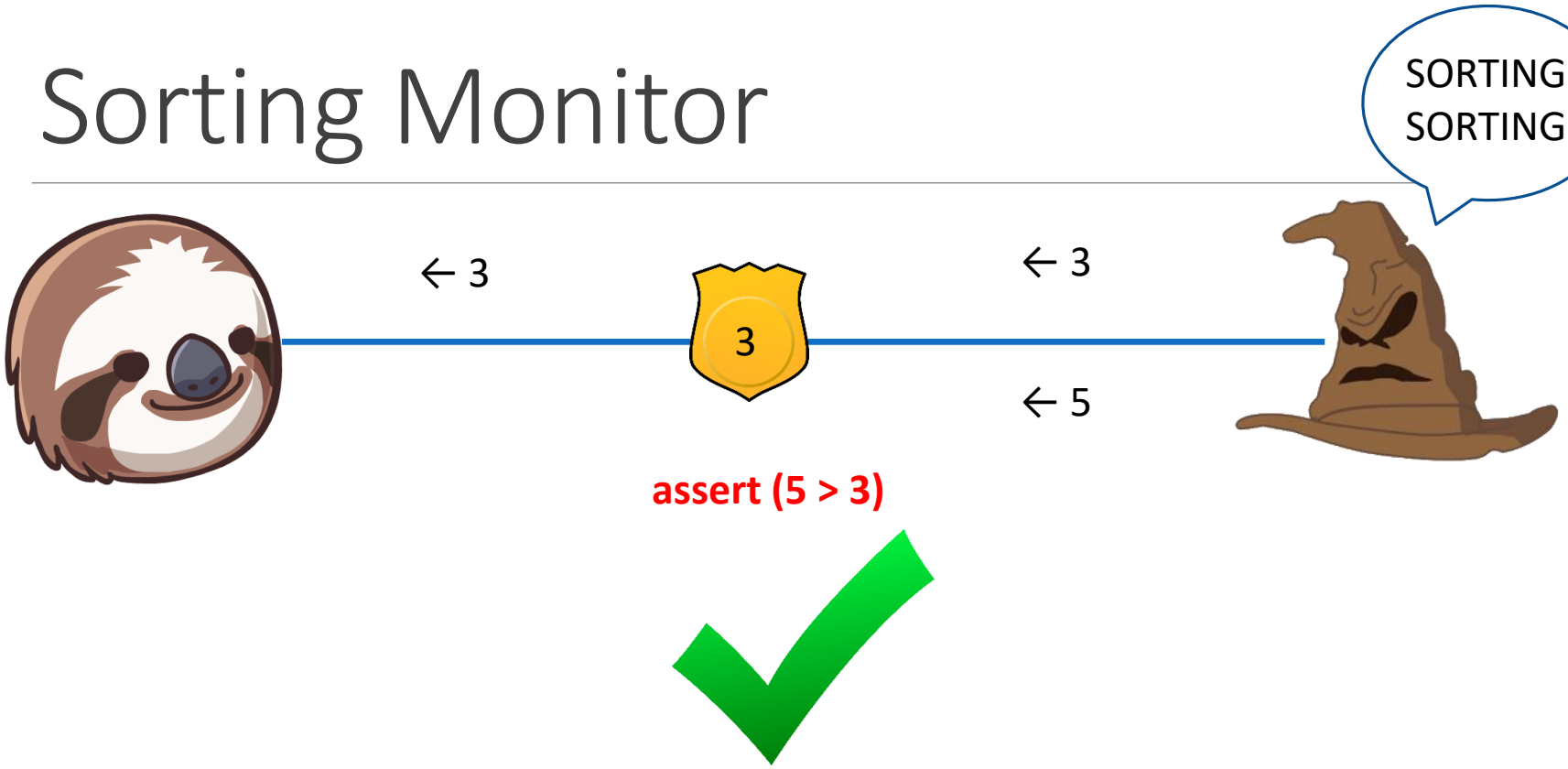
Sorting Monitor



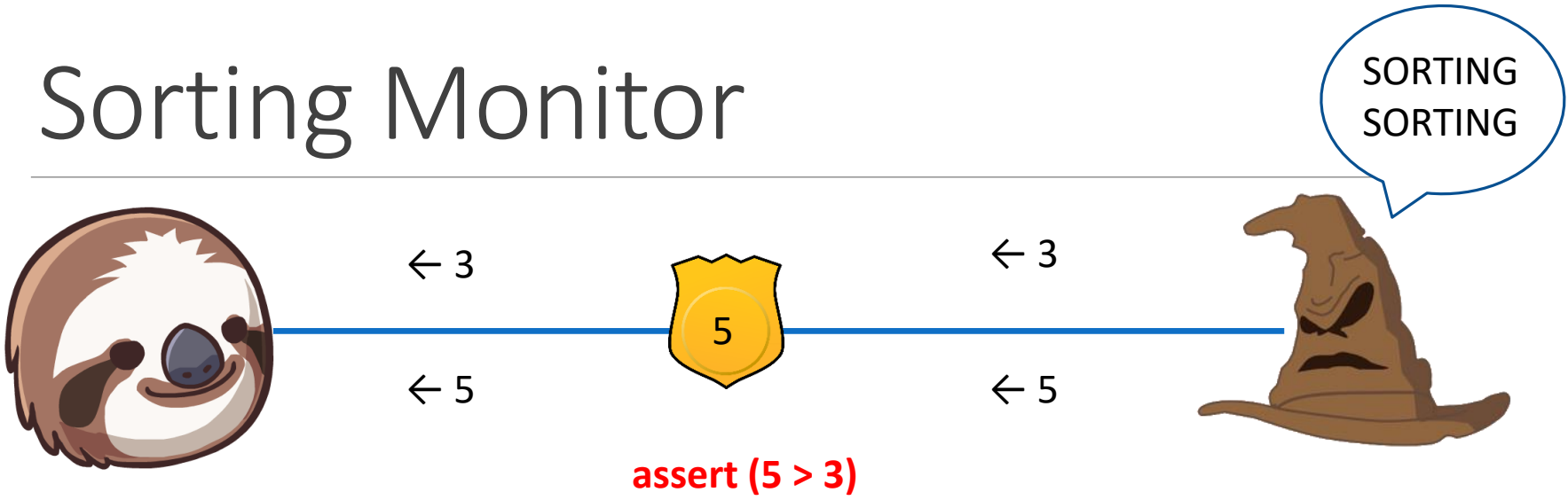
Sorting Monitor



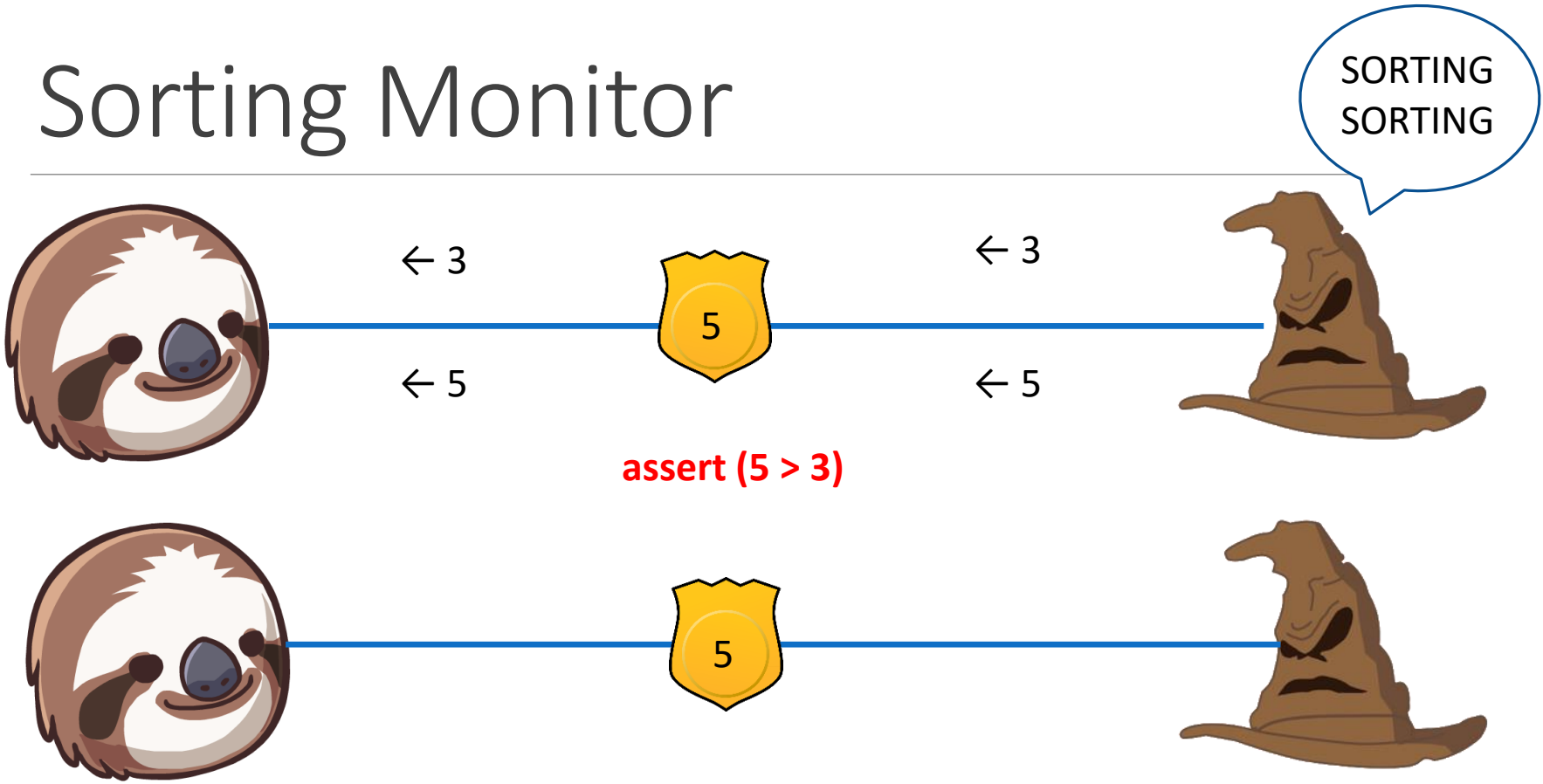
Sorting Monitor



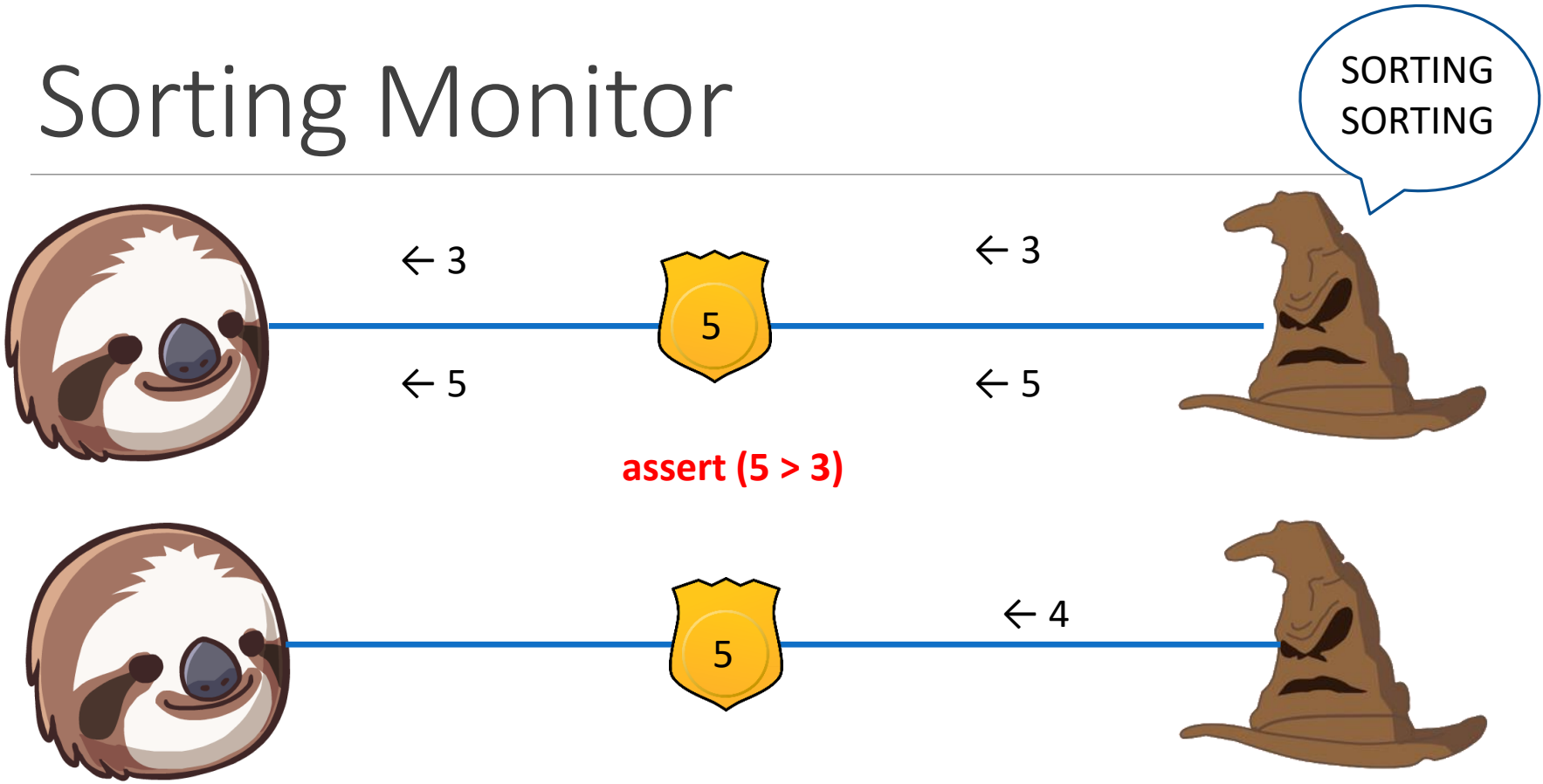
Sorting Monitor



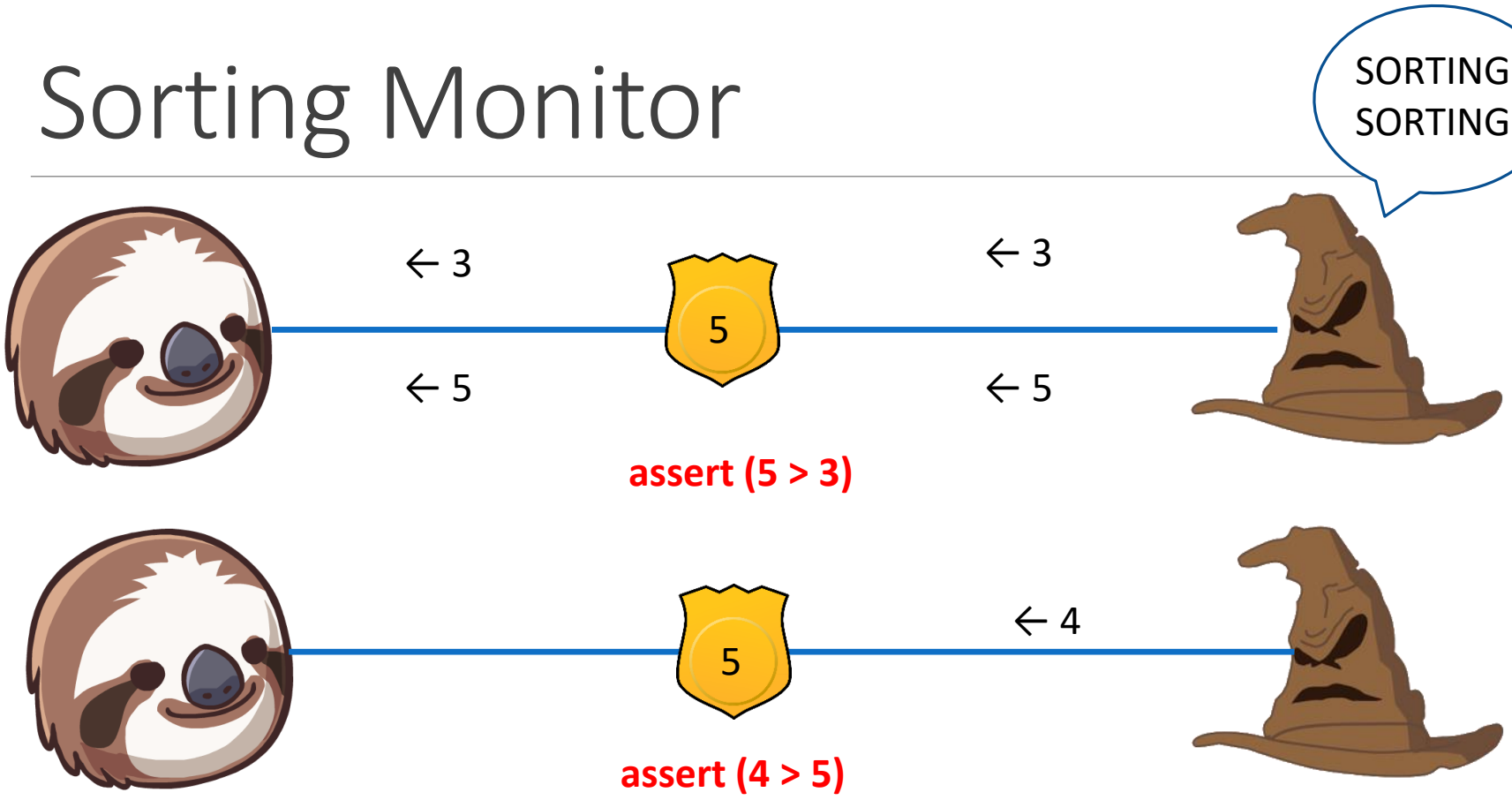
Sorting Monitor



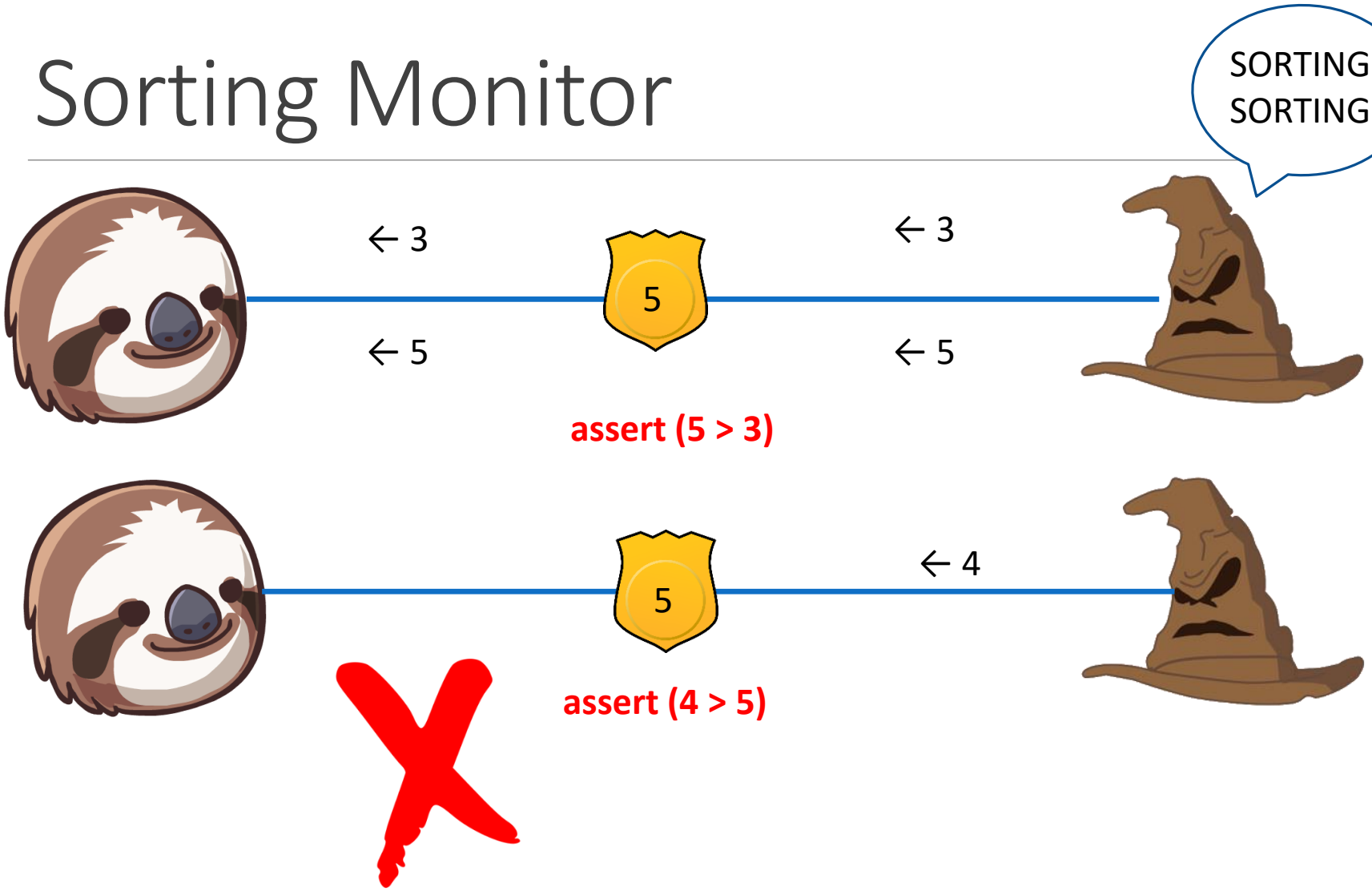
Sorting Monitor



Sorting Monitor



Sorting Monitor



List Identity Process

list_id : list ← list

a ← list_id b =

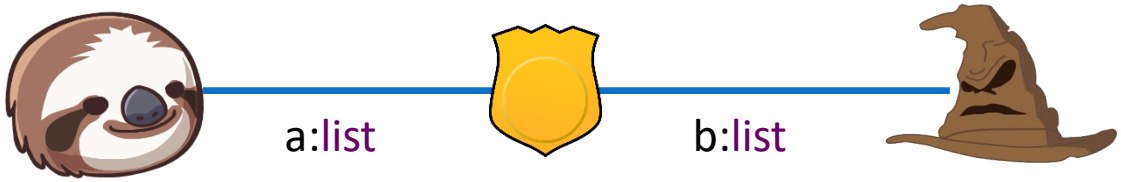
    case b of

        | nil ⇒ a.nil; wait b; close a

        | cons ⇒ x ← recv b;

                 a.cons(x);

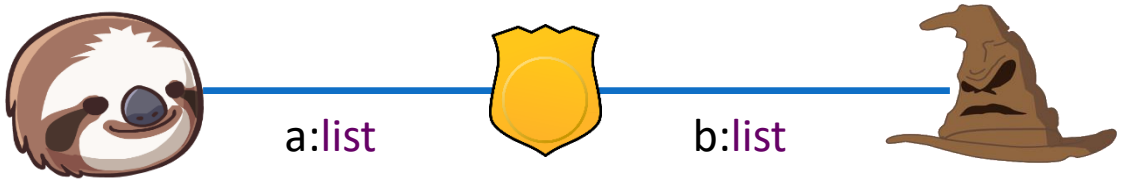
                 a ← list_id b;



$$list = \oplus \{cons: int \wedge list; nil: 1\}$$

List Identity Process

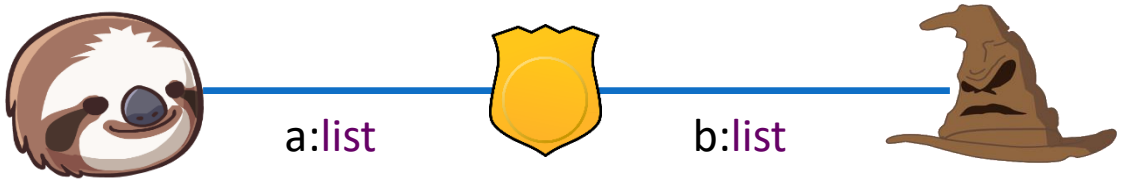
```
list_id : list ← list
a ← list_id  b =
    case b of
        | nil ⇒ a.nil; wait b; close a
        | cons ⇒ x ← recv b;
                 a.cons(x);
                 a ← list_id  b;
```



$$\mathit{list} = \oplus\{\mathit{cons}: \mathit{int} \wedge \mathit{list}; \mathit{nil}: \mathbf{1}\}$$

List Identity Process

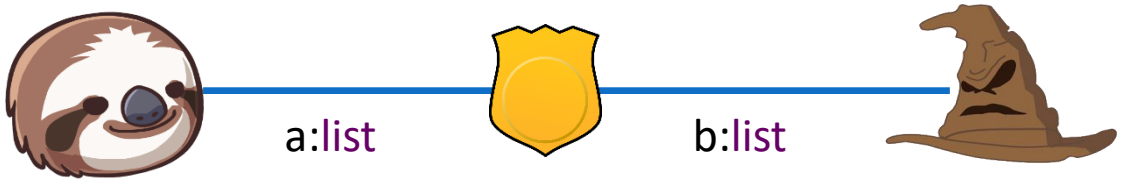
```
list_id : list ← list
a ← list_id  b =
    case b of
        | nil ⇒ a.nil; wait b; close a
        | cons ⇒ x ← recv b;
                 a.cons(x);
                 a ← list_id  b;
```



$$\mathit{list} = \oplus\{\mathit{cons}: \mathit{int} \wedge \mathit{list}; \mathit{nil}: \mathbf{1}\}$$

List Identity Process

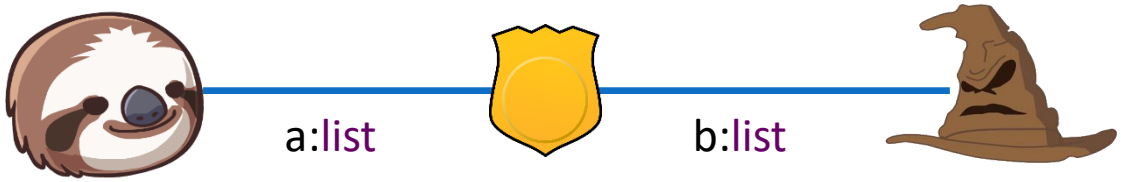
```
list_id : list ← list
a ← list_id  b =
    case b of
        | nil ⇒ a.nil; wait b; close a
        | cons ⇒ x ← recv b;
                 a.cons(x);
                 a ← list_id  b;
```



$$\mathit{list} = \oplus\{\mathit{cons}: \mathit{int} \wedge \mathit{list}; \mathit{nil}: \mathbf{1}\}$$

List Identity Process

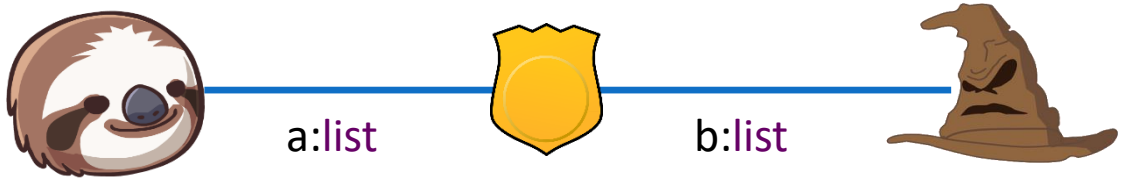
```
list_id : list ← list
a ← list_id  b =
    case b of
        | nil ⇒ a.nil; wait b; close a
        | cons ⇒ x ← recv b;
                 a.cons(x);
                 a ← list_id  b;
```



$$\mathit{list} = \oplus\{\mathit{cons}: \mathit{int} \wedge \mathit{list}; \mathit{nil}: \mathbf{1}\}$$

Sorted List

```
asc_mon: list ← int option, list ;;
a ← asc_mon bound b =
    case b of
      | nil ⇒ a.nil; wait b; close a
      | cons ⇒ x ← recv b;
```



$$list = \oplus\{cons: int \wedge list; nil: 1\}$$

```
        case bound of
         | None  ⇒ a.cons(x);  a ← asc_mon (Some x)  b
         | Some y ⇒ assert (x >= y);
                    a.cons(x); a ← asc_mon (Some x)  b;
```

Other Exciting Contracts

Contract	Monitor State
List of parentheses is well matched	Stack – push for (, pop for)
List of integers corresponds to correctly serialized binary tree	Stack – push for node, pop for leaf
Sorting procedure permutes elements of a list	Sum of hash values of each element
Doubling process mapped over a list of integers is monotonic	Input integer
Refinements on integers	None
Factoring procedures outputs integers that multiply to input	Input integer

Checking Refinements

- Encode refinements as type casts
- Can generate a partial identity monitor to check the refinement

$$\{A\}_a \Leftarrow \{B\}_b$$

Integer Refinement Translation

$$\{n\colon int \mid n > 0 \wedge A\}_a \Leftarrow \{n\colon int \wedge B\}_b$$

Integer Refinement Translation

$$\{n\colon int \mid n > 0 \wedge A\}_a \Leftarrow \{n\colon int \wedge B\}_b$$

x ← recv b;
send a x;

Integer Refinement Translation

$$\{n\colon int \mid n > 0 \wedge A\}_a \Leftarrow \{n\colon int \wedge B\}_b$$

x ← recv b;

assert (x > 0)

send a x;

Integer Refinement Translation

$$\{n\colon int \mid n > 0 \wedge A\}_a \Leftarrow \{n\colon int \wedge B\}_b$$

x ← recv b;

assert (x > 0)

send a x;

$$\{A\}_a \Leftarrow \{B\}_b$$

Label Refinement Translation

$$\oplus\{\boldsymbol{cons}: \boldsymbol{int} \wedge \boldsymbol{list}\}_{\boldsymbol{a}} \Leftarrow \oplus\{\boldsymbol{cons}: \boldsymbol{int} \wedge \boldsymbol{list}; \boldsymbol{nil}: \mathbf{1}\}_{\boldsymbol{b}}$$

Label Refinement Translation

$$\oplus\{\boldsymbol{cons}: \boldsymbol{int} \wedge \boldsymbol{list}\}_{\boldsymbol{a}} \Leftarrow \oplus\{\boldsymbol{cons}: \boldsymbol{int} \wedge \boldsymbol{list}; \boldsymbol{nil}: \mathbf{1}\}_{\boldsymbol{b}}$$

```
case b of
| nil =>
| cons =>
```

Label Refinement Translation

$$\oplus\{\boldsymbol{cons}: \boldsymbol{int} \wedge \boldsymbol{list}\}_{\boldsymbol{a}} \Leftarrow \oplus\{\boldsymbol{cons}: \boldsymbol{int} \wedge \boldsymbol{list}; \boldsymbol{nil}: \mathbf{1}\}_{\boldsymbol{b}}$$

```
case b of
| nil => assert (false)
| cons => a.cons;
```

Label Refinement Translation

$$\oplus\{\boldsymbol{cons}: \boldsymbol{int} \wedge \boldsymbol{list}\}_{\boldsymbol{a}} \Leftarrow \oplus\{\boldsymbol{cons}: \boldsymbol{int} \wedge \boldsymbol{list}; \boldsymbol{nil}: \mathbf{1}\}_{\boldsymbol{b}}$$

```
case b of
| nil => assert (false)
| cons => a.cons;
```

$$\{\boldsymbol{int} \wedge \boldsymbol{list}\}_{\boldsymbol{a}} \Leftarrow \{\boldsymbol{int} \wedge \boldsymbol{list}\}_{\boldsymbol{b}}$$

Translation is Transparent

- Prove that generated monitors are partial identity processes

$$\mathrm{a}:\mathrm{A} \leftarrow \mathrm{b}:\mathrm{B}$$

$$\approx$$

$$\mathrm{a} \leftarrow \boxed{\{\boldsymbol{A}\}_{\boldsymbol{a}} \Leftarrow \{\boldsymbol{B}\}_{\boldsymbol{b}}} \leftarrow \mathrm{b}$$

Redundant Monitor

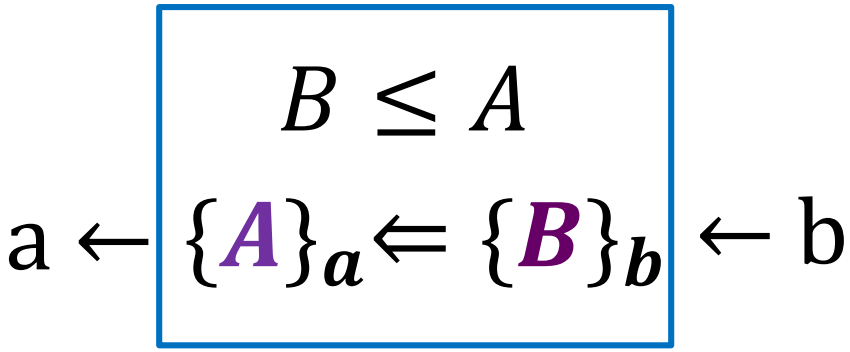
$$\oplus\{\boldsymbol{cons}: \boldsymbol{int} \wedge \boldsymbol{list}; \boldsymbol{nil}: \mathbf{1}\}_{\boldsymbol{a}} \Leftarrow \oplus\{\boldsymbol{cons}: \boldsymbol{int} \wedge \boldsymbol{list}\}_{\boldsymbol{b}}$$

```
case b of
| cons => a.cons;
```

$$\{\boldsymbol{int} \wedge \boldsymbol{list}\}_{\boldsymbol{a}} \Leftarrow \{\boldsymbol{int} \wedge \boldsymbol{list}\}_{\boldsymbol{b}}$$

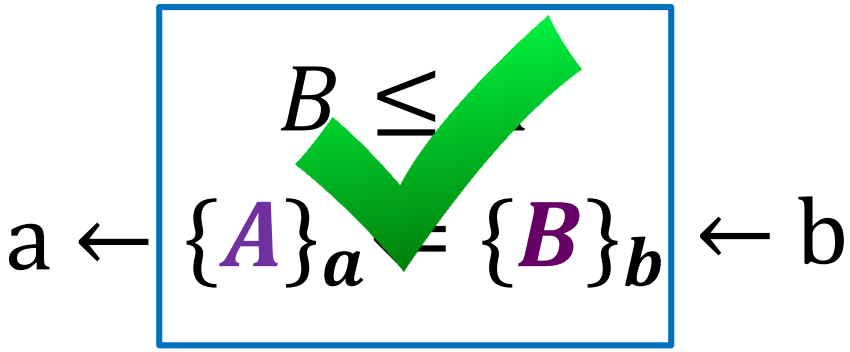
Safety Theorem

- Prove that subtype casts do not cause assertion failures

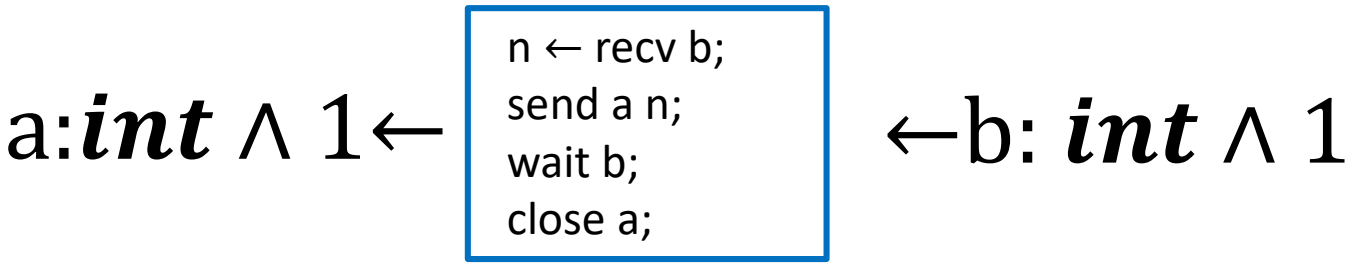


Safety Theorem

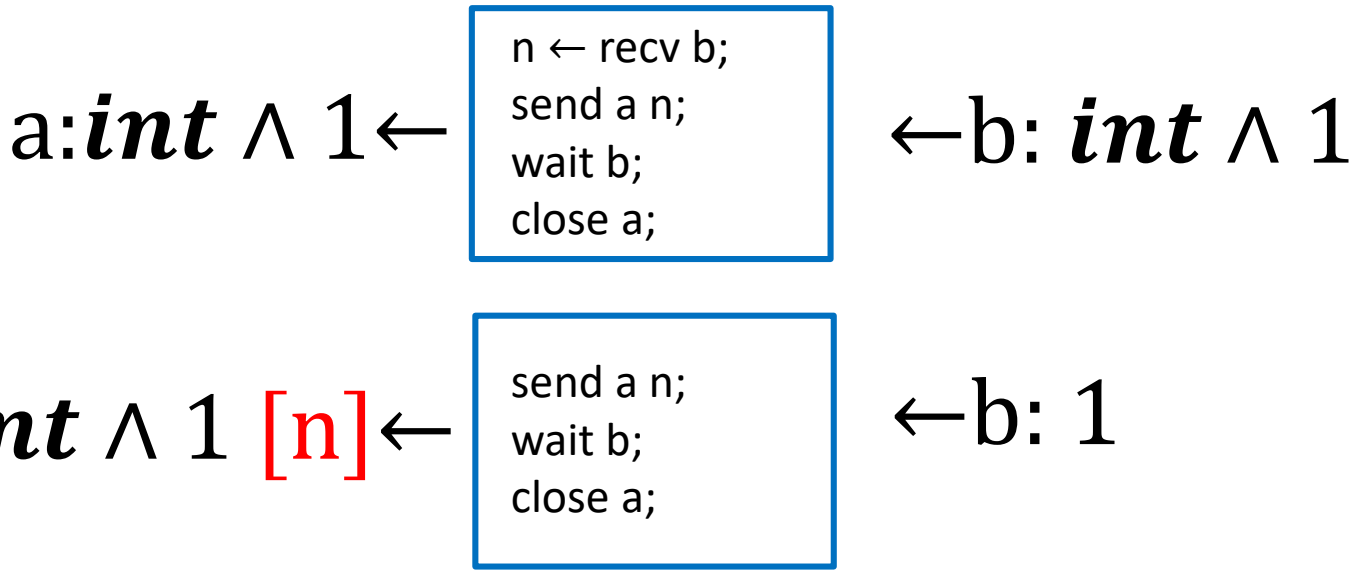
- Prove that subtype casts do not cause assertion failures



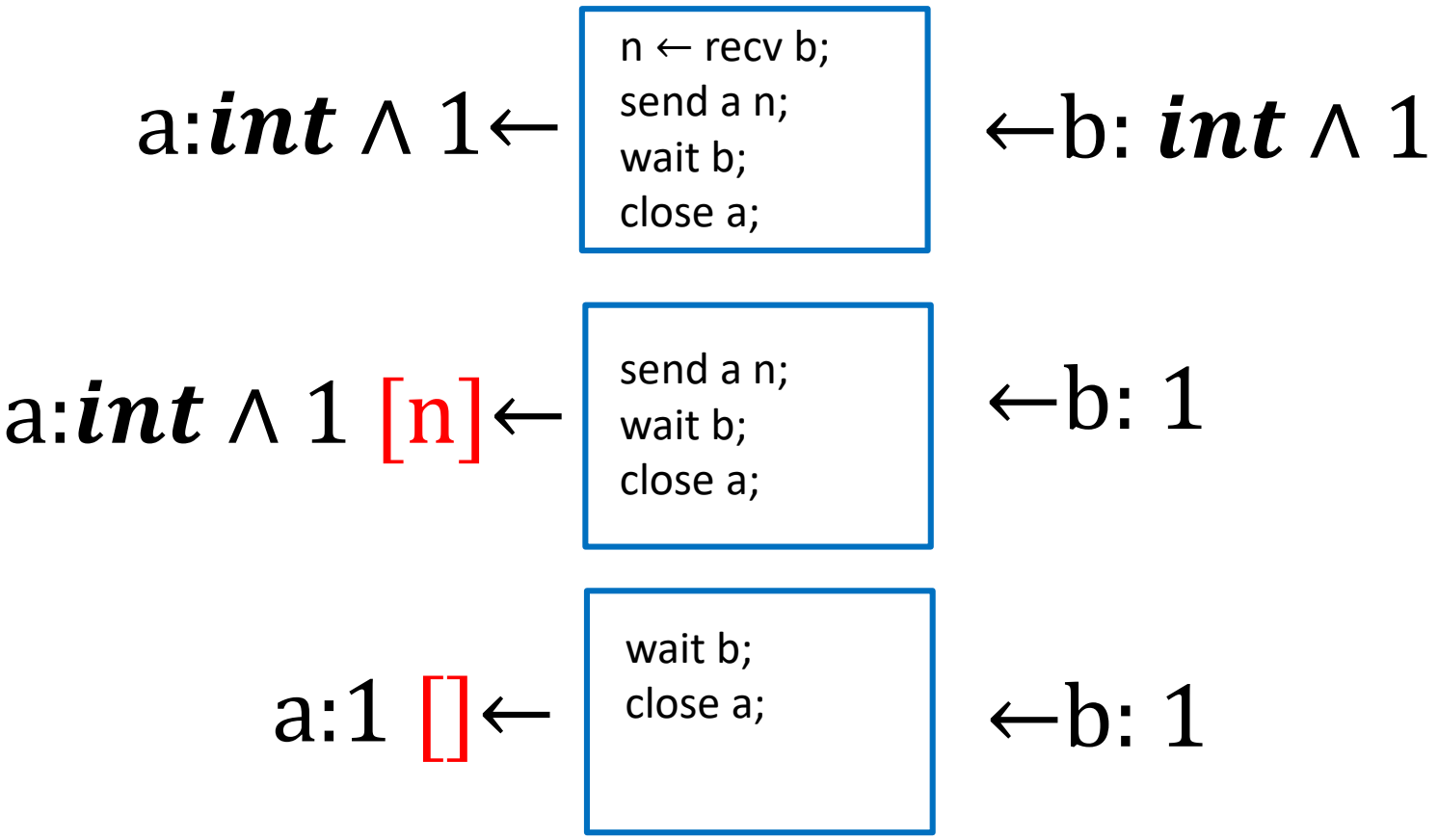
Partial Identity Rules



Partial Identity Rules



Partial Identity Rules



Partial Identity Criterion

- Complications occur when monitors spawn other monitors (higher order case)
- Construct a bisimulation between monitoring processes and identity processes showing that every observable message is the same

Future Work

- Even more exciting contracts!
  - Contracts which require distributed state (ghost messages), ie permutation checking
  - Noninterference/information flow contracts

Related Work

- Recently:
  - Melgratti & Padovani (ICFP 2017)
  - Waye et al (ICFP 2017)
- Contracts: Findler & Felleisen (2002), Dimoulas et al (2011,2012), Disney et al (2012)

Takeaway

Partial identity monitors provide a mechanism to express complex stateful contracts in the same language as the target code