Thesis Defense: Session-Typed Concurrent Contracts

Hannah Gommerstadt

Frank Pfenning (co-chair)
Limin Jia (co-chair)
Jan Hofmann
Bernardo Toninho (Universidade Nova de Lisboa)
Adrian Francalanza (University of Malta)

Concurrent Contracts

Enforceable communication agreement between multiple parties collaborating on a concurrent computation.

Why Monitor Contracts?

- Concurrent multi-process systems are everywhere
- When a single process deviates from its prescribed role in the computation, the impact of the misbehavior propagates
- The goal is to detect and contain process misbehavior

Why Dynamic Monitoring?

- Static checking requires running checker on all processes which may be written in different languages
- Unrealistic to assume that will have access to whole computing base

Thesis Statement

Session-typed monitors give rise to novel techniques for dynamically monitoring expressive classes of concurrent contracts and provide strong theoretical guarantees.

Concurrent Contracts

Session types express communication contracts between concurrent processes.

Honda [1993]

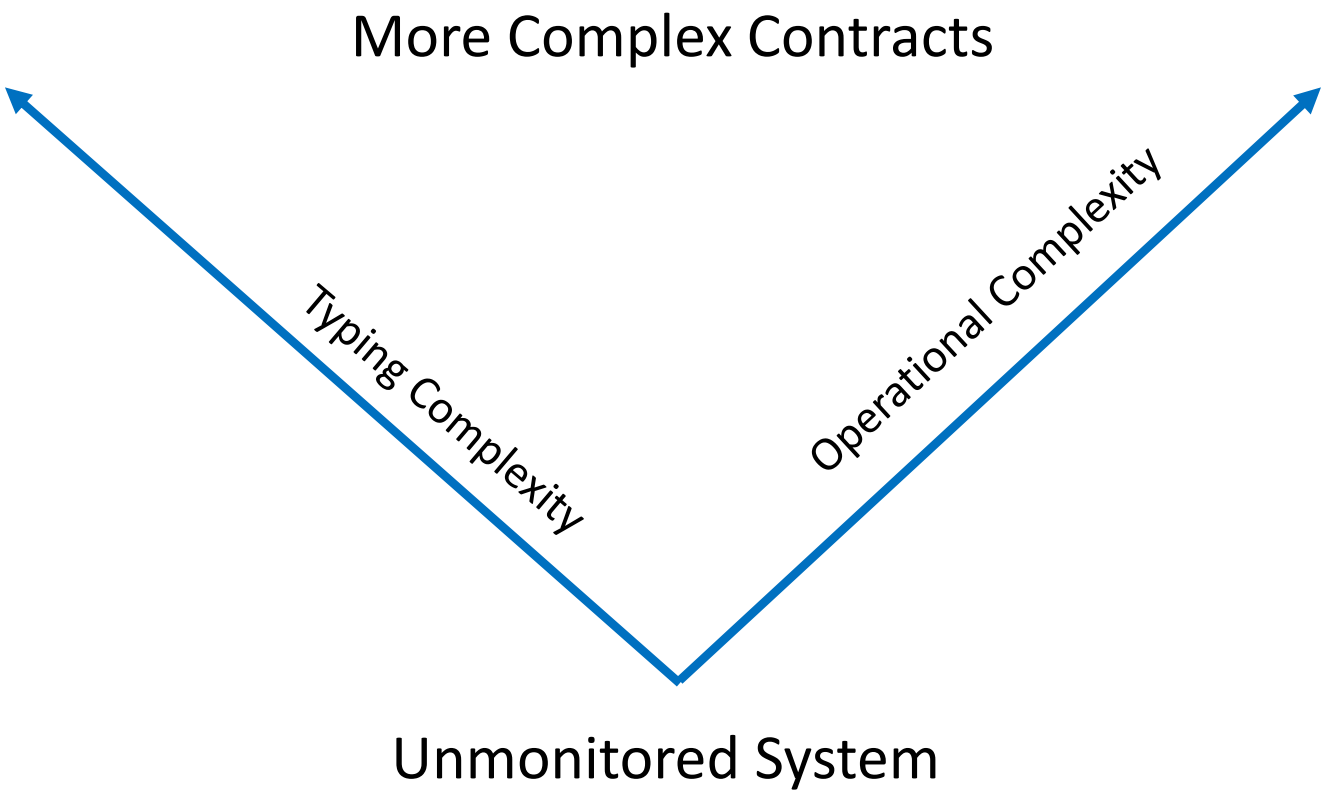
Contracts Related Work

- Concurrent/Distributed Contracts:
  - Melgratti & Padovani (ICFP 2017)
  - Waye et al (ICFP 2017)
- Higher-Order Functional Contracts: Findler & Felleisen (2002), Dimoulas et al (2011,2012), Disney et al (2012)

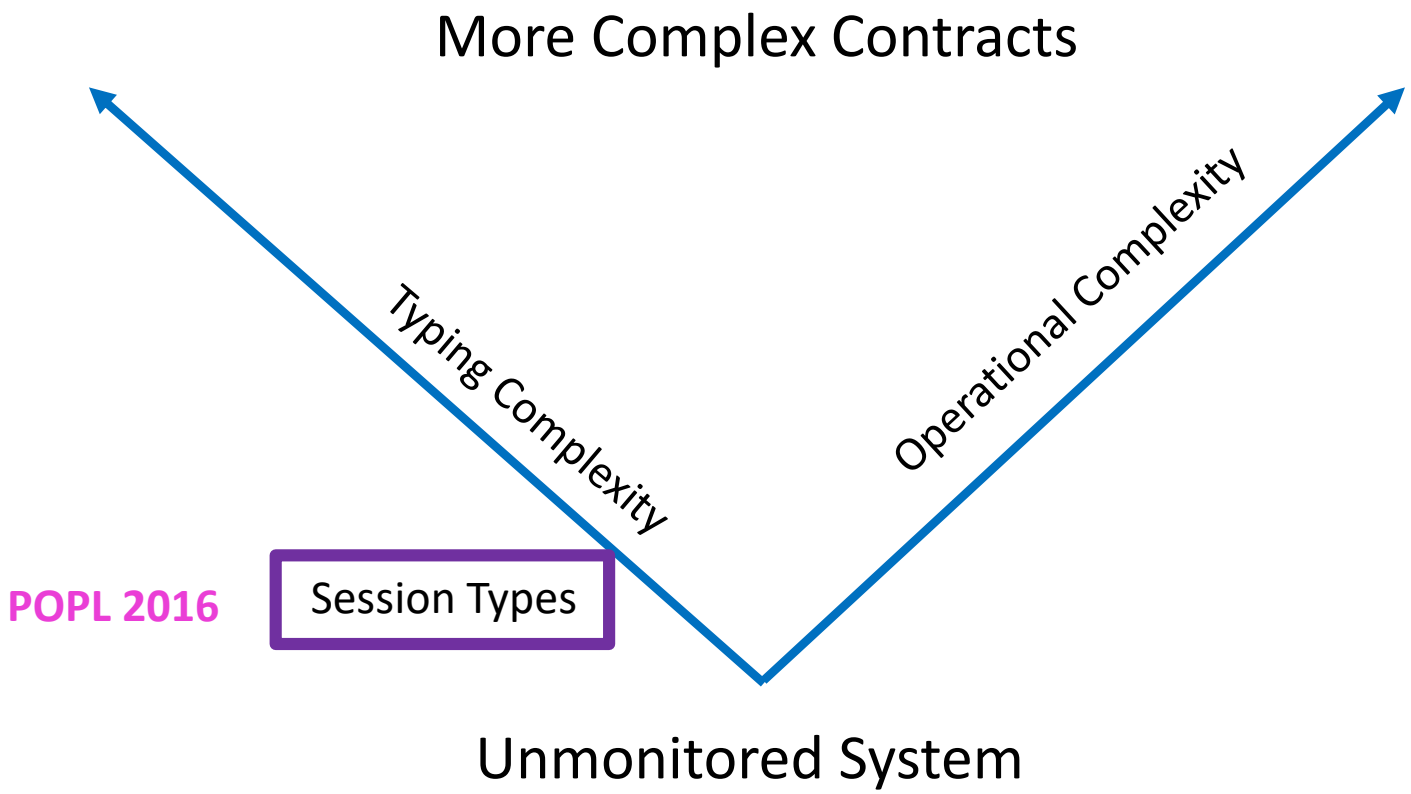
Contract Classes

Unmonitored System

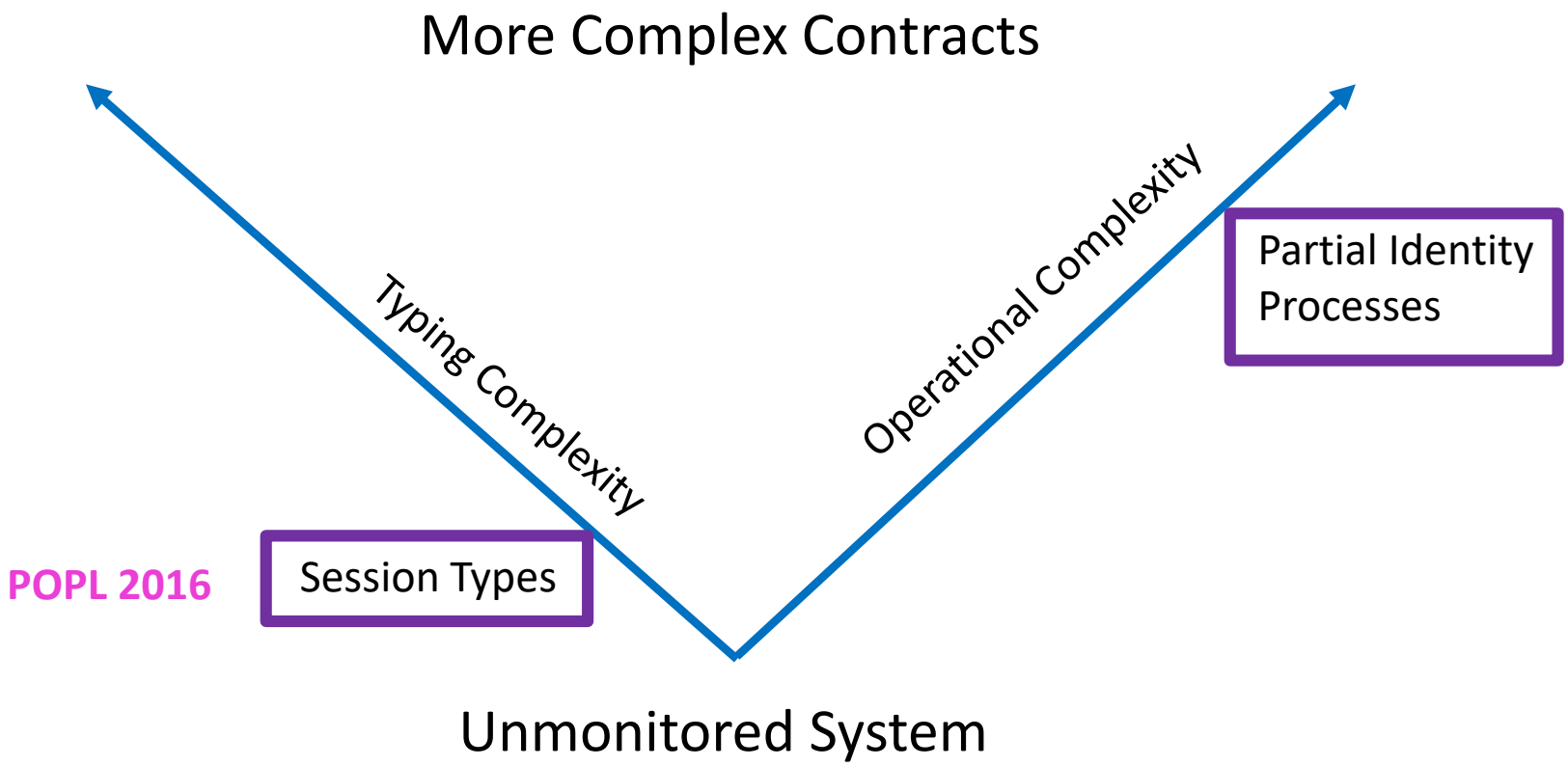
Contract Classes



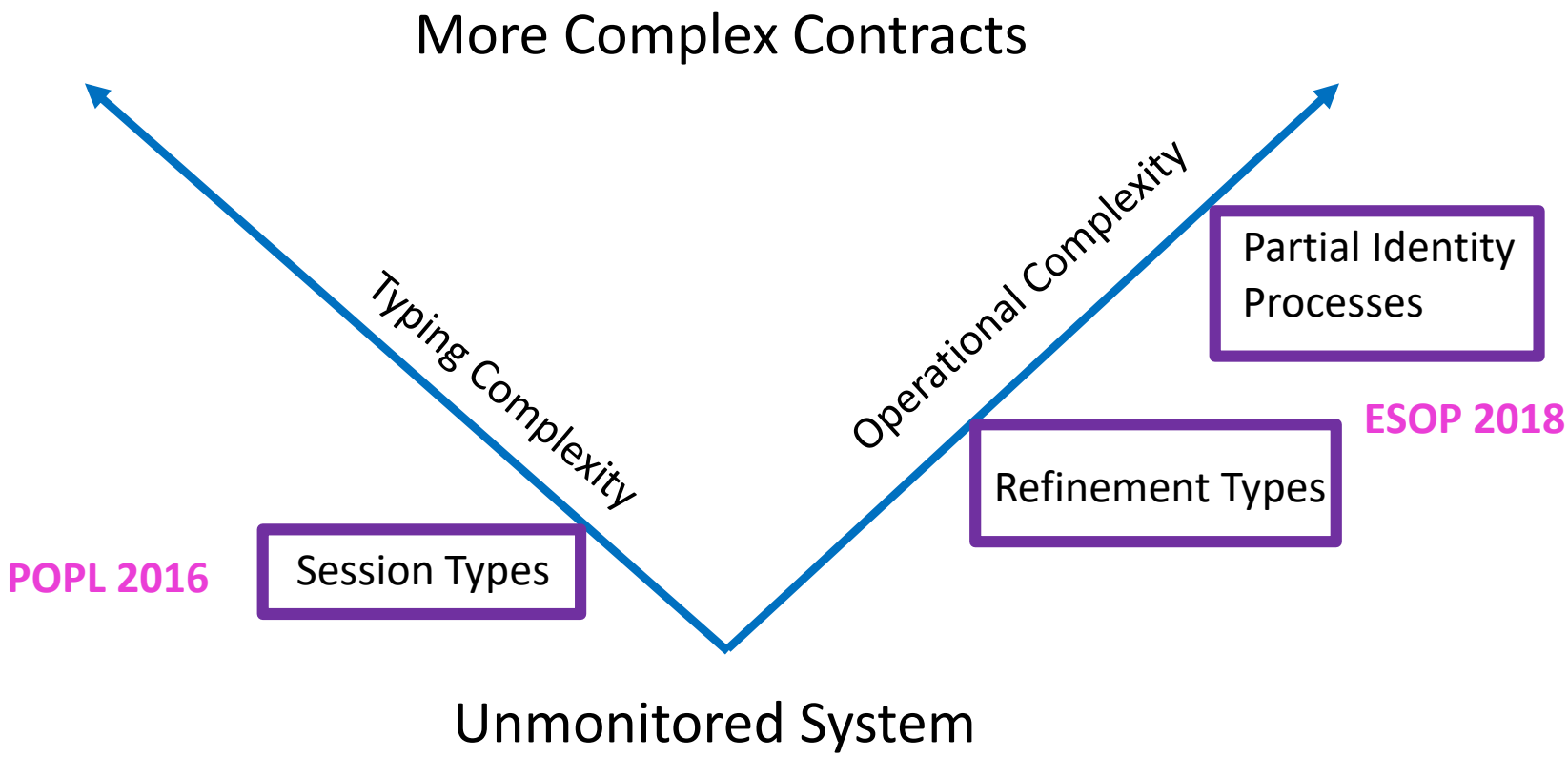
Contract Classes



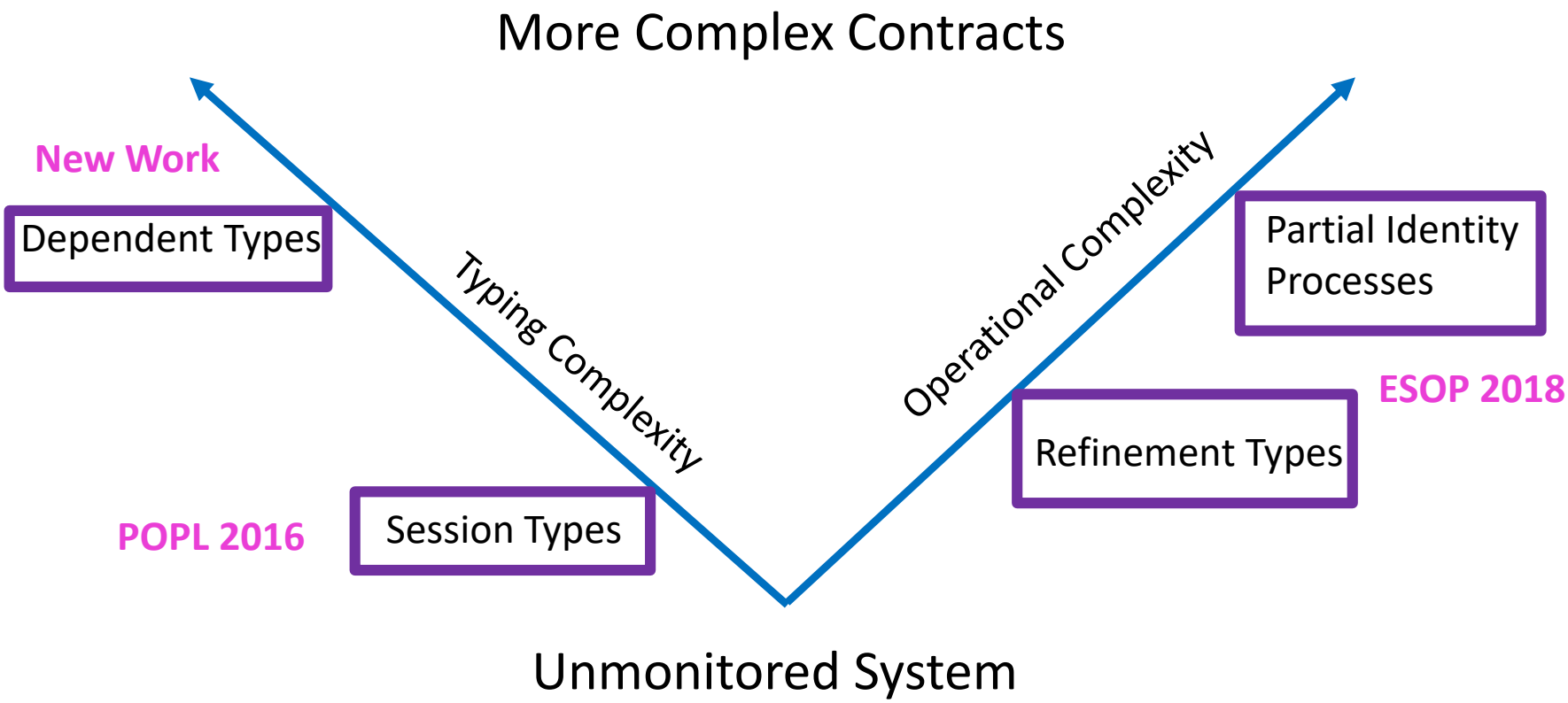
Contract Classes



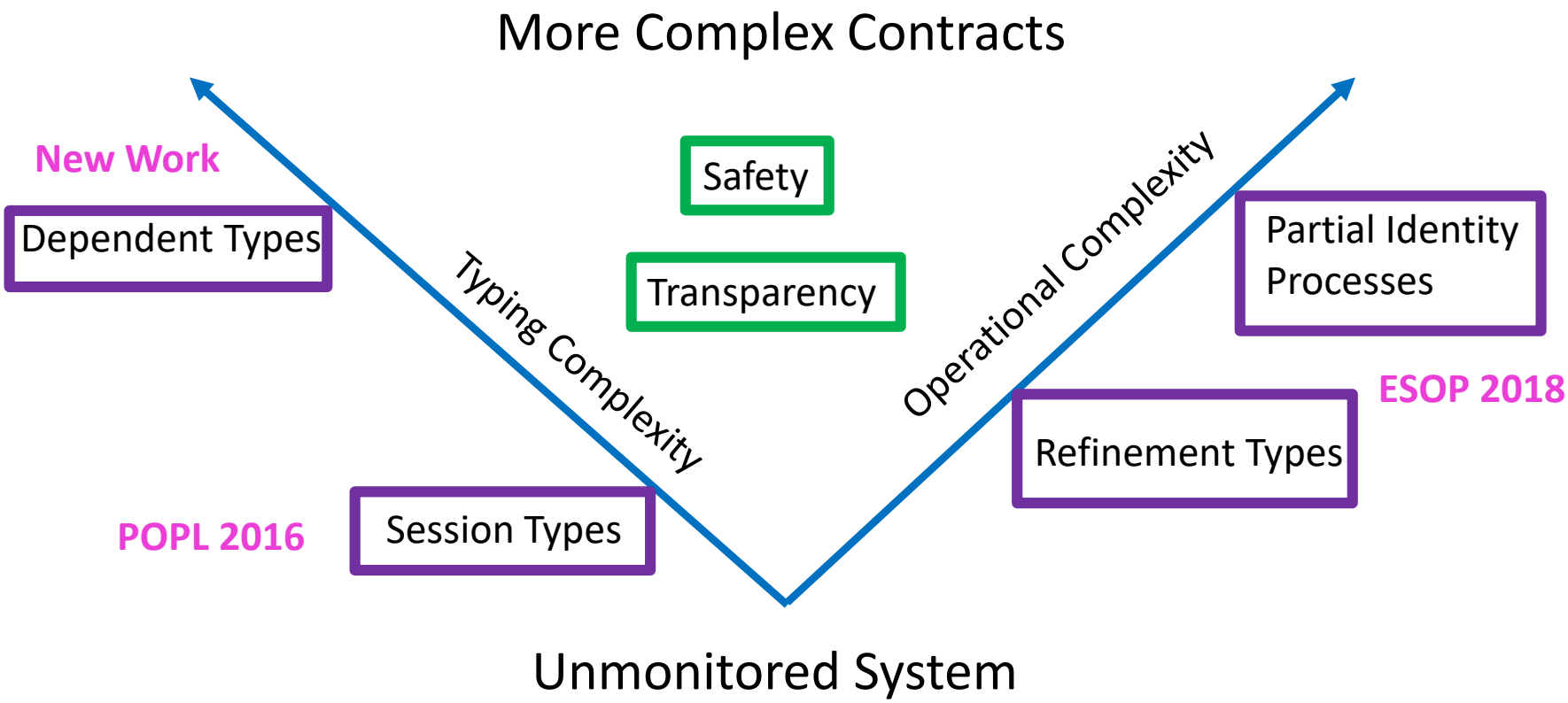
Contract Classes



Contract Classes



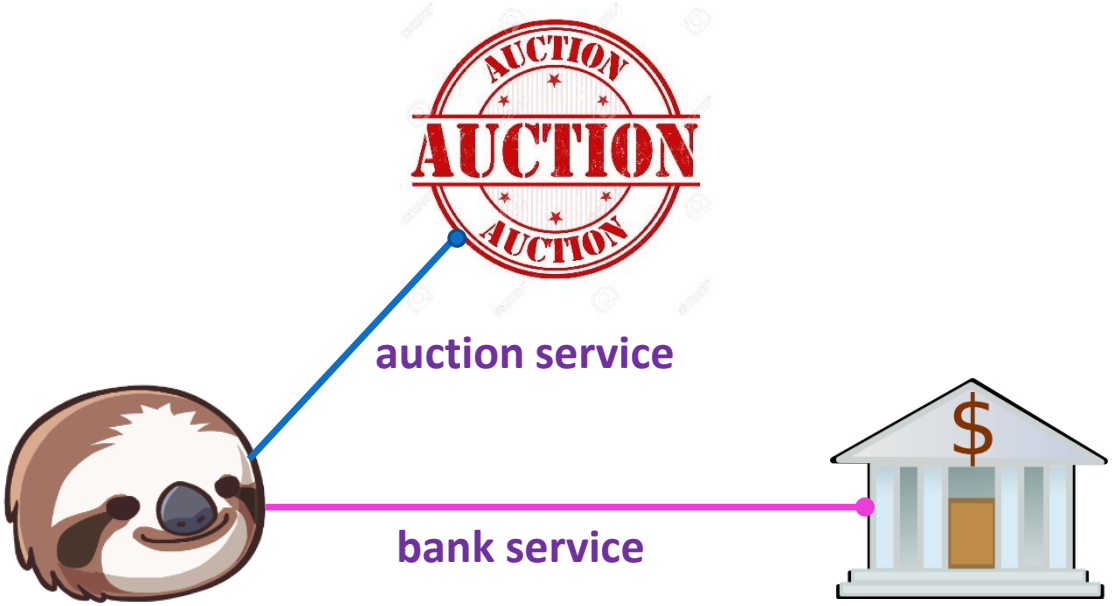
Contract Classes



Example: Auction Protocol

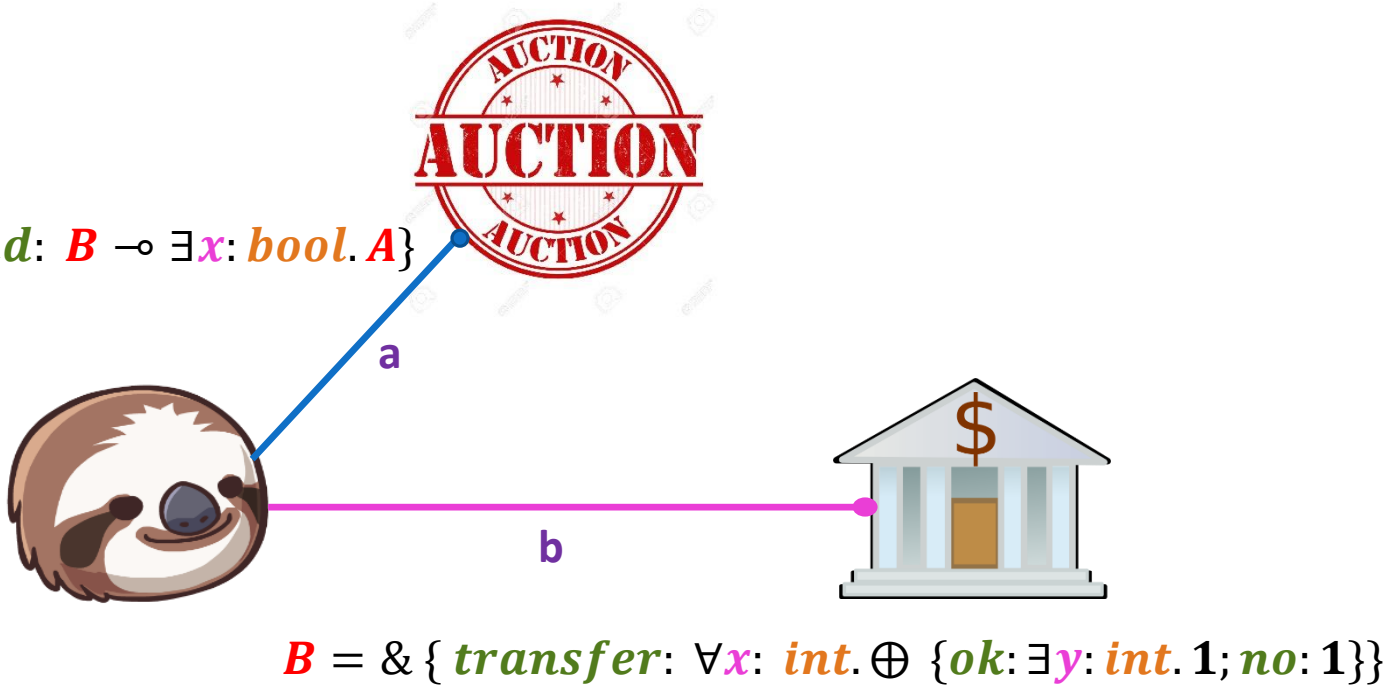
- User starts a bid with the auction
- User gives auction direct access to her bank account
- Auction requests transfer from bank
- Bank sends receipt to auction
- Auction acknowledges success to user

Communication Channels

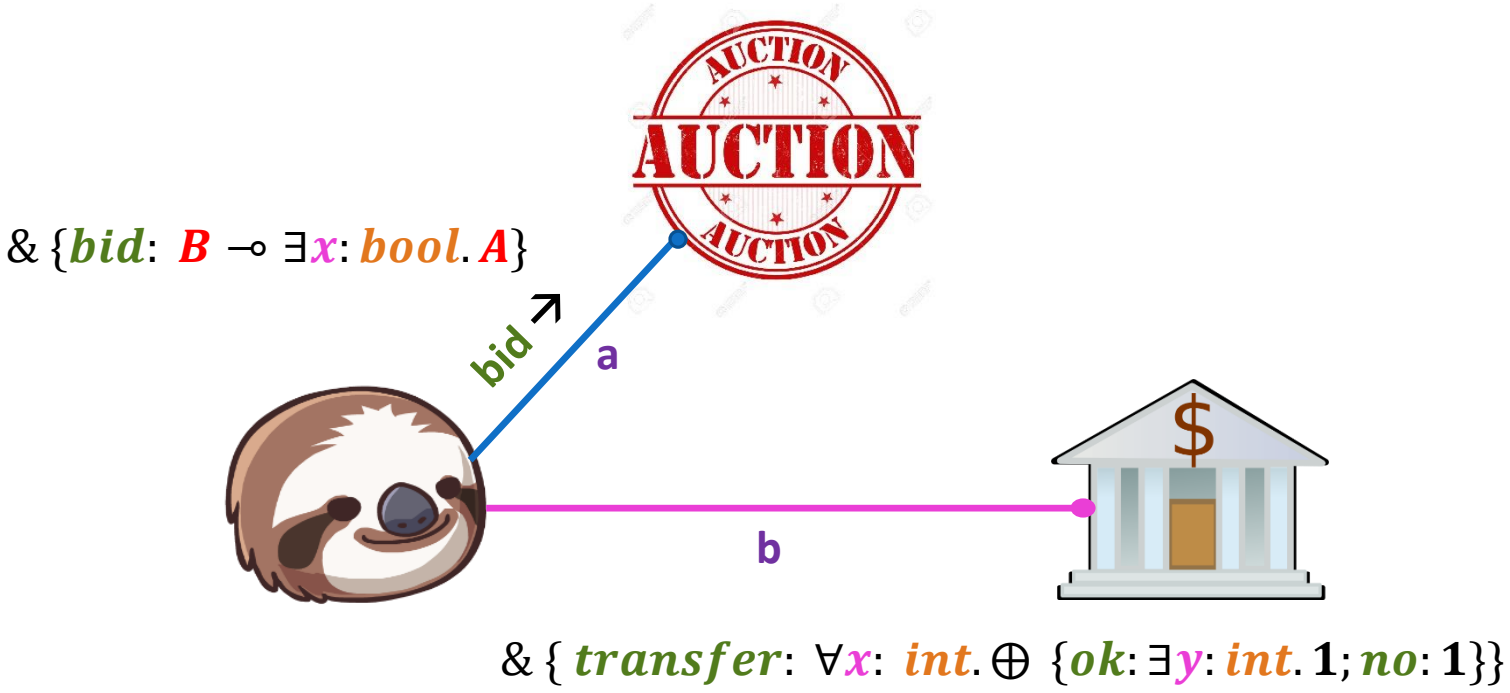


process provides a service along one channel

Initial Session Types

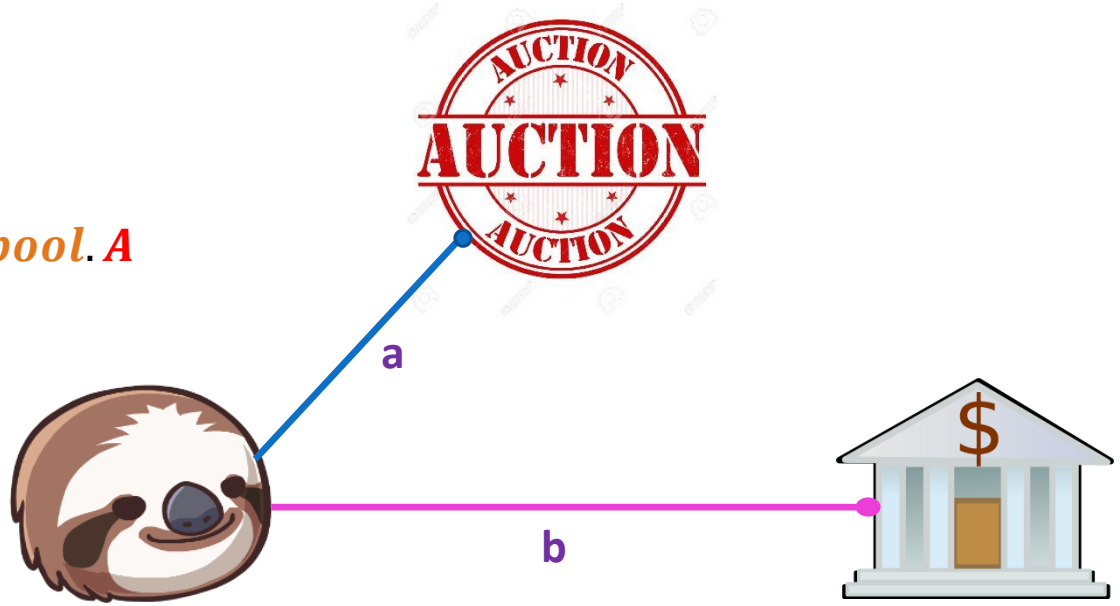


Start Bid


& {bid: B ⊸ ∃x: bool. A}
AUCTION
AUCTION
AUCTION
bid
a
b
$
& { transfer: ∀x: int. ⊕ {ok: ∃y: int. 1; no: 1}}

Type Changes

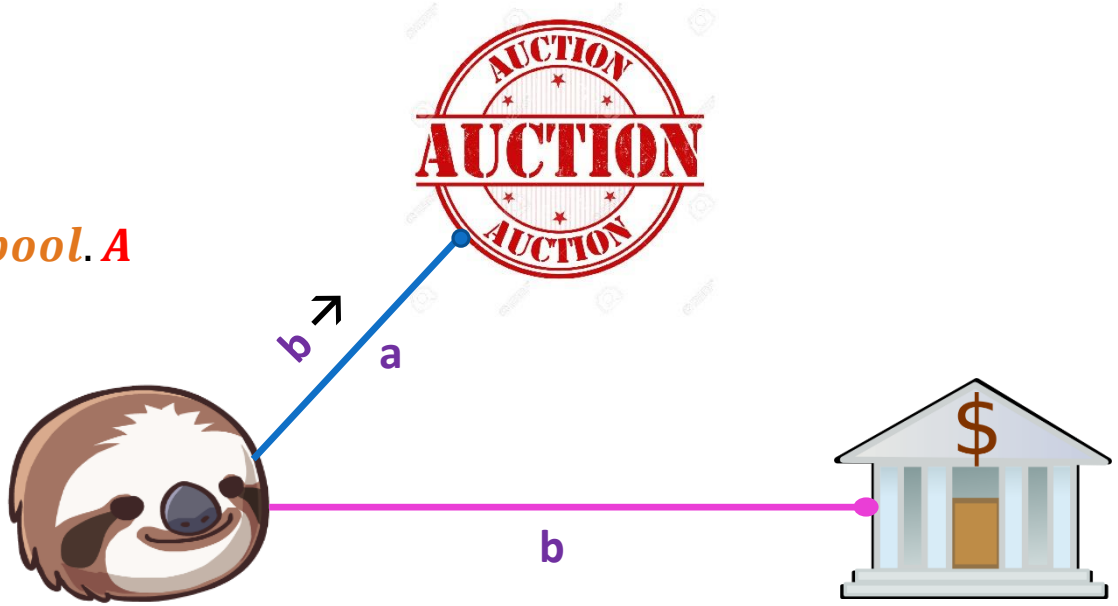
$B \multimap \exists x\colon bool.\, A$



$\&\{\, transfer\colon \forall x\colon int.\, \oplus\, \{ok\colon \exists y\colon int.\, \mathbf{1}; no\colon \mathbf{1}\}\}$

Send Bank Connection

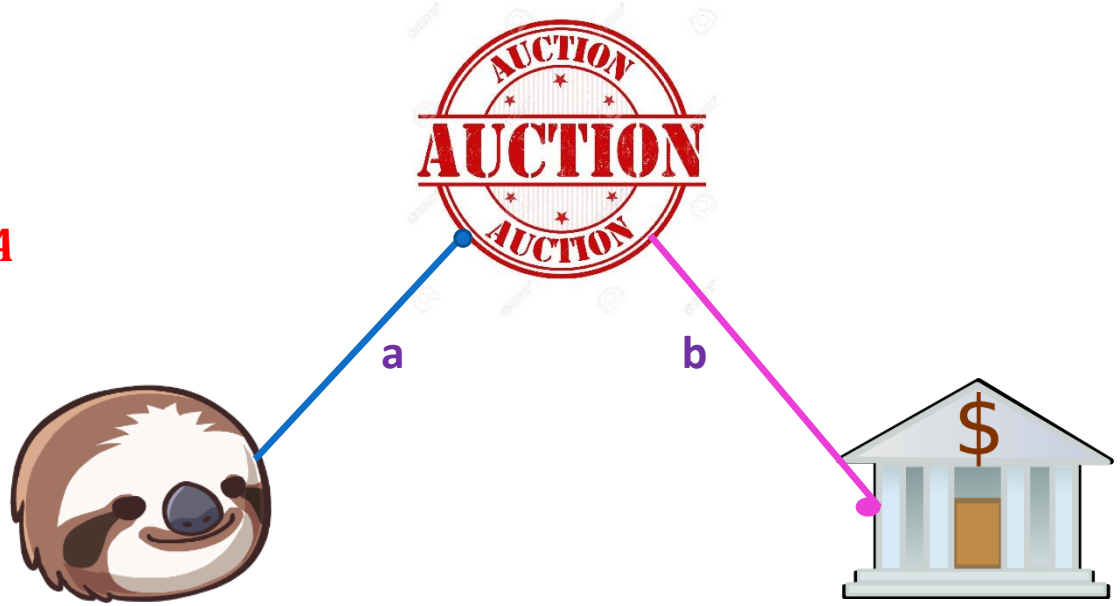
$B \multimap \exists x\colon bool.\, A$



$\&\{\, transfer\colon \forall x\colon int.\, \oplus\, \{ok\colon \exists y\colon int.\, \mathbf{1}; no\colon \mathbf{1}\}\}$

Communication Reconfigures

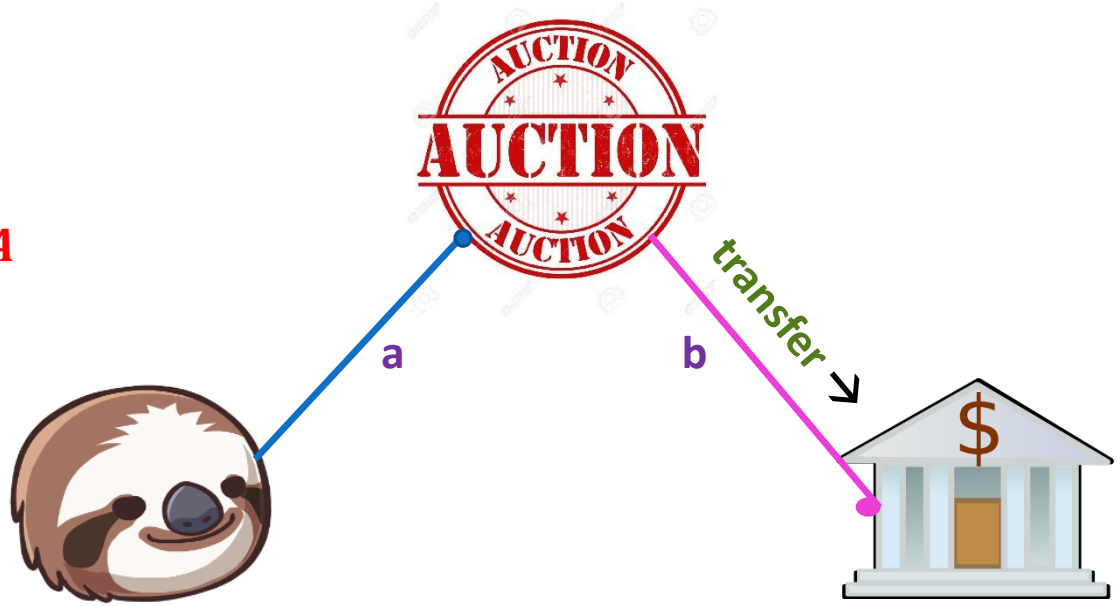
$\exists x : bool . A$



$\& \{ transfer : \forall x : int . \oplus \{ ok : \exists y : int . 1 ; no : 1 \} \}$

Start Money Transfer

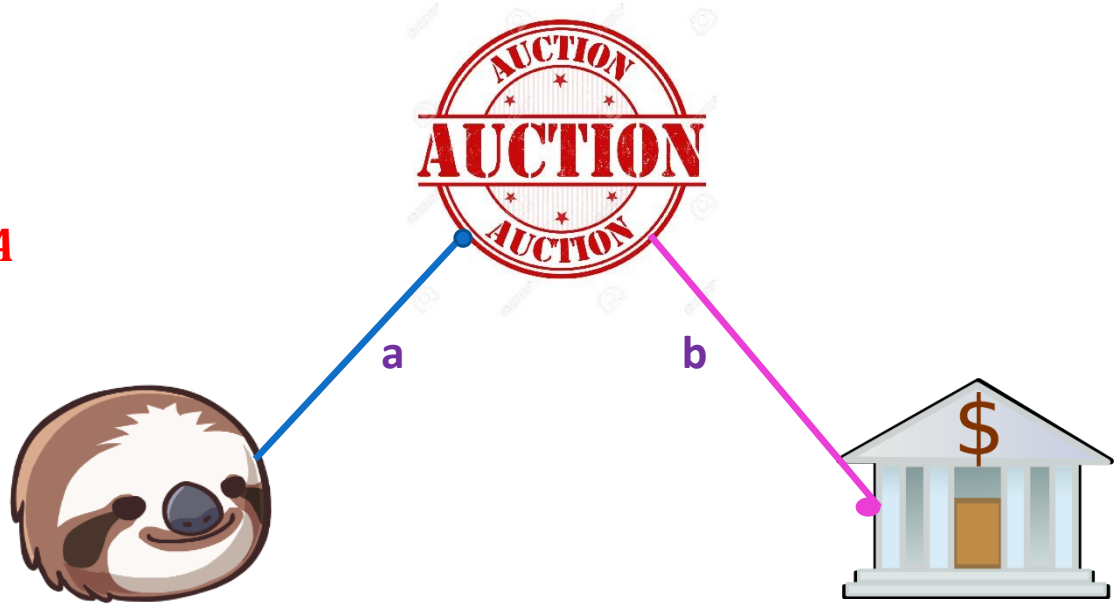
$\exists x : bool . A$



$\& \{ transfer : \forall x : int . \oplus \{ ok : \exists y : int . 1 ; no : 1 \} \}$

Type Changes!

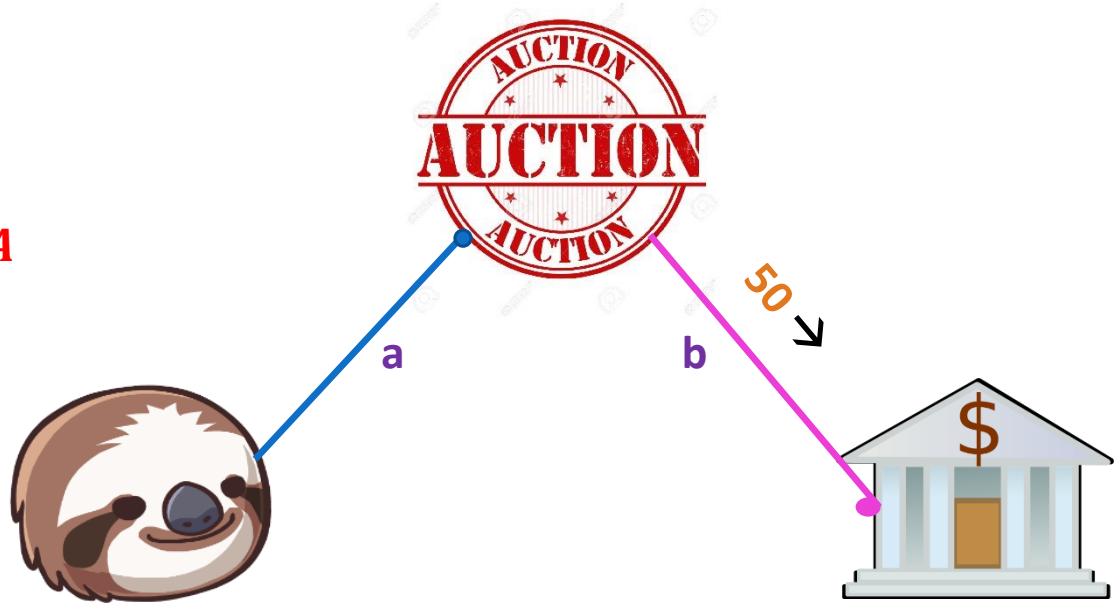
$\exists x : bool . A$



$\forall x : int . \oplus \{ok : \exists y : int . 1; no : 1\}$

Request Amount

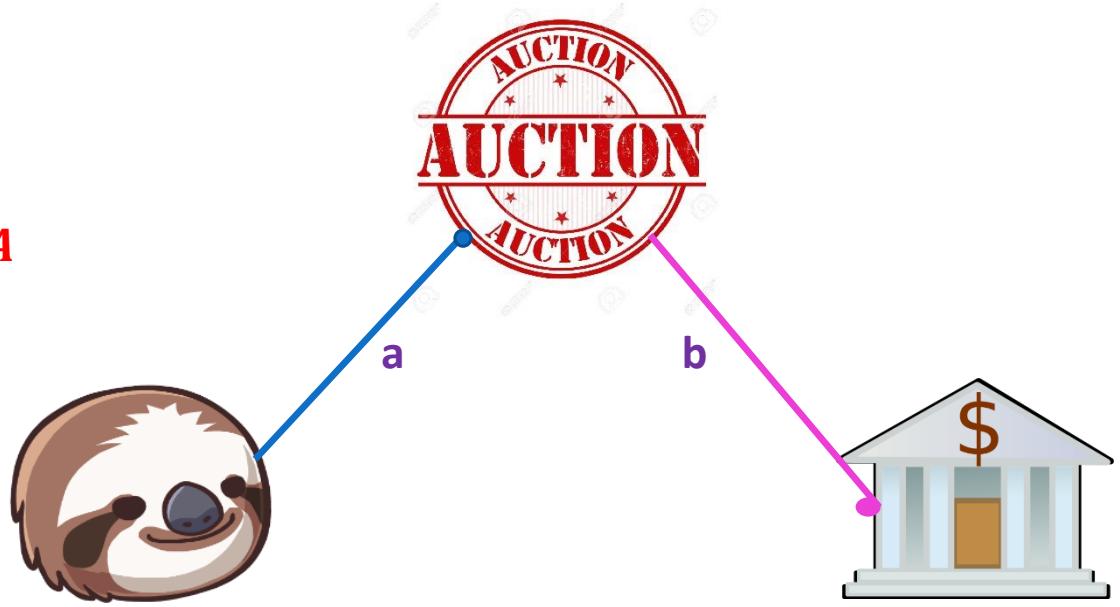
$\exists x : bool . A$



$\forall x : int . \oplus \{ok : \exists y : int . 1; no : 1\}$

Type Changes!

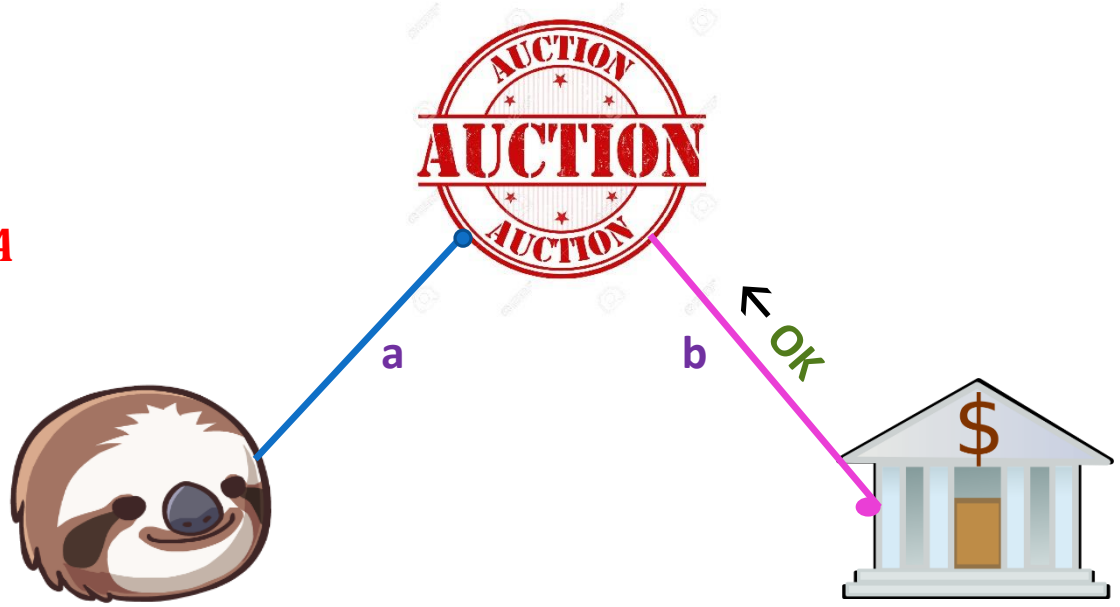
$\exists x : bool.\ A$



$\oplus\ \{ok : \exists y : int.\ 1;\ no : 1\}$

Bank Acknowledgement

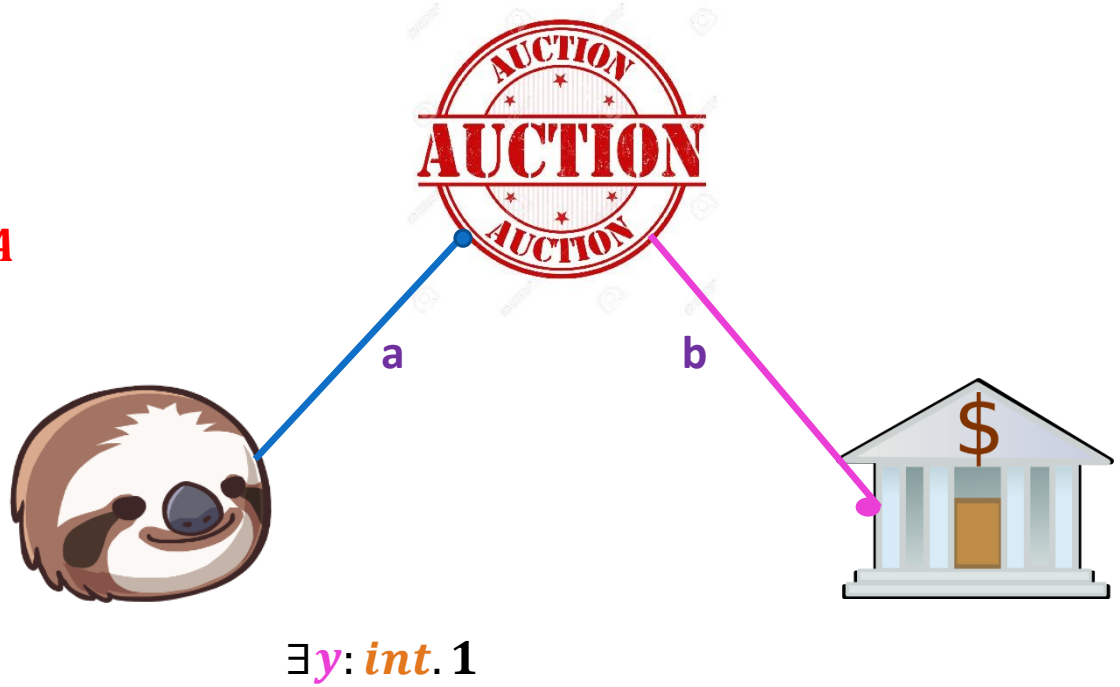
$\exists x : bool . A$



$\oplus \{ok : \exists y : int . 1; no : 1\}$

Type Changes!

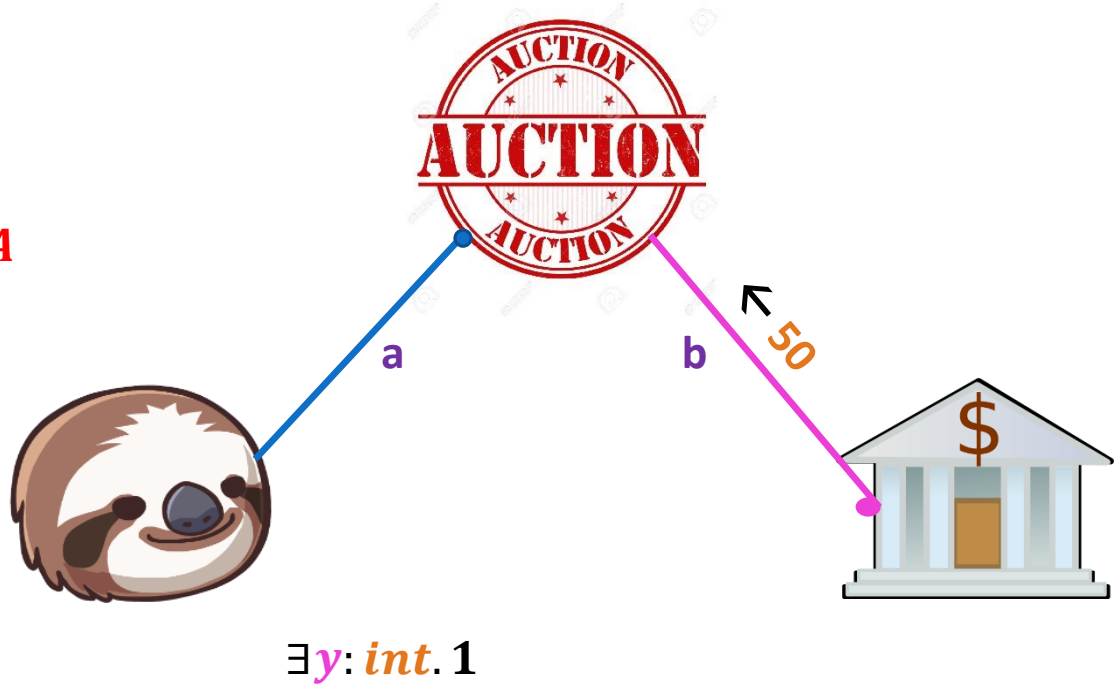
$\exists x : bool . A$



$\exists y : int . 1$

Bank Receipt

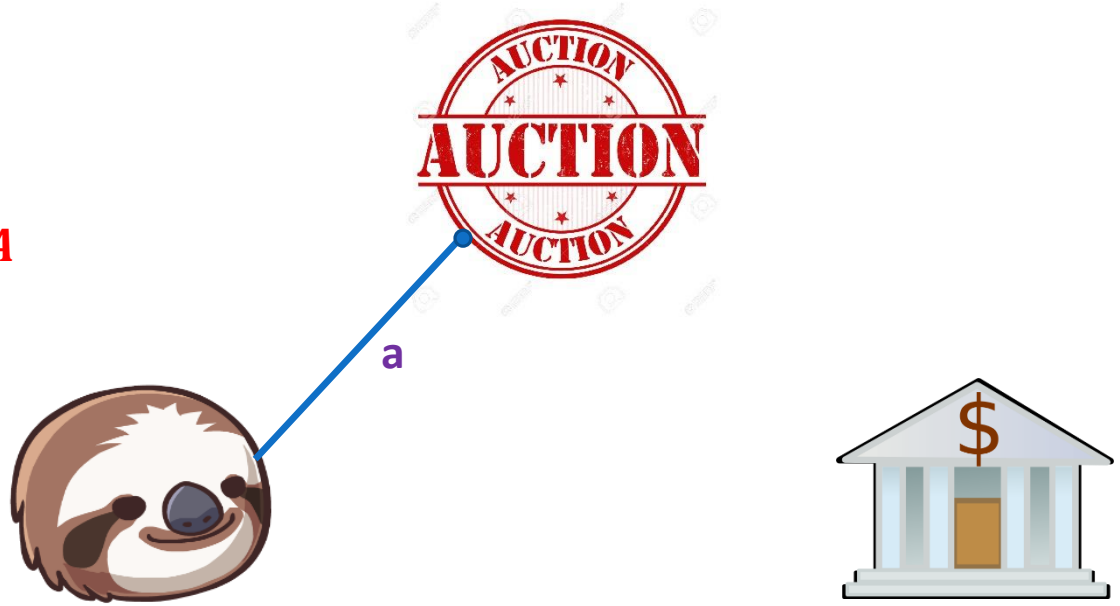
$\exists x : bool . A$



$\exists y : int . 1$

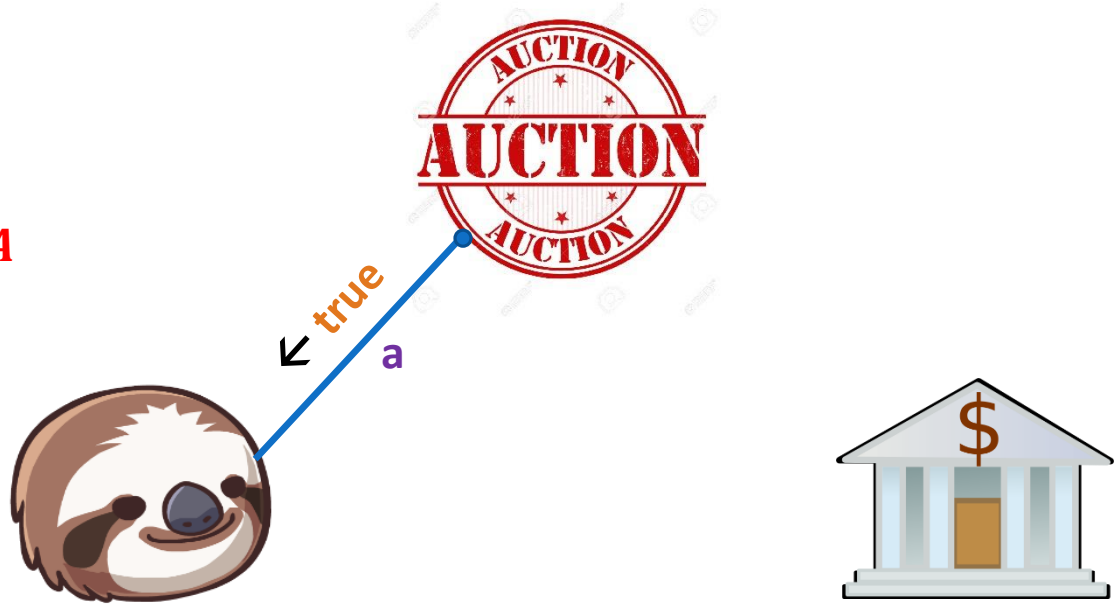
Bank Connection Terminates

$\exists x : bool . A$

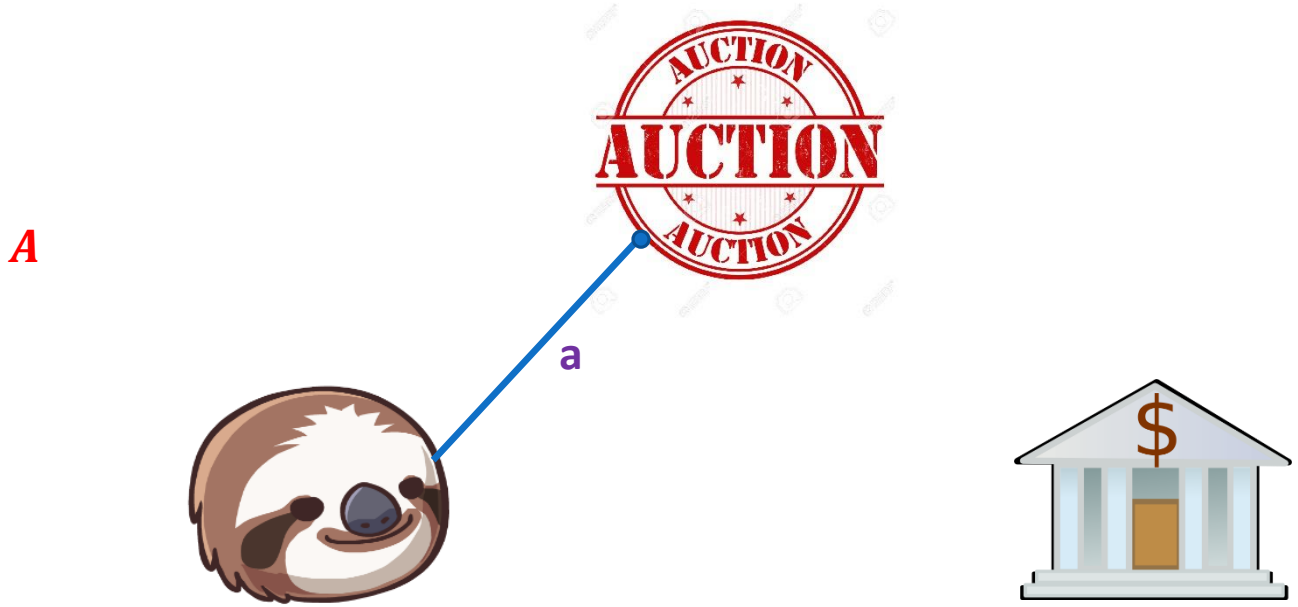


Auction Acknowledgement

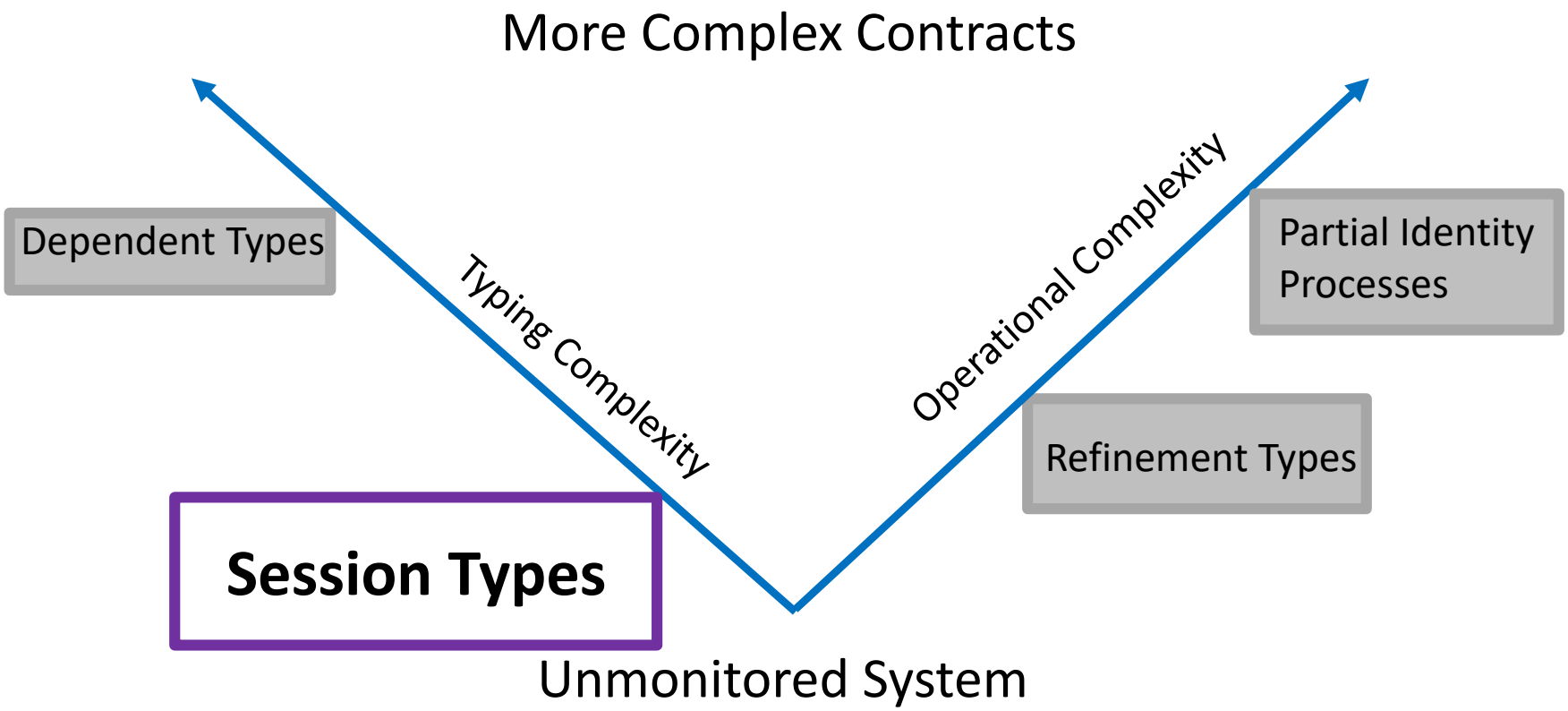
$\exists x : bool . A$



Auction Protocol Complete



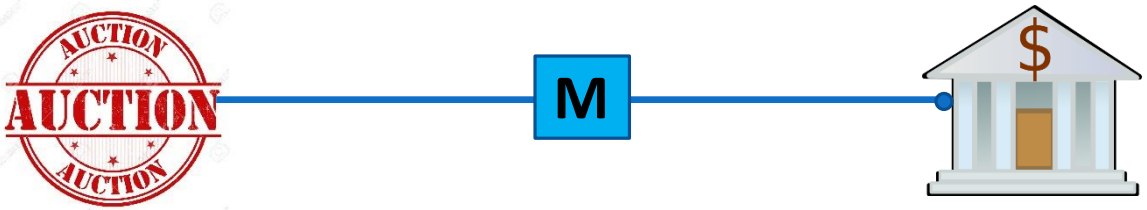
Contract Classes



Monitors for Session-Types

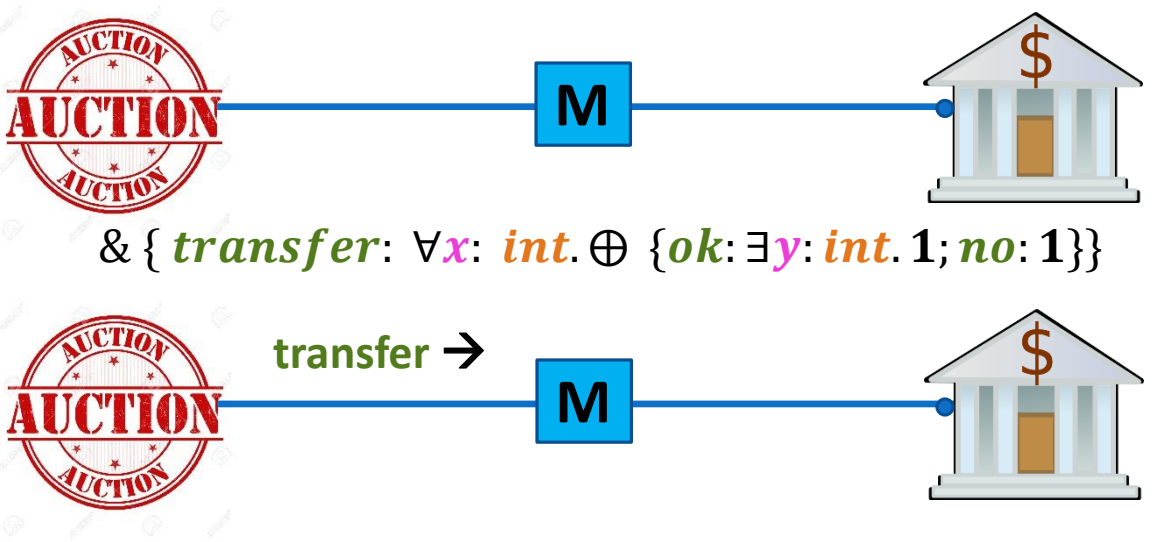
- Every channel has a monitor on it
- Can only observe communicated values, have no access to process internals
- If error is detected, raise alarm and stop computation
- If no alarm is raised, monitors do not change system behavior

Simple Bank Monitor

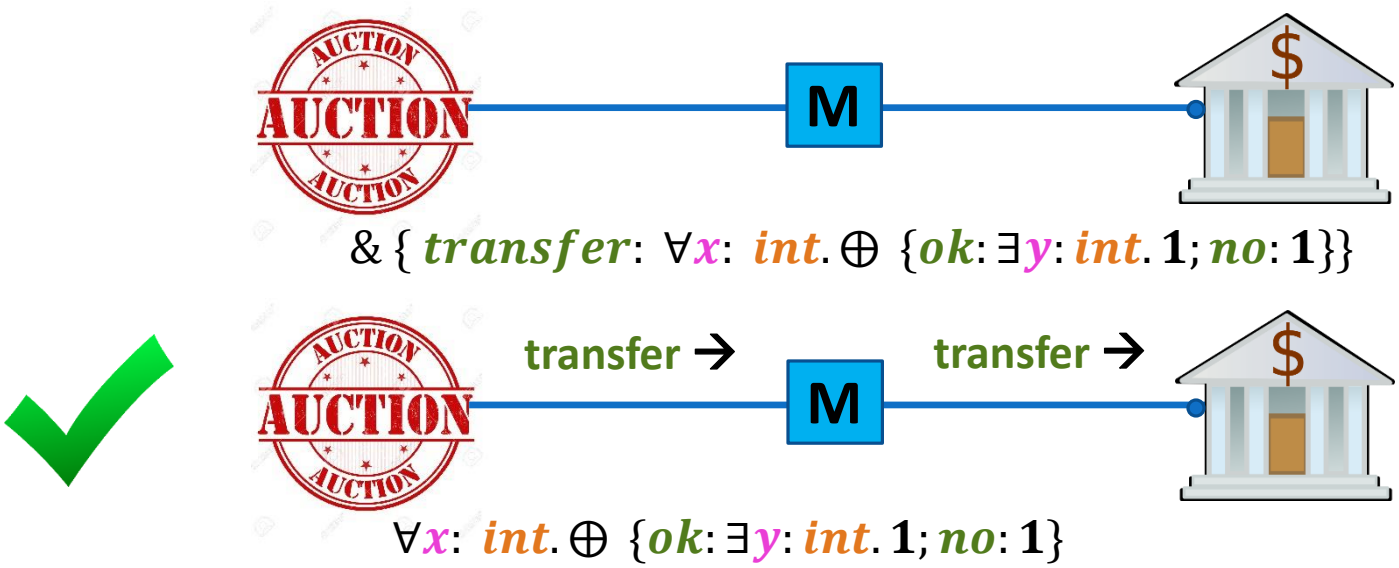


$$\&\{\mathit{transfer}\colon \forall x\colon \mathit{int}.\oplus\{\mathit{ok}\colon \exists y\colon \mathit{int}.\mathbf{1};\mathit{no}\colon \mathbf{1}\}\}$$

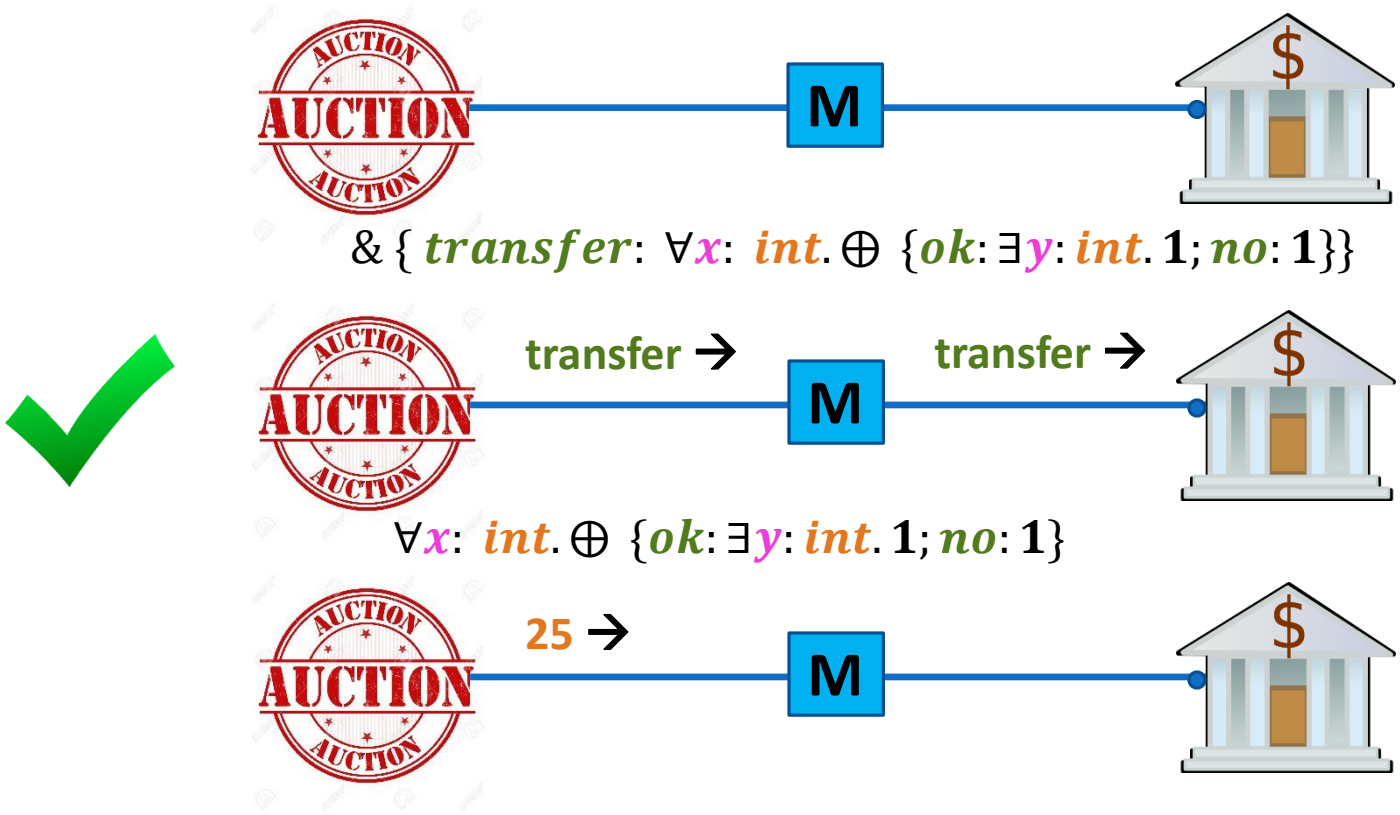
Simple Bank Monitor



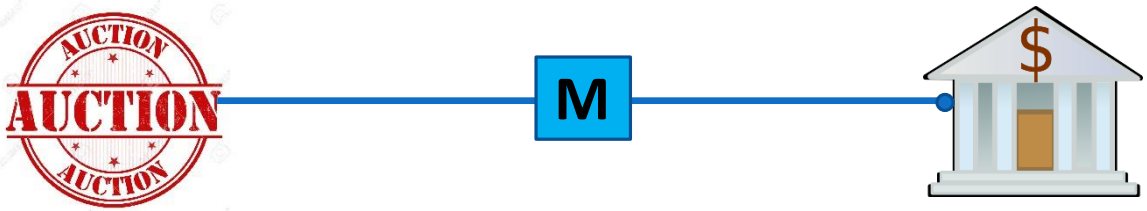
Simple Bank Monitor



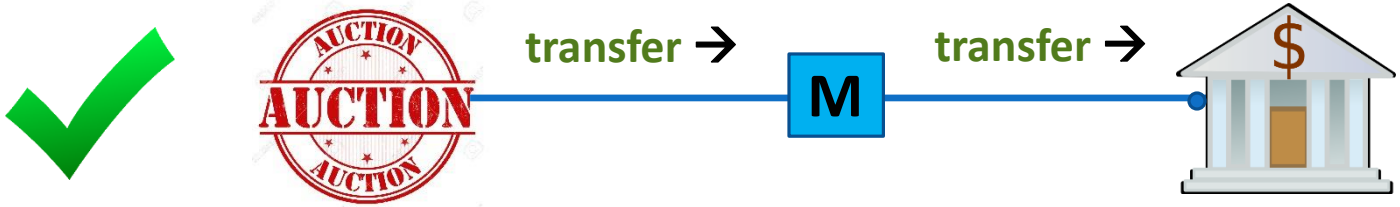
Simple Bank Monitor



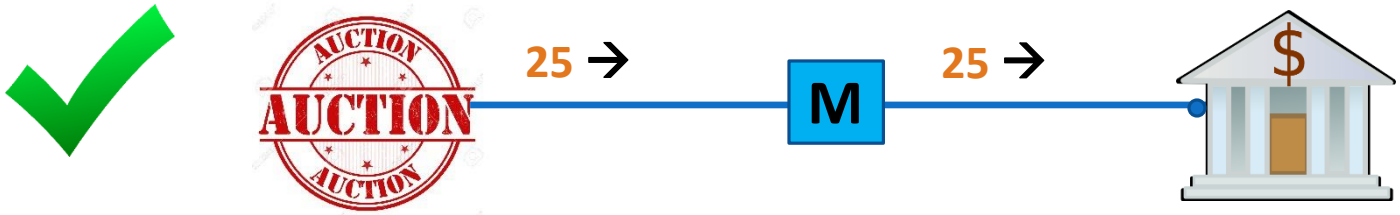
Simple Bank Monitor



$\&\{\ transfer\colon \forall x\colon int.\oplus\{ok\colon \exists y\colon int.\ 1; no\colon 1\}\}$

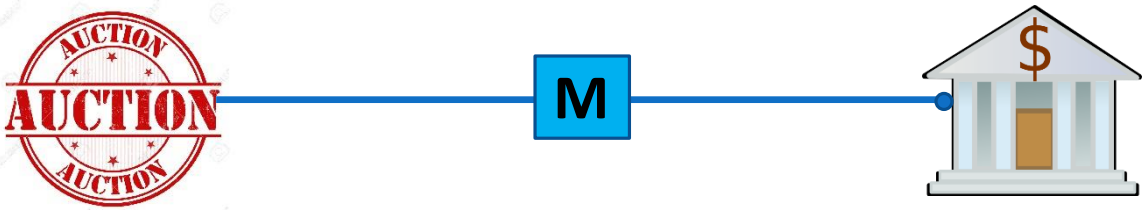


$\forall x\colon int.\oplus\{ok\colon \exists y\colon int.\ 1; no\colon 1\}$



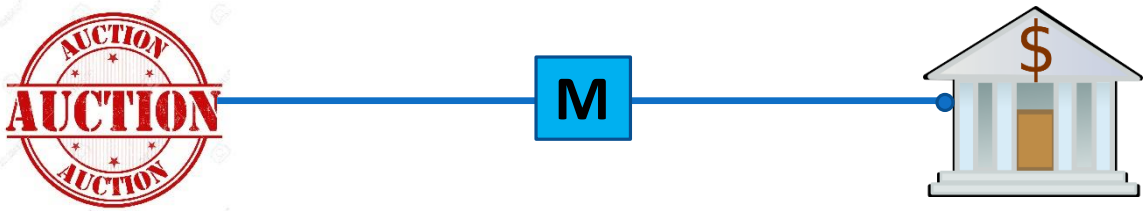
$\oplus\{ok\colon \exists y\colon int.\ 1; no\colon 1\}$

Simple Bank Monitor

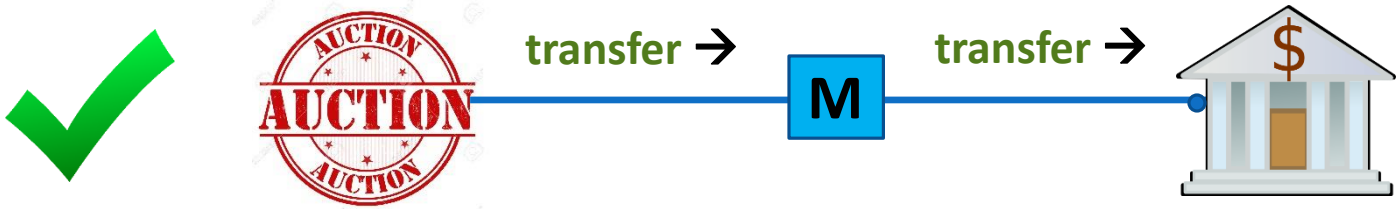


$$\&\{\boldsymbol{transfer}\colon \forall \boldsymbol{x}\colon \boldsymbol{int}.\oplus\{\boldsymbol{ok}\colon \exists \boldsymbol{y}\colon \boldsymbol{int}.\mathbf{1};\boldsymbol{no}\colon \mathbf{1}\}\}$$

Simple Bank Monitor

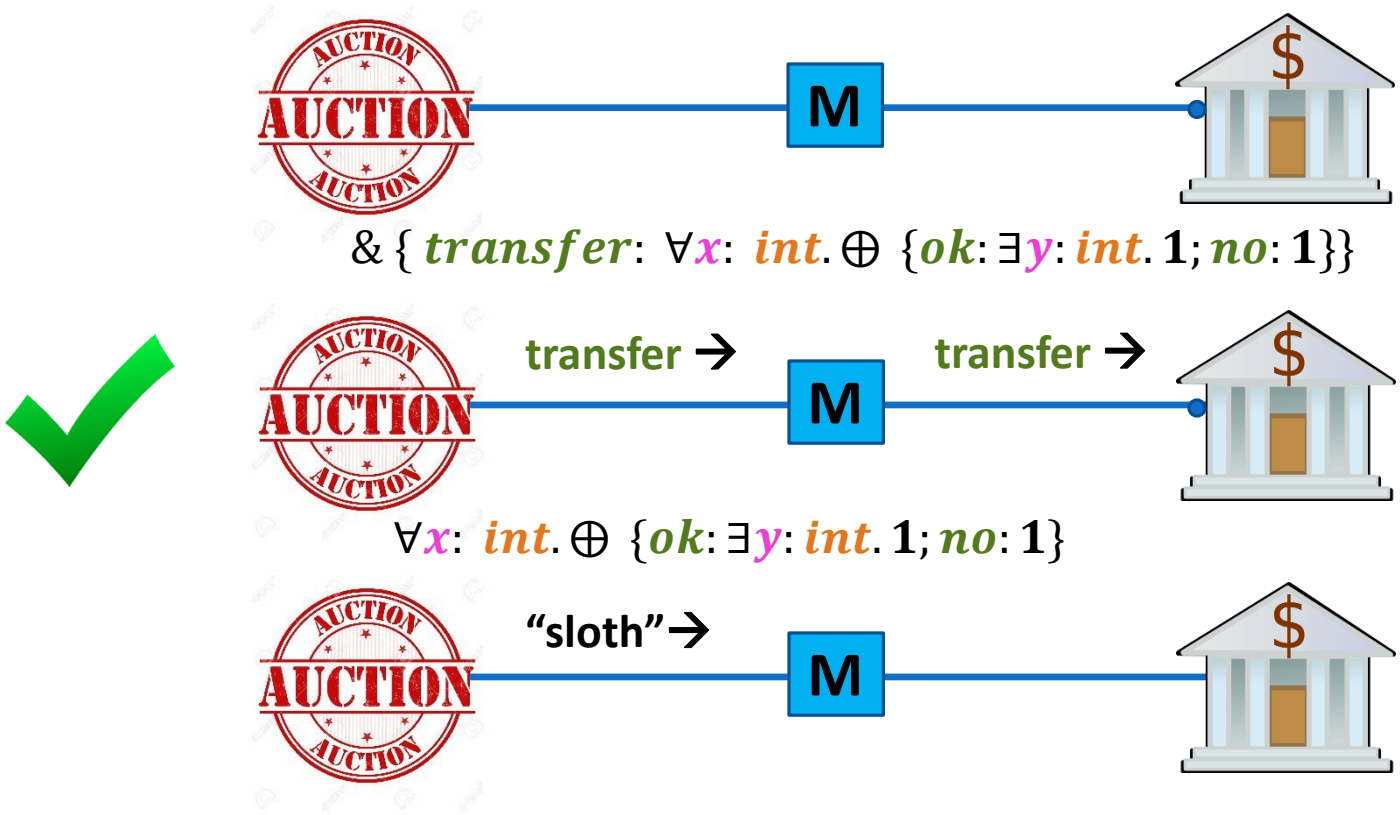


$\&\{\ transfer\colon \forall x\colon int.\oplus\{ok\colon \exists y\colon int.\mathbf{1};no\colon \mathbf{1}\}\}$

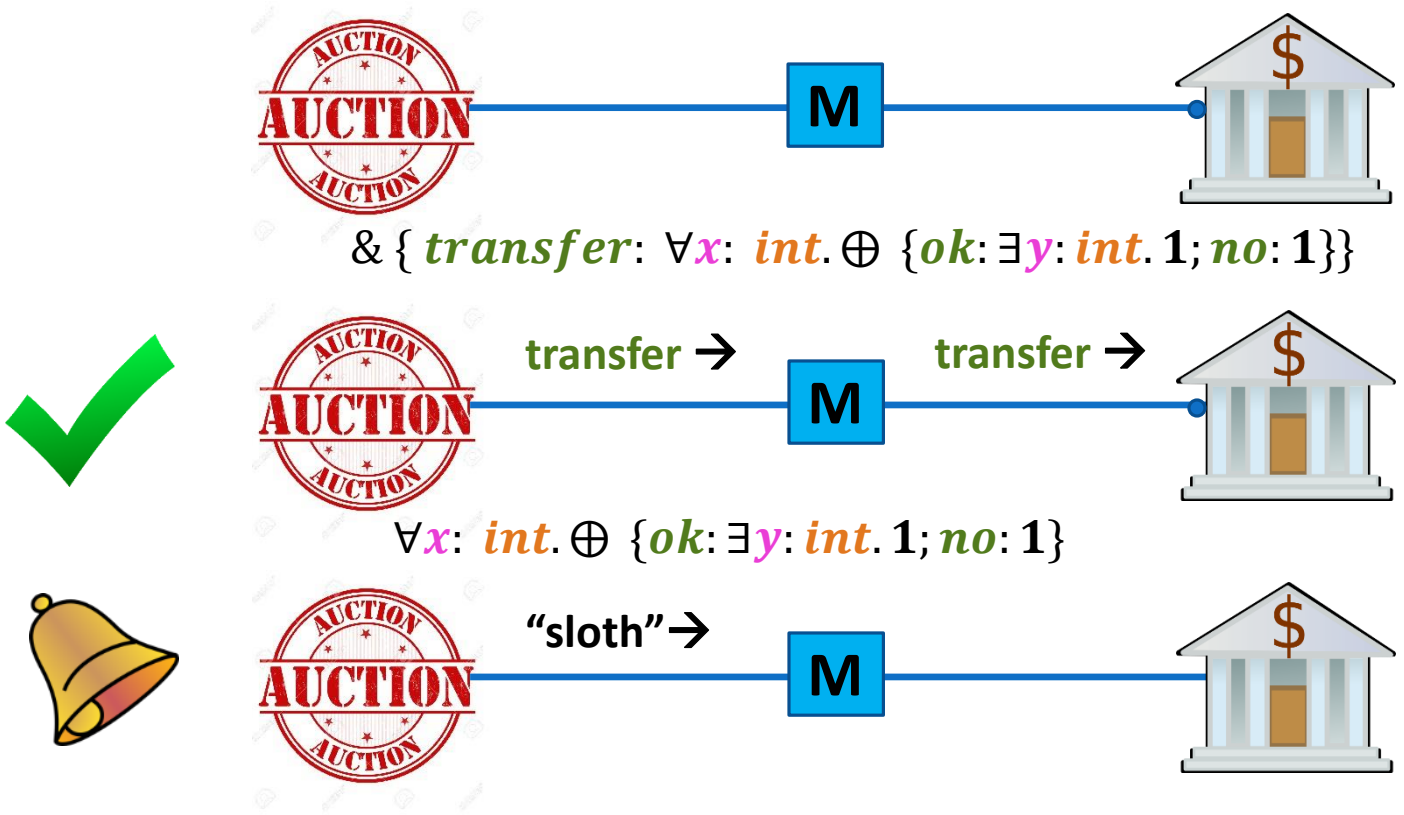


$\forall x\colon int.\oplus\{ok\colon \exists y\colon int.\mathbf{1};no\colon \mathbf{1}\}$

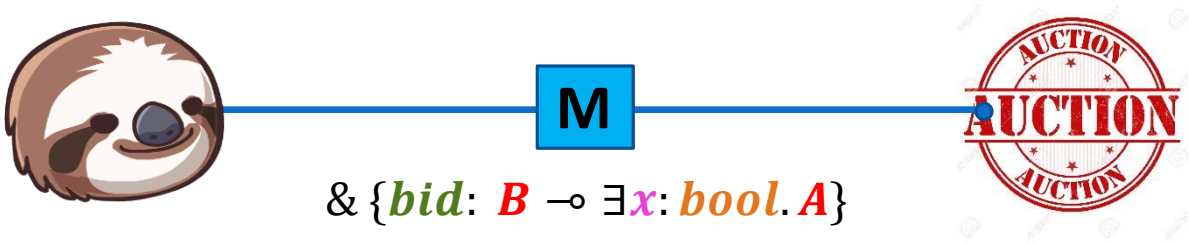
Simple Bank Monitor



Simple Bank Monitor

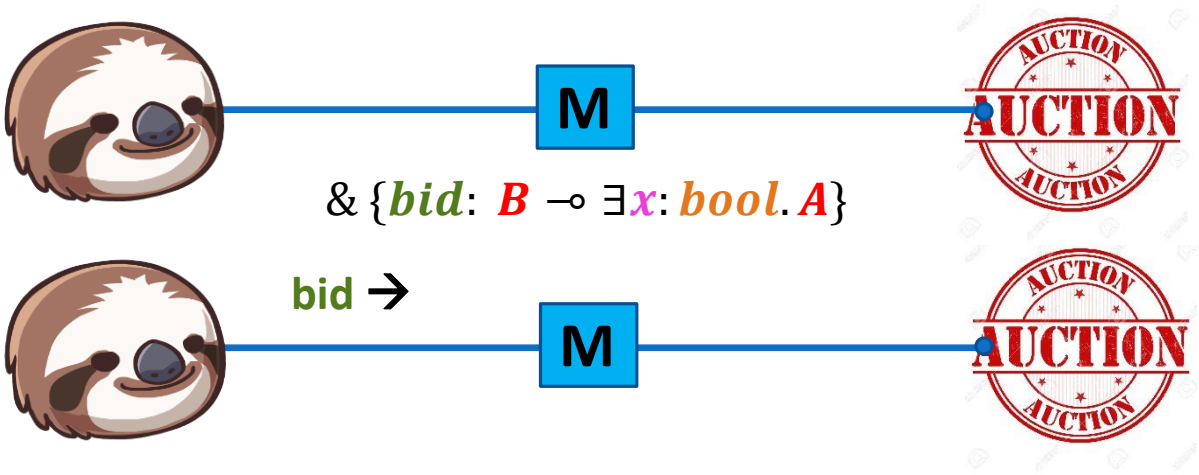


Higher-Order Auction Monitor

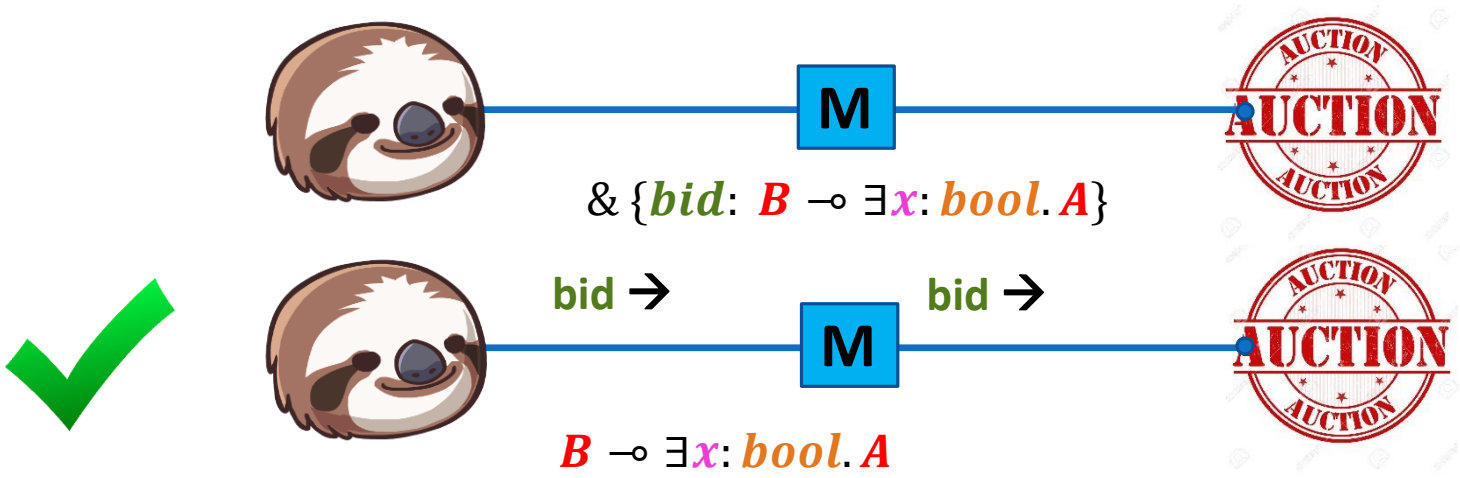


$$\&\{bid\colon B \multimap \exists x\colon bool.\, A\}$$

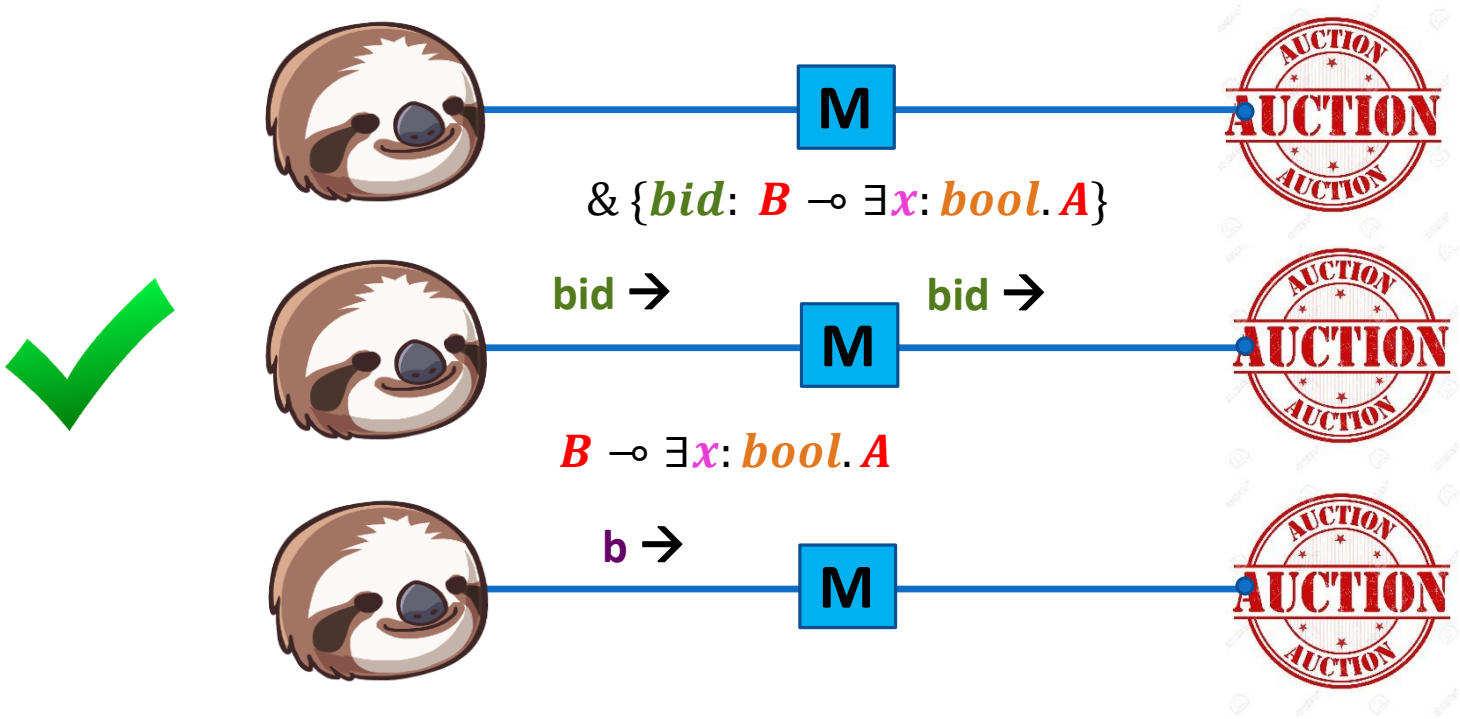
Higher-Order Auction Monitor



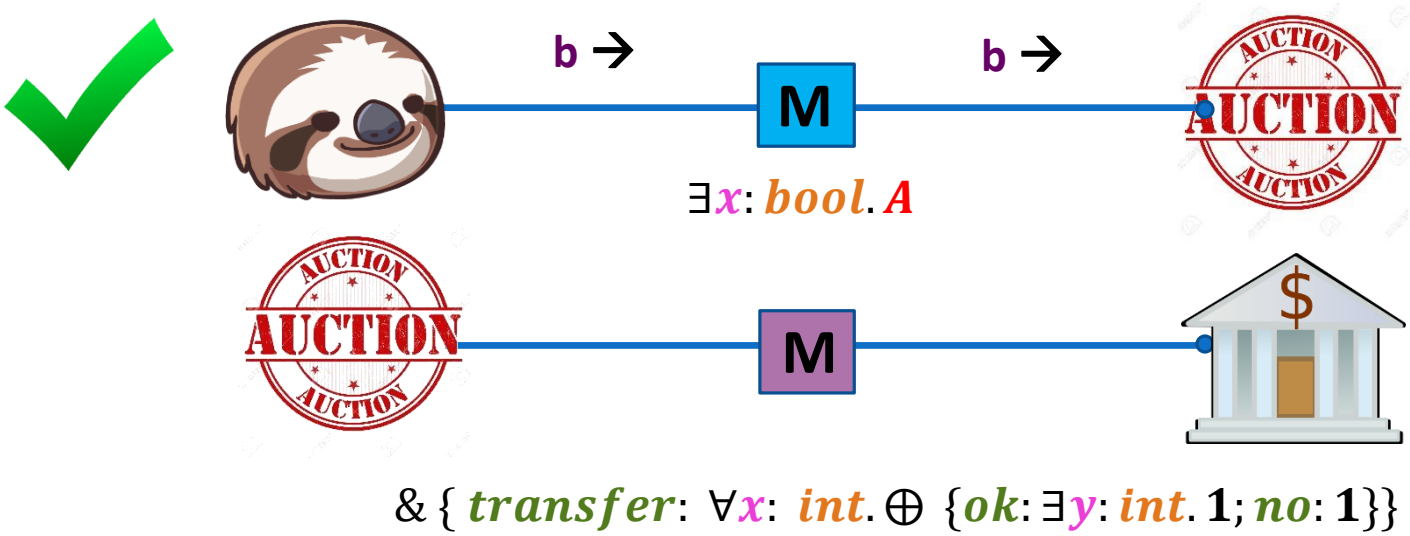
Higher-Order Auction Monitor



Higher-Order Auction Monitor



Higher-Order Auction Monitor



Blame Assignment

- Can determine which process caused the observed problem
- Based on system assumptions, our blame assignment can single out the rogue process or blame a set of processes
- Processes spawn other processes and delegate communication which constantly reconfigures communication

Theoretical Contributions

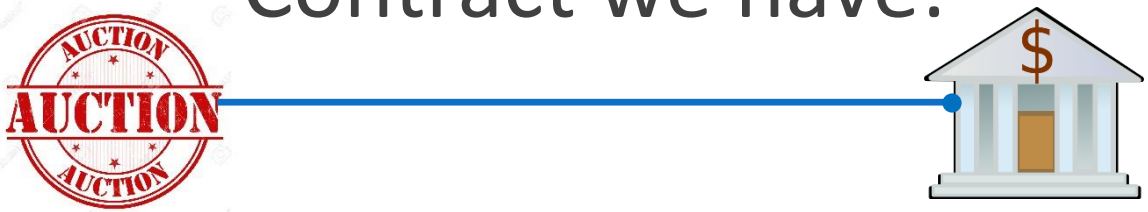
- Correctness of Blame Assignment (Safety)
  - One of indicated process(es) went rogue
- Monitor Transparency
  - Monitors do not change system behavior for well-behaved processes

New Problem

What if the auction asks the bank for $50 and the bank sends back a receipt for $25?

New Problem

Contract we have:



$\& \{ transfer\colon \forall x\colon int.\, \oplus \{ok\colon \exists y\colon int.\, \mathbf{1}; no\colon \mathbf{1}\}\}$

Contract we want:

Bank sends receipt for requested amount

New Problem

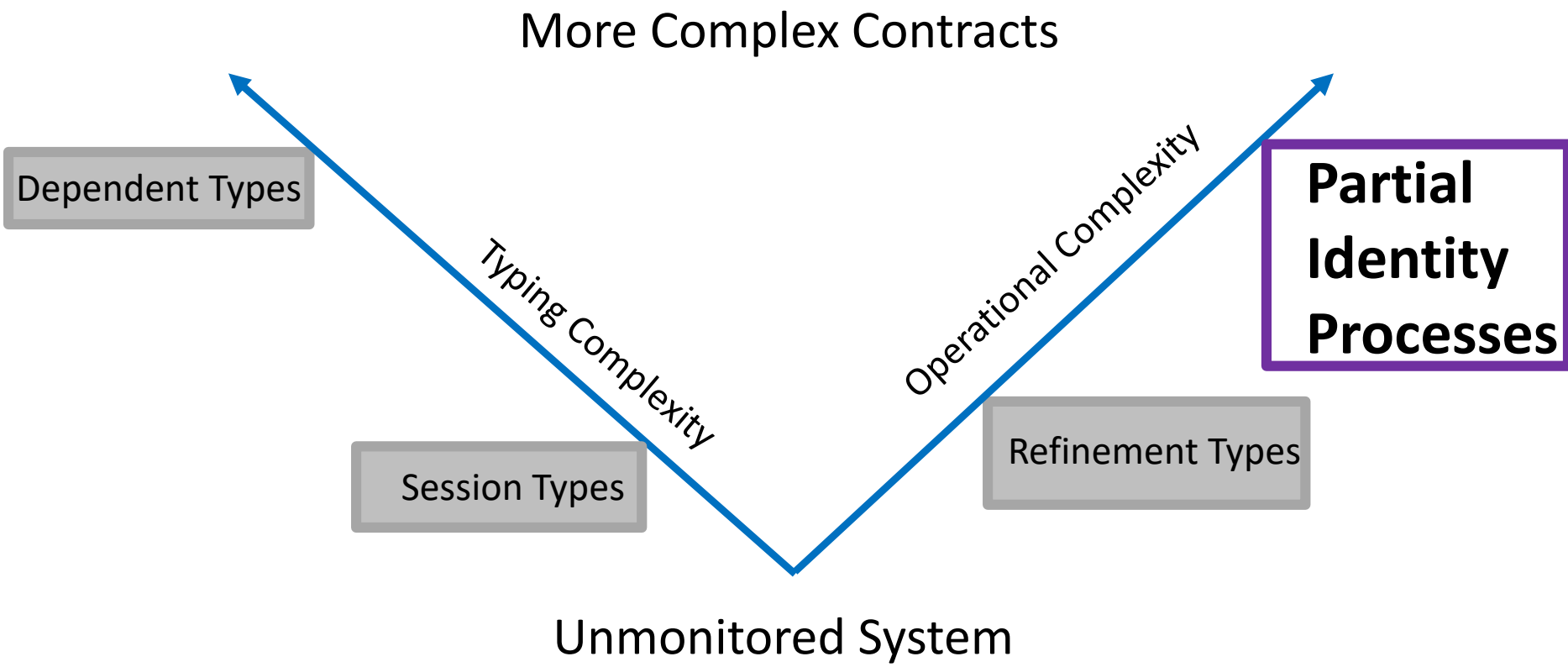
Contract we have:

=

Contract we want:

Bank sends receipt for requested amount

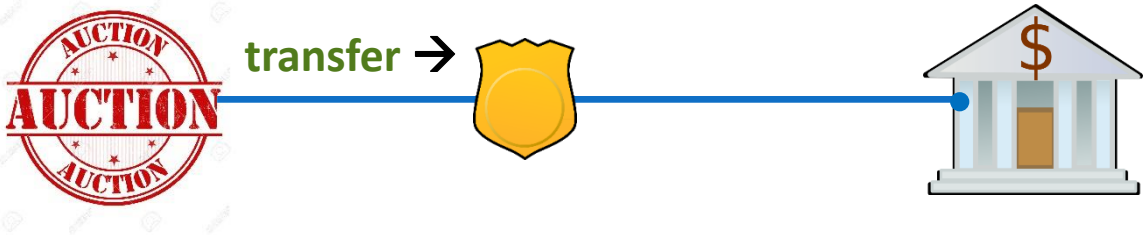
Contract Classes



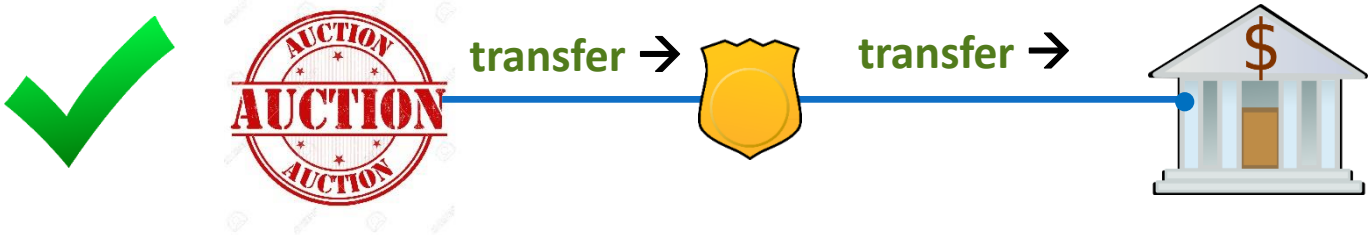
Monitoring Processes

- Defined as partial identity processes
- Abort if contract is violated, otherwise are not observable
- Check properties by using internal state

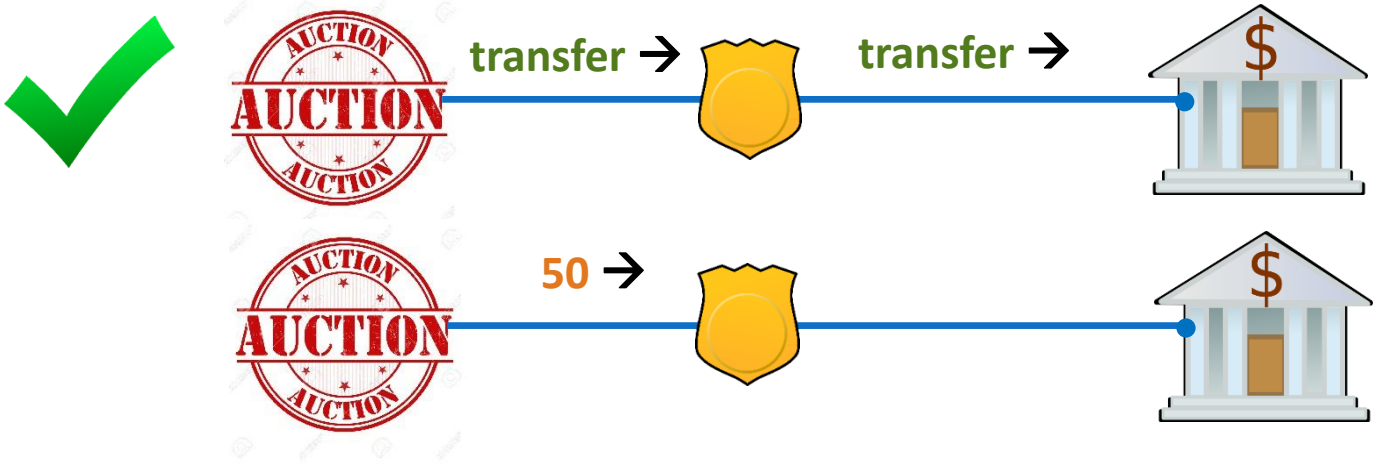
Stateful Bank Monitor



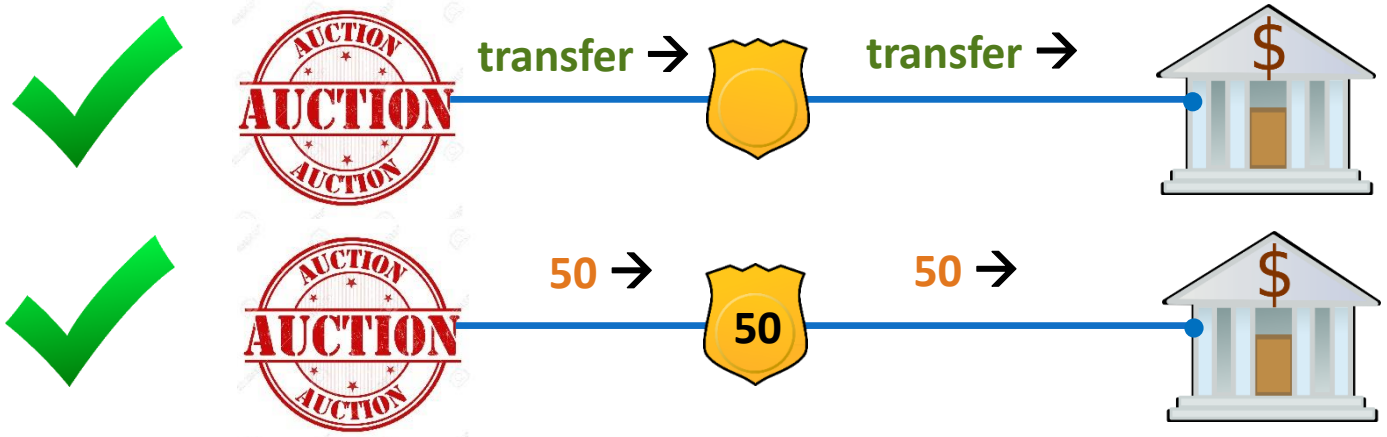
Stateful Bank Monitor



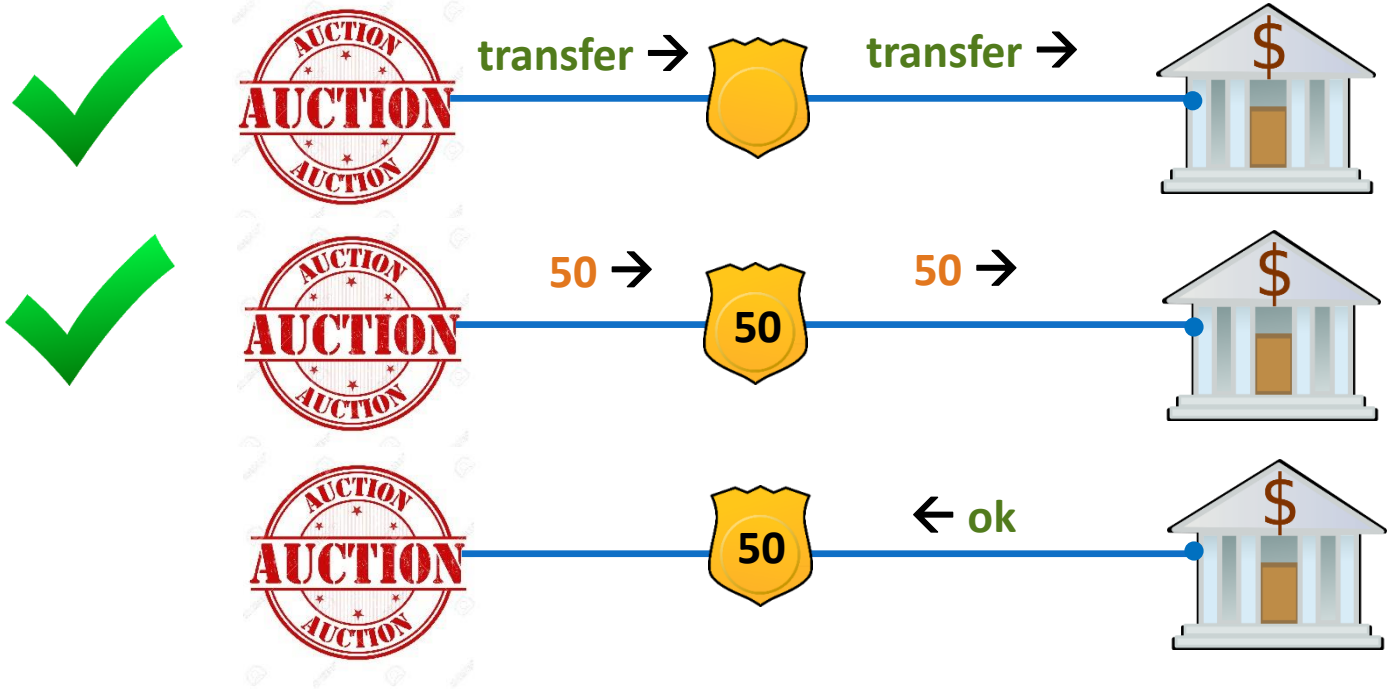
Stateful Bank Monitor



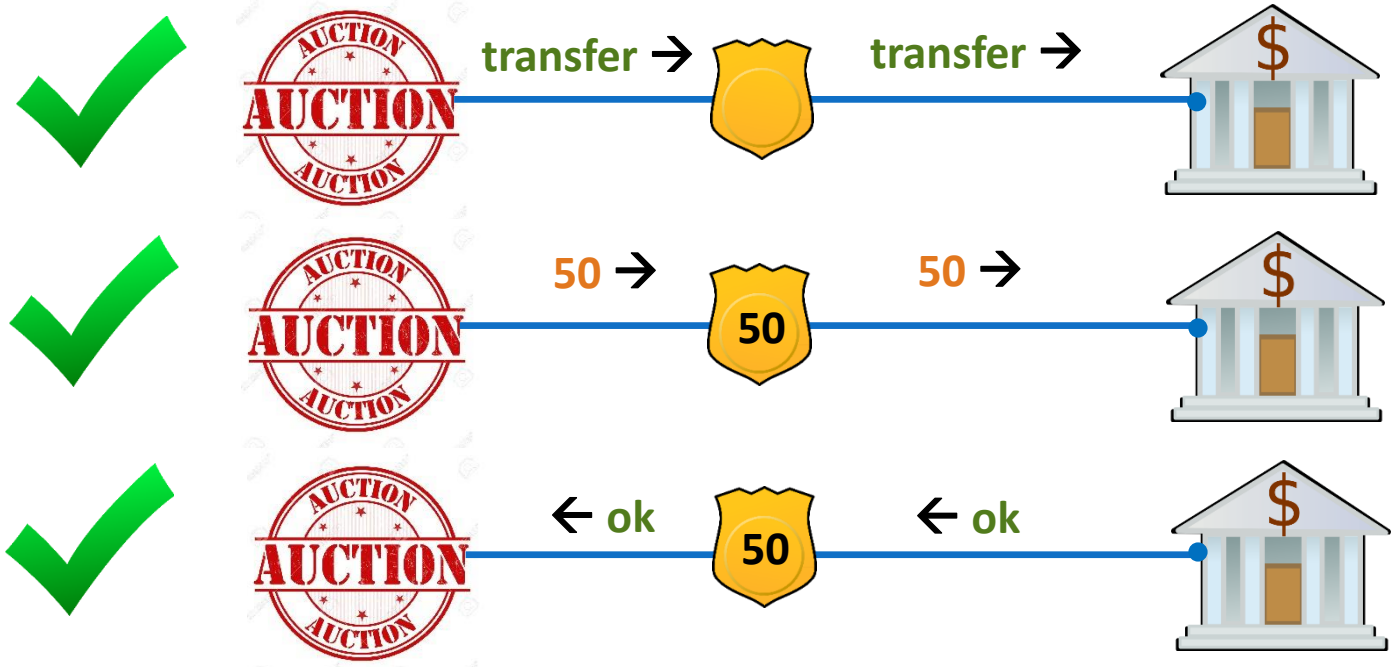
Stateful Bank Monitor



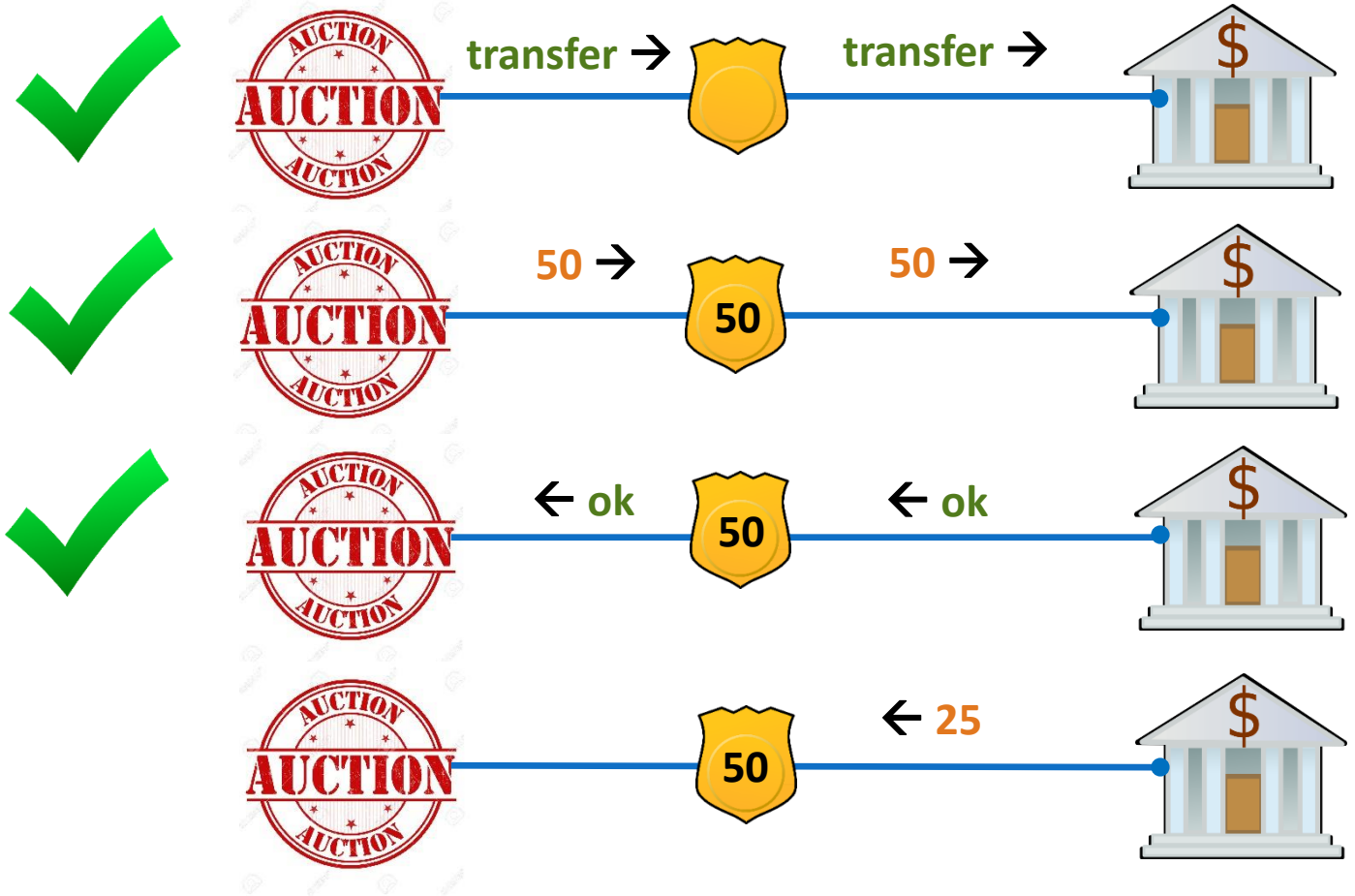
Stateful Bank Monitor



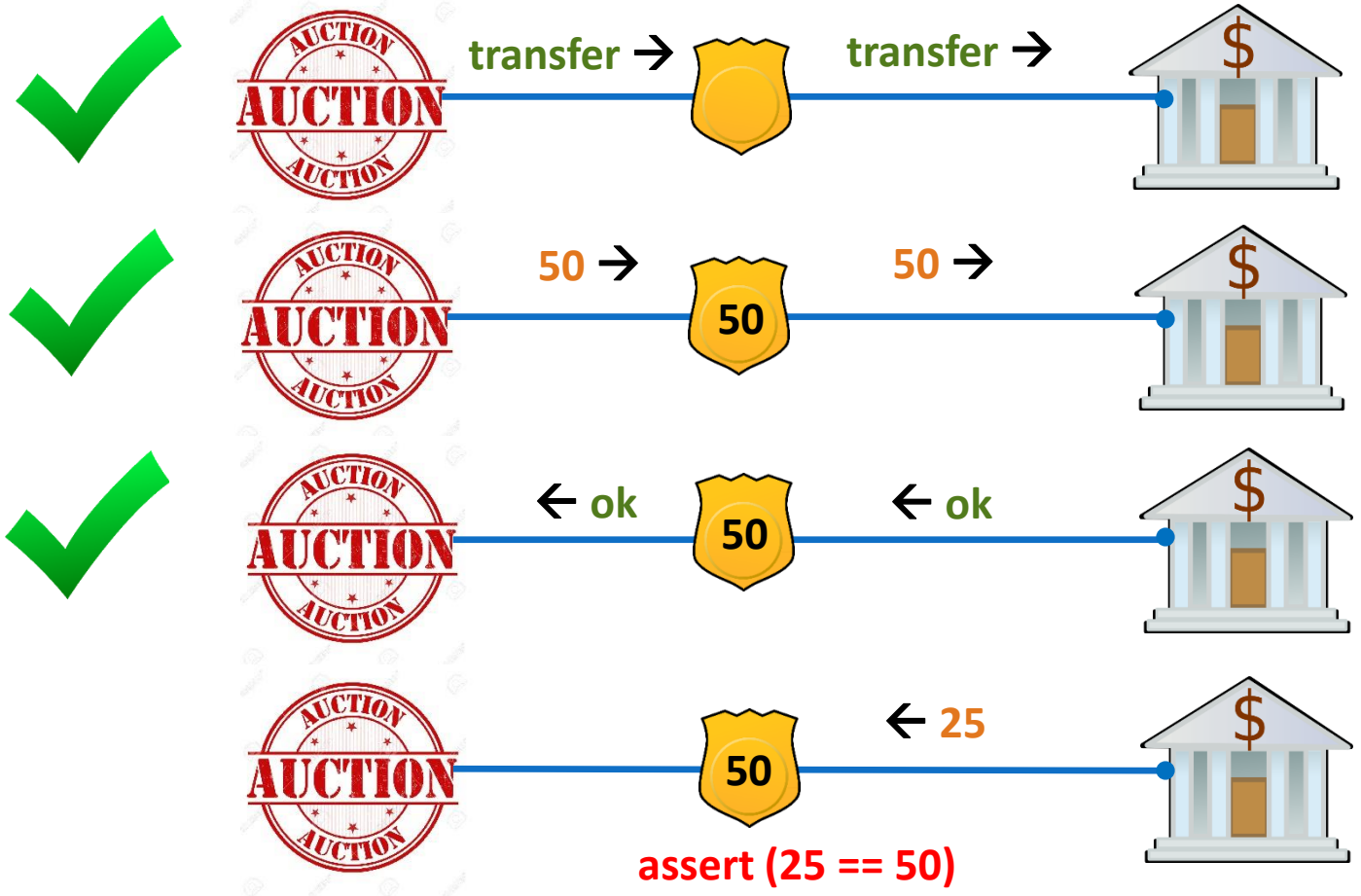
Stateful Bank Monitor



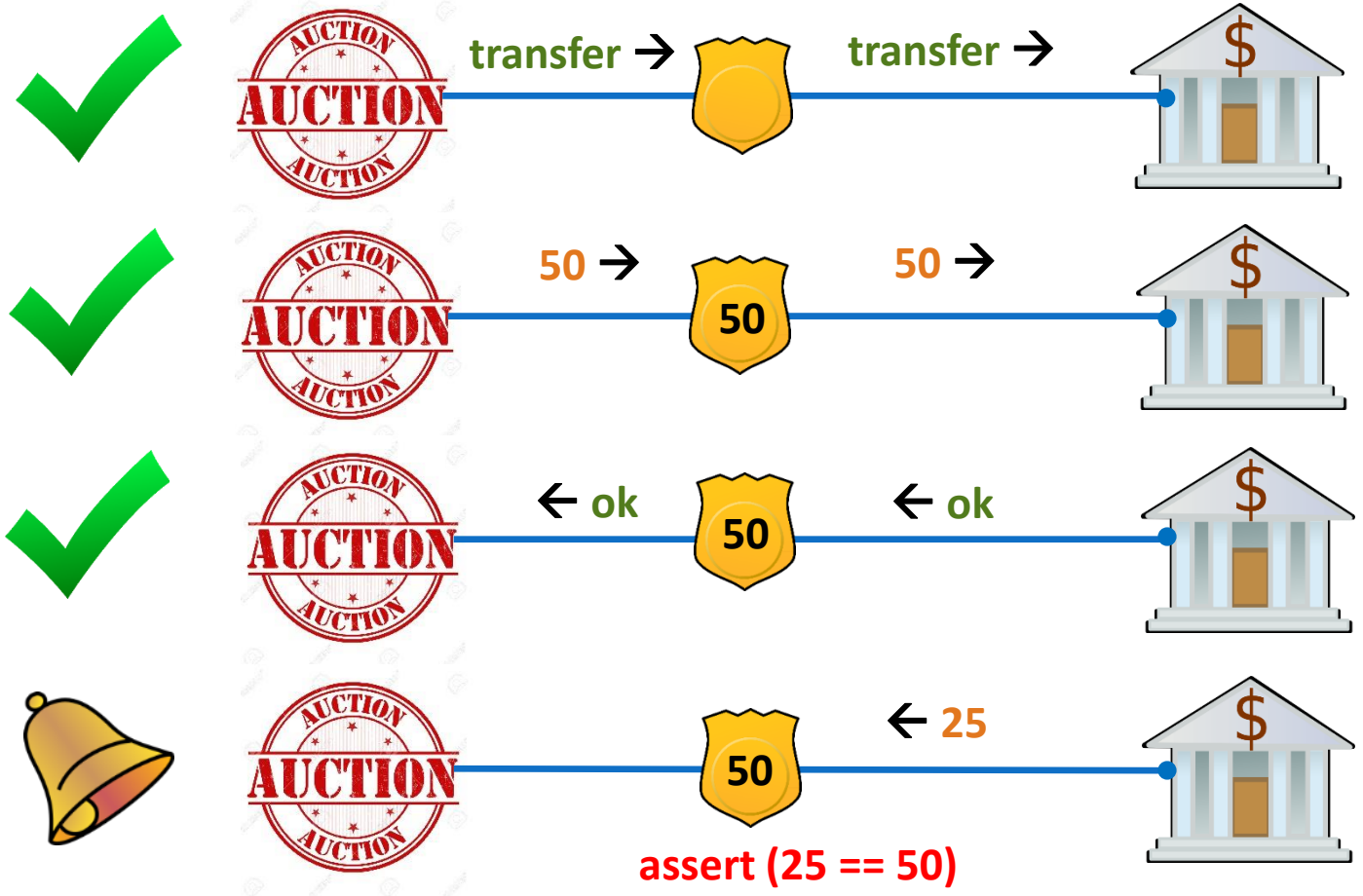
Stateful Bank Monitor



Stateful Bank Monitor



Stateful Bank Monitor



Bank Monitor Code

bank_id : B ← B

b’ ← bank_id b =

case b’ of

| transfer ⇒ x ← recv b’;

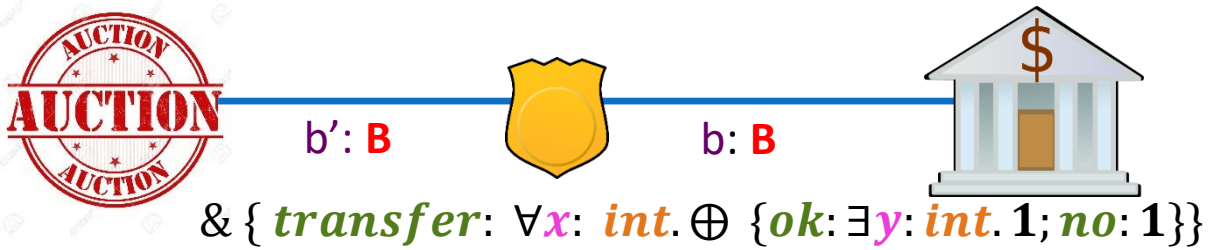
b.transfer; send b x;

case b of

| ok ⇒ y ← recv b;

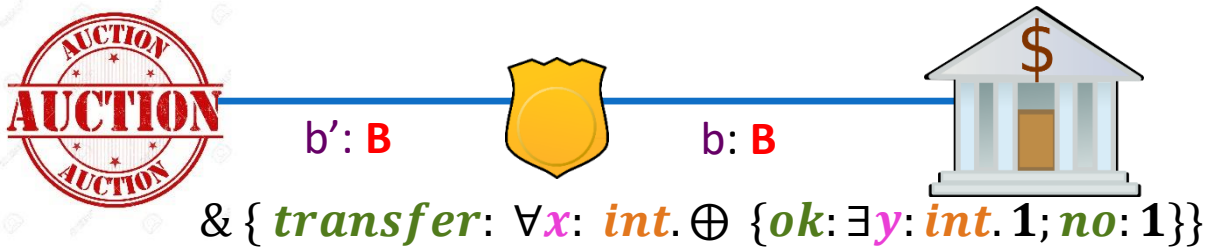
b’.ok; send b’ y;

| no ⇒ b’.no;



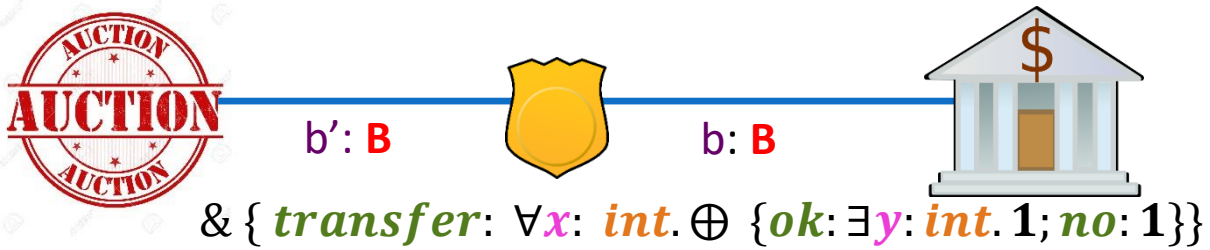
Bank Monitor Code

```
bank_id : B ← B
b' ← bank_id  b =
    case b' of
      | transfer ⇒ x ← recv b';
                   b.transfer; send b x;
                   case b of
                     | ok ⇒ y ← recv b;
                            b'.ok; send b' y;
                     | no ⇒ b'.no;
```



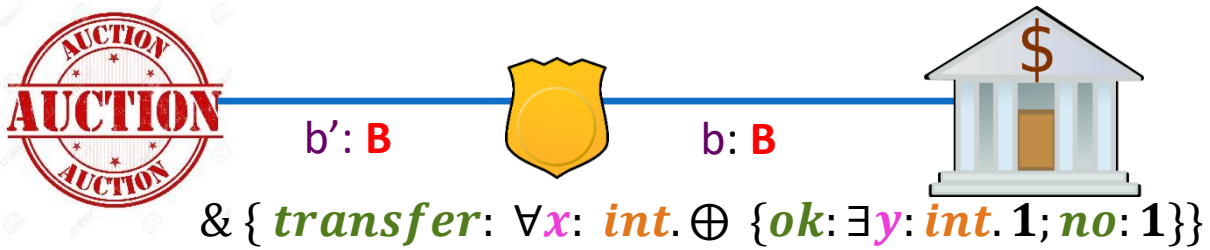
Bank Monitor Code

```
bank_id : B ← B
b' ← bank_id  b =
    case b' of
        | transfer ⇒ x ← recv b';
                     b.transfer; send b x;
                     case b of
                        | ok ⇒ y ← recv b;
                               b'.ok; send b' y;
                        | no ⇒ b'.no;
```



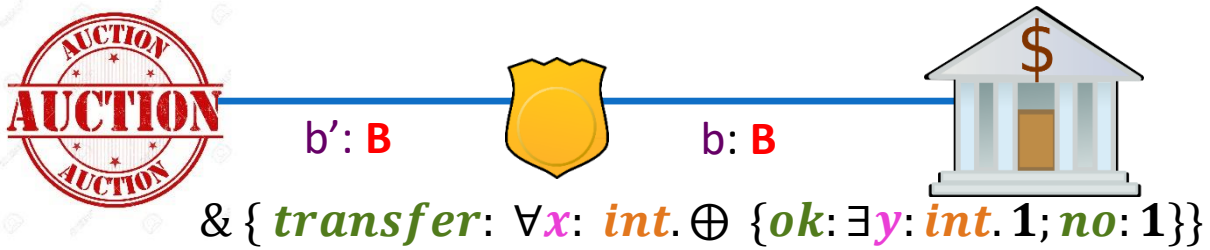
Bank Monitor Code

```
bank_id : B ← B
b' ← bank_id  b =
    case b' of
      | transfer ⇒ x ← recv b';
                   b.transfer; send b x;
                   case b of
                     | ok ⇒ y ← recv b;
                            b'.ok; send b' y;
                     | no ⇒ b'.no;
```

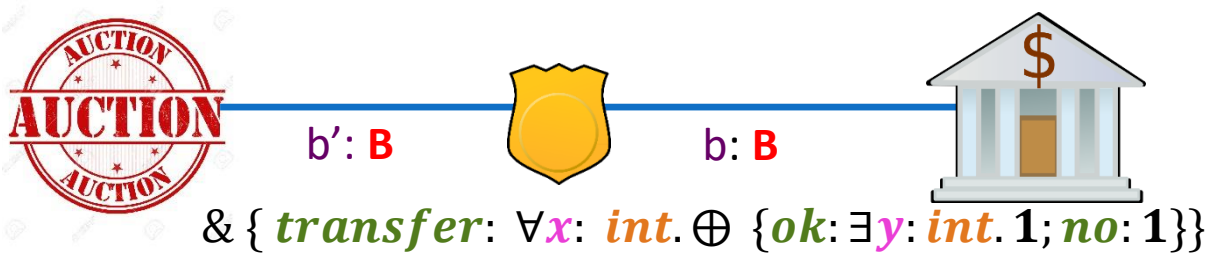


Bank Monitor Code

```
bank_id : B ← B
b' ← bank_id  b =
    case b' of
        | transfer ⇒ x ← recv b';
                     b.transfer; send b x;
                     case b of
                        | ok ⇒ y ← recv b;
                               b'.ok; send b' y;
                        | no ⇒ b'.no;
```



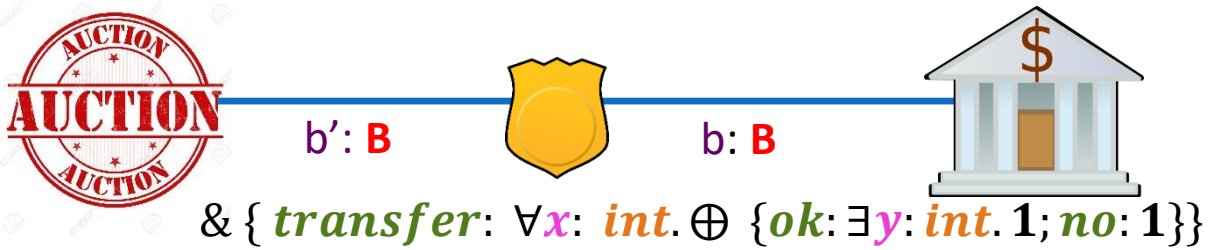
Bank Monitor Code




```
bank_id : B ← B
b' ← bank_id  b =
    case b' of
        | transfer ⇒ x ← recv b';
                     b.transfer; send b x;
                     case b of
                        | ok ⇒ y ← recv b;
                               b'.ok; send b' y;
                        | no ⇒ b'.no;
```

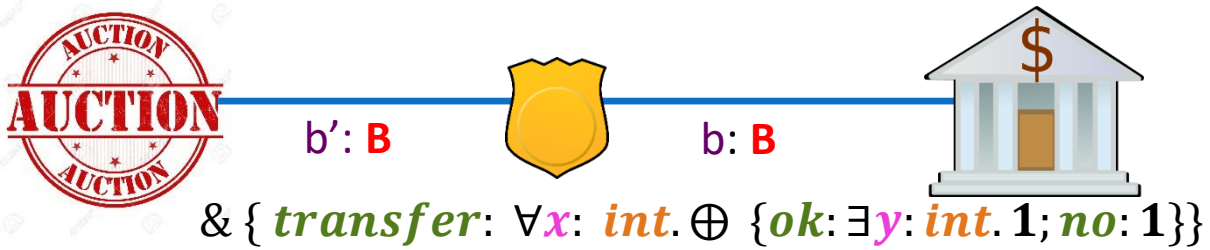
Bank Monitor Code

```
bank_id : B ← B
b' ← bank_id  b =
    case b' of
        | transfer ⇒ x ← recv b';
                     b.transfer; send b x;
                     case b of
                        | ok ⇒ y ← recv b;
                               b'.ok; send b' y;
                        | no ⇒ b'.no;
```

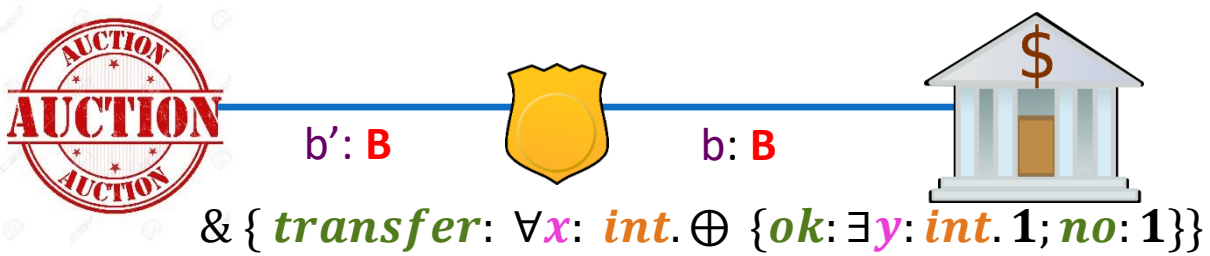


Bank Monitor Code

```
bank_id : B ← B
b' ← bank_id  b =
    case b' of
        | transfer ⇒ x ← recv b';
                     b.transfer; send b x;
                     case b of
                        | ok ⇒ y ← recv b;
                               b'.ok; send b' y;
                        | no ⇒ b'.no;
```



Bank Monitor Code




```
bank_id : B ← B
b' ← bank_id  b =
    case b' of
        | transfer ⇒ x ← recv b';
                     b.transfer; send b x;
                     case b of
                        | ok ⇒ y ← recv b; assert (y = x);
                                b'.ok; send b' y;
                        | no ⇒ b'.no;
```

More Stateful 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
List of integers is in ascending order	Previous integer
Factoring procedure outputs integers that multiply to input (n = p * q)	Input integer (n)

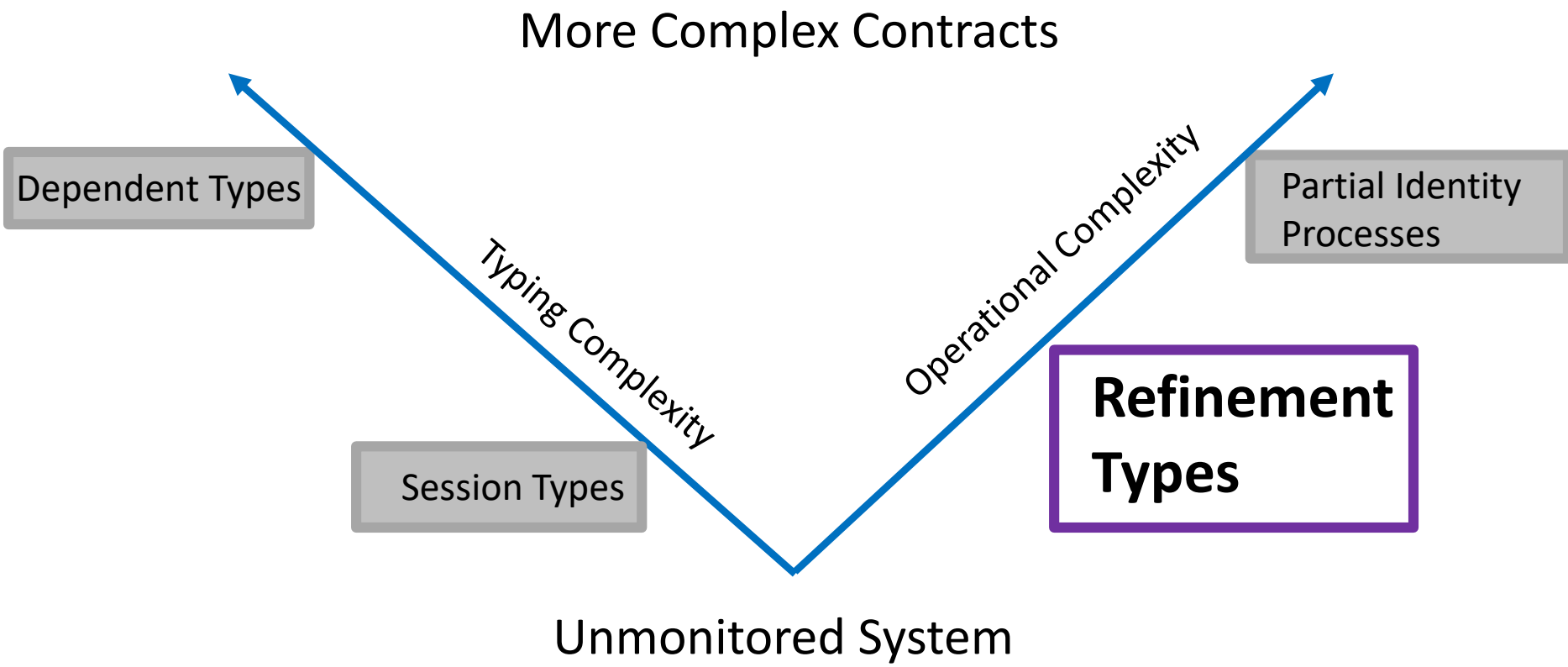
Partial Identity Criterion

- Provide rules for verifying that processes are partial identities
- Prove that any process satisfying criterion is indeed a partial identity processes
- Complications occur when monitors spawn other monitors (higher order case)

Theoretical Contributions

- Transparency
  - Partial identity criterion guarantees that monitors do not change system behavior for well-behaved processes
- Safety
  - Prove a safety theorem for a fragment of this class of contracts

Contract Classes



Refinement Fragment

- Can generate partial identity monitors to check type refinements
- Encode refinements as type casts and translate them to monitors
- Refinements on base types and labels

Refinement Examples

$$\{n : int \mid n > 5\} \Leftarrow \{n : int\}$$

integer refinement

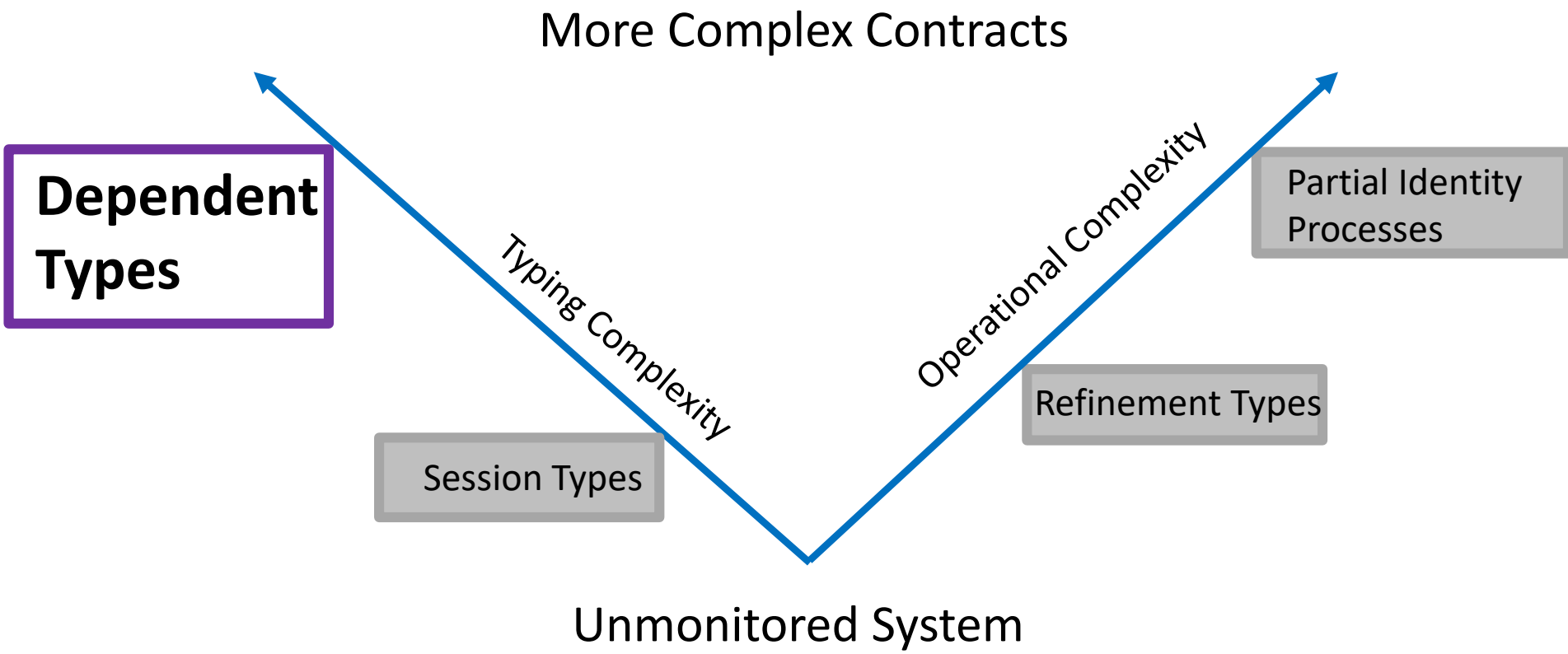
$$\oplus\{cons : \exists y : int.\, list\} \Leftarrow \oplus\{cons : \exists y : int.\, list; nil : \mathbf{1}\}$$

label refinement

Theoretical Contributions

- Safety
  - Generated refinement monitors are well-typed
- Transparency
  - Generated refinement monitors are partial identities

Contract Classes

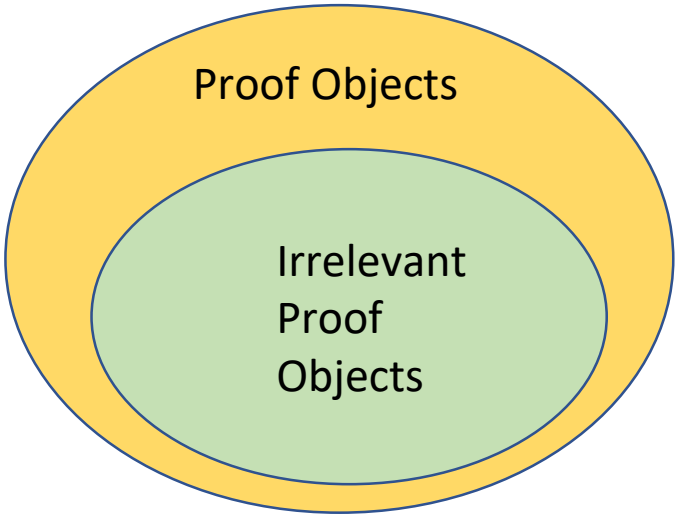


Dependent Types

- Dependent types are encoded with proof objects
- Proof objects must be generated and sent through the system
- Requires significant infrastructure (Proof Carrying Code, Necula 1997)

Proof Irrelevance

- Some proof objects play no computational role in the program
- We can erase irrelevant proof objects in order to avoid sending them (Pfenning et al 2011)



Factoring Type

$$\forall n{:}\ \mathit{int}.\ \exists p{:}\ \mathit{int}.\ \exists q{:}\ \mathit{int}.\ \mathbf{1}$$

Proof Object

$$\forall n{:}\ int.\ \exists p{:}\ int.\ \exists q{:}\ int.\ \exists s{:}\ (n = p * q).\ \mathbf{1}$$

s is a proof object!

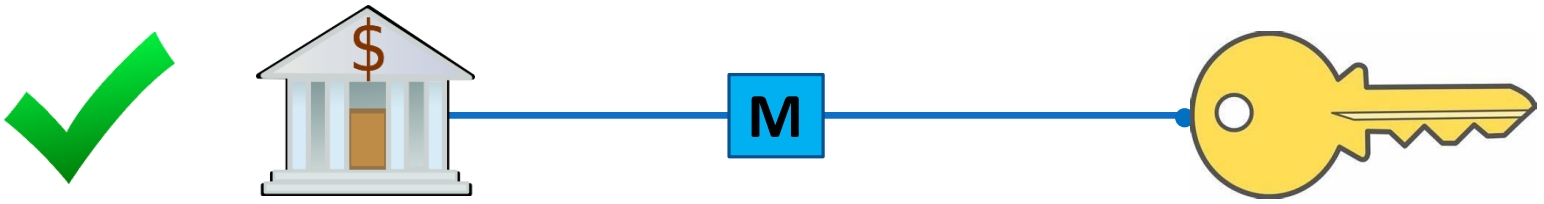
Irrelevant Proof Object

$$\forall n{:}\ int.\ \exists p{:}\ int.\ \exists q{:}\ int.\ \exists s \div [n = p * q].\ \mathbf{1}$$

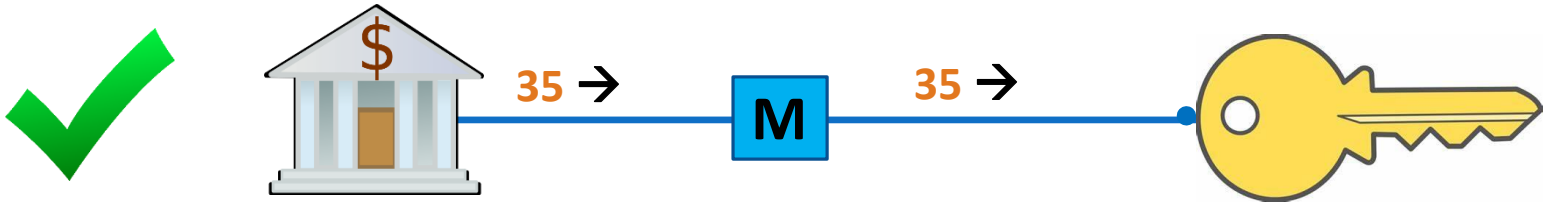
s is an **irrelevant** proof object!

Can construct a proof for **n = p * q**

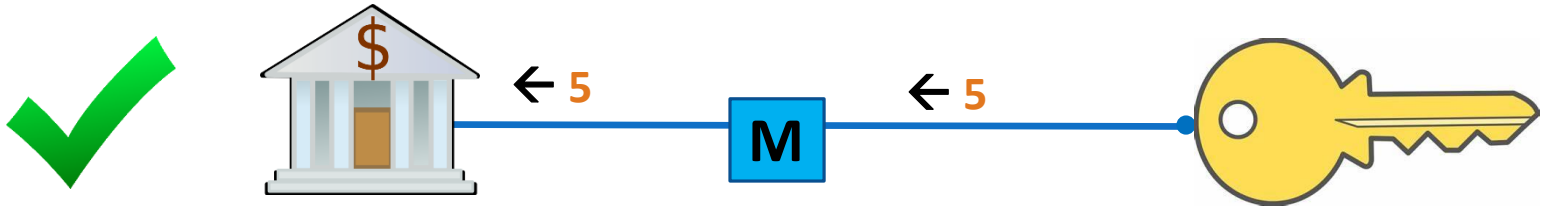
Irrelevant Factoring Monitor



$$\forall n{:}\ int.\exists p{:}\ int.\exists q{:}\ int.\exists s \div [n = p * q].\mathbf{1}$$

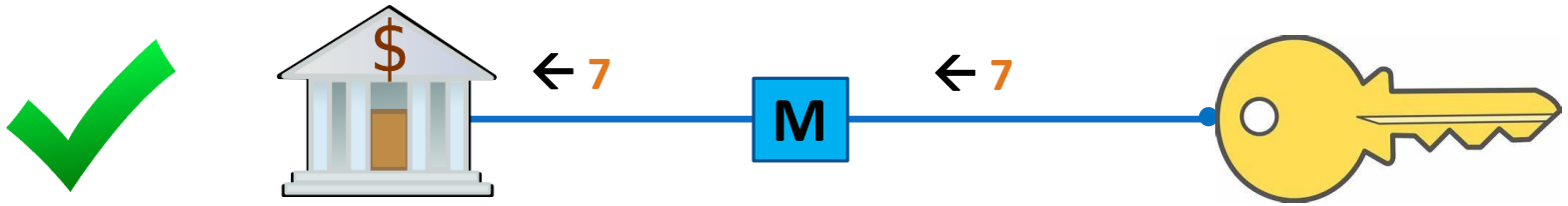


$$\exists p{:}\ int.\exists q{:}\ int.\exists s \div [n = p * q].\mathbf{1}$$



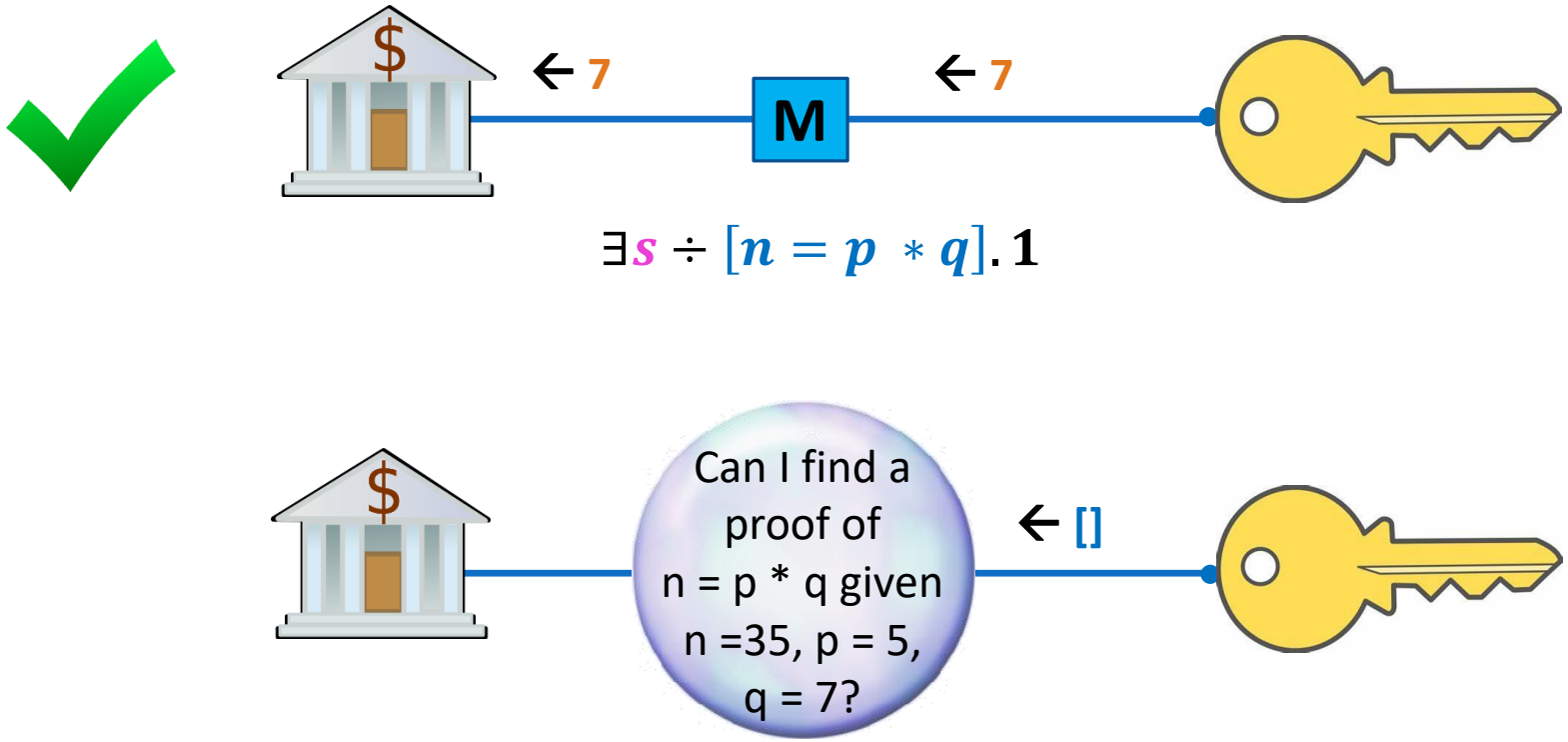
$$\exists q{:}\ int.\exists s \div [n = p * q].\mathbf{1}$$

Irrelevant Factoring Monitor

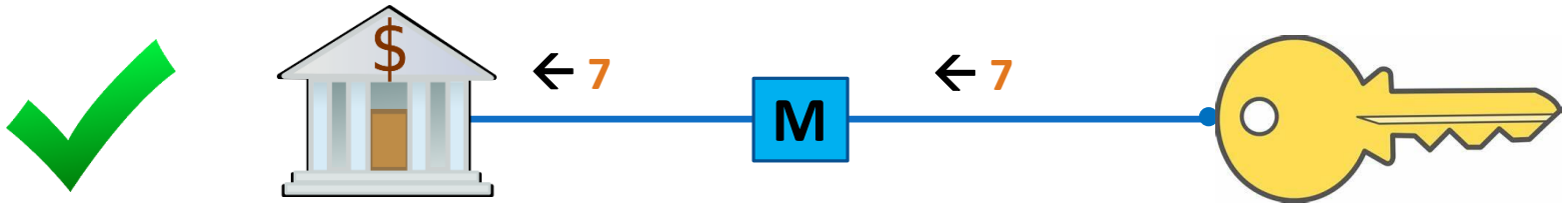


$$\exists s \div [n = p * q].1$$

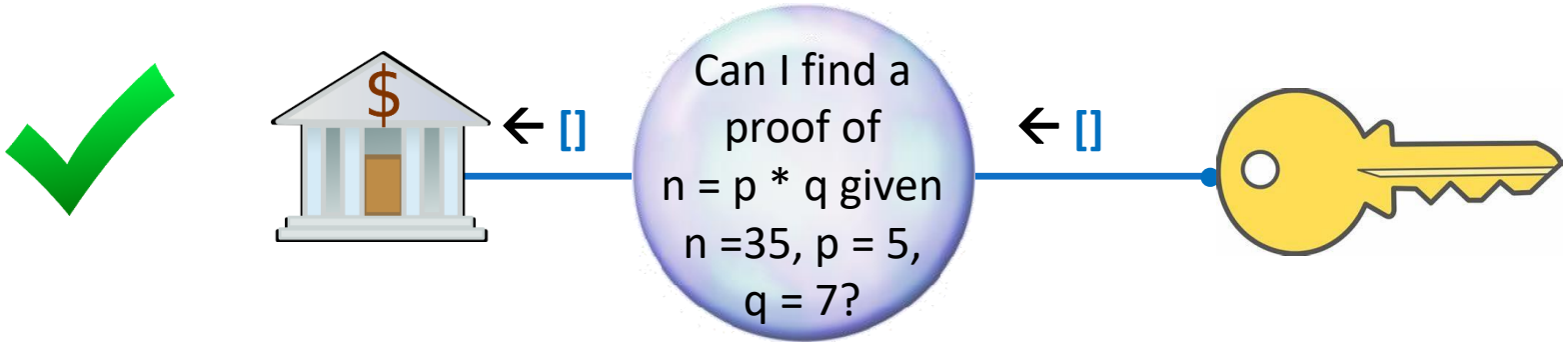
Irrelevant Factoring Monitor



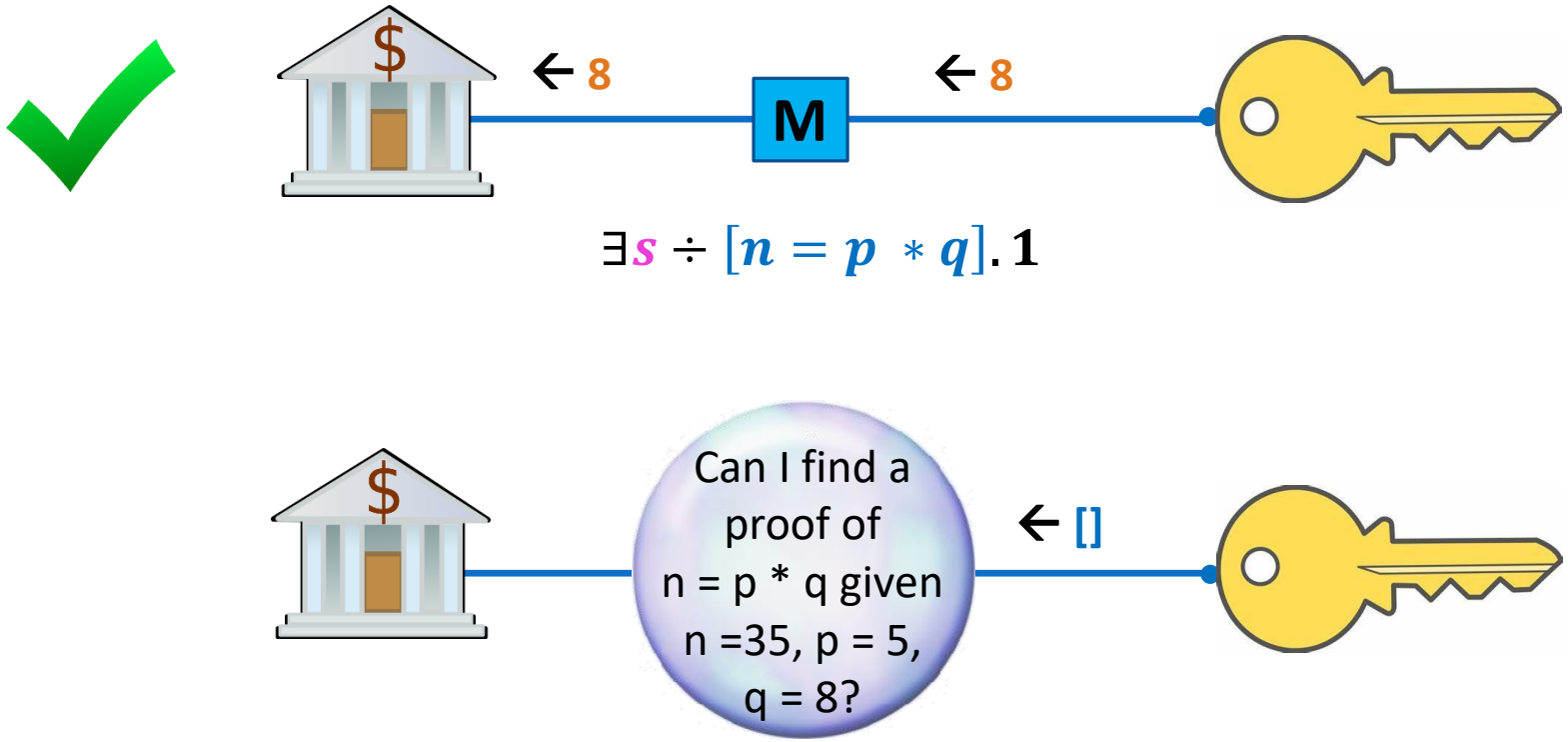
Irrelevant Factoring Monitor



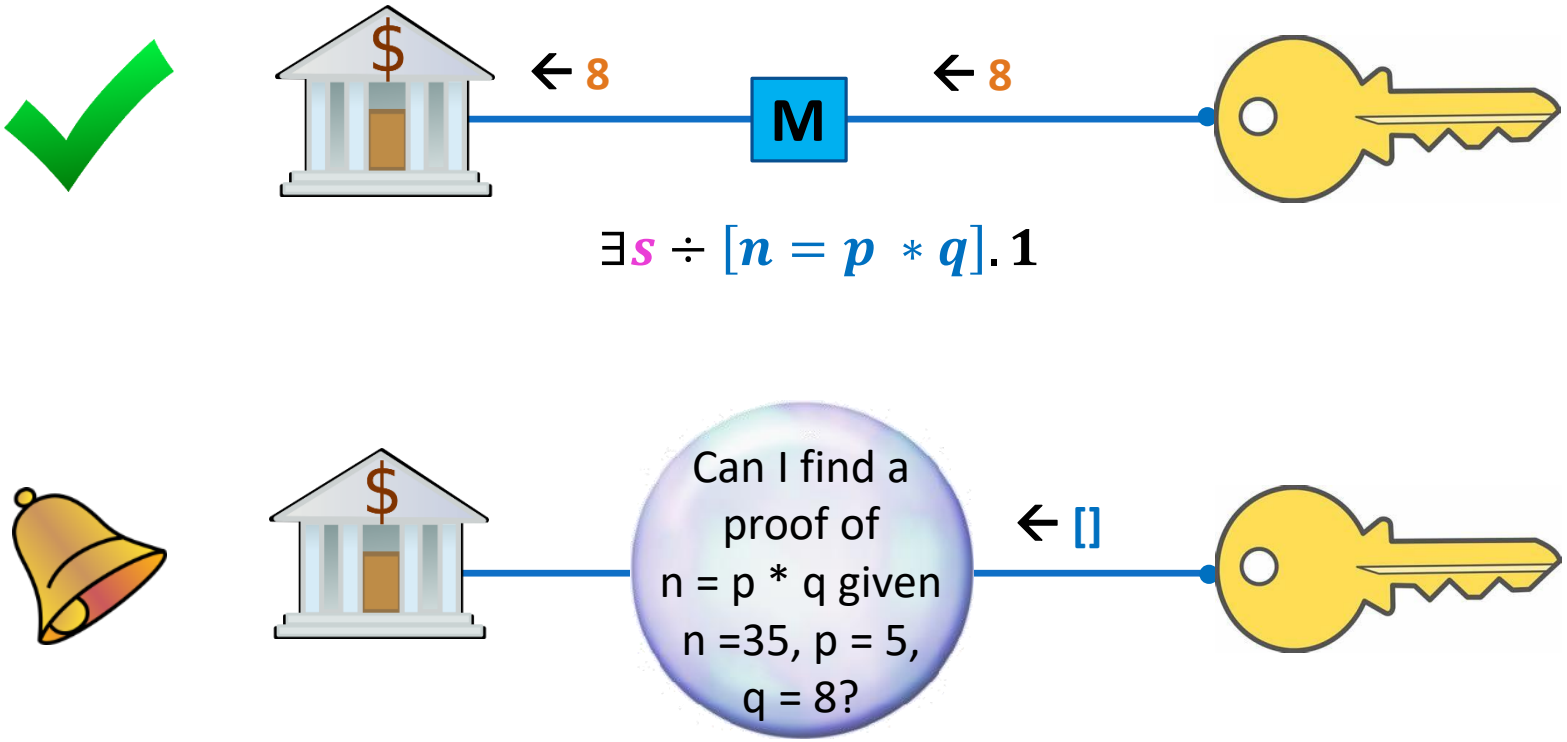
$$\exists s \div [n = p * q].1$$



Irrelevant Factoring Monitor

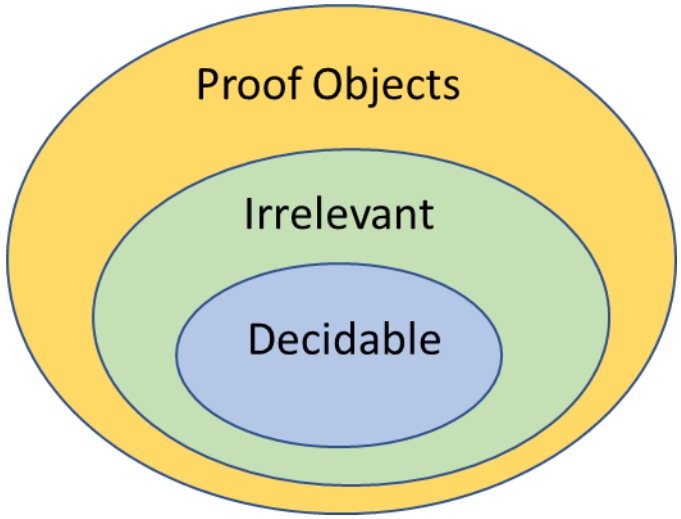


Irrelevant Factoring Monitor



Decidability

- The monitor will only be able to construct a proof for irrelevant objects that are in a decidable fragment
- Example: Pressburger arithmetic



“Undecidable” Factoring

$$\forall n\colon \mathit{int}.\, \exists s \div [\exists p\colon \mathit{int}.\, \exists q\colon \mathit{int}.\, n = p * q].\, \mathbf{1}$$

- Monitor cannot construct proof for the proof object even though it is irrelevant
- The actual proof must be sent through the system

Erasure

- Formally verify that irrelevant terms do not need to be computed
- Perform erasure on types, contexts, and processes
- Relevant computation cannot depend on irrelevant computation

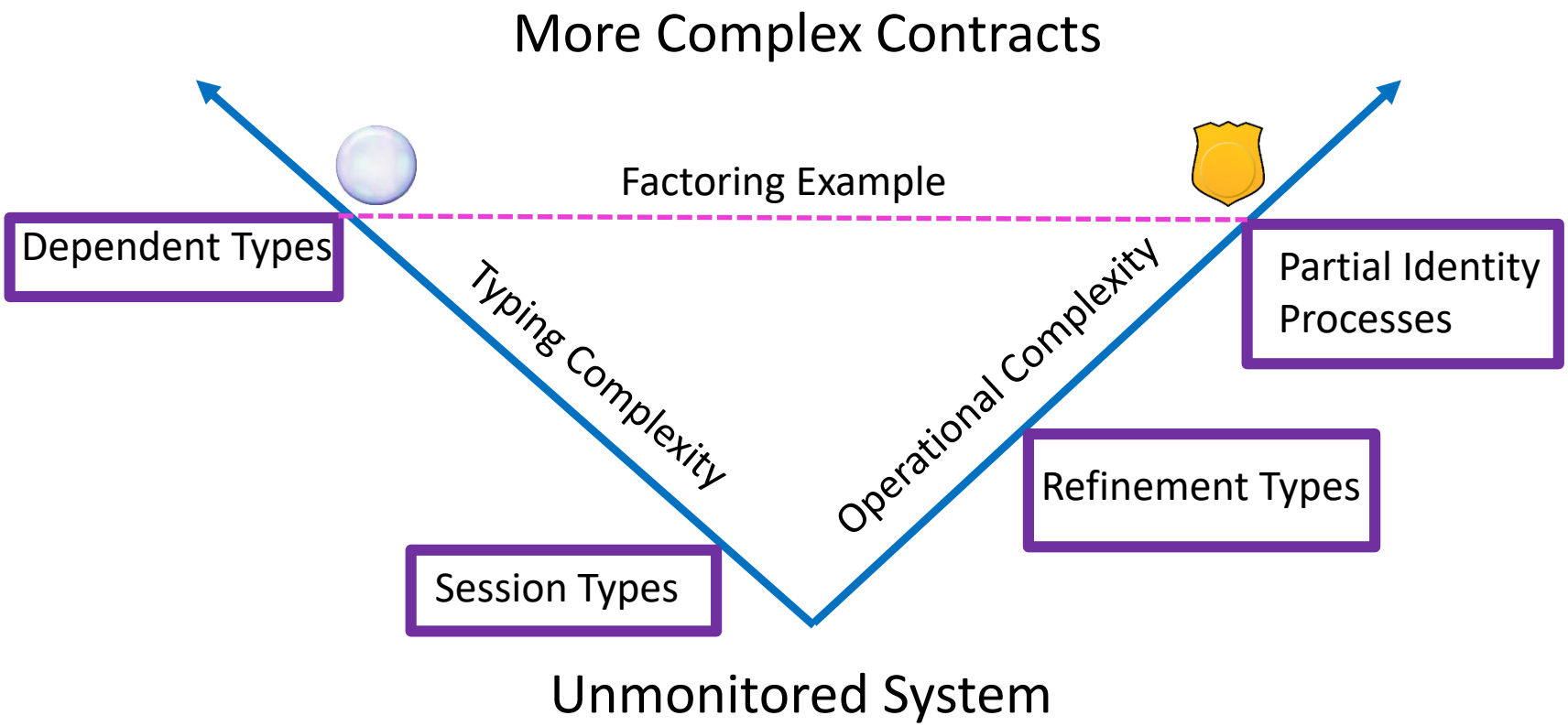
Theoretical Contributions

- Safety
  - Correctness of Erasure
    - If a process is well-typed, the erased version is also well-typed
    - Erasure and process execution commute
  - Correctness of Blame Assignment
- Transparency

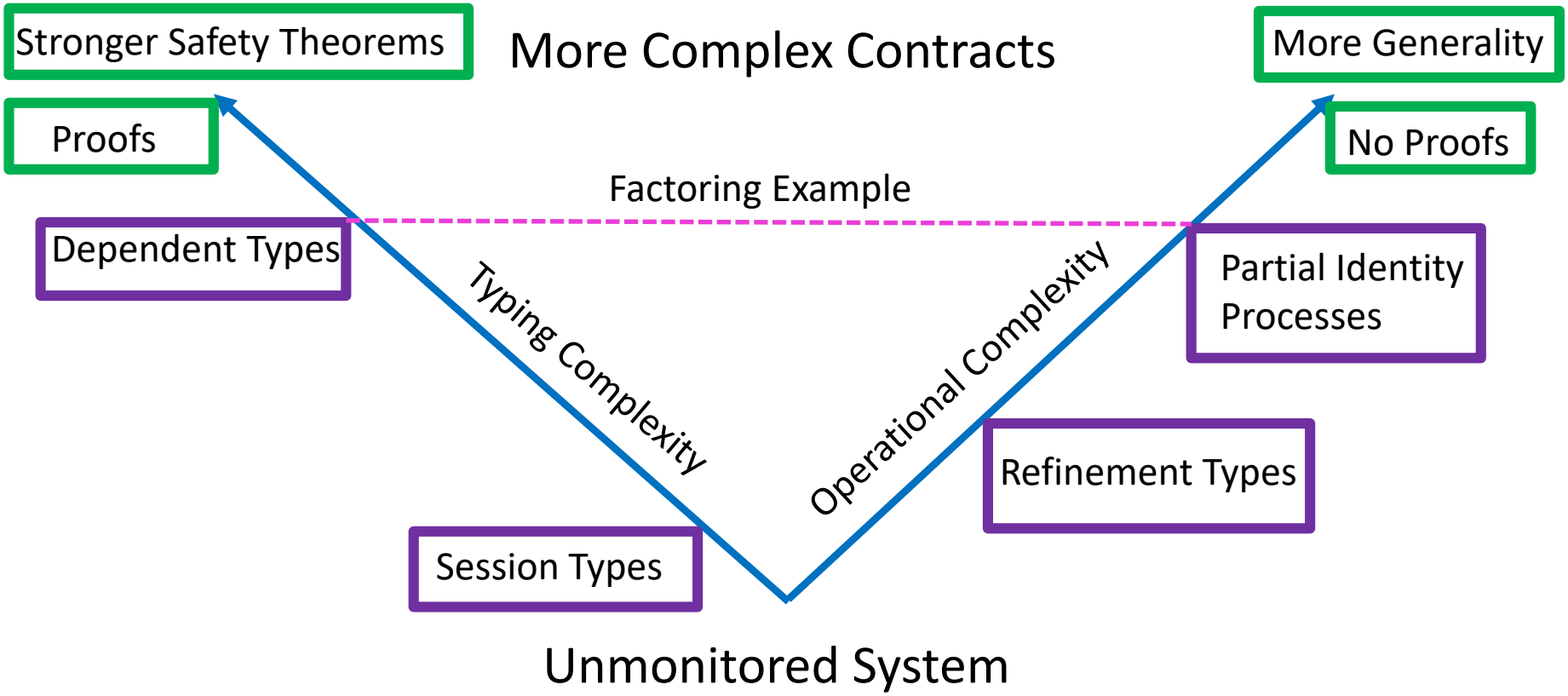
Thesis Statement

Session-typed monitors give rise to novel techniques for dynamically monitoring expressive classes of concurrent contracts and provide strong theoretical guarantees.

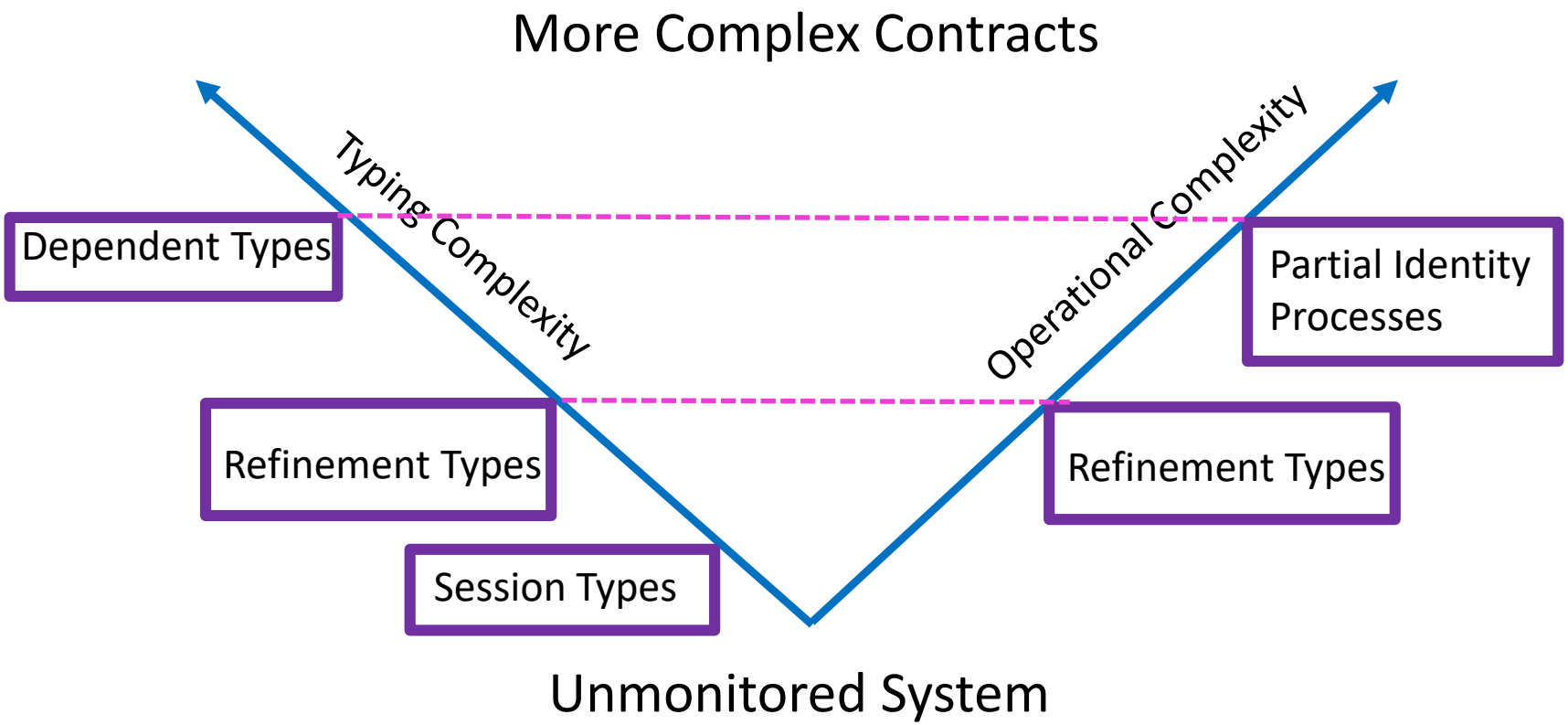
Contract Classes



Contract Classes



Refinement Types?



Refinements are Special

- Handled by dependent-typed monitors
- Can also generate partial-identity monitors for type refinements
  - This method allows refining based on labels not only base types

$$\oplus\{\,\boldsymbol{cons}\colon \exists \boldsymbol{y}\colon \boldsymbol{int}.\,\boldsymbol{list}\} \Leftarrow \oplus\{\,\boldsymbol{cons}\colon \exists \boldsymbol{y}\colon \boldsymbol{int}.\,\boldsymbol{list};\ \boldsymbol{nil}\colon \mathbf{1}\}$$

Future Work

- Partial-identity monitors for shared channels with a copying semantics
- Monitoring deadlock in a semantics with shared resources
- Monitoring information flow and other 2-trace properties