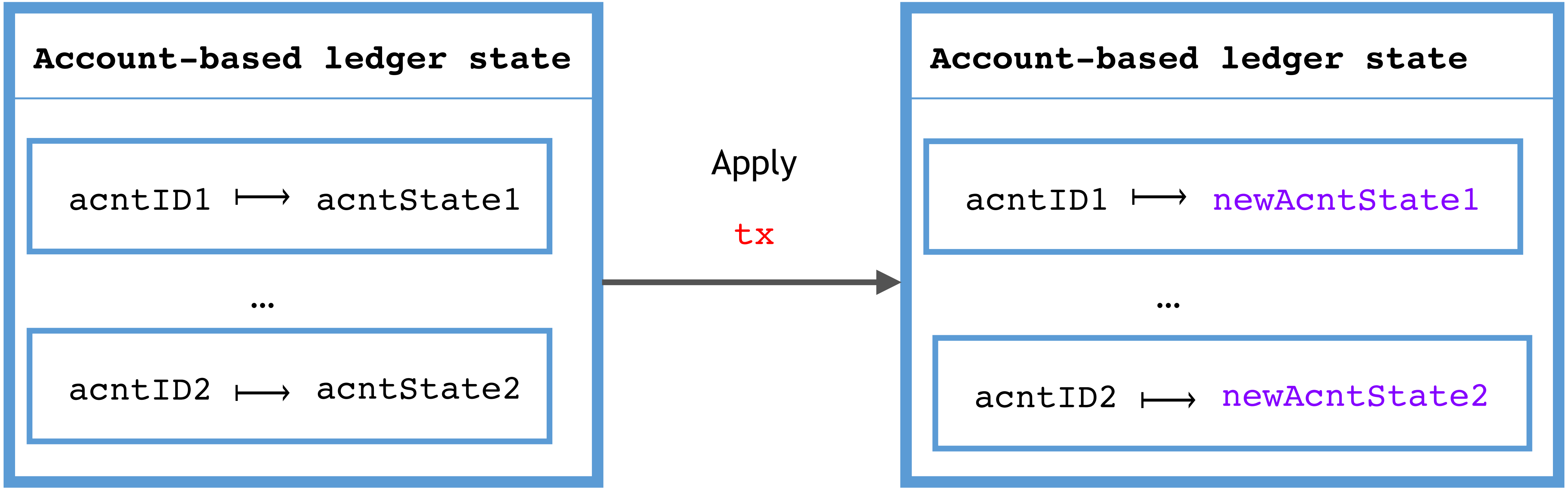
INPUT | OUTPUT

Structured Contracts on Cardano

Statefulness in the EUTxO model

Polina Vinogradova, **Orestis Melkonian**, Philip Wadler, Manuel Chakravarty, Jacco Krijnen,
Michael Peyton Jones, James Chapman, Tudor Ferariu

Account-based Ledgers



EUTxO Ledger

```
UTxO set

txin1 ↦ (myScriptAddr1, value1, datum1)

 …

txin2 ↦ (myScriptAddr2, value2, datum2)
```

`txin = (txId, ix)`
- Pointer to a specific output of transaction `tx`

`txID`
- Encoding of the transaction `tx` whose output `txin` points to

`ix`
- Index of corresponding output of `tx` in its list of outputs

EUTxO Ledger

UTxO set

```
txin1 ↦ (myScriptAddr1, value1, datum1)

 …

txin2 ↦ (myScriptAddr2, value2, datum2)
```

Script

- Stateless user-defined code with a boolean output
- Executed when a transaction spends the UTxO entry

EUTxO Ledger

UTxO set

```
txin1 ↦ (myScriptAddr1, value1, datum1)

 …

txin2 ↦ (myScriptAddr2, value2, datum2)
```

Asset bundle

- A mix of different tokens

Script

- Stateless user-defined code with a boolean output
- Executed when a transaction spends the UTxO entry

EUTxO Ledger

UTxO set

```
txin1 ↦ (myScriptAddr1, value1, datum1)

 …

txin2 ↦ (myScriptAddr2, value2, datum2)
```

Datum
- Some user-specified data

Asset bundle
- A mix of different tokens

Script
- Stateless user-defined code with a boolean output
- Executed when a transaction spends the UTxO entry

EUTxO Ledger Update Specification

Using small-step operational semantics

Environment	Ledger start state	Input	Ledger end state

`Slot s`	**`UTxO set`**
	`txin_1 ↦ …`
	`…`
	`txin_k ↦ …`
		`tx` →	**`UTxO set`**
			`txin_a ↦ …`
			`…`
			`txin_k ↦ …`

Check:

- `s ∈ validityInterval (tx)`
- Check signatures
- Run validator scripts
- Run minting policies
- Check UTxOs being spent exist
- etc.

EUTxO

Challenges :

- Non-conventional programming **paradigm**
- Programming in **stateless** predicates

Advantages :

- **Predictable**
  - gas cost
  - outcome of contract execution
  - ledger changes made by valid transaction
- Amenable to **formal verification**

Examples :

- Cardano
- Ergo

Account-based

Challenges :

- **Can have unpredictable**
  - gas cost
  - outcome of contract execution
  - ledger changes made by valid transaction
- Formal verification is harder

Advantages :

- **Familiar** programming paradigm
- Straightforward use of **account states**

Examples :

- Ethereum
- Tezos

Motivation : Simulating Accounts

- **Account ID**
- **State** :
  - owner, assets
- **API** :
  - withdraw, deposit, open, close, transfer

EUTxO implementation :
- How do we **specify** this?
- What does it mean to **implement this program** using stateless predicates on transaction data?
- How can we be sure distinct implementations **meet the same specification**?

Motivation : Simulating Accounts

- **State** :
  - unique account ID, owner, assets

- **API** :
  - withdraw, deposit, open, close, transfer

EUTxO implementation :

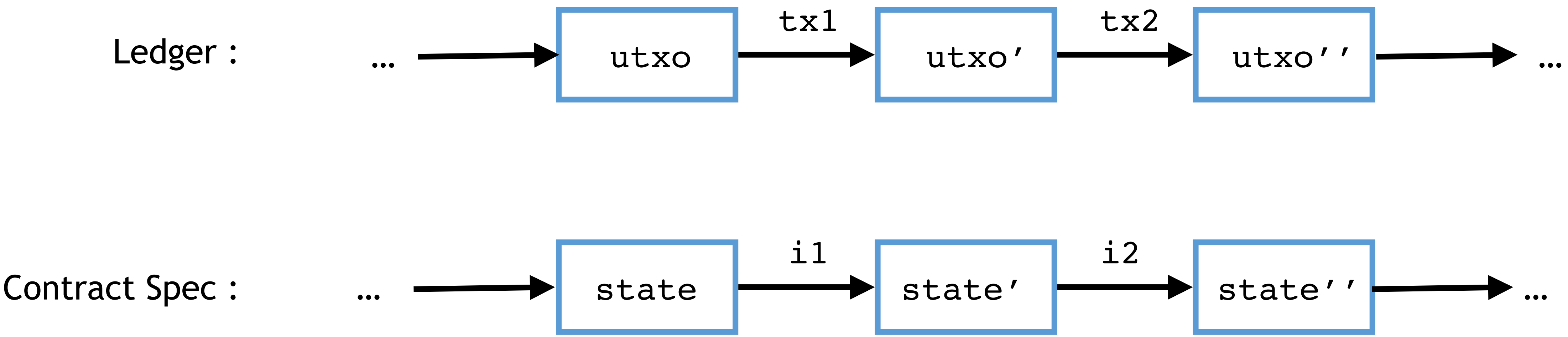
- How do we **specify** this?
- What does it mean to **implement this program** using stateless predicates on transaction data?
- How can we be sure distinct implementations **meet the same specification**?

We need a model of stateful computation here!

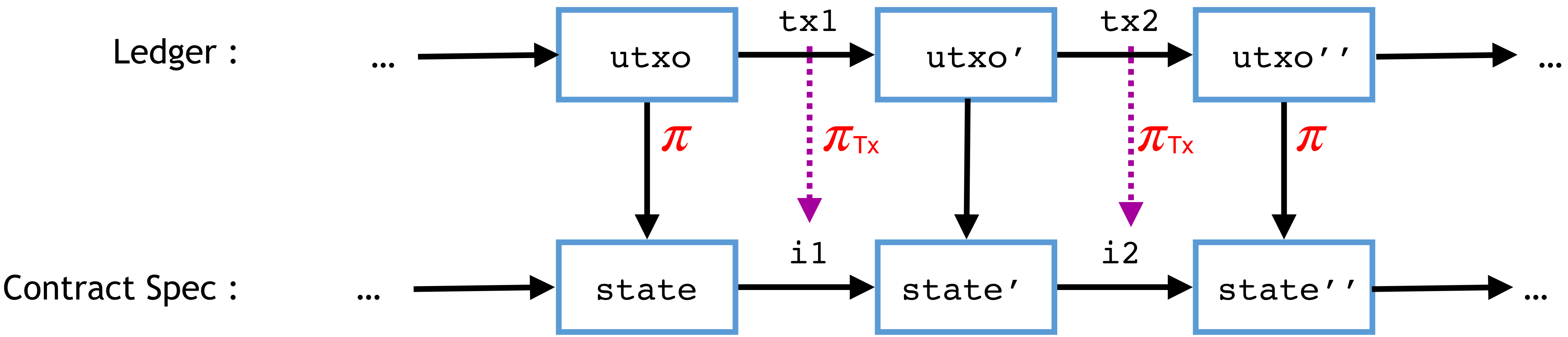
Enter “Structured Contracts”

Ledger : $\dots \longrightarrow \boxed{\texttt{utxo}} \xrightarrow{\texttt{tx1}} \boxed{\texttt{utxo'}} \xrightarrow{\texttt{tx2}} \boxed{\texttt{utxo''}} \longrightarrow \dots$

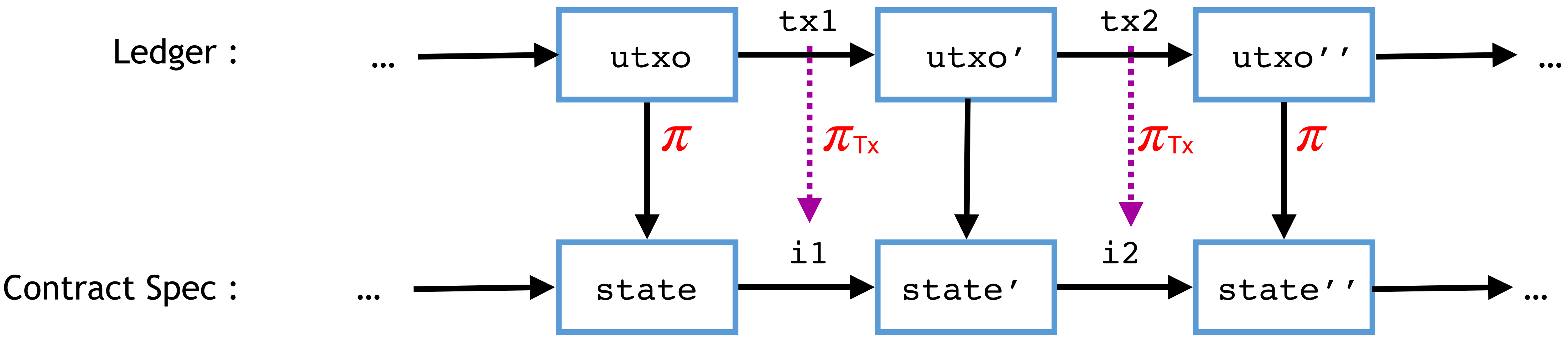
Enter “Structured Contracts”



Enter “Structured Contracts”



Enter “Structured Contracts”



An **instance** of a structured contract requires :

- **Specification** (in small-step operational semantics)
- **Projections** π (partial function), π_{Tx}
- **Proof** of commutativity of any square

Example 1 : NFT

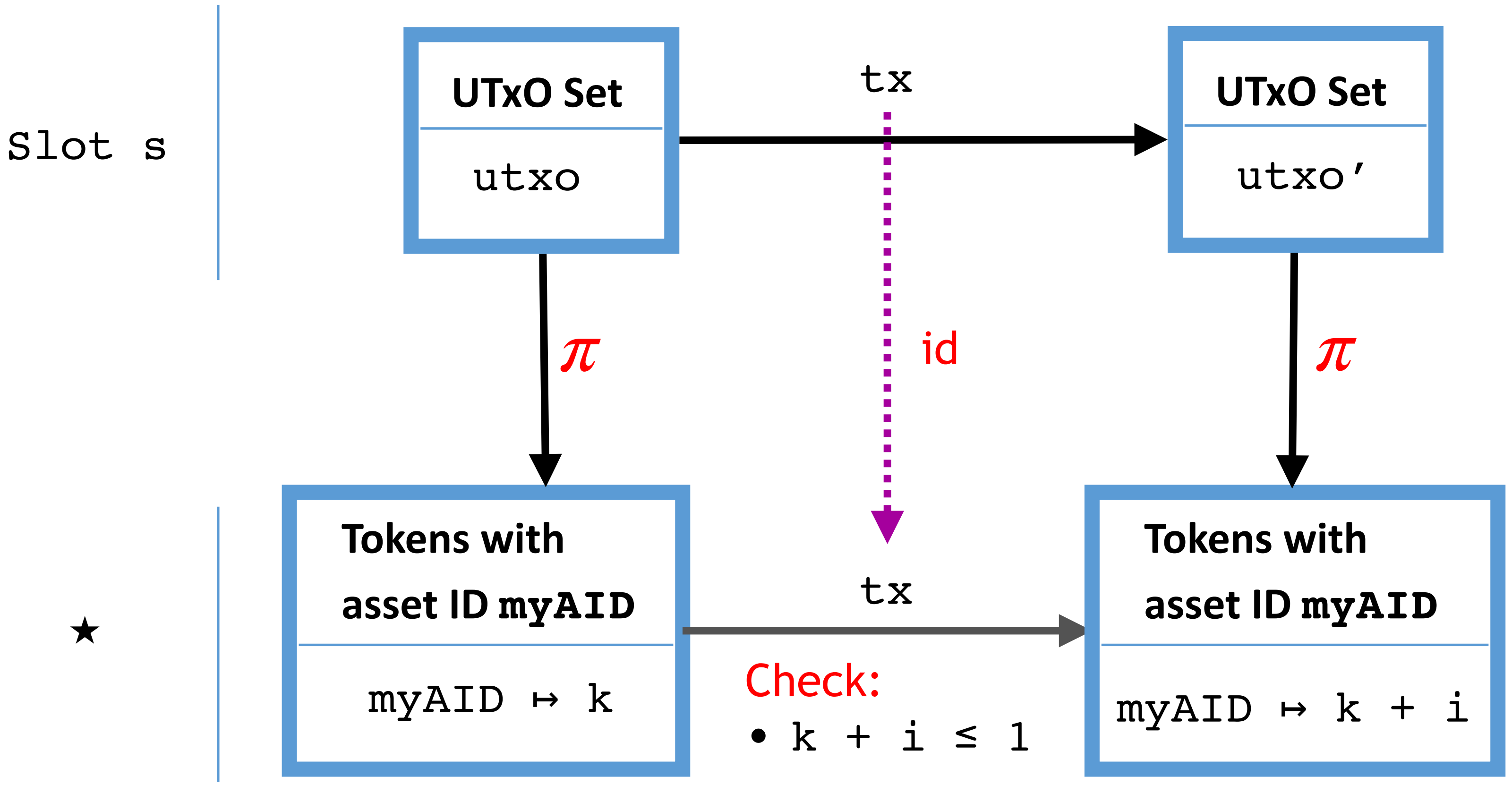
Defining property :

- “If one exists on the ledger, another one cannot be minted”
- suggests a state transition system
- can specify and implement

Environment	Contract state	Input	Contract state
★	**Tokens with asset ID `myAID`** `myAID` $\mapsto$ `k`	`tx` $\longrightarrow$ Check: • `k + i ≤ 1`	**Tokens with asset ID `myAID`** `myAID` $\mapsto$ `k + i`

`i` := # tokens with `myAID` in `mint (tx)`

NFT Implementation

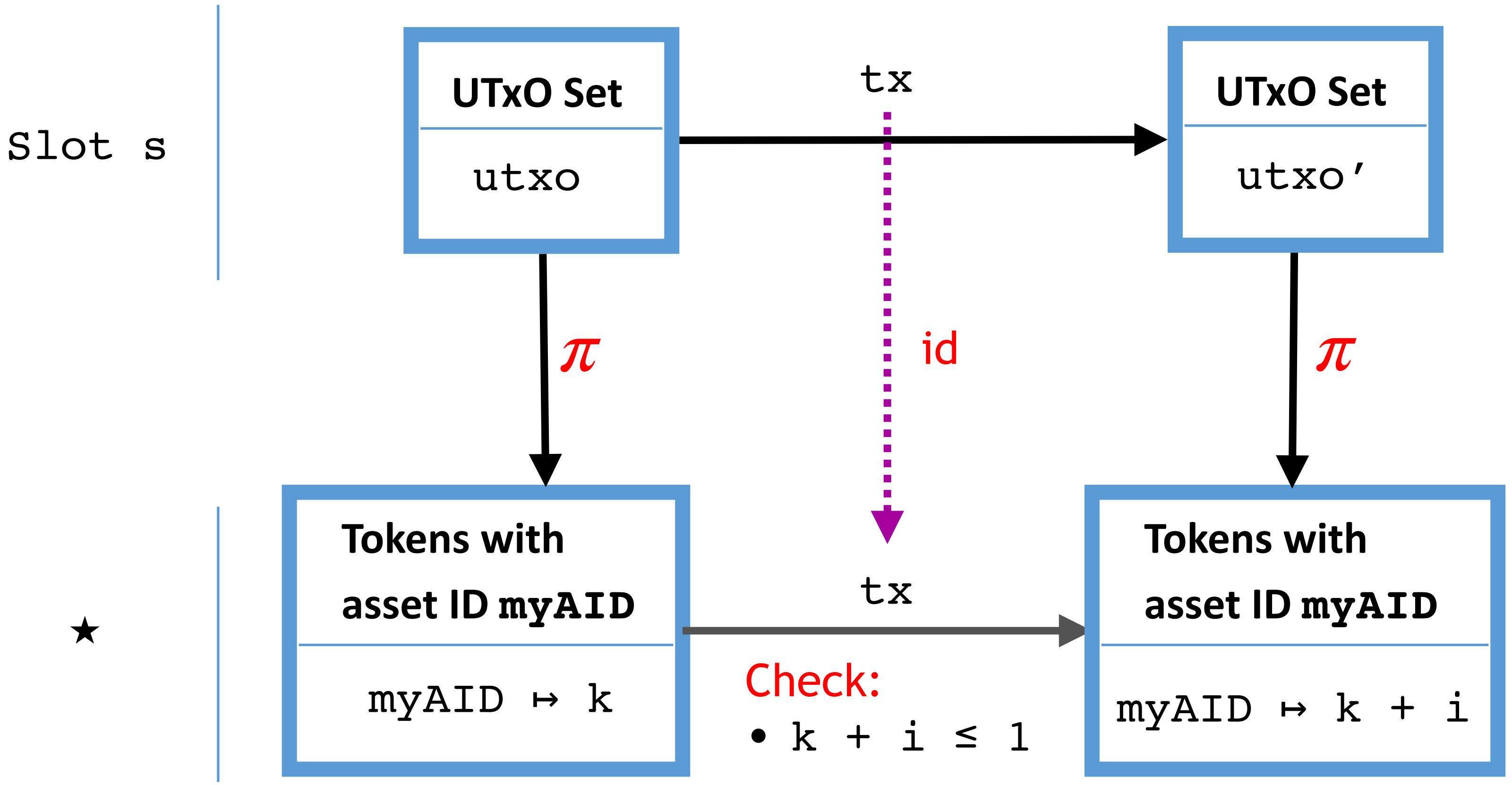


i := # tokens with `myAID` in `mint (tx)`

Need to :

- Define π
- Prove square commutes

NFT Implementation



π (utxo) :=

myAID ↦ k if (k ≤ 1) & …

★ otherwise

where

k := # tokens with `myAID` in `utxo`

i := # tokens with `myAID` in `mint (tx)`

NFT Implementation

Proving correctness

`myAID` includes to a **minting policy**, which checks :

- A **specific UTxO entry** is being spent by `tx`
- The **quantity** of assets with `myAID` being **minted** is 1

π (utxo) :=

$$\begin{cases} \texttt{myAID} \mapsto \texttt{k} & \text{if } (\texttt{k} \leq 1) \;\&\; \ldots \\ \star & \text{otherwise} \end{cases}$$

where

`k` := # tokens with `myAID` in `utxo`

To prove commutativity :

- assume **replay protection**
- **exclude** the case where π (utxo) = $\star$
  - starting UTxO has at **most 1 token** with `myAID`

NFT Defining Property

- From definition of π, we have :

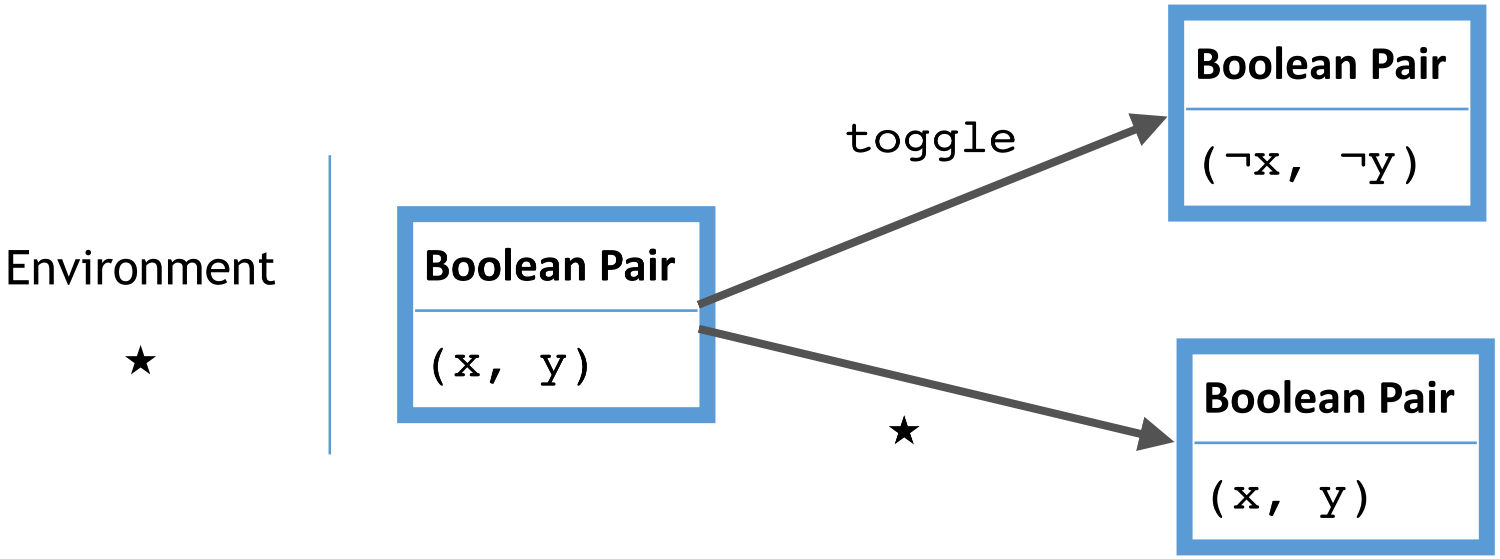
For any $\mathtt{utxo}$,

$$\pi\,(\mathtt{utxo}) \neq \star \quad \Rightarrow \quad \pi\,(\mathtt{utxo}) \leq (\mathtt{myAID} \mapsto 1)$$

- **Commutativity of square** implies that

$$\pi\,(\mathtt{utxo}) \neq \star \quad \Rightarrow \quad \pi\,(\mathtt{utxo'}) \leq (\mathtt{myAID} \mapsto 1)$$

Example 2 : TOGGLE



TOGGLE Implementations

Naive

UTxO set
txin ↦ (toggleVal, NFTpointer, (x, y))

Distributed

UTxO set
txin_x ↦ (toggleVal', NFTpointerX, x)
txin_y ↦ (toggleVal', NFTpointerY, y)

TOGGLE Implementations

Naive

UTxO set
txin ↦ (toggleVal, NFTpointer, (x, y))

Distributed

UTxO set
txin_x ↦ (toggleVal’, NFTpointerX, x)
txin_y ↦ (toggleVal’, NFTpointerY, y)

Not pictured here : `NFTpointer`, `NFTpointerX`, and `NFTpointerY` policy code, `toggleVal` code, and commutativity proof obigation

TOGGLE Implementations

Naive

UTxO set

txin ↦ (toggleVal, NFTpointer, (x, y))

Distributed

UTxO set

txin_x ↦ (toggleVal’, NFTpointerX, x)

txin_y ↦ (toggleVal’, NFTpointerY, y)

- Both implement the **same spec**
- Developers can **compare** implementations across memory use, parallelizability, etc.

Structured contracts (SCs)

As a model of stateful computation on the EUTxO ledger

- **Generalization** of constraint-emitting machines (CEMs), in which :
  - projections π, π_{Tx} are **fixed**
  - implementations are **fixed** and **automatically generated**
- **Principled, uniform** approach to reasoning about stateful computation
- SCs define a class of **all stateful contracts**
  - that can be implemented via **user-defined scripts**
  - where correct **ledger** evolution $\Rightarrow$ correct on-chain **contract state** evolution
- Enable **comparison** of implementations if a given spec

Structured contracts

Limitations

- **No automation** for implementation of simulation proof
  - difficult b/c user decides on the implementation
  - Future work

- Hard to guarantee **existence of valid transaction** corresponding to given state update
  - Even more difficult **in practice** : user has no control over UTxO state, slot, fees, etc. that their transaction will actually be applied to
  - Also future work!

Structured contracts

Mechanized in Agda

https://omelkonian.github.io/structured-contracts/