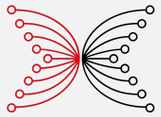


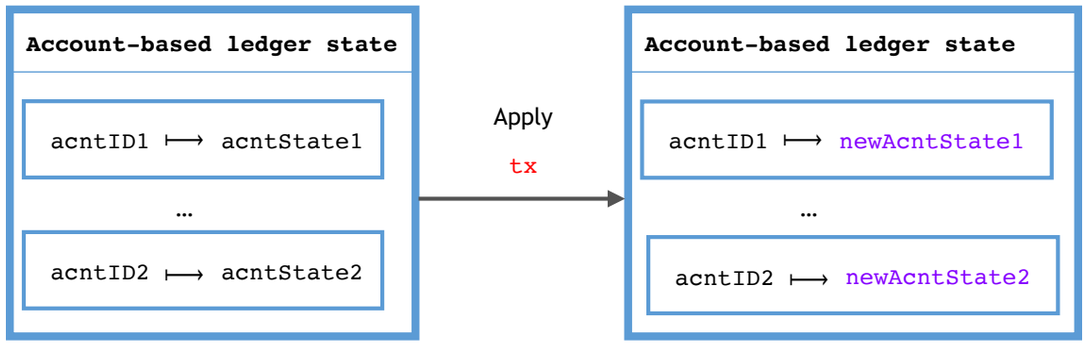
INPUT | OUTPUT

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



- Programming account-based ledgers is concerned with specifying how the state of an account is updates as a result of applying a transaction

EUTxO Ledger

UTxO set

```
txin1 ↦ (myScriptAddr1, value1, datum1)
 …
txin2 ↦ (myScriptAddr2, value2, datum2)
```

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

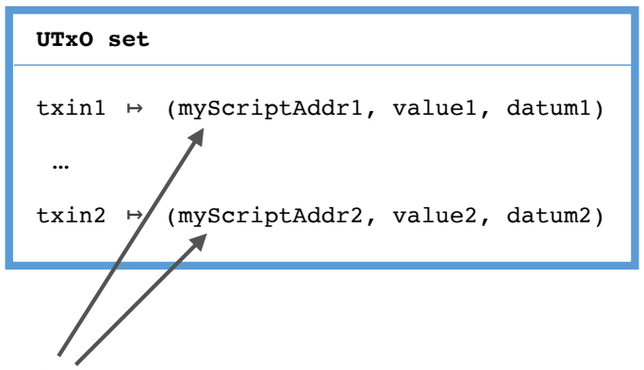
`txID`
- Encoding of the transaction `tx` whose output `txin` points to

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

- Programming an extended UTxO ledger follows a different paradigm, to explain which I need to first discuss the structure of the UTxO set

- An output reference “txin” is a unique identifier of an entry in the UTxO set
- It is a pair of txid, which is an (encoding of) the transaction which added the entry to the UTxO, and ix, which is the index of an output in the list of outputs of that transaction
- The output for which txin is a reference is the output at index ix in the list of outputs of the transaction which which txid encodes

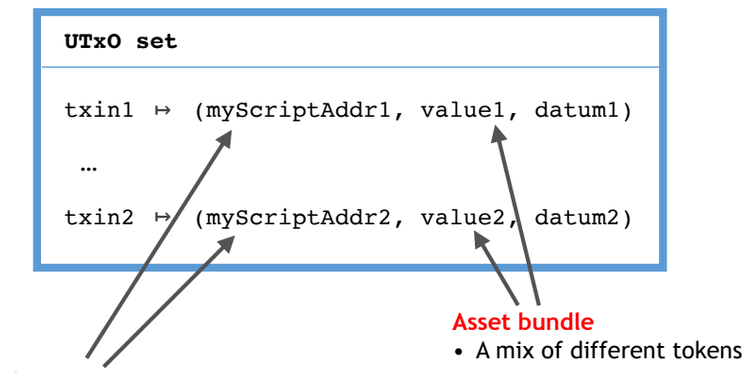
EUTxO Ledger



- The outputs in the UTxO contain a triple of the following data :
  1. a script that is run any time a transaction spends the output to determine if the script permits the transaction to do so. This script is stateless, user-defined code with a boolean output

- to implement more sophisticated programs in the EUTxO model, we must be clever about using script interactions, eg. between UTxO-locking (validator) scripts and minting policies
(Continued on next slide's notes)

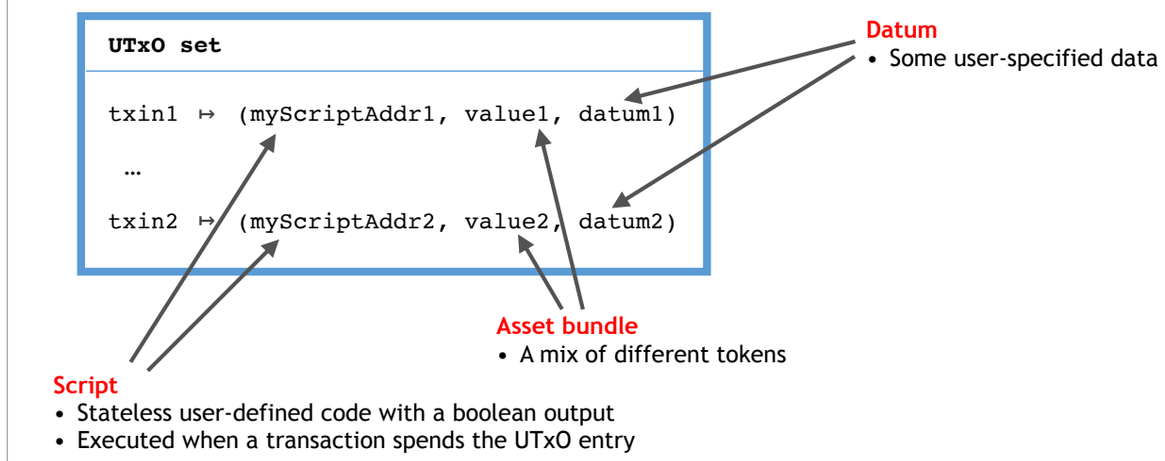
EUTxO Ledger



Asset bundle
- A mix of different tokens

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

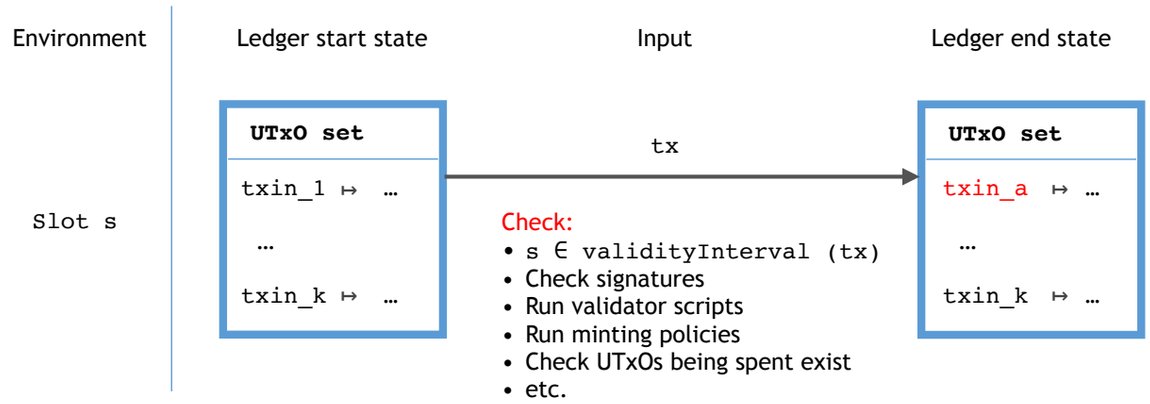
2. An asset bundle, which is a data structure containing multiple different kinds of assets, each specified in terms of its Asset IDs, and the quantity. Each asset ID includes a minting policy, which is run when the asset is minted or burned. (cont'd)



3. The datum, which is some user-specified data. Note that it is incorrect to refer to this as the “state” of the script in the output, since an UTxO entry can never be updated (only added or removed)

EUTxO Ledger Update Specification

Using small-step operational semantics



- the ledger update specification is given in terms of operational small-step semantics, where each step is the application of a transaction to a given ledger state.
- Explain the role of each component
- This is a labelled transition system, with the pair of the environment and input as the “label”
- Transition from start to end state is only permitted if the checks all pass

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

- EUTxO programming is less conventional than Account-based programming, but there are both challenges and advantages for both.
- Go over the ones mentioned on the slide

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**?

- the motivation for this work came from attempting to simulate accounts on the EUTxO ledger, which is a component required for a lot of interesting smart contract use cases, such as most financial contracts
- We specified what info an account contains, including the account ID, public key (or code) representing the owner in control of the account, and the assets stored in the account
- We specified an API
- We wanted to compare different implementations of this program

- This raised questions : how do we specify this account program in a way that we can claim that it was implemented correctly, and also allows us to have different implementations?

Motivation : Simulating Accounts

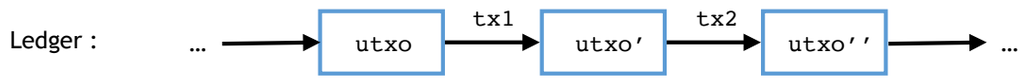
- **State** :
  - unique account ID, owner, assets
- **API** :
  - withdraw, deposit, open, close, transfer

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!

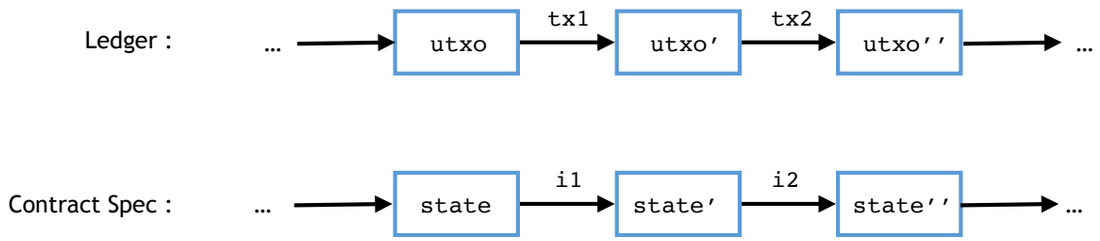
- need to have a specification of how each API call changes the state of the contract
- Need to have a model of stateful computation on the EUTxO ledger
- Need to be able to show that the specification is implemented in this model

Enter “Structured Contracts”



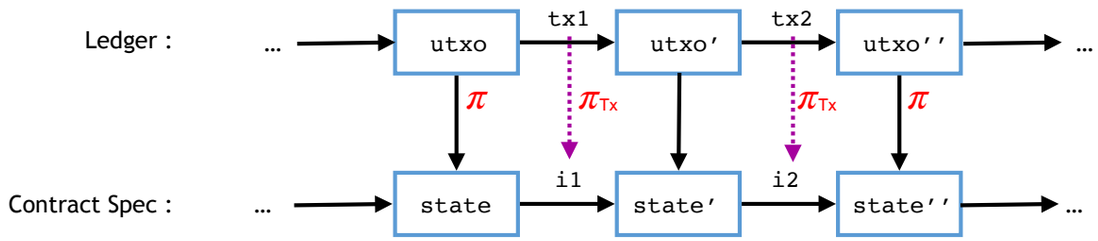
- structured contracts is a framework that, to be instantiated, requires the definition of all the components required for “stateful computation” as discussed on the previous slide. I will explain in detail what those are now
- We start with the ledger transition system specified in terms of small-step semantics

Enter “Structured Contracts”



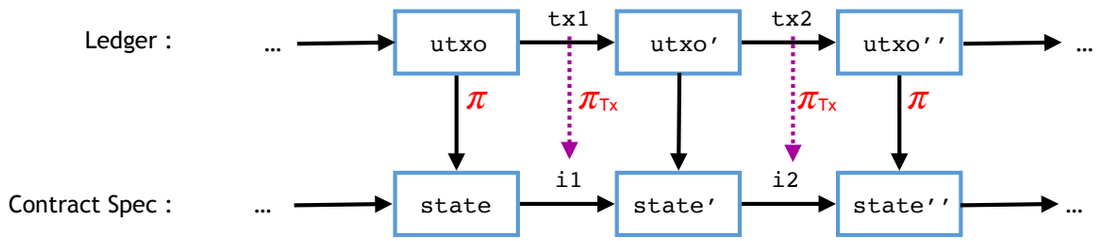
- then, we require a small-step specification of the contract itself

Enter “Structured Contracts”



- next, we need to define the relation between the two specifications.
- We need a function pi which computes the contract state from the UTxO state (e.g. picks out the UTxO containing the datum representing the contract state)
- We need a function pi_Tx which computes the input to the contract from the input to the ledger update (i.e. a transaction)

Enter “Structured Contracts”



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

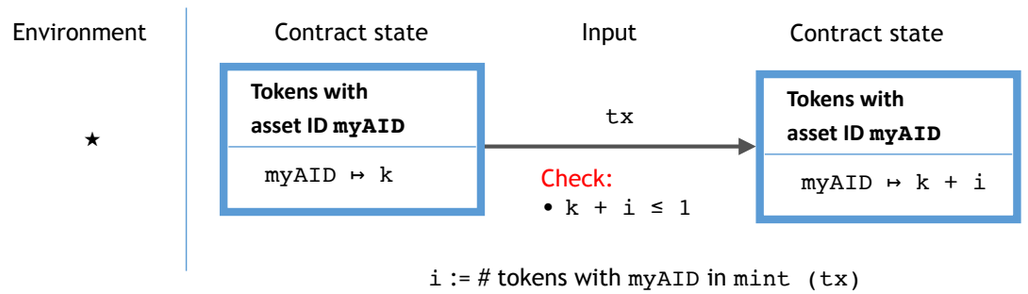
- **Specification** (in small-step operational semantics)
- **Projections** π (partial function), π_{Tx}
- **Proof** of commutativity of any square

- in addition to the specification and the projections, we require a proof of a simulation relation between the ledger transition system and the contract transition system
- This guarantees that no transaction can update the ledger state in a way that “incorrectly” changes the contract state recorded on that ledger state, that is, contrary to its specification

Example 1 : NFT

Defining property :

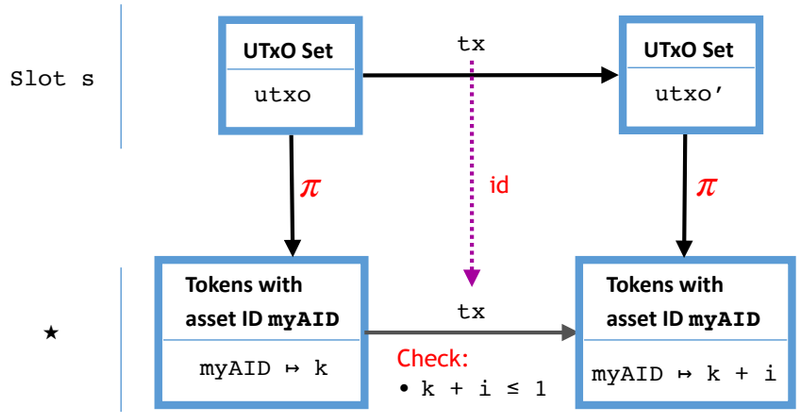
- “If one exists on the ledger, another one cannot be minted”
- suggests a state transition system
- can specify and implement



- in the paper we argue that a model of stateful computation really captures all computation of interest in the EUTxO model. This is because all user-defined scripts represent permissions for a transaction to edit some specific data stored in the ledger state, e.g. remove some UTxOs, or change the supply of some tokens. That is, the purpose of scripts is to control the evolution of some part of the ledger state.

- Our first example has to do with controlling the evolution of the amount of a specific NFT on the ledger state
- Specifying what an NFT is, or its defining property, has previously only been done in an ad-hoc way
- With a stateful specification and associated integrity proof, we give the NFT creators a template for how to construct their contract in a way that allows them to obtain a formal guarantee of the NFT defining property

NFT Implementation

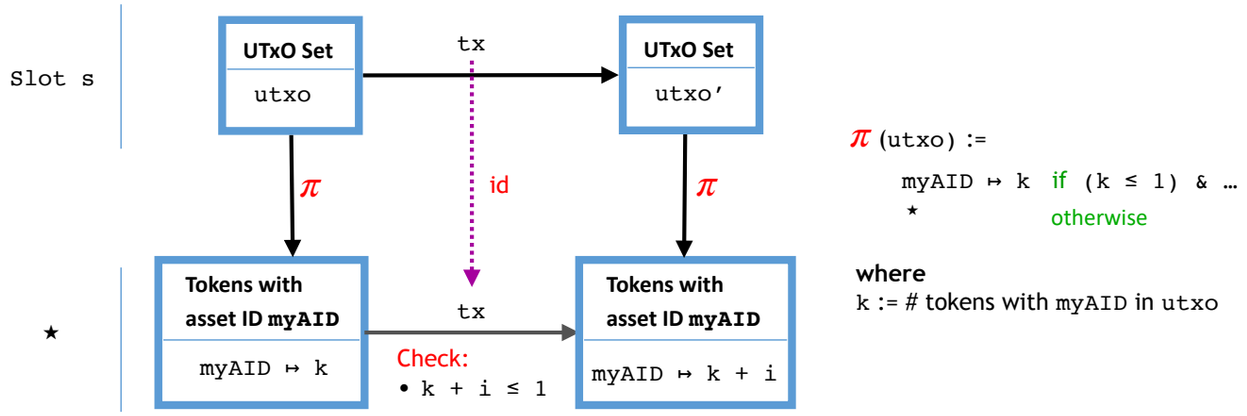


Need to :
- Define π
- Prove square commutes

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

- The specification from the previous slide is now the bottom of the square
- to instantiate the NFT structured contract, we need to define the projection pi here (note that pi_Tx is already defined, and is the identity), and demonstrate the square commutativity

NFT Implementation



π (utxo) :=

myAID ↦ k if (k ≤ 1) & …

★ otherwise

where

k := # tokens with myAID in utxo

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

- we define pi utxo to be sum of all the tokens in all UTxO entries that have the asset ID of the NFT, myAID
- Note that pi is partial, so that
- If this sum is such that the quantity of the NFT is greater than 1, pi utxo returns star
- It does not make sense to reason about how an NFT contract behaves on a ledger where there is already more than one of such NFT
-

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) :=`

`myAID ↦ k` if `(k ≤ 1) & …`

`⋆` otherwise

To prove commutativity :

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

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

* If there is not enough time, probably can skip this slide *
- Next, we must demonstrate the contract integrity
- NFT minting policy ensures that it can only be executed once by using the following trick : a specific UTxO entry must be spent for it to validate
- The fact that a utxo entry cannot re-appear in a ledger state trace is guaranteed by replay protection
- Replay protection is a UTxO ledger property that guarantees that the same transaction cannot be applied twice within a single trace of ledger states

NFT Defining Property

- From definition of π, we have :

For any `utxo`,

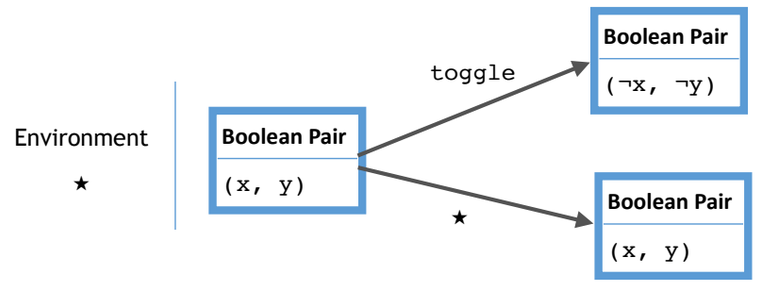
π (utxo) ≠ ⋆ ⇒ π (utxo) ≤ (myAID ↦ 1)

- **Commutativity of square** implies that

π (utxo) ≠ ⋆ ⇒ π (utxo') ≤ (myAID ↦ 1)

We can now state the defining property of the NFT contract (read slide)

Example 2 : TOGGLE



- the next example we give in the paper is the TOGGLE contract, which has a state made up of two booleans.
- When the state is updated with input *, nothing happens
- When the state is updated with input toggle, both booleans are flipped
- Toggle is an example of a contract where the update of a part of the state necessarily requires that another part of the state is also updated in a dependent way
- This is meant to conceptually resemble the constraints of transfer from one account to another - both have to be updated simultaneously

TOGGLE Implementations

Naive

UTxO set

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

Distributed

UTxO set

txin_x ↦ (toggleVal', NFTpointerX, x)

txin_y ↦ (toggleVal', NFTpointerY, y)

- we can give distinct implementations of TOGGLE and compare them
- The first implementation has both booleans stored in the datum of a single output, marked with a special NFT token
- The second implementation splits the booleans into separate outputs, each pointed to by a special NFT

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 obligation

- In both cases the output-locking scripts (toggleVal or toggleVal'), and the the NFT minting policies can be defined to ensure correct behaviour (see paper)
- I will not give a full comparison here, just an idea of the difference of how the two implementations work

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.

- 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 ⇒ correct on-chain **contract state** evolution
- Enable **comparison** of implementations if a given spec

- just say what’s on the slide

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!

- just say what’s on the slide again

Structured contracts

Mechanized in Agda

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

tx sends msg1 to account with ID acntID1, causing it to execute the contract in charge of that account, updating its state
this causes account with ID acntID1 to send msg2 to account with ID acntID2
both contracts process messages sent to them, and update their states