Structured Contracts on Cardano

Statefulness in the EUTxO model

Presenter : Polina Vinogradova

Motivation

Evolution of a stateful program (smart contract)

`initState` → `s1` → `s2` → … → `finalState`

Eg. :

- anything implemented with current state machine library
- simulating accounts in EUTxO
- Marlowe?
- Manuel's proposal?

Ledger

```
UTxO set

txin ↦ (myScriptAddr, v, d)

…

…
```

`myScriptAddr = hash (`**`myScript`**`)`

Predicates on ledger and transaction data :

`myScript`` :: Datum -> Redeemer -> Context -> Bool`

Motivation

Evolution of a stateful program (smart contract)

`initState` → `s1` → `s2` → …

→ `finalState`

Eg. :

- anything implemented with current state machine library
- simulating accounts in EUTxO
- Marlowe?
- Manuel's proposal?

How do we relate the two?!

?

Ledger

UTxO set

```
txin ↦ (myScriptAddr, v, d)

  …
```

…

`myScriptAddr = hash (`**`myScript`**`)`

Predicates on ledger and transaction data :

`myScript`` :: Datum -> Redeemer -> Context -> Bool`

Current Situation : Stateful Contracts

- There are **no best practices** for implementing stateful contracts on Cardano

Current Situation : Stateful Contracts

- There are **no best practices** for implementing stateful contracts on Cardano

- There is a **library** for writing instances of **constraint-emitting state machines**
  - NFT-based state tracking mechanism
    - NFT indicates **the unique UTxO whose datum specifies the current state** of the SM
  - Along with updated state, returns **predicates on the updating the state transaction** (eg. validity interval must start after a specific time)
  - Not practical because
    - Constraints library is **not expressive enough**
    - Generated code is **too big** to use on-chain

Current Situation : Stateful Contracts

- There are **no best practices** for implementing stateful contracts on Cardano
- There is a **library** for writing instances of **constraint-emitting state machines**
  - NFT-based state tracking mechanism
    - NFT indicates **the unique UTxO whose datum specifies the current state** of the SM
  - Along with updated state, returns **predicates on the updating the state transaction** (eg. validity interval must start after a specific time)
  - Not practical because
    - Constraints library is **not expressive enough**
    - Generated code is **too big** to use on-chain
- There is a **simplified Agda model** of the ledger and the **SM library**
  - This contains a **proof** that starting from a ledger state without the NFT, every state of the SM that can appear on the ledger is one that can be reached by the SM
  - https://github.com/omelkonian/formal-utxo/blob/master/Bisimulation/Soundness.agda

Current Situation : Ledger

- Ledger is **specified**
  - **semi-formally**
  - using **small-steps semantics**,
  - state transitions are expressed as **functional relations**

Current Situation : Ledger

- Ledger is **specified**
  - **semi-formally**
  - using **small-steps semantics**,
  - state transitions are expressed as **functional relations**

- There is work to build an **Agda formal specification** of the ledger that replaces the semi-formal spec with an automatically generated one
  - https://github.com/input-output-hk/formal-ledger-specifications

Current Situation : Ledger

- Ledger is **specified**
    - **semi-formally**
    - using **small-steps semantics**,
    - state transitions are expressed as **functional relations**

- There is work to build an **Agda formal specification** of the ledger that replaces the semi-formal spec with an automatically generated one
    - https://github.com/input-output-hk/formal-ledger-specifications

- The plan is to have a **CI** that tests the **conformance of the implementation to the formal specification**
    - part of the chain of evidence that connects the ledger design to the implementation

Recall the ledger spec…

This is the **small-step-semantics specification** of how to **apply a transaction** to the ledger state

Ledger transition type :

$$\_ \vdash \_ \xrightarrow[\text{LEDGER}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{LEnv} \times \mathsf{LState} \times \mathsf{Tx} \times \mathsf{LState})$$

`LEDGER` transition (`isValid = True` rule) :

$$\text{ledger-V}\;\frac{\begin{array}{c}\mathsf{isValid}\;tx = \mathsf{True} \\[1em] \begin{array}{c} slot \\ txIx \\ pp \\ tx \\ acnt \end{array} \vdash dpstate \xrightarrow[\text{DELEGS}]{\mathsf{txcerts}\;(\mathsf{txbody}\;tx)} dpstate' \\[1em] (dstate, pstate) := dpstate \\ (\_,\_,\_,\_,\_, genDelegs, \_) := dstate \\ (poolParams, \_, \_) := pstate \\[1em] \begin{array}{c} slot \\ pp \\ poolParams \\ genDelegs \end{array} \vdash utxoSt \xrightarrow[\text{UTXOW}]{tx} utxoSt' \end{array}}{\begin{array}{c} slot \\ txIx \\ pp \\ acnt \end{array} \vdash \begin{pmatrix} utxoSt \\ dpstate \end{pmatrix} \xrightarrow[\text{LEDGER}]{tx} \begin{pmatrix} \boldsymbol{utxoSt'} \\ \boldsymbol{dpstate'} \end{pmatrix}}$$

Small-steps for stateful contracts

We can specify stateful smart contracts using **the same formalism as the ledger spec!**

Small-steps for stateful contracts

We can specify stateful smart contracts using **the same formalism as the ledger spec!**

Recall here that we are working towards implementing stateful contracts via the ledger

Small-steps for stateful contracts

We can specify stateful smart contracts using **the same formalism as the ledger spec!**

Recall here that we are working towards implementing stateful contracts via the ledger

- **Why specify state transitions for contracts at all?**
  - Spec-first approach is evidence-based, faster than testing/patching implementation
  - Can prove desired properties
  - Can go from semi-formal to **mechanized Agda** code later, for higher assurance

Small-steps for stateful contracts

We can specify stateful smart contracts using **the same formalism as the ledger spec!**

Recall here that we are working towards implementing stateful contracts via the ledger

- **Why specify state transitions for contracts at all?**
    - Spec-first approach is evidence-based, faster than testing/patching implementation
    - Can prove desired properties
    - Can go from semi-formal to **mechanized Agda** code later, for higher assurance
- Why use **small steps**?
    - We can relate directly to the ledger formalism without translation
    - Once contract specs are in Agda, can **link** it directly **to the current ledger Agda spec**

What is implementable on the ledger?

We relate the ledger state transition to a contract state transition by saying that what we care about is

What is implementable on the ledger?

We relate the ledger state transition to a contract state transition by saying that what we care about is

"Any state transition system that can be simulated by the ledger"

What is implementable on the ledger?

We relate the ledger state transition to a contract state transition by saying that what we care about is

> **“Any state transition system that can be simulated by the ledger”**

We use the term **structured contract** here to describe such a state transition system

- instead of state machine or stateful contract - because contracts are **not truly stateful** in an EUTxO ledger (they are simulated)

We call the following function which defines the simulation relation the **implementation** of the contract

- `pi : LState -> State`

Instance of a structured contract

An EUTxO structured contract is given by specifying the following data :

(i) Surjective projection $\pi_{\mathsf{State}} \in \mathsf{UTxOState} \to \mathsf{State}$

(ii) Surjective projection $\pi_{\mathsf{Input}} \in \mathsf{TxInfo} \to \mathsf{Input}^{?}$

(iii) Some set of inference rules that specify the transition of type `SMUP` $\_ \vdash \_ \xrightarrow[\text{SMUP}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{TxInfo} \times \mathsf{State} \times \mathsf{Input} \times \mathsf{State})$

(iv) a proof that the **`StatefulStep`** and **`StatefulNoStep`** property is satisfied by the data in (i), (ii), and (iii)

Structured Contracts Simulation Relation

$\texttt{SMUP}$ is simulated by the ledger when $\texttt{\textbf{StatefulNoStep}}$ and $\texttt{\textbf{StatefulNoStep}}$ are satisfied

$$\text{StatefulStep}\ \frac{\begin{array}{c} \mathit{txInfo} := \mathsf{txInfo}\ \mathsf{EI}\ \mathsf{SysSt}\ \mathsf{Lang}\ \mathit{pp}\ (\mathsf{getUTxO}\ \mathit{utxoSt})\ \mathit{tx} \\ \mathsf{isValid}\ \mathit{tx} = \mathsf{True} \qquad\qquad \pi_{\mathit{Input}}\ \mathit{txInfo} \neq \Diamond \\ \\ \begin{array}{c} \mathit{slot} \\ \mathit{txIx} \\ \mathit{pp} \\ \mathit{account} \end{array} \vdash \begin{pmatrix} \mathit{utxoSt} \\ \mathit{dpstate} \end{pmatrix} \xrightarrow[\text{LEDGER}]{\mathit{tx}} \begin{pmatrix} \mathit{utxoSt}' \\ \mathit{dpstate}' \end{pmatrix} \end{array}}{\mathit{txInfo} \vdash (\pi_{\mathit{State}}\ \mathit{utxoSt}) \xrightarrow[\text{SMUP}]{\pi_{\mathit{Input}}\ \mathit{txInfo}} (\pi_{\mathit{State}}\ \mathit{utxoSt}')}$$

Structured Contracts Simulation Relation

SMUP is simulated by the ledger when **StatefulNoStep** and **StatefulNoStep** are satisfied

This is not a rule, it is a **constraint** that may or may not be satisfied in general

- but has to be, by definition, if SMUP is a structured contract

$$\text{StatefulStep}\ \frac{\begin{array}{c} \mathit{txInfo} := \mathsf{txInfo\ El\ SysSt\ Lang}\ \mathit{pp}\ (\mathsf{getUTxO}\ \mathit{utxoSt})\ \mathit{tx} \\ \mathsf{isValid}\ \mathit{tx} = \mathsf{True} \qquad \pi_{Input}\ \mathit{txInfo} \neq \diamond \\ \begin{array}{c}\mathit{slot}\\ \mathit{txIx}\\ \mathit{pp}\\ \mathit{account}\end{array} \vdash \begin{pmatrix}\mathit{utxoSt}\\ \mathit{dpstate}\end{pmatrix} \xrightarrow[\text{LEDGER}]{\mathit{tx}} \begin{pmatrix}\mathit{utxoSt}'\\ \mathit{dpstate}'\end{pmatrix}\end{array}}{\mathit{txInfo} \vdash (\pi_{State}\ \mathit{utxoSt}) \xrightarrow[\text{SMUP}]{\pi_{Input}\ \mathit{txInfo}} (\boldsymbol{\pi_{State}\ \mathit{utxoSt}'})}$$

Current State Machine Library Small-Steps
See : NFTCE Transition

$$\text{MintsNFT}\ \frac{\begin{array}{c}\text{mints thread token NFT (spends utxoNFT)}\\[1em] \text{puts thread token NFT into UTxO with correct validator}\end{array}}{txInfo \vdash (\diamond) \xrightarrow[\text{NFTCE}]{smi} \begin{array}{c}\textbf{fst initState}\\ \textbf{snd initState}\end{array}}$$

$$\text{PropagatesNFT}\ \frac{\begin{array}{c}\forall p \in \mathsf{buildConstraits}\ txInfo,\ p\ sms\ smi\\[1em] \text{propagates thread token into correct output}\\[1em] txInfo \vdash \begin{array}{c}sms\\ val\end{array} \xrightarrow[\text{SMPEC}]{smi} \begin{array}{c}\boldsymbol{sms'}\\ \boldsymbol{val'}\end{array}\end{array}}{txInfo \vdash \begin{array}{c}sms\\ val\end{array} \xrightarrow[\text{NFTCE}]{smi} \begin{array}{c}\boldsymbol{sms'}\\ \boldsymbol{val'}\end{array}}$$

Current State Machine Library Small-Steps
See : `NFTCE` Transition

$\boxed{}$ Validator code shows up here

$$\text{MintsNFT}\;\frac{\boxed{\text{mints thread token NFT (spends utxoNFT)}} \quad \boxed{\text{puts thread token NFT into UTxO with correct validator}}}{txInfo \vdash (\diamond) \xrightarrow[\text{NFTCE}]{smi} \begin{array}{r}\textbf{fst initState}\\ \textbf{snd initState}\end{array}}$$

$$\text{PropagatesNFT}\;\frac{\forall p \in \boxed{\text{buildConstraits } txInfo,\ p\ sms\ smi} \quad \boxed{\text{propagates thread token into correct output}} \quad txInfo \vdash \begin{array}{r} sms\\ val\end{array} \xrightarrow[\text{SMPEC}]{smi} \begin{array}{l} sms'\\ val'\end{array}}{txInfo \vdash \begin{array}{r} sms\\ val\end{array} \xrightarrow[\text{NFTCE}]{smi} \begin{array}{l} sms'\\ val'\end{array}}$$

Current State Machine Library Implementation

- We must also :
  - **specify** the projection functions
  - **prove** that the `NFTCE` relation satisfies **`StatefulStep`** and **`StatefulNoStep`** constraints
    - this proof involves reasoning about the Plutus scripts we construct for `NFTCE` definition

◇ ⟶ `(myScript, myValue + myNFT, initState)` ⟶ … ⟶ …

⟶ `(myScript, myValue + myNFT, myState)`

Example Structured Contract Spec : Account Simulation

$$\text{Open}\ \frac{\begin{array}{c} accIn \in \mathsf{OArgs} \\[1em] \mathsf{pk}\ accIn \in \mathsf{txInfoSignatories}\ txInfo \qquad id := \mathsf{id}\ accIn \qquad id \notin \mathrm{dom}\ accts \\[1em] \mathsf{pk}\ (id,\ newAcct) = \mathsf{pk}\ accIn \qquad\qquad \mathsf{val}\ newAcct = \mathsf{zero} \end{array}}{txInfo \vdash \left(accts\right) \xrightarrow[\text{ACCNT}]{accIn} \left(accts \cup \{id \mapsto newAcct\}\right)}$$

$$\text{Close}\ \frac{\begin{array}{c} accIn \in \mathsf{CArgs} \\[1em] \mathsf{id}\ accIn \mapsto acntToClose \in accts \\ \mathsf{pk}\ accIn \in \mathsf{txInfoSignatories}\ txInfo \qquad \mathsf{val}\ acntToClose = \mathsf{zero} \end{array}}{txInfo \vdash \left(accts\right) \xrightarrow[\text{ACCNT}]{accIn} \left((\mathsf{id}\ accIn) \ntriangleleft accts\right)}$$

Why this approach?

- **Separate** specification from implementation
  - Maintain **provable formal relation** (simulation) between them
  - Hopefully do this in **Agda** so it remains **in sync with current ledger spec**

Why this approach?

- **Separate** specification from implementation
  - Maintain **provable formal relation** (simulation) between them
  - Hopefully do this in **Agda** so it remains **in sync with current ledger spec**

- **Prove** things about contracts at spec level
  - **translate** these properties to **ledger**-implemented contracts
  - allow for **different implementations** (each with its own simulation proof obligation)
  - Examples : **accounts** (naive vs … vs multi-threaded),
    - **messages-passing** (eg. all messages in one UTxO or one message per UTxO)

Why this approach?

- **Separate** specification from implementation
  - Maintain **provable formal relation** (simulation) between them
  - Hopefully do this in **Agda** so it remains **in sync with current ledger spec**

- **Prove** things about contracts at spec level
  - **translate** these properties to **ledger**-implemented contracts
  - allow for **different implementations** (each with its own simulation proof obligation)
  - Examples : **accounts** (naive vs … vs multi-threaded),
    - **messages-passing** (eg. all messages in one UTxO or one message per UTxO)

- **Non-functional relations** allow for **flexible** and **varied implementations**
  - **Implementations** will **force functionality**, since the ledger is functional

Why this approach?

- **Separate** specification from implementation
  - Maintain **provable formal relation** (simulation) between them
  - Hopefully do this in **Agda** so it remains **in sync with current ledger spec**

- **Prove** things about contracts at spec level
  - **translate** these properties to **ledger**-implemented contracts
  - allow for **different implementations** (each with its own simulation proof obligation)
  - Examples : **accounts** (naive vs … vs multi-threaded),
    - **messages-passing** (eg. all messages in one UTxO or one message per UTxO)

- **Non-functional relations** allow for **flexible** and **varied implementations**
  - **Implementations** will **force functionality**, since the ledger is functional

- Delineates a **design space for stateful contracts** implementable on the ledger

Why this approach?

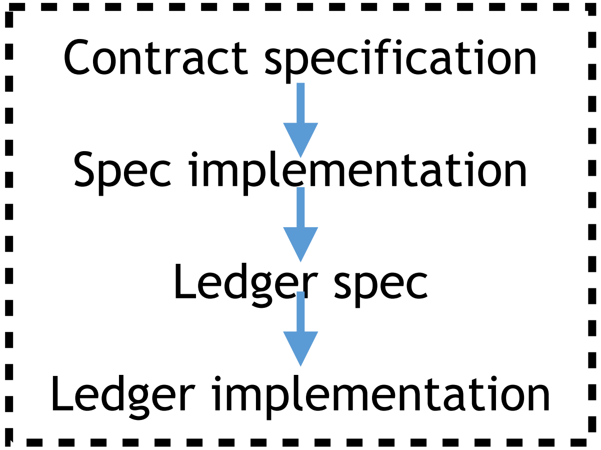
- **Separate** specification from implementation
  - Maintain **provable formal relation** (simulation) between them
  - Hopefully do this in **Agda** so it remains **in sync with current ledger spec**

- **Prove** things about contracts at spec level
  - **translate** these properties to **ledger**-implemented contracts
  - allow for **different implementations** (each with its own simulation proof obligation)
  - Examples : **accounts** (naive vs … vs multi-threaded),
    - **messages-passing** (eg. all messages in one UTxO or one message per UTxO)

- **Non-functional relations** allow for **flexible** and **varied implementations**
  - **Implementations** will **force functionality**, since the ledger is functional

- Delineates a **design space for stateful contracts** implementable on the ledger

- **Chain of evidence** that connects a program and its implementation on the ledger



But…

- This still **doesn’t guarantee** we will have usable/expressible enough state-simulating mechanisms
    - but it helps use look, and specifies what it means to simulate state
- **Devs** would have to learn **Agda** and this **specification** style
    - will have to develop tools to make this easier (like the current CEM library but more general)
- Might have to **translate** Agda validator predicates to Plutus by hand?
    - more tools for this?
- Right now, this says nothing about **liveness**
    - liveness is hard!
- Proofs about the **real ledger** (rather than a simplified Agda model) are hard!

… → ?? → ?? → ?? → QUESTIONS???