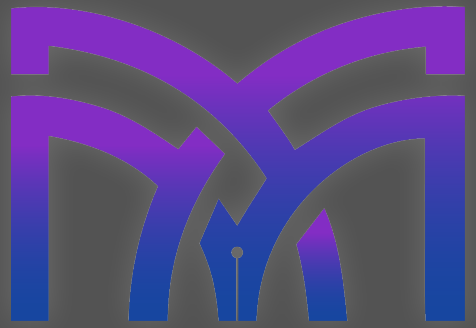
Marlowe

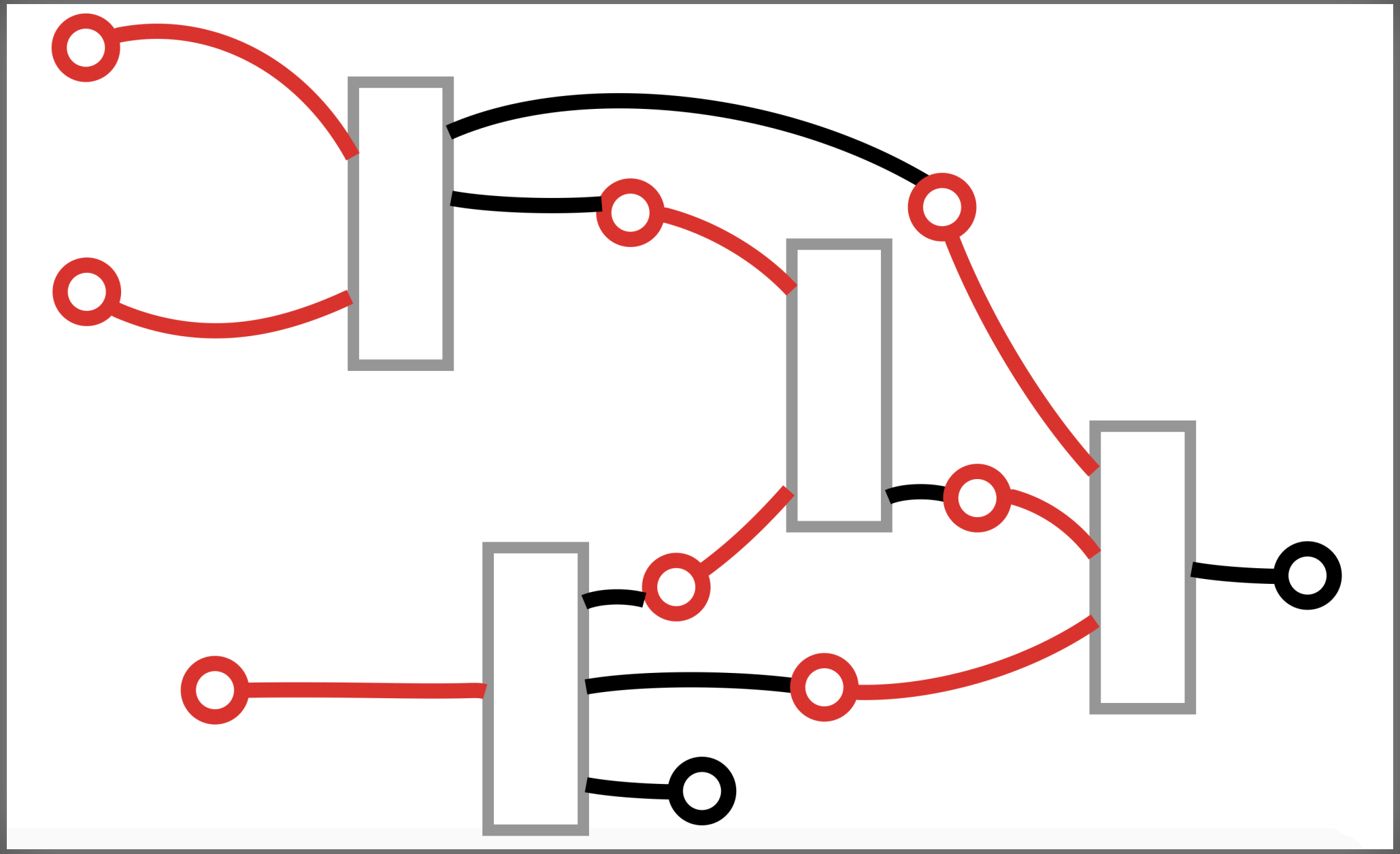
Designing a domain-specific language for financial contracts on blockchain

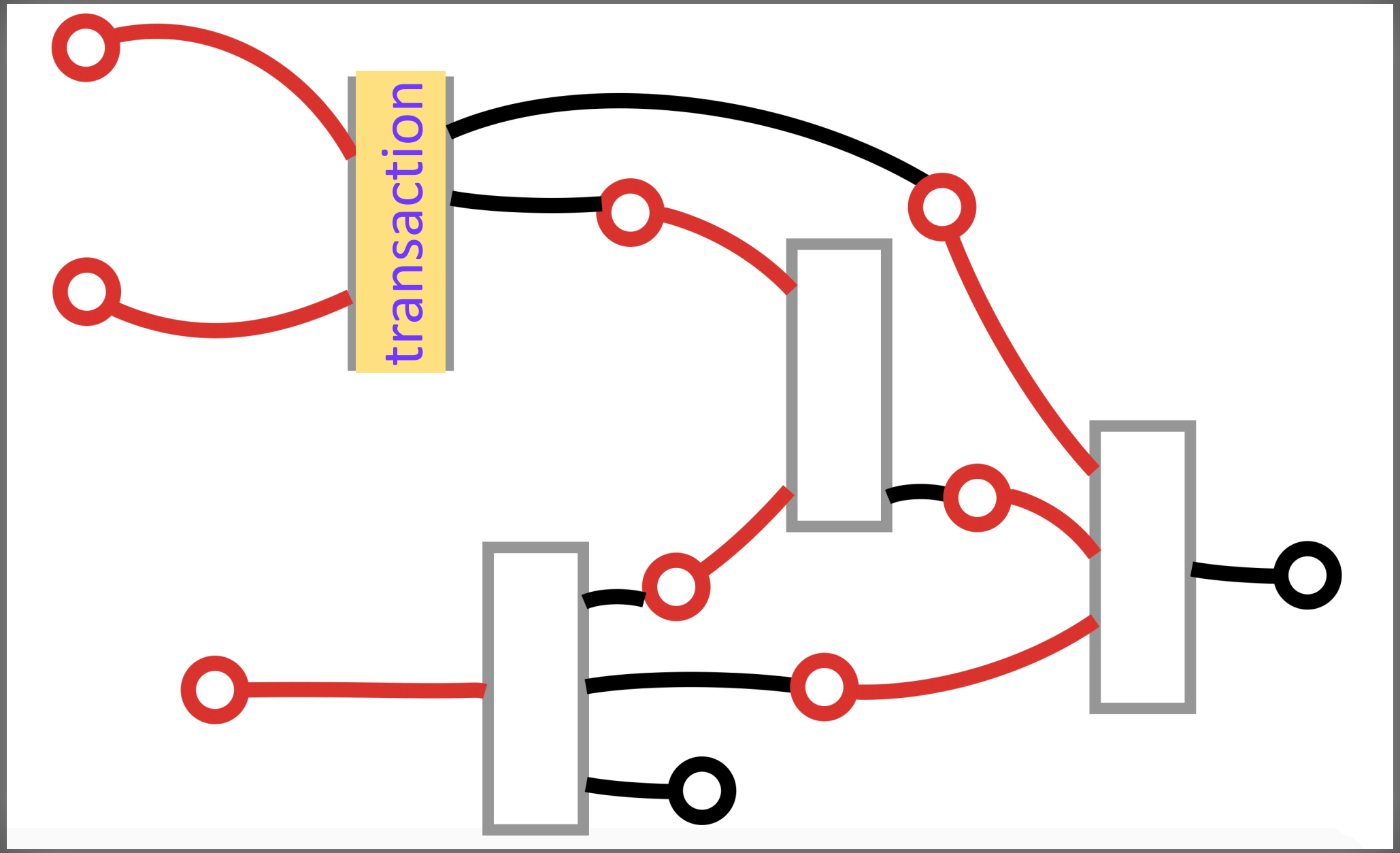


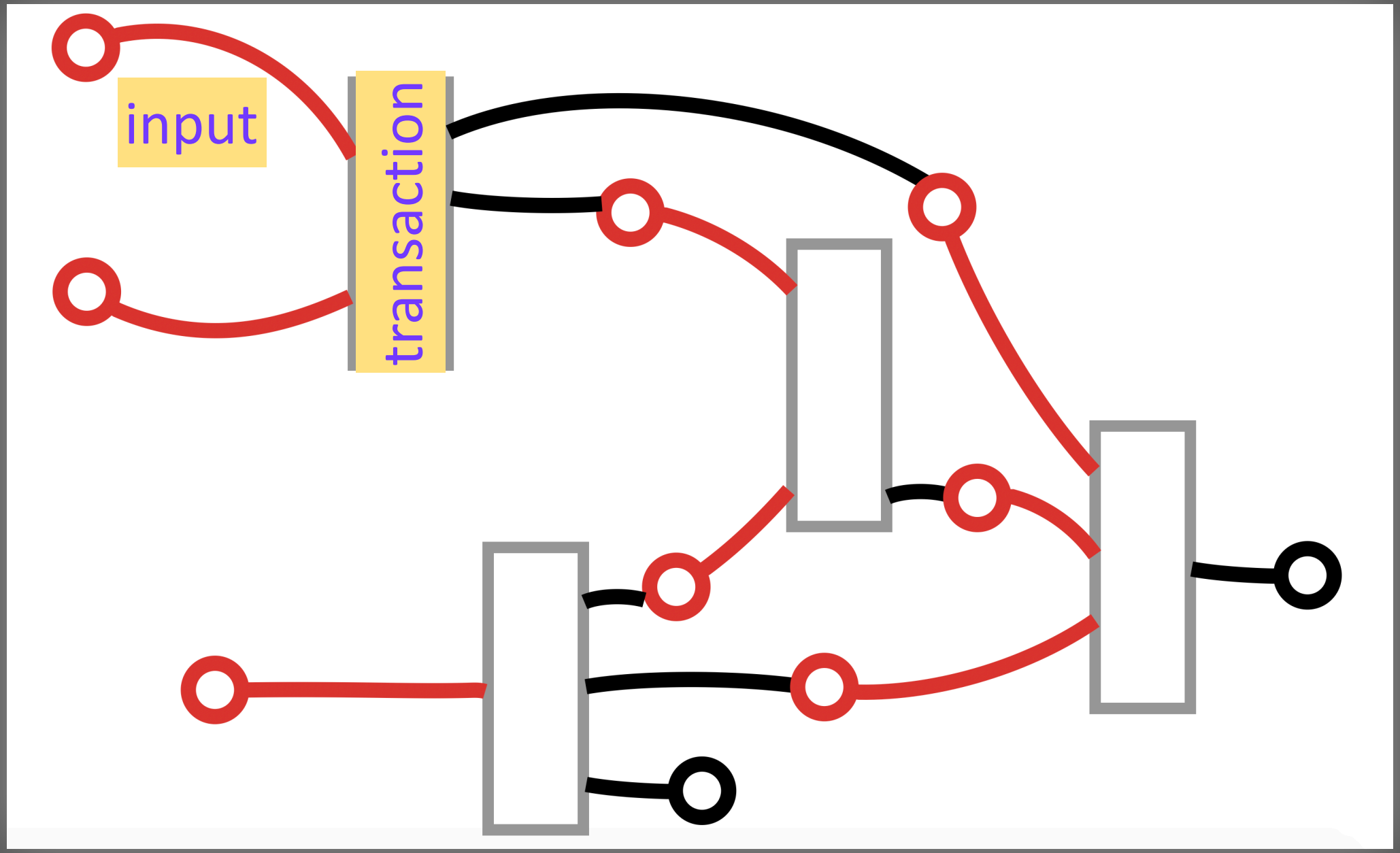
Simon Thompson, IOHK and University of Kent

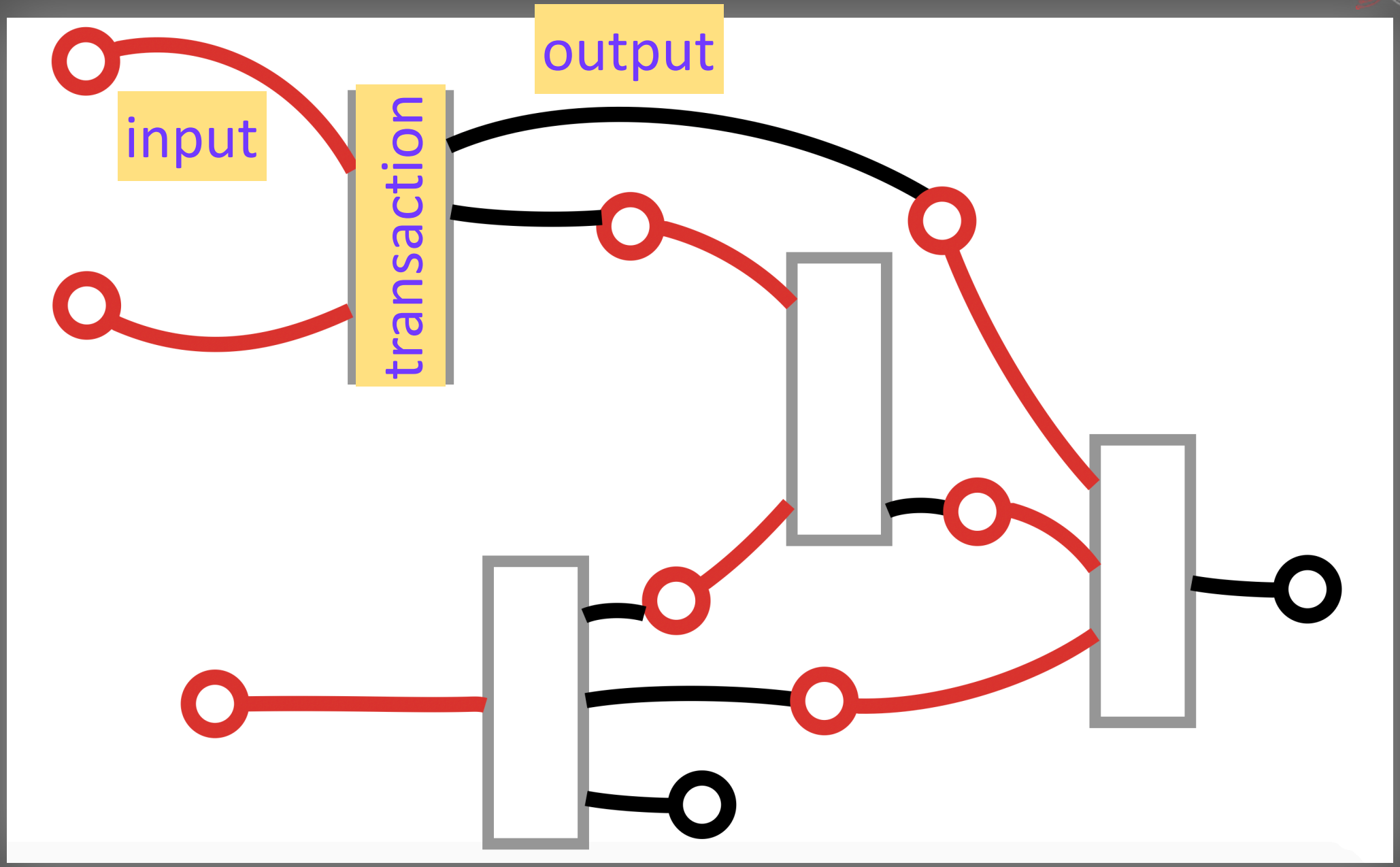
Designing a domain-specific language for financial contracts on blockchain

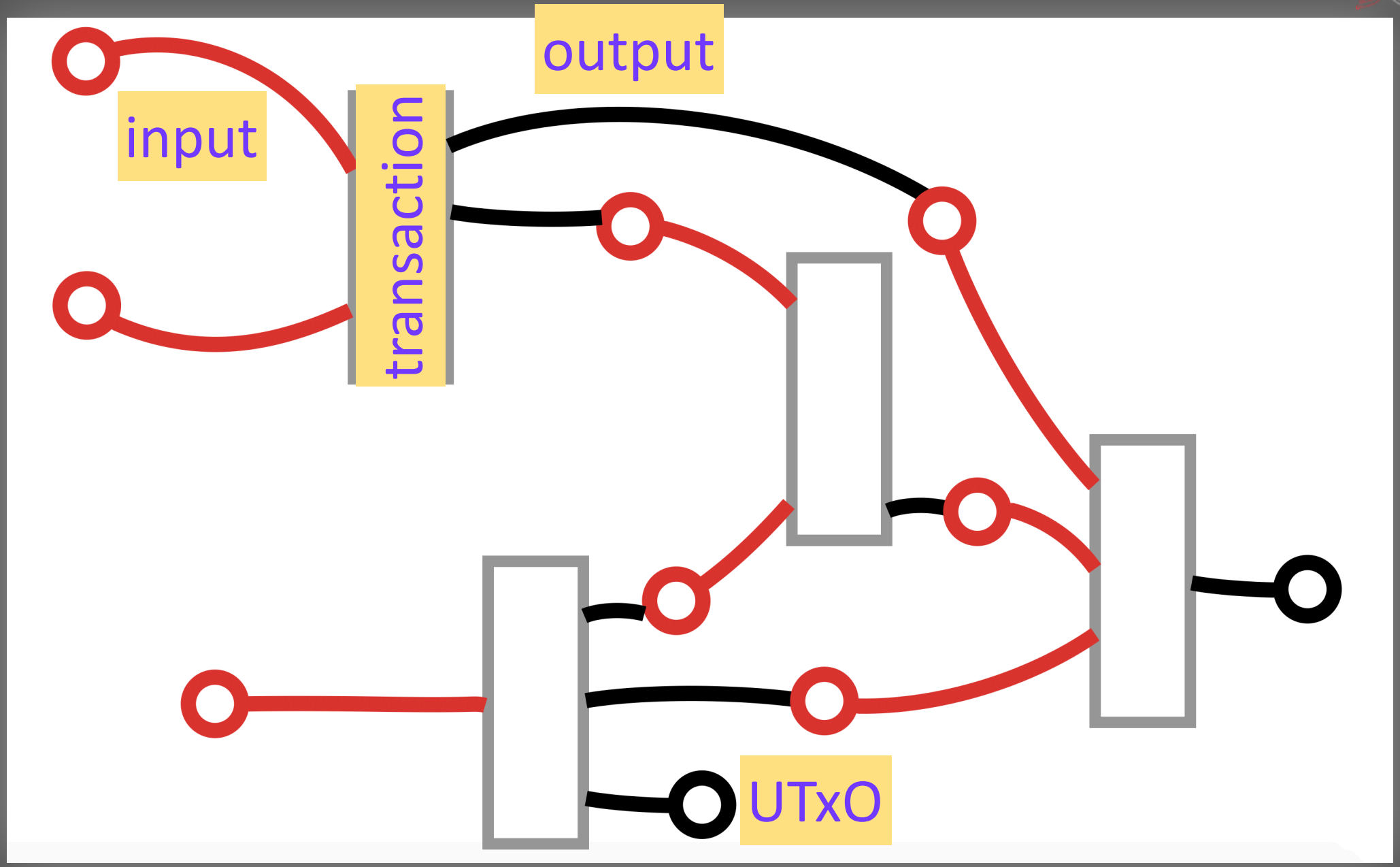
Designing a domain-specific language for financial contracts on blockchain

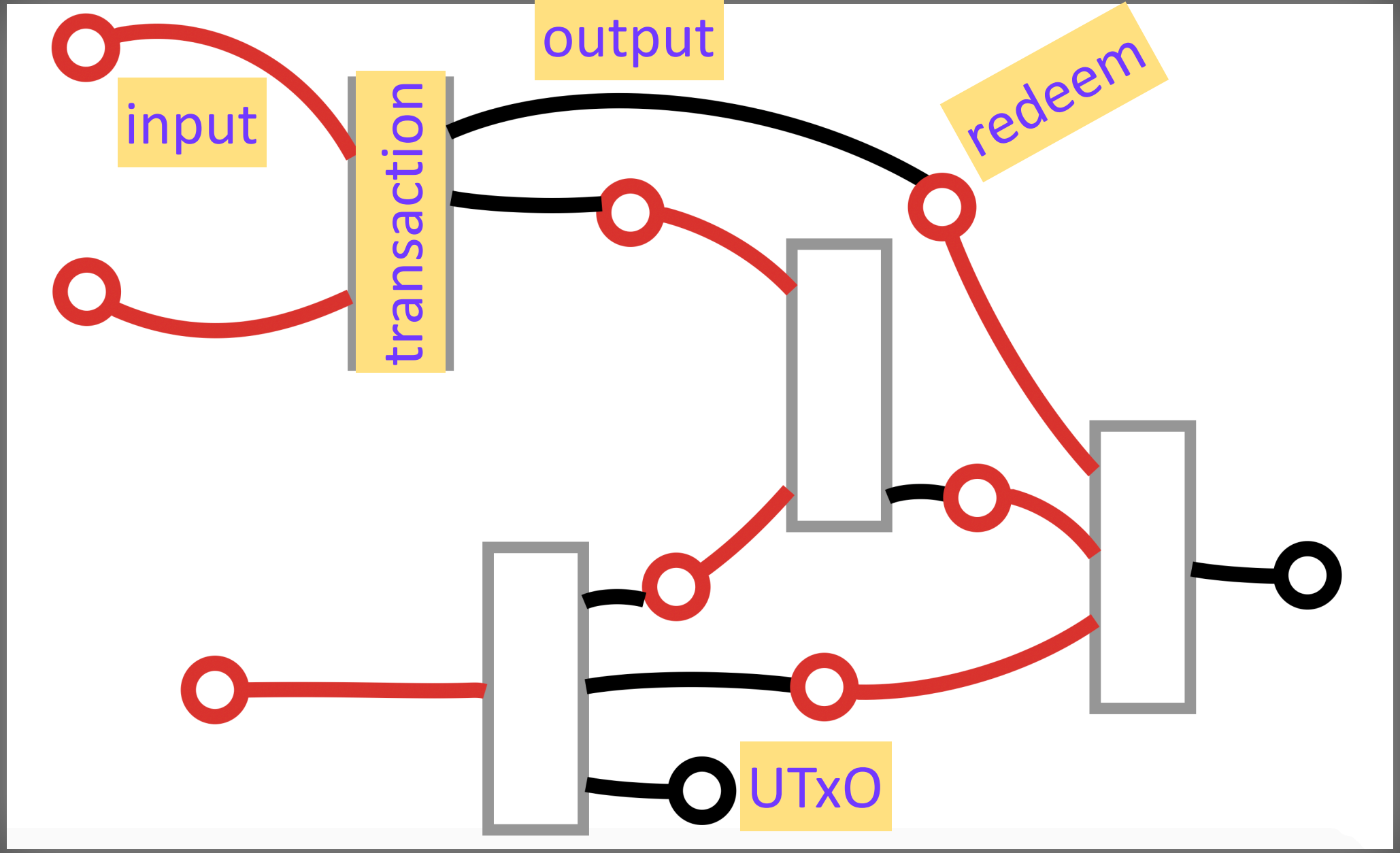


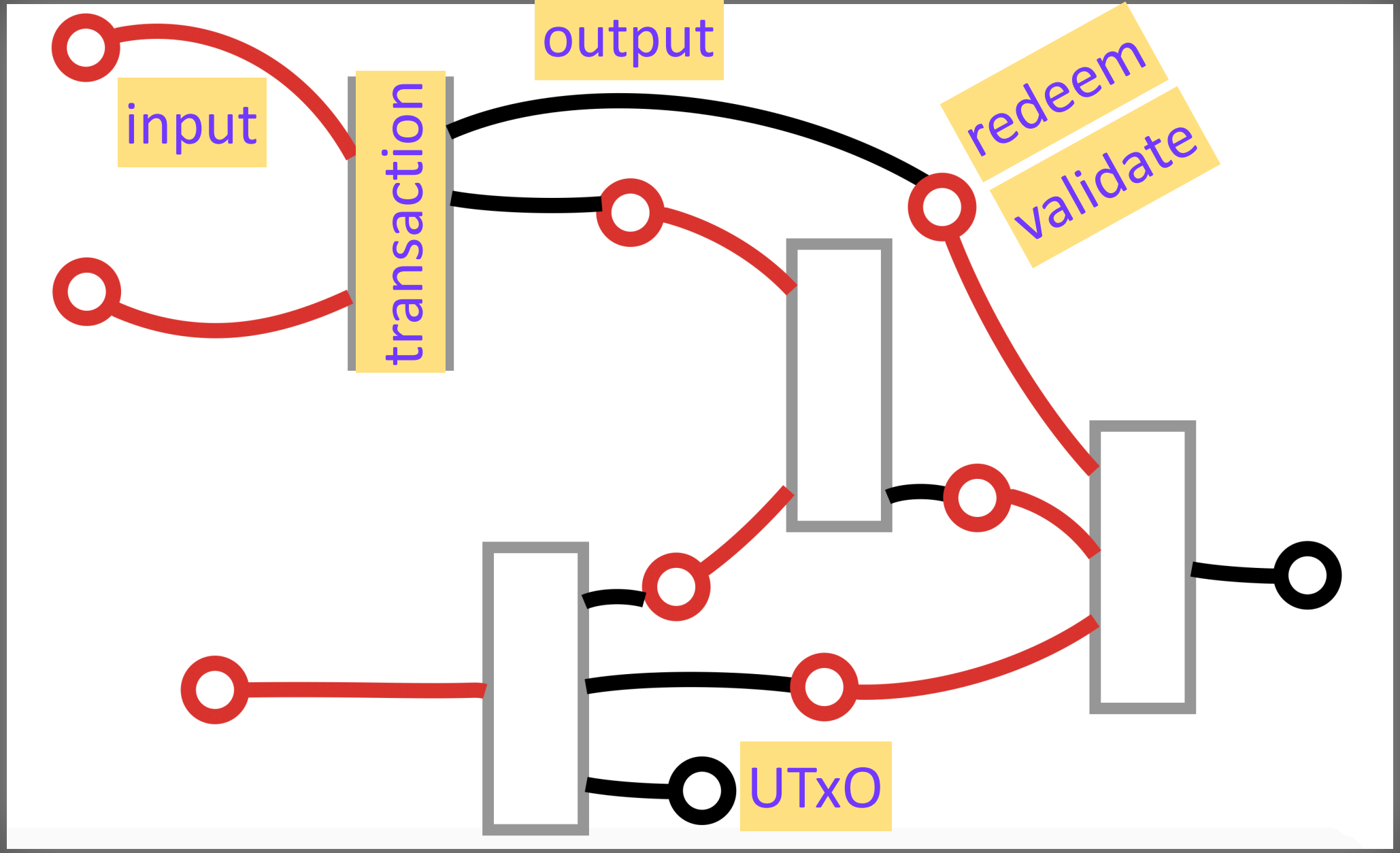

transaction


input
transaction

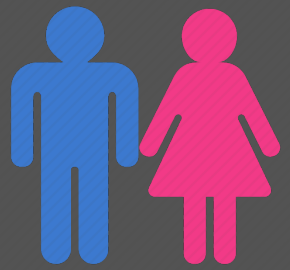

output
input
transaction


output
input
transaction
UTxO


output
input
transaction
redeem
UTxO

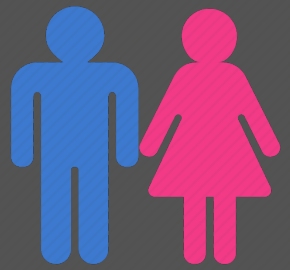

output
input
transaction
redeem
validate
UTxO

Crypto 101



To *send* an output to Bob, Alice encrypts it with Bob's public key: only he can decrypt with his private key.

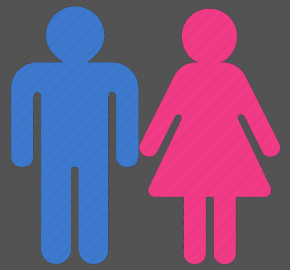
Crypto 101



To *send* an output to Bob, Alice encrypts it with Bob's public key: only he can decrypt with his private key.

To *sign* a message to Bob, Alice encrypts it with her private key: he can decrypt with her public key … only she could have sent it.

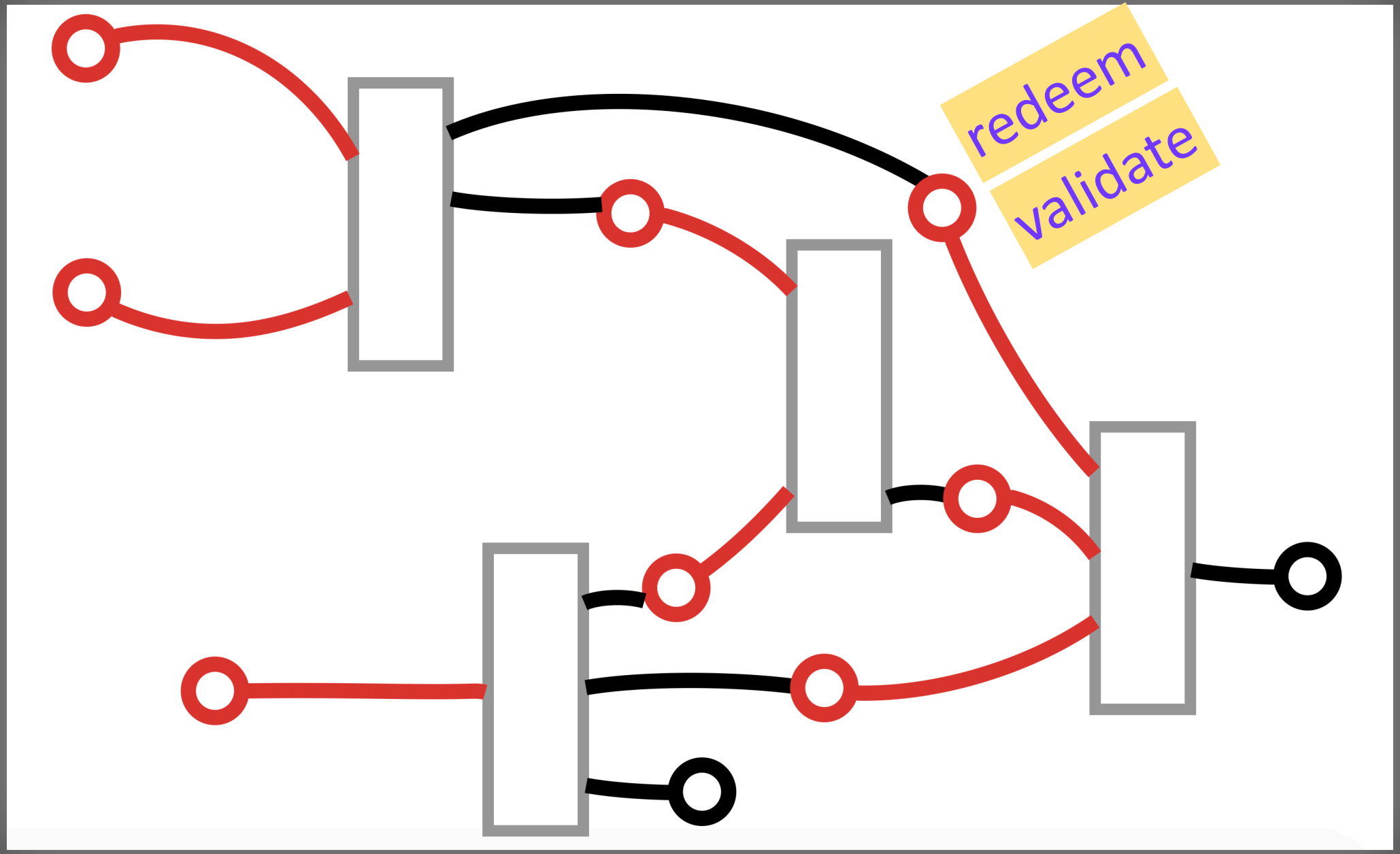
Crypto 101



To *send* an output to Bob, Alice encrypts it with Bob's public key: only he can decrypt with his private key.

To *sign* a message to Bob, Alice encrypts it with her private key: he can decrypt with her public key … only she could have sent it.

To *send* a *signed* message, Alice encrypts it with her private key, and then with Bob's public key. Only he can read … only she could have sent.


redeem
validate

Crypto-economics

Use cryptography to secure the past …
… and financial incentives to shape the future.

Vitalik Buterin / Vlad Zamfir

Ledger

An irrevocable
record of transactions:
ownership, payments, …

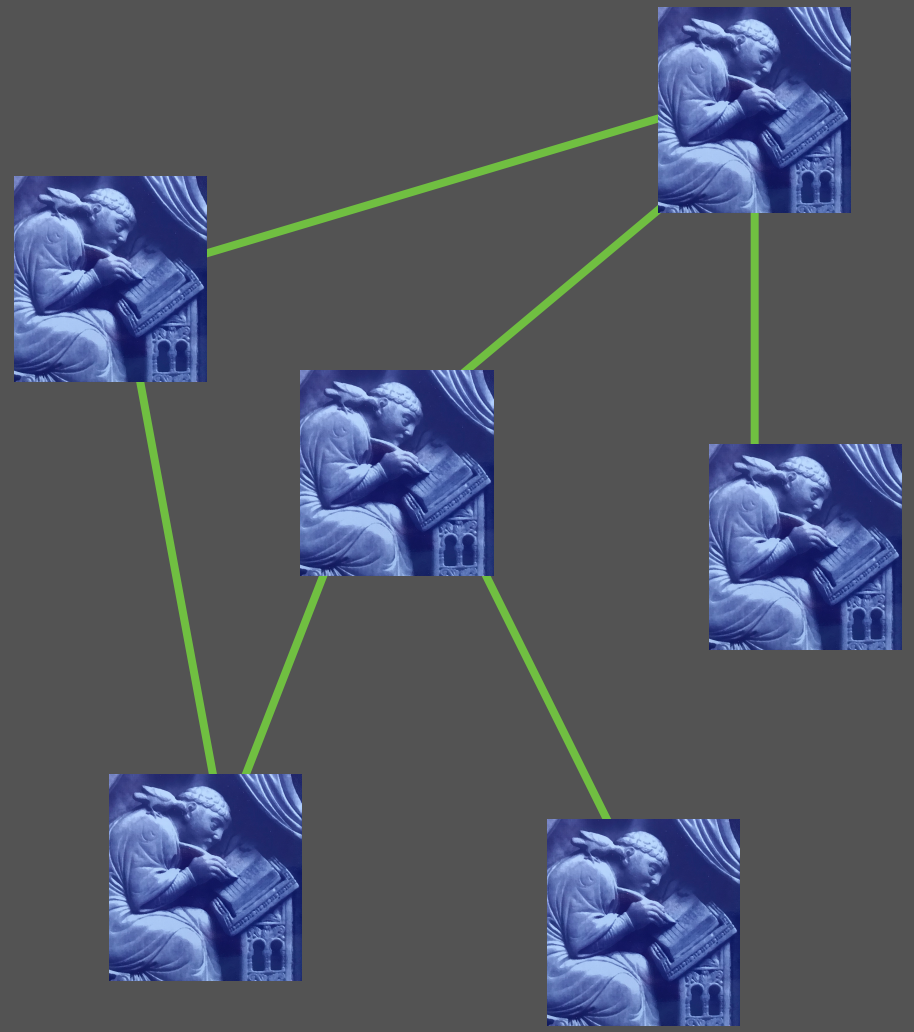
Trust the monk

Distributed ledger

Share information and update accordingly

CAP Tolerance

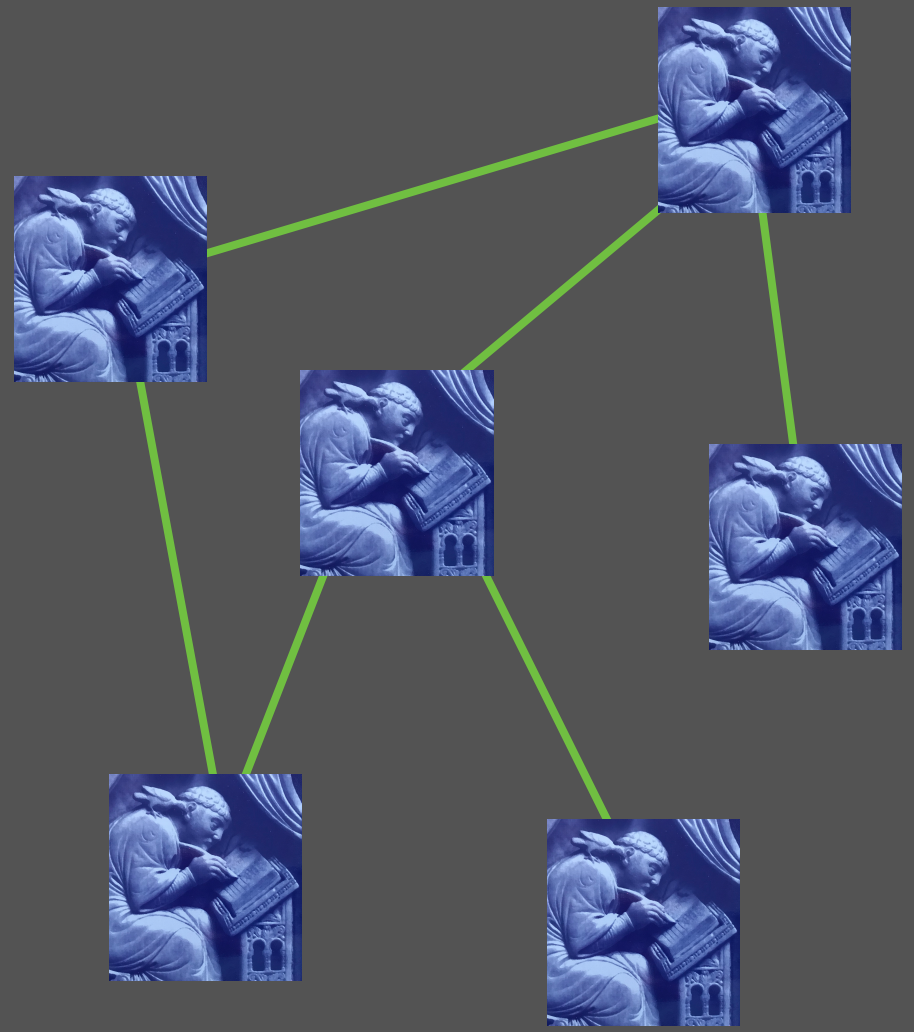
Trust the monks



But what if we can’t trust the monks?

What if some participants are actively hostile?

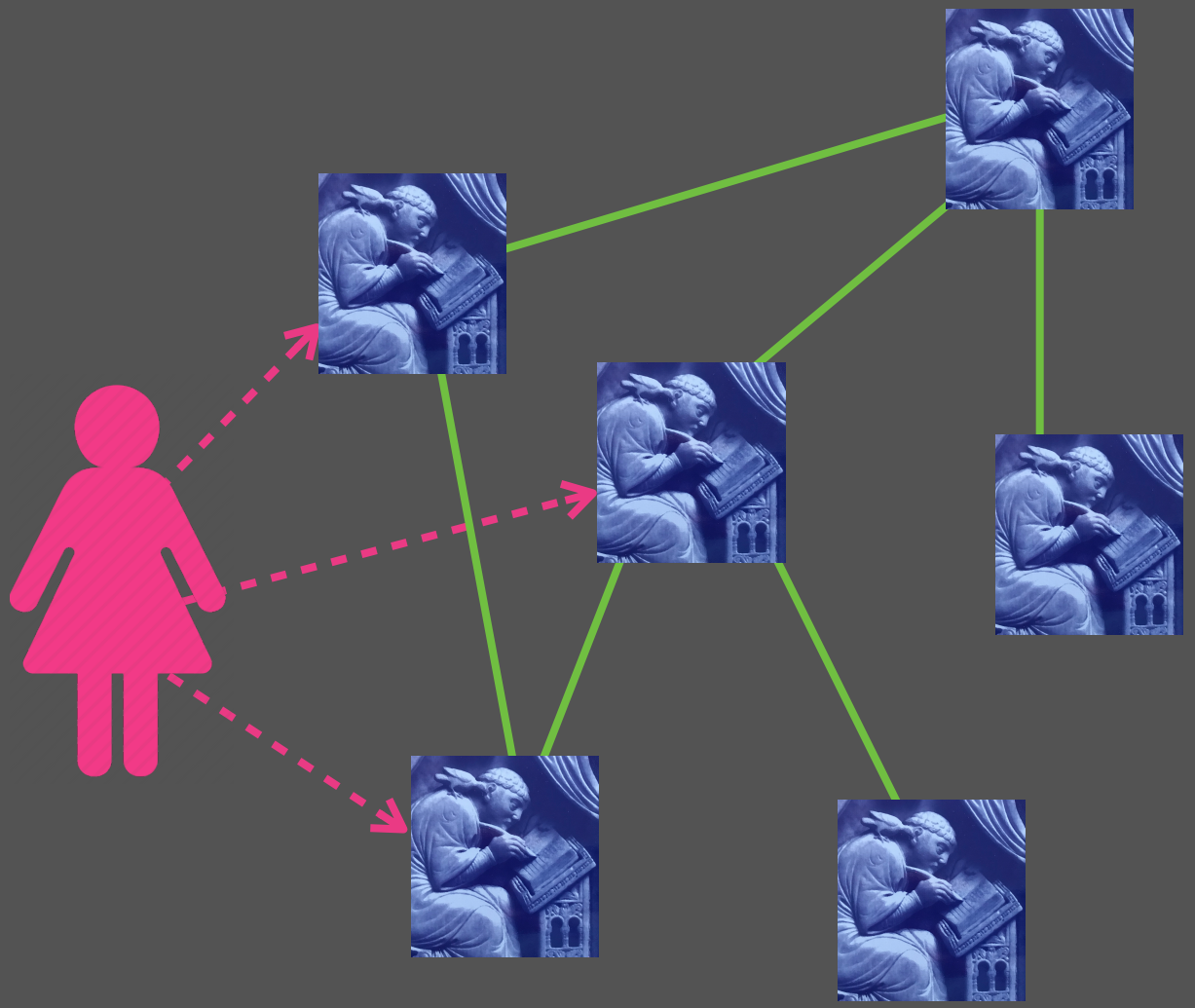
What if some just walk away?



A ledger of transactions

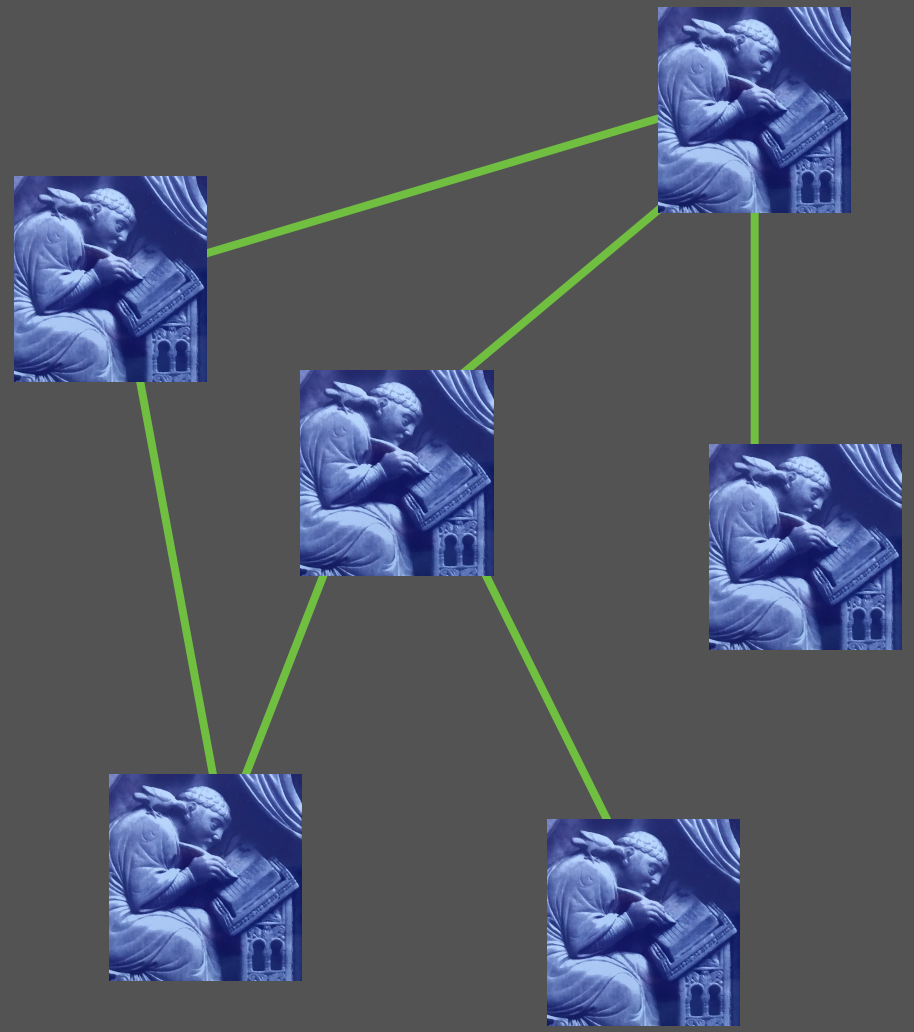
A ledger of transactions: amount X paid to Y by Z, signed by Z and based on this earlier transaction.

Transactions are broadcast to the entire network.



Bitcoin basics - block creation

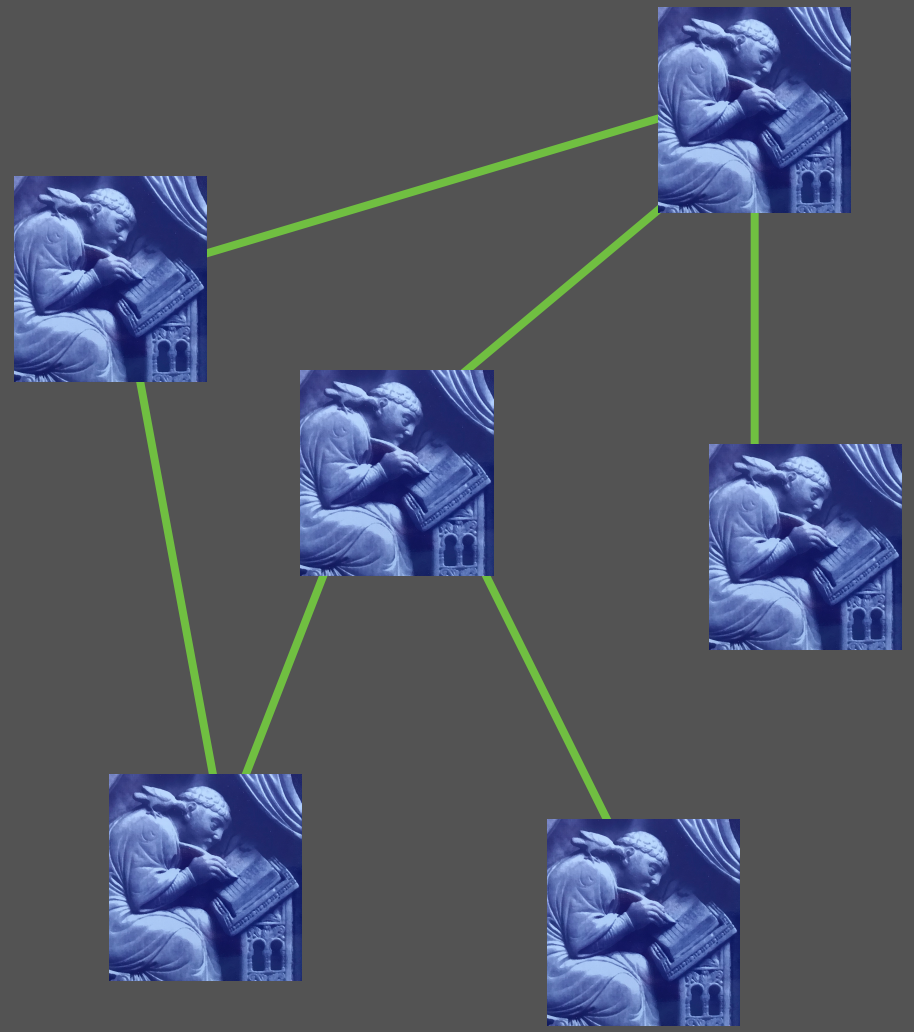
Each node collects new transactions into a block.



Bitcoin basics - block creation

Each node collects new transactions into a block.

Each round, an identified node broadcasts its block.

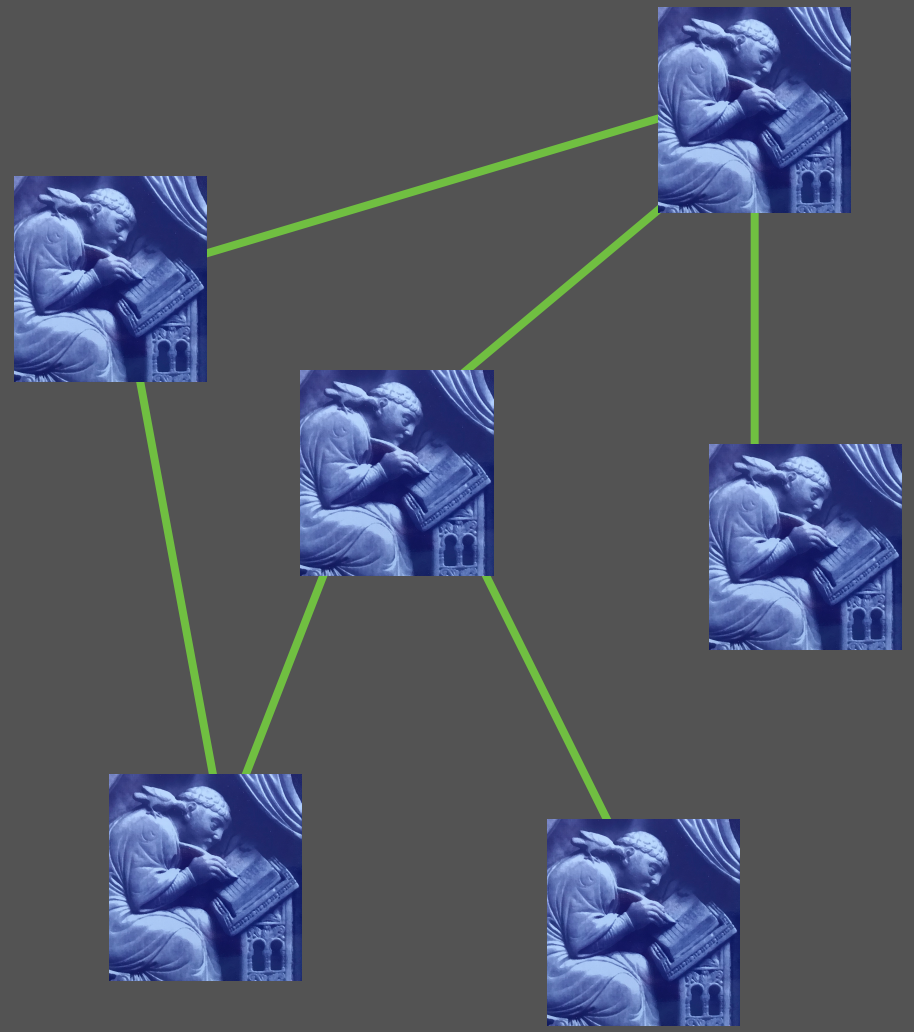


Bitcoin basics - block creation

Each node collects new transactions into a block.

Each round, an identified node broadcasts its block.

All nodes check the validity of the proposed new block: are all transactions valid?



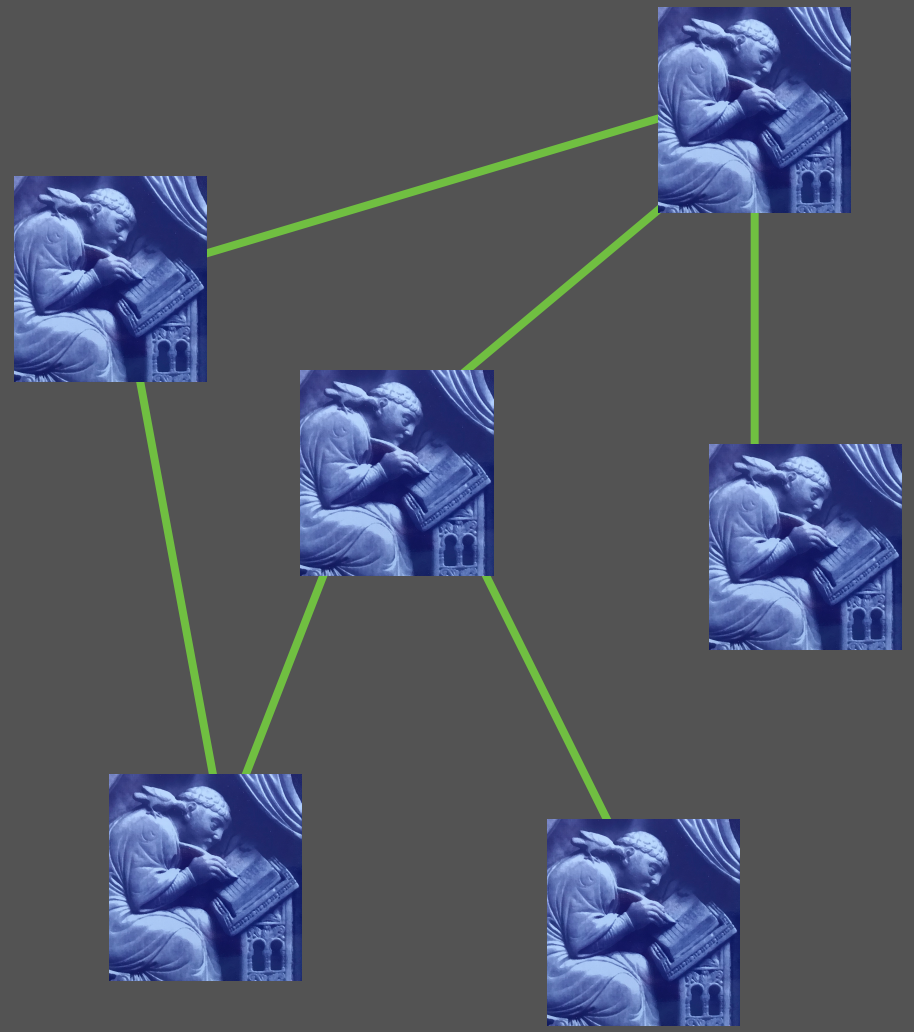
Bitcoin basics - block creation

Each node collects new transactions into a block.

Each round, an identified node broadcasts its block.

All nodes check the validity of the proposed new block: are all transactions valid?

A node accepts a block by including its hash in the next created block.



The essence of blockchain

Irrefutable record of linked *transactions*.

On-chain checks of redemption and validation.

The chain is *passive*: transactions created off-chain.

Off-chain wallets interact with chain.

Distributed system with replicated computation.

Incentives required to keep the chain alive; crypto secures the past.

Designing a domain-specific language for financial contracts on blockchain

Domain-specific language

The language of the user …

… not the machine.

Domain-specific language

Some mistakes made impossible.

Analysis more accurate and easier.

Tools designed to fit the DSL.

Domain-specific language

Embed in full programming language.

Run programs

Simplify

Make a valuation

Designing a domain-specific language for financial contracts on blockchain

We have a model …

Composing contracts: an adventure in financial engineering

Functional pearl

Simon Peyton Jones
Microsoft Research, Cambridge
simonpj@microsoft.com

Jean-Marc Eber
LexiFi Technologies, Paris
jeanmarc.eber@lexifi.com

Julian Seward
University of Glasgow
v-sewardj@microsoft.com

23rd August 2000

Abstract

Financial and insurance contracts do not sound like promising territory for functional programming and formal semantics, but in fact we have discovered that insights from programming languages bear directly on the complex subject of describing and valuing a large class of contracts.

We introduce a combinator library that allows us to describe such contracts precisely, and a compositional denotational semantics that says what such contracts are worth.

At this point, any red-blooded functional programmer should start to foam at the mouth, yelling "build a combinator library". And indeed, that turns out to be not only possible, but tremendously beneficial.

The finance industry has an enormous vocabulary of jargon for typical combinations of financial contracts (swaps, futures, caps, floors, swaptions, spreads, straddles, captions, European options, American options, ...the list goes on). Treating each of these individually is like having a large catalogue of prefabricated components. The trouble is that

Example: escrow contract

Alice wants to buy a cat from Bob,
but neither of them trusts the other.

They both trust Carol, though.

Example: escrow contract

Alice wants to buy a cat from Bob,

but neither of them trusts the other.

They both trust Carol, though.

Example: escrow contract

Alice wants to buy a cat from Bob,

but neither of them trusts the other.

They both trust Carol, though.

Example: escrow contract

Alice wants to buy a cat from Bob,

but neither of them trusts the other.

They both trust Carol, though.

Example: escrow contract

Alice wants to buy a cat from Bob,

but neither of them trusts the other.

They both trust Carol, though.

Example: escrow contract

Alice wants to buy a cat from Bob,
but neither of them trusts the other.

They both trust Carol, though.

Example: escrow contract

Alice wants to buy a cat from Bob,

but neither of them trusts the other.

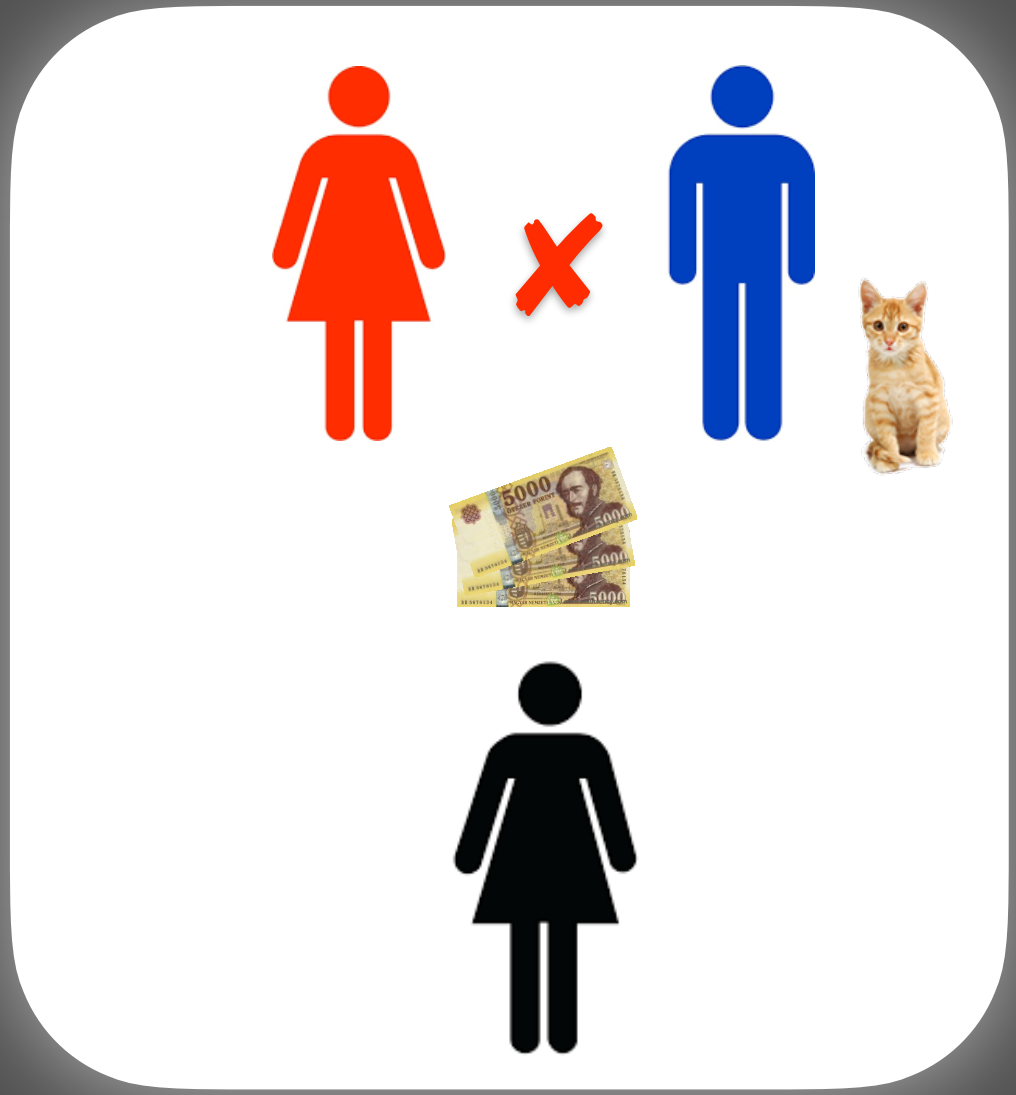
They both trust Carol, though.

Example: escrow contract

Alice wants to buy a cat from Bob,

but neither of them trusts the other.

They both trust Carol, though.

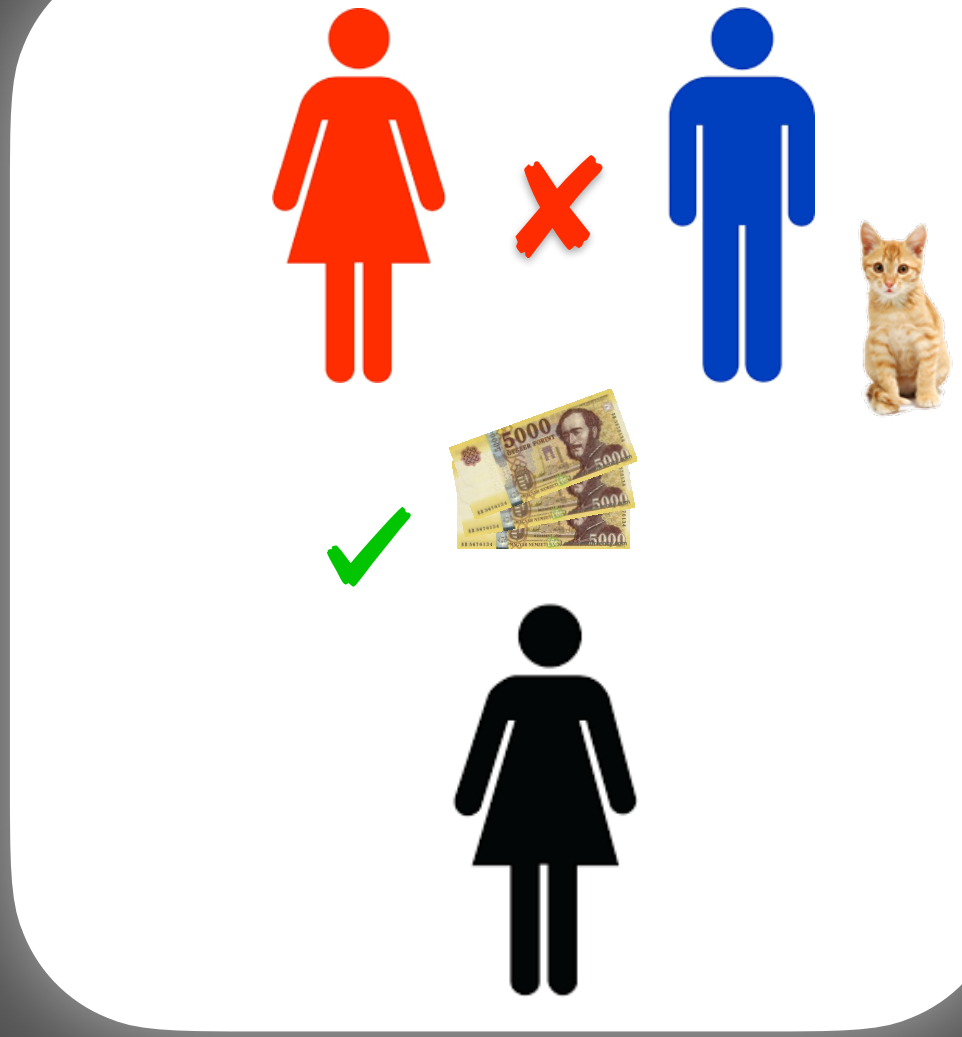


Example: escrow contract

Alice wants to buy a cat from Bob,

but neither of them trusts the other.

They both trust Carol, though.



Example: escrow contract

Alice wants to buy a cat from Bob,
but neither of them trusts the other.

They both trust Carol, though.

Escrow contract example

```
When aliceChoice
    (When bobChoice
        (If (aliceChosen `ValueEQ` bobChosen)
            agreement
            arbitrate)
```

Escrow contract example

```
When aliceChoice
    (When bobChoice
        (If (aliceChosen `ValueEQ` bobChosen)
            If (isPay aliceChosen)
               (Pay "alice" (Party "bob") price …)
               Close
            arbitrate)
```

Escrow contract example

```
When aliceChoice
    (When bobChoice
        (If (aliceChosen `ValueEQ` bobChosen)
            If (isPay aliceChosen)
                (Pay "alice" (Party "bob") price …)
                Close
            arbitrate)
```

Escrow contract example

```
When aliceChoice
    (When bobChoice
        (If (aliceChosen `ValueEQ` bobChosen)
            If (isPay aliceChosen)
                (Pay "alice" (Party "bob") price …)
                Close
            arbitrate)
```

Escrow contract example

```
When aliceChoice
    (When bobChoice
         (If (aliceChosen `ValueEQ` bobChosen)
             If (isPay aliceChosen)
                (Pay "alice" (Party "bob") price …)
                Close
             arbitrate)
```

Escrow contract example

```
If (isPay aliceChosen)
   (Pay "alice" (Party "bob") price …)
    Close
```

Escrow contract example

```
If (isPay aliceChosen)
    (Pay "alice" (Party "bob") price …)
    Close
```

Escrow contract example

```
(Pay "alice" (Party "bob") price …)
```

Designing a domain-specific language for financial contracts on blockchain

The essence of blockchain

Irrefutable record of linked *transactions*.

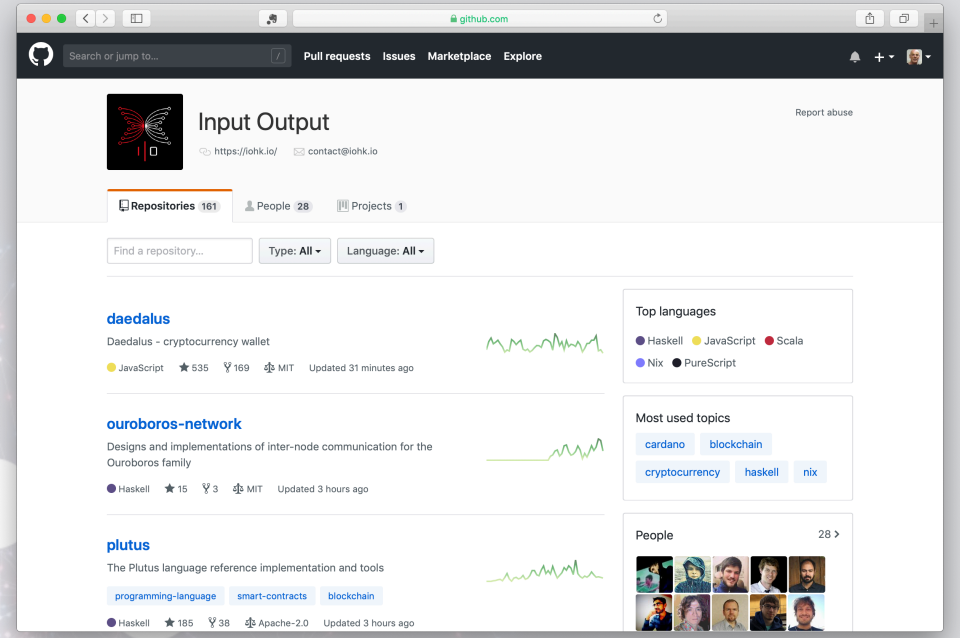
On-chain checks of redemption and validation.

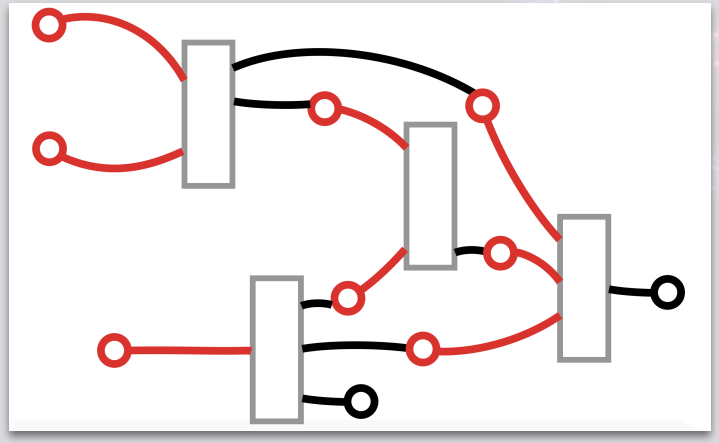
The chain is *passive*: transactions created off-chain.

Off-chain wallets interact with chain.

Distributed system with replicated computation.

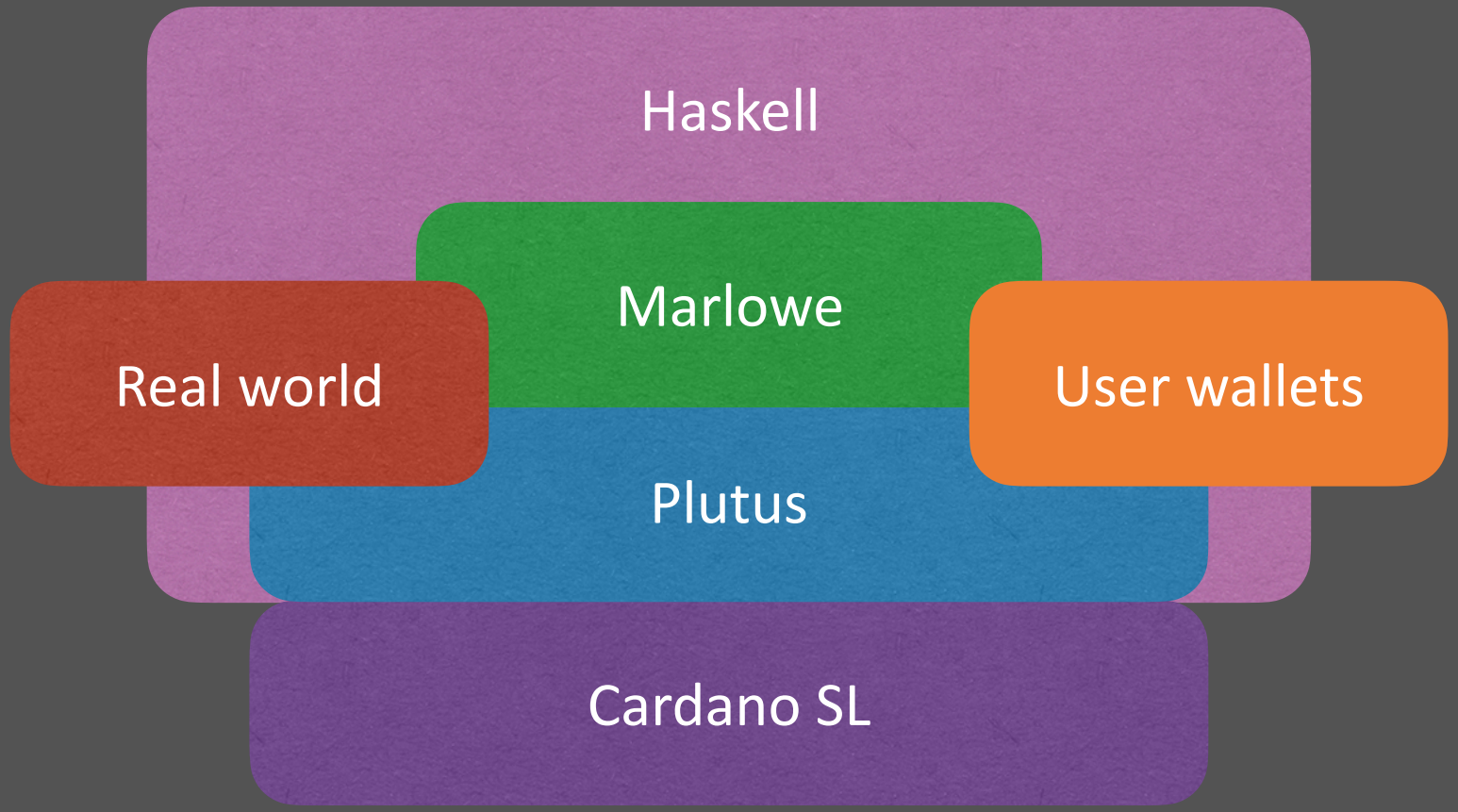
Incentives required to keep the chain alive; crypto secures the past.





```
contribute :: Campaign -> Value -> MockWallet ()
contribute campaign value = do
  when (value <= 0) $
    throwOtherError "Must contribute a positive value"

  ownPK <- ownPubKey
  tx    <- payToScript
             (Ledger.scriptAddress (contributionScript
       Ledger.fromCompiledCode $$(PlutusTx.compile
         [|| (\Campaign{..} action contrib tx ->
           let
             PendingTx ps outs _ _ (Height h) _ _ = tx
             isValid = case action of
                         Refund ->  h >
```

Cardano



Onto blockchain

Enforcement

The legal system ensures financial contracts …

… but should a contract on blockchain enforce itself?

Onto blockchain

Enforcement

The legal system ensures financial contracts …

… but should a contract on blockchain enforce itself?

Double spend

Blockchain designed to prevent spending the same money twice …

… but that's precisely how credit works.

Operation

Modality: “pull” and “push”

The contract can make some things happen, e.g. *payment* …

… while others actuated by participants, e.g. *deposit*.

Operation

Modality: “pull” and “push”

The contract can make some things happen, e.g. *payment* …

… while others actuated by participants, e.g. *deposit*.

Correctness

Not just a matter of *avoiding bad behaviour* …

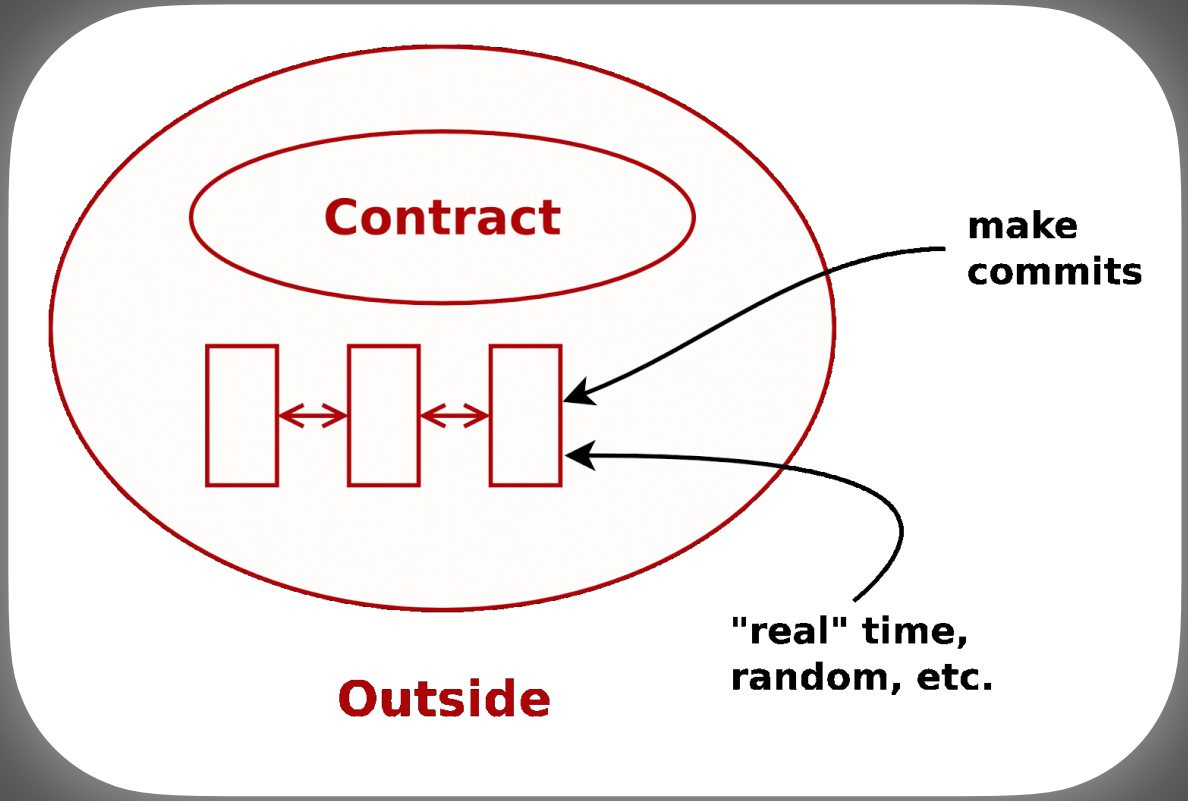
… but also *make good things happen*.

Interactions with the outside world

“The spot price of oil in Oslo at 12:00, 31-12-2019”.

Random values.

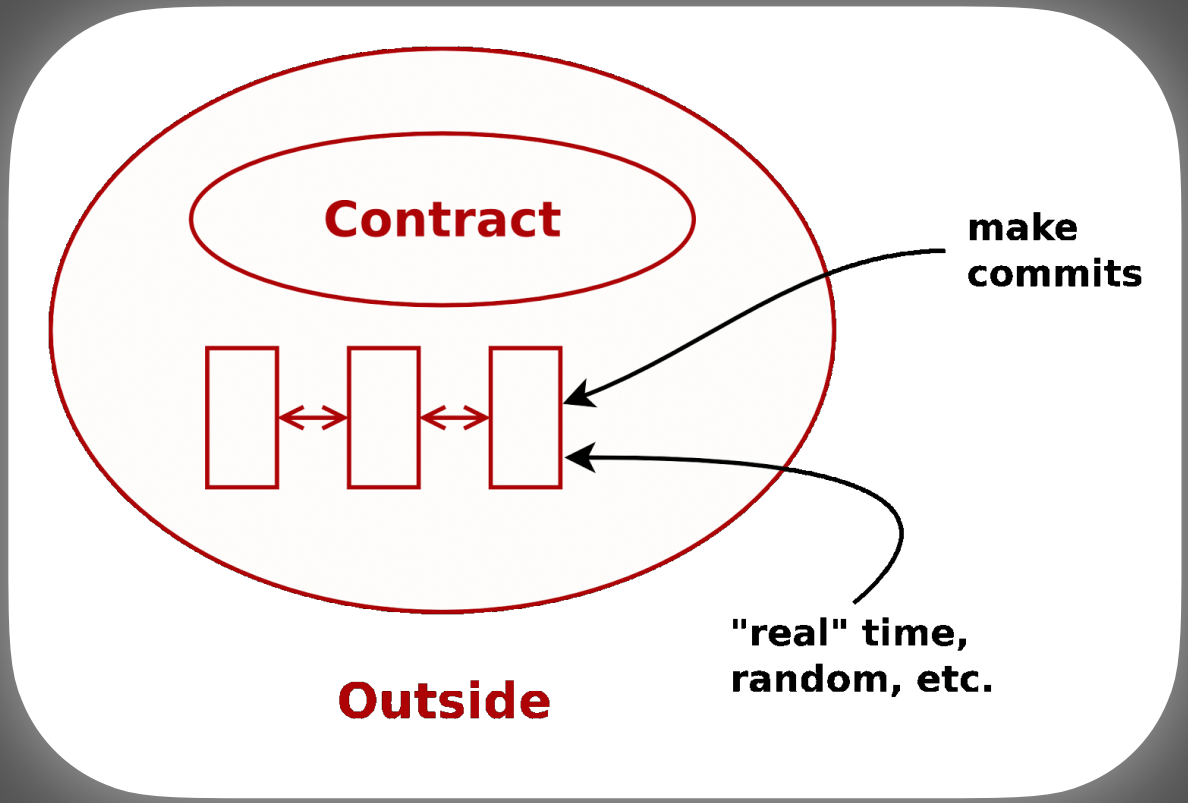
Treat as a *participant* performing an action.



Commitments

Commit money for a finite time, and refund after that.

Need to avoid “walk away” …



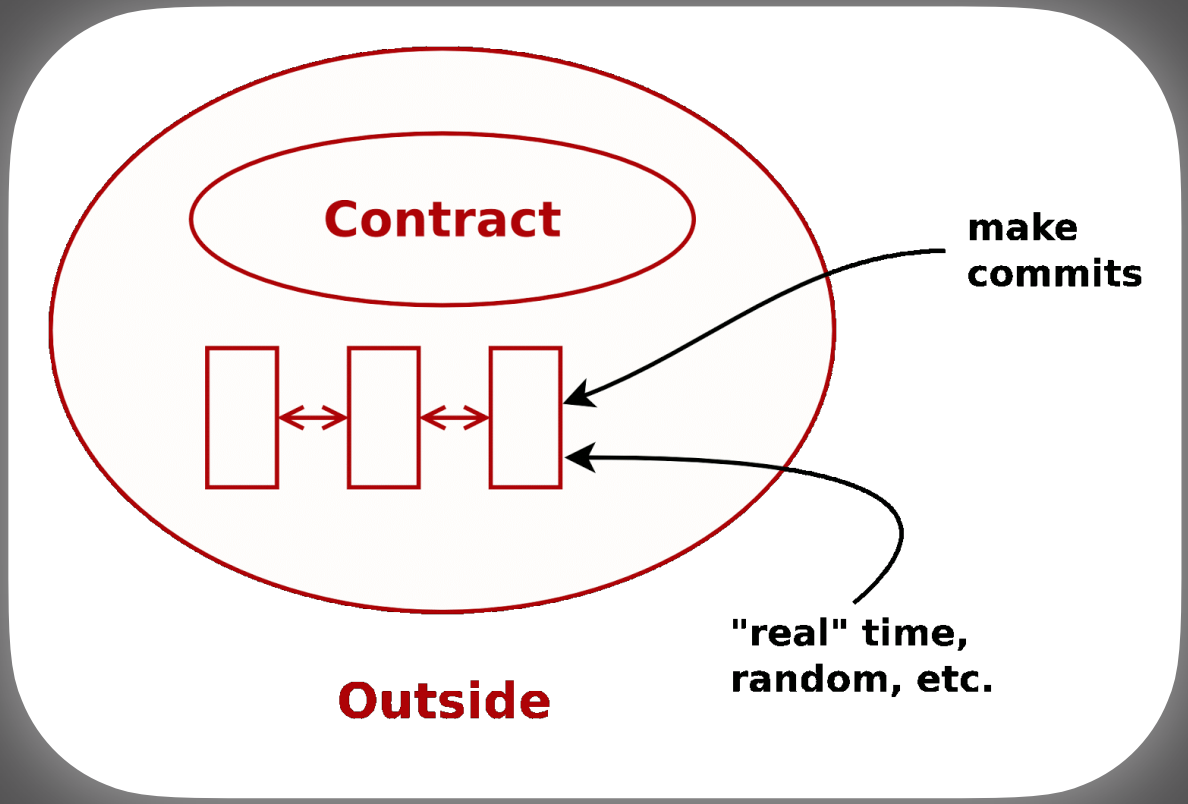
Commitments and timeouts

Commit money for a finite time, and refund after that.

Need to avoid “walk away” …

We can’t *require* a deposit: can only *ask for* one …

… and only wait a fixed time for the deposit to happen.



Escrow contract: timeouts

```
When [ Case aliceChoice
         (When … ),
       Case bobChoice
         (When … )
     ]
```

Escrow contract: timeouts

```
When [ Case aliceChoice
            (When … ),
       Case bobChoice
            (When … )
     ]
```

We wait for whichever `Case` happens: here Alice or Bob can choose first.

Escrow contract: timeouts

```
When [ Case aliceChoice
            (When … ),
       Case bobChoice
            (When … )
     ]
     60
     Close
```

We wait for whichever `Case` happens: here Alice or Bob can choose first.

But only **wait until** slot 60: then `Close` the contract and refund any money left in the contract

Escrow contract: deposits and accounts

```
(When [ Case aliceChoice
              (When … ),
          Case bobChoice
              (When … )
        ]
        60
        Close)
```

Escrow contract: deposits and accounts

```
When [Case (Deposit "alice" "alice" price)
      (When [ Case aliceChoice
                   (When … ),
                Case bobChoice
                   (When … )
            ]
            60
            Close)
```

Wait for Alice to deposit `price` in the account belonging to her: `"alice"`.

Escrow contract: deposits and accounts

```
When [Case (Deposit "alice" "alice" price)
      (When [ Case aliceChoice
                   (When … ),
                Case bobChoice
                   (When … )
            ]
            60
            Close)
      40
      Close
```

Wait for Alice to deposit `price` in the account belonging to her: `"alice"`.

But only **wait until** slot 40: then `Close` the contract, refunding all the money.

The `Contract` data type

```
data Contract = Close
              | Pay AccountId Payee Value Contract
              | If Observation Contract Contract
              | When [Case] Timeout Contract
              | Let ValueId Value Contract


data Case = Case Action Contract
```

The `Contract` data type

```
data Contract = Close
              | Pay AccountId Payee Value Contract
              | If Observation Contract Contract
              | When [Case] Timeout Contract
              | Let ValueId Value Contract

data Case = Case Action Contract
```

Marlowe in a nutshell

An embedded DSL = a Haskell `data` type.

Executable small-step semantics.

Runs on extended UTxO blockchain.

Interpreter = Plutus program.

Analysis with Z3; proof in Isabelle.

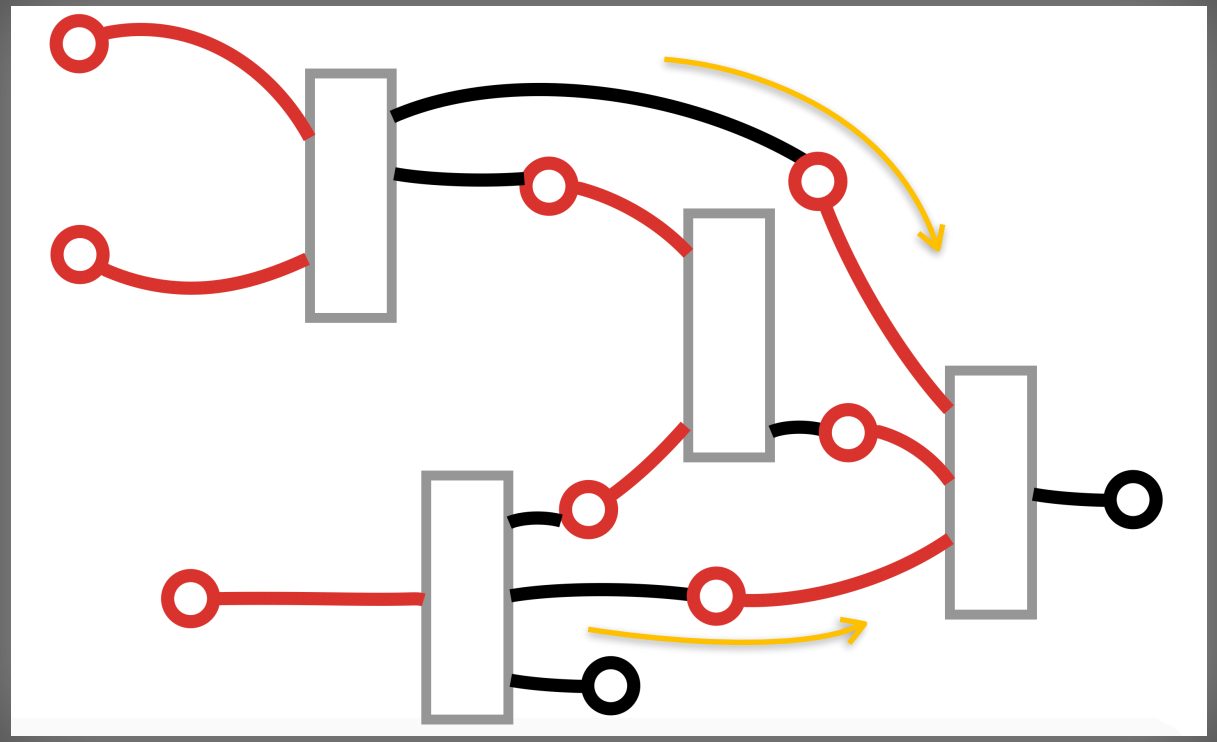
Marlowe Playground: develop, analyse and simulate.

Marlowe on Cardano blockchain

Extended UTxO.

Marlowe interpreter as a single Plutus program.

Largest such program.



```
contribute :: Campaign -> Value -> MockWallet ()
contribute campaign value = do
  when (value <= 0) $
    throwOtherError "Must contribute a positive value"

  ownPK <- ownPubKey
  tx    <- payToScript
             (Ledger.scriptAddress (contributionScript
campaign))
```

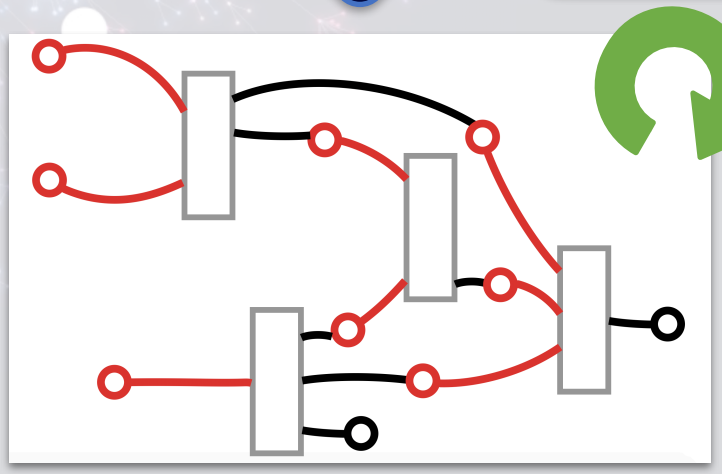
```
      Ledger.fromCompiledCode $$(PlutusTx.compile
        [|| (\Campaign{..} action contrib tx ->
          let
            PendingTx ps outs _ _ (Height h) _ _ = tx
            isValid = case action of
                        Refund ->  h >
```

User's wallet

User's wallet will generate transactions to go onto blockchain.

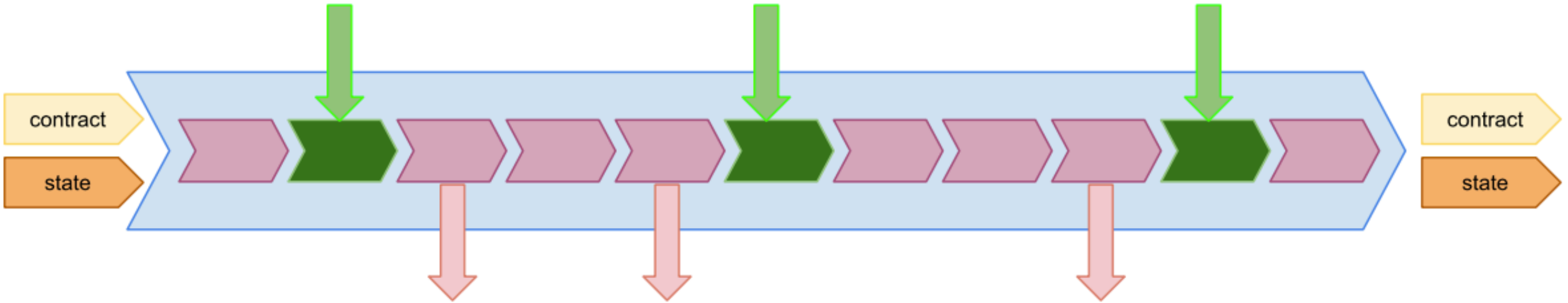
Plutus Core code is used to validate constraints on transactions.

PlutusTX is a subset of Haskell, and is compiled into Plutus Core, that runs on the blockchain

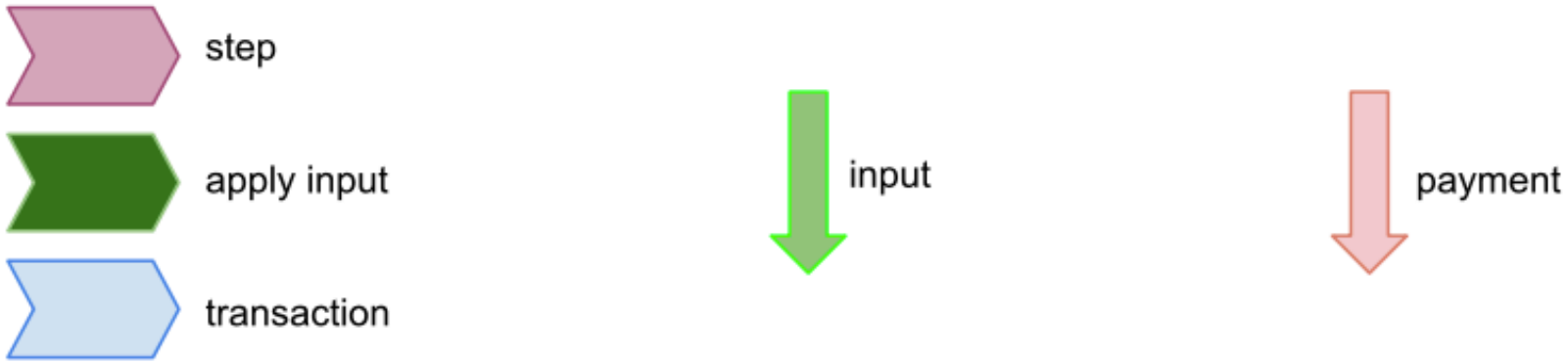


The basic ledger rules guarantee the integrity of the blockchain.

Building a transaction



Legend:



How can we write real contracts?

Embedded domain-specific language

```
When aliceChoice
    (When bobChoice
        (If (aliceChosen `ValueEQ` bobChosen)
            If (isPay aliceChosen)
               (Pay "alice" (Party "bob") price …)
               Close
            arbitrate)
```

Embedded domain-specific language

```
When aliceChoice
    (When bobChoice
        (If (aliceChosen `ValueEQ` bobChosen)
            If (isPay aliceChosen)
                (Pay "alice" (Party "bob") price …)
                Close
            arbitrate)
```

Embedded domain-specific language

```
When aliceChoice
    (When bobChoice
        (If (aliceChosen `ValueEQ` bobChosen)
            If (isPay aliceChosen)
                (Pay "alice" (Party "bob") price …)
                Close
            arbitrate)
```

```
-- Name actions

aliceChoice :: Action
aliceChoice = choice "alice" both
```

Embedded domain-specific language

```
When aliceChoice
    (When bobChoice
        (If (aliceChosen `ValueEQ` bobChosen)
            If (isPay aliceChosen)
                (Pay "alice" (Party "bob") price …)
                Close
            arbitrate)
```

```
-- Name actions

aliceChoice :: Action
aliceChoice = choice "alice" both
```

```
-- Predicate for choices

isPay :: Value -> Bool
isPay x = x `ValueEQ` (Constant 0)
```

OUTPUT

Embedded

```
-- Carol has to arbitrate between Alice and Bob.

arbitrate :: Contract
arbitrate =
  When  [ Case carolRefund Close,
          Case carolPay
               (Pay "alice" (Party "bob") price Close) ]
        100
        Close
```

```
When aliceChoice
    (When bobChoice
         (If (aliceChosen `ValueEQ` bobChosen)
             If (isPay aliceChosen)
                (Pay "alice" (Party "bob") price …)
                Close
             arbitrate)
```

```
-- Name actions

aliceChoice :: Action
aliceChoice = choice "alice" both
```

```
-- Predicate for choices

isPay :: Value -> Bool
isPay x = x `ValueEQ` (Constant 0)
```

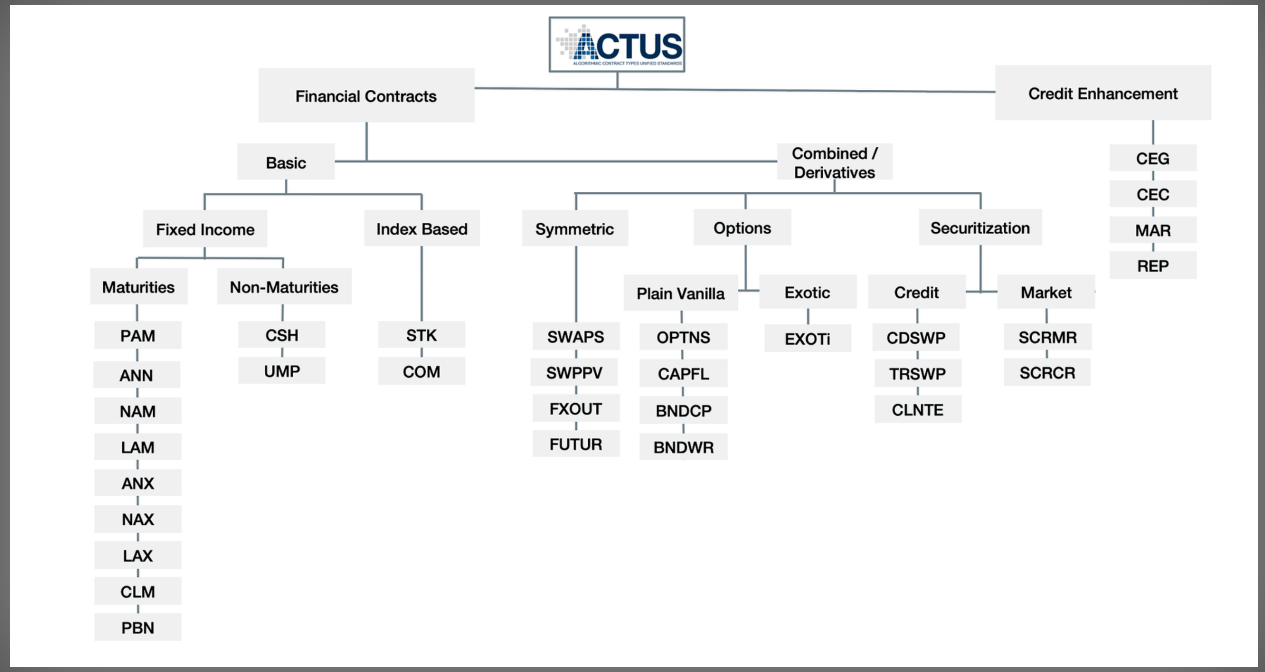
ACTUS

Financial standard.

Onto blockchain …

… into Marlowe.

https://www.actusfrf.org



Zero coupon bond

This ZCB bond doesn’t pay interest, but delivers a profit on maturity.

Zero coupon bond

This ZCB bond doesn’t pay interest, but delivers a profit on maturity.

The bond is made by the `issuer` to an `investor`. At `startDate` the `issuer` receives the `notional` value minus a `discount`, and they should pay back the full `notional` at `maturityDate`.

Guarantees

How do we ensure that the bond is redeemed?

- The `issuer` makes an up-front commitment
- No guarantee: the `investor` takes a risk.
- A third party `guarantor`: analogous to collateral.

```
zeroCouponBondGuaranteed :: Party -> Party -> Party -> Integer -> Integer -> Timeout -> Timeout -> Contract
zeroCouponBondGuaranteed issuer investor guarantor notional discount startDate maturityDate =
  When [ Case (Deposit guarantorAcc guarantor (Constant notional))
              (When [Case (Deposit investorAcc investor (Constant (notional - discount)))
                          (When []
                               startDate
                               (Pay investorAcc (Party issuer) (AvailableMoney investorAcc)
                                    (When [ Case (Deposit investorAcc issuer (Constant notional))
                                                 Close
                                          ]
                                          maturityDate
                                          (Pay guarantorAcc (Account investorAcc) (AvailableMoney guarantorAcc) Close)
                           ))))]
                    startDate
                    Close)
         ]
        startDate
        Close
```

```
zeroCouponBondGuaranteed :: Party -> Party -> Party -> Integer -> Integer -> Timeout -> Timeout -> Contract
zeroCouponBondGuaranteed issuer investor guarantor notional discount startDate maturityDate =
  When [ Case (Deposit guarantorAcc guarantor (Constant notional))
              (When [Case (Deposit investorAcc investor (Constant (notional - discount)))
                          (When []
                                startDate
                                (Pay investorAcc (Party issuer) (AvailableMoney investorAcc)
                                     (When [ Case (Deposit investorAcc issuer (Constant notional))
                                                  Close
                                           ]
                                           maturityDate
                                           (Pay guarantorAcc (Account investorAcc) (AvailableMoney guarantorAcc) Close)
                          ))]
                    startDate
                    Close)
         ]
       startDate
       Close
```

```
zeroCouponBondGuaranteed :: Party -> Party -> Party -> Integer -> Integer -> Timeout -> Timeout -> Contract
zeroCouponBondGuaranteed issuer investor guarantor notional discount startDate maturityDate =
  When [ Case (Deposit guarantorAcc guarantor (Constant notional))
              (When [Case (Deposit investorAcc investor (Constant (notional - discount)))
                          (When []
                                startDate
                                (Pay investorAcc (Party issuer) (AvailableMoney investorAcc)
                                     (When [ Case (Deposit investorAcc issuer (Constant notional))
                                                  Close
                                           ]
                                           maturityDate
                                           (Pay guarantorAcc (Account investorAcc) (AvailableMoney guarantorAcc) Close)
                          ))))]
                startDate
                Close)
    ]
  startDate
  Close
```

```
zeroCouponBondGuaranteed :: Party -> Party -> Party -> Integer -> Integer -> Timeout -> Timeout -> Contract
zeroCouponBondGuaranteed issuer investor guarantor notional discount startDate maturityDate =
  When [ Case (Deposit guarantorAcc guarantor (Constant notional))
              (When [Case (Deposit investorAcc investor (Constant (notional - discount)))
                          (When []
                                startDate
                                (Pay investorAcc (Party issuer) (AvailableMoney investorAcc)
                                     (When [ Case (Deposit investorAcc issuer (Constant notional))
                                                  Close
                                           ]
                                           maturityDate
                                           (Pay guarantorAcc (Account investorAcc) (AvailableMoney guarantorAcc) Close)
                          ))]
                    startDate
                    Close)
         ]
       startDate
       Close
```

```
zeroCouponBondGuaranteed :: Party -> Party -> Party -> Integer -> Integer -> Timeout -> Timeout -> Contract
zeroCouponBondGuaranteed issuer investor guarantor notional discount startDate maturityDate =
  When [ Case (Deposit guarantorAcc guarantor (Constant notional))
              (When [Case (Deposit investorAcc investor (Constant (notional - discount)))
                          (When []
                                startDate
                                (Pay investorAcc (Party issuer) (AvailableMoney investorAcc)
                                     (When [ Case (Deposit investorAcc issuer (Constant notional))
                                                  Close
                                           ]
                                           maturityDate
                                           (Pay guarantorAcc (Account investorAcc) (AvailableMoney guarantorAcc) Close)
                           ))))]
                    startDate
                    Close)
         ]
       startDate
       Close
```

```
zeroCouponBondGuaranteed :: Party -> Party -> Party -> Integer -> Integer -> Timeout -> Timeout -> Contract
zeroCouponBondGuaranteed issuer investor guarantor notional discount startDate maturityDate =
  When [ Case (Deposit guarantorAcc guarantor (Constant notional))
              (When [Case (Deposit investorAcc investor (Constant (notional - discount)))
                          (When []
                                startDate
                                (Pay investorAcc (Party issuer) (AvailableMoney investorAcc)
                                     (When [ Case (Deposit investorAcc issuer (Constant notional))
                                                  Close
                                           ]
                                           maturityDate
                                           (Pay guarantorAcc (Account investorAcc) (AvailableMoney guarantorAcc) Close)
                           ))))]
                    startDate
                    Close)
         ]
       startDate
       Close
```

```
zeroCouponBondGuaranteed :: Party -> Party -> Party -> Integer -> Integer -> Timeout -> Timeout -> Contract
zeroCouponBondGuaranteed issuer investor guarantor notional discount startDate maturityDate =
  When [ Case (Deposit guarantorAcc guarantor (Constant notional))
              (When [Case (Deposit investorAcc investor (Constant (notional - discount)))
                          (When []
                                startDate
                                (Pay investorAcc (Party issuer) (AvailableMoney investorAcc)
                                     (When [ Case (Deposit investorAcc issuer (Constant notional))
                                                  Close
                                           ]
                                           maturityDate
                                           (Pay guarantorAcc (Account investorAcc) (AvailableMoney guarantorAcc) Close)
                           ))))]
                    startDate
                    Close)
         ]
       startDate
       Close
```

Visually … The Marlowe Playground

The Marlowe Playground: browser-based tool

Develop embedded contracts

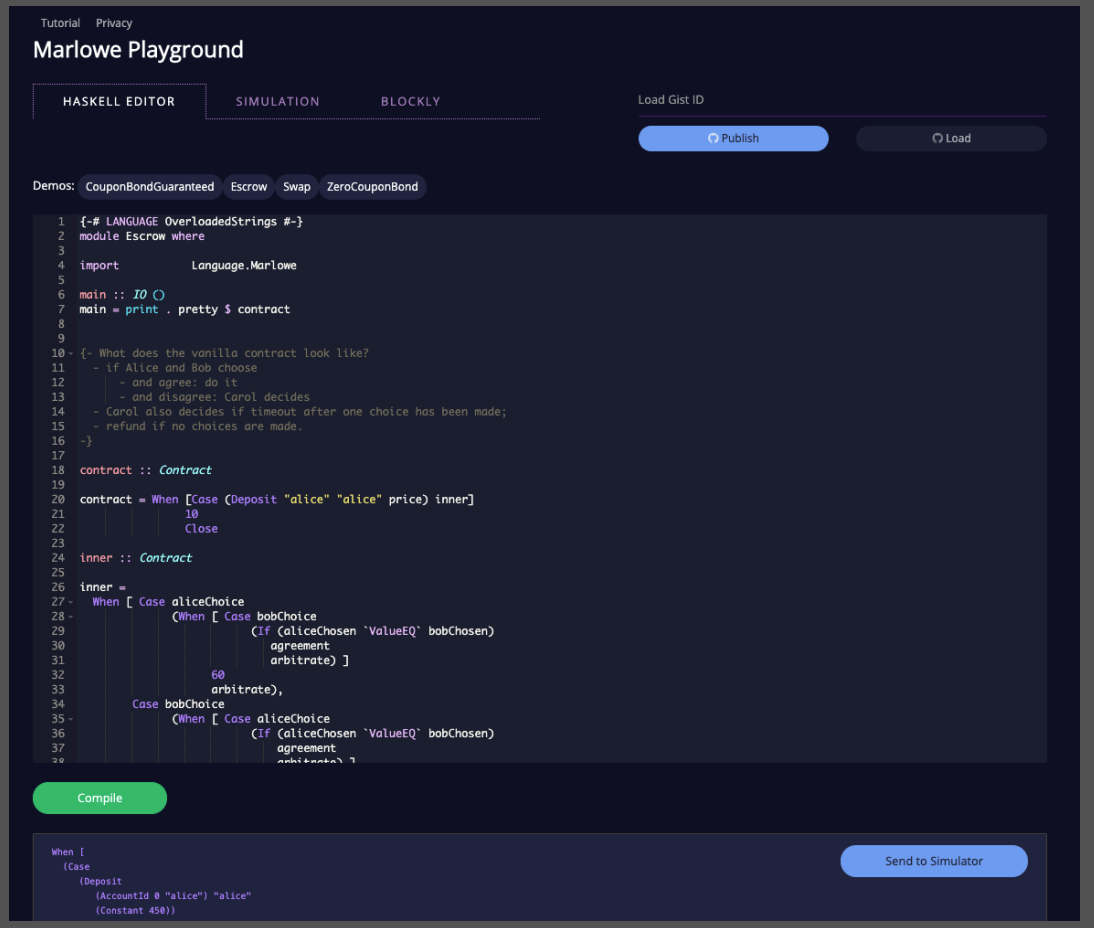
Convert to pure Marlowe

Analysis in the cloud

Simulate with smart inputs

Develop in Blockly

alpha.marlowe.iohkdev.io



Tutorial Privacy

Marlowe Playground

HASKELL EDITOR | SIMULATION | BLOCKLY

Load Gist ID

Publish | Load

Demos: CouponBondGuaranteed | Escrow | Swap | ZeroCouponBond

```
{-# LANGUAGE OverloadedStrings #-}
module Escrow where

import           Language.Marlowe

main :: IO ()
main = print . pretty $ contract


{- What does the vanilla contract look like?
  - if Alice and Bob choose
    - and agree: do it
    - and disagree: Carol decides
  - Carol also decides if timeout after one choice has been made;
  - refund if no choices are made.
-}

contract :: Contract

contract = When [Case (Deposit "alice" "alice" price) inner]
                10
                Close

inner :: Contract

inner =
  When [ Case aliceChoice
              (When [ Case bobChoice
                           (If (aliceChosen `ValueEQ` bobChosen)
                              agreement
                              arbitrate) ]
                    60
                    arbitrate),
```

OUTPUT

```
module Escrow where

import           Language.Marlowe

main :: IO ()
main = print . pretty $ contract


{- What does the vanilla contract look like?
  - if Alice and Bob choose
    | - and agree: do it
    | - and disagree: Carol decides
  - Carol also decides if timeout after one choice has been made;
  - refund if no choices are made.
-}

contract :: Contract

contract = When [Case (Deposit "alice" "alice" price) inner]
                10
                Close

inner :: Contract

inner =
  When [ Case aliceChoice
              (When [ Case bobChoice
                           (If (aliceChosen `ValueEQ` bobChosen)
                              agreement
                              arbitrate) ]
                    60
                    arbitrate),
         Case bobChoice
              (When [ Case aliceChoice
                           (If (aliceChosen `ValueEQ` bobChosen)
                               agreement
                               arbitrate) ]
```

Compile

```
When [
  (Case
     (Deposit
        (AccountId 0 "alice") "alice"
        (Constant 450))
```

Send to Simulator

Tutorial Privacy

Marlowe Playground

HASKELL EDITOR | SIMULATION | BLOCKLY

Load Gist ID

Publish | Load

Input Composer

Person "carol"

+ Choice **choice** : Choose value 0
+ Choice **choice** : Choose value 1

Transaction Composer

Input list

No inputs in the transaction

Apply Transaction | Next Block | Reset | Undo

State

Current Block: 0 Money in contract: 450 ADA

Accounts

Account id	Participant	Money
0	alice	450

Choices

Choice id	Participant	Chosen value
choice	alice	1
choice	bob	0

Payments

No payments have been recorded

Marlowe Contract

Code to Blockly

Demos: CouponBondGuaranteed | Escrow | Swap | ZeroCouponBond

Input Composer

Person "carol"

+ Choice **choice** : Choose value 0
+ Choice **choice** : Choose value 1

Transaction Composer

Input list

No inputs in the transaction

Apply Transaction | Next Block | Reset | Undo

State

Current Block: 0 Money in contract: 450 ADA

Accounts

Account id	Participant	Money
0	alice	450

Choices

Choice id	Participant	Chosen value
choice	alice	1
choice	bob	0

Payments

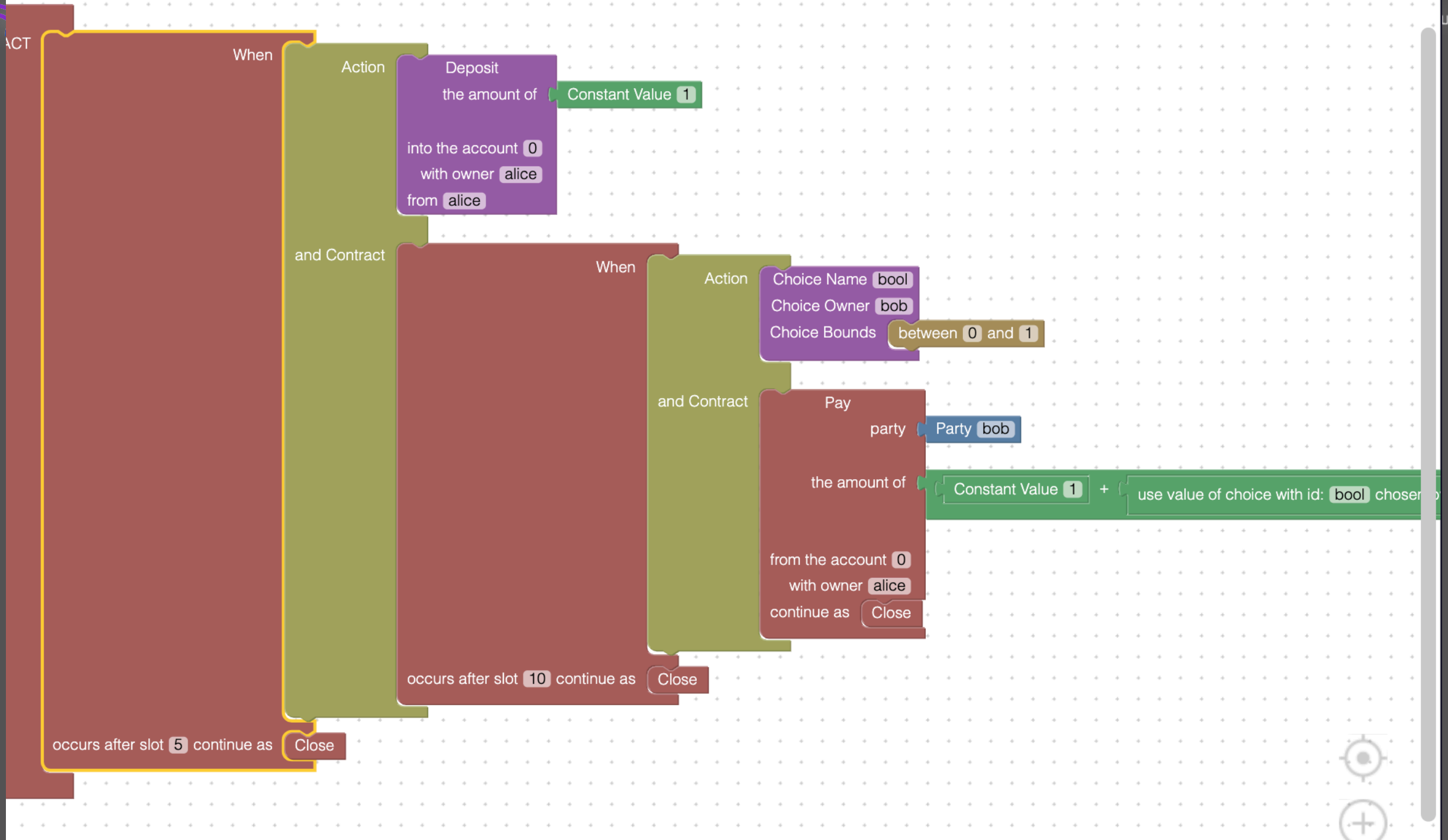
No payments have been recorded

Marlowe Contract

Code to Blockly

Demos: CouponBondGuaranteed | Escrow | Swap | ZeroCouponBond

```
When [Case (Choice (ChoiceId "choice" "carol") [(Bound 1 1)])
     Close
     ,Case (Choice (ChoiceId "choice" "carol") [(Bound 0 0)])

     (Pay (AccountId 0 "alice")
        (Party "bob")
        (Constant 450) Close)] 100 Close
```


When
Action
Deposit
the amount of
Constant Value 1
into the account 0
with owner alice
from alice
and Contract
When
Action
Choice Name bool
Choice Owner bob
Choice Bounds
between 0 and 1
and Contract
Pay
party
Party bob
the amount of
Constant Value 1
+
use value of choice with id: bool
from the account 0
with owner alice
continue as
Close
occurs after slot 10 continue as
Close
occurs after slot 5 continue as
Close

Analysis

Marlowe Contract

Code to Blockly

Demos: CouponBondGuaranteed | Escrow | Swap | ZeroCouponBond

```
When [Case (Choice (ChoiceId "choice" "carol") [(Bound 1 1)])
     Close
     ,Case (Choice (ChoiceId "choice" "carol") [(Bound 0 0)])

     (Pay (AccountId 0 "alice")
        (Party "bob")
        (Constant 450) Close)] 100 Close
```

Static analysis

Analysis Result: Pass

Static analysis could not find any execution that results in any warning.

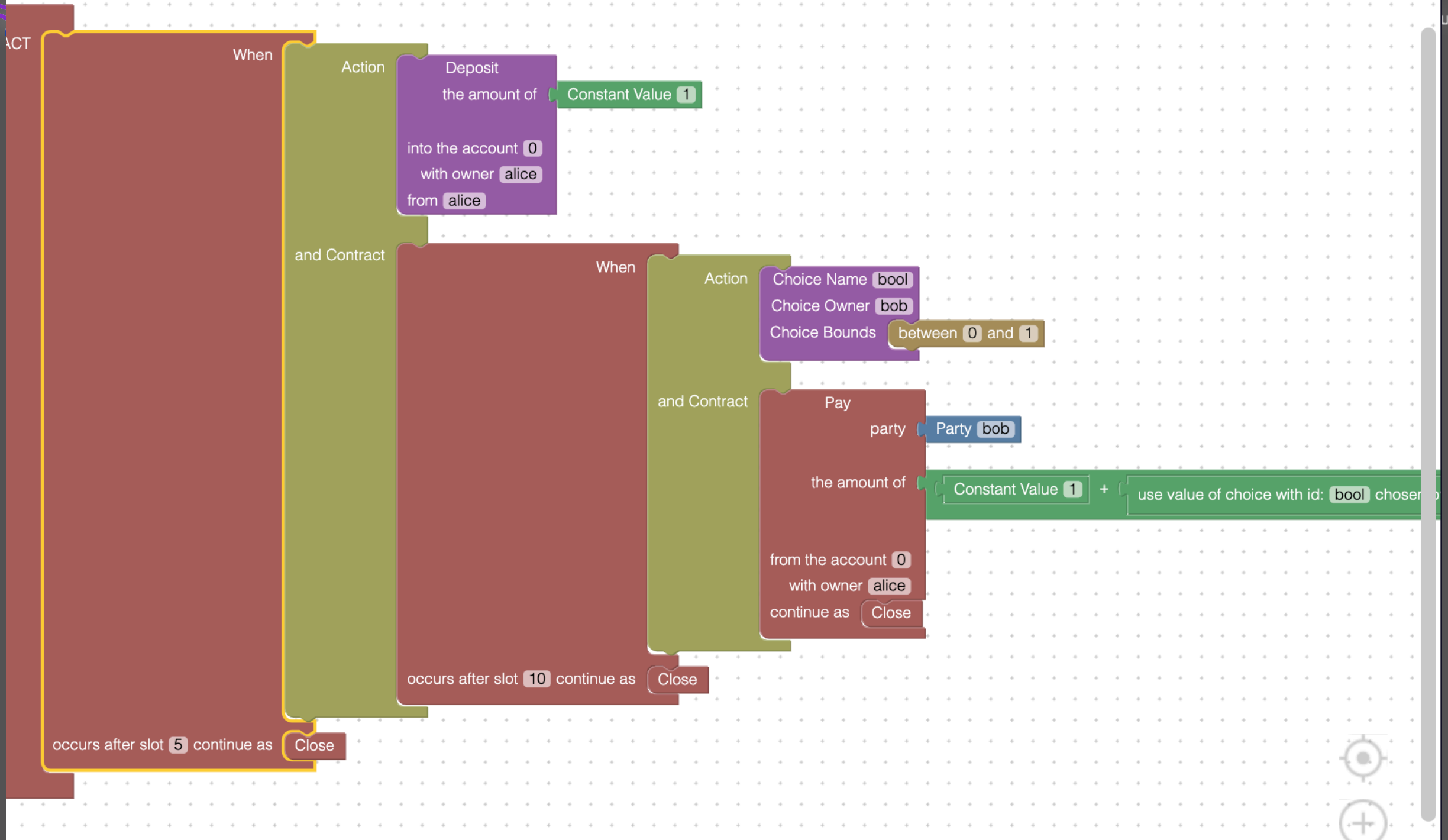
```
3       ,Case (Choice (ChoiceId "choice" "carol") [(Bound 0 0)])
4
5       (Pay (AccountId 0 "alice")
6          (Party "bob")
7          (Constant 450) Close)] 100 Close
```

Static analysis

Analysis Result: Pass

Static analysis could not find any execution that results in any warning.

Analyse Contract


When
Action
Deposit
the amount of
Constant Value 1
into the account 0
with owner alice
from alice
and Contract
When
Action
Choice Name bool
Choice Owner bob
Choice Bounds
between 0 and 1
and Contract
Pay
party
Party bob
the amount of
Constant Value 1
+
use value of choice with id: bool
from the account 0
with owner alice
continue as
Close
occurs after slot 10 continue as
Close
occurs after slot 5 continue as
Close

Marlowe Contract

Code to Blockly

```
When [Case (Deposit (AccountId 0 "alice") "alice" (Constant 1))

  (When [Case (Choice (ChoiceId "bool" "bob") [(Bound 0 1)])

     (Pay (AccountId 0 "alice")
        (Party "bob")
        (AddValue
           (Constant 1)
           (ChoiceValue (ChoiceId "bool" "bob")
             (Constant 0))) Close)] 10 Close)] 5 Close
```

Marlowe Contract

Code to Blockly

```
1  When [Case (Deposit (AccountId 0 "alice") "alice" (Constant 1))
2
3    (When [Case (Choice (ChoiceId "bool" "bob") [(Bound 0 1)])
4
5       (Pay (AccountId 0 "alice")
6          (Party "bob")
7          (AddValue
8             (Constant 1)
9             (ChoiceValue (ChoiceId "bool" "bob")
10              (Constant 0))) Close)] 10 Close)] 5 Close
```

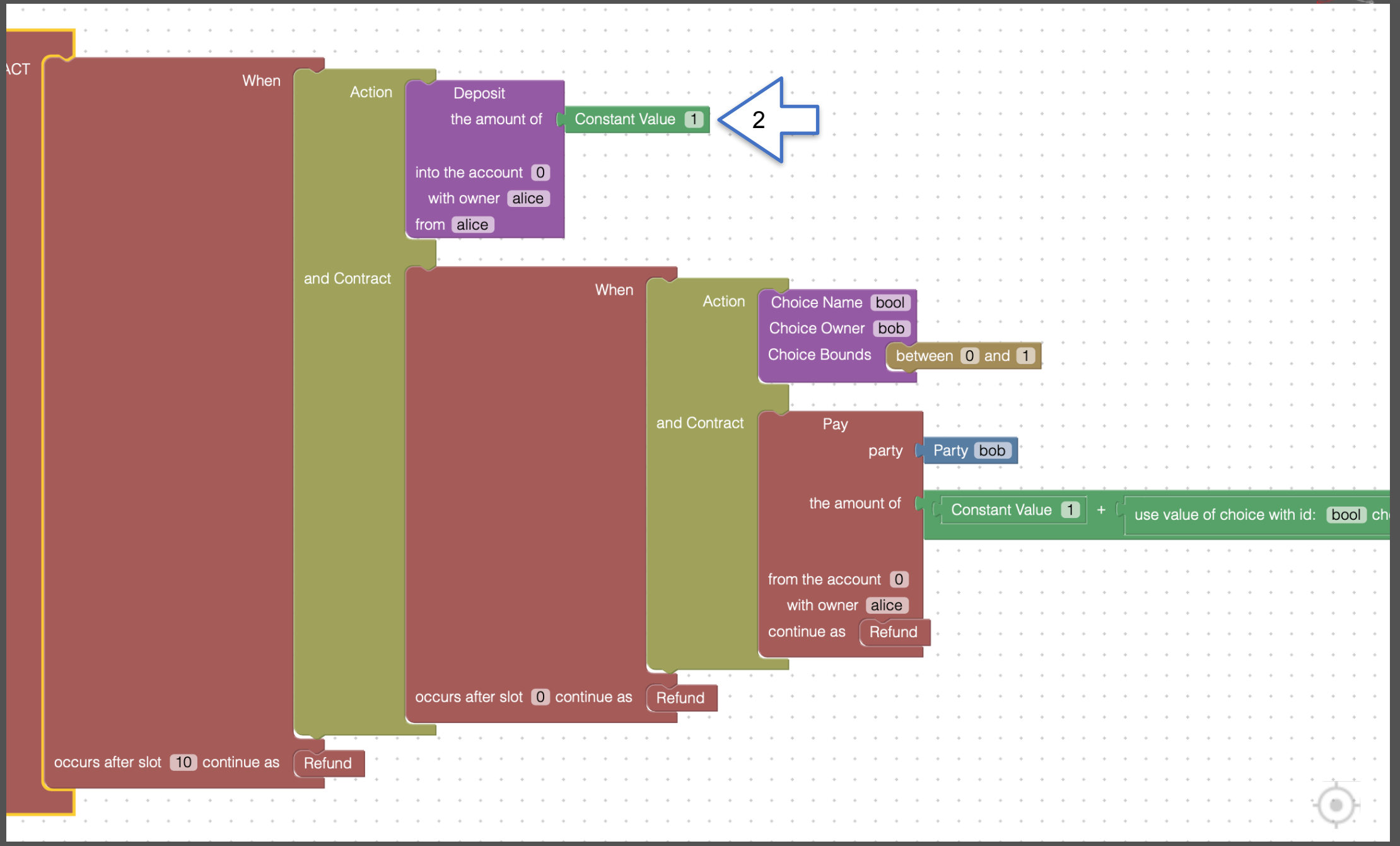
Static analysis

Analysis Result: Fail

Static analysis found the following counterexample:

- Initial slot: -11839
- Offending transaction list:
    1. Transaction with slot interval -9401 to -1036 and inputs:
        1. IDeposit - Party "alice" deposits 1 Lovelace into account 0 of "alice".
        2. IChoice - Party "bob" chooses number 1 for choice "bool".
- Warnings issued:
    1. TransactionPartialPay - The contract is suppoused to make a payment of 2 Lovelace from account 0 of "alice" to party "bob" but there is only 1 Lovelace.

Analyse Contract


When
Action
Deposit
the amount of
Constant Value 1
2
into the account 0
with owner alice
from alice
and Contract
When
Action
Choice Name bool
Choice Owner bob
Choice Bounds
between 0 and 1
and Contract
Pay
party
Party bob
the amount of
Constant Value 1
+
use value of choice with id: bool
from the account 0
with owner alice
continue as
Refund
occurs after slot 0 continue as
Refund
occurs after slot 10 continue as
Refund

Marlowe Contract

Code to Blockly

```
1  When [Case (Deposit (AccountId 0 "alice") "alice" (Constant 1))
2
3    (When [Case (Choice (ChoiceId "bool" "bob") [(Bound 0 1)])
4
5       (Pay (AccountId 0 "alice")
6          (Party "bob")
7          (AddValue
8             (Constant 1)
9             (ChoiceValue (ChoiceId "bool" "bob")
10               (Constant 0))) Close)] 10 Close)] 5 Close
```

Marlowe Contract

Code to Blockly

```
1  When [Case (Deposit (AccountId 0 "alice") "alice" (Constant 1))
2
3    (When [Case (Choice (ChoiceId "bool" "bob") [(Bound 0 1)])
4
5       (Pay (AccountId 0 "alice")
6          (Party "bob")
7          (AddValue
8             (Constant 1)
9             (ChoiceValue (ChoiceId "bool" "bob")
10              (Constant 0))) Close)] 10 Close)] 5 Close
```

Static analysis

Analysis Result: Pass

Static analysis could not find any execution that results in any warning.

Analyse Contract

Static analysis

Static analysis

Automatic and exhaustive.

When there's a problem we get a counter-example.

Works for particular *instances* of contracts: concrete payment schedules.

Formal verification in Isabelle

Formal verification

Done “by hand” but machine checked / supported. and exhaustive.

Prove properties of the platform: all accounts non-negative.

Can also prove properties of *templates* for contracts: all possible payment schedules.

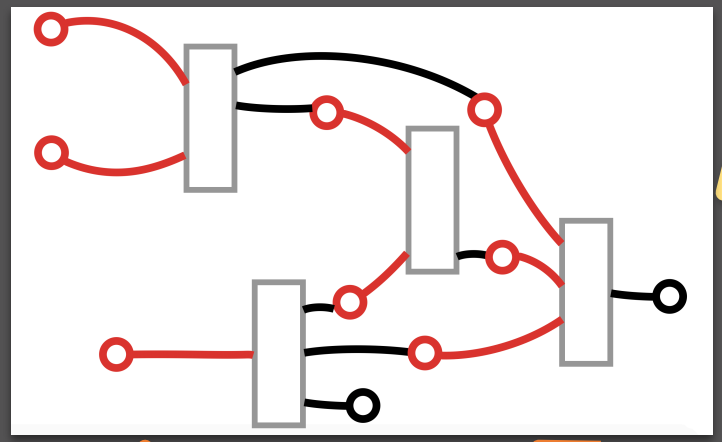
```
                        | ReducePartialPay AccountId Payee Money Money
                        | ReduceShadowing ValueId int int

datatype ReduceStepResult = Reduced ReduceWarning ReduceEffect State Contract
                          | NotReduced
                          | AmbiguousSlotIntervalReductionError

fun reduceContractStep :: "Environment ⇒ State ⇒ Contract ⇒ ReduceStepResult" where
"reduceContractStep _ state Close =
  (case refundOne (accounts state) of
     Some ((party, money), newAccount) ⇒
       let newState = state ⦇ accounts := newAccount ⦈ in
       Reduced ReduceNoWarning (ReduceWithPayment (Payment party money)) newState Close
   | None ⇒ NotReduced)" |
"reduceContractStep env state (Pay accId payee val cont) =
  (let moneyToPay = evalValue env state val in
   if moneyToPay ≤ 0
   then Reduced (ReduceNonPositivePay accId payee moneyToPay) ReduceNoPayment state cont
   else (let balance = moneyInAccount accId (accounts state) in
        (let paidMoney = min balance moneyToPay in
         let newBalance = balance - paidMoney in
         let newAccs = updateMoneyInAccount accId newBalance (accounts state) in
         let warning = (if paidMoney < moneyToPay
                        then ReducePartialPay accId payee paidMoney moneyToPay
                        else ReduceNoWarning) in
         let (payment, finalAccs) = giveMoney payee paidMoney newAccs in
         Reduced warning payment (state ⦇ accounts := finalAccs ⦈) cont)))" |
"reduceContractStep env state (If obs cont1 cont2) =
  (let cont = (if evalObservation env state obs
               then cont1
               else cont2) in
   Reduced ReduceNoWarning ReduceNoPayment state cont)" |
"reduceContractStep env state (When _ timeout cont) =
  (let (startSlot, endSlot) = slotInterval env in
```

```
   | None ⇒ NotReduced)" |
"reduceContractStep env state (Pay accId payee val cont) =
  (let moneyToPay = evalValue env state val in
   if moneyToPay ≤ 0
   then Reduced (ReduceNonPositivePay accId payee moneyToPay) ReduceNoPayment state cont
   else (let balance = moneyInAccount accId (accounts state) in
        (let paidMoney = min balance moneyToPay in
         let newBalance = balance - paidMoney in
         let newAccs = updateMoneyInAccount accId newBalance (accounts state) in
         let warning = (if paidMoney < moneyToPay
                        then ReducePartialPay accId payee paidMoney moneyToPay
                        else ReduceNoWarning) in
         let (payment, finalAccs) = giveMoney payee paidMoney newAccs in
         Reduced warning payment (state ⦇ accounts := finalAccs ⦈) cont)))" |
"reduceContractStep env state (If obs cont1 cont2) =
  (let cont = (if evalObservation env state obs
               then cont1
               else cont2) in
   Reduced ReduceNoWarning ReduceNoPayment state cont)" |
"reduceContractStep env state (When _ timeout cont) =
  (let (startSlot, endSlot) = slotInterval env in
   if endSlot < timeout
   then NotReduced
   else (if timeout ≤ startSlot
         then Reduced ReduceNoWarning ReduceNoPayment state cont
         else AmbiguousSlotIntervalReductionError))" |
"reduceContractStep env state (Let valId val cont) =
  (let evaluatedValue = evalValue env state val in
   let boundVals = boundValues state in
   let newState = state ⦇ boundValues := MList.insert valId evaluatedValue boundVals ⦈ in
   let warn = case lookup valId boundVals of
                Some oldVal ⇒ ReduceShadowing valId oldVal evaluatedValue
              | None ⇒ ReduceNoWarning in
   Reduced warn ReduceNoPayment newState cont)"
```

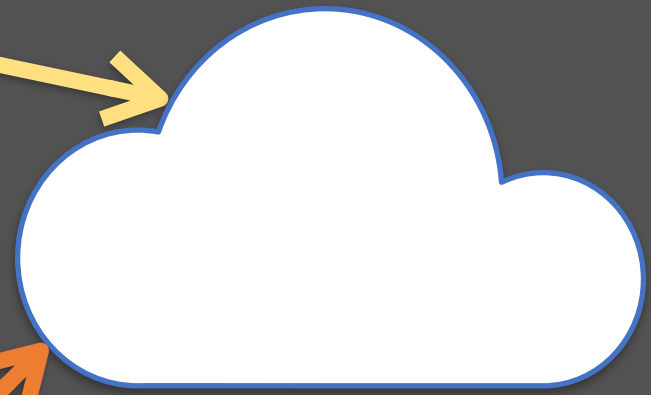
```
theorem computeTransaction_gtZero :
  "valid_state state ⟹
   (∀x. positiveMoneyInAccountOrNoAccount x (accounts state)) ⟹
   computeTransaction tx state contract = (TransactionOutput txOut) ⟹
   positiveMoneyInAccountOrNoAccount y (accounts (txOutState txOut))"

  apply (simp only:computeTransaction.simps)
  apply (cases "fixInterval (interval tx) state")
  apply (simp only:IntervalResult.case)
  subgoal for env state
    apply (simp only:Let_def)
    apply (cases "applyAllInputs env state contract (inputs tx)")
    apply (simp only:ApplyAllResult.case)
    apply (metis TransactionOutput.distinct(1) TransactionOutput.inject(1) TransactionOutputRecord.select_convs(3) applyAllI
    by auto
  by auto
```

```
theorem computeTransactionIsQuiescent : "validAndPositive_state sta ⟹
                                         computeTransaction traIn sta cont = TransactionOutput traOut ⟹
                                         isQuiescent (txOutContract traOut) (txOutState traOut)"

  apply (cases "traIn")
  apply (cases "traOut")
  by (smt IntervalResult.exhaust IntervalResult.simps(6) Transaction.update_convs(3) TransactionOutput.distinct(1) Transa
```

Looking forward …



```
theorem computeTransaction_gtZero :
  "valid_state state ⟹
   (∀x. positiveMoneyInAccountOrNoAccount x (accounts state)) ⟹
   computeTransaction tx state contract = (TransactionOutput txOut) ⟹
   positiveMoneyInAccountOrNoAccount y (accounts (txOutState txOut))"

  apply (simp only:computeTransaction.simps)
  apply (cases "fixInterval (interval tx) state")
  apply (simp only:IntervalResult.case)
  subgoal for env state
    apply (simp only:Let_def)
    apply (cases "applyAllInputs env state contract (inputs tx)")
    apply (simp only:ApplyAllResult.case)
    apply (metis TransactionOutput.distinct(1) TransactionOutput.inject(1) TransactionOutputRecord.select_convs(3) applyAllI
    by auto
  by auto
```



Tutorial Privacy

Marlowe Playground

HASKELL EDITOR | SIMULATION | BLOCKLY

Load Gist ID

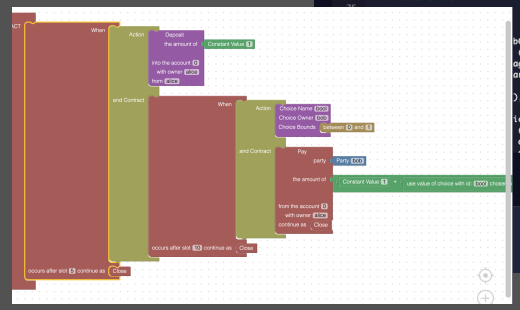
Publish | Load

Demos: CouponBondGuaranteed | Escrow | Swap | ZeroCouponBond

```
{-# LANGUAGE OverloadedStrings #-}
module Escrow where

import           Language.Marlowe

main :: IO ()
main = print . pretty $ contract


{- What does the vanilla contract look like?
   - if Alice and Bob choose
       - and agree: do it
       - and disagree: Carol decides
   - Carol also decides if timeout after one choice has been made;
   - refund if no choices are made.
-}

contract :: Contract

contract = When [Case (Deposit "alice" "alice" price) inner]
                10
                Close

inner :: Contract
```

```
bChoice
 (aliceChosen `ValueEQ` bobChosen)
agreement
arbitrate) ]

),

iceChoice
 (aliceChosen `ValueEQ` bobChosen)
agreement
```

Send to Simulator



Enhance: roles✓, tokenisation✓ and securitisation, using multi-currency.

Wallet Integration: use SCB + Marlowe Playground view. API gen / automation

Playground UX: working with IOHK web team on this, already briefed.

Scaling: efficiency of Purescript, libraries✓ SMT solving: SBV replacement.

Partners: ACTUS and quanterall; valuable feedback + examples.

Developer experience: playground *vs* IDE. Who are our users and customers?

Marlowe in a nutshell

An embedded DSL = a Haskell `data` type.

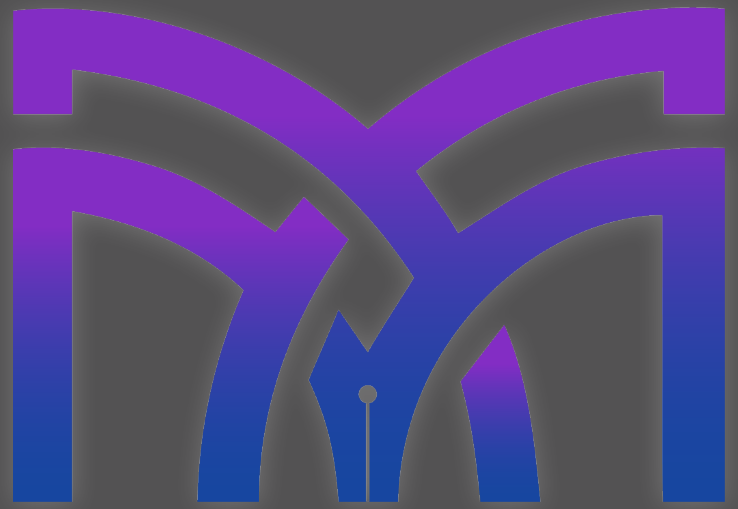
Executable small-step semantics.

Runs on extended UTxO blockchain.

Interpreter = Plutus program.

Analysis with Z3; proof in Isabelle.

Marlowe Playground: develop, analyse and simulate.



alpha.marlowe.iohkdev.io