SIMON THOMPSON, IOHK & KENT UNI

FUNCTIONAL PROGRAMMING FOR 3G BLOCKCHAIN

Elm

LISP

Erlang

Miranda

Idris

Scala

Haskell

F#

OCaml

Elixir

Elm

LISP

Erlang

Miranda

Idris

Scala

Haskell

F#

Elixir

OCaml

higher-order functions

pattern matching

data

types

lambdas

recursion

higher-order functions

pattern matching

data

types

lambdas

recursion

lenses

dependent types

reactive

higher-order functions

DSLs

lazy

pattern matching

data

types

monoids

types,
types,
types,
...

lambdas

recursion

effects

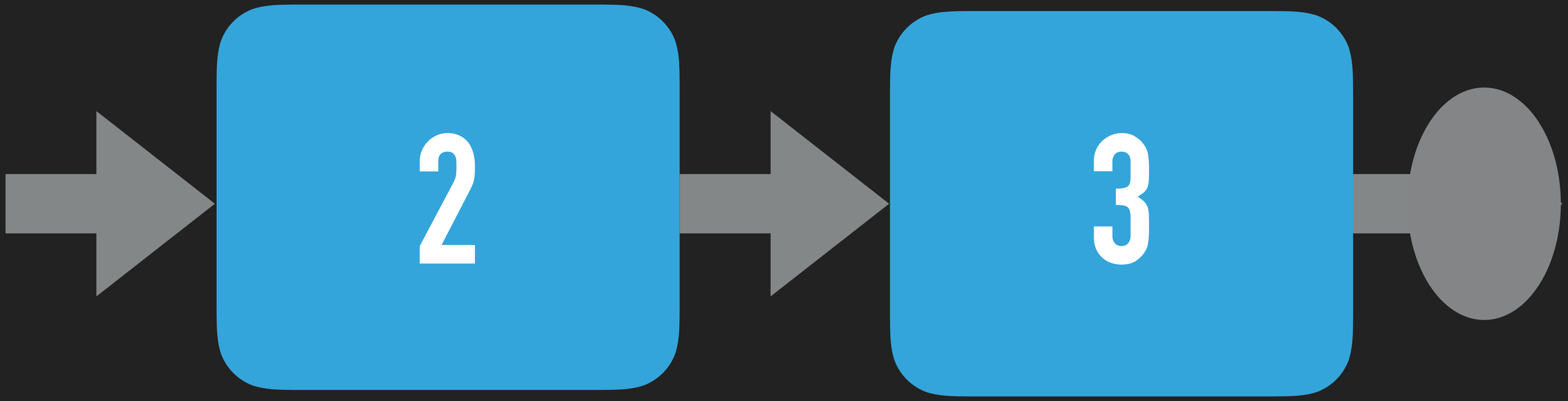
monads

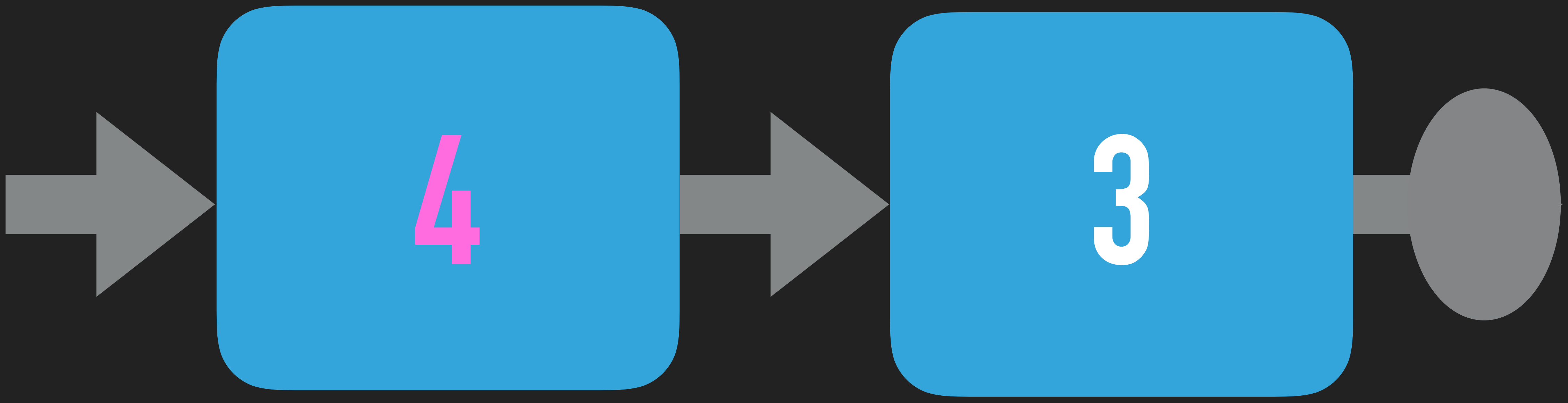
immutability

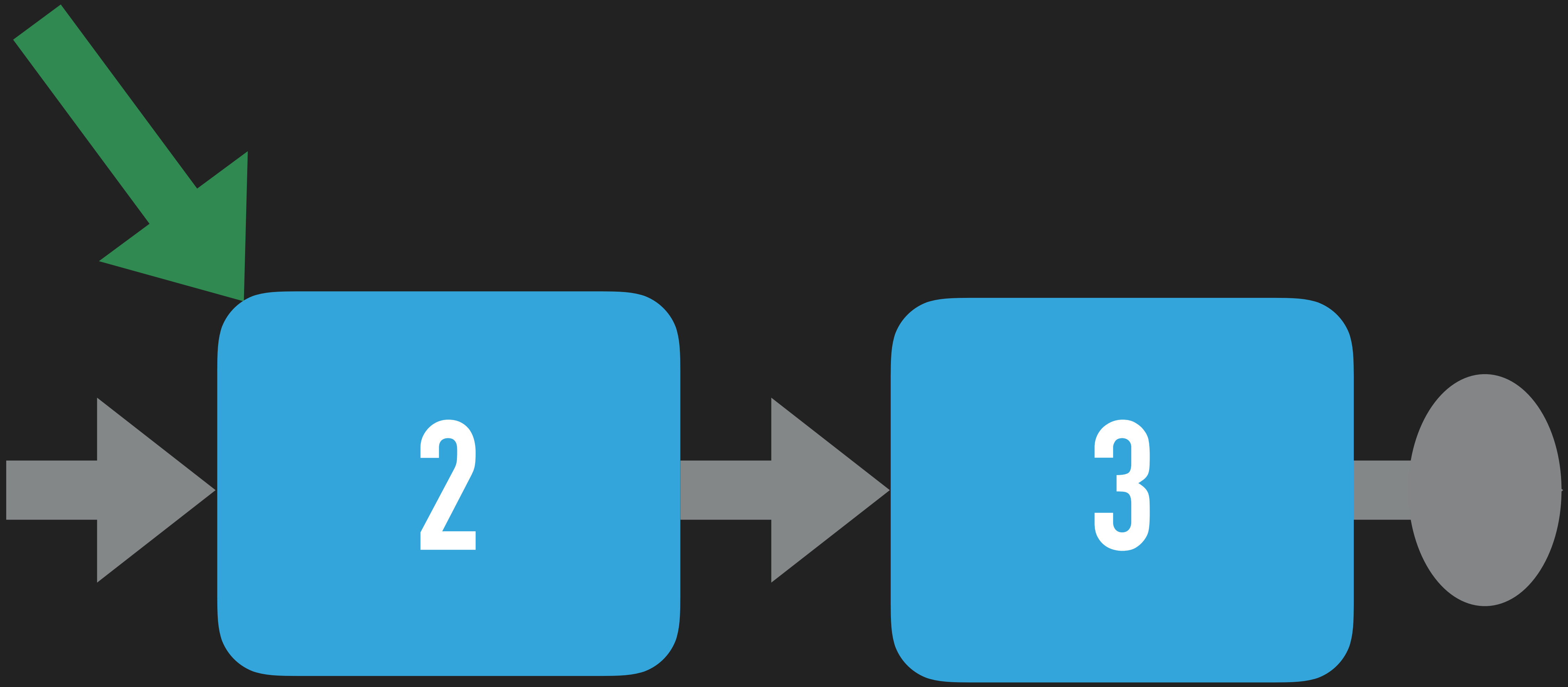
fun

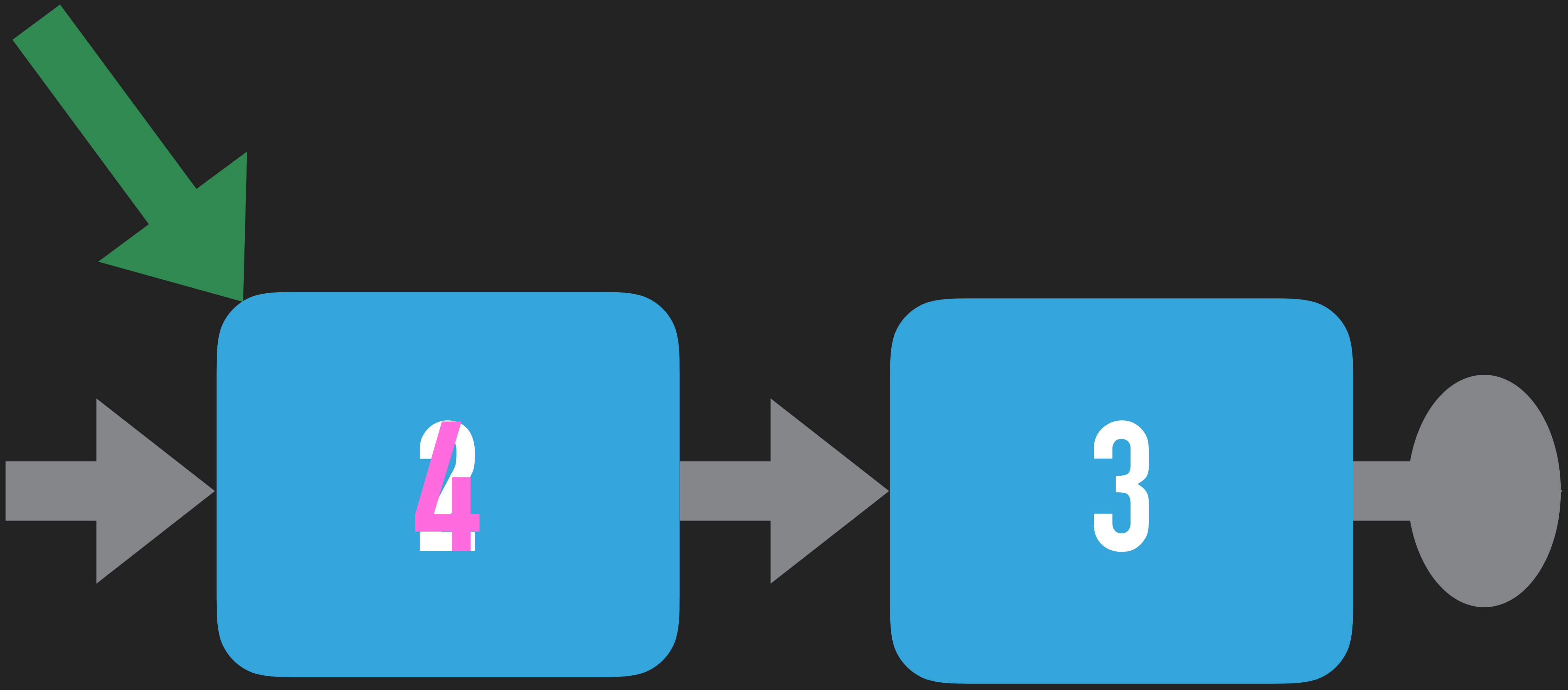
Model the world as data

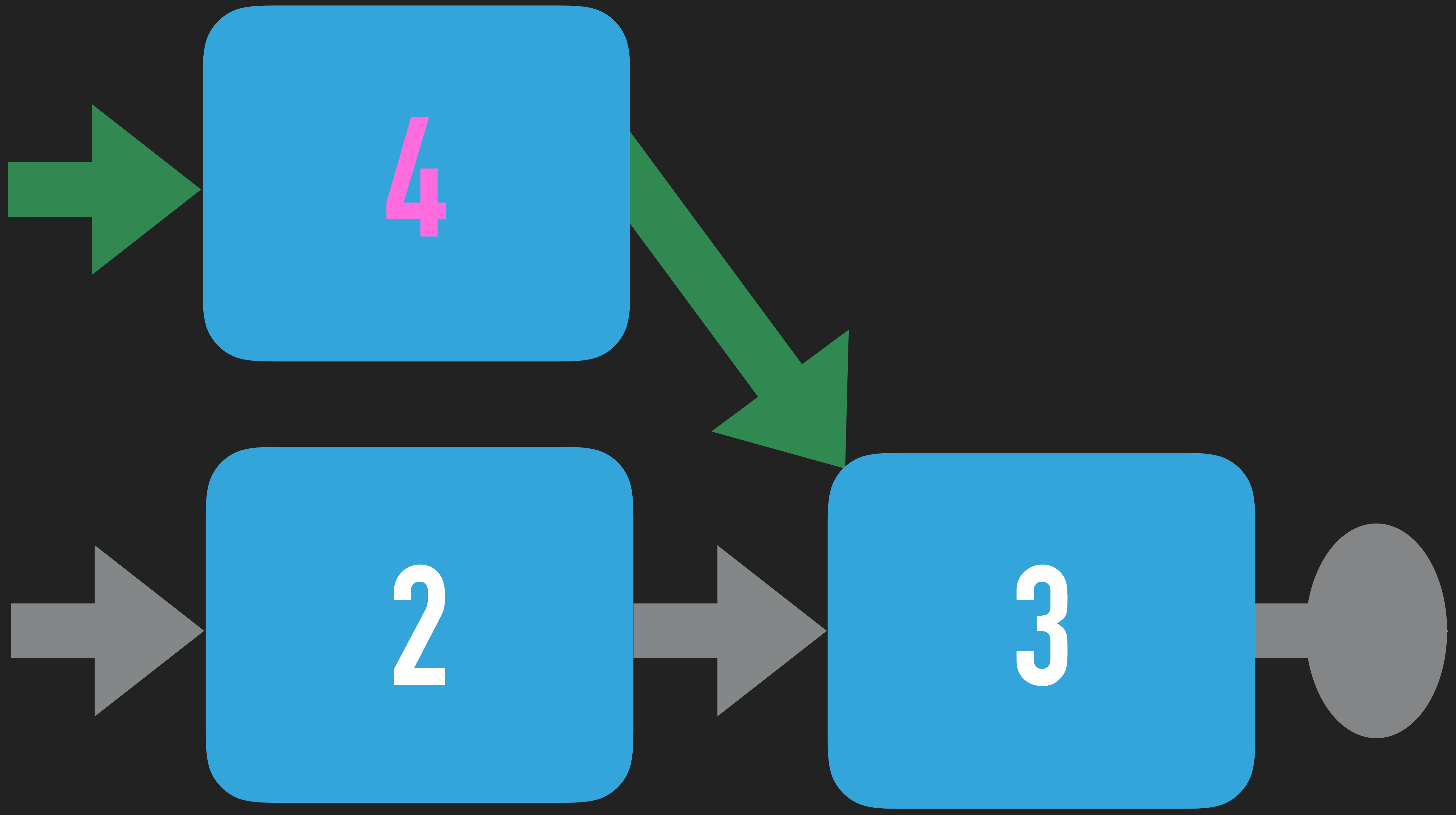
immutable data


2
3


4
3


2
3


2
4
3


4
2
3

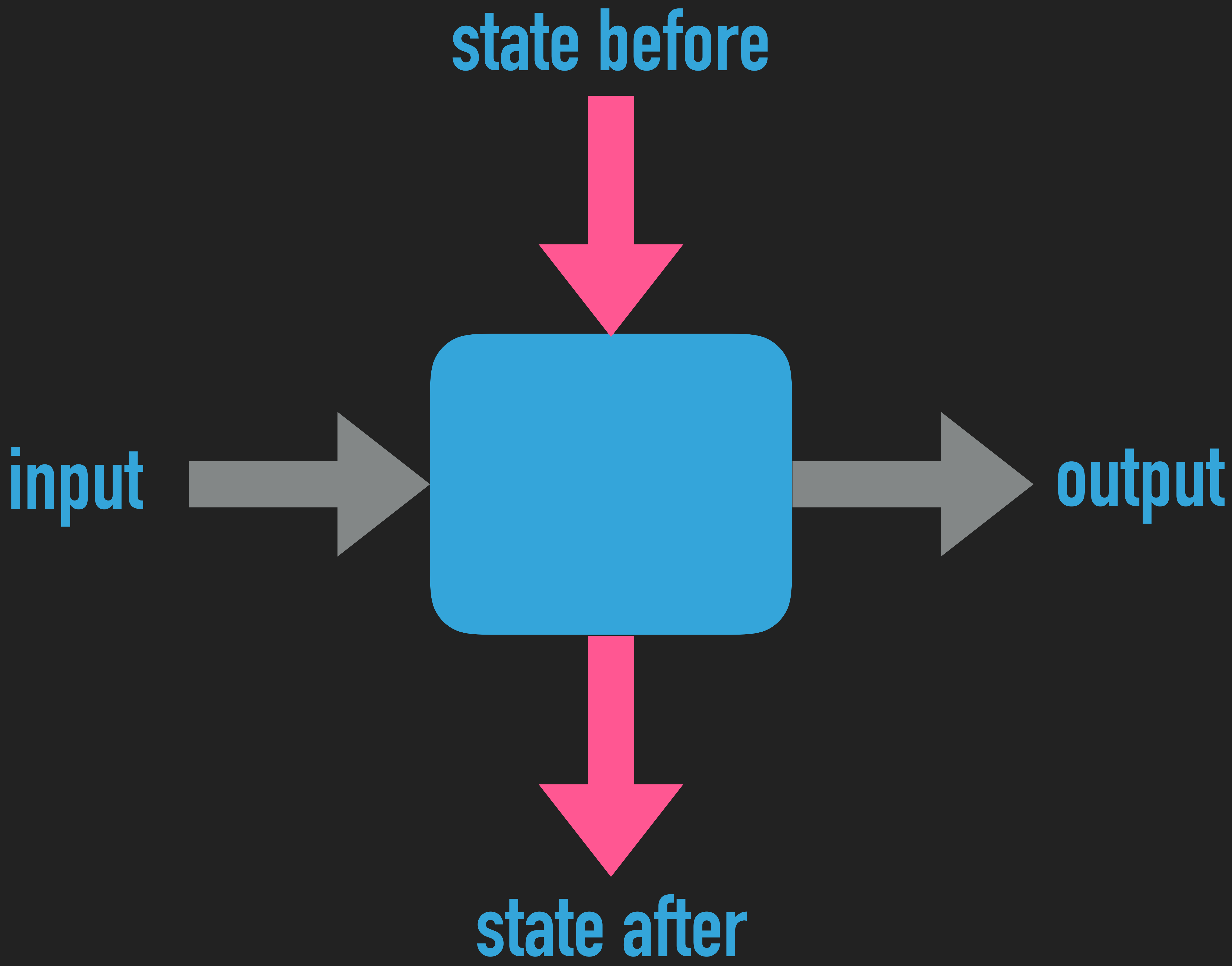
Model the world as data
+
Functions over the data

And when we say "function" . . .

We mean in the mathematical sense, taking inputs to outputs, and doing **nothing else**!

So what about side-effects?

input
output


state before
input
output
state after

Model the world as data

+

Functions over the data

+

Functions as data

Behaviour becomes data:

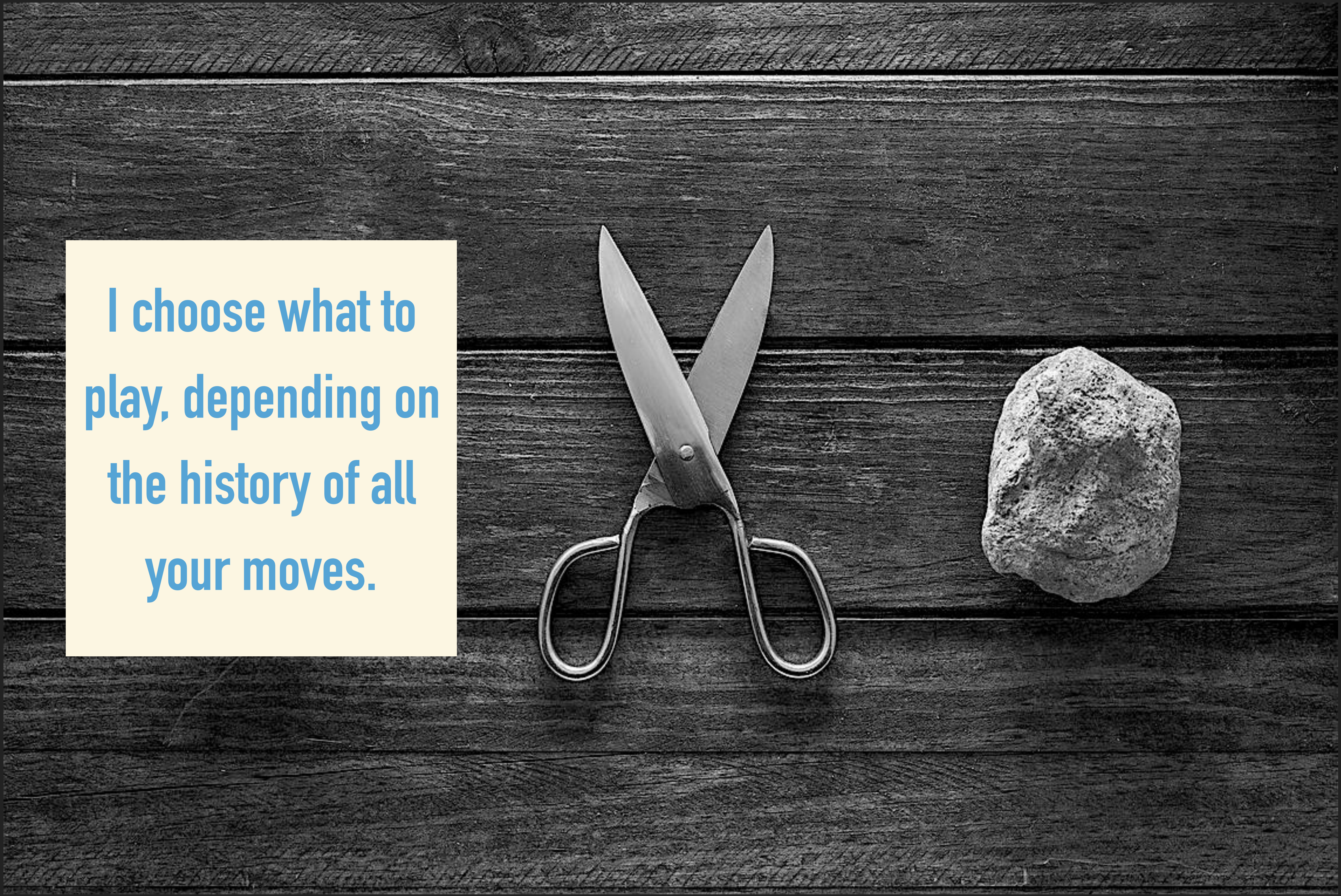
Behaviour becomes data: map/reduce, monads, APIs, laziness …

rock

paper

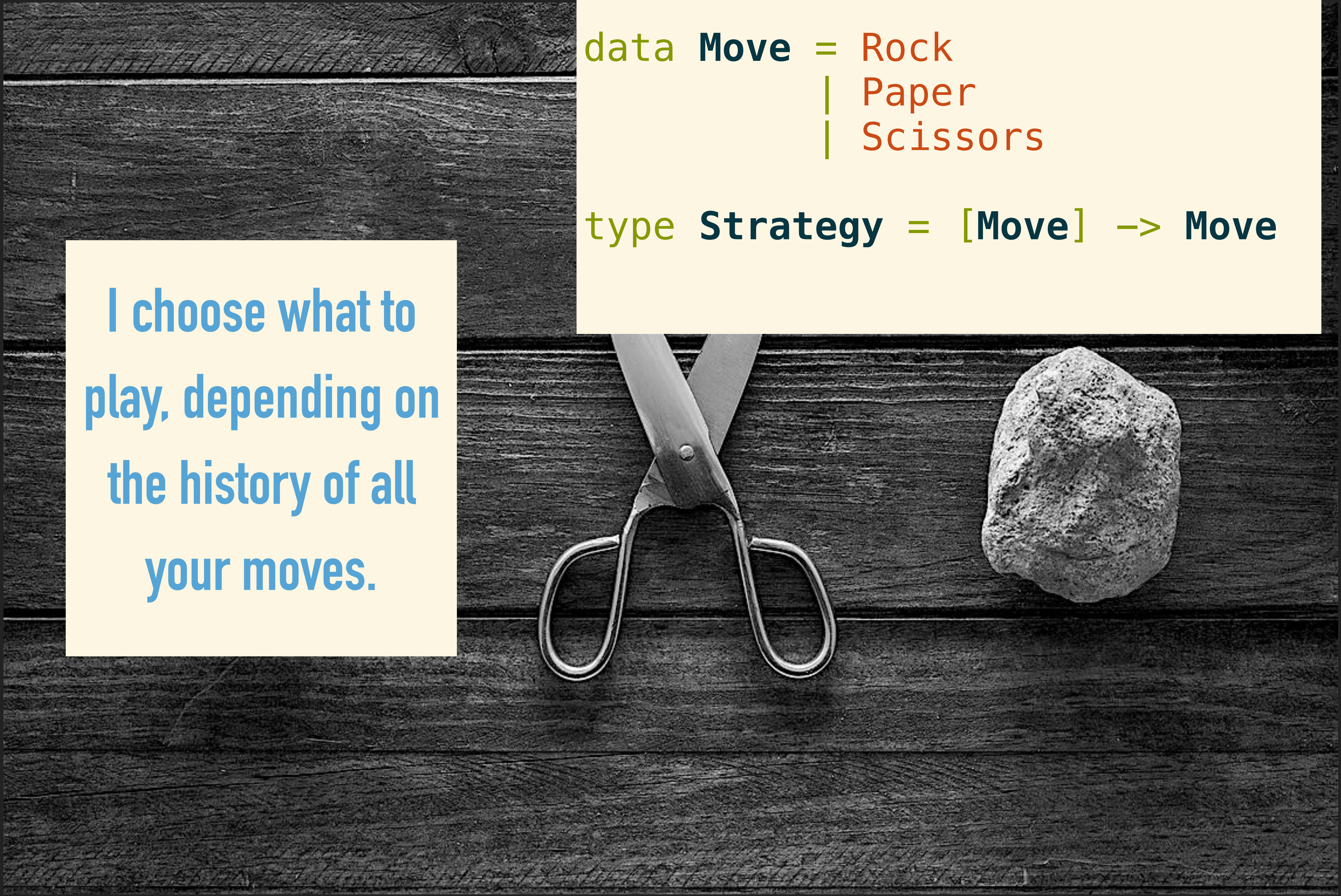
scissors

Original image: https://www.thishopeanchors.com/single-post/2017/04/06/Rock-Paper-Scissors



I choose what to play, depending on the history of all your moves.

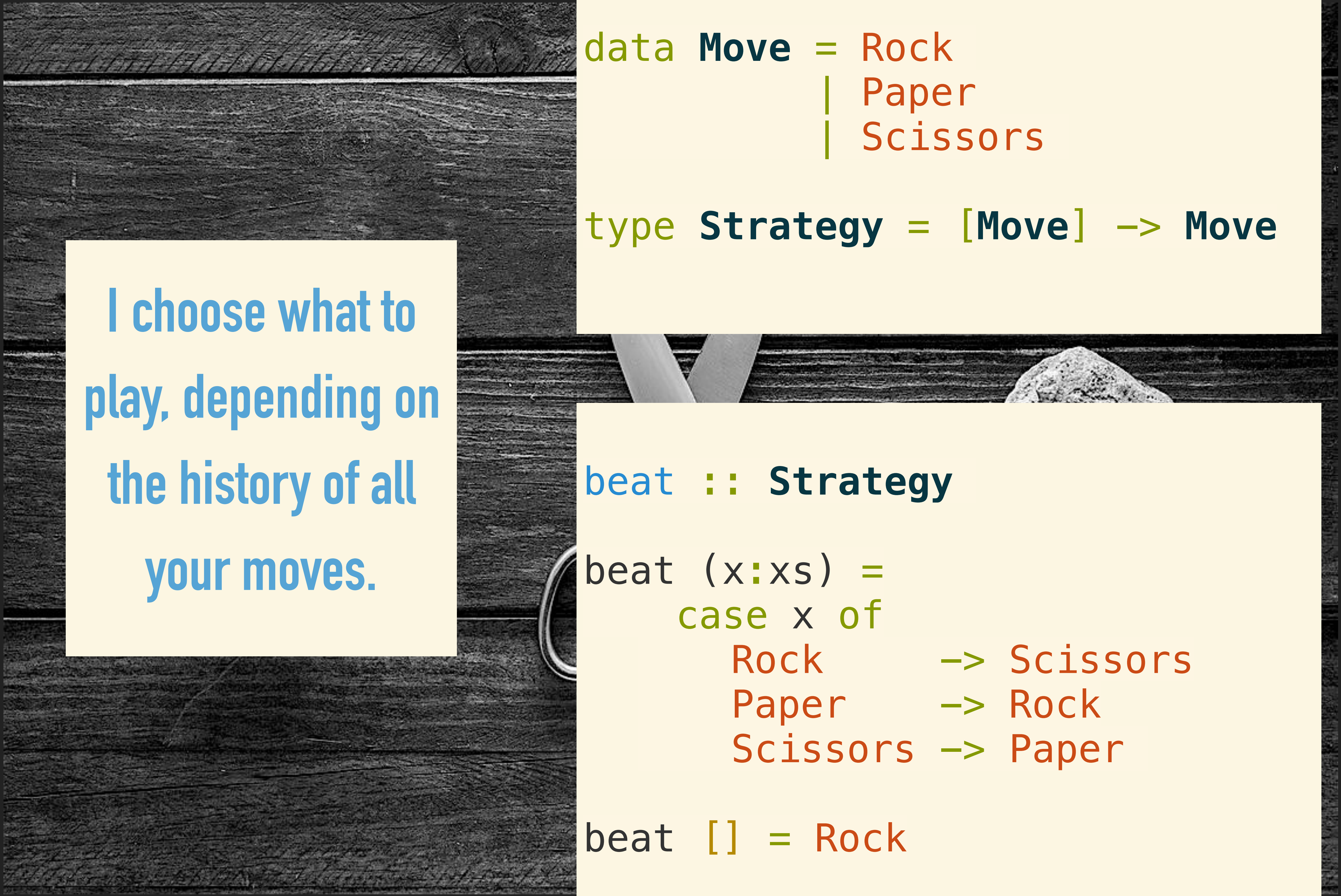
Original image: https://www.thishopeanchors.com/single-post/2017/04/06/Rock-Paper-Scissors



```haskell
data Move = Rock
          | Paper
          | Scissors

type Strategy = [Move] -> Move
```

I choose what to play, depending on the history of all your moves.

Original image: https://www.thishopeanchors.com/single-post/2017/04/06/Rock-Paper-Scissors



I choose what to play, depending on the history of all your moves.

```
data Move = Rock
          | Paper
          | Scissors

type Strategy = [Move] -> Move
```

```
beat :: Strategy

beat (x:xs) =
    case x of
      Rock     -> Scissors
      Paper    -> Rock
      Scissors -> Paper

beat [] = Rock
```

Original image: https://www.thishopeanchors.com/single-post/2017/04/06/Rock-Paper-Scissors

Original image: http://www.metso.com/services/spare-wear-parts-conveyors/conveyor-belts/

types

Functions give us expressivity

+

Types help to constrain that

+

Type-driven development

```haskell
type Point = (Float,Float)


data Shape = Circle Point Float
           | Rectangle Point Float Float


area :: Shape -> Float

area (Circle _ r)      = pi*r*r
area (Rectangle _ h w) = h*w
```

nub: remove all duplicates

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

type inference

type inference
polymorphism = generics

type inference

polymorphism = generics

type classes = overloading

type inference
polymorphism = generics
type classes = overloading
monads, monoids, lenses, …

type inference
polymorphism = generics
type classes = overloading
monads, monoids, lenses, . . .
dependent types

not just “types”
effects
information flow
regions
...

calculation

nub: remove all duplicates

```
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

The nub of a list is the list with all duplicates removed.

```
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Rewrite … work “top down”

```
nub [1,2,1]
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
       then nub xs
       else x : nub xs
```

Rewrite … work “top down”

```haskell
nub [1,2,1]
= nub (1:[2,1])
```

The nub of a list is the list with all duplicates removed.

```
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Rewrite … work “top down”

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
```

The nub of a list is the list with all duplicates removed.

```
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
       then nub xs
       else x : nub xs
```

Rewrite . . . work “top down”

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
```

The nub of a list is the list with all duplicates removed.

```
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
        then nub xs
        else x : nub xs
```

Rewrite … work “top down”

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
= 2 : nub [1]
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
       then nub xs
       else x : nub xs
```

Rewrite … work “top down”

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
= 2 : nub [1]
= 2 : nub (1:[])
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
       then nub xs
       else x : nub xs
```

Rewrite … work “top down”

```haskell
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
= 2 : nub [1]
= 2 : nub (1:[])
= 2 : 1 : nub []
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
       then nub xs
       else x : nub xs
```

Rewrite . . . work “top down”

```haskell
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
= 2 : nub [1]
= 2 : nub (1:[])
= 2 : 1 : nub []
= 2 : 1 : []
```

The nub of a list is the list with all duplicates removed.

```haskell
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
       then nub xs
       else x : nub xs
```

Rewrite . . . work “top down”

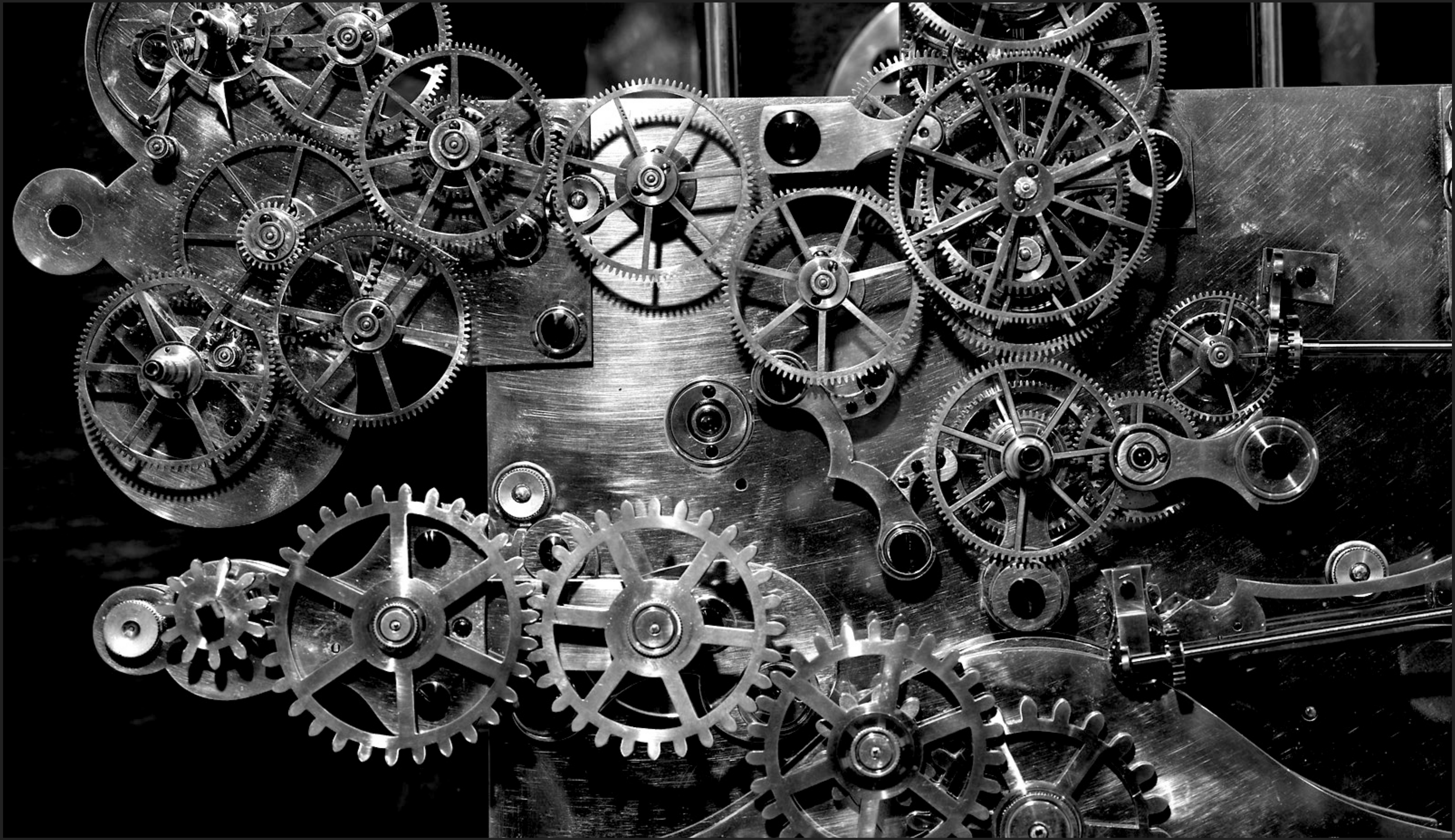
```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
= 2 : nub [1]
= 2 : nub (1:[])
= 2 : 1 : nub []
= 2 : 1 : []
= 2 :[1]
```

The nub of a list is the list with all duplicates removed.

```
nub :: Eq a => [a] -> [a]

nub [] = []

nub(x:xs) =
    if elem x xs
       then nub xs
       else x : nub xs
```

Rewrite . . . work “top down”

```
nub [1,2,1]
= nub (1:[2,1])
= nub [2,1]
= nub (2:[1])
= 2 : nub [1]
= 2 : nub (1:[])
= 2 : 1 : nub []
= 2 : 1 : []
= 2 :[1]
= [2,1]
```



Not just in theory ...

tools

metaprogramming

startups

libraries

experienced
people

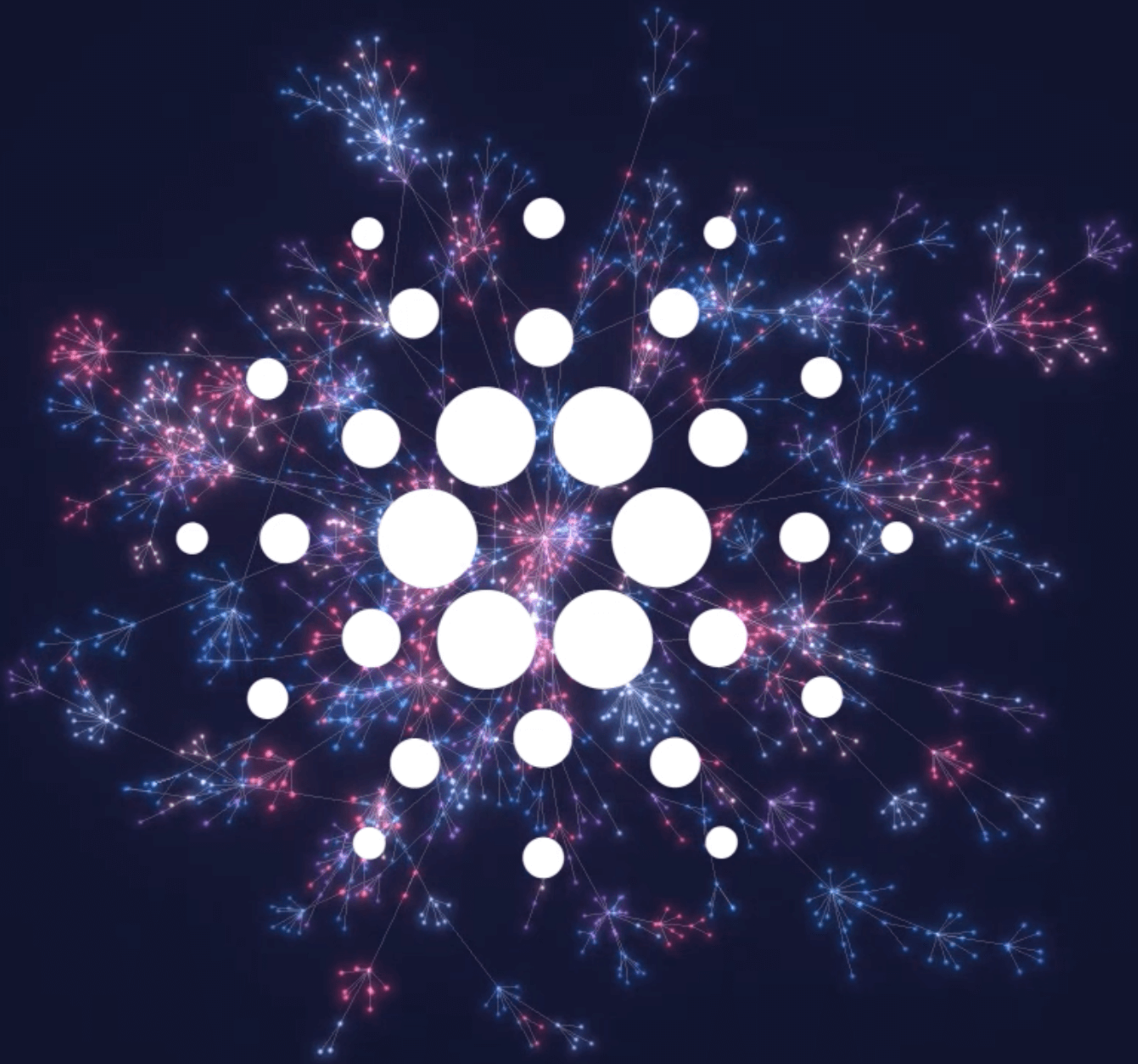
stack/cabal

Haskell

GHC

IDEs

beacon language

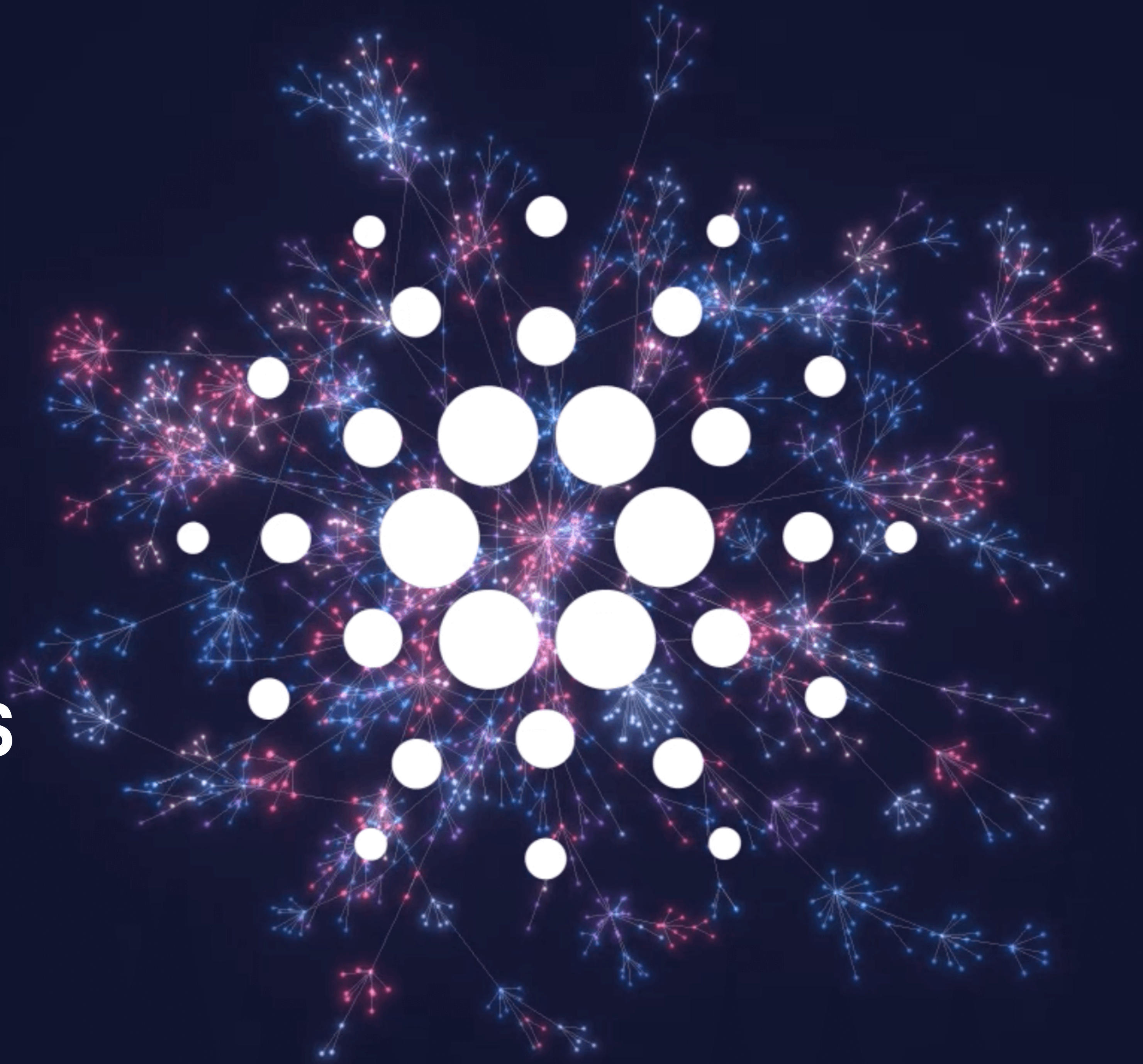


Cardano

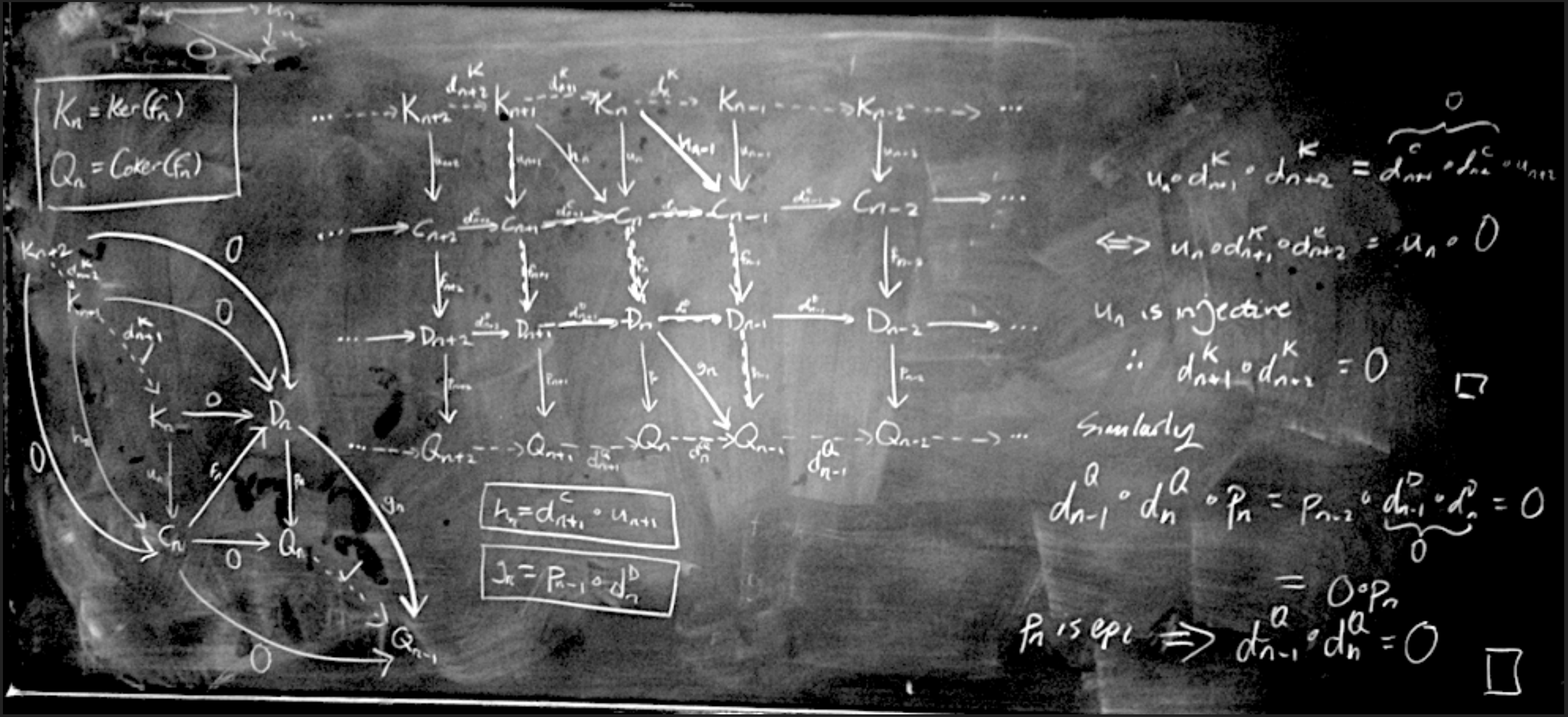
3rd-gen

PoS

Sidechains



This board belongs to Newcastle University PhD student Tom Fisher, who is doing research in homological algebra.
Thanks to Christian Perfect for the photo.
whatsonmyblackboard.wordpress.com



$K_n = \mathrm{Ker}(f_n)$

$Q_n = \mathrm{Coker}(f_n)$

$h_n = d^C_{n+1} \circ u_{n+1}$

$g_n = p_{n-1} \circ d^D_n$

$u_n \circ d^K_{n+1} \circ d^K_{n+2} = \underbrace{d^C_{n+1} \circ d^C_{n+2}}_{0} \circ u_{n+2}$

$\Leftrightarrow u_n \circ d^K_{n+1} \circ d^K_{n+2} = u_n \circ 0$

u_n is injective

$\therefore d^K_{n+1} \circ d^K_{n+2} = 0$ $\square$

Similarly

$d^Q_{n-1} \circ d^Q_n \circ p_n = p_{n-2} \circ \underbrace{d^D_{n-1} \circ d^D_n}_{0} = 0$

$= 0 \circ p_n$

p_n is epi $\Rightarrow d^Q_{n-1} \circ d^Q_n = 0$ $\square$

Secure foundations

Secure foundations

Immutable data

Explicit effects

Functions as data

Expressive types

Secure foundations

Immutable data

Explicit effects

Functions as data

Expressive types

Develop from a formal spec

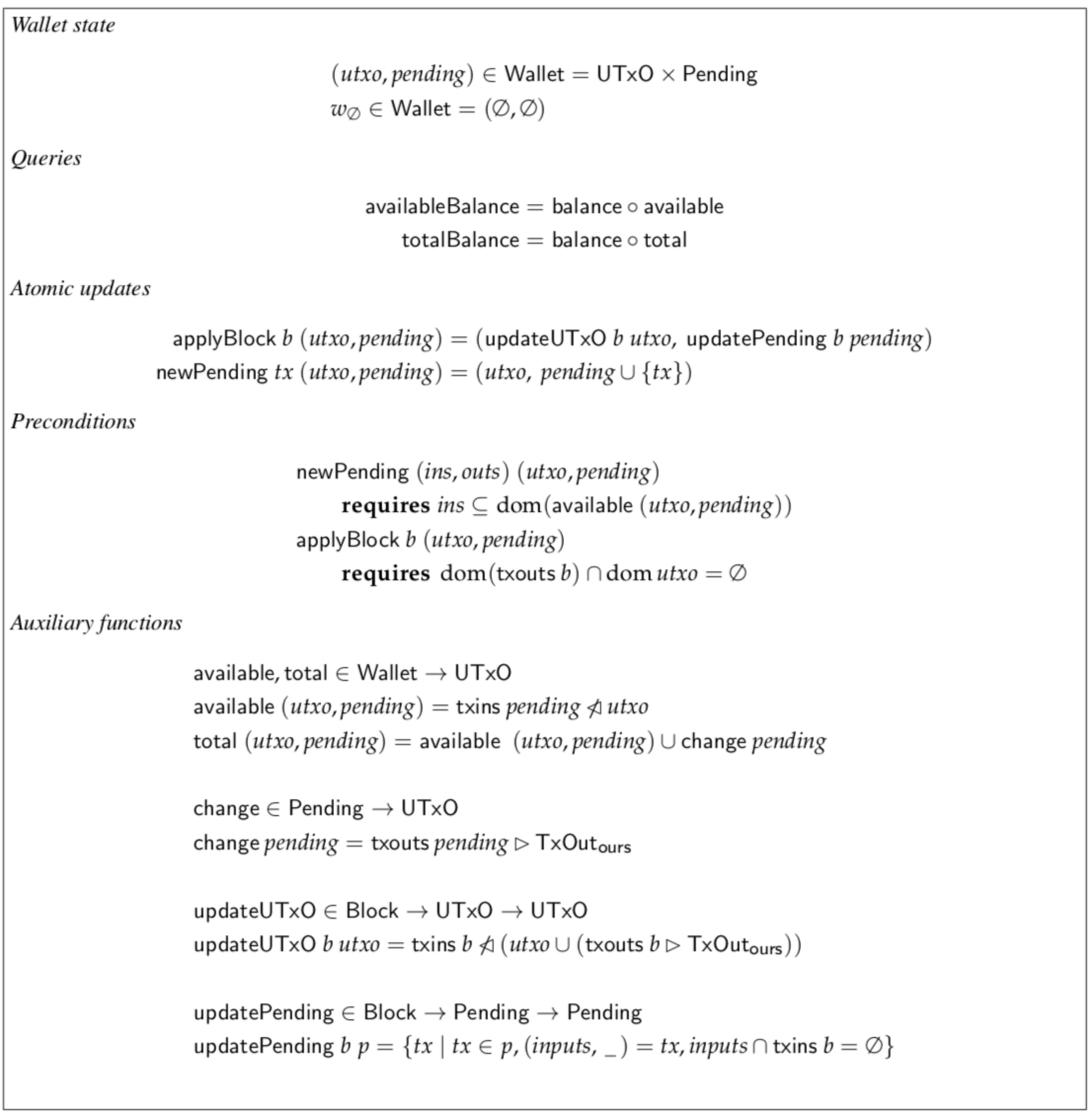
🧪 Research Driven

↓

$\forall \exists \vdash$ Formal Methods

↓

λ^{τ} Functional Programming



Wallet state

$$\begin{aligned} (utxo, pending) &\in \mathsf{Wallet} = \mathsf{UTxO} \times \mathsf{Pending} \\ w_{\emptyset} &\in \mathsf{Wallet} = (\emptyset, \emptyset) \end{aligned}$$

Queries

$$\begin{aligned} \mathsf{availableBalance} &= \mathsf{balance} \circ \mathsf{available} \\ \mathsf{totalBalance} &= \mathsf{balance} \circ \mathsf{total} \end{aligned}$$

Atomic updates

$$\begin{aligned} \mathsf{applyBlock}\ b\ (utxo, pending) &= (\mathsf{updateUTxO}\ b\ utxo,\ \mathsf{updatePending}\ b\ pending) \\ \mathsf{newPending}\ tx\ (utxo, pending) &= (utxo,\ pending \cup \{tx\}) \end{aligned}$$

Preconditions

$$\begin{aligned} &\mathsf{newPending}\ (ins, outs)\ (utxo, pending) \\ &\qquad \mathbf{requires}\ ins \subseteq \mathrm{dom}(\mathsf{available}\ (utxo, pending)) \\ &\mathsf{applyBlock}\ b\ (utxo, pending) \\ &\qquad \mathbf{requires}\ \ \mathrm{dom}(\mathsf{txouts}\ b) \cap \mathrm{dom}\ utxo = \emptyset \end{aligned}$$

Auxiliary functions

$$\begin{aligned} &\mathsf{available}, \mathsf{total} \in \mathsf{Wallet} \to \mathsf{UTxO} \\ &\mathsf{available}\ (utxo, pending) = \mathsf{txins}\ pending \ntriangleleft utxo \\ &\mathsf{total}\ (utxo, pending) = \mathsf{available}\ \ (utxo, pending) \cup \mathsf{change}\ pending \\ \\ &\mathsf{change} \in \mathsf{Pending} \to \mathsf{UTxO} \\ &\mathsf{change}\ pending = \mathsf{txouts}\ pending \rhd \mathsf{TxOut}_{\mathsf{ours}} \\ \\ &\mathsf{updateUTxO} \in \mathsf{Block} \to \mathsf{UTxO} \to \mathsf{UTxO} \\ &\mathsf{updateUTxO}\ b\ utxo = \mathsf{txins}\ b \ntriangleleft (utxo \cup (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}})) \\ \\ &\mathsf{updatePending} \in \mathsf{Block} \to \mathsf{Pending} \to \mathsf{Pending} \\ &\mathsf{updatePending}\ b\ p = \{tx \mid tx \in p, (inputs, \_) = tx, inputs \cap \mathsf{txins}\ b = \emptyset\} \end{aligned}$$

Wallet state

$$\begin{aligned}(\mathit{utxo}, \mathit{pending}) &\in \mathsf{Wallet} = \mathsf{UTxO} \times \mathsf{Pending} \\ w_{\emptyset} &\in \mathsf{Wallet} = (\emptyset, \emptyset)\end{aligned}$$

Queries

$$\begin{aligned}\mathsf{availableBalance} &= \mathsf{balance} \circ \mathsf{available} \\ \mathsf{totalBalance} &= \mathsf{balance} \circ \mathsf{total}\end{aligned}$$

Atomic updates

$$\begin{aligned}\mathsf{applyBlock}\ b\ (\mathit{utxo}, \mathit{pending}) &= (\mathsf{updateUTxO}\ b\ \mathit{utxo},\ \mathsf{updatePending}\ b\ \mathit{pending}) \\ \mathsf{newPending}\ \mathit{tx}\ (\mathit{utxo}, \mathit{pending}) &= (\mathit{utxo},\ \mathit{pending} \cup \{\mathit{tx}\})\end{aligned}$$

Preconditions

$$\begin{aligned}&\mathsf{newPending}\ (\mathit{ins}, \mathit{outs})\ (\mathit{utxo}, \mathit{pending}) \\ &\qquad \textbf{requires}\ \mathit{ins} \subseteq \mathrm{dom}(\mathsf{available}\ (\mathit{utxo}, \mathit{pending})) \\ &\mathsf{applyBlock}\ b\ (\mathit{utxo}, \mathit{pending}) \\ &\qquad \textbf{requires}\ \mathrm{dom}(\mathsf{txouts}\ b) \cap \mathrm{dom}\ \mathit{utxo} = \emptyset\end{aligned}$$

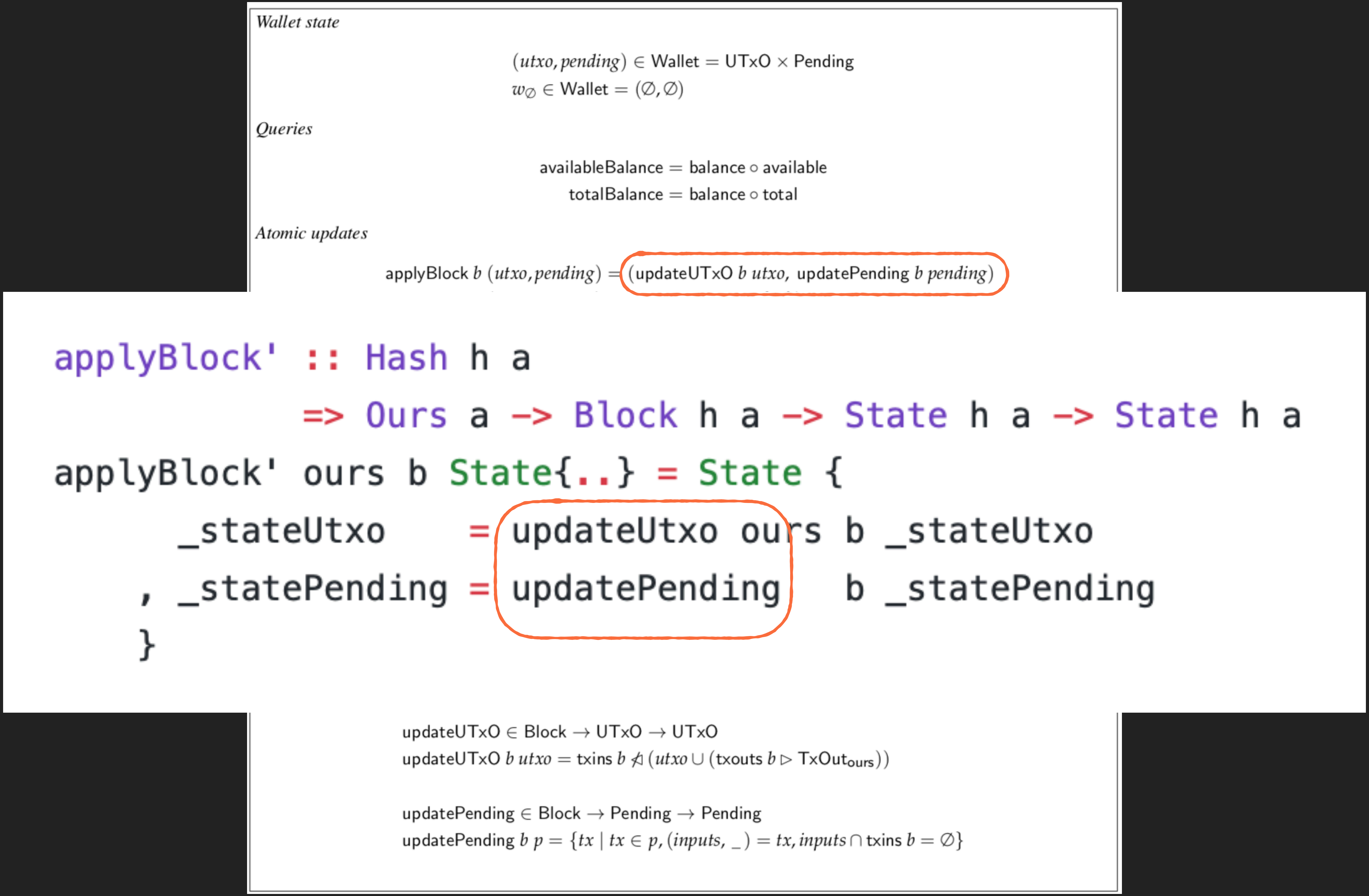
Auxiliary functions

$$\begin{aligned}&\mathsf{available}, \mathsf{total} \in \mathsf{Wallet} \to \mathsf{UTxO} \\ &\mathsf{available}\ (\mathit{utxo}, \mathit{pending}) = \mathsf{txins}\ \mathit{pending} \ntriangleleft \mathit{utxo} \\ &\mathsf{total}\ (\mathit{utxo}, \mathit{pending}) = \mathsf{available}\ \ (\mathit{utxo}, \mathit{pending}) \cup \mathsf{change}\ \mathit{pending}\end{aligned}$$

$$\begin{aligned}&\mathsf{change} \in \mathsf{Pending} \to \mathsf{UTxO} \\ &\mathsf{change}\ \mathit{pending} = \mathsf{txouts}\ \mathit{pending} \triangleright \mathsf{TxOut}_{\mathsf{ours}}\end{aligned}$$

$$\begin{aligned}&\mathsf{updateUTxO} \in \mathsf{Block} \to \mathsf{UTxO} \to \mathsf{UTxO} \\ &\mathsf{updateUTxO}\ b\ \mathit{utxo} = \mathsf{txins}\ b \ntriangleleft (\mathit{utxo} \cup (\mathsf{txouts}\ b \triangleright \mathsf{TxOut}_{\mathsf{ours}}))\end{aligned}$$

$$\begin{aligned}&\mathsf{updatePending} \in \mathsf{Block} \to \mathsf{Pending} \to \mathsf{Pending} \\ &\mathsf{updatePending}\ b\ p = \{\mathit{tx} \mid \mathit{tx} \in p, (\mathit{inputs}, \_) = \mathit{tx}, \mathit{inputs} \cap \mathsf{txins}\ b = \emptyset\}\end{aligned}$$



Wallet state

$$(utxo, pending) \in \mathsf{Wallet} = \mathsf{UTxO} \times \mathsf{Pending}$$
$$w_{\emptyset} \in \mathsf{Wallet} = (\emptyset, \emptyset)$$

Queries

$$\mathsf{availableBalance} = \mathsf{balance} \circ \mathsf{available}$$
$$\mathsf{totalBalance} = \mathsf{balance} \circ \mathsf{total}$$

Atomic updates

$$\mathsf{applyBlock}\ b\ (utxo, pending) = (\mathsf{updateUTxO}\ b\ utxo,\ \mathsf{updatePending}\ b\ pending)$$

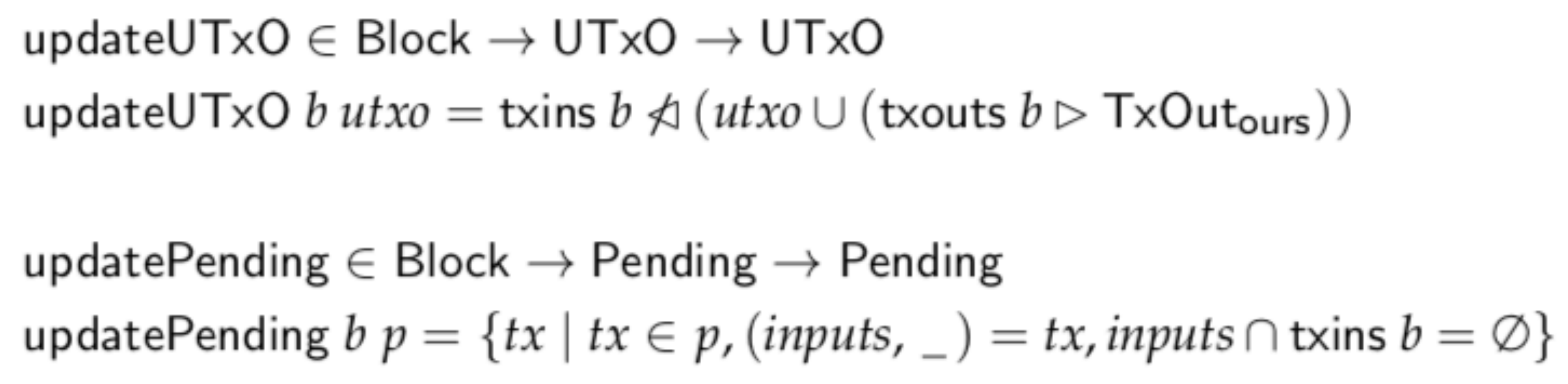
```
applyBlock' :: Hash h a
            => Ours a -> Block h a -> State h a -> State h a
applyBlock' ours b State{..} = State {
      _stateUtxo    = updateUtxo ours b _stateUtxo
    , _statePending = updatePending   b _statePending
    }
```

$$\mathsf{updateUTxO} \in \mathsf{Block} \rightarrow \mathsf{UTxO} \rightarrow \mathsf{UTxO}$$
$$\mathsf{updateUTxO}\ b\ utxo = \mathsf{txins}\ b \not\lhd (utxo \cup (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}}))$$

$$\mathsf{updatePending} \in \mathsf{Block} \rightarrow \mathsf{Pending} \rightarrow \mathsf{Pending}$$
$$\mathsf{updatePending}\ b\ p = \{tx \mid tx \in p, (inputs, \_) = tx, inputs \cap \mathsf{txins}\ b = \emptyset\}$$

$$\mathsf{updateUTxO} \in \mathsf{Block} \rightarrow \mathsf{UTxO} \rightarrow \mathsf{UTxO}$$
$$\mathsf{updateUTxO}\ b\ utxo = \mathsf{txins}\ b \not\lhd (utxo \cup (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}}))$$

$$\mathsf{updatePending} \in \mathsf{Block} \rightarrow \mathsf{Pending} \rightarrow \mathsf{Pending}$$
$$\mathsf{updatePending}\ b\ p = \{tx \mid tx \in p, (inputs,\ \_) = tx, inputs \cap \mathsf{txins}\ b = \varnothing\}$$

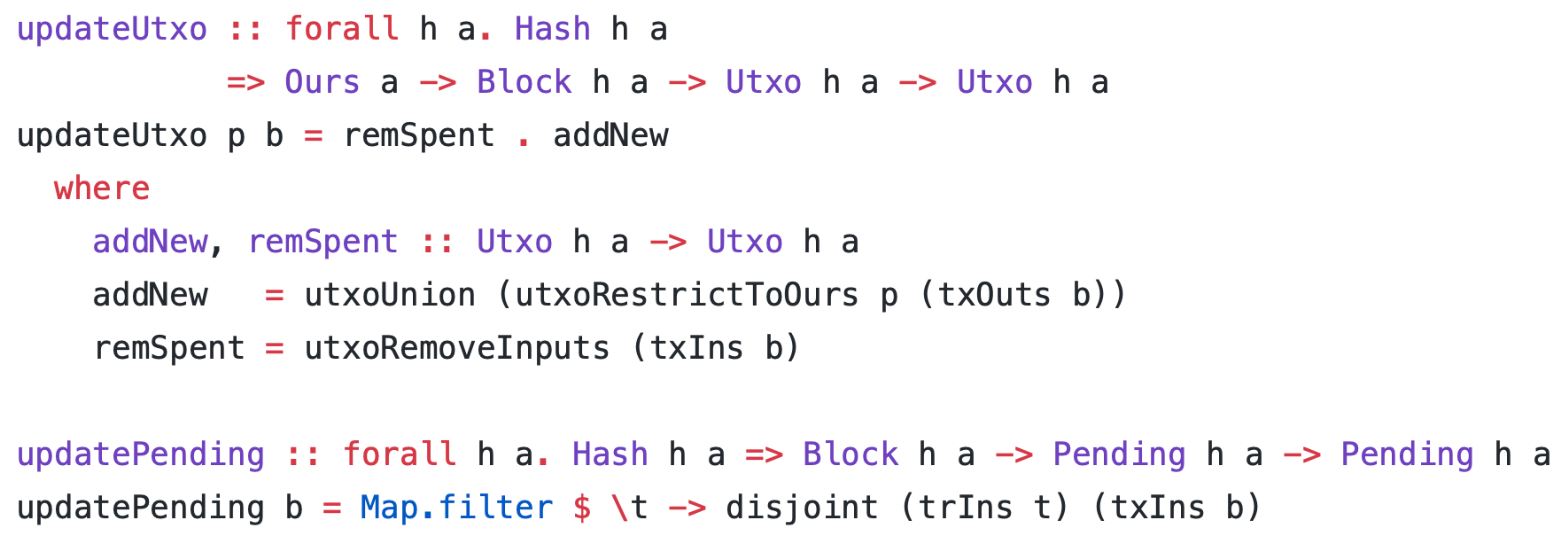


$$\mathsf{updateUTxO} \in \mathsf{Block} \rightarrow \mathsf{UTxO} \rightarrow \mathsf{UTxO}$$

$$\mathsf{updateUTxO}\ b\ utxo = \mathsf{txins}\ b \ntriangleleft (utxo \cup (\mathsf{txouts}\ b \rhd \mathsf{TxOut}_{\mathsf{ours}}))$$

$$\mathsf{updatePending} \in \mathsf{Block} \rightarrow \mathsf{Pending} \rightarrow \mathsf{Pending}$$

$$\mathsf{updatePending}\ b\ p = \{tx \mid tx \in p, (inputs, \_) = tx, inputs \cap \mathsf{txins}\ b = \emptyset\}$$



```
updateUtxo :: forall h a. Hash h a
           => Ours a -> Block h a -> Utxo h a -> Utxo h a
updateUtxo p b = remSpent . addNew
  where
    addNew, remSpent :: Utxo h a -> Utxo h a
    addNew   = utxoUnion (utxoRestrictToOurs p (txOuts b))
    remSpent = utxoRemoveInputs (txIns b)

updatePending :: forall h a. Hash h a => Block h a -> Pending h a -> Pending h a
updatePending b = Map.filter $ \t -> disjoint (trIns t) (txIns b)
```

Secure foundations

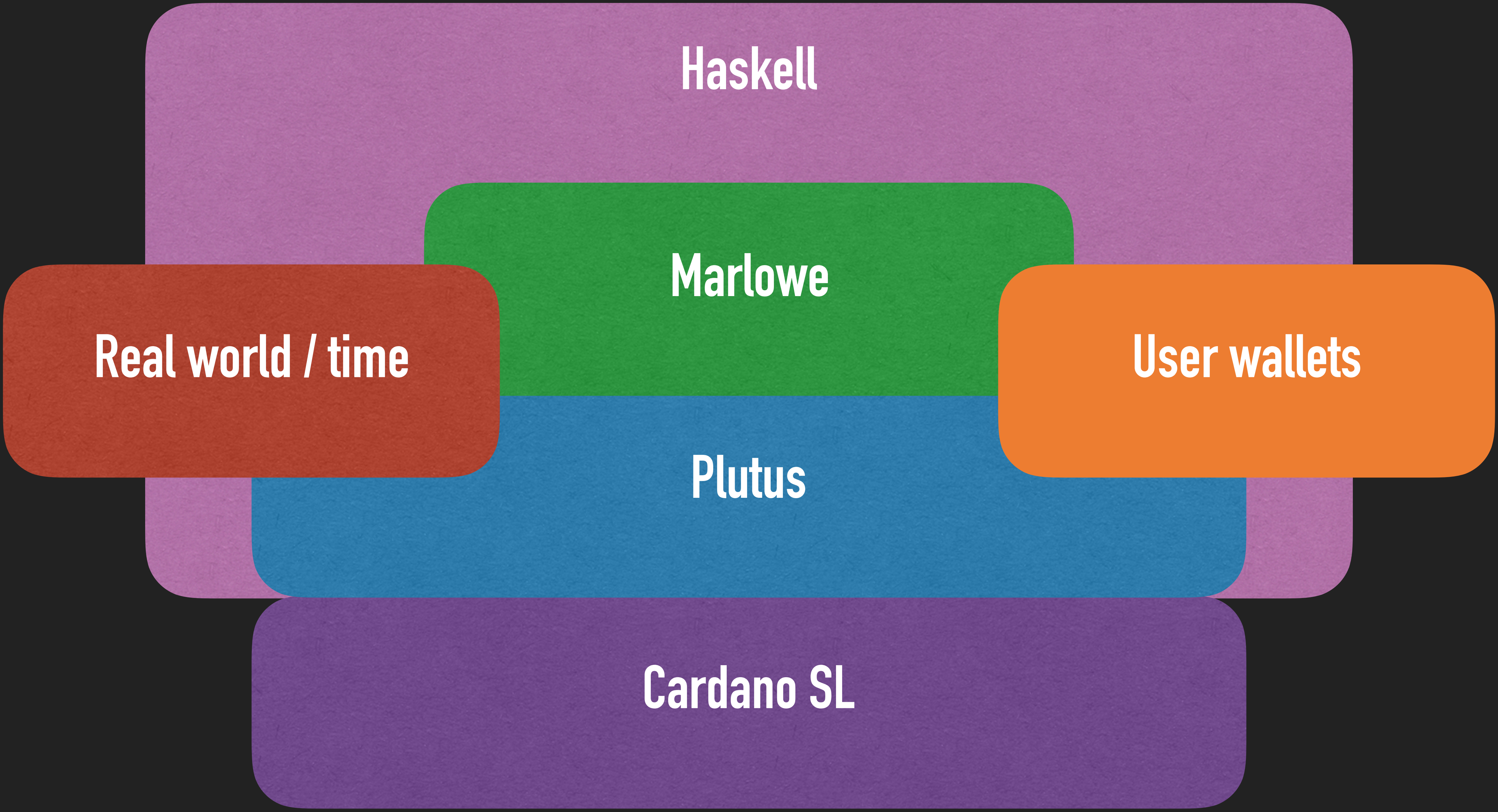
Develop from a formal spec

Property-based random tests

Model in a proof assistant

Minimal “napkin” machine

Scripting Cardano



Haskell

Marlowe

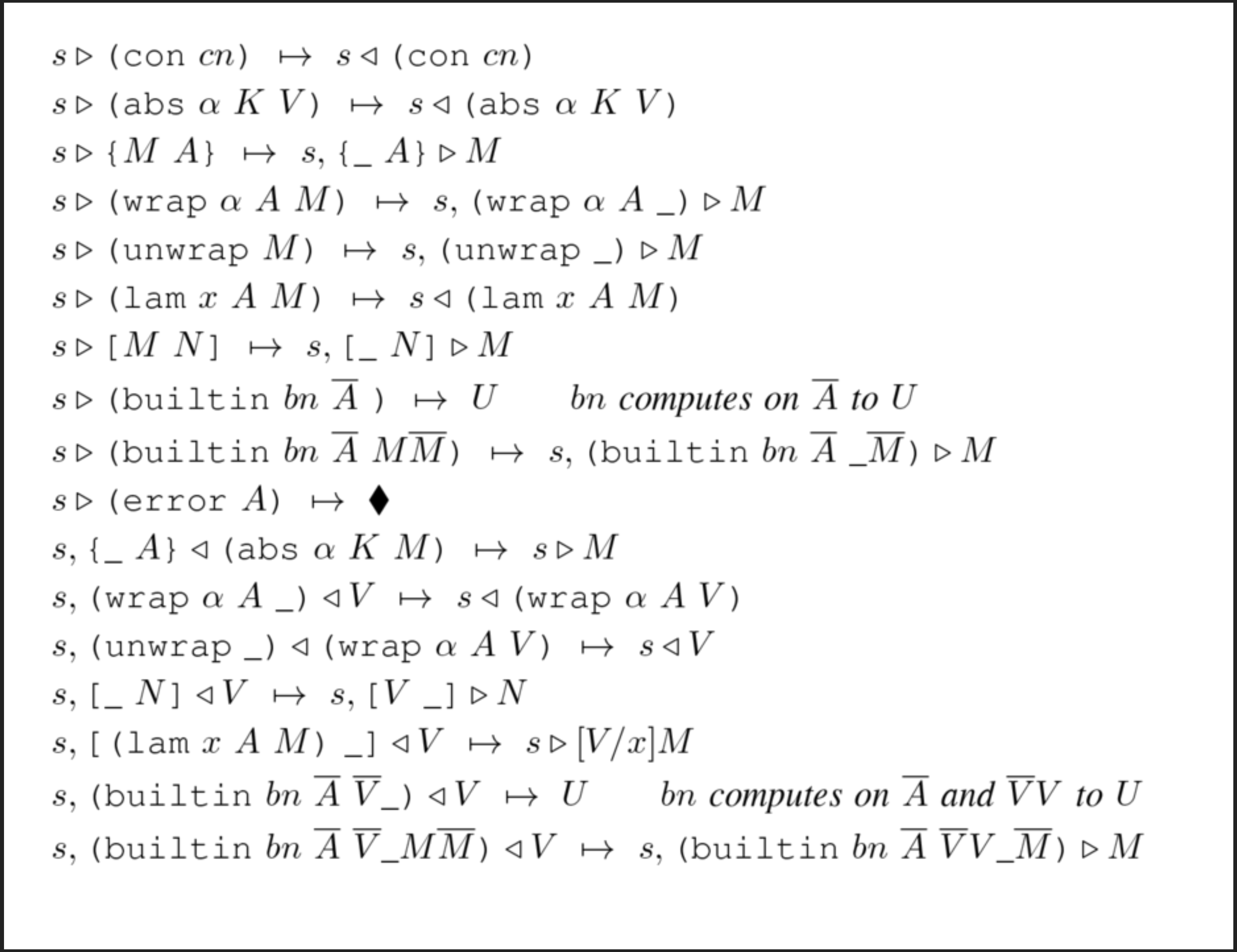
Real world / time

User wallets

Plutus

Cardano SL

Minimal “napkin” machine



$$s \rhd \mathtt{(con}\ cn\mathtt{)} \;\mapsto\; s \lhd \mathtt{(con}\ cn\mathtt{)}$$

$$s \rhd \mathtt{(abs}\ \alpha\ K\ V\mathtt{)} \;\mapsto\; s \lhd \mathtt{(abs}\ \alpha\ K\ V\mathtt{)}$$

$$s \rhd \{M\ A\} \;\mapsto\; s, \{\_\ A\} \rhd M$$

$$s \rhd \mathtt{(wrap}\ \alpha\ A\ M\mathtt{)} \;\mapsto\; s, \mathtt{(wrap}\ \alpha\ A\ \_\mathtt{)} \rhd M$$

$$s \rhd \mathtt{(unwrap}\ M\mathtt{)} \;\mapsto\; s, \mathtt{(unwrap}\ \_\mathtt{)} \rhd M$$

$$s \rhd \mathtt{(lam}\ x\ A\ M\mathtt{)} \;\mapsto\; s \lhd \mathtt{(lam}\ x\ A\ M\mathtt{)}$$

$$s \rhd \mathtt{[}M\ N\mathtt{]} \;\mapsto\; s, \mathtt{[}\_\ N\mathtt{]} \rhd M$$

$$s \rhd \mathtt{(builtin}\ bn\ \overline{A}\ \mathtt{)} \;\mapsto\; U \qquad \textit{bn computes on } \overline{A} \textit{ to } U$$

$$s \rhd \mathtt{(builtin}\ bn\ \overline{A}\ M\overline{M}\mathtt{)} \;\mapsto\; s, \mathtt{(builtin}\ bn\ \overline{A}\ \_\overline{M}\mathtt{)} \rhd M$$

$$s \rhd \mathtt{(error}\ A\mathtt{)} \;\mapsto\; \blacklozenge$$

$$s, \{\_\ A\} \lhd \mathtt{(abs}\ \alpha\ K\ M\mathtt{)} \;\mapsto\; s \rhd M$$

$$s, \mathtt{(wrap}\ \alpha\ A\ \_\mathtt{)} \lhd V \;\mapsto\; s \lhd \mathtt{(wrap}\ \alpha\ A\ V\mathtt{)}$$

$$s, \mathtt{(unwrap}\ \_\mathtt{)} \lhd \mathtt{(wrap}\ \alpha\ A\ V\mathtt{)} \;\mapsto\; s \lhd V$$

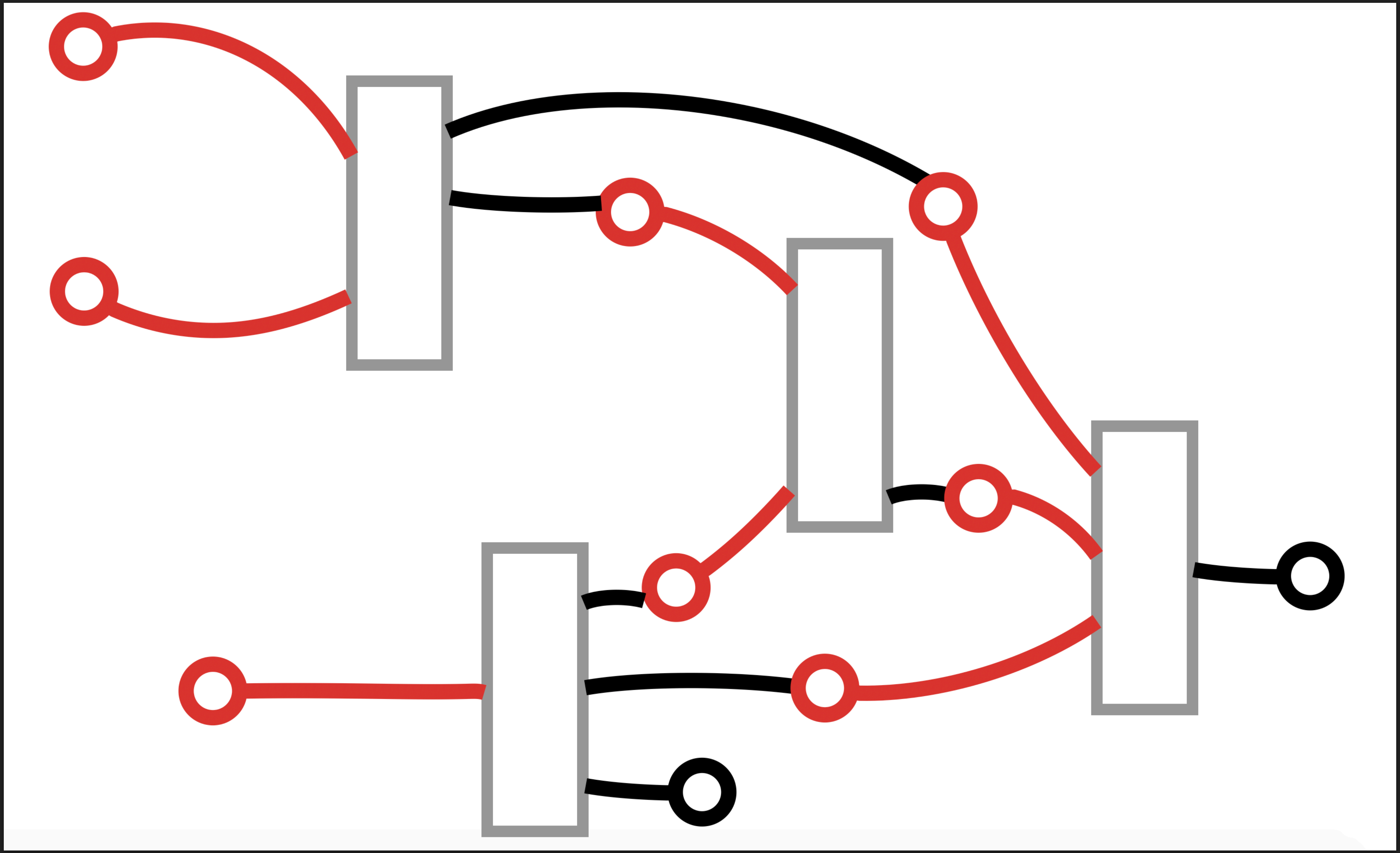
$$s, \mathtt{[}\_\ N\mathtt{]} \lhd V \;\mapsto\; s, \mathtt{[}V\ \_\mathtt{]} \rhd N$$

$$s, \mathtt{[}\ \mathtt{(lam}\ x\ A\ M\mathtt{)}\ \_\mathtt{]} \lhd V \;\mapsto\; s \rhd [V/x]M$$

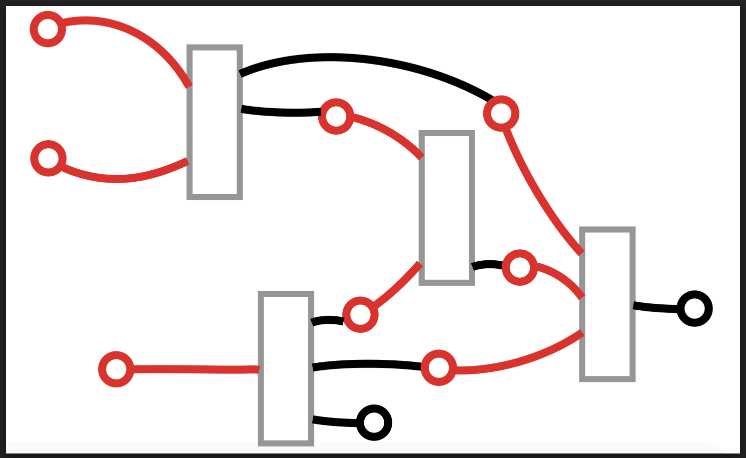
$$s, \mathtt{(builtin}\ bn\ \overline{A}\ \overline{V}\_\mathtt{)} \lhd V \;\mapsto\; U \qquad \textit{bn computes on } \overline{A} \textit{ and } \overline{V}V \textit{ to } U$$

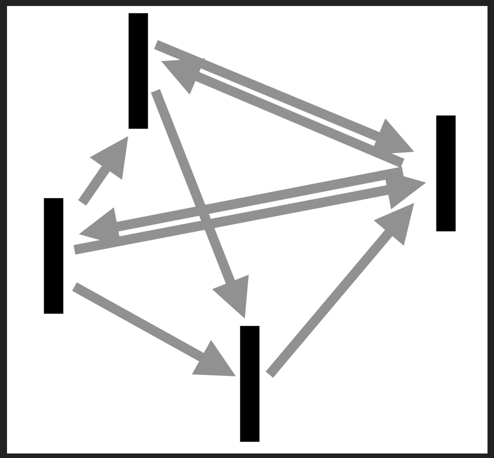
$$s, \mathtt{(builtin}\ bn\ \overline{A}\ \overline{V}\_M\overline{M}\mathtt{)} \lhd V \;\mapsto\; s, \mathtt{(builtin}\ bn\ \overline{A}\ \overline{V}V\_\overline{M}\mathtt{)} \rhd M$$

Functional transaction model



UTxO vs Accounts





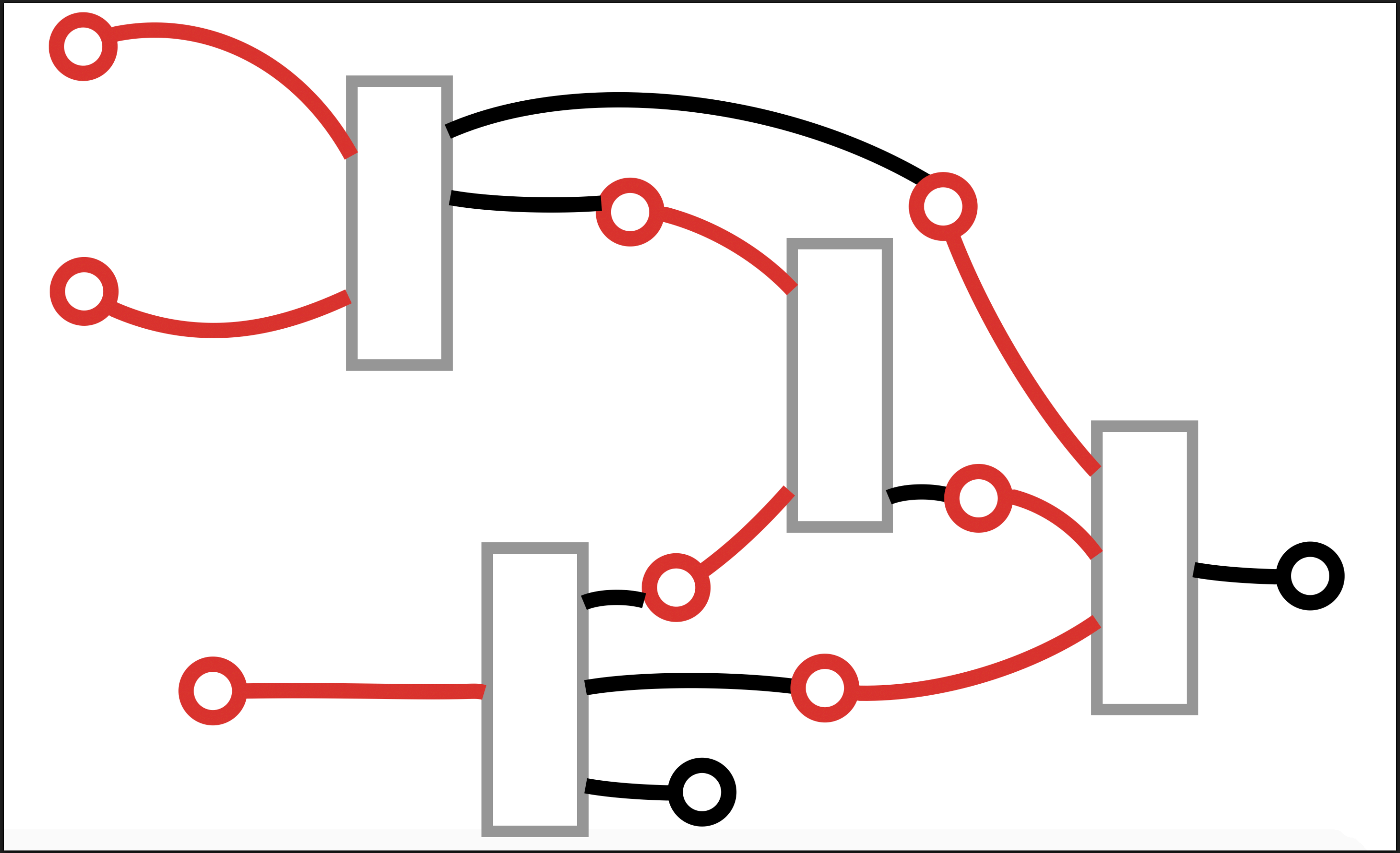
Functional, dataflow

Compositional

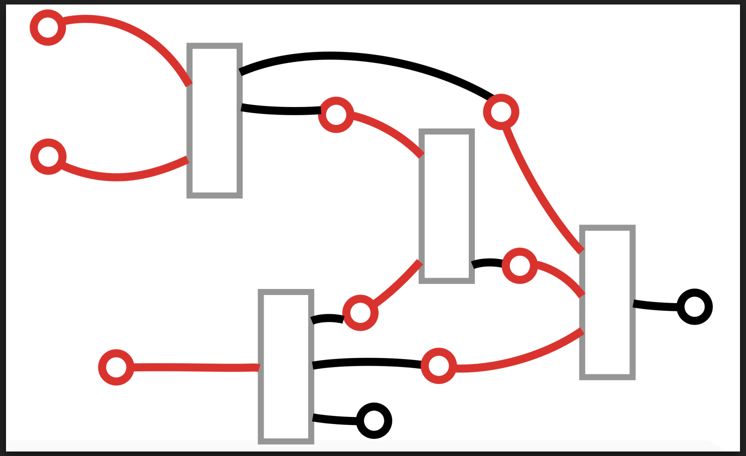
Imperative, entangled

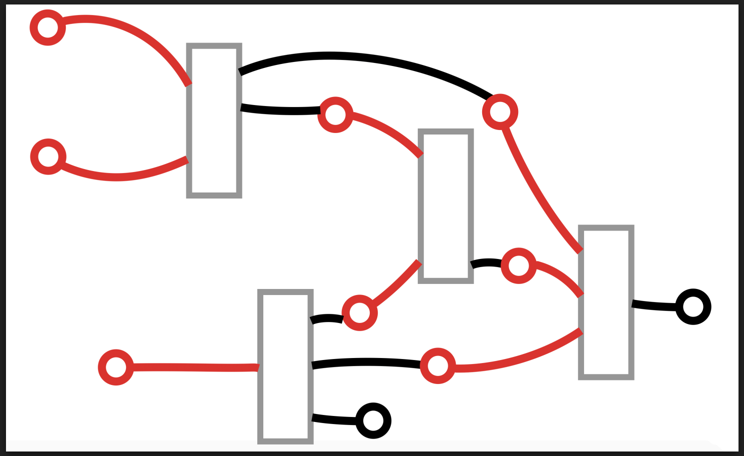
Shared state

Extended UTxO



UTxO vs Extended UTxO

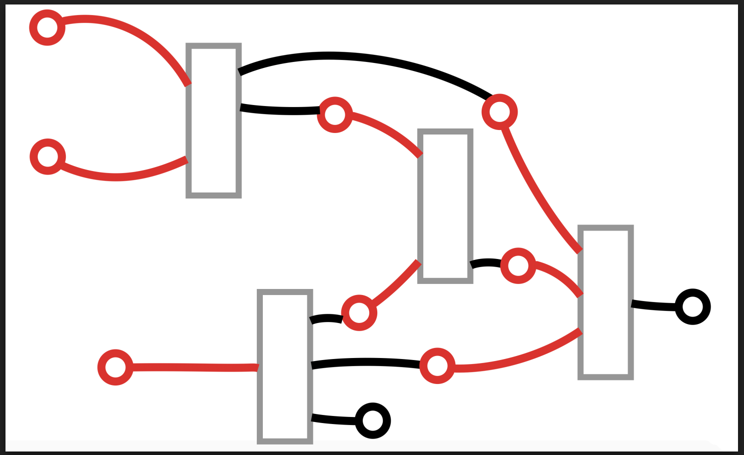




Validator(Redeemer) = True

Validator(Redeemer,
Data,
State) = True

UTxO vs Extended UTxO



Validator(Redeemer,
Data,
State) = True

data flows with value

scripts have an identity

a transaction can control how its UTxOs are spent

Haskell “all the way down”

Unify on- & off-chain code

Meta-programming

```
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))
             value
             DataScript (Ledger.lifted ownPK)

  register (refundTrigger campaign)
           (refundHandler (Ledger.hashTx tx)
campaign)
```

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

contributorOnly outs &&
                                   $$(txSignedBy)
tx contrib
                        Collect -> h > deadline
```

Domain-specific languages

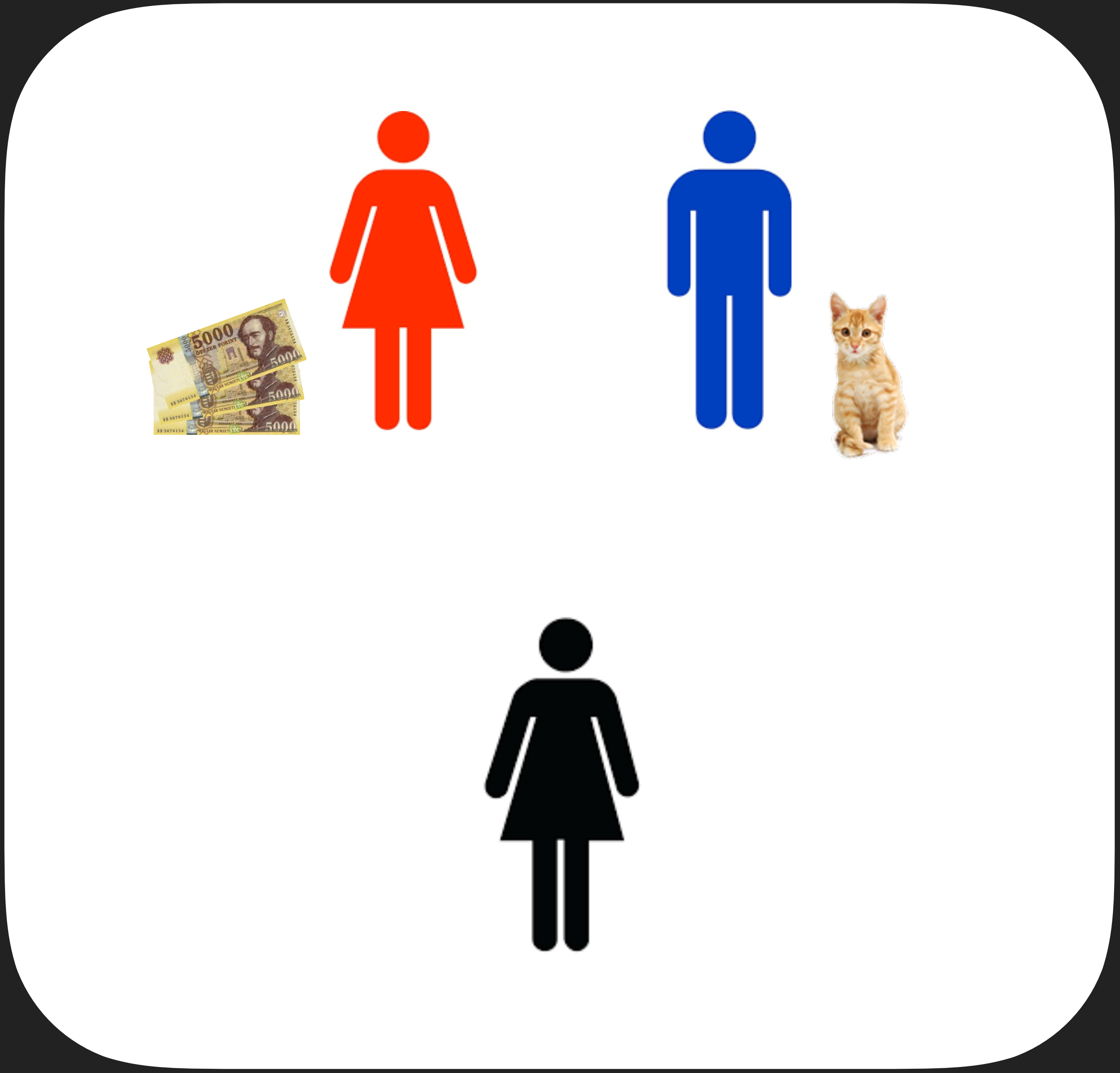
Domain-specific languages

Domain-specific languages . . .

. . . as data types, monads,

. . . and embedded in Haskell

Marlowe



Marlowe

```
(When (Or (majority_chose refund)
          (majority_chose pay))

      (Choice (majority_chose pay)
              (Pay alice bob AvailableMoney)
              redeem_original)
```

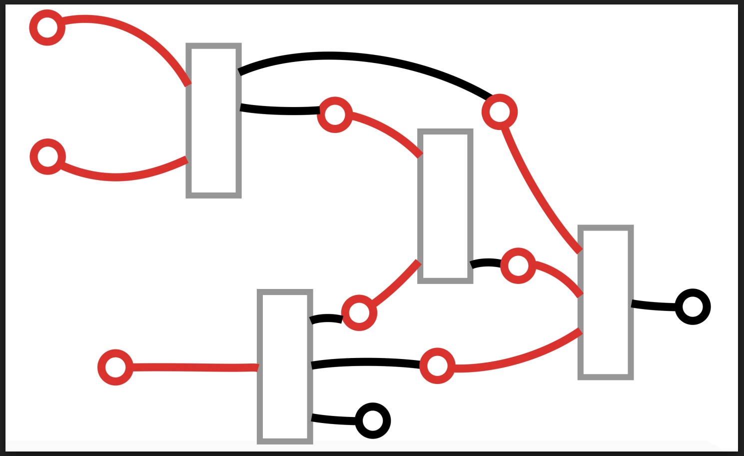
Marlowe

```
(When (Or (majority_chose refund)
          (majority_chose pay))
      90
      (Choice (majority_chose pay)
              (Pay alice bob AvailableMoney)
              redeem_original)
      redeem_original)
```

Marlowe

```
(CommitCash id1 alice 15000 10 100
   (When (Or (majority_chose refund)
             (majority_chose pay))
         90
         (Choice (majority_chose pay)
                 (Pay alice bob AvailableMoney)
                 redeem_original)
         redeem_original)
 Null)
```

Implementing Marlowe



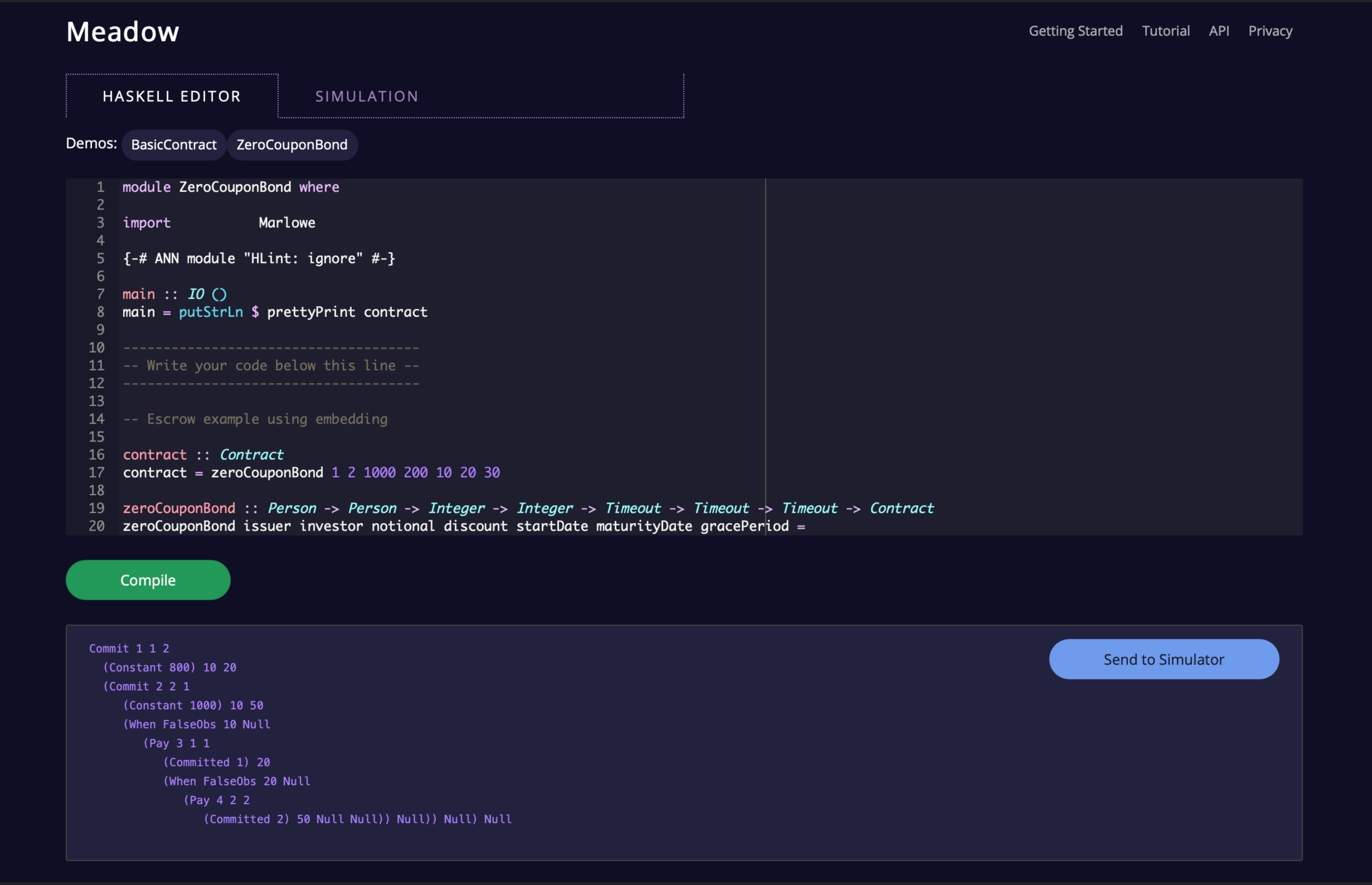
the Marlowe interpreter is a single Plutus script

use the Data script for residual contract

we could also compile …

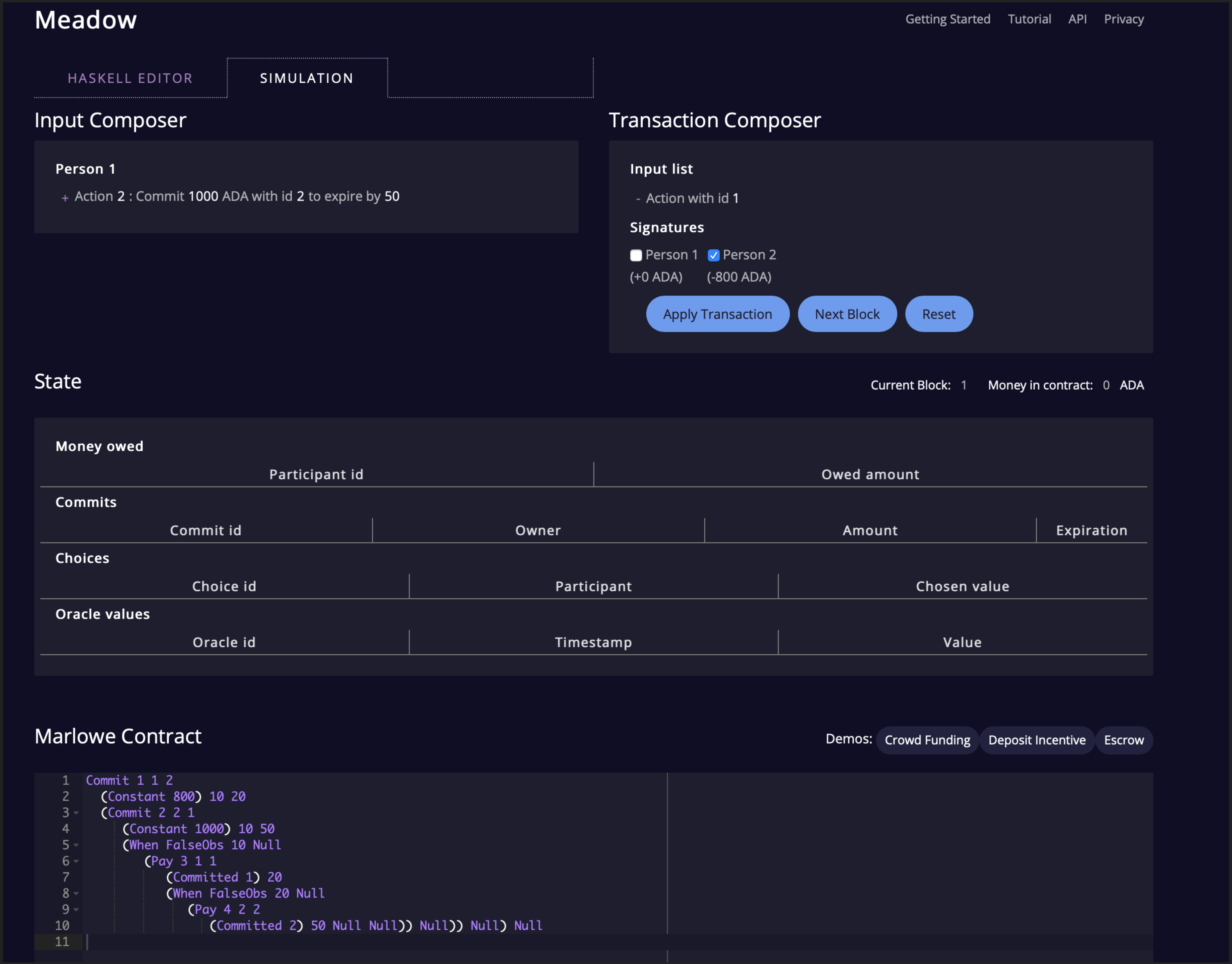
question of fees, code reuse, libraries, …

Validator(Redeemer,
Data,
State) = True



```haskell
module ZeroCouponBond where

import           Marlowe

{-# ANN module "HLint: ignore" #-}

main :: IO ()
main = putStrLn $ prettyPrint contract

-------------------------------------
-- Write your code below this line --
-------------------------------------

-- Escrow example using embedding

contract :: Contract
contract = zeroCouponBond 1 2 1000 200 10 20 30

zeroCouponBond :: Person -> Person -> Integer -> Integer -> Timeout -> Timeout -> Timeout -> Contract
zeroCouponBond issuer investor notional discount startDate maturityDate gracePeriod =
```

```
Commit 1 1 2
  (Constant 800) 10 20
  (Commit 2 2 1
     (Constant 1000) 10 50
     (When FalseObs 10 Null
        (Pay 3 1 1
           (Committed 1) 20
           (When FalseObs 20 Null
              (Pay 4 2 2
                 (Committed 2) 50 Null Null)) Null)) Null) Null
```



Input Composer

Person 1

+ Action 2 : Commit 1000 ADA with id 2 to expire by 50

Transaction Composer

Input list

- Action with id 1

Signatures

☐ Person 1 (+0 ADA) ☑ Person 2 (-800 ADA)

State

Current Block: 1 Money in contract: 0 ADA

Money owed

Participant id	Owed amount

Commits

Commit id	Owner	Amount	Expiration

Choices

Choice id	Participant	Chosen value

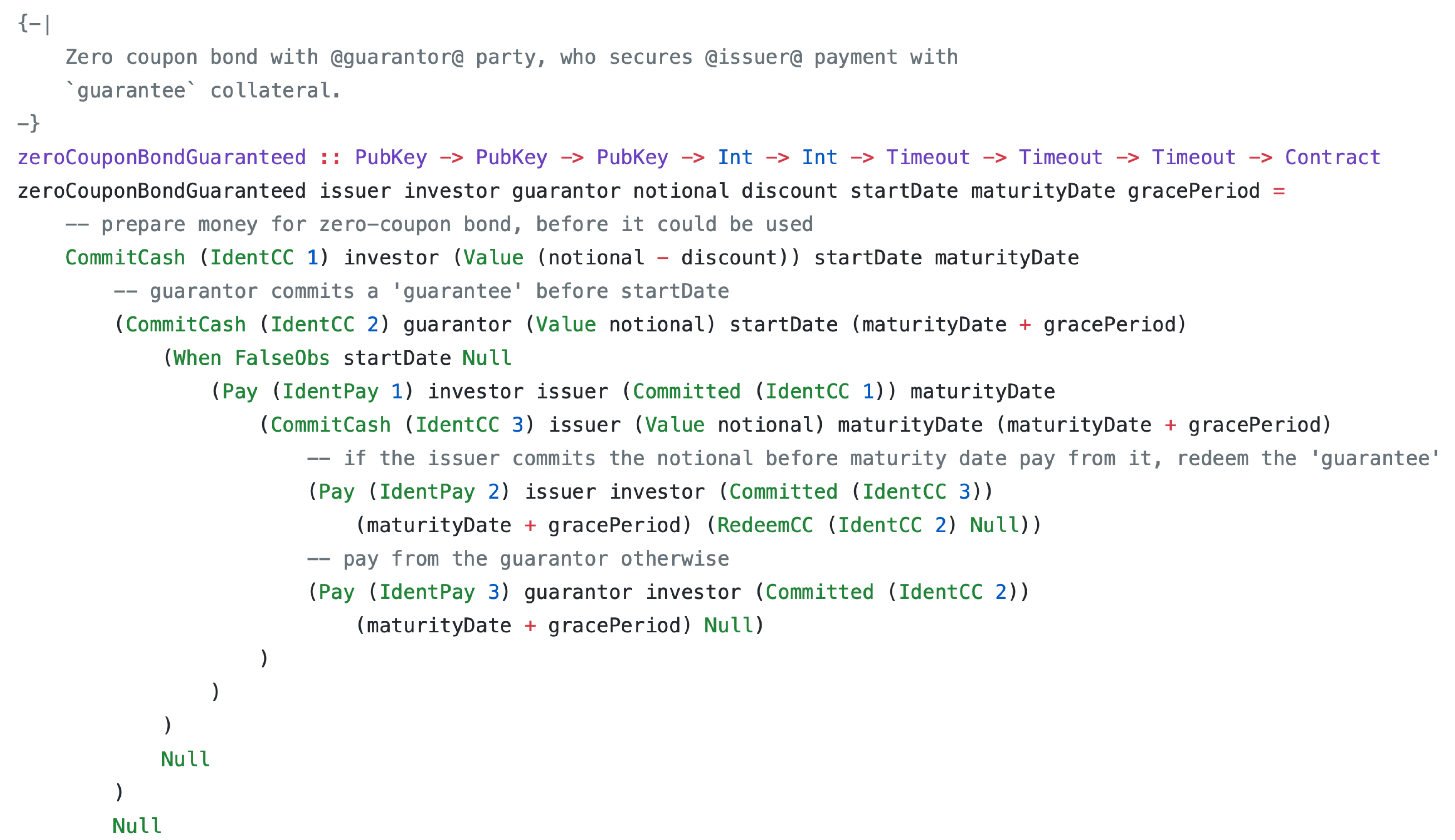
Oracle values

Oracle id	Timestamp	Value

Marlowe Contract

```
Commit 1 1 2
  (Constant 800) 10 20
  (Commit 2 2 1
    (Constant 1000) 10 50
    (When FalseObs 10 Null
      (Pay 3 1 1
        (Committed 1) 20
        (When FalseObs 20 Null
          (Pay 4 2 2
            (Committed 2) 50 Null Null)) Null)) Null) Null
```

Marlowe & ACTUS



```
{-|
    Zero coupon bond with @guarantor@ party, who secures @issuer@ payment with
    `guarantee` collateral.
-}
zeroCouponBondGuaranteed :: PubKey -> PubKey -> PubKey -> Int -> Int -> Timeout -> Timeout -> Timeout -> Contract
zeroCouponBondGuaranteed issuer investor guarantor notional discount startDate maturityDate gracePeriod =
    -- prepare money for zero-coupon bond, before it could be used
    CommitCash (IdentCC 1) investor (Value (notional - discount)) startDate maturityDate
        -- guarantor commits a 'guarantee' before startDate
        (CommitCash (IdentCC 2) guarantor (Value notional) startDate (maturityDate + gracePeriod)
            (When FalseObs startDate Null
                (Pay (IdentPay 1) investor issuer (Committed (IdentCC 1)) maturityDate
                    (CommitCash (IdentCC 3) issuer (Value notional) maturityDate (maturityDate + gracePeriod)
                        -- if the issuer commits the notional before maturity date pay from it, redeem the 'guarantee'
                        (Pay (IdentPay 2) issuer investor (Committed (IdentCC 3))
                            (maturityDate + gracePeriod) (RedeemCC (IdentCC 2) Null))
                        -- pay from the guarantor otherwise
                        (Pay (IdentPay 3) guarantor investor (Committed (IdentCC 2))
                            (maturityDate + gracePeriod) Null)
                    )
                )
            )
            Null
        )
        Null
```

immutable + explicit effects

immutable + explicit effects

immutable + explicit effects

strongly typed + formal specs

immutable + explicit effects

strongly typed + formal specs

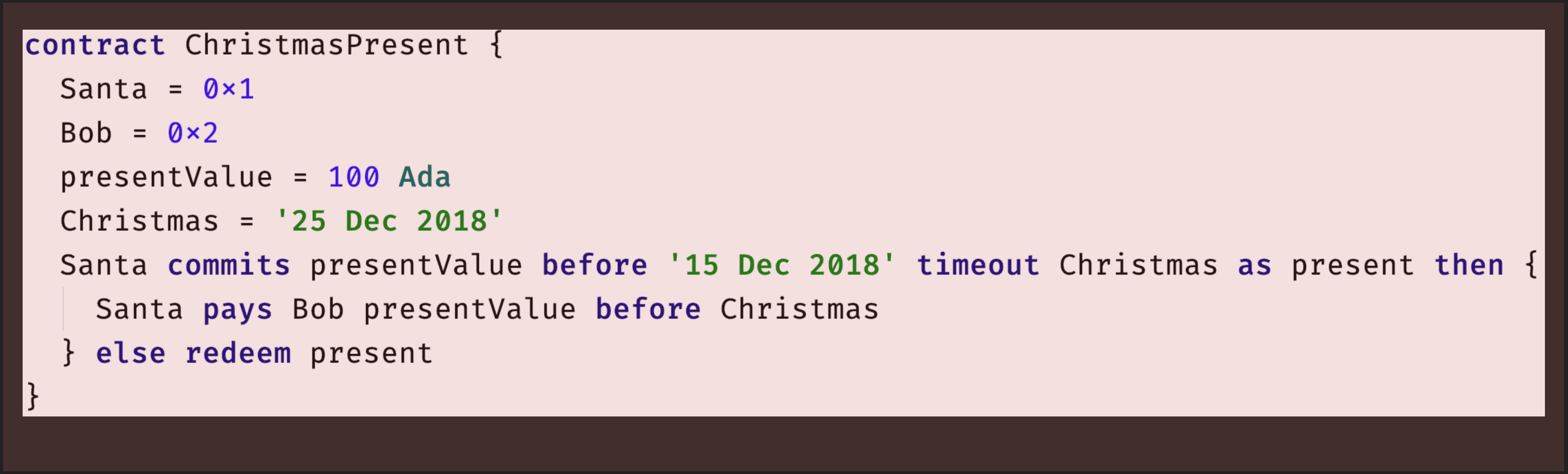
full stack + ecosystem

www.iohk.io

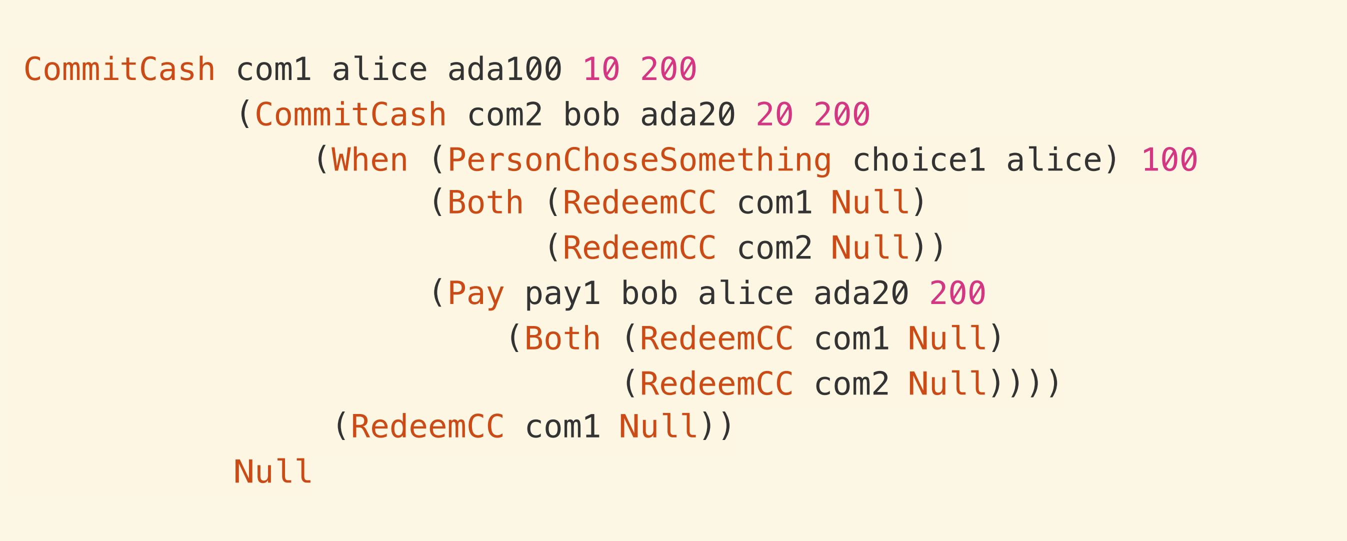
github.com/input-output-hk/marlowe

extra slides

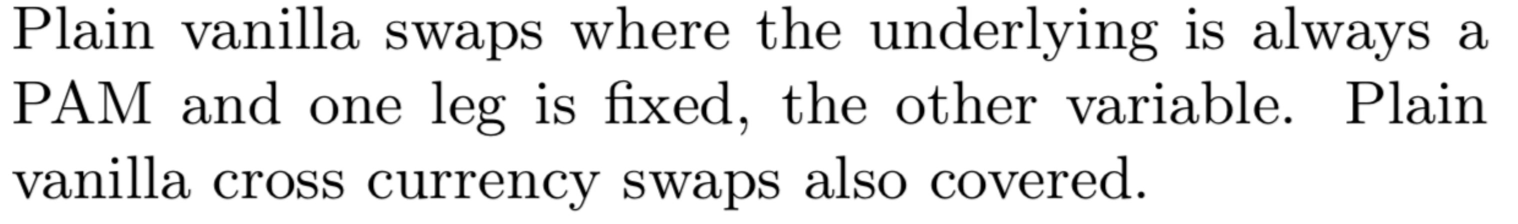
Syntax WTF!



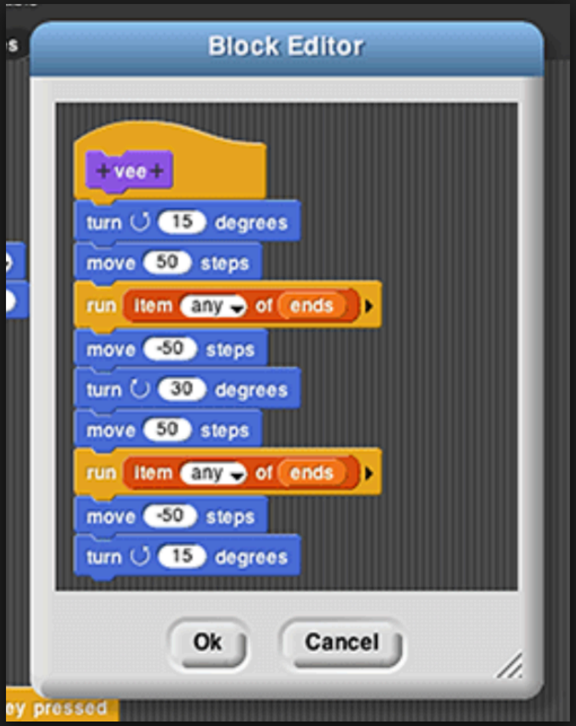
```
contract ChristmasPresent {
  Santa = 0×1
  Bob = 0×2
  presentValue = 100 Ada
  Christmas = '25 Dec 2018'
  Santa commits presentValue before '15 Dec 2018' timeout Christmas as present then {
    Santa pays Bob presentValue before Christmas
  } else redeem present
}
```



```
CommitCash com1 alice ada100 10 200
          (CommitCash com2 bob ada20 20 200
              (When (PersonChoseSomething choice1 alice) 100
                    (Both (RedeemCC com1 Null)
                          (RedeemCC com2 Null))
                    (Pay pay1 bob alice ada20 200
                        (Both (RedeemCC com1 Null)
                              (RedeemCC com2 Null))))
               (RedeemCC com1 Null))
          Null
```



Plain vanilla swaps where the underlying is always a PAM and one leg is fixed, the other variable. Plain vanilla cross currency swaps also covered.



```
+vee+
turn ↺ 15 degrees
move 50 steps
run item any of ends
move -50 steps
turn ↺ 30 degrees
move 50 steps
run item any of ends
move -50 steps
turn ↺ 15 degrees
```