Marlowe: financial contracts on blockchain

Pablo Lamela Seijas and Simon Thompson
School of Computing, University of Kent

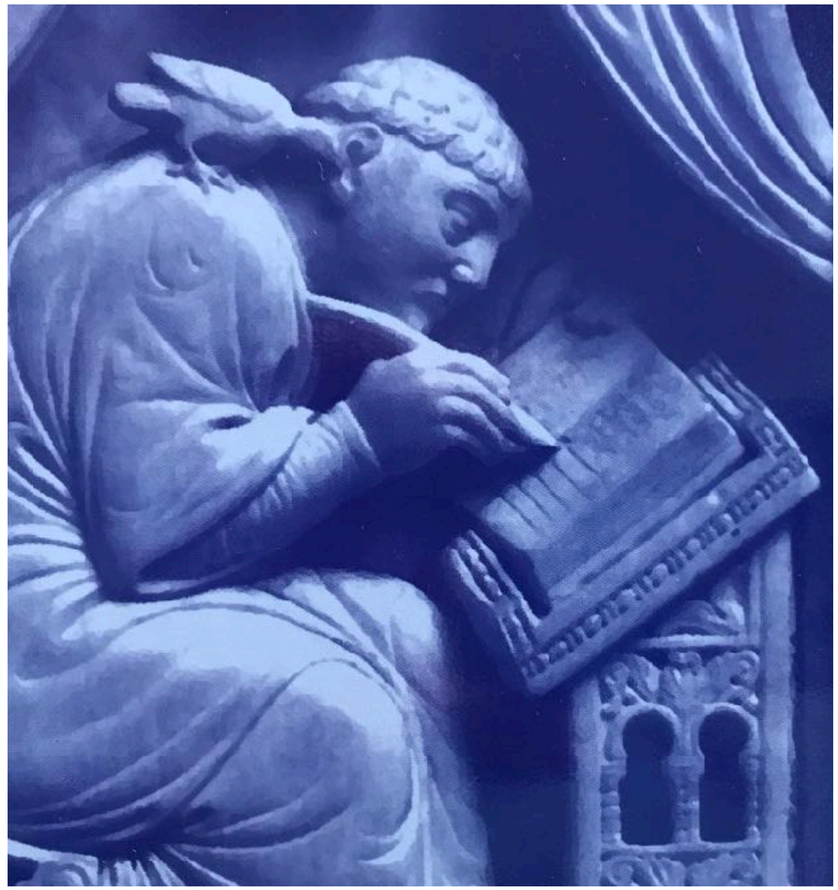
Blockchains

Ledger

An irrevocable record of …

… ownership, monetary transactions, etc.

Trust the monk

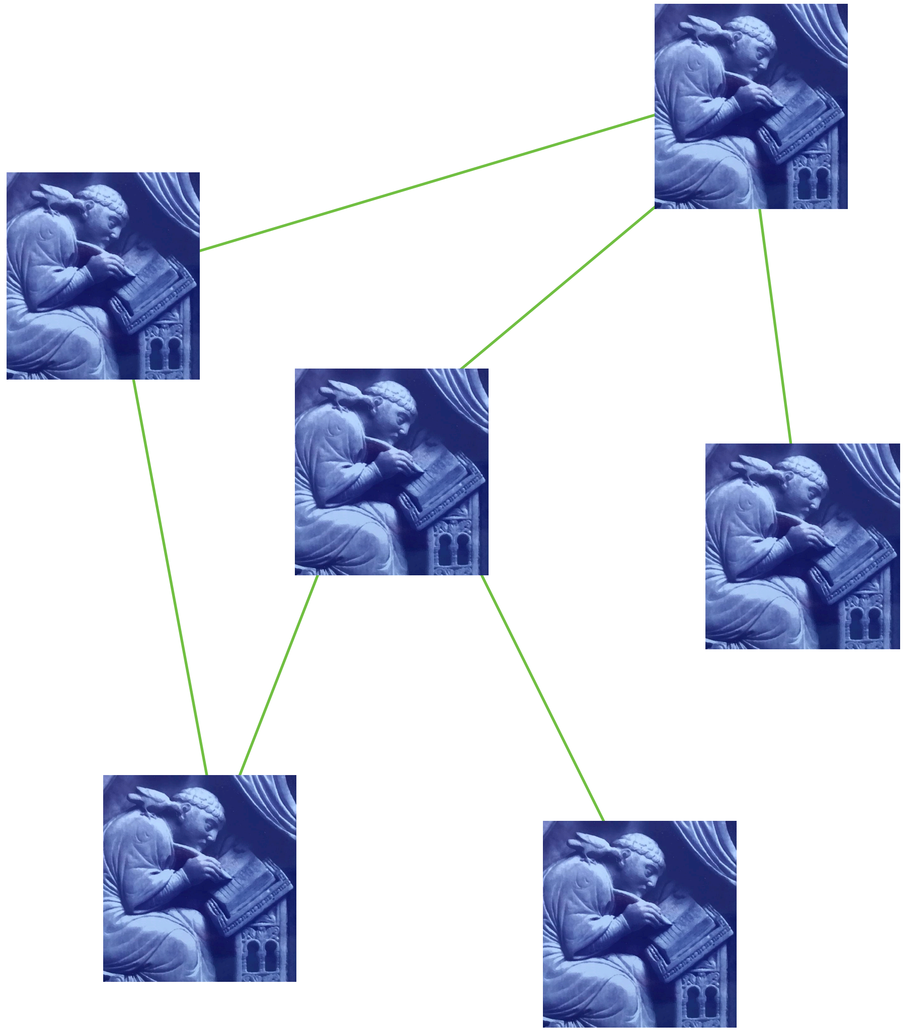


Distributed ledger

Share information and update accordingly.

Consistency / Availability / Partition Tolerance

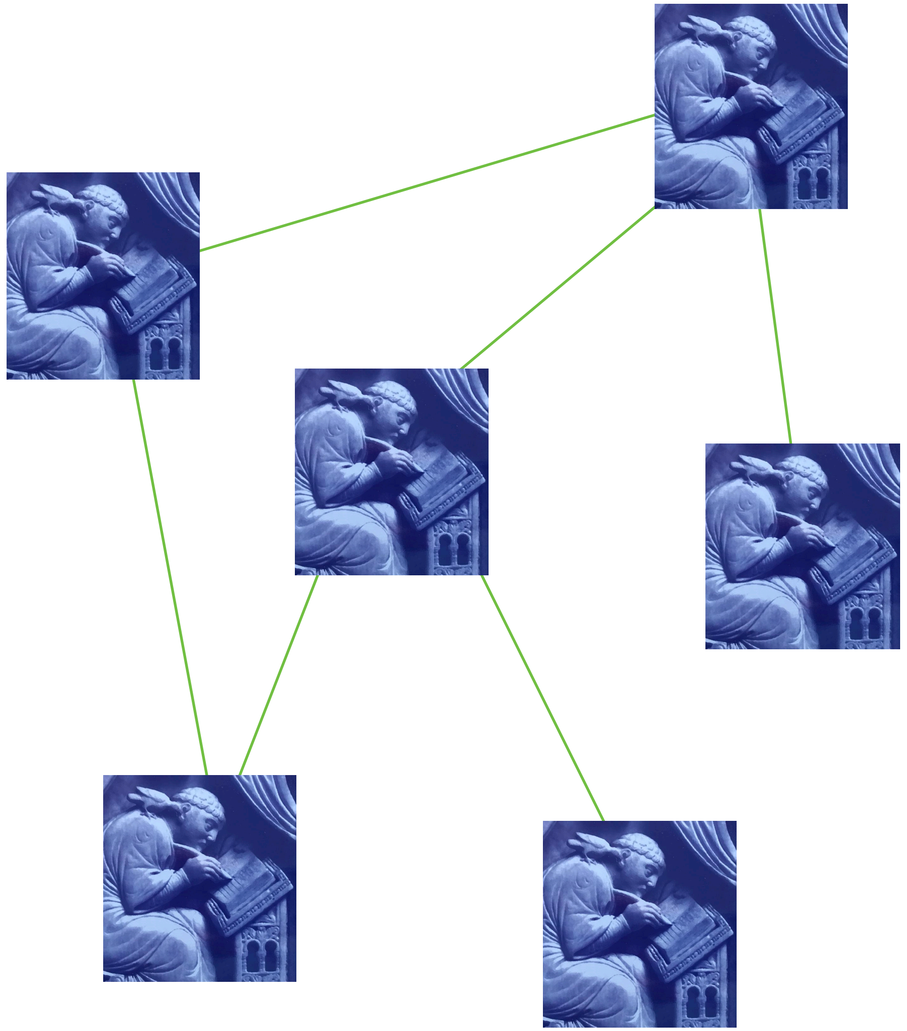
Trust the monks



But what if we can’t trust the monks?

What if some participants are actively hostile?

What if some just walk away?



Crypto-economics

Use cryptography to secure the past …

… and financial incentives to shape the future.

Vitalik Buterin, the proposer of Ethereum / Vlad Zamfir

Crypto 101

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

Crypto 101

To **send** a message 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** a message 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.

Cryptographic hash functions and hash pointers

A hash function takes a string of any size to a fixed-size output (e.g. 256 bits) and is efficiently computable.

Cryptographic hash functions and hash pointers

A hash function takes a string of any size to a fixed-size output (e.g. 256 bits) and is efficiently computable.

A cryptographic hash function also needs to be
- collision resistant,
- hard to invert, and
- "puzzle friendly"

Cryptographic hash functions and hash pointers

A hash function takes a string of any size to a fixed-size output (e.g. 256 bits) and is efficiently computable.

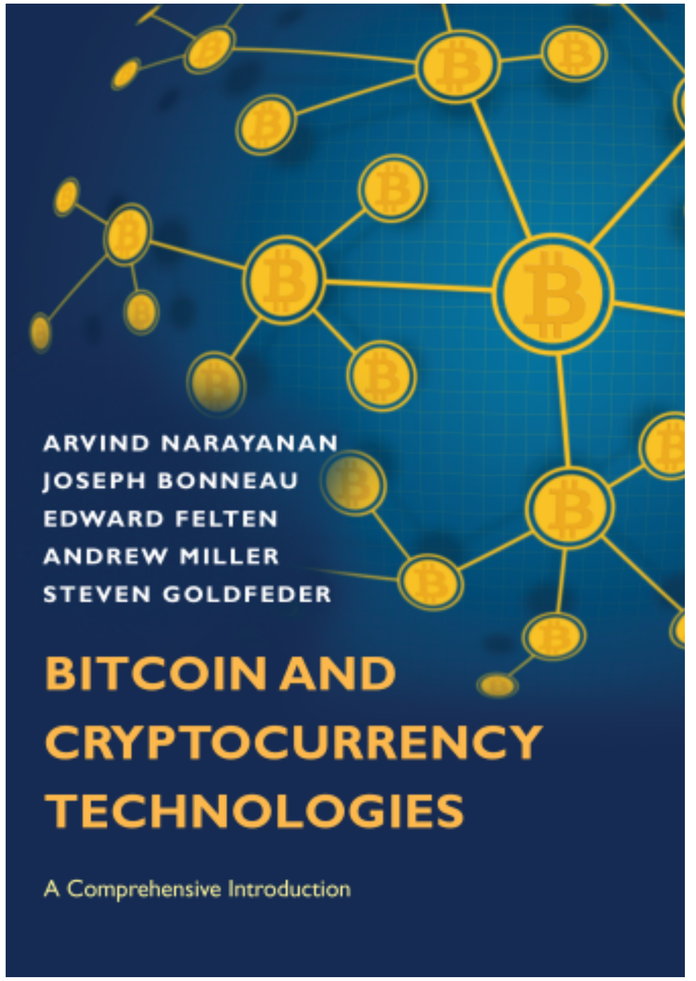
A cryptographic hash function also needs to be
- collision resistant,
- hard to invert, and
- "puzzle friendly"

A hash pointer is a pointer to some information **plus** a hash of the information itself.
Why? Can verify that the information hasn't changed.

Further reading

http://bitcoinbook.cs.princeton.ed

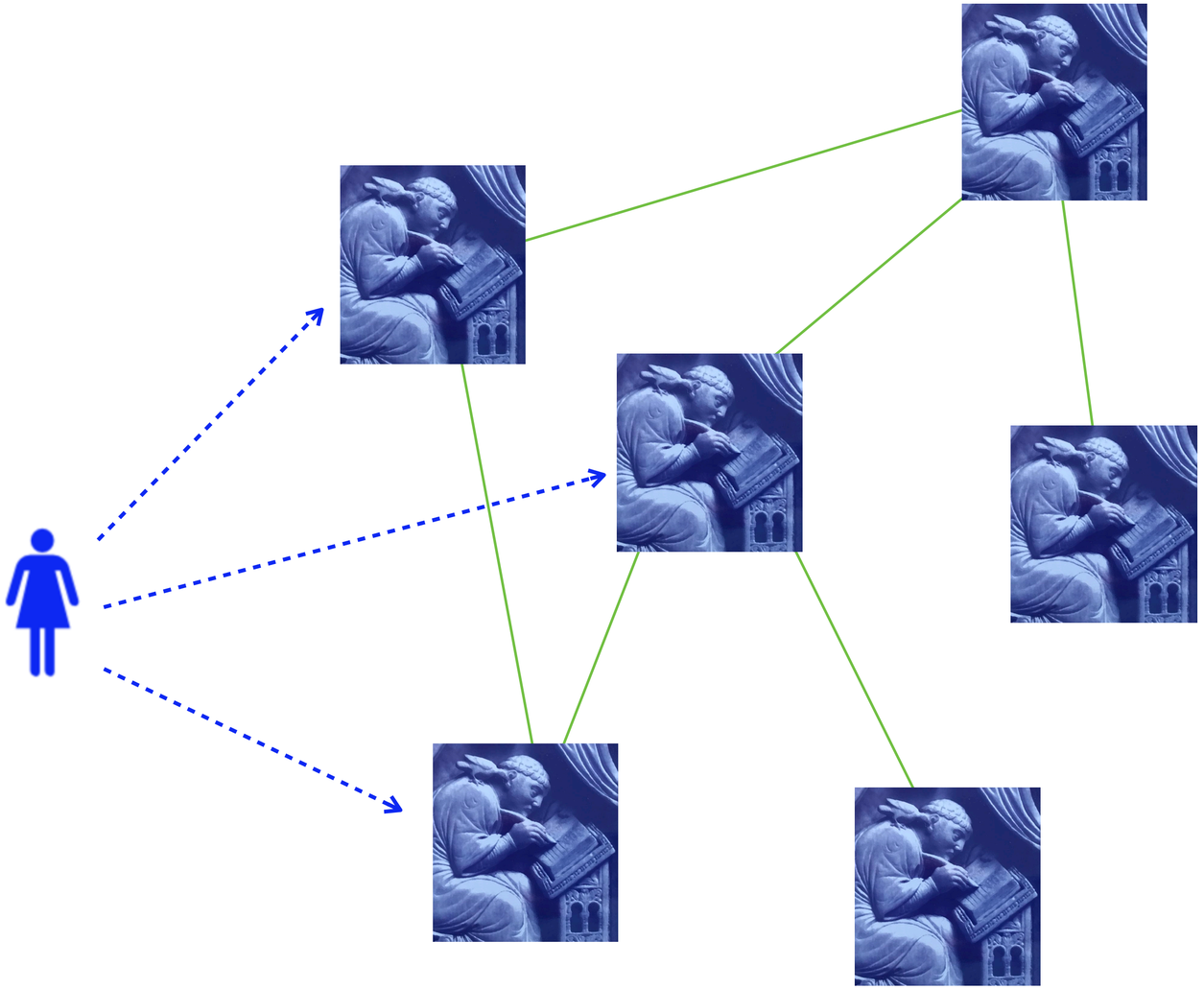
Some diagrams in these slides are illustrations from the pre-publication PDF version of the book.



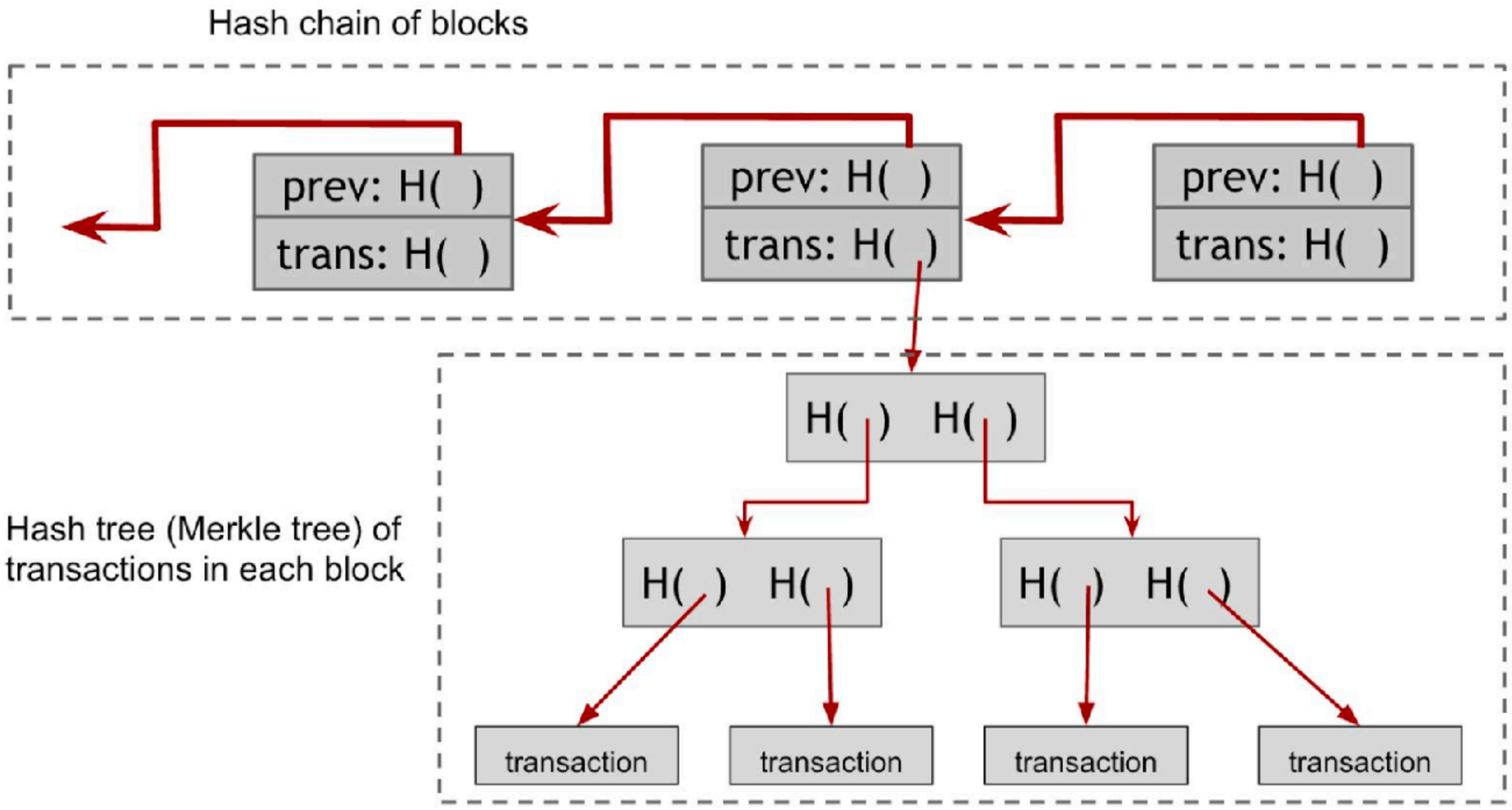
Bitcoin basics - transactions

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.

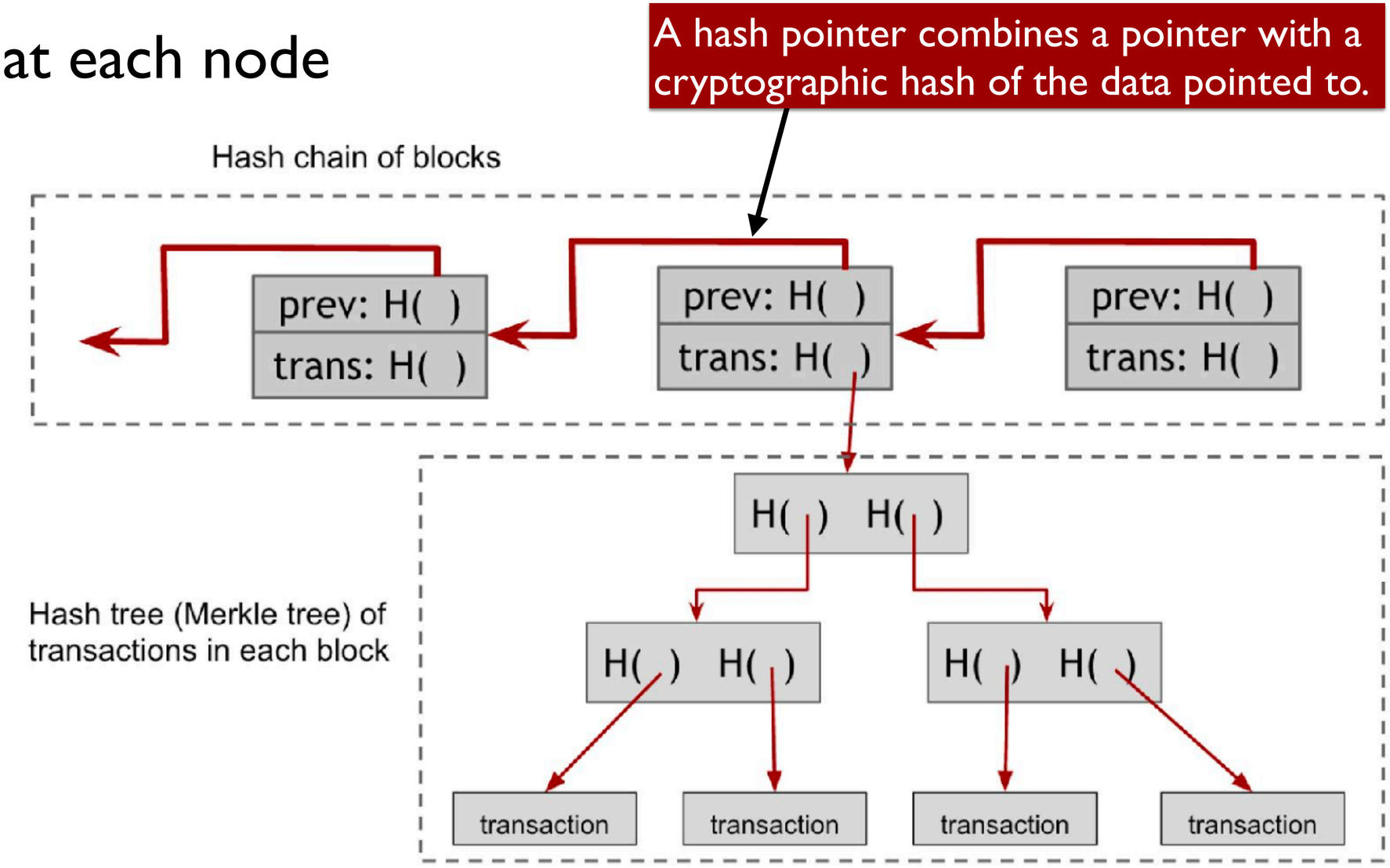


Blockchain at each node



http://bitcoinbook.cs.princeton.ed

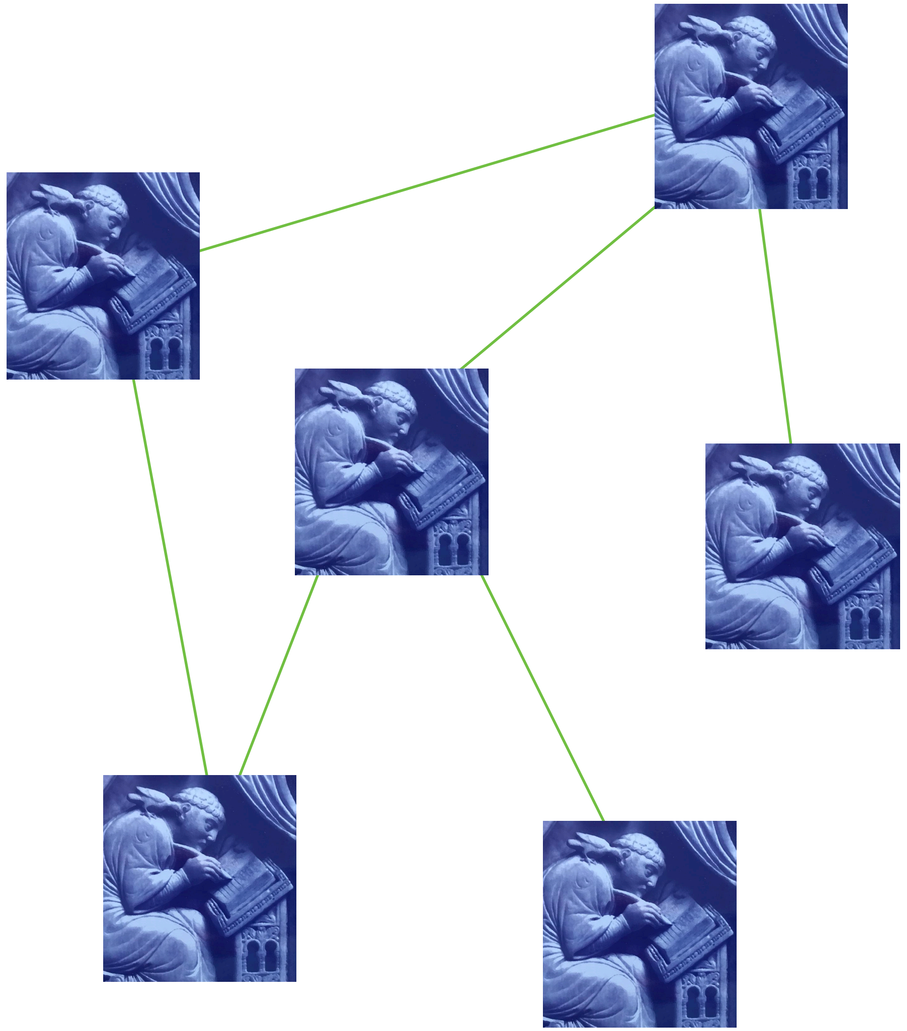
Blockchain at each node



http://bitcoinbook.cs.princeton.ed

Bitcoin basics - block creation

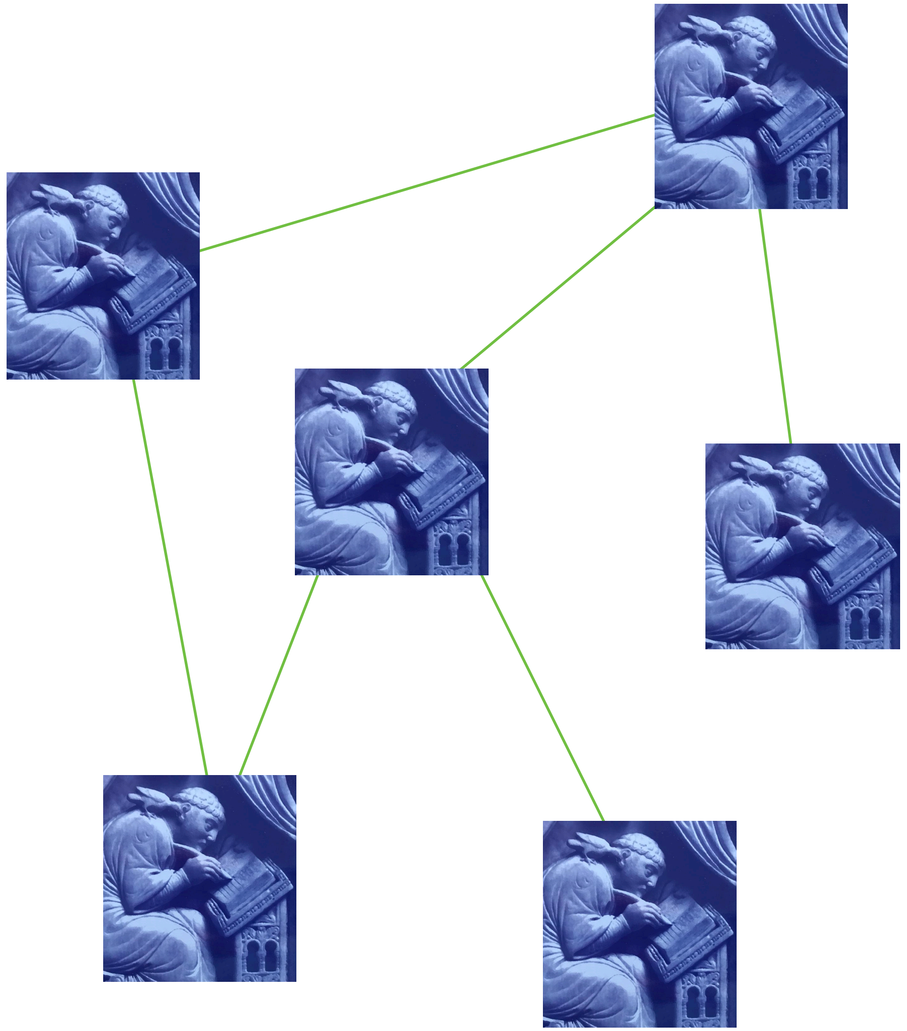
Each node collects new transactions into a block.



Bitcoin basics - block creation

Each node collects new transactions into a block.

Each round, a random node broadcasts its block.

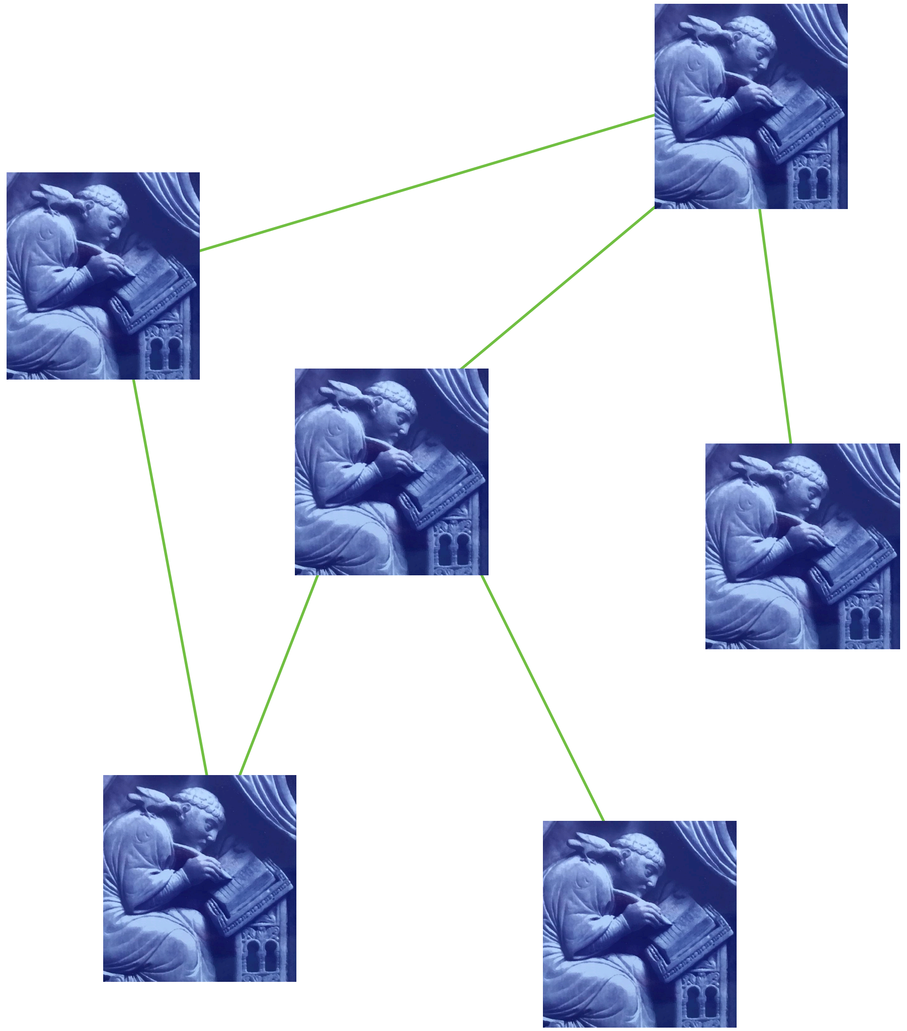


Bitcoin basics - block creation

Each node collects new transactions into a block.

Each round, a random node broadcasts its block.

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



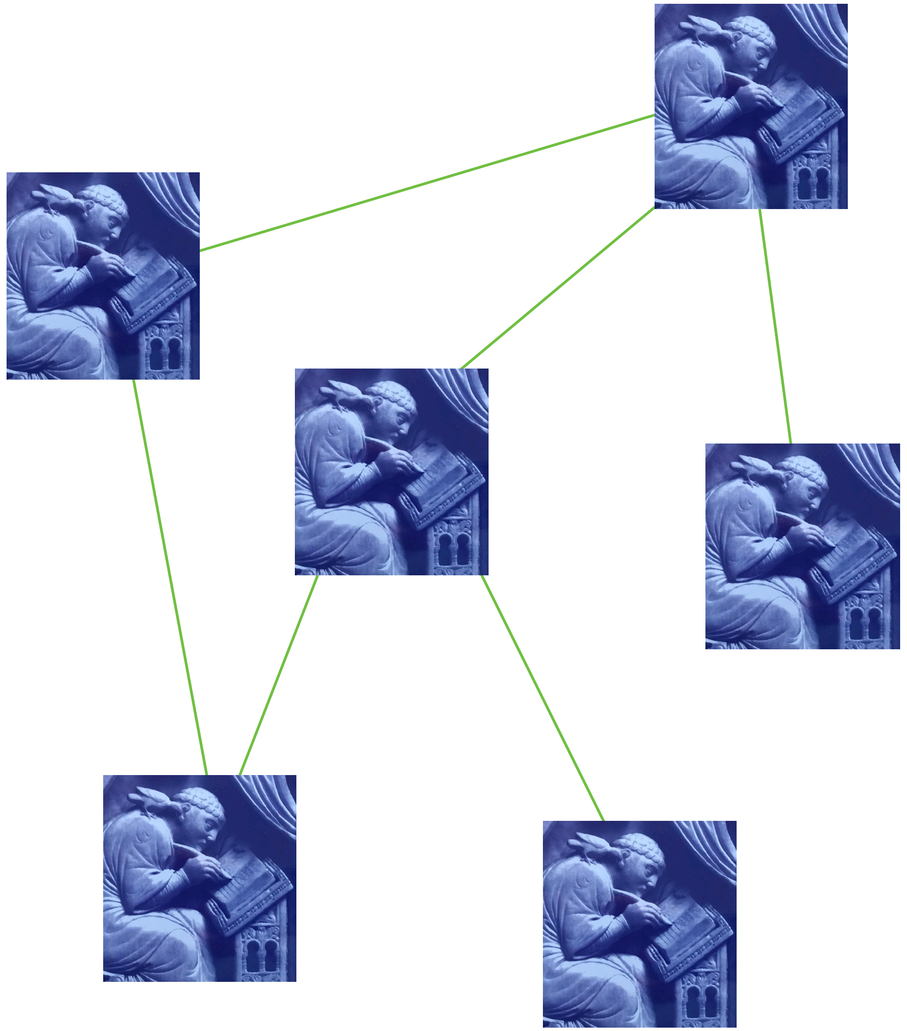
Bitcoin basics - block creation

Each node collects new transactions into a block.

Each round, a random 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.



Bitcoin basics - keeping things honest

Reward a node with Ƀ for block creation.

Collect transaction fees for the block too.

Bitcoin basics - keeping things honest

Reward a node with Ƀ for block creation.

Collect transaction fees for the block too.

Select the “random” node by mining:

find a “small enough” hash value of:
- the previous block hash,
- the transactions in this block, and
- a random nonce.

This is “proof of work”

Bitcoin basics - keeping things honest

Reward a node with Ƀ for block creation.

Collect transaction fees for the block too.

Select the "random" node by mining:

find a "small enough" hash value of:
- the previous block hash,
- the transactions in this block, and
- a random nonce.

This is "proof of work"

Does this work? Yes (so far).

Can assume N% honest and model …

… or use game theory where each player acts to maximise their payoff.

Theorems not yet proved.

Bitcoin basics - keeping things honest

Reward a node with Ƀ for block creation.

Collect transaction fees for the block too.

Select the "random" node by mining:

find a "small enough" hash value of:
- the previous block hash,
- the transactions in this block, and
- a random nonce.

This is "proof of work"

Does this work? Yes (so far).

Can assume N% honest and model …

… or use game theory where each player acts to maximise their payoff.

Theorems not yet proved.

"Proof of stake" also available …

What do we need to carry forward?

Validation, and computation in general, is replicated at every node.

Incentives needed to ensure that no bad things happen …

… and indeed to ensure that things continue to happen.

We can assume "crypto as a service".

Computations on blockchain

Computation = Scripting

Computation model

- registers
- stack,
- functional, …

Computation = Scripting

Computation model

- registers
- stack,
- functional, …

Restricted

- Turing (in)complete
- bound stack size, steps, …
- avoid DoS etc.

Computation = Scripting

Computation model	Restricted	Interaction
- registers - stack, - functional, …	- Turing (in)complete - bound stack size, steps, … - avoid DoS etc.	- outside world - other contracts - randomness, …

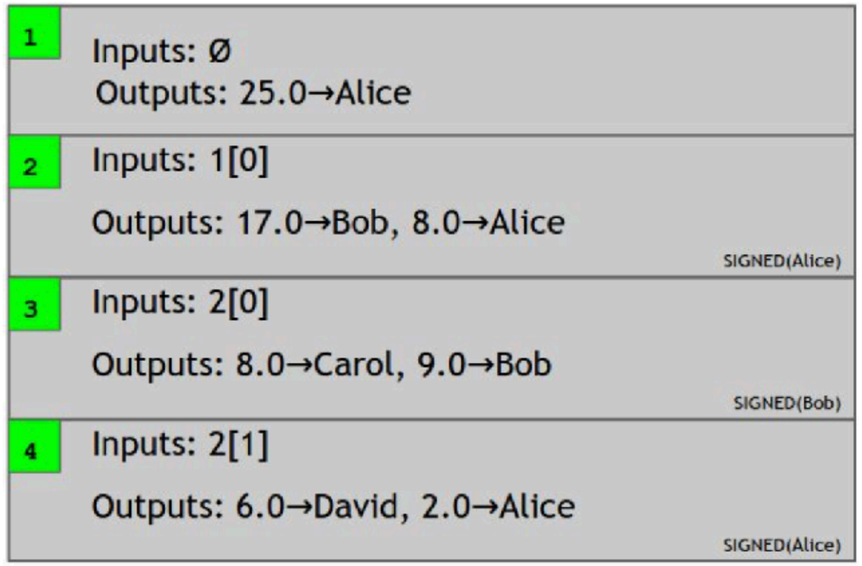
What is a bitcoin transaction?

Each transaction has inputs and outputs.

It needs to be signed by the input provider …

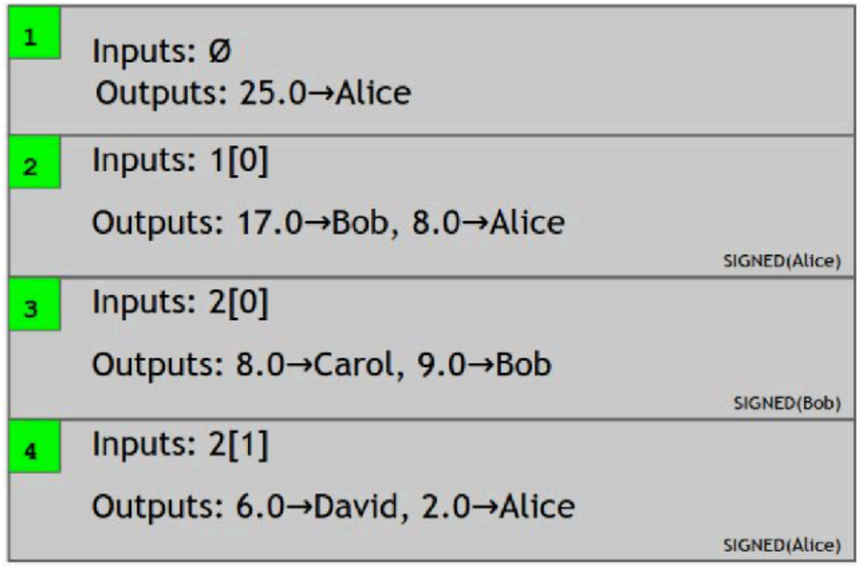
… who received output in the earlier transaction.

Validation? Does the output of the earlier transaction match the input of the current?



Abstractly

```
do
   x <- redeemer
   validator x
```



But what actually happens in Bitcoin?

Concatenate `scriptSig` with `scriptPubKey`

Effect is to compare a signature with what is expected,

Both are *scripts* run in a Forth-like model.

Computation in Bitcoin

Forth-like: stack computation.

Limited: conditionals, but no loops or recursion. Bounds on size.

Limited set of opcodes: 256 and many disabled. E.g.

```
OP_EQUALVERIFY
OP_CHECKSIG
OP_CHECKMULTISIG
```

Pay to script hash `P2SH`: send your coins to the hash of this script.

Limited … can't point `P2SH` to another `P2SH` script.

Plutus Core - the Cardano Settlement Layer

```
do
  x <- redeemer
  validator x
```

Typed, strict, λ-calculus: transaction validation scripting language on blockchain systems.

`Comp`: reader monad with a failure model and access to environment values: `txhash`, `blocktime`, `blocknumber`.

Limit computation steps for each contract, so need predictability …

Ethereum: a “global distributed computer”

- Stack-based
- Turing complete
- Persistent state
- Solidity

- Payment of gas (ether) for computation,
- stack bounded, …

Rational reconstruction of EVM, IELE, may be the target of our DSL work.

Other models are available …

Nxt

A “fat” API which the implementers argue is safer than a VM/language

… but which presents a broad attack surface.

Other languages

Tezos: stack-based, static types. OCaml implemented.

Hawk = Ethereum + Zerocash for anonymity.

Other languages

- BPMN
- FSMs
- FCL
- Rholang
- Hyperledger: docker + code

Computation = Scripting

Computation model	Restricted	Interaction
- registers - stack, - functional, …	- Turing (in)complete - bound stack size, steps, … - avoid DoS etc.	- outside world - other contracts - randomness, …

DSLs for Smart Contracts

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

… but not as we know it

Enforcement

The legal system ensures financial contracts …

… but a contract on blockchain must enforce itself.

… but not as we know it

Enforcement

The legal system ensures financial contracts …

… but a contract on blockchain must enforce itself.

Double spend

Blockchain designed to prevent spending the same money twice …

… but that's the antithesis of the way that credit works.

And other domains …

How to Use Bitcoin to Play Decentralized Poker

Ranjit Kumaresan
MIT
ranjit@csail.mit.edu

Tal Moran
IDC Herzliya
talm@idc.ac.il

Iddo Bentov
Technion
idddo@cs.technion.ac.il

ABSTRACT

Back and Bentov (arXiv 2014) and Andrychowicz *et al.* (Security and Privacy 2014) introduced techniques to perform secure multi-party computations on Bitcoin. Among other things, these works constructed lottery protocols that ensure that any party that aborts after learning the outcome pays a monetary penalty to all other parties. Following this, Andrychowicz *et al.* (Bitcoin Workshop 2014) and concurrently Bentov and Kumaresan (Crypto 2014) extended the solution to arbitrary secure function evaluation while guaranteeing fairness in the following sense: any party that aborts after learning the output pays a monetary penalty to all parties that did not learn the output. Andrychowicz *et al.* (Bitcoin Workshop 2014) also suggested extending to scenarios where parties receive a payoff according to the output of a secure function evaluation, and outlined a 2-party protocol for the same that in addition satisfies the

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*

Keywords

Secure Computation; Bitcoin; Smart Contracts; Markets; Poker.

1. INTRODUCTION

Once there were two "mental chess" experts who had become tired of their favorite pastime. Let's play "mental poker" for some variety suggested one. "Sure" said the other. "Just let me deal!"

One or many DSLs?

Different application areas

Finance, gaming, …

Multiple levels

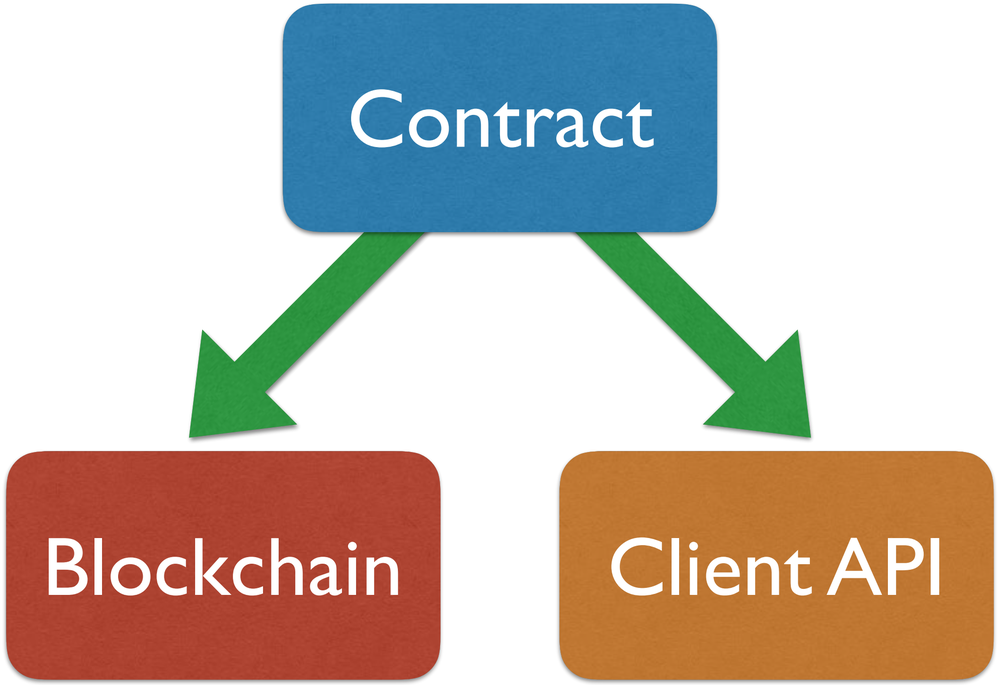
Settlement vs Computation layers in Cardarno.

Infrastructure

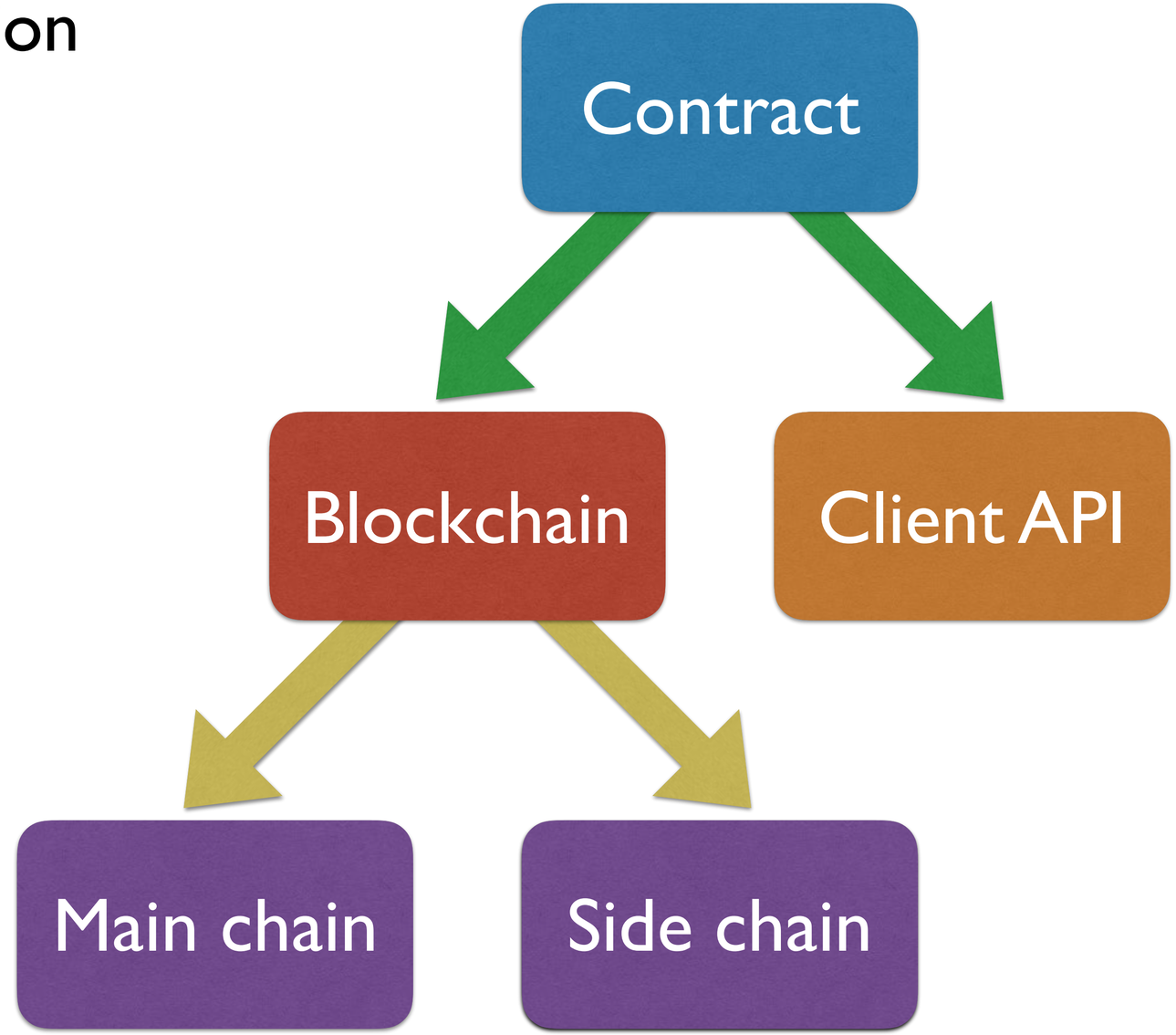
E.g. languages for crypto.

Compilation

Compilation



Compilation



Marlowe

A DSL as a Haskell `data` type.

An executable small-step semantics.

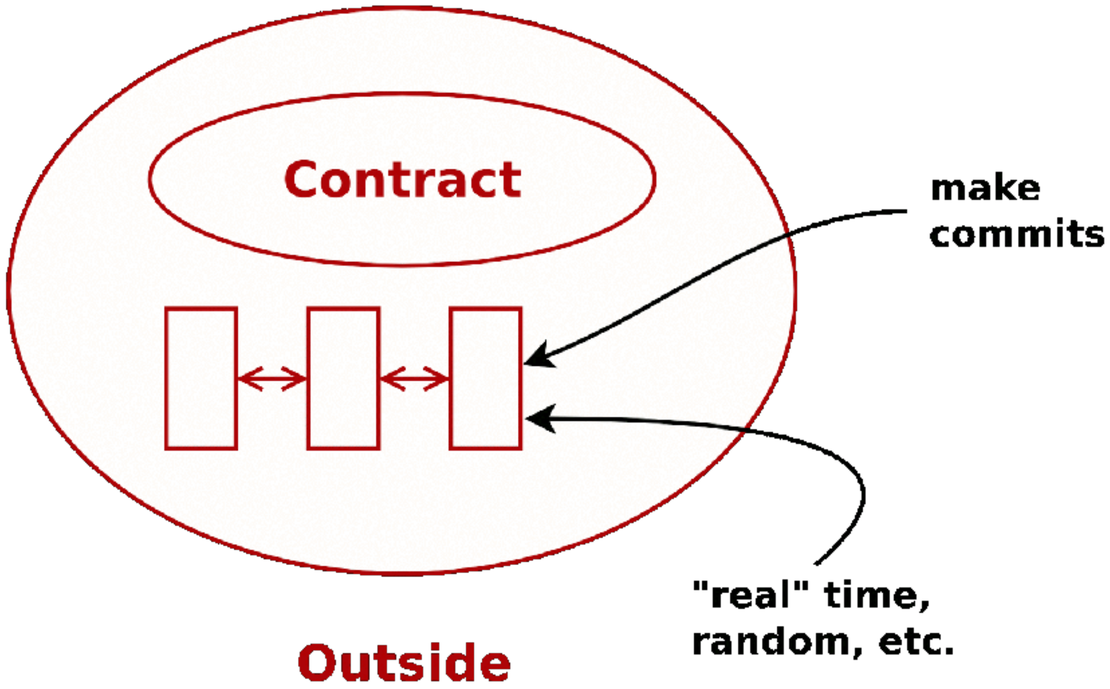
Observations and questions.

Marlowe

A DSL as a Haskell `data` type.

An executable small-step semantics.

Observations and questions.

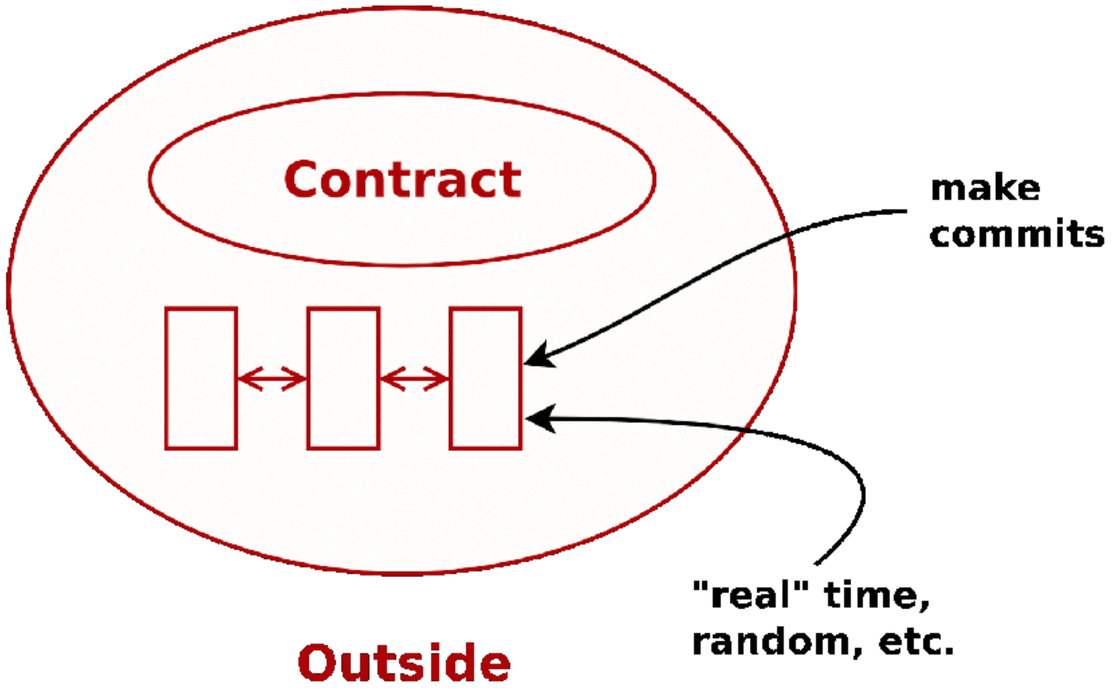


Interactions with the outside world

Real values: e.g. *"the spot price of oil in Aberdeen at 12 noon, 31-12-17"*.

Random values.

Commitments …



Commitments

Cash

Commit a certain amount of cash.

Needs to be time-limited to avoid “walk away” …

Commitments

Cash

Commit a certain amount of cash.

Needs to be time-limited to avoid “walk away” …

Modality

We can’t *require* a commitment: can only *ask for* one.

Only wait bounded time.

Commitments

Cash

Commit a certain amount of cash.

Needs to be time-limited to avoid "walk away" …

Modality

We can't *require* a commitment: can only *ask for* one.

Only wait bounded time.

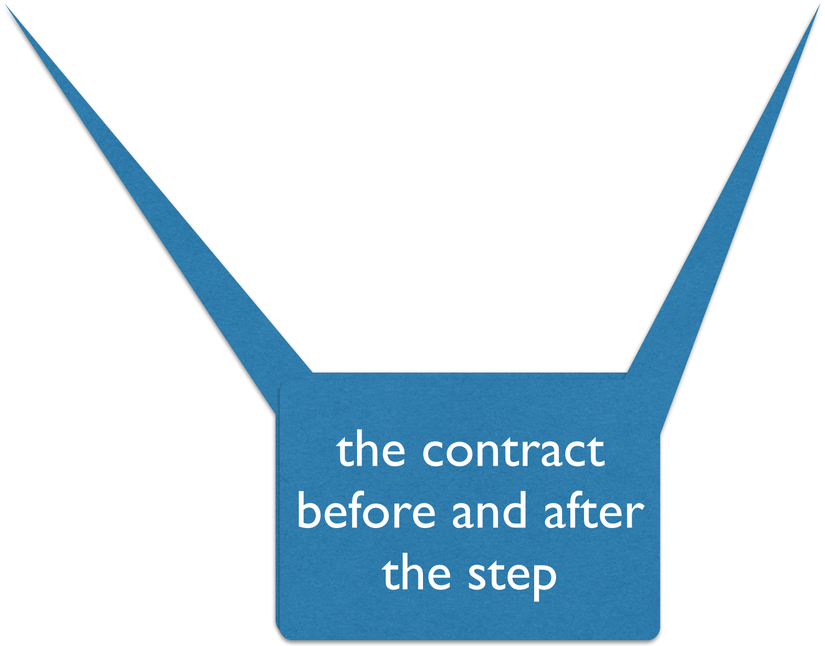
Choices

Commit to a choice of a value from a certain set.

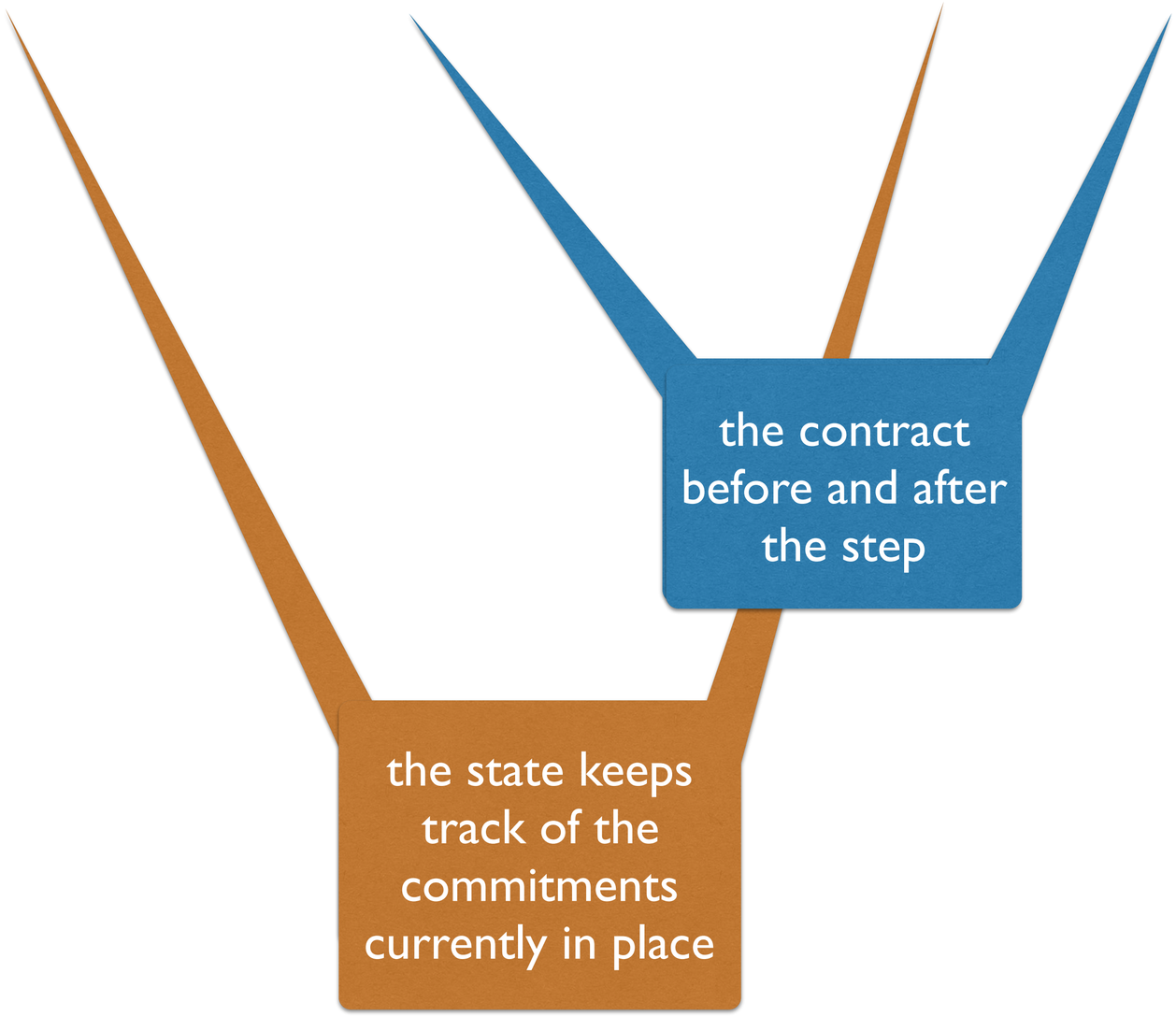
To play games need to commit to hidden choices.

```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```

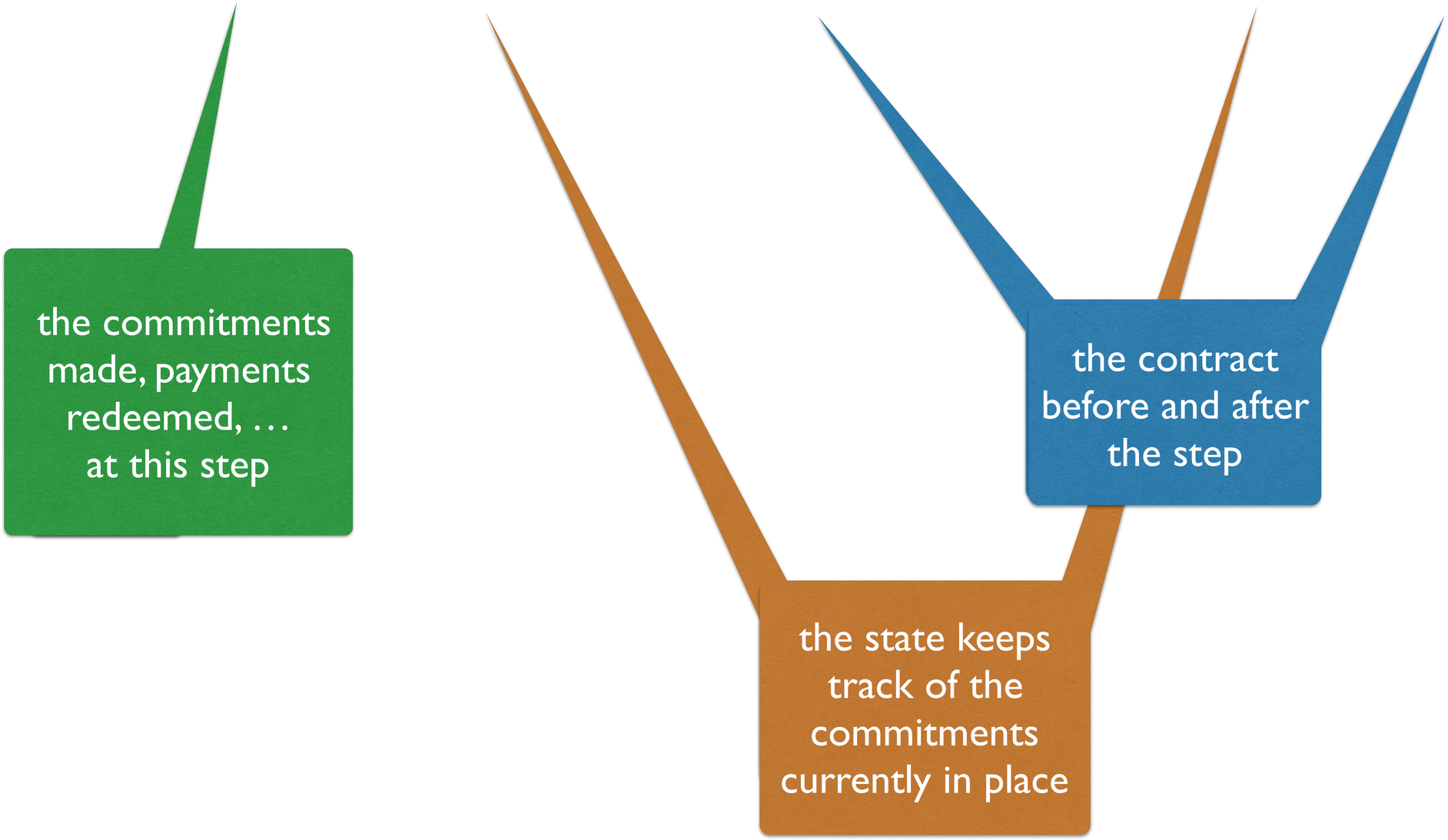
```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```



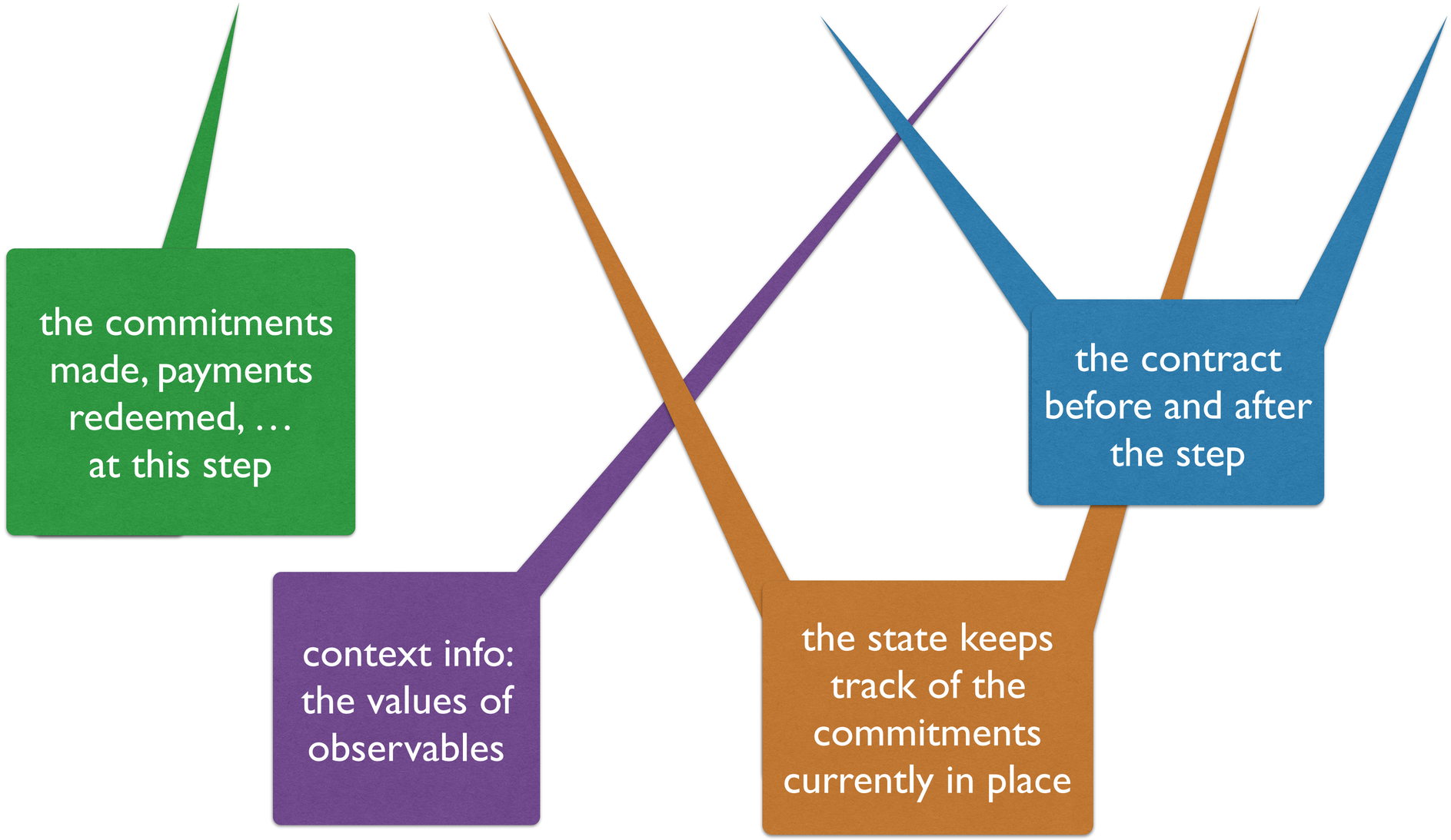
```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```



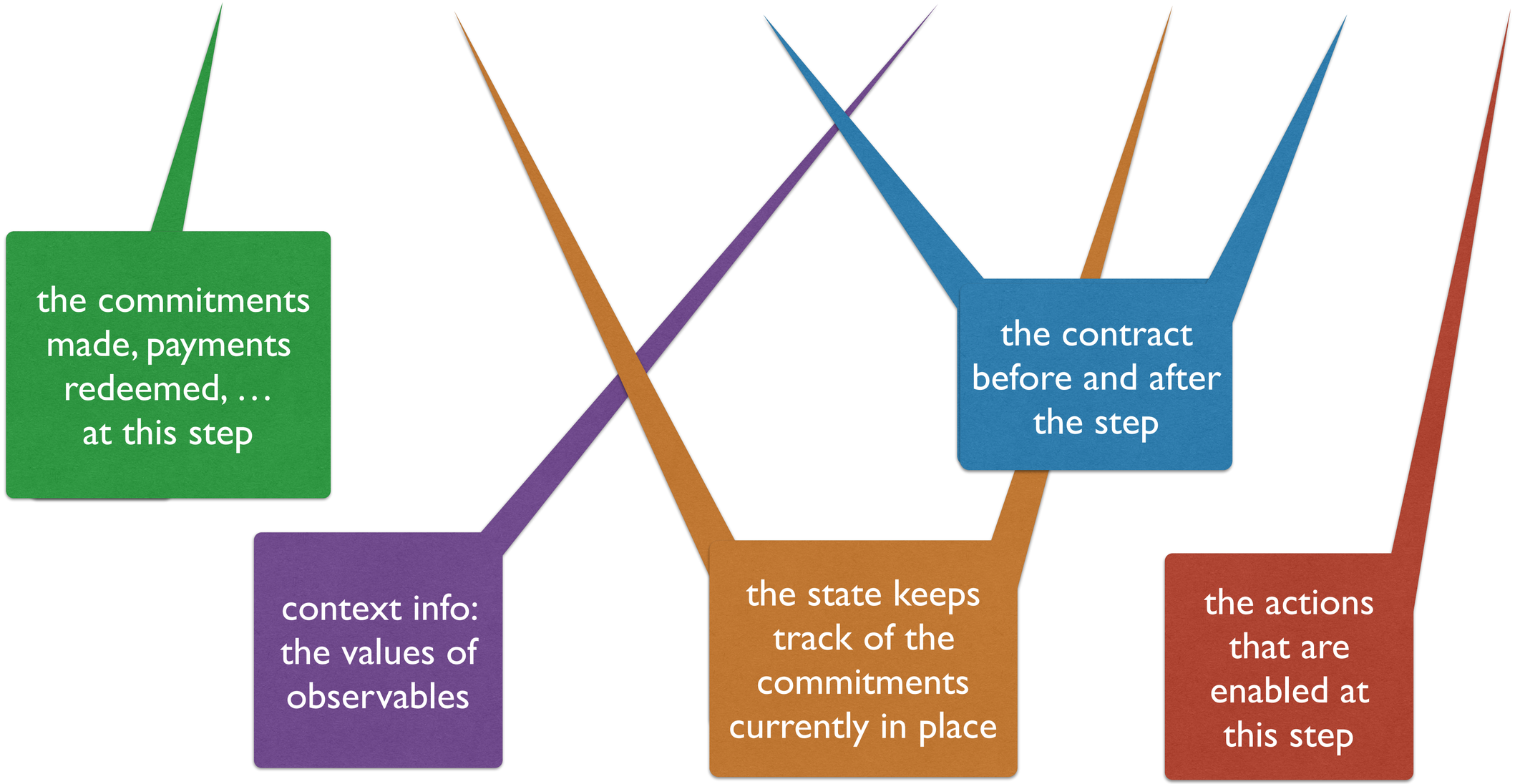
```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```



```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```



```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```



```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```

Observations

We assume that these are recorded in a way that can be reused in verification step.

```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```

Observations

We assume that these are recorded in a way that can be reused in verification step.

Actions

These, too, are assumed to have an effect on the blockchain by trans-actions being issued.

```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```

Observations

We assume that these are recorded in a way that can be reused in verification step.

Actions

These, too, are assumed to have an effect on the blockchain by trans-actions being issued.

Quiescence

We say a step is *quiescent* if the same contract results: it *makes progress* otherwise.

```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```

Redemption

We wrap each call to `step` by `fullStep` which also enables redeeming expired commitments.

```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```

Redemption

We wrap each call to `step` by `fullStep` which also enables redeeming expired commitments.

Monadic?

Possibilities of making a monad here … but for the moment kept completely concrete.

```
data Contract =
    Null |
    CommitCash IdentCC Person Cash Timeout Timeout Contract Contract |
    RedeemCC IdentCC Contract |
    Pay IdentPay Person Person Cash Timeout Contract |
    Both Contract Contract |
    Choice Observation Contract Contract |
    When Observation Timeout Contract Contract
                deriving (Eq,Ord,Show,Read)
```

```
data Contract =
    Null |
    CommitCash IdentCC Person Cash Timeout Timeout Contract Contract |
```

`CommitCash idCC p n t1 t2 k1 k2`

For this contract to make progress, either

- before the timeout `t1` the user `p` makes a cash commitment of value `n` and timeout `t2` with the identifier `idCC`: generate `SuccessfulCommit` action, continue as `k1`;
- or timeout `t1` exceeded and continue as `k2`.

Otherwise it is quiescent. At timeout `t2` remaining committed cash can be redeemed, and `fullStep` enables that.

```
data Contract =
    Null |
    CommitCash IdentCC Person Cash Timeout Timeout Contract Contract |
    RedeemCC IdentCC Contract |
```

`RedeemCC idCC k`

Enables a commiter of cash to redeem it before the commitment times out.

- If the commit has already expired and was redeemed, it does nothing.
- If it has already been redeemed, then don't, and issue `DuplicateRedeem` action.

```
data Contract =
    Null |
    CommitCash IdentCC Person Cash Timeout Timeout Contract Contract |
    RedeemCC IdentCC Contract |
    Pay IdentPay Person Person Cash Timeout Contract |
```

`Pay idpay from to val expi con`

Enables a payment of `val` from `from` to `to` before `expi`, and continues as `con`

- Payment identified as `idpay`.

```
data Contract =
    Null |
    CommitCash IdentCC Person Cash Timeout Timeout Contract Contract |
    RedeemCC IdentCC Contract |
    Pay IdentPay Person Person Cash Timeout Contract |
    Both Contract Contract |
    Choice Observation Contract Contract |
    When Observation Timeout Contract Contract
                deriving (Eq,Ord,Show,Read)
```

```
data Contract =
    Null |
```

```
When obs expi k1 k2
```

Will progress either

- when the observation `obs` becomes true, and continues as `k1`, or
- when the timeout `expi` reached, and continues as `k2`.

```
    When Observation Timeout Contract Contract
                deriving (Eq,Ord,Show,Read)
```

```
data Contract =
    Null |
    CommitCash IdentCC Person Cash Timeout Timeout Contract Contract |
    RedeemCC IdentCC Contract |
    Pay IdentPay Person Person Cash Timeout Contract |
    Both Contract Contract |
    Choice Observation Contract Contract |
    When Observation Timeout Contract Contract
                deriving (Eq,Ord,Show,Read)
```

Escrow contract

```
CommitCash (IdentCC 1) 1 100 10 200
           (CommitCash (IdentCC 2) 2 20 20 200
                       (When (PersonChoseSomething (IdentChoice 1) 1) 100
                             (Both (RedeemCC (IdentCC 1) Null)
                                   (RedeemCC (IdentCC 2) Null))
                             (Pay (IdentPay 1) 2 1 20 200
                                  (Both (RedeemCC (IdentCC 1) Null)
                                        (RedeemCC (IdentCC 2) Null))))
                       (RedeemCC (IdentCC 1) Null))
           Null
```

Wait until time 10 for player 1 to commit 100 ADA until time 200.

```
CommitCash (IdentCC 1) 1 100 10 200
           (CommitCash (IdentCC 2) 2 20 20 200
                       (When (PersonChoseSomething (IdentChoice 1) 1) 100
                             (Both (RedeemCC (IdentCC 1) Null)
                                   (RedeemCC (IdentCC 2) Null))
                             (Pay (IdentPay 1) 2 1 20 200
                                  (Both (RedeemCC (IdentCC 1) Null)
                                        (RedeemCC (IdentCC 2) Null))))
                       (RedeemCC (IdentCC 1) Null))
           Null
```

Wait until time 10 for player 1 to commit 100 ADA until time 200.

similarly for player 2

```
CommitCash (IdentCC 1) 1 100 10 200
           (CommitCash (IdentCC 2) 2 20 20 200
                       (When (PersonChoseSomething (IdentChoice 1) 1) 100
                             (Both (RedeemCC (IdentCC 1) Null)
                                   (RedeemCC (IdentCC 2) Null))
                             (Pay (IdentPay 1) 2 1 20 200
                                  (Both (RedeemCC (IdentCC 1) Null)
                                        (RedeemCC (IdentCC 2) Null))))
                       (RedeemCC (IdentCC 1) Null))
           Null
```

Wait until time 10 for player 1 to commit 100 ADA until time 200.

similarly for player 2

```
CommitCash (IdentCC 1) 1 100 10 200
           (CommitCash (IdentCC 2) 2 20 20 200
                       (When (PersonChoseSomething (IdentChoice 1) 1) 100
                             (Both (RedeemCC (IdentCC 1) Null)
                                   (RedeemCC (IdentCC 2) Null))
                             (Pay (IdentPay 1) 2 1 20 200
                                  (Both (RedeemCC (IdentCC 1) Null)
                                        (RedeemCC (IdentCC 2) Null))))
                       (RedeemCC (IdentCC 1) Null))
           Null
```

if player 1 chooses to before time 100, both people get their money back

Wait until time 10 for player 1 to commit 100 ADA until time 200.

similarly for player 2

```
CommitCash (IdentCC 1) 1 100 10 200
           (CommitCash (IdentCC 2) 2 20 20 200
                       (When (PersonChoseSomething (IdentChoice 1) 1) 100
                             (Both (RedeemCC (IdentCC 1) Null)
                                   (RedeemCC (IdentCC 2) Null))
                             (Pay (IdentPay 1) 2 1 20 200
                                  (Both (RedeemCC (IdentCC 1) Null)
                                        (RedeemCC (IdentCC 2) Null))))
                       (RedeemCC (IdentCC 1) Null))
```

if player 1 chooses to before time 100, both people get their money back

otherwise, player 1 gets her money and the 20 ADA from player 2

```
CommitCash (IdentCC 1) 1 100 10 200
           (CommitCash (IdentCC 2) 2 20 20 200
                       (When (PersonChoseSomething (IdentChoice 1) 1) 100
                             (Both (RedeemCC (IdentCC 1) Null)
                                   (RedeemCC (IdentCC 2) Null))
                             (Pay (IdentPay 1) 2 1 20 200
                                  (Both (RedeemCC (IdentCC 1) Null)
                                        (RedeemCC (IdentCC 2) Null))))
                       (RedeemCC (IdentCC 1) Null))
```

Wait until time 10 for player 1 to commit 100 ADA until time 200.

similarly for player 2

if player 1 chooses to before time 100, both people get their money back

otherwise, player 1 gets her money and the 20 ADA from player 2

action if player 2 didn't commit in time

Implementation

Stepping through

Interactively step through the evaluation of a contract: input commitments, values at each stage; corresponding actions generated.

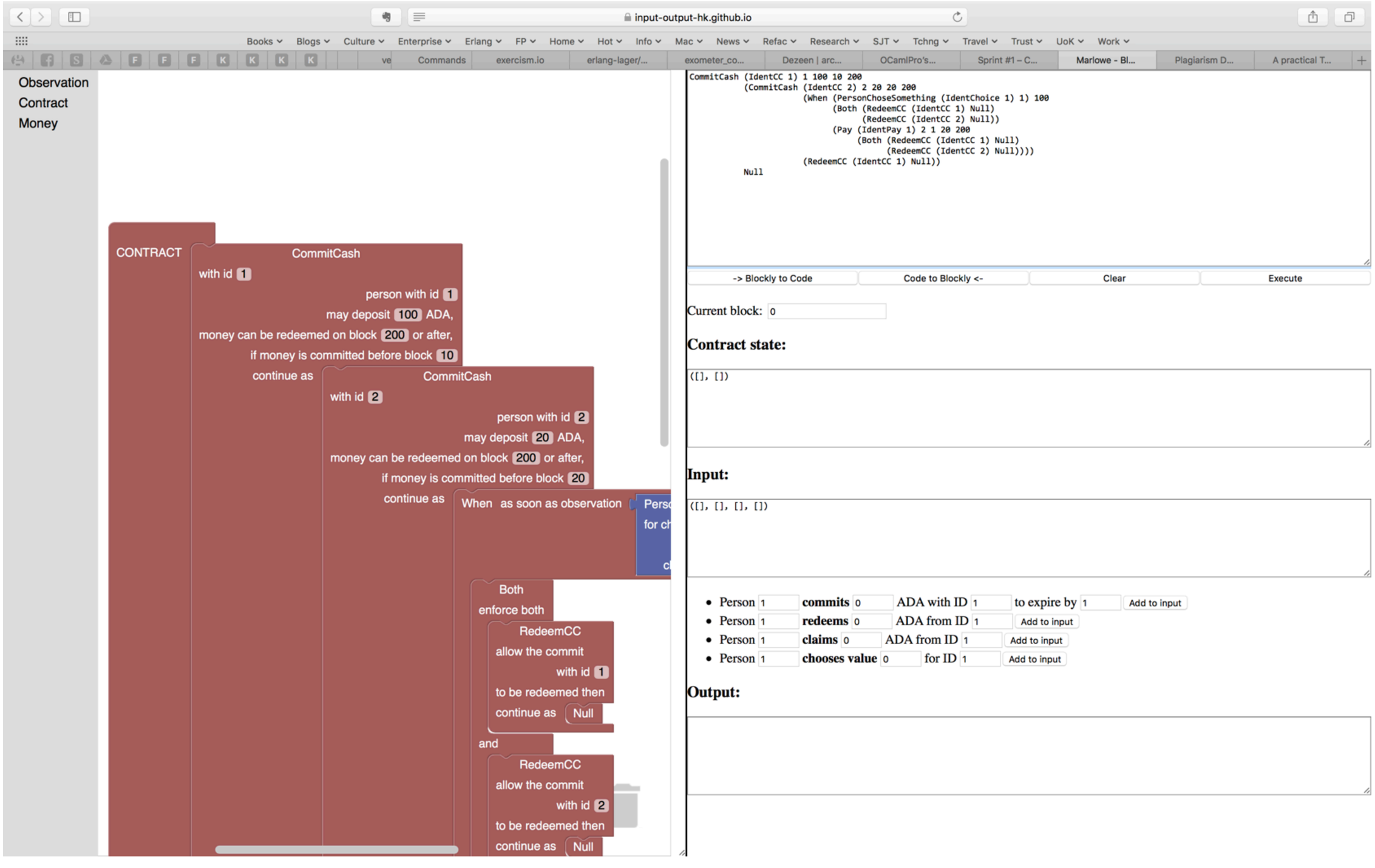
Implementation

Stepping through

Interactively step through the evaluation of a contract: input commitments, values at each stage; corresponding actions generated.

Visualisation

Visualise as finitely branching decision trees.


input-output-hk.github.io
Books Blogs Culture Enterprise Erlang FP Home Hot Info Mac News Refac Research SJT Tchng Travel Trust UoK Work
Commands exercism.io erlang-lager/... exometer_co... Dezeen | arc... OCamlPro's... Sprint #1 – C... Marlowe - Bl... Plagiarism D... A practical T...
Observation
Contract
Money
CONTRACT
CommitCash
with id 1
person with id 1
may deposit 100 ADA,
money can be redeemed on block 200 or after,
if money is committed before block 10
continue as
CommitCash
with id 2
person with id 2
may deposit 20 ADA,
money can be redeemed on block 200 or after,
if money is committed before block 20
continue as
When as soon as observation
Both
enforce both
RedeemCC
allow the commit
with id 1
to be redeemed then
continue as Null
and
RedeemCC
allow the commit
with id 2
to be redeemed then
continue as Null
CommitCash (IdentCC 1) 1 100 10 200
(CommitCash (IdentCC 2) 2 20 20 200
(When (PersonChoseSomething (IdentChoice 1) 1) 100
(Both (RedeemCC (IdentCC 1) Null)
(RedeemCC (IdentCC 2) Null))
(Pay (IdentPay 1) 2 1 20 200
(Both (RedeemCC (IdentCC 1) Null)
(RedeemCC (IdentCC 2) Null))))
(RedeemCC (IdentCC 1) Null))
Null
-> Blockly to Code
Code to Blockly <-
Clear
Execute
Current block: 0
Contract state:
([], [])
Input:
([], [], [], [])
Person 1 commits 0 ADA with ID 1 to expire by 1 Add to input
Person 1 redeems 0 ADA from ID 1 Add to input
Person 1 claims 0 ADA from ID 1 Add to input
Person 1 chooses value 0 for ID 1 Add to input
Output:

Design decisions: push or pull?

Push

The contract makes things happen, such as payments.

Pull

The contract enables things to happen, but participants must perform them.

Design decisions: accounts or UTxO?

Accounts

The blockchain keeps a record of how much each participant has.

UTxO

Only transactions: spend the **U**nspent **Tx**ransaction **O**utputs in another Tx.

Design decisions

Time

We've been vague here: may need block number, real time and slot time.

Monadic

Could use monads to hide state, side-effects, identifier generation etc.

The `Both` constructor

This is a little problematic: not commutative because of side-effects.

Formal analyses for the model

Evaluation

Can we show that all contracts can be run to completion, if certain conditions / inputs etc?

Formal analyses for the model

Evaluation

Can we show that all contracts can be run to completion, if certain conditions / inputs etc?

Validity

Is it possible to characterise those contracts whose evaluation cannot lead to a `FailedPay` action?

What next?

Implementation

Incorporate the DSL into the Cardano platform, fitting with the Plutus Core langauge.

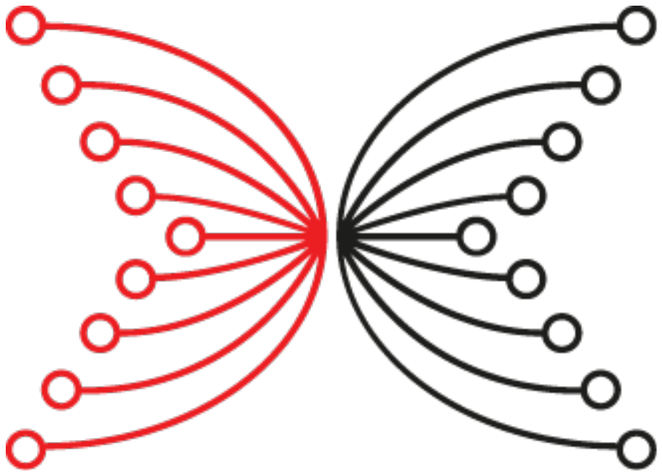
Extension

What needs to be added?
- *"k of n signature"*
- multi-party in general
- multi-step, …

Bigger picture

- client vs chain
- main vs side chains
- proofs of computation?
- monadic, …

This work has been supported by IOHK https://iohk.io



INPUT | OUTPUT

https://github.com/input-output-hk/scdsl

Questions?