Marlowe

Financial Contracts on Blockchain

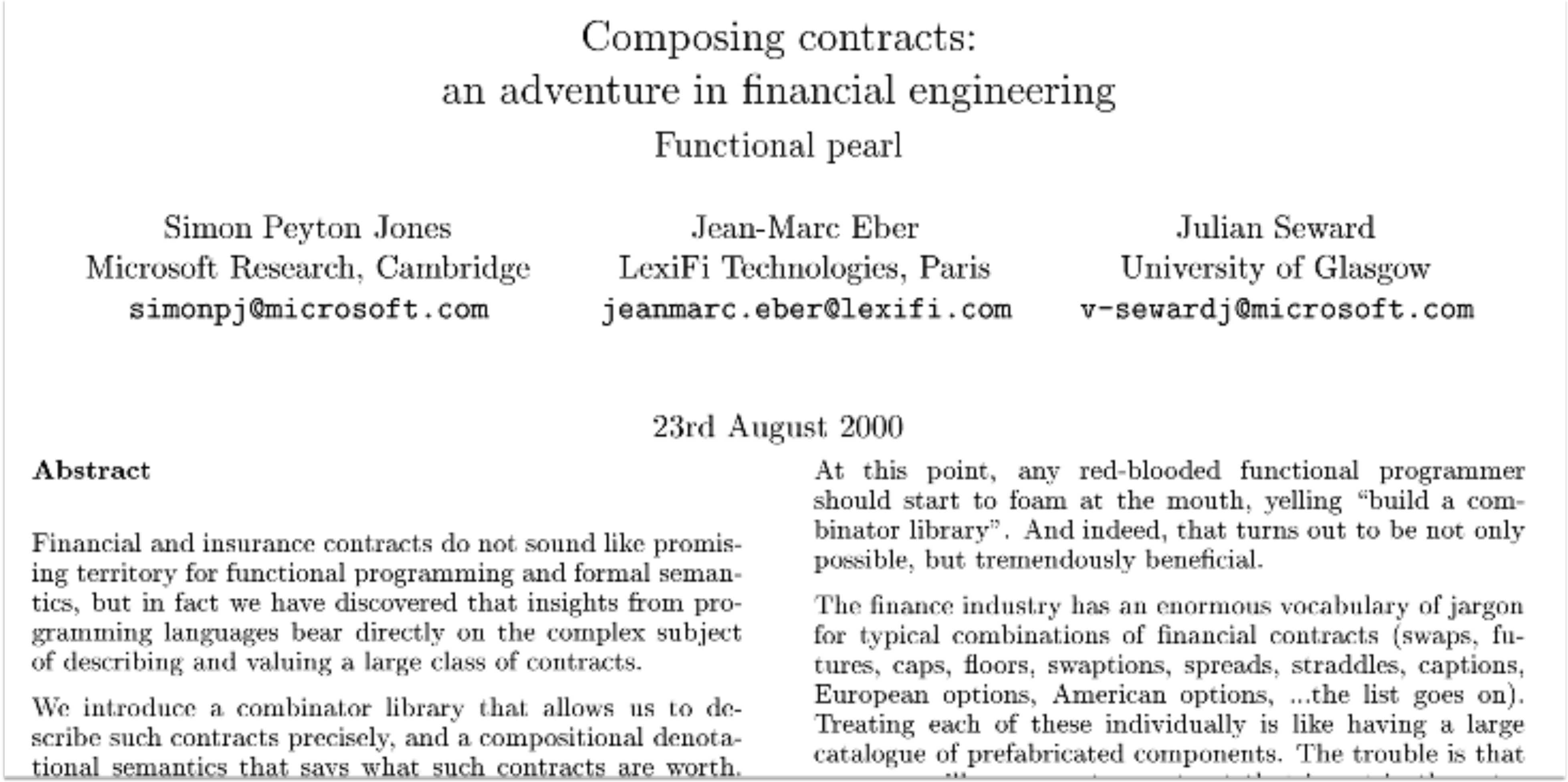
Pablo Lamela Seijas, Simon Thompson

School of Computing, University of Kent, UK

ISoLA 2018

Financial DSLs aren’t new

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

An embedded domain specific language

User-level, not programmer-level.

Some errors made impossible … others less likely.

An EDSL can use host language features … selectively.

It's a language: can transform, analyse, interpret …

Specificity: analysis and proof can do more.

Example

```
(When (Or (two_chose alice bob carol refund)
          (two_chose alice bob carol pay))
      (Choice (two_chose alice bob carol pay)
              (Pay alice bob AvailableMoney)
              redeem_original))
```

Example

```
(When (Or (two_chose alice bob carol refund)
          (two_chose alice bob carol pay))
      (Choice (two_chose alice bob carol pay)
              (Pay alice bob AvailableMoney)
              redeem_original))
```

Onto blockchain

Onto blockchain

Enforcement

The legal system ensures financial contracts …

… but a contract on blockchain should enforce itself.

Onto blockchain

Enforcement

The legal system ensures financial contracts …
… but a contract on blockchain should enforce itself.

Double spend

Blockchain designed to prevent spending the same money twice …
… but that’s precisely how credit works.

Cardano

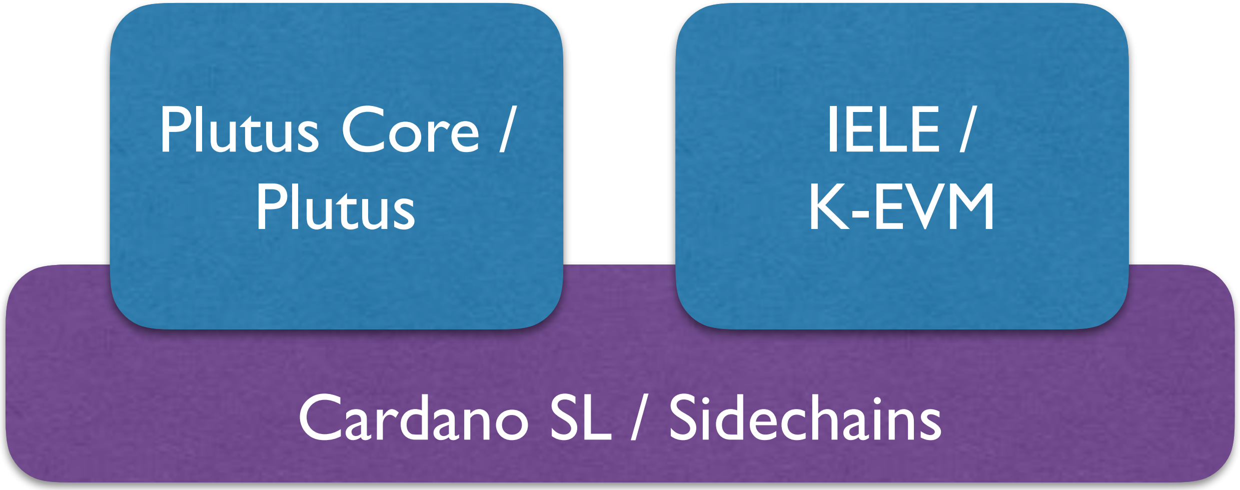
Cardano SL / Sidechains

Cardano

Plutus Core / Plutus

Cardano SL / Sidechains

Cardano

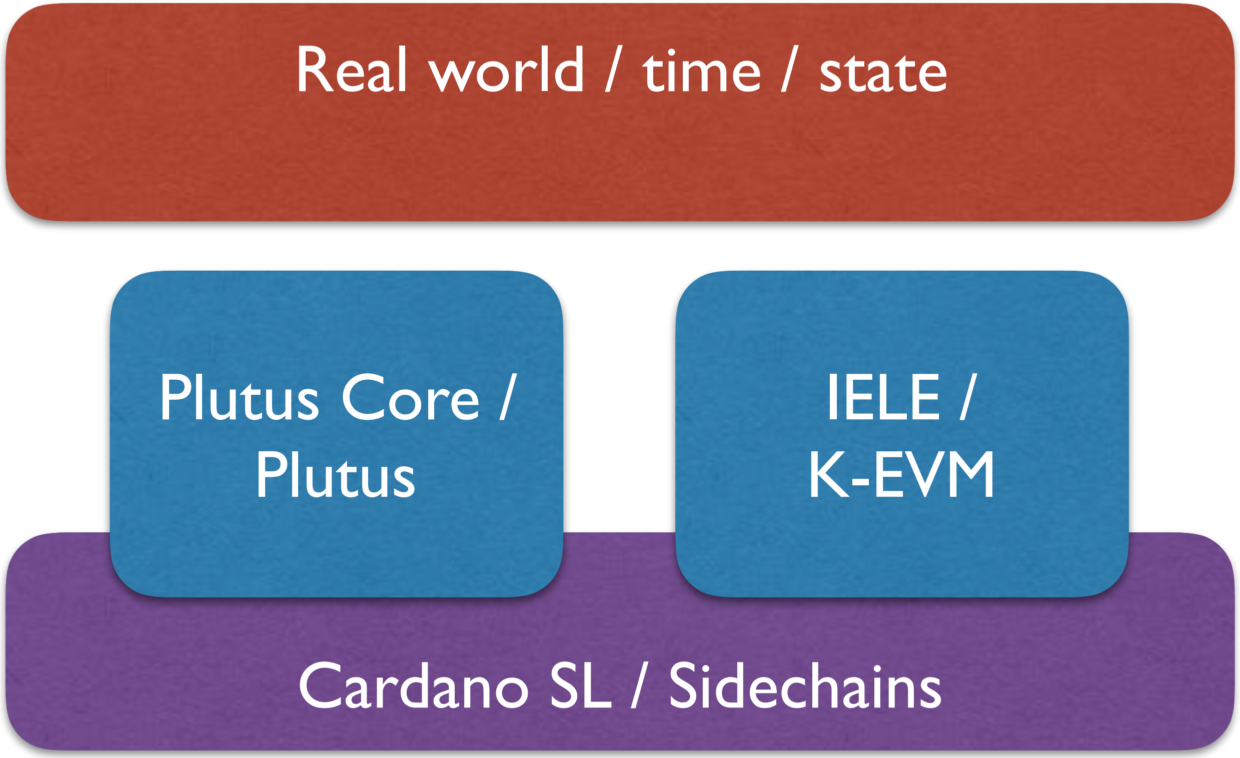


Plutus Core / Plutus

IELE / K-EVM

Cardano SL / Sidechains

Cardano



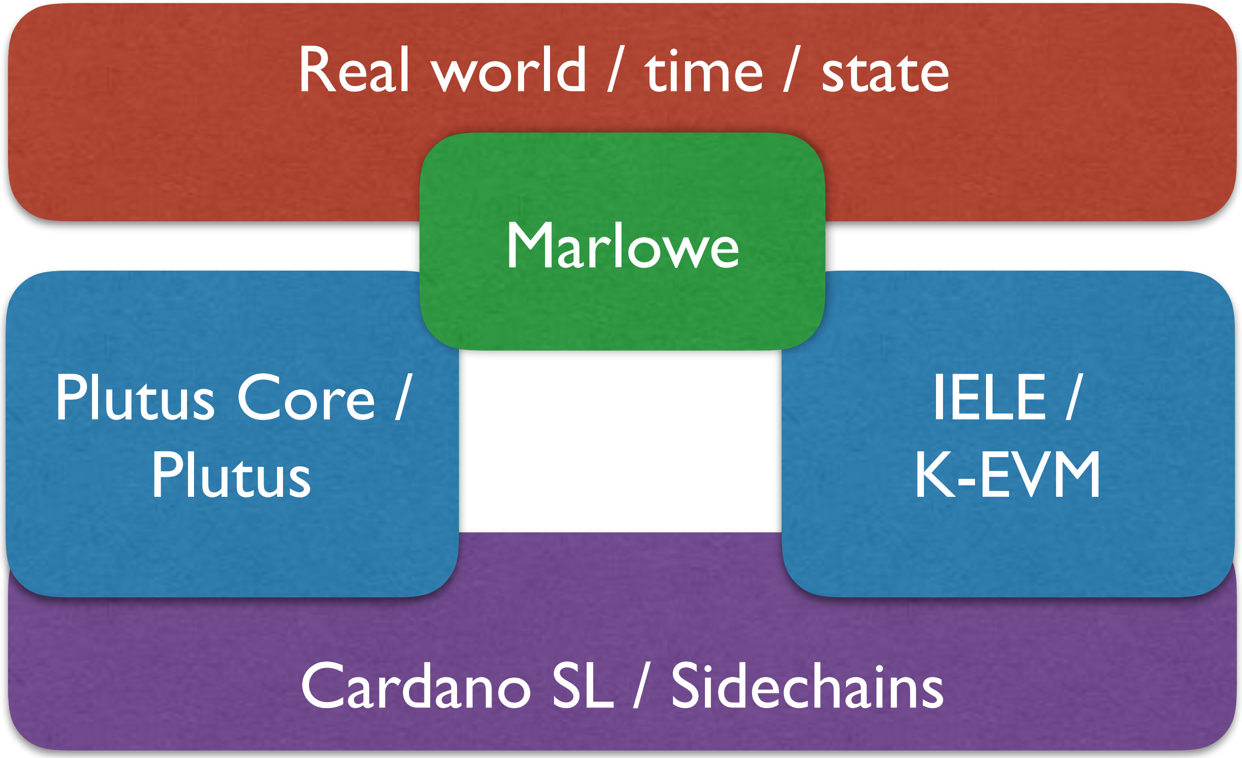
Real world / time / state

Plutus Core / Plutus

IELE / K-EVM

Cardano SL / Sidechains

Cardano



Real world / time / state

Marlowe

Plutus Core / Plutus

IELE / K-EVM

Cardano SL / Sidechains

Crypto-economics

Make the past irrefutable through cryptography.

Shape behaviour through financial incentives.

Avoid bad behaviour … and "walk away".

Marlowe

Why Marlowe?

Understand the implications for smart contract languages …

… for blockchain,

… and for Cardano in particular.

Why Marlowe?

Understand the implications for smart contract languages …

… for blockchain,

… and for Cardano in particular.

It’s a distinct service … and a model for others.

Marlowe

An EDSL as a Haskell `data` type.

Executable small-step semantics.

Analyses and proof.

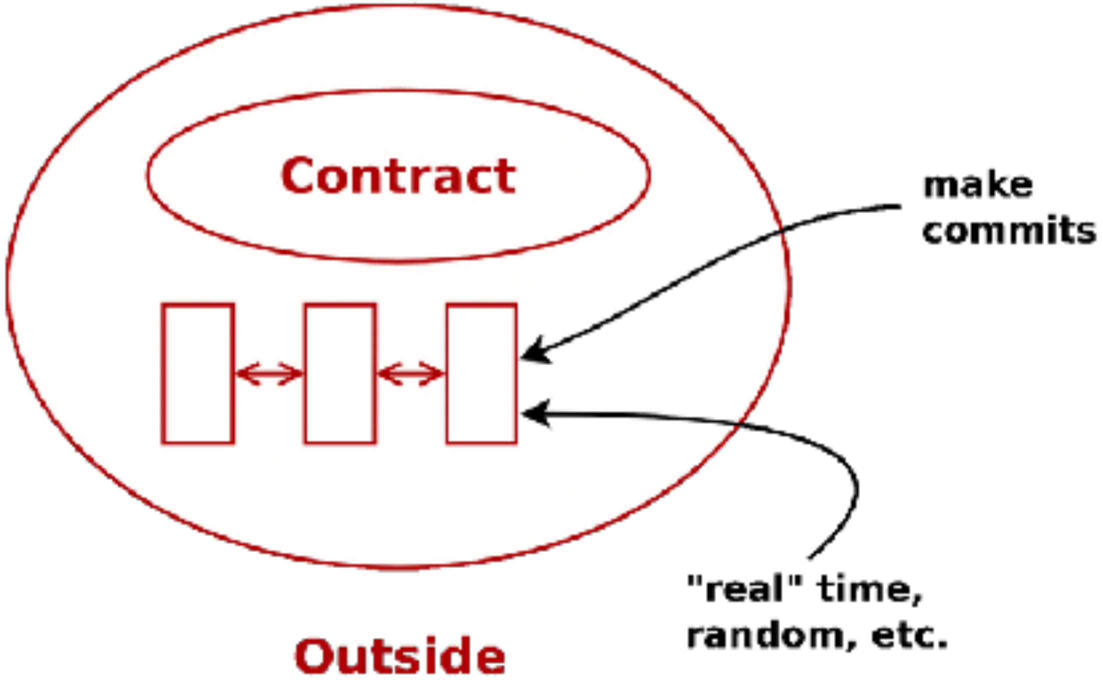
Compile from original DSLs.

Meadow interactive demo.

Interactions with the outside world

Real values: e.g. *"the spot price of oil in Aberdeen at 12:00, 31-12-17"*.

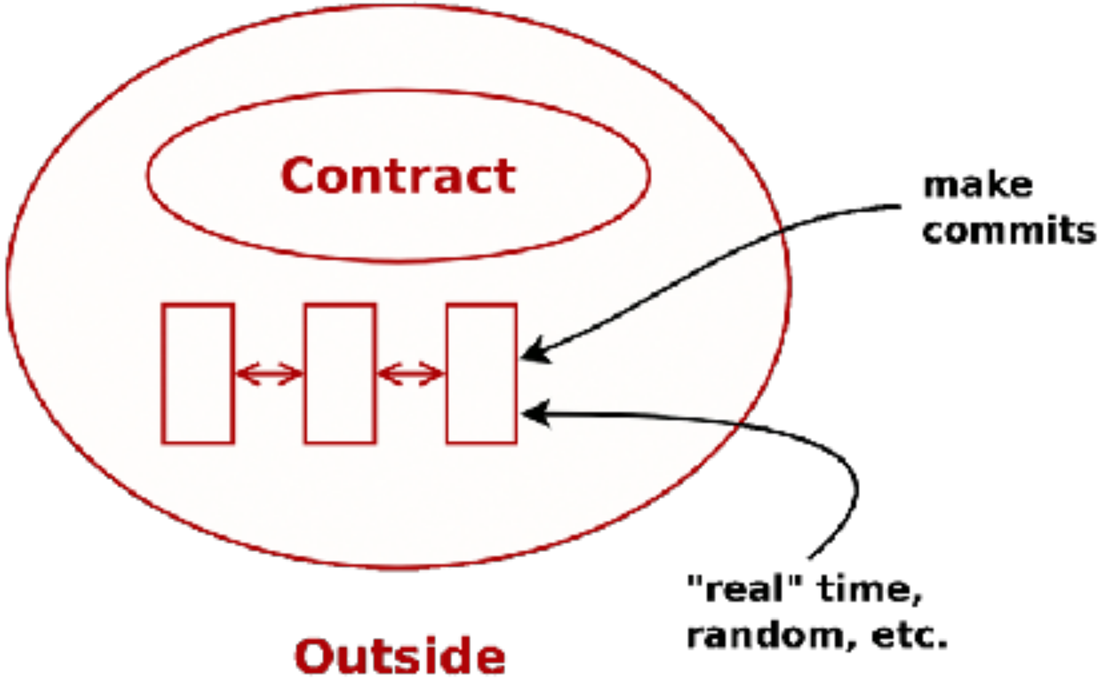
Random values.



Commitments

Commit a certain amount of cash for a finite time.

Need to avoid “walk away” …



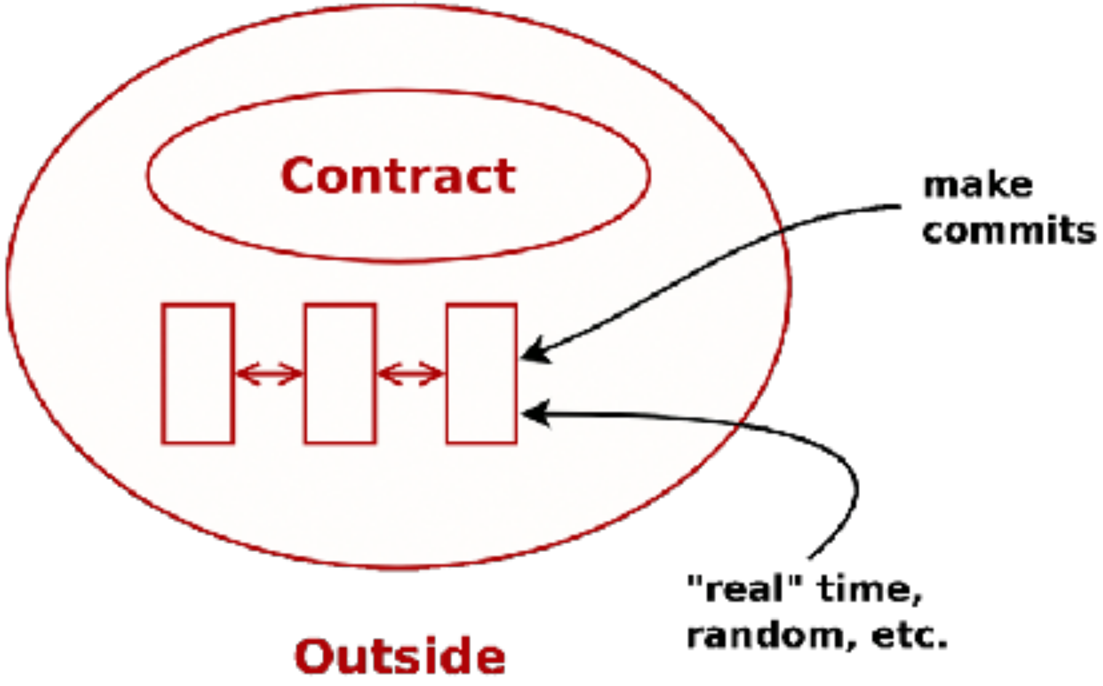
Commitments and Timeouts

Commit a certain amount of cash for a finite time.

Need to avoid "walk away" …

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

… and only wait a bounded time for the commitment to be made.



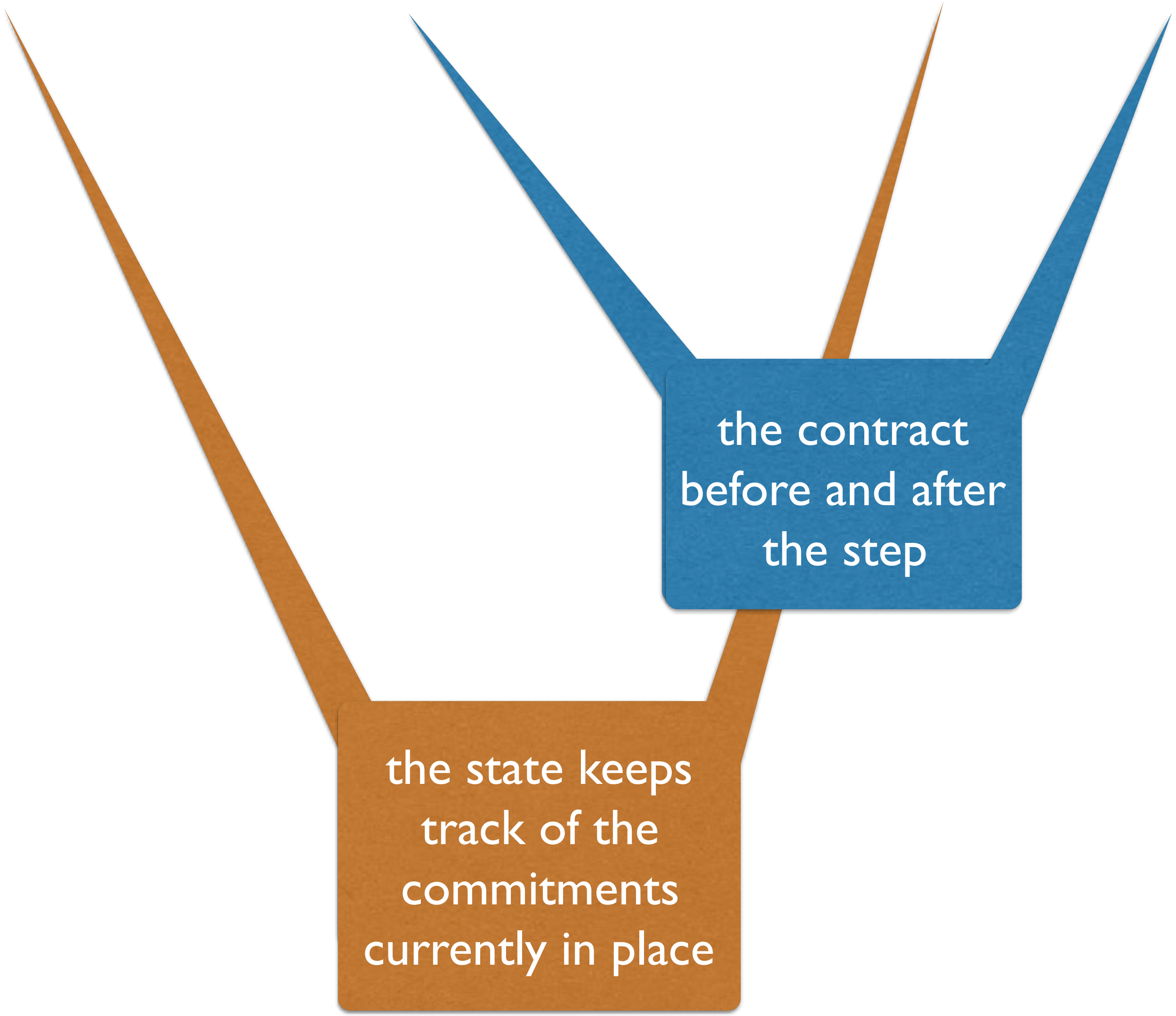
Step and Contract

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

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

the contract before and after the step

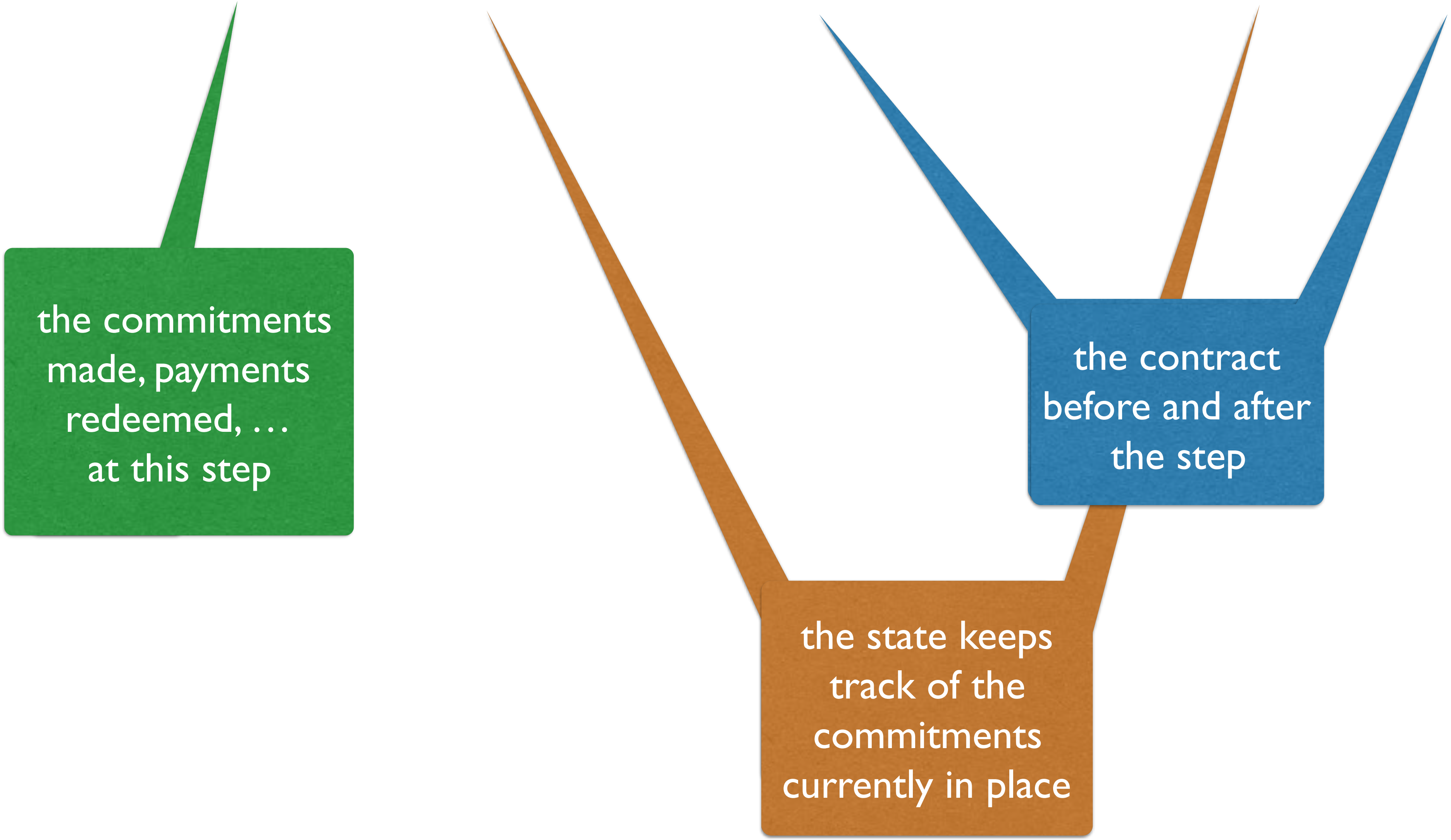
```haskell
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```



the contract before and after the step

the state keeps track of the commitments currently in place

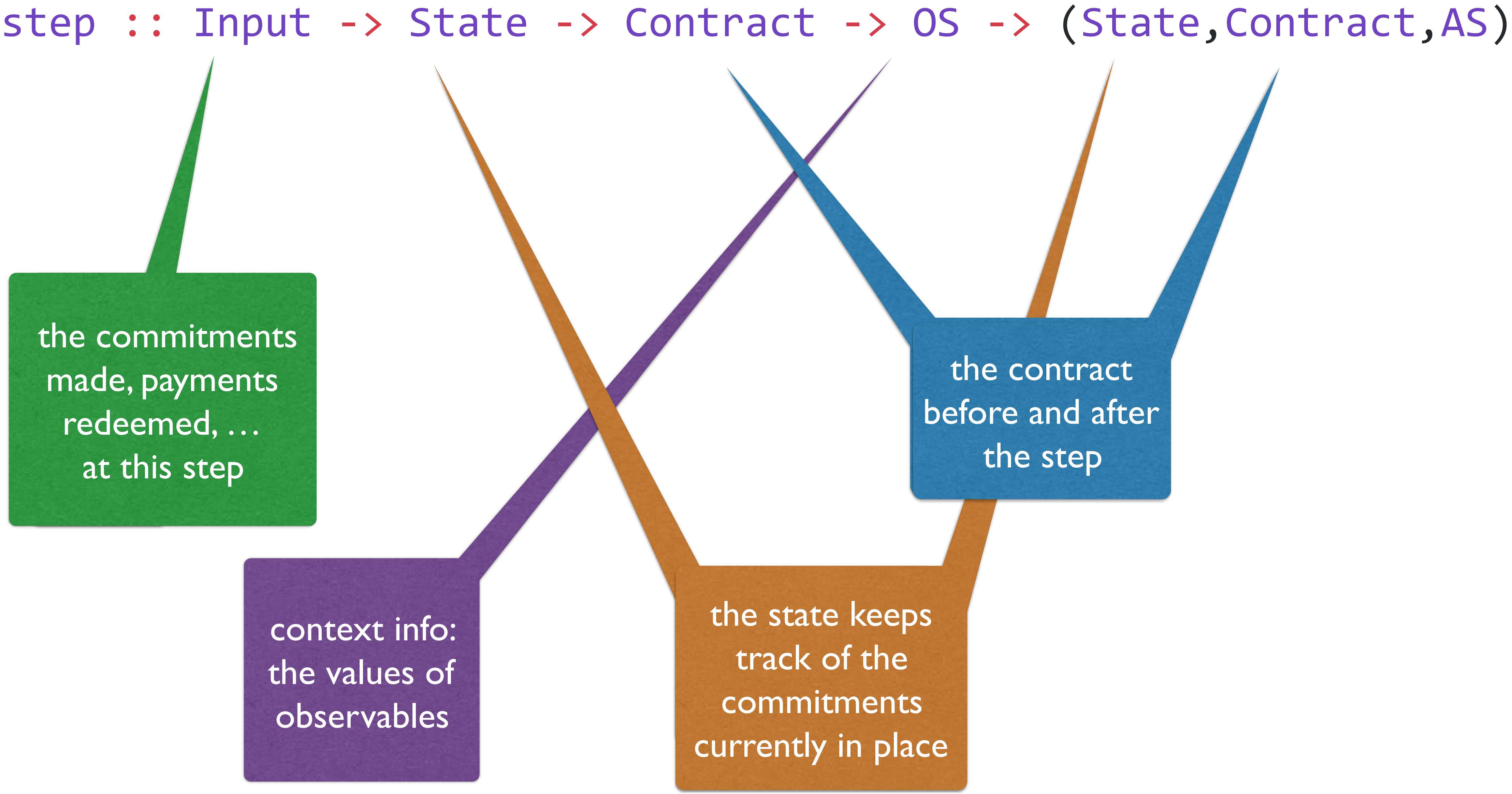
```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```



the commitments made, payments redeemed, … at this step

the contract before and after the step

the state keeps track of the commitments currently in place



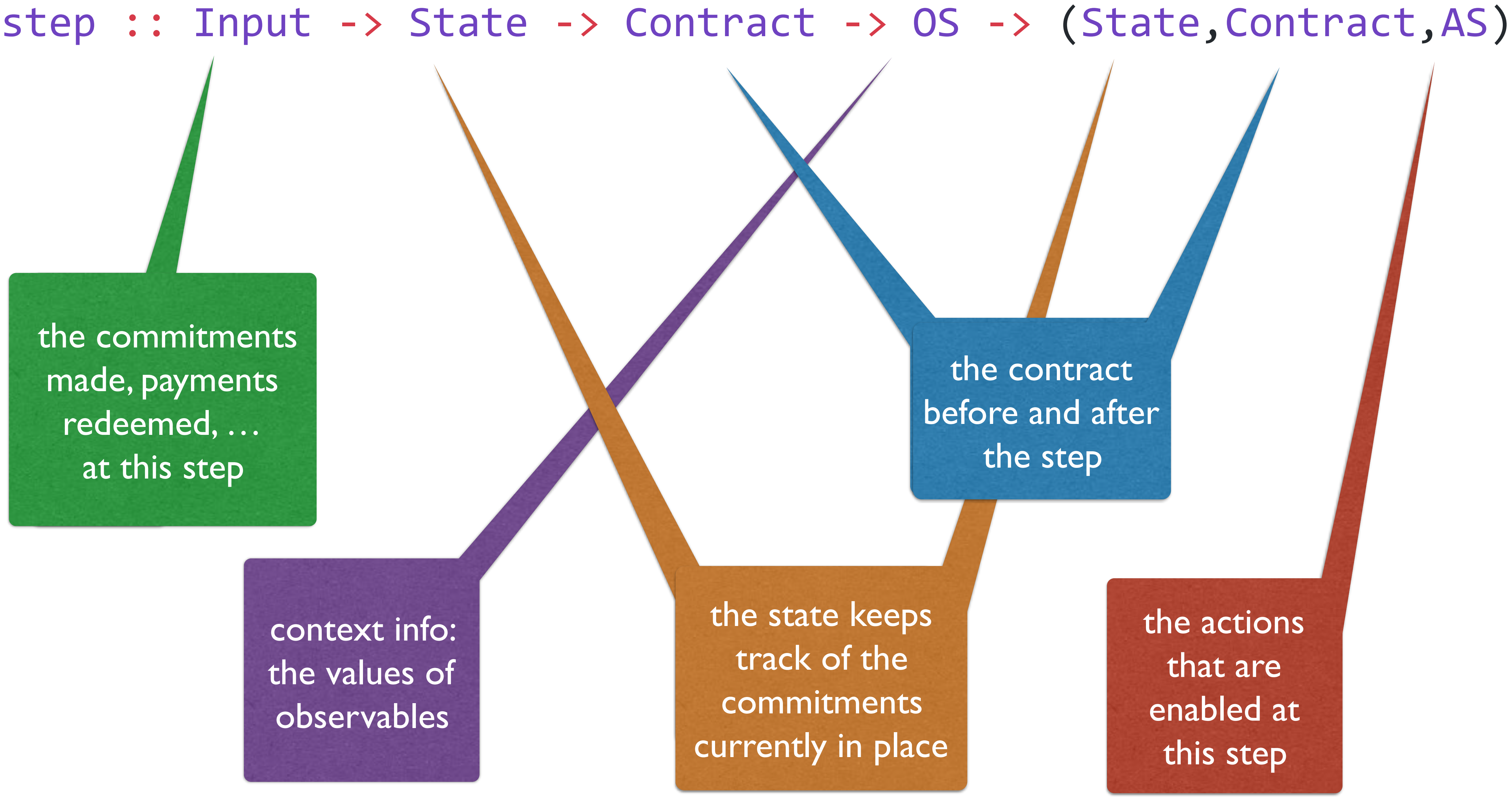
```
step :: Input -> State -> Contract -> OS -> (State,Contract,AS)
```

the commitments made, payments redeemed, … at this step

context info: the values of observables

the state keeps track of the commitments currently in place

the contract before and after the step



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

the commitments made, payments redeemed, … at this step

context info: the values of observables

the state keeps track of the commitments currently in place

the contract before and after the step

the actions that are enabled at this step

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

Observations are recorded to be reused in verification step.

Actions affect on the blockchain: e.g. by transactions being issued.

Step is quiescent if same contract results: it makes progress otherwise.

At each block, run `step` until quiescent.

Redemption: at each block check for expired commitments, etc.

The `Contract` data type

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

The `Contract` data type

```
data Contract =
    Null |
    CommitCash IdentCC Person Money 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 money commitment of `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.

The `Contract` data type

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

`RedeemCC idCC k`

Enables a committer 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.

The `Contract` data type

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

The `Contract` data type

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

The `Contract` data type

```
data Contract =
    Choice Observation Contract Contract |
    When Observation Timeout Contract Contract
                   deriving (Eq,Ord,Show,Read)
```

`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`.

The `Contract` data type

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

Deposit incentive

```
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
```

Deposit incentive

Wait until time 10 for alice to commit 100 ADA until time 200.

```
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
```

Deposit incentive

Wait until time 10 for alice to commit 100 ADA until time 200.

similarly for bob

```
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
```

Deposit incentive

```
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
```

Wait until time `10` for `alice` to commit `100` ADA until time `200`.

similarly for `bob`

if `alice` chooses to before time `100`, both people get their money back

Deposit incentive

Wait until time 10 for alice to commit 100 ADA until time 200.

similarly for bob

```
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))
```

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

otherwise, alice gets her money and the 20 ADA from bob

Deposit incentive

```
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))
```

Wait until time 10 for alice to commit 100 ADA until time 200.

similarly for bob

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

otherwise, alice gets her money and the 20 ADA from bob

action if bob didn't commit in time

Implementation

Implementations

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

Visualise as finitely-branching decision trees.

Embedded DSL

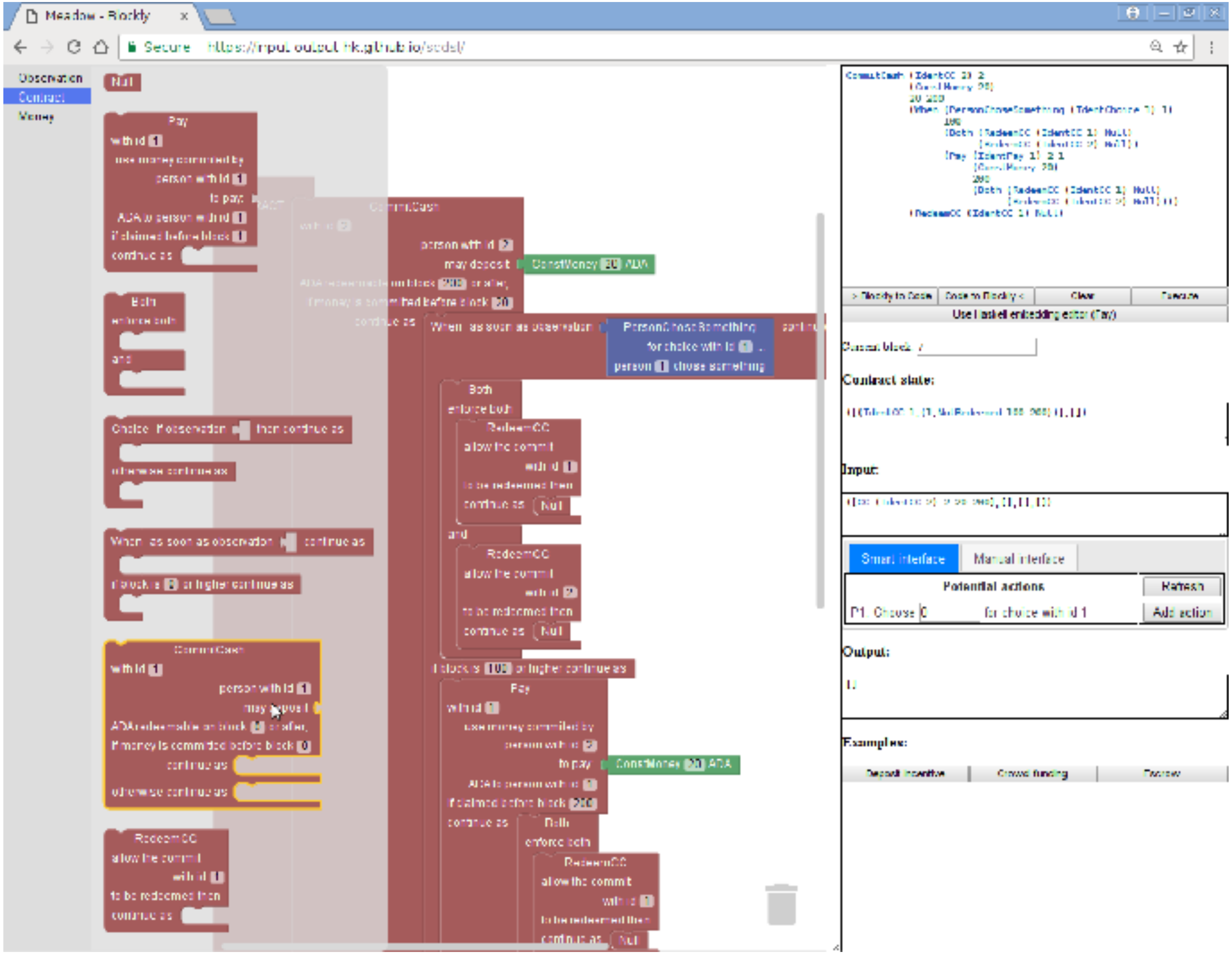
```
(When (Or (two_chose alice bob carol refund)
          (two_chose alice bob carol pay))
      (Choice (two_chose alice bob carol pay)
              (Pay alice bob AvailableMoney)
              redeem_original))
```

Embedded DSL

```
(When (Or (two_chose alice bob carol refund)
          (two_chose alice bob carol pay))
      (Choice (two_chose alice bob carol pay)
              (Pay alice bob AvailableMoney)
              redeem_original))
```

Embedded DSL

```
(When (Or (two_chose alice bob carol refund)
           pay_chosen)
      (Choice pay_chosen
              (Pay alice bob AvailableMoney)
               redeem_original))
              where
              pay_chosen =
                two_chose alice bob carol pay
```



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

Meadow: implementation in Blockly

In-browser implementation: Haskell semantics, compiled into JavaScript.

Two forms of interaction: choose from arbitrary actions …

… or in the smart interface choose from those applicable at each point.

Embedded editor for Marlowe / Haskell scripts.

Blockly is an open source project, which we have adapted.

Analysis

What can we check?

Semantics termination

Step semantics always reduce contract (or are quiescent)

Properties about particular contracts

What can we check?

- Semantics termination
- Step semantics always reduce contract (or are quiescent)
- Properties about particular contracts
  - Is it possible to produce a `FailedPay` action?
  - Is it possible to produce a `DuplicateRedeem` action?
  - Are there redefined identifiers for commits and payments?

`FailedPay` analysis: decidable by ILP / SMT

One symbolic trace per execution path:

Symbolic trace ⇨

Concrete trace ⇨

Result

If symbolic traces for all execution paths are either:

- unsolvable, or
- do not produce `FailedPay`

then it is impossible for a `FailedPay` to occur.

Symbolic traces

Global variables. For example:

- When is this commit issued? Call it X
- What is the value of this choice? Call it Y

Variables are constrained by logical combinations of integer inequalities.

Cover all possible paths, but …

… may cover impossible paths, which are discarded through execution.

Work in progress

Revising the language and implementation

```
data Contract =
    ...
    Observation !Timeout !Contract !Contract |
    Scale !Value !Value !Value !Contract |
    Let !IdentLet !Contract !Contract |
    Use !IdentLet
```

Coq formalisation of semantics

Translate Haskell semantics of Marlowe to Coq.

Extract Haskell from Coq … and QuickCheck the two equivalent.

Properties about contracts in general

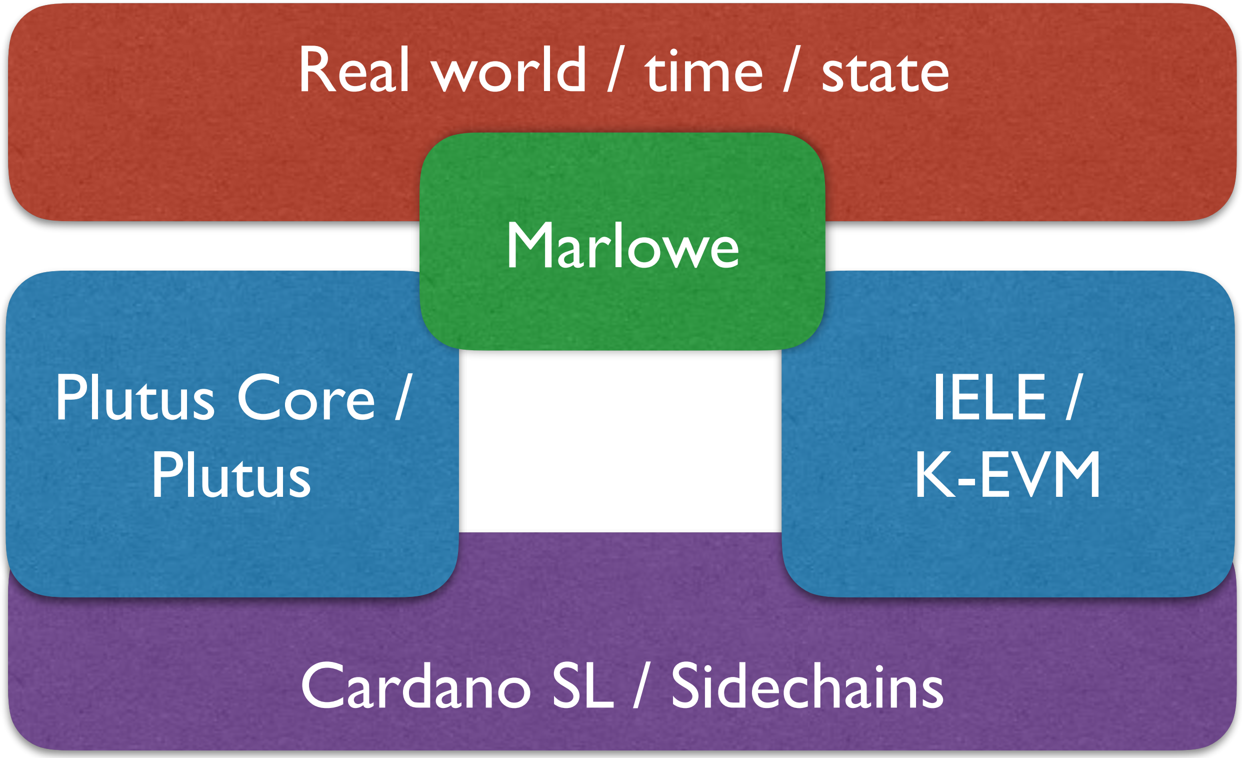
- Can contracts of this form produce a `FailedPay` action?
- Can contracts of this form produce a `DuplicateRedeem` action?
- …

Compile from original DSL to Marlowe

Estimate commitments to ensure no failed payments.

Include (default) commitments.

Integrate with the Cardano SL



Push *vs* pull model

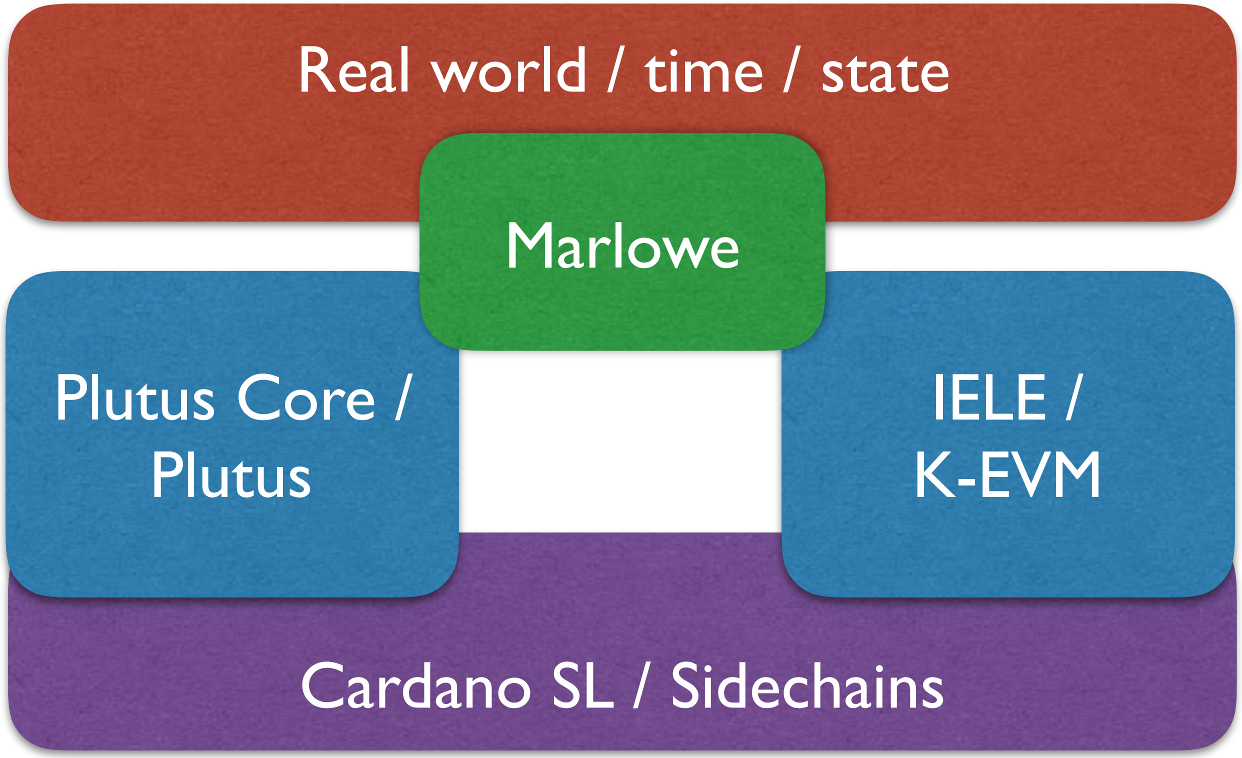
UTxO *vs* accounts

Redeemer/validator model

Observations

Wallet/IDE

But first integrate with the “mockchain”



Real world / time / state

Marlowe

Plutus Core / Plutus

IELE / K-EVM

Cardano SL / Sidechains

Push *vs* pull model

UTxO *vs* accounts

Redeemer/validator model

Observations

Wallet/IDE

Marlowe

An EDSL as a Haskell `data` type.

Executable small-step semantics.

Analyses and proof.

Compile from original DSLs.

Meadow interactive demo.

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

Marlowe and UTxO