Contracts as Automata

From Specification to Implementation

Manuel M T Chakravarty
IOG


mchakravarty


TacticalGrace


justtesting.org

Ledger Models

Coins versus accounts

What is the biggest conceptual difference between Ethereum contracts (account model) and Cardano contracts (eUTXO)?

What is the biggest conceptual difference between Ethereum contracts (account model) and Cardano contracts (eUTXO)?

Ethereum contracts compute.
Cardano contracts verify.

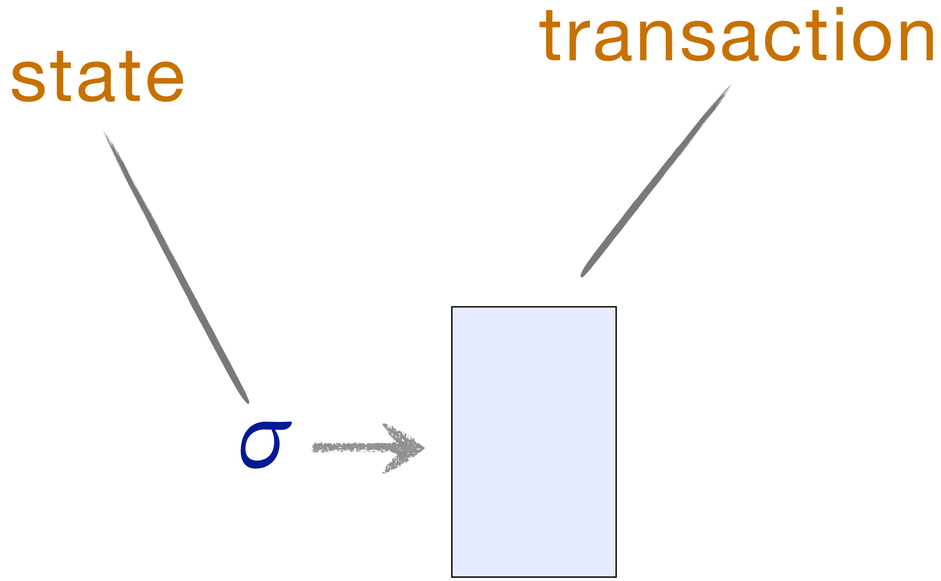
σ

eUTXO

Account

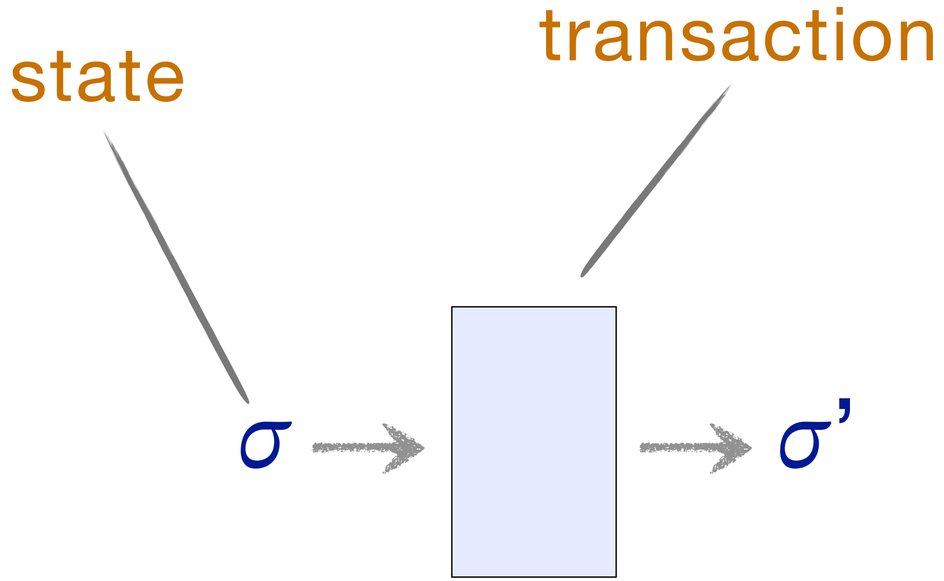
eUTXO

Account



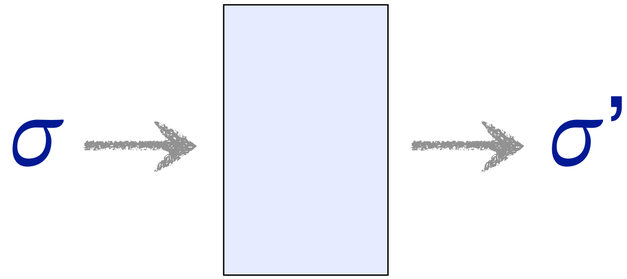
eUTXO

Account



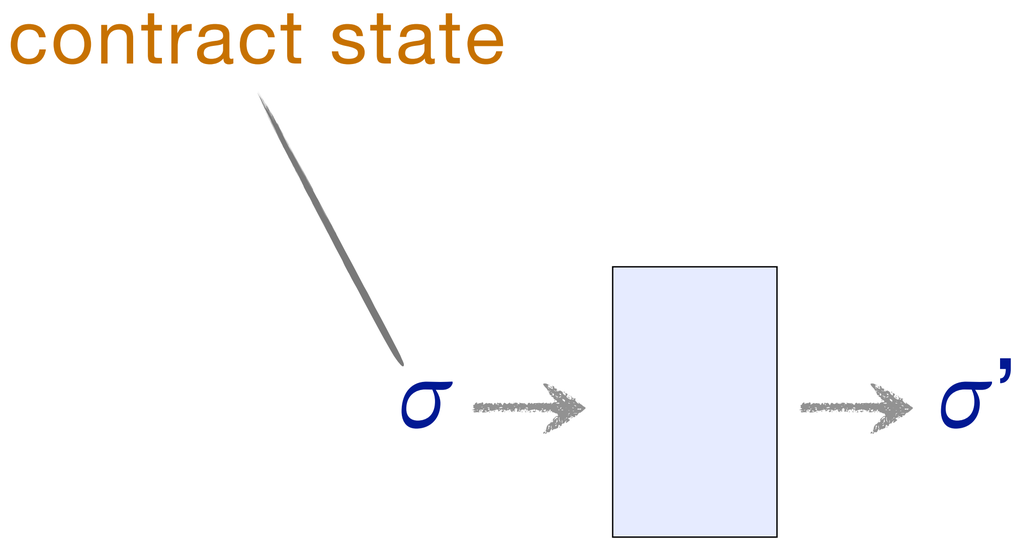
eUTXO

Account



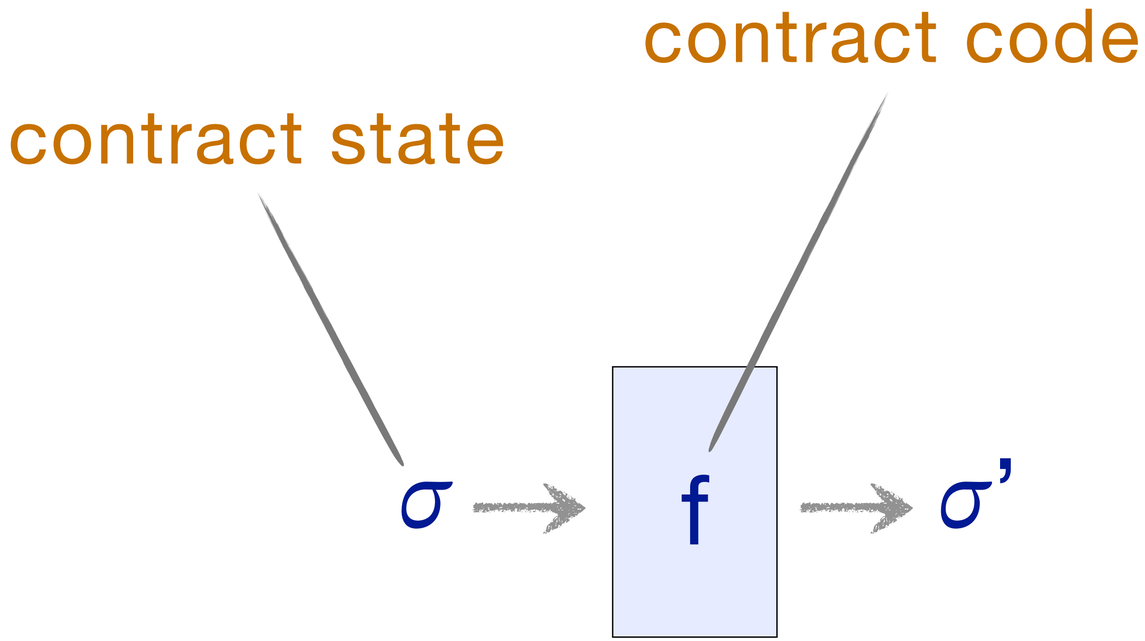
eUTXO

Account



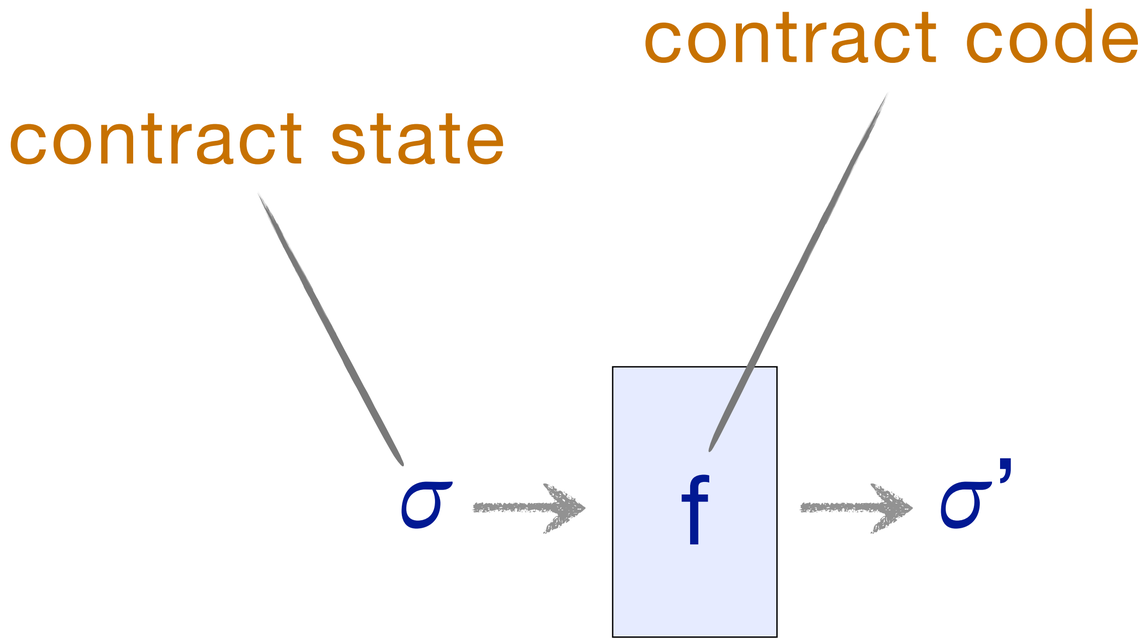
eUTXO

Account



eUTXO

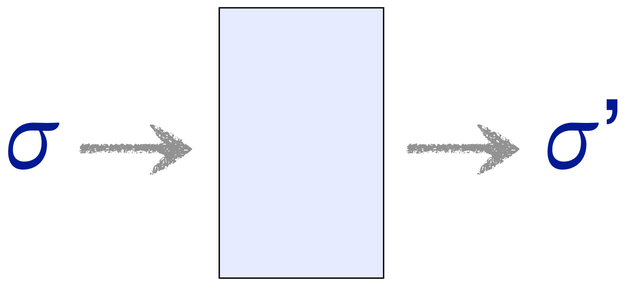
Account

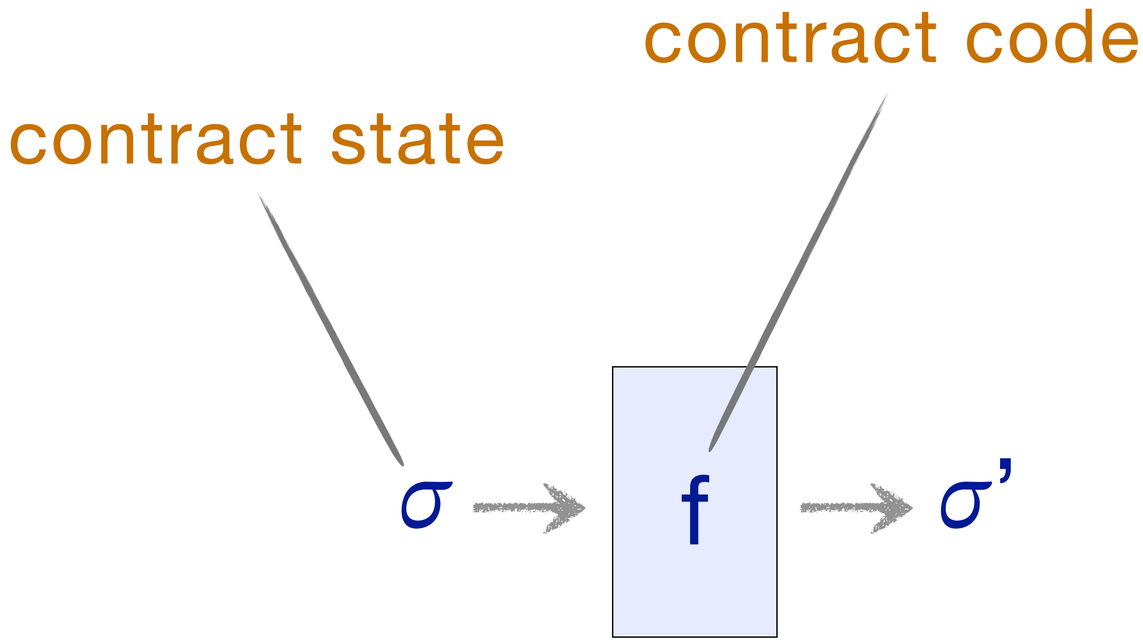


$f(\sigma) = \sigma'$

eUTXO

Account

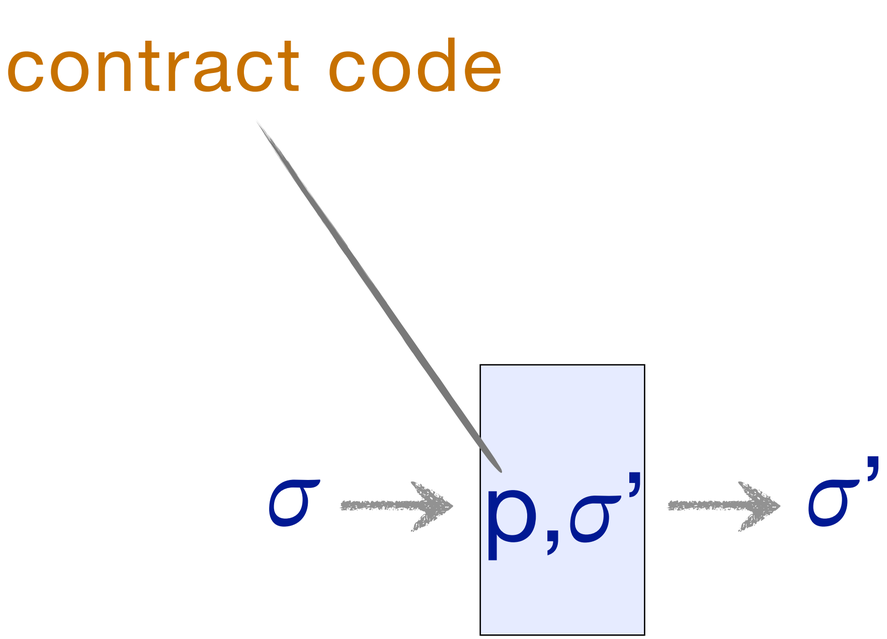




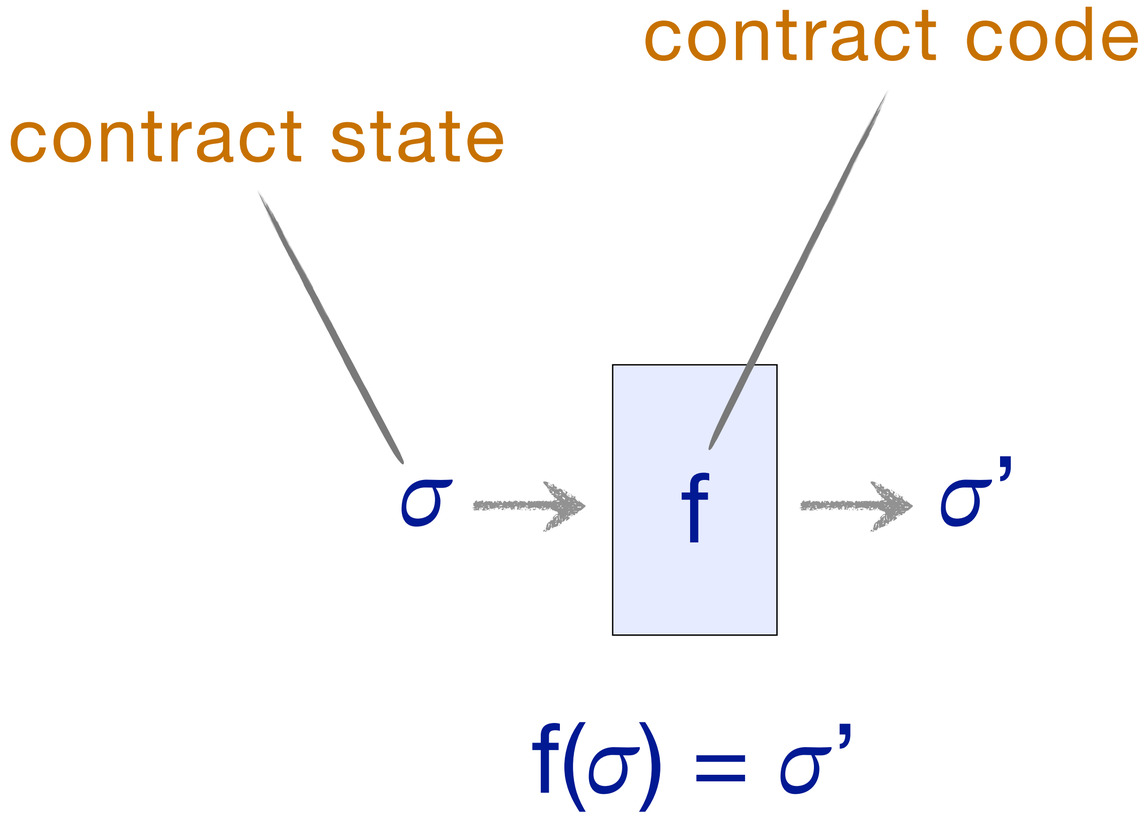
$f(\sigma) = \sigma'$

eUTXO

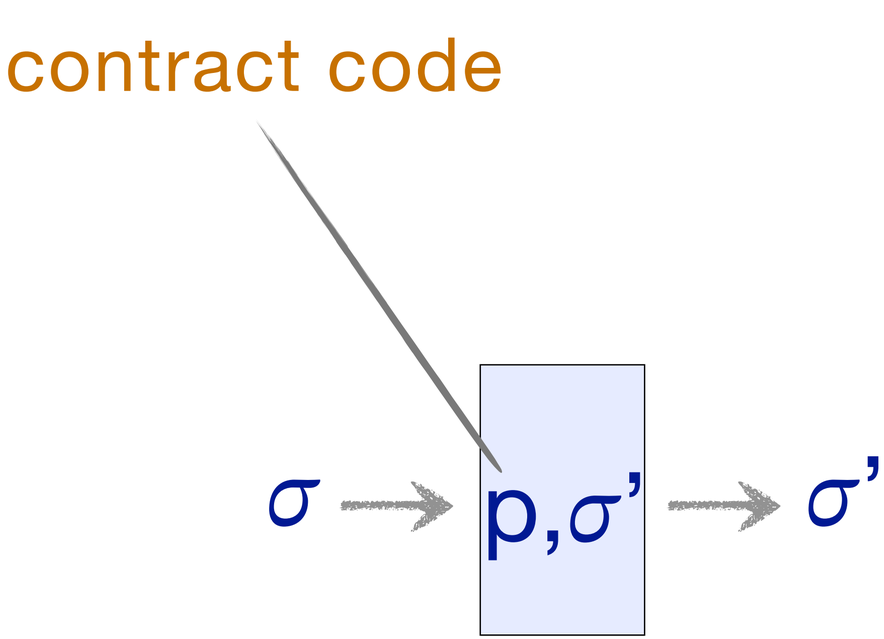
Account



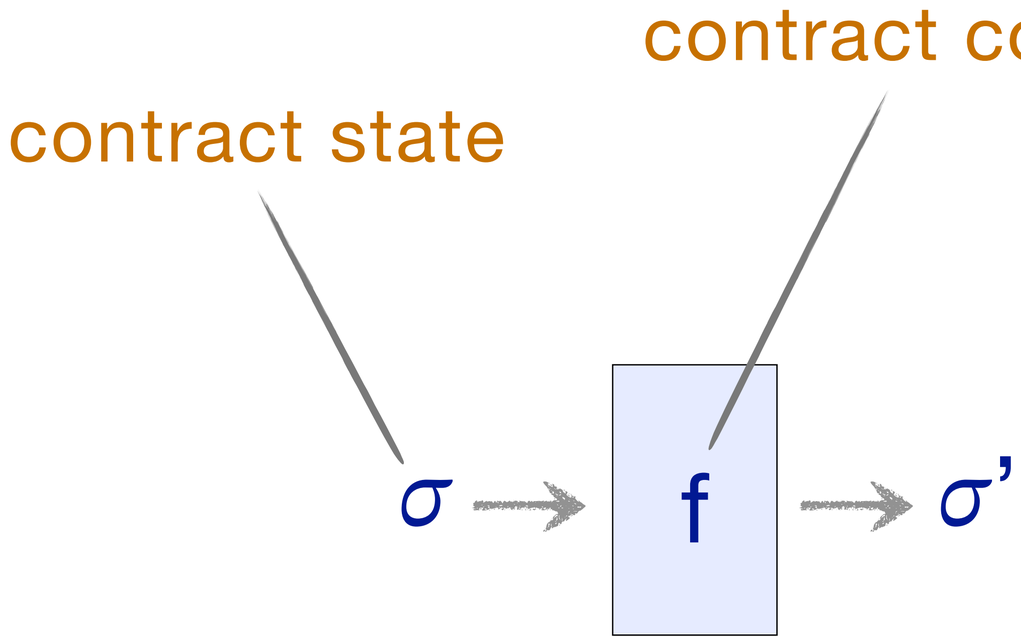
eUTXO



Account



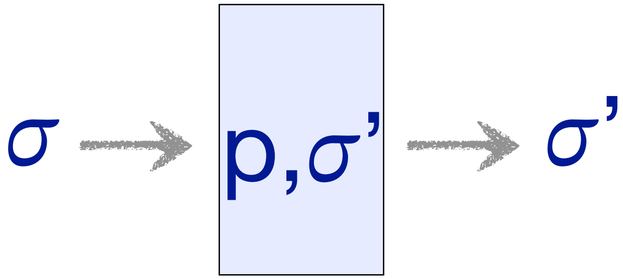
p(σ, σ’) = true



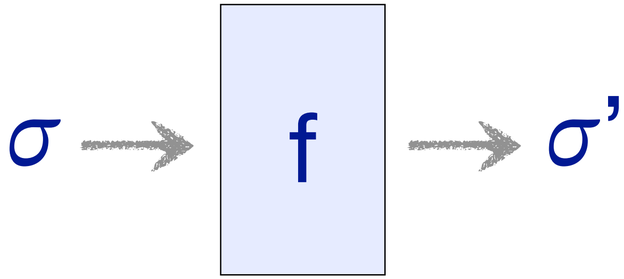
f(σ) = σ’

eUTXO

Account



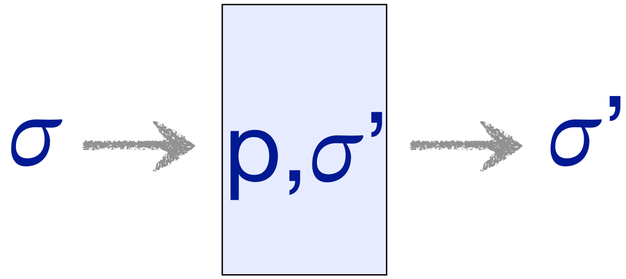
p(σ, σ’) = true



f(σ) = σ’

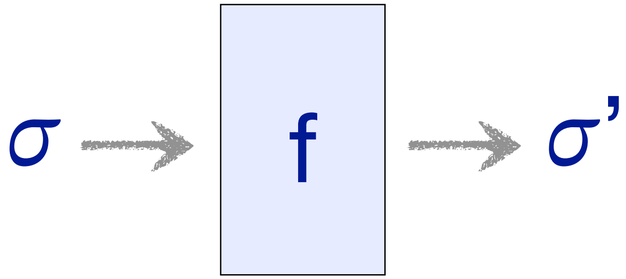
eUTXO

Account



$p(\sigma, \sigma') = \text{true}$

verification

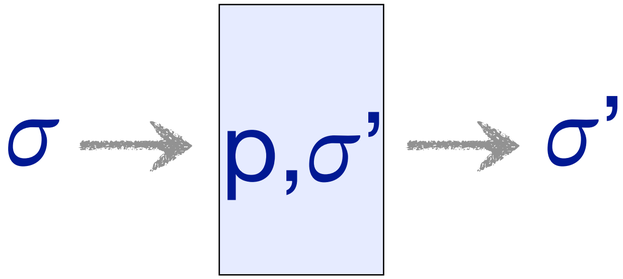


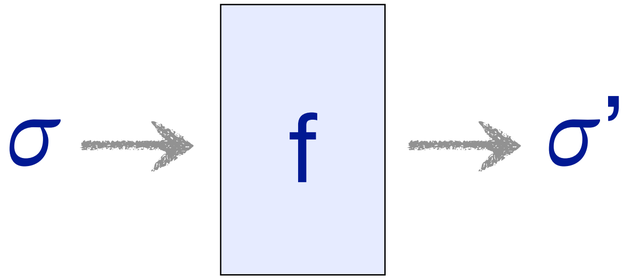
$f(\sigma) = \sigma'$

computation

eUTXO

Account





$p(\sigma, \sigma') = true$

$f(\sigma) = \sigma'$

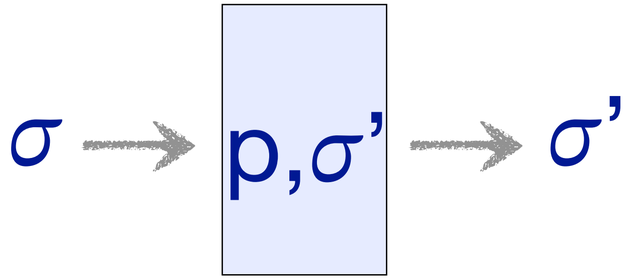
verification

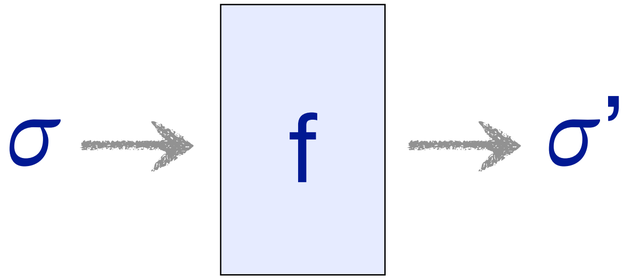
computation

determinism

eUTXO

Account





$p(\sigma, \sigma') = true$

$f(\sigma) = \sigma'$

verification

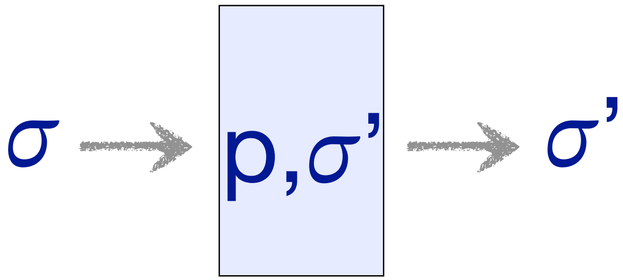
computation

determinism

computation happens off-chain

eUTXO

Account



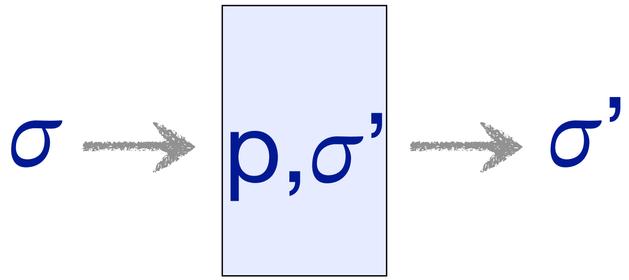
p(σ, σ’) = true

verification

determinism

computation happens off-chain

eUTXO



Needs a programming model

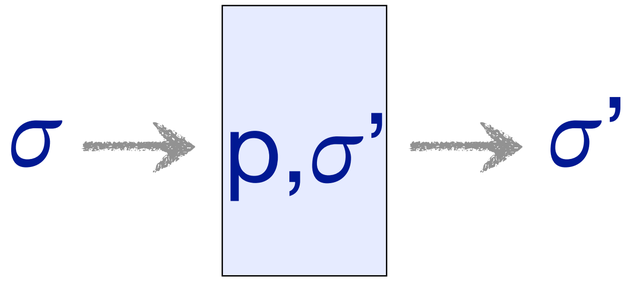
p(σ, σ’) = true

verification

determinism

computation happens off-chain

eUTXO



Needs a programming model

integrate on-chain and off-chain

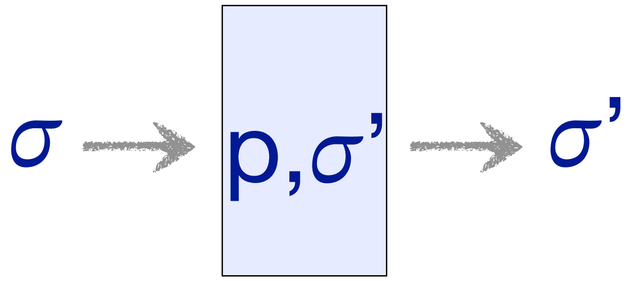
p(σ, σ’) = true

verification

determinism

computation happens off-chain

eUTXO



$p(\sigma, \sigma') = true$

Needs a programming model

integrate on-chain and off-chain

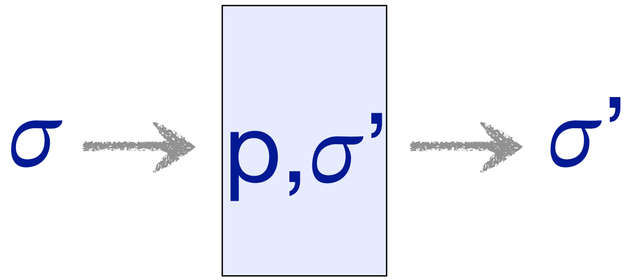
hide implementation details

verification

determinism

computation happens off-chain

eUTXO



$p(\sigma, \sigma') = true$

Needs a programming model

integrate on-chain and off-chain

hide implementation details

formal reasoning

verification

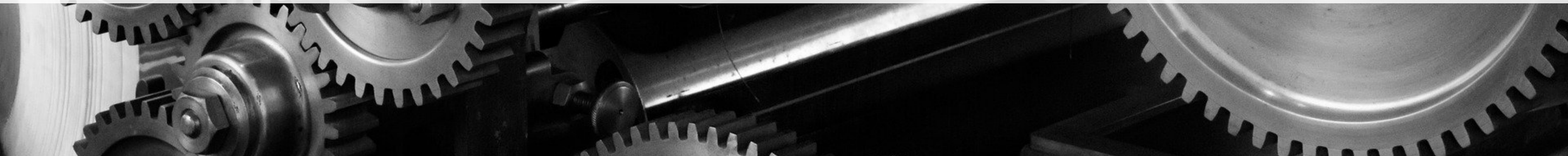
determinism

computation happens off-chain

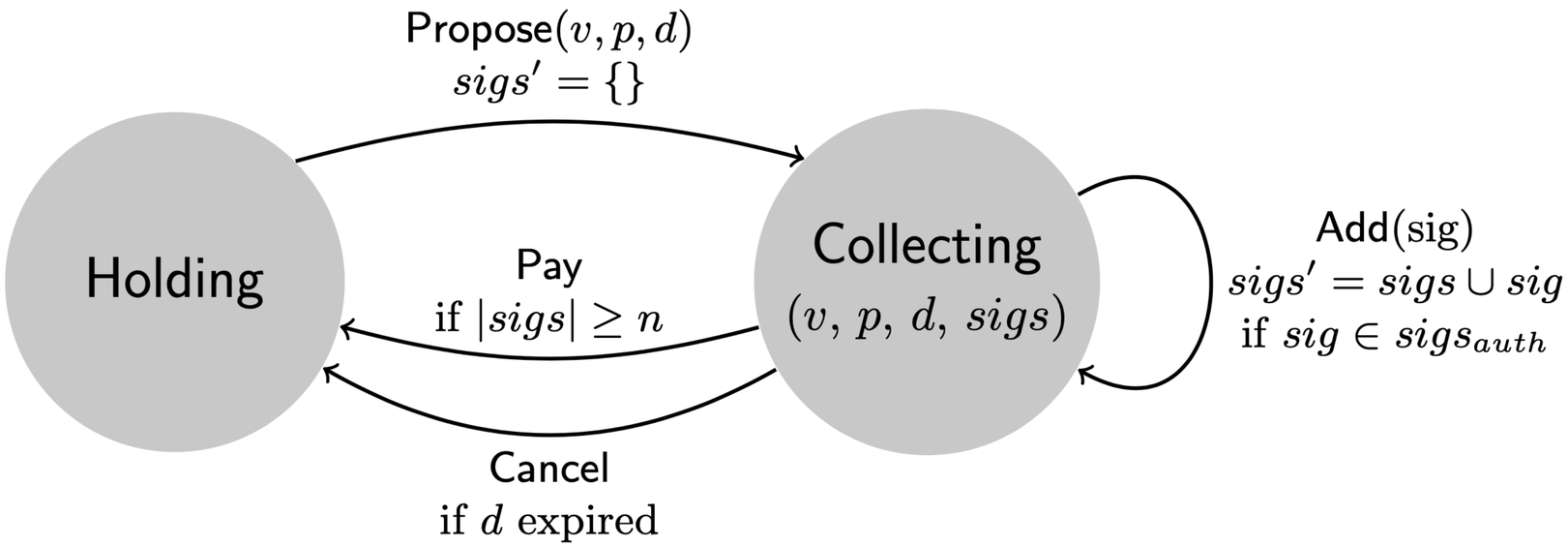
eUTXO

State Machines

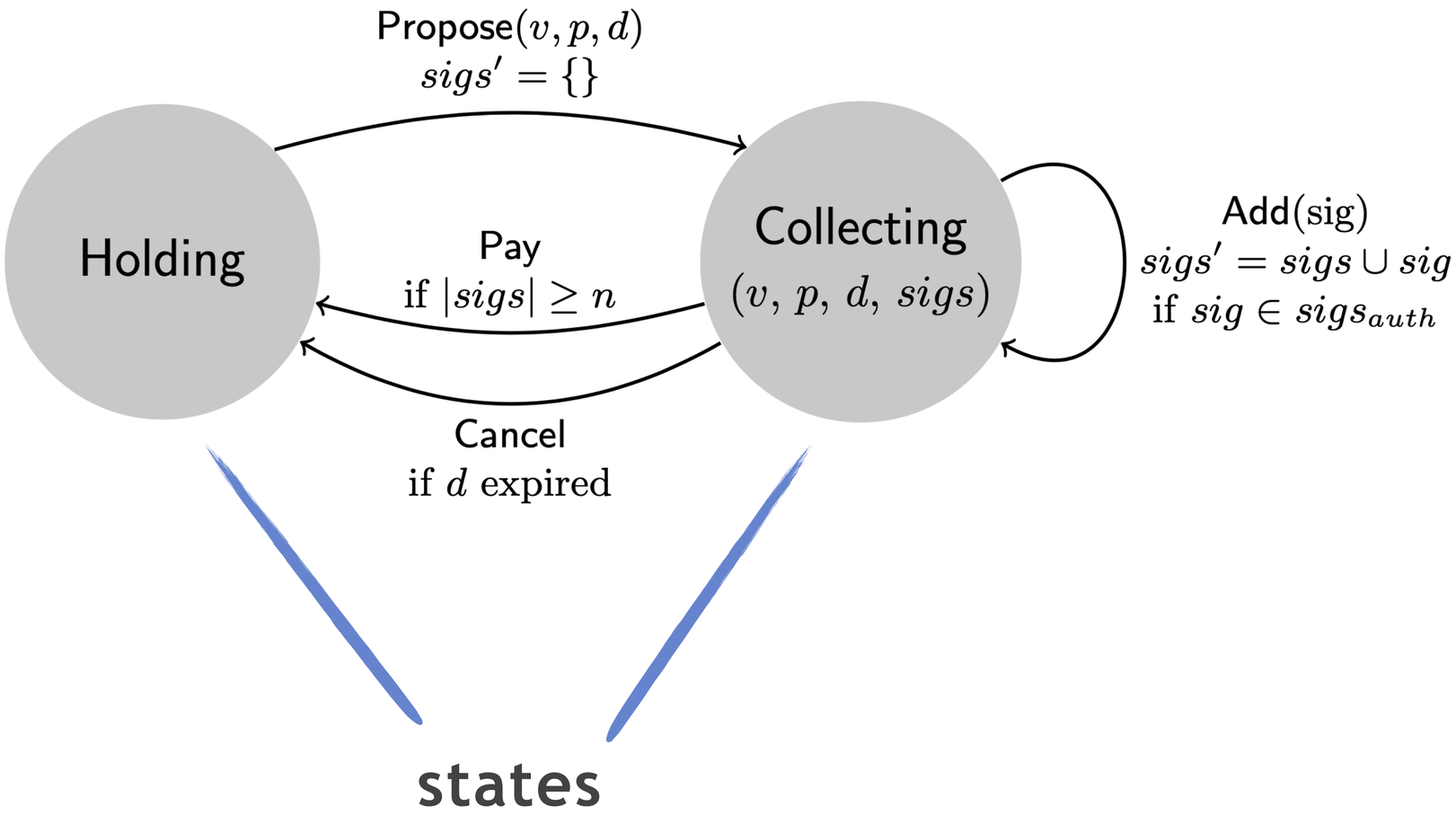
Logical model for contracts



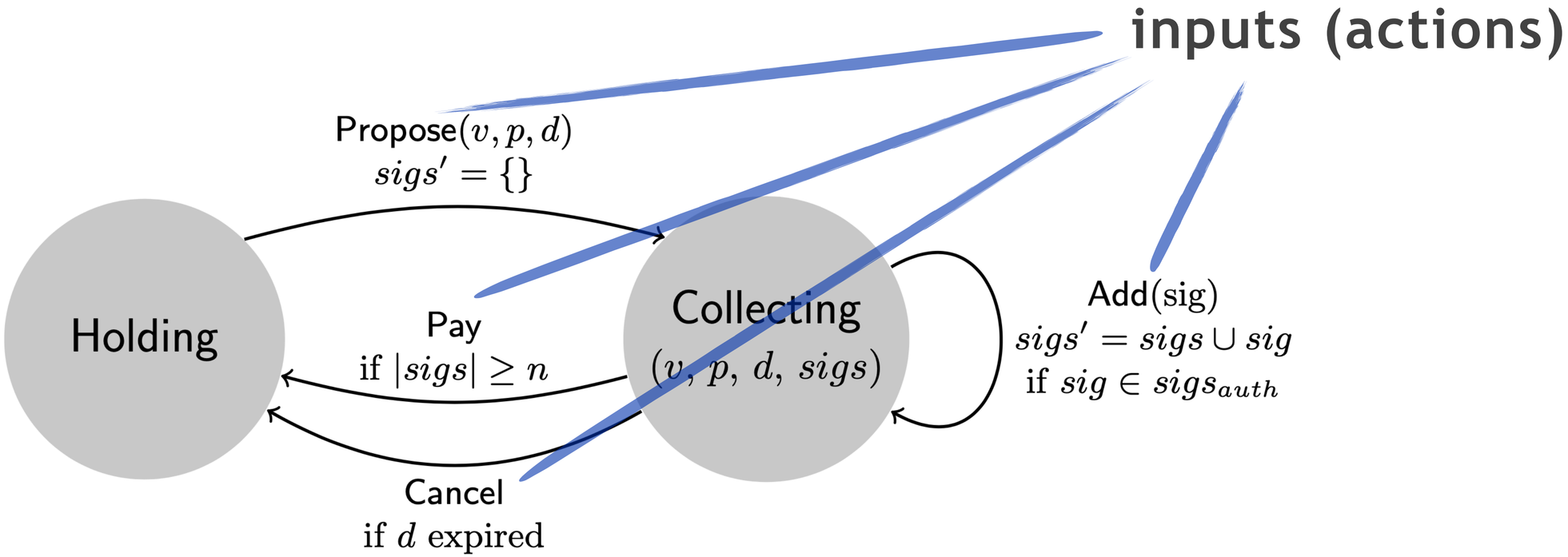
Constraint Emitting Machines (CEMs)



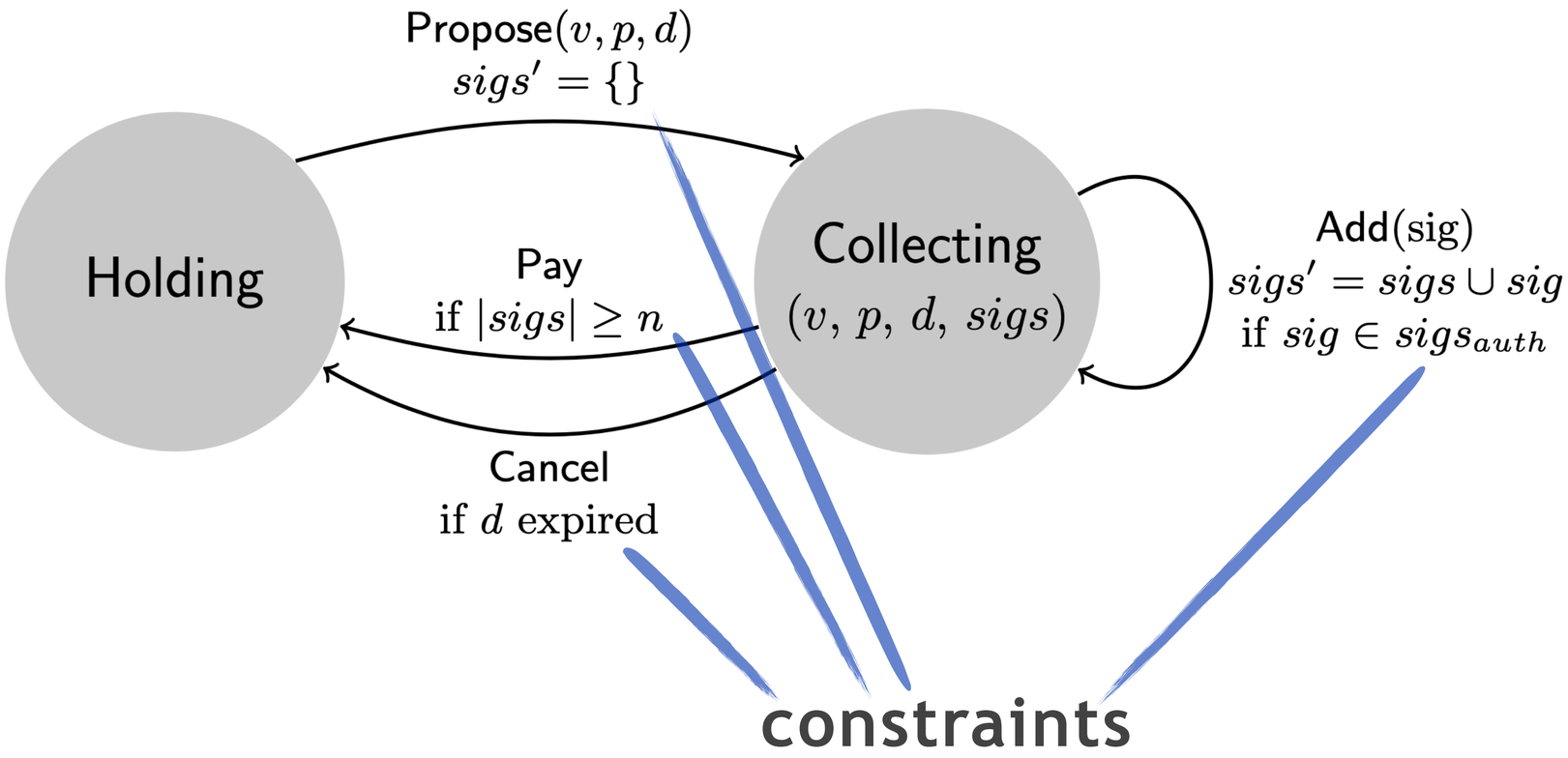
Constraint Emitting Machines (CEMs)

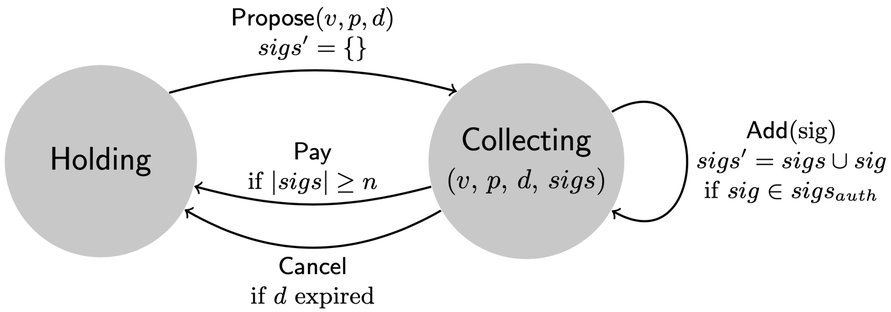


Constraint Emitting Machines

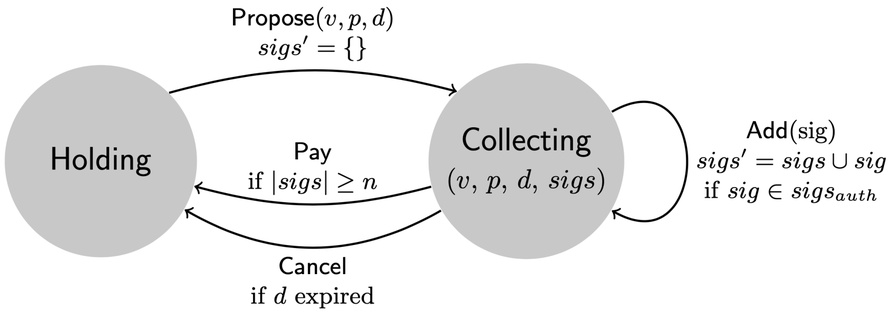


Constraint Emitting Machines



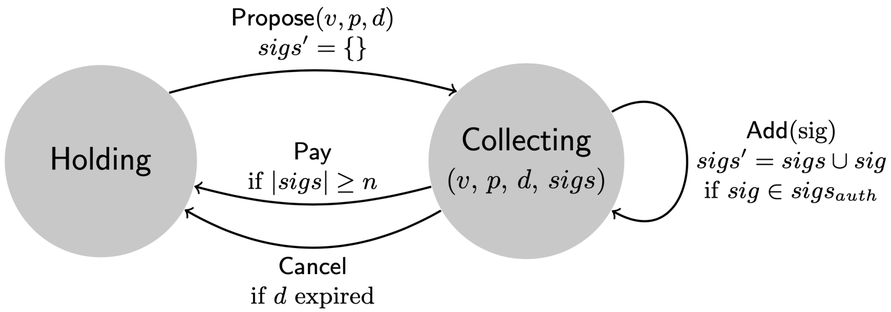


Plutus state machine library is too inefficient



Plutus state machine library is too inefficient

CEMs not captured by standard automata theory



Plutus state machine library is too inefficient

CEMs not captured by standard automata theory

Constraints serve as guards, but treated like outputs

Automata Contracts

Specification & implementation

Deterministic Finite Automata

Deterministic Finite Automata

base states and transition labels

Deterministic Finite Automata

contract state and actions

augmenting DFA states and labels

base states and transition labels

Deterministic Finite Automata

contract state and actions

augmenting DFA states and labels

specification

base states and transition labels

Deterministic Finite Automata

contract state and actions

augmenting DFA states and labels

specification

relational

base states and transition labels

Deterministic Finite Automata

contract state and actions

augmenting DFA states and labels

specification

relational

whole system view

base states and transition labels

Deterministic Finite Automata

contract state and actions

augmenting DFA states and labels

specification

relational

whole system view

refinement

implementation

base states and transition labels

Deterministic Finite Automata

contract state and actions

augmenting DFA states and labels

specification

relational

whole system view

refinement

implementation

functional

base states and transition labels

Deterministic Finite Automata

contract state and actions

augmenting DFA states and labels

specification relational whole system view

refinement

implementation functional on-chain versus off-chain

Do we need a new specification language and framework?

Do we need a new specification language and framework?

Maybe we can get started with Agda

Do we need a new specification language and framework?

Maybe we can get started with Agda

reasoning about the specification

Do we need a new specification language and framework?

Maybe we can get started with Agda

reasoning about the specification

correctness of refinement

Do we need a new specification language and framework?

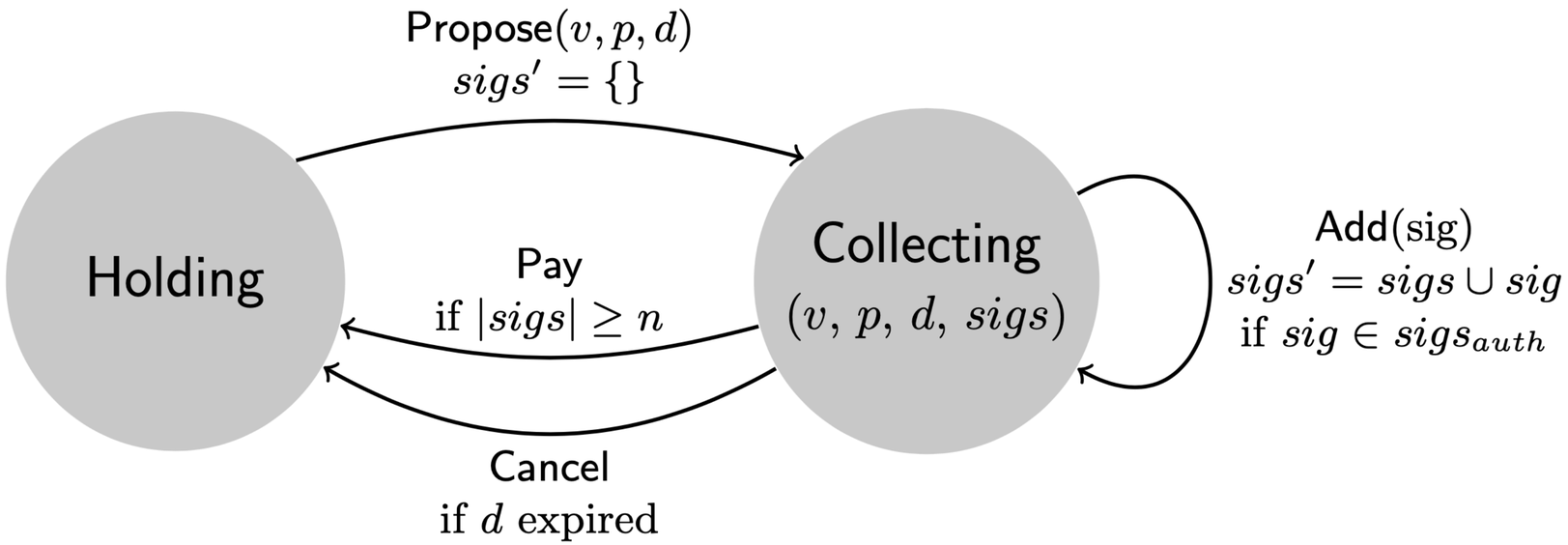
Maybe we can get started with Agda

reasoning about the specification

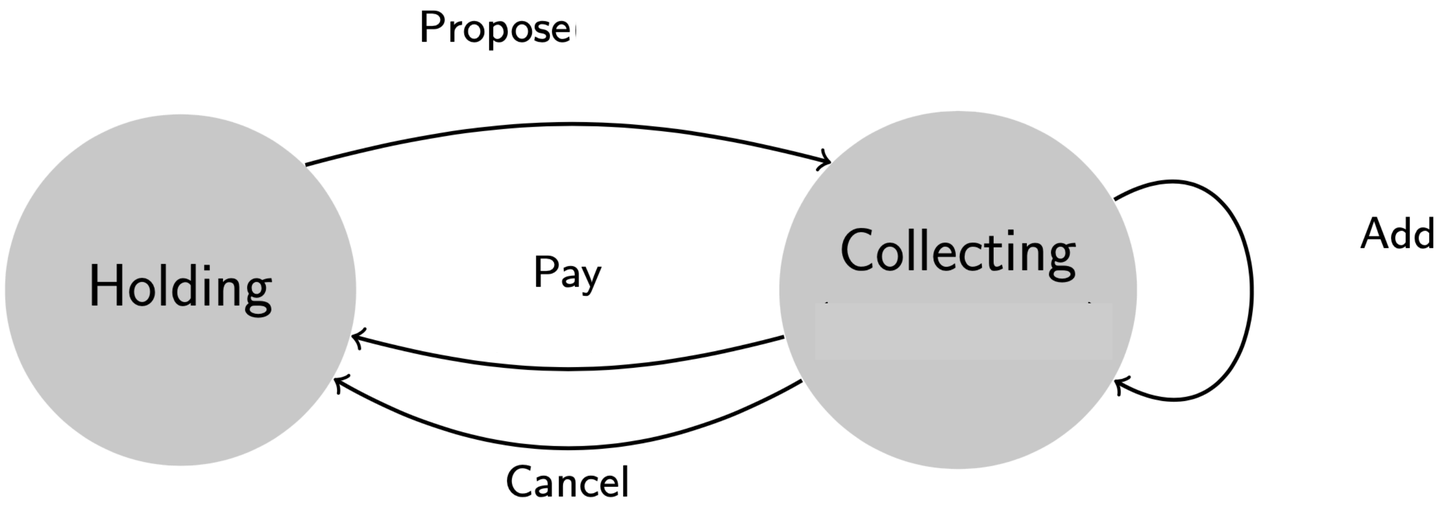
correctness of refinement

code extraction to Haskell

Our Trusted Example

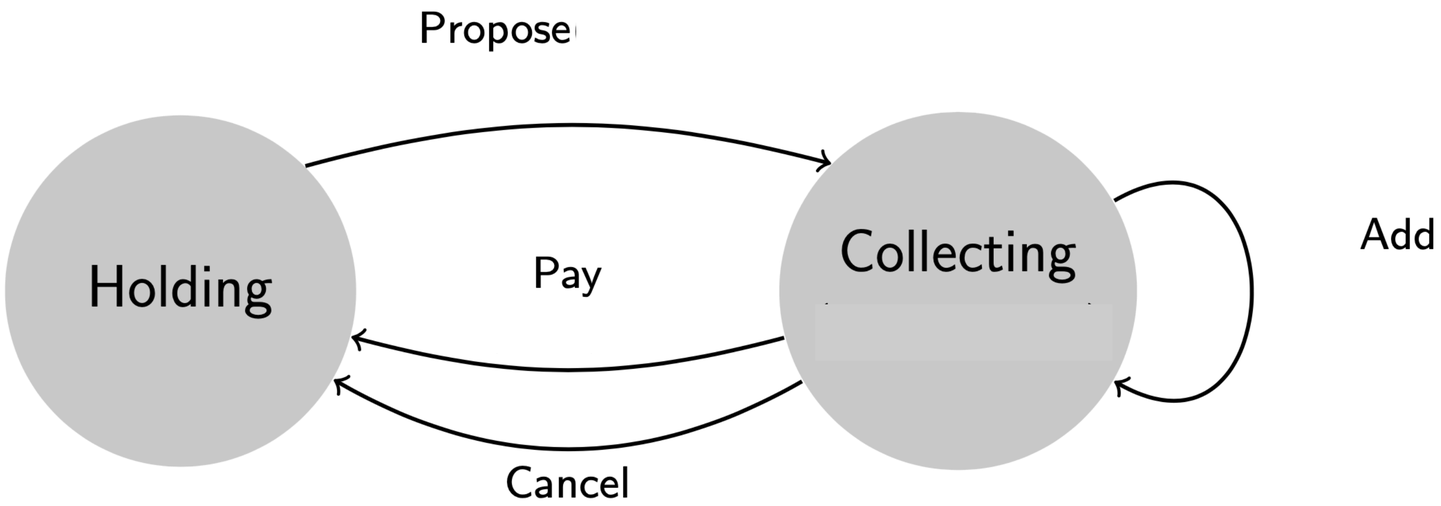


Automata Specification



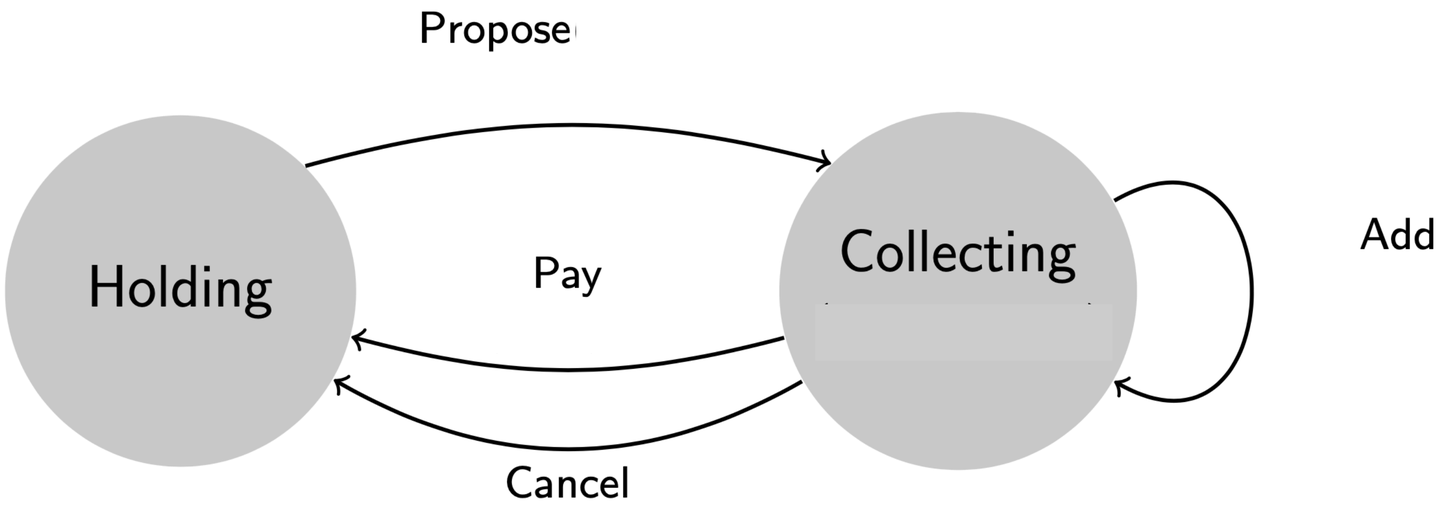
Automata Specification

The Base DFA



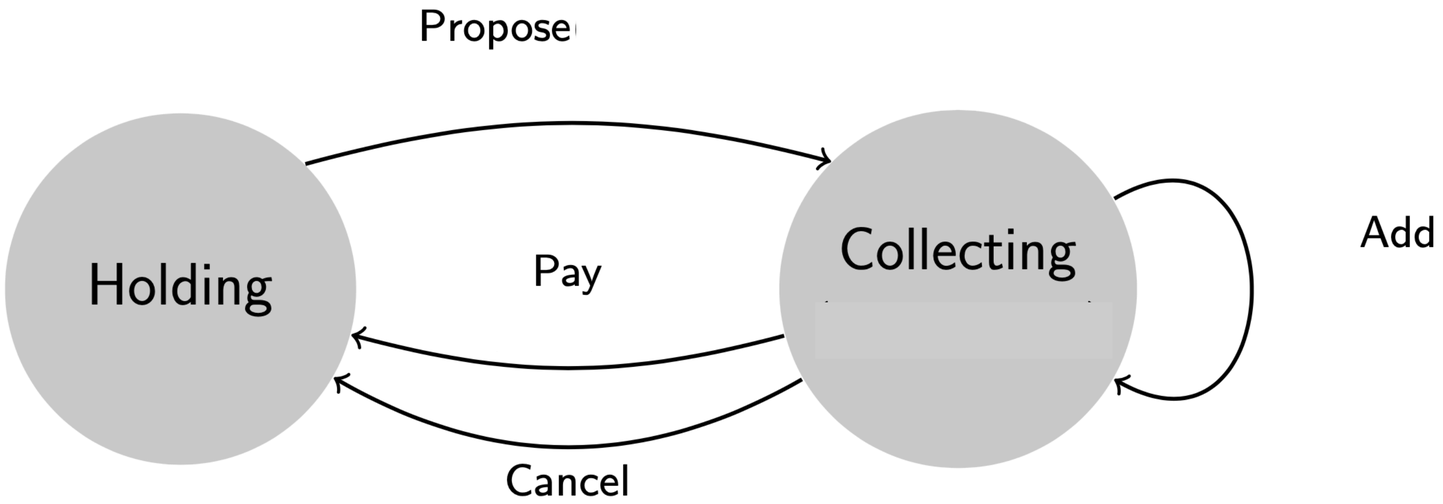
Automata Specification

The Base DFA

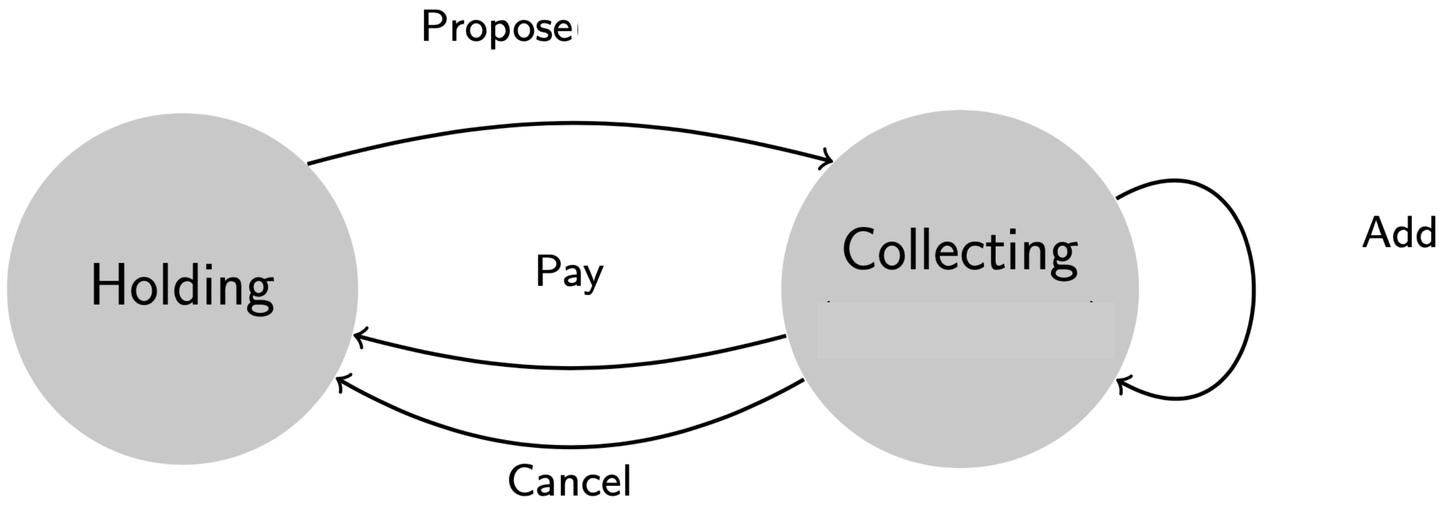



```
data State = Holding | Collecting
data Label = Propose | Add | Pay | Cancel
```

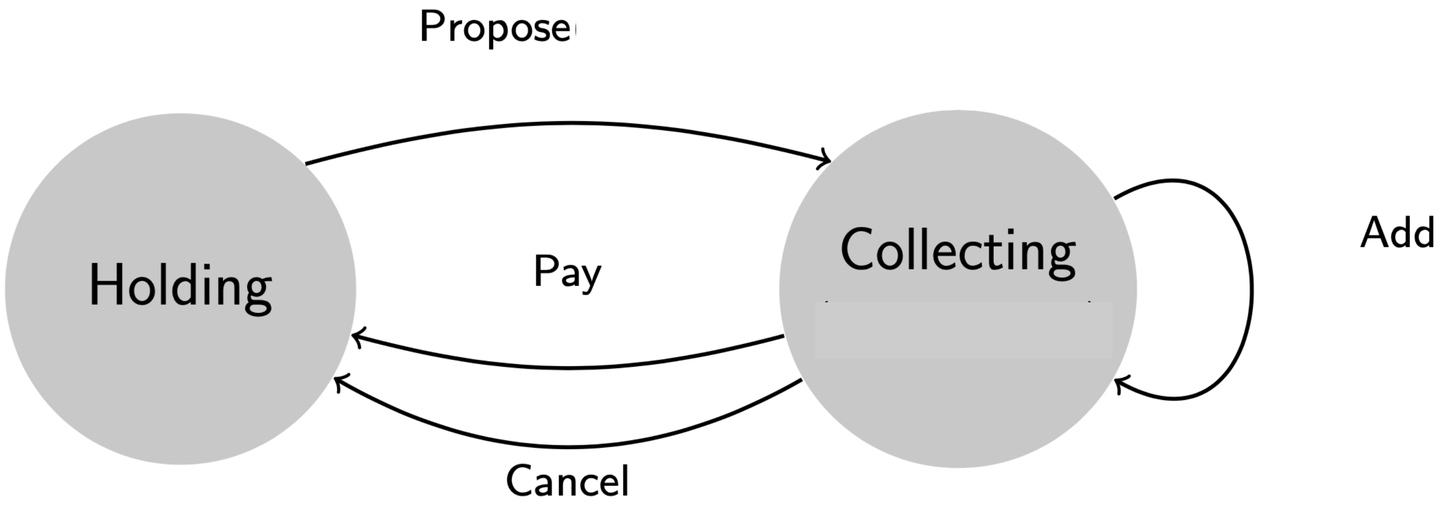
Automata Specification




```
data State = Holding | Collecting
data Label = Propose | Add | Pay | Cancel
```

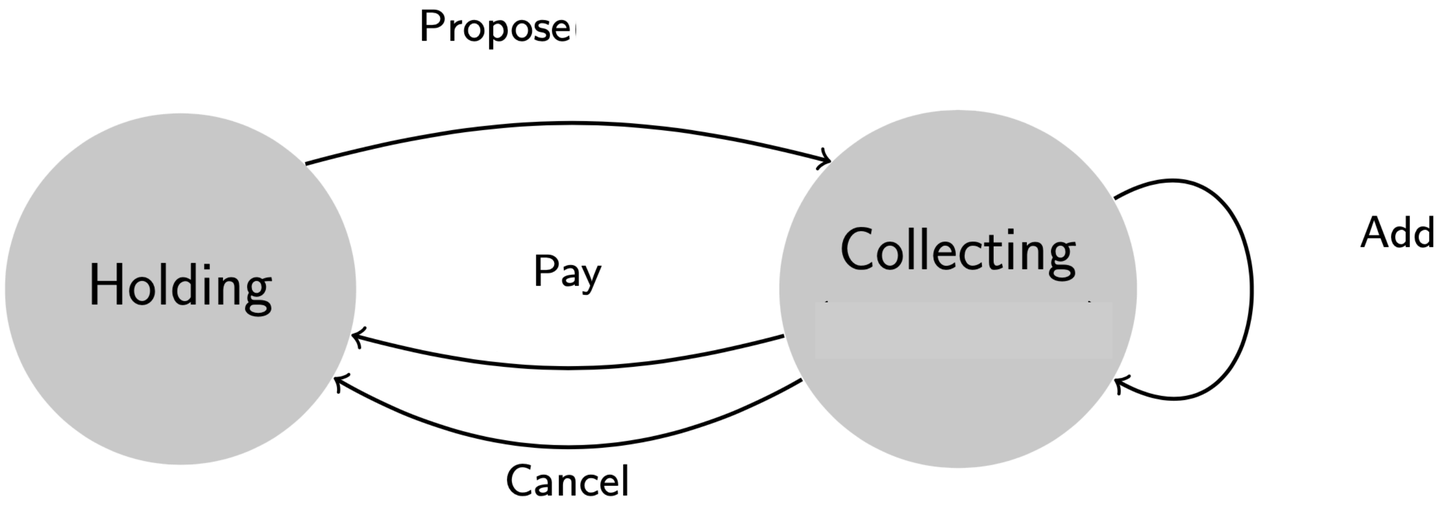



```
data State = Holding | Collecting
data Label = Propose | Add | Pay | Cancel
```



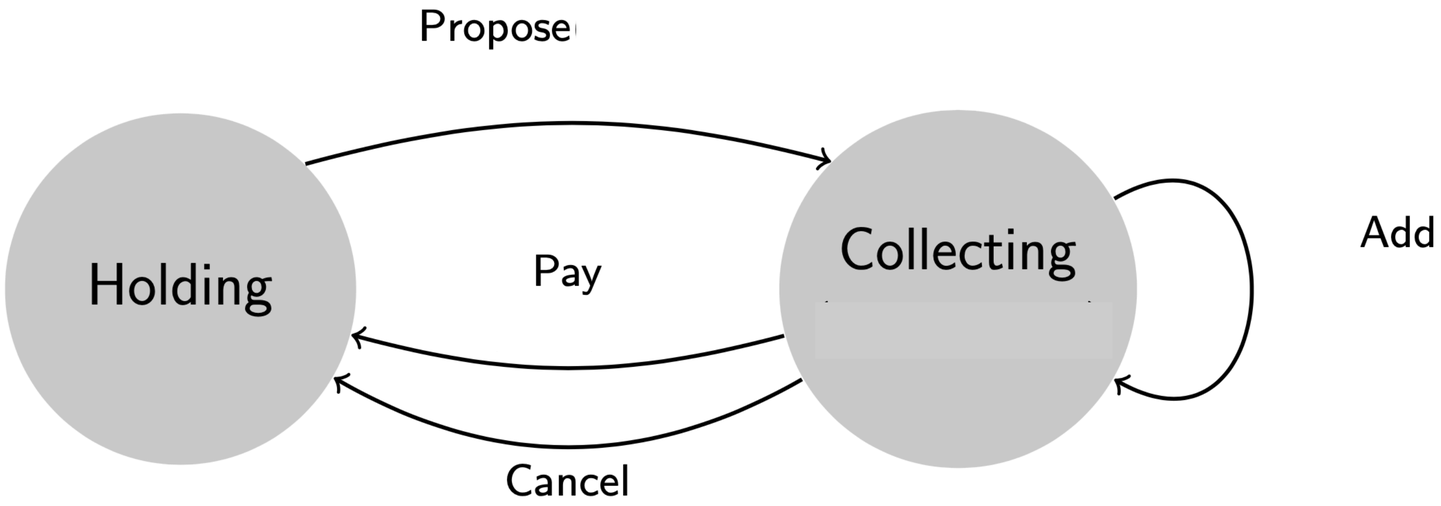

```
data State = Holding | Collecting
data Label = Propose | Add | Pay | Cancel
```

```
type family Transition (s: State) (l: Label) : State where
```



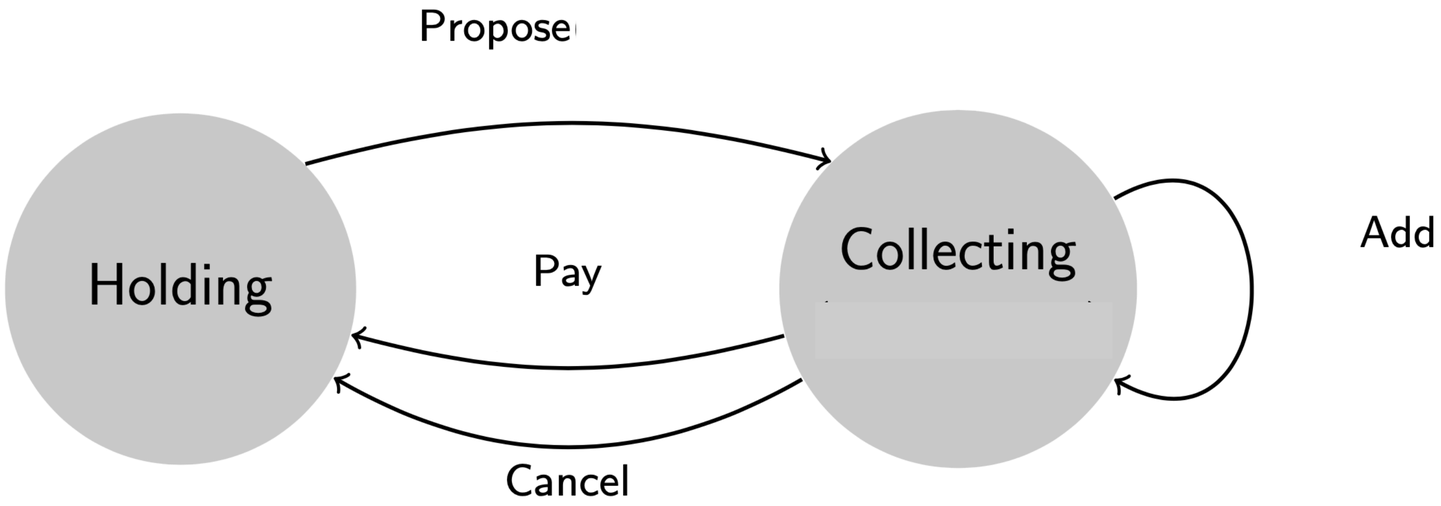

```
data State = Holding | Collecting
data Label = Propose | Add | Pay | Cancel
```

```
type family Transition (s: State) (l: Label) : State where
  Transition Holding    Propose = Collecting
```



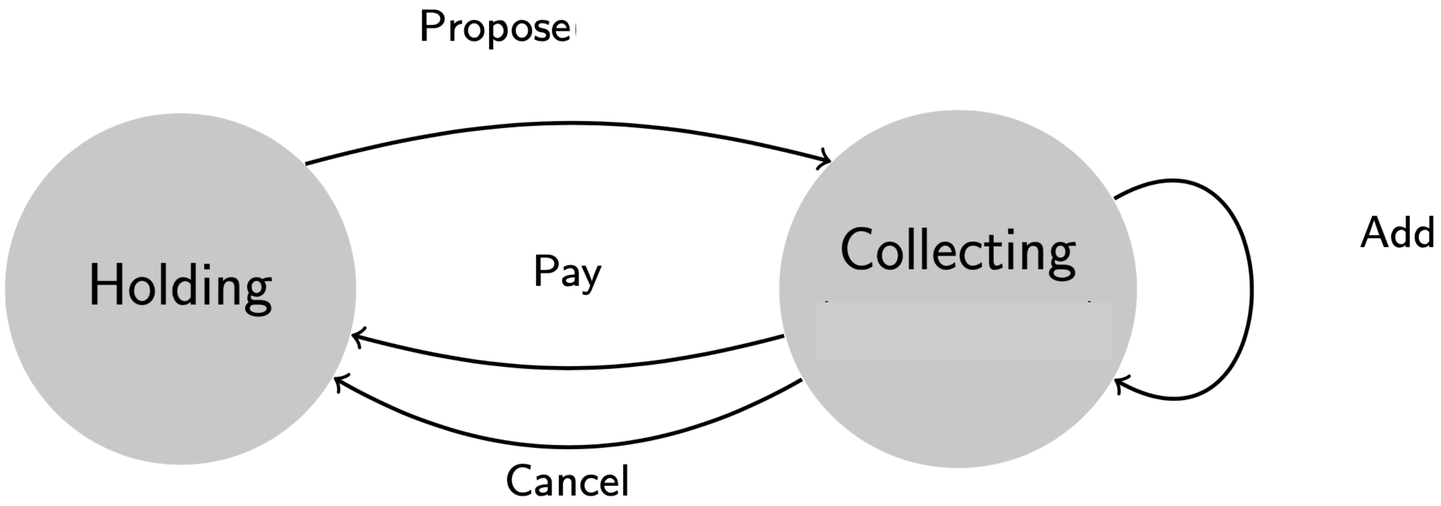

```
data State = Holding | Collecting
data Label = Propose | Add | Pay | Cancel
```

```
type family Transition (s: State) (l: Label) : State where
  Transition Holding    Propose = Collecting
  Transition Collecting Add     = Collecting
```




```
data State = Holding | Collecting
data Label = Propose | Add | Pay | Cancel
```

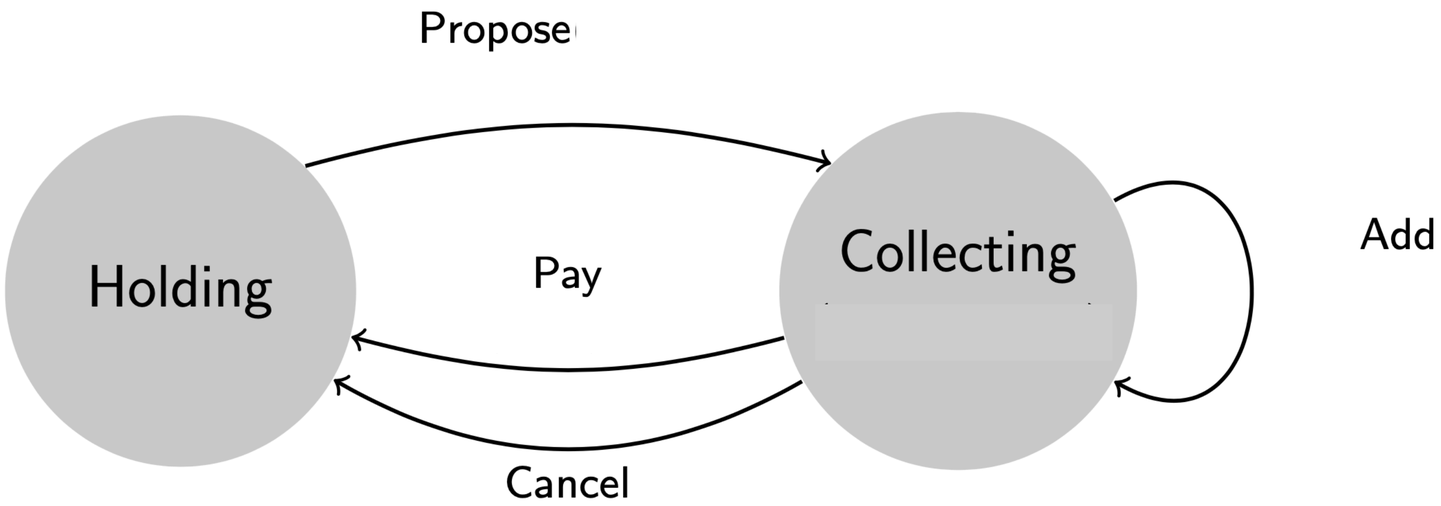
```
type family Transition (s: State) (l: Label) : State where
  Transition Holding    Propose = Collecting
  Transition Collecting Add     = Collecting
  Transition Collecting Pay     = Holding
```



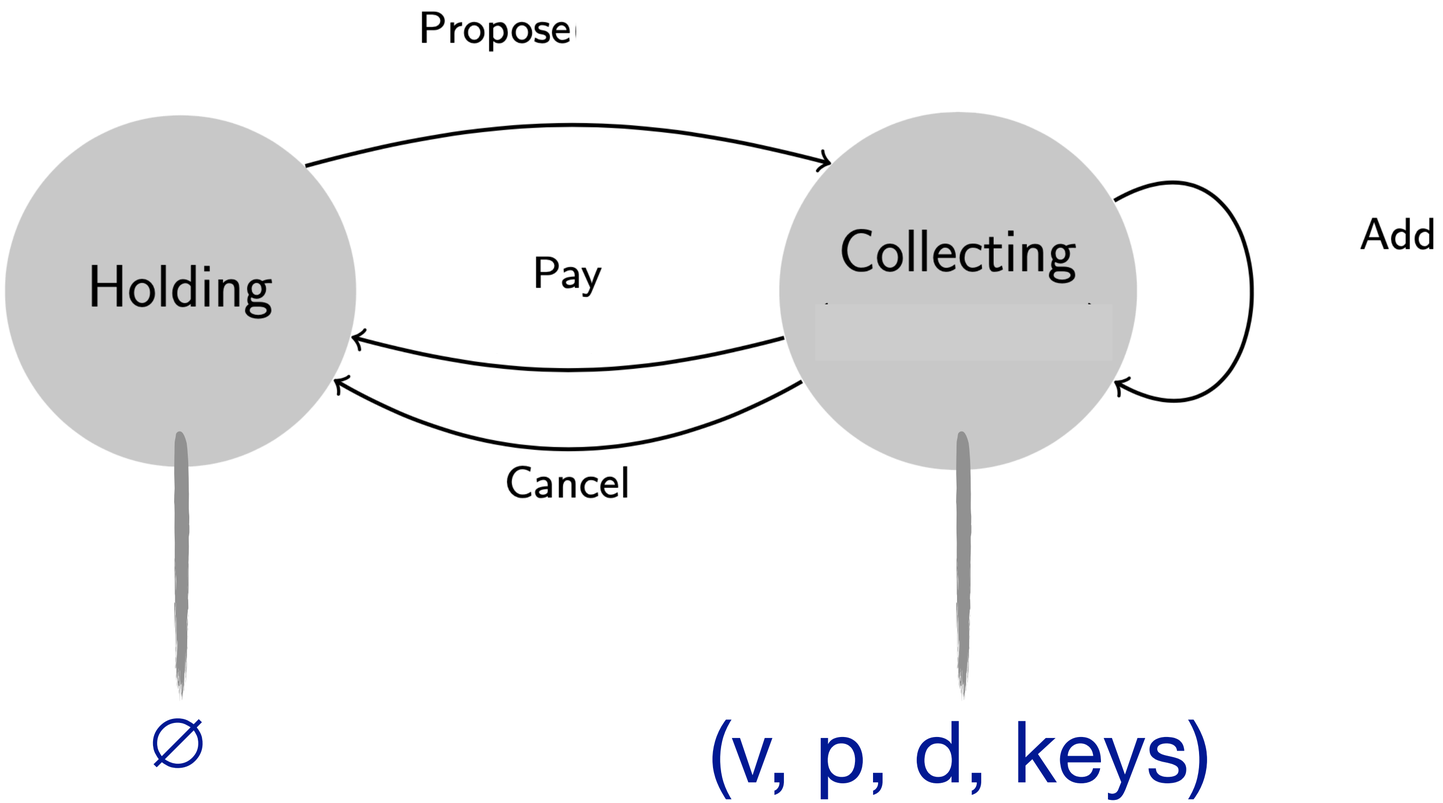

```
data State = Holding | Collecting
data Label = Propose | Add | Pay | Cancel
```

```
type family Transition (s: State) (l: Label) : State where
  Transition Holding    Propose = Collecting
  Transition Collecting Add     = Collecting
  Transition Collecting Pay     = Holding
  Transition Collecting Cancel  = Holding
```

Augmentation



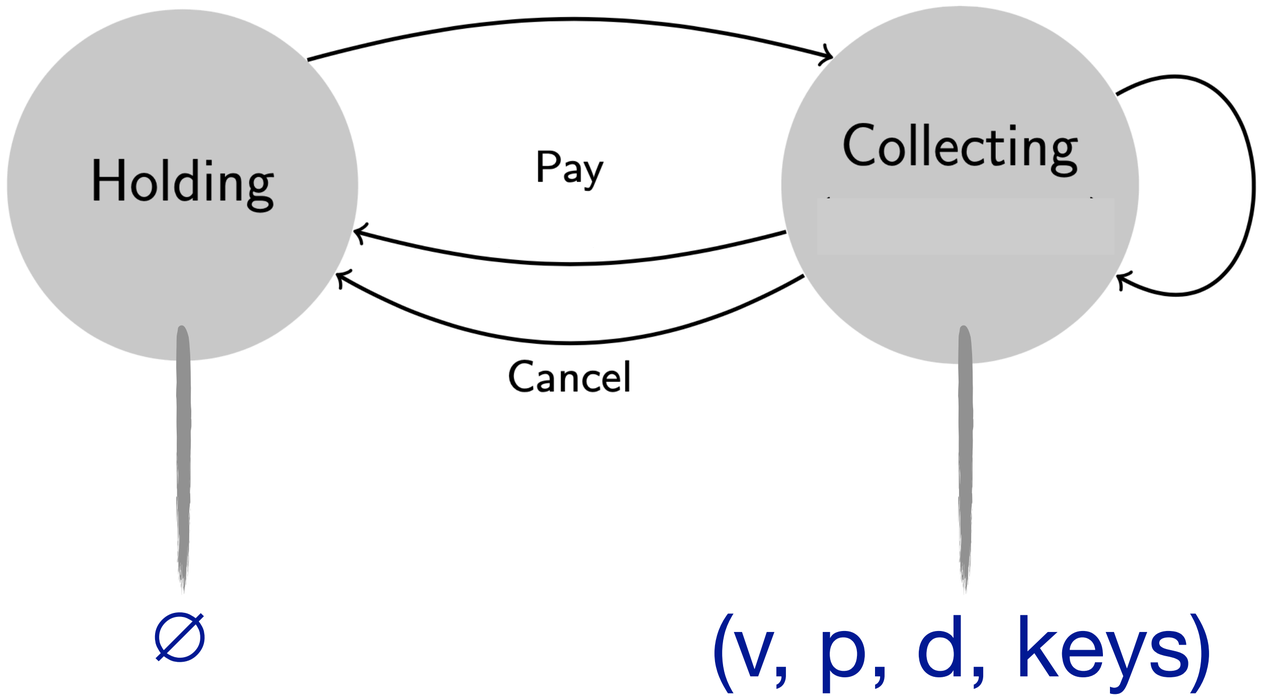
Augmentation



contract state

Augmentation

action arguments



contract state

Configurations — Contract State & Actions

Configurations — Contract State & Actions

```
type Payment = (amount: Value, recipient: Key, deadline: Int)
```

Configurations — Contract State & Actions

```
type Payment = (amount: Value, recipient: Key, deadline: Int)
```

```
data family ConfState (state : State) where
```

Configurations — Contract State & Actions

```
type Payment = (amount: Value, recipient: Key, deadline: Int)
```

```
data family ConfState (state : State) where
 ConfState Holding    = Holding
```

Configurations — Contract State & Actions

```
type Payment = (amount: Value, recipient: Key, deadline: Int)
```

```
data family ConfState (state : State) where
 ConfState Holding    = Holding
 ConfState Collecting = Collecting Payment (Set KeyHash)
```

Configurations — Contract State & Actions

```
type Payment = (amount: Value, recipient: Key, deadline: Int)
```

```
data family ConfState (state : State) where
 ConfState Holding    = Holding
 ConfState Collecting = Collecting Payment (Set KeyHash)

data family Action (label : Label) where
```

Configurations — Contract State & Actions

```
type Payment = (amount: Value, recipient: Key, deadline: Int)
```

```
data family ConfState (state : State) where
 ConfState Holding    = Holding
 ConfState Collecting = Collecting Payment (Set KeyHash)

data family Action (label : Label) where
 Action Propose = Propose Payment
```

Configurations — Contract State & Actions

```
type Payment = (amount: Value, recipient: Key, deadline: Int)
```

```
data family ConfState (state : State) where
 ConfState Holding    = Holding
 ConfState Collecting = Collecting Payment (Set KeyHash)

data family Action (label : Label) where
 Action Propose = Propose Payment
 Action Add     = Add     KeyHash
```

Configurations — Contract State & Actions

```
type Payment = (amount: Value, recipient: Key, deadline: Int)
```

```
data family ConfState (state : State) where
 ConfState Holding    = Holding
 ConfState Collecting = Collecting Payment (Set KeyHash)

data family Action (label : Label) where
 Action Propose = Propose Payment
 Action Add     = Add     KeyHash
 Action Pay     = Pay
```

Configurations — Contract State & Actions

```
type Payment = (amount: Value, recipient: Key, deadline: Int)
```

```
data family ConfState (state : State) where
 ConfState Holding    = Holding
 ConfState Collecting = Collecting Payment (Set KeyHash)

data family Action (label : Label) where
 Action Propose = Propose Payment
 Action Add     = Add     KeyHash
 Action Pay     = Pay
 Action Cancel  = Cancel
```

```
type Payment = (amount: Value, recipient: Key, deadline: Int)

data family ConfState (state: State) where
 ConfState Holding    = Holding
 ConfState Collecting = Collecting Payment (Set KeyHash)

data ReceiveFrom = ReceiveFrom Value KeyHash

data PayTo = PayTo Value KeyHash
```

```
type Payment = (amount: Value, recipient: Key, deadline: Int)

data family ConfState (state: State) where
 ConfState Holding    = Holding
 ConfState Collecting = Collecting Payment (Set KeyHash)

data ReceiveFrom = ReceiveFrom Value KeyHash

data PayTo = PayTo Value KeyHash
```

```
type Conf (s: State) = ( state:    ConfState s
```

```
type Payment = (amount: Value, recipient: Key, deadline: Int)

data family ConfState (state: State) where
 ConfState Holding    = Holding
 ConfState Collecting = Collecting Payment (Set KeyHash)

data ReceiveFrom = ReceiveFrom Value KeyHash

data PayTo = PayTo Value KeyHash
```

```
type Conf (s: State) = ( state:    ConfState s
                       , value:    Value
```

```
type Payment = (amount: Value, recipient: Key, deadline: Int)

data family ConfState (state: State) where
 ConfState Holding    = Holding
 ConfState Collecting = Collecting Payment (Set KeyHash)

data ReceiveFrom = ReceiveFrom Value KeyHash

data PayTo = PayTo Value KeyHash
```

```
type Conf (s: State) = ( state:    ConfState s
                       , value:    Value
                       , inputs:   [ReceiveFrom]
```

```
type Payment = (amount: Value, recipient: Key, deadline: Int)

data family ConfState (state: State) where
 ConfState Holding    = Holding
 ConfState Collecting = Collecting Payment (Set KeyHash)

data ReceiveFrom = ReceiveFrom Value KeyHash

data PayTo = PayTo Value KeyHash
```

```
type Conf (s: State) = ( state:    ConfState s
                       , value:    Value
                       , inputs:   [ReceiveFrom]
                       , outputs:  [PayTo]
```

```
type Payment = (amount: Value, recipient: Key, deadline: Int)

data family ConfState (state: State) where
 ConfState Holding    = Holding
 ConfState Collecting = Collecting Payment (Set KeyHash)

data ReceiveFrom = ReceiveFrom Value KeyHash

data PayTo = PayTo Value KeyHash
```

```
type Conf (s: State) = ( state:    ConfState s
                       , value:    Value
                       , inputs:   [ReceiveFrom]
                       , outputs:  [PayTo]
                       , duration: (Int, Int)
```

```
type Payment = (amount: Value, recipient: Key, deadline: Int)

data family ConfState (state: State) where
 ConfState Holding    = Holding
 ConfState Collecting = Collecting Payment (Set KeyHash)

data ReceiveFrom = ReceiveFrom Value KeyHash

data PayTo = PayTo Value KeyHash
```

```
type Conf (s: State) = ( state:    ConfState s
                       , value:    Value
                       , inputs:   [ReceiveFrom]
                       , outputs:  [PayTo]
                       , duration: (Int, Int)
                       , signers:  Set KeyHash
                       )
```

```
type Conf (s: State) = ( state:    ConfState s
                       , value:    Value
                       , inputs:   [ReceiveFrom]
                       , outputs:  [PayTo]
                       , duration: (Int, Int)
                       , signers:  Set KeyHash
                       )
```

```
type Conf (s: State) = ( state:    ConfState s
                       , value:    Value
                       , inputs:   [ReceiveFrom]
                       , outputs:  [PayTo]
                       , duration: (Int, Int)
                       , signers:  Set KeyHash
                       )
```

```
initial : Value -> Conf Holding
```

```
type Conf (s: State) = ( state:    ConfState s
                       , value:    Value
                       , inputs:   [ReceiveFrom]
                       , outputs:  [PayTo]
                       , duration: (Int, Int)
                       , signers:  Set KeyHash
                       )
```

```
initial : Value -> Conf Holding
initial val = ( state    = Holding
```

```
type Conf (s: State) = ( state:    ConfState s
                       , value:    Value
                       , inputs:   [ReceiveFrom]
                       , outputs:  [PayTo]
                       , duration: (Int, Int)
                       , signers:  Set KeyHash
                       )
```

```
initial : Value -> Conf Holding
initial val = ( state    = Holding
              , value    = val
```

```
type Conf (s: State) = ( state:    ConfState s
                       , value:    Value
                       , inputs:   [ReceiveFrom]
                       , outputs:  [PayTo]
                       , duration: (Int, Int)
                       , signers:  Set KeyHash
                       )
```

```
initial : Value -> Conf Holding
initial val = ( state    = Holding
              , value    = val
              , inputs   = [ReceiveFrom val owner]
```

```
type Conf (s: State) = ( state:    ConfState s
                       , value:    Value
                       , inputs:   [ReceiveFrom]
                       , outputs:  [PayTo]
                       , duration: (Int, Int)
                       , signers:  Set KeyHash
                       )
```

```
initial : Value -> Conf Holding
initial val = ( state    = Holding
              , value    = val
              , inputs   = [ReceiveFrom val owner]
              , outputs  = []
```

```
type Conf (s: State) = ( state:    ConfState s
                       , value:    Value
                       , inputs:   [ReceiveFrom]
                       , outputs:  [PayTo]
                       , duration: (Int, Int)
                       , signers:  Set KeyHash
                       )
```

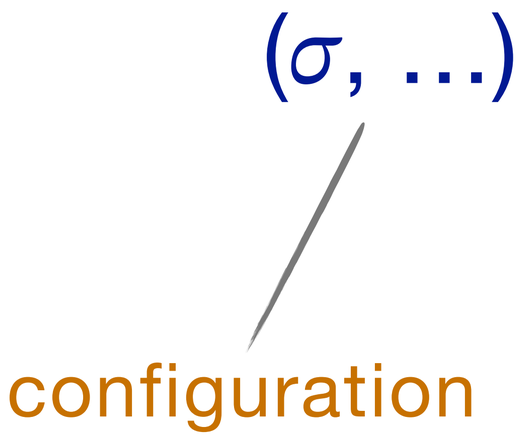
```
initial : Value -> Conf Holding
initial val = ( state    = Holding
              , value    = val
              , inputs   = [ReceiveFrom val owner]
              , outputs  = []
              , duration = (startTime, startTime)
```

```
type Conf (s: State) = ( state:    ConfState s
                       , value:    Value
                       , inputs:   [ReceiveFrom]
                       , outputs:  [PayTo]
                       , duration: (Int, Int)
                       , signers:  Set KeyHash
                       )
```

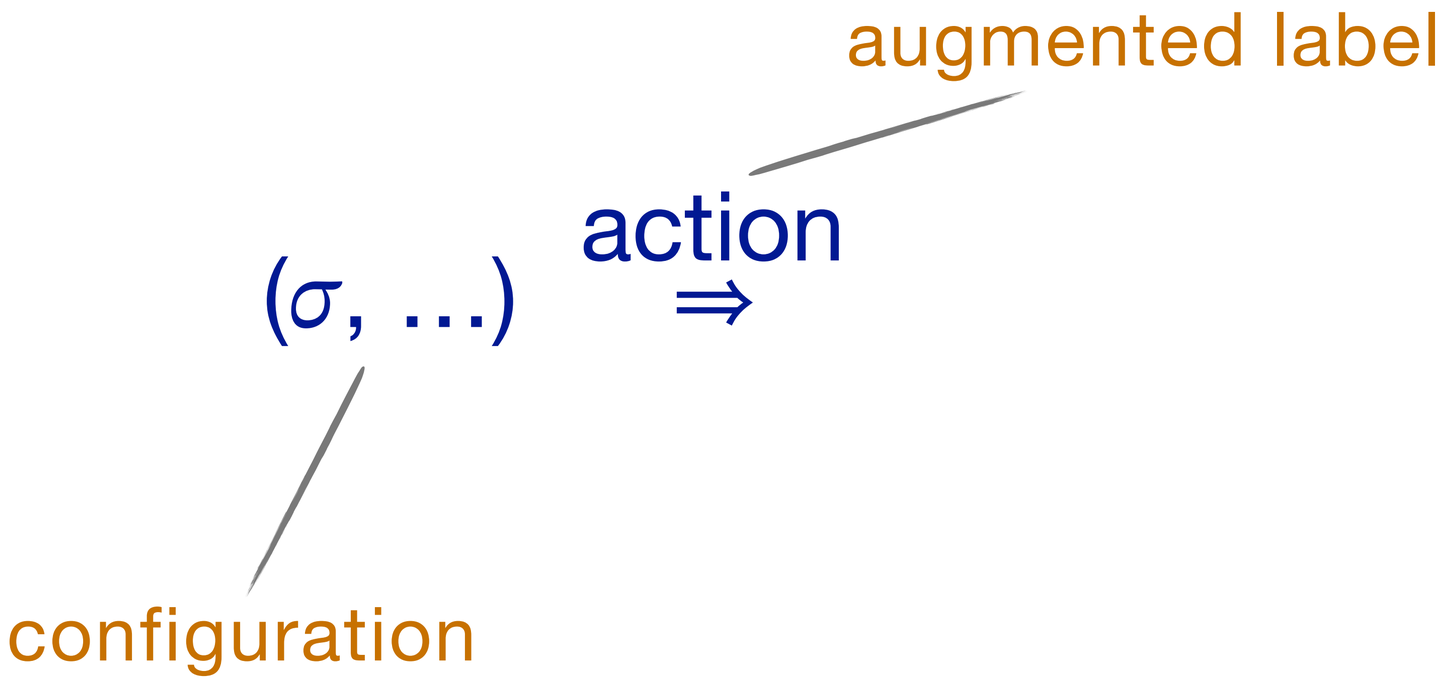
```
initial : Value -> Conf Holding
initial val = ( state    = Holding
              , value    = val
              , inputs   = [ReceiveFrom val owner]
              , outputs  = []
              , duration = (startTime, startTime)
              , signers  = { owner }
              )
```

Configuration Relation

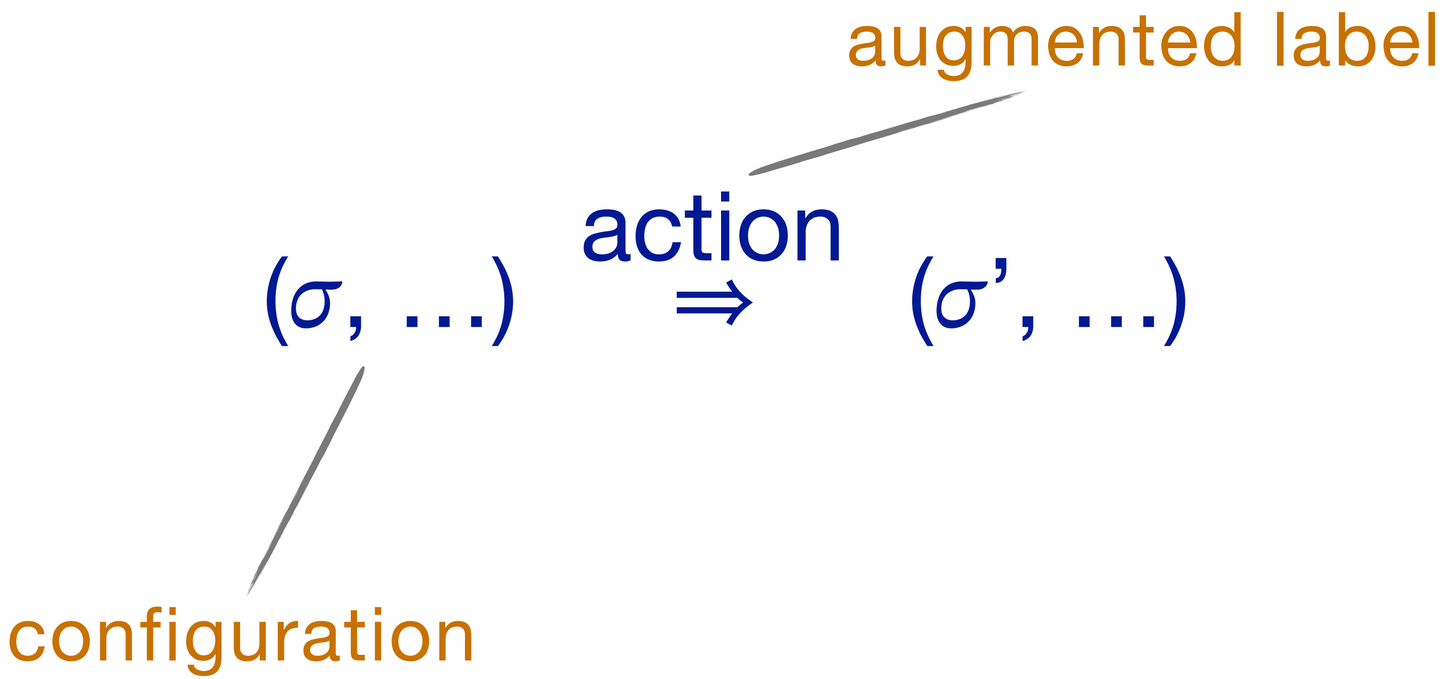
Configuration Relation



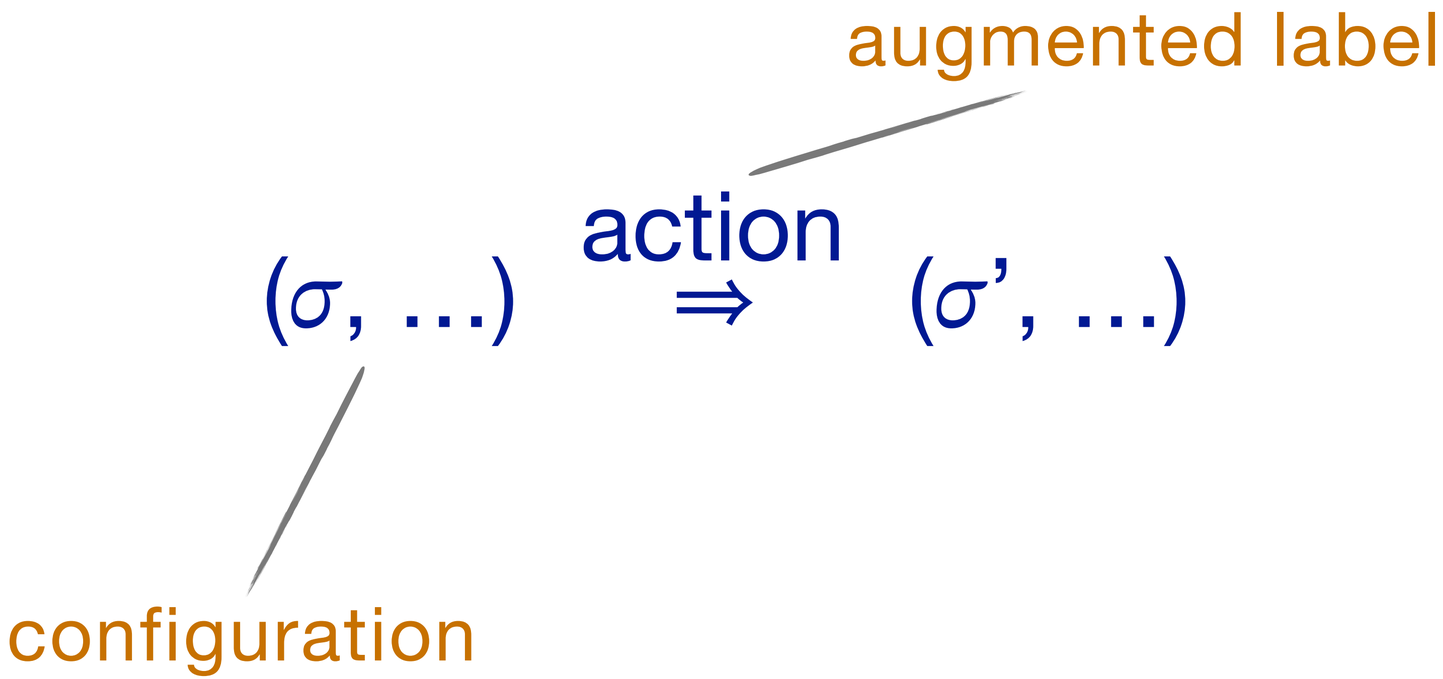
Configuration Relation



Configuration Relation



Configuration Relation




```
(=>) : Conf state * Action label * Conf (Transition state label)
```

Configuration Relation

Configuration Relation

```
(=>) : Conf state * Action label * Conf (Transition state label)
```

Configuration Relation

```
(=>) : Conf state * Action label * Conf (Transition state label)
```

```
c { state = Holding, value }
```

means: `inputs = []`, `outputs = []`,
and `durations` & `signers` are arbitrary

Configuration Relation

```
(=>) : Conf state * Action label * Conf (Transition state label)
```

```
c { state = Holding, value }
=> { Propose payment }
```

means: `inputs = []`, `outputs = []`,
and `durations` & `signers` are arbitrary

Configuration Relation

```
(=>) : Conf state * Action label * Conf (Transition state label)
```

```
c { state = Holding, value }
=> { Propose payment }
c' { state = Collecting payment {}, value }
```

means: `inputs = []`, `outputs = []`,
and `durations` & `signers` are arbitrary

Configuration Relation

```
(=>) : Conf state * Action label * Conf (Transition state label)
```

```
c { state = Holding, value }
=> { Propose payment }
c' { state = Collecting payment {}, value }
where
```

means: `inputs = []`, `outputs = []`,
and `durations` & `signers` are arbitrary

Configuration Relation

```
(=>) : Conf state * Action label * Conf (Transition state label)
```

```
c { state = Holding, value }
=> { Propose payment }
c' { state = Collecting payment {}, value }
where
  0 < payment.amount <= value
```

means: `inputs = []`, `outputs = []`,
and `durations` & `signers` are arbitrary

Configuration Relation

```
(=>) : Conf state * Action label * Conf (Transition state label)
```

```
c { state = Holding, value }
=> { Propose payment }
c' { state = Collecting payment {}, value }
where
  0 < payment.amount <= value
  payment.deadline > c'.duration.endTime
```

means: `inputs = []`, `outputs = []`,
and `durations` & `signers` are arbitrary

```
c { state = Collecting payment keys, value }
```

```
c { state = Collecting payment keys, value }
=> { Add key }
```

```
c { state = Collecting payment keys, value }
=> { Add key }
c’ { state = Collecting payment ({key} u keys), value }
```

```
c { state = Collecting payment keys, value }
=> { Add key }
c’ { state = Collecting payment ({key} u keys), value }
where
  key ∈ authorisedKeys
```

```
c { state = Collecting payment keys, value }
=> { Add key }
c’ { state = Collecting payment ({key} u keys), value }
where
  key ∈ authorisedKeys
```

```
c { state = Collecting payment keys, value }
```

```
c { state = Collecting payment keys, value }
=> { Add key }
c’ { state = Collecting payment ({key} u keys), value }
where
  key ∈ authorisedKeys
```

```
c { state = Collecting payment keys, value }
=> { Pay }
```

```
c { state = Collecting payment keys, value }
=> { Add key }
c’ { state = Collecting payment ({key} u keys), value }
where
  key ∈ authorisedKeys
```

```
c { state = Collecting payment keys, value }
=> { Pay }
c’ { state = Holding, value - payment.value
```

```
c { state = Collecting payment keys, value }
=> { Add key }
c' { state = Collecting payment ({key} u keys), value }
where
  key ∈ authorisedKeys
```

```
c { state = Collecting payment keys, value }
=> { Pay }
c' { state = Holding, value - payment.value
   , outputs = [PayTo payment.value payment.recipient]
```

```
c { state = Collecting payment keys, value }
=> { Add key }
c' { state = Collecting payment ({key} u keys), value }
where
  key ∈ authorisedKeys
```

```
c { state = Collecting payment keys, value }
=> { Pay }
c' { state = Holding, value - payment.value
   , outputs = [PayTo payment.value payment.recipient]
   }
```

```
c { state = Collecting payment keys, value }
=> { Add key }
c' { state = Collecting payment ({key} u keys), value }
where
  key ∈ authorisedKeys
```

```
c { state = Collecting payment keys, value }
=> { Pay }
c' { state = Holding, value - payment.value
   , outputs = [PayTo payment.value payment.recipient]
   }
where
  c'.duration.endTime <= payment.deadline
```

```
c { state = Collecting payment keys, value }
=> { Add key }
c' { state = Collecting payment ({key} u keys), value }
where
  key ∈ authorisedKeys
```

```
c { state = Collecting payment keys, value }
=> { Pay }
c' { state = Holding, value - payment.value
   , outputs = [PayTo payment.value payment.recipient]
   }
where
  c'.duration.endTime <= payment.deadline
  |keys| > noRequiredSignatures
```

```
c { state = Collecting payment keys, value }
=> { Cancel }
c’ { state = Holding, value }
where
  c’.duration.startTime > payment.deadline
```

Configuration Relation: Wellformedness

Configuration Relation: Wellformedness

Given $c_1 \Rightarrow^a c_2$, we require

Configuration Relation: Wellformedness

Given $c_1 \Rightarrow^a c_2$, we require

- c_1.state.label $\longrightarrow_{a.label}$ c_2.state.label

DFA transition

Configuration Relation: Wellformedness

DFA transition

Given $c_1 \Rightarrow^a c_2$, we require

- c_1.state.label $\longrightarrow_{a.label}$ c_2.state.label
- c_1.duration.endTime $<$ c_2.duration.endTime

Configuration Relation: Wellformedness

DFA transition

Given $c_1 \Rightarrow^a c_2$, we require

- c_1.state.label $\longrightarrow_{a.label}$ c_2.state.label
- c_1.duration.endTime $<$ c_2.duration.endTime

Anything else?

Contract Implementation

Contract Implementation

```
transition :: Conf (state :: State)
```

Contract Implementation

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
```

Contract Implementation

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  — inputs to consume
```

Contract Implementation

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  — inputs to consume
           -> Time                           — current time
```

Contract Implementation

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  -- inputs to consume
           -> Time                           -- current time
           -> Maybe (Conf (Transition state label))
```

Contract Implementation

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  — inputs to consume
           -> Time                           — current time
           -> Maybe (Conf (Transition state label))
```

```
validate   :: ConfState (state :: State)
```

Contract Implementation

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  — inputs to consume
           -> Time                           — current time
           -> Maybe (Conf (Transition state label))
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
```

Contract Implementation

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  -- inputs to consume
           -> Time                           -- current time
           -> Maybe (Conf (Transition state label))
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
```

Contract Implementation

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  — inputs to consume
           -> Time                           — current time
           -> Maybe (Conf (Transition state label))
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  - inputs to consume
           -> Time                           - current time
           -> Maybe (Conf (Transition state label))
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Holding } (Propose payment) [] time
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Holding } (Propose payment) [] time
  | 0 < amount payment && amount payment <= value conf
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Holding } (Propose payment) [] time
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > standardInterval
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Holding } (Propose payment) [] time
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > standardInterval
  = Just $
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Holding } (Propose payment) [] time
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > standardInterval
  = Just $
      ( state    = Collecting payment Set.empty
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Holding } (Propose payment) [] time
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > standardInterval
  = Just $
      ( state    = Collecting payment Set.empty
      , value    = value conf
```

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  -- inputs to consume
           -> Time                           -- current time
           -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Holding } (Propose payment) [] time
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > standardInterval
  = Just $
      ( state    = Collecting payment Set.empty
      , value    = value conf
      , inputs   = []
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Holding } (Propose payment) [] time
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > standardInterval
  = Just $
      ( state    = Collecting payment Set.empty
      , value    = value conf
      , inputs   = []
      , outputs  = []
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Holding } (Propose payment) [] time
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > standardInterval
  = Just $
      ( state    = Collecting payment Set.empty
      , value    = value conf
      , inputs   = []
      , outputs  = []
      , duration = (time, time + standardInterval)
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Holding } (Propose payment) [] time
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > standardInterval
  = Just $
      ( state    = Collecting payment Set.empty
      , value    = value conf
      , inputs   = []
      , outputs  = []
      , duration = (time, time + standardInterval)
      , signers  = Set.empty
      )
```

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  - inputs to consume
           -> Time                           - current time
           -> Maybe (Conf (Transition state label))
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys }
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys }
           (Add key) [] time
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys }
           (Add key) [] time
  | key `Set.elem` authorisedKeys
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys }
           (Add key) [] time
  | key `Set.elem` authorisedKeys
  = Just $
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys }
           (Add key) [] time
  | key `Set.elem` authorisedKeys
  = Just $
      conf{ state    = Collecting payment
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys }
           (Add key) [] time
  | key `Set.elem` authorisedKeys
  = Just $
      conf{ state    = Collecting payment
                                  (Set.addElement key keys)
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys }
           (Add key) [] time
  | key `Set.elem` authorisedKeys
  = Just $
      conf{ state    = Collecting payment
                                  (Set.addElement key keys)
          , duration = (time, time + standardInterval)
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys }
           (Add key) [] time
  | key `Set.elem` authorisedKeys
  = Just $
      conf{ state    = Collecting payment
                                  (Set.addElement key keys)
          , duration = (time, time + standardInterval)
          , signers  = Set.unit key
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys }
           (Add key) [] time
  | key `Set.elem` authorisedKeys
  = Just $
      conf{ state    = Collecting payment
                                  (Set.addElement key keys)
          , duration = (time, time + standardInterval)
          , signers  = Set.unit key
          }
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys }
           (Add key) [] time
  | key `Set.elem` authorisedKeys
  = Just $
      conf{ state    = Collecting payment
                                  (Set.addElement key keys)
          , duration = (time, time + standardInterval)
          , signers  = Set.unit key
          }
```

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  - inputs to consume
           -> Time                           - current time
           -> Maybe (Conf (Transition state label))
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys } Pay [] time
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys } Pay [] time
  | Set.count keys >= noRequiredSignatures
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys } Pay [] time
  | Set.count keys >= noRequiredSignatures
    && time < deadline payment
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys } Pay [] time
  | Set.count keys >= noRequiredSignatures
    && time < deadline payment
  = Just $
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys } Pay [] time
  | Set.count keys >= noRequiredSignatures
    && time < deadline payment
  = Just $
      ( state    = Holding
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys } Pay [] time
  | Set.count keys >= noRequiredSignatures
    && time < deadline payment
  = Just $
      ( state    = Holding
      , value    = value conf - amount payment
```

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  -- inputs to consume
           -> Time                           -- current time
           -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys } Pay [] time
  | Set.count keys >= noRequiredSignatures
    && time < deadline payment
  = Just $
      ( state    = Holding
      , value    = value conf - amount payment
      , inputs   = []
```

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  - inputs to consume
           -> Time                           - current time
           -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys } Pay [] time
  | Set.count keys >= noRequiredSignatures
    && time < deadline payment
  = Just $
      ( state    = Holding
      , value    = value conf - amount payment
      , inputs   = []
      , outputs  = [PayTo (amount payment)
```

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  -- inputs to consume
           -> Time                           -- current time
           -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys } Pay [] time
  | Set.count keys >= noRequiredSignatures
    && time < deadline payment
  = Just $
      ( state    = Holding
      , value    = value conf - amount payment
      , inputs   = []
      , outputs  = [PayTo (amount payment)
                          (recipient payment)]
```

```
  transition :: Conf (state :: State)
             -> Action (label :: Label)
             -> [ReceiveFrom]                  - inputs to consume
             -> Time                           - current time
             -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys } Pay [] time
  | Set.count keys >= noRequiredSignatures
    && time < deadline payment
  = Just $
      ( state    = Holding
      , value    = value conf - amount payment
      , inputs   = []
      , outputs  = [PayTo (amount payment)
                          (recipient payment)]
      , duration = (time, (time + standardInterval)
```

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  -- inputs to consume
           -> Time                           -- current time
           -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys } Pay [] time
  | Set.count keys >= noRequiredSignatures
    && time < deadline payment
  = Just $
      ( state    = Holding
      , value    = value conf - amount payment
      , inputs   = []
      , outputs  = [PayTo (amount payment)
                          (recipient payment)]
      , duration = (time, (time + standardInterval)
                          `min` deadline payment)
```

```
transition :: Conf (state :: State)
           -> Action (label :: Label)
           -> [ReceiveFrom]                  -- inputs to consume
           -> Time                           -- current time
           -> Maybe (Conf (Transition state label))
```

```
transition conf{ state = Collecting payment keys }
           Cancel [] time
  | time > deadline payment
  = Just $
      ( state    = Holding
      , value    = value conf
      , inputs   = []
      , outputs  = []
      , duration = (time, time + standardInterval)
      , signers  = Set.empty
      )
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate Holding (Propose payment)
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate Holding (Propose payment)
         conf{ state = Collecting payment keys, outputs = [] }
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate Holding (Propose payment)
         conf{ state = Collecting payment keys, outputs = [] }
  | 0 < amount payment && amount payment <= value conf
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate Holding (Propose payment)
         conf{ state = Collecting payment keys, outputs = [] }
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > snd (duration conf)
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate Holding (Propose payment)
         conf{ state = Collecting payment keys, outputs = [] }
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > snd (duration conf)
    && Set.isEmpty keys
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate Holding (Propose payment)
         conf{ state = Collecting payment keys, outputs = [] }
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > snd (duration conf)
    && Set.isEmpty keys
  = true
```

```
  validate   :: ConfState (state :: State)
             -> Action (label :: Label)
             -> Conf (Transition state label)
             -> Bool
```

```
validate Holding (Propose payment)
         conf{ state = Collecting payment keys, outputs = [] }
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > snd (duration conf)
    && Set.isEmpty keys
  = true
validate (Collecting payment keys) (Add key)
```

```
  validate   :: ConfState (state :: State)
             -> Action (label :: Label)
             -> Conf (Transition state label)
             -> Bool
```

```
validate Holding (Propose payment)
         conf{ state = Collecting payment keys, outputs = [] }
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > snd (duration conf)
    && Set.isEmpty keys
  = true
validate (Collecting payment keys) (Add key)
         conf{ state = Collecting payment keys, outputs = [] }
```

```
  validate   :: ConfState (state :: State)
             -> Action (label :: Label)
             -> Conf (Transition state label)
             -> Bool
```

```
validate Holding (Propose payment)
         conf{ state = Collecting payment keys, outputs = [] }
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > snd (duration conf)
    && Set.isEmpty keys
  = true
validate (Collecting payment keys) (Add key)
         conf{ state = Collecting payment keys, outputs = [] }
  | key `Set.elem` signers conf
```

```
  validate   :: ConfState (state :: State)
             -> Action (label :: Label)
             -> Conf (Transition state label)
             -> Bool
```

```
validate Holding (Propose payment)
         conf{ state = Collecting payment keys, outputs = [] }
  | 0 < amount payment && amount payment <= value conf
    && deadline payment > snd (duration conf)
    && Set.isEmpty keys
  = true
validate (Collecting payment keys) (Add key)
         conf{ state = Collecting payment keys, outputs = [] }
  | key `Set.elem` signers conf
    && key `Set.elem` authorisedKeys && key `Set.elem` keys
  = true
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate (Collecting payment keys) Pay
```

```
  validate   :: ConfState (state :: State)
             -> Action (label :: Label)
             -> Conf (Transition state label)
             -> Bool
```

```
validate (Collecting payment keys) Pay
         conf{ state = Holding,
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate (Collecting payment keys) Pay
         conf{ state = Holding,
               outputs = PayTo paidAmount paidRecipient : _ }
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate (Collecting payment keys) Pay
         conf{ state = Holding,
               outputs = PayTo paidAmount paidRecipient : _ }
  | Set.count keys >= noRequiredSignatures
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate (Collecting payment keys) Pay
         conf{ state = Holding,
               outputs = PayTo paidAmount paidRecipient : _ }
  | Set.count keys >= noRequiredSignatures
    && snd (duration conf) =< payment.deadline
```

```
  validate   :: ConfState (state :: State)
             -> Action (label :: Label)
             -> Conf (Transition state label)
             -> Bool
```

```
validate (Collecting payment keys) Pay
         conf{ state = Holding,
                outputs = PayTo paidAmount paidRecipient : _ }
  | Set.count keys >= noRequiredSignatures
    && snd (duration conf) =< payment.deadline
    && paidAmount == amount payment
```

```
  validate   :: ConfState (state :: State)
             -> Action (label :: Label)
             -> Conf (Transition state label)
             -> Bool
```

```
validate (Collecting payment keys) Pay
         conf{ state = Holding,
                outputs = PayTo paidAmount paidRecipient : _ }
  | Set.count keys >= noRequiredSignatures
    && snd (duration conf) =< payment.deadline
    && paidAmount == amount payment
    && paidRecipient == recipient payment
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate (Collecting payment keys) Pay
         conf{ state = Holding,
                outputs = PayTo paidAmount paidRecipient : _ }
  | Set.count keys >= noRequiredSignatures
    && snd (duration conf) =< payment.deadline
    && paidAmount == amount payment
    && paidRecipient == recipient payment
  = true
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate   :: ConfState (state :: State)
           -> Action (label :: Label)
           -> Conf (Transition state label)
           -> Bool
```

```
validate (Collecting payment keys) Cancel
```

```
  validate   :: ConfState (state :: State)
             -> Action (label :: Label)
             -> Conf (Transition state label)
             -> Bool
```

```
validate (Collecting payment keys) Cancel
         conf{ state = Holding, outputs = [] }
```

```
  validate   :: ConfState (state :: State)
             -> Action (label :: Label)
             -> Conf (Transition state label)
             -> Bool
```

```
validate (Collecting payment keys) Cancel
         conf{ state = Holding, outputs = [] }
  | fst (duration conf) > payment.deadline
```

```
validate :: ConfState (state :: State)
         -> Action (label :: Label)
         -> Conf (Transition state label)
         -> Bool
```

```
validate (Collecting payment keys) Cancel
         conf{ state = Holding, outputs = [] }
  | fst (duration conf) > payment.deadline
  = true
```

```
  validate   :: ConfState (state :: State)
             -> Action (label :: Label)
             -> Conf (Transition state label)
             -> Bool
```

```
validate (Collecting payment keys) Cancel
         conf{ state = Holding, outputs = [] }
  | fst (duration conf) > payment.deadline
  = true
validate _ _ _
```

```
  validate   :: ConfState (state :: State)
             -> Action (label :: Label)
             -> Conf (Transition state label)
             -> Bool
```

```
validate (Collecting payment keys) Cancel
         conf{ state = Holding, outputs = [] }
  | fst (duration conf) > payment.deadline
  = true
validate _ _ _
  = false
```

Correctness

For all configurations c, labels l, actions a :: Action l, inputs is, time t, we have

if transition $c\ a\ is\ t = \text{Just}\ c'$,
then $c \Rightarrow^{a} c'$ and c'.inputs $\subseteq is$

Correctness

For all configurations c, labels l, actions a :: Action l, inputs is, time t, we have

if transition $c\ a\ is\ t = \text{Just}\ c'$,
then validate c.state $a\ c' = \text{True}$

Completeness

???

Thank you!

Image Attribution

https://pixabay.com/images/id-1595995/
https://pixabay.com/images/id-1236578/
https://pixabay.com/images/id-682010/

Icons licensed
from Noun Project