A Pegged and Crypto-Backed Algorithmic Stablecoin

Dr. Bruno Woltzenlogel Paleo
Technical Director, Input Output

Joint work with:

ERGO

- Alexander Chepurnoy
- Amitabh Saxena

INPUT | OUTPUT

- Colin Edwards
- Dr. Dmytro Kaidalov
- Dr. Jann Müller

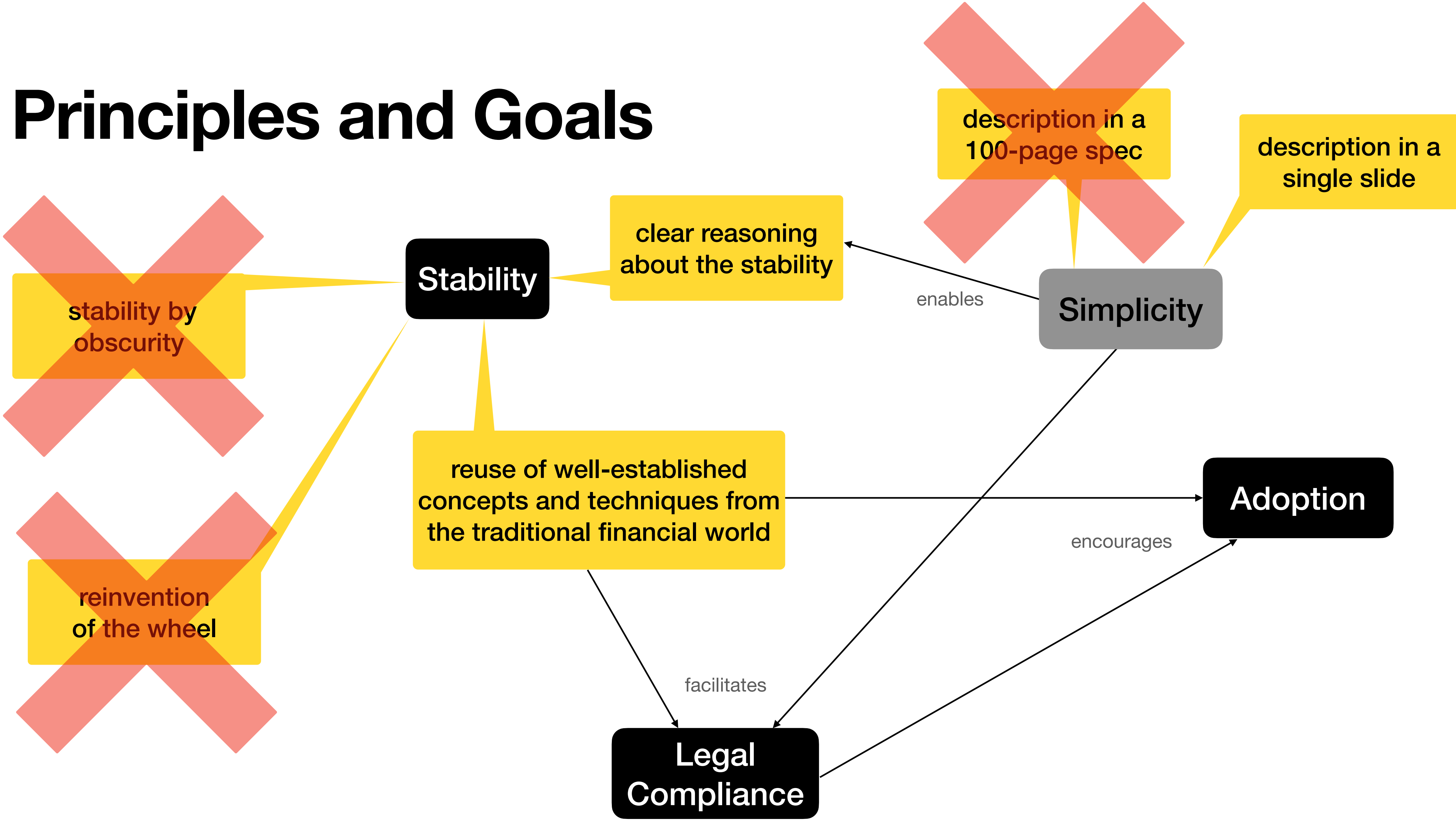
EMURGO

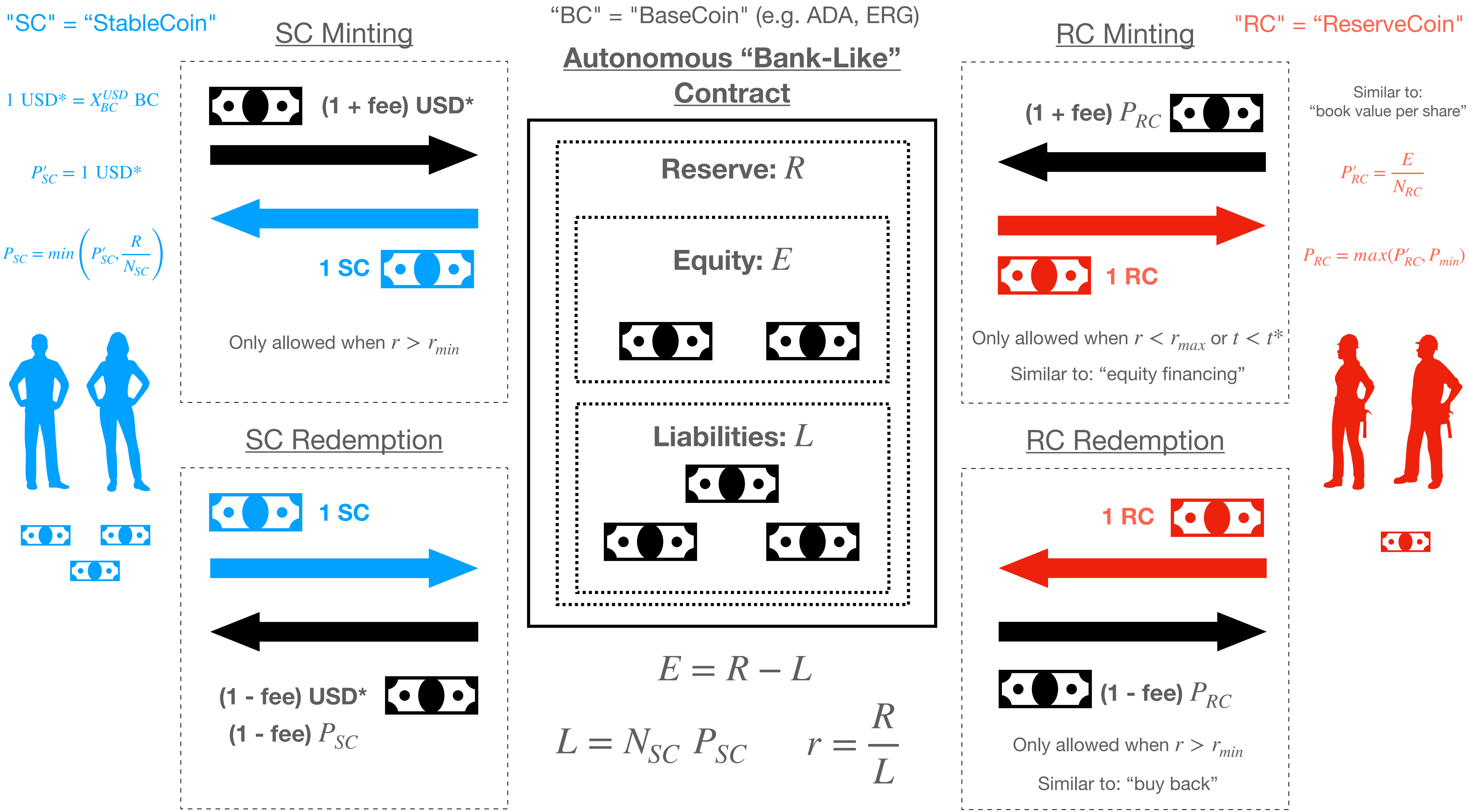
- Nicolas Arqueros
- Robert Kornacki

Definitions

- **Stablecoin**: a digital asset whose price w.r.t. another asset has low volatility
    - Asset examples:
        - USD, EUR, BTC, Gold, Silver, stocks, S&P500, inflation indexes…
    - **Pegged**: tries to keep the volatility as close as possible to zero.
    - **Backed**: maintains *reserves* to enable the stabilization mechanisms.
    - **Crypto-backed**: reserves are made of cryptocurrencies (e.g. ADA, ERG).
    - **Algorithmic**: stabilization mechanisms follow an algorithm.

Principles and Goals





"SC" = "StableCoin"
SC Minting
"BC" = "BaseCoin" (e.g. ADA, ERG)
Autonomous "Bank-Like" Contract
RC Minting
"RC" = "ReserveCoin"
$1\ \text{USD}^* = X_{BC}^{USD}\ \text{BC}$
(1 + fee) USD*
Reserve: R
(1 + fee) P_{RC}
Similar to: "book value per share"
$P'_{SC} = 1\ \text{USD}^*$
$P'_{RC} = \frac{E}{N_{RC}}$
$P_{SC} = min\left(P'_{SC}, \frac{R}{N_{SC}}\right)$
1 SC
Equity: E
1 RC
$P_{RC} = max(P'_{RC}, P_{min})$
Only allowed when $r > r_{min}$
Only allowed when $r < r_{max}$ or $t < t^*$
Similar to: "equity financing"
SC Redemption
Liabilities: L
RC Redemption
1 SC
1 RC
(1 - fee) USD*
(1 - fee) P_{SC}
(1 - fee) P_{RC}
$E = R - L$
Only allowed when $r > r_{min}$
$L = N_{SC}\ P_{SC}$
$r = \frac{R}{L}$
Similar to: "buy back"

Why is it stable?

- It would be futile for someone to try to:
  - sell a SC for more than 1 USD*
    - potential buyers could buy SC directly from the contract for 1 USD* instead.
  - buy a SC for less than 1 USD*
    - potential sellers could sell SC directly to the contract for 1 USD* instead.

Under which assumptions?

- $r > r_{min}$
  - otherwise buying SCs directly from the contract is not enabled.
- $E > 0$
  - otherwise selling an SC directly to the contract will give the seller less than 1 USD*.
- negligible blockchain congestion

Are the assumptions reasonable?

- RC minting (aka "equity financing") encourages the maintenance of $r > r_{min}$ and $E > 0$
- Potential RC buyers are encouraged to buy:
  - to profit from the accumulation of fees
  - because they are protected from "dilution" by r_{max}

Why a *pegged* stablecoin? Why USD as the peg?

- The most popular stablecoins nowadays are pegged to the USD
- Still no USD-pegged stablecoin on Ergo or Cardano
- A fiat-pegged stablecoin is potentially useful as a means of exchange
- A USD-pegged stablecoin is useful for people in countries with weaker currencies

- Pegging to an off-chain asset requires an oracle

- Our design could work with other pegs as well
- In the longer term, unpegged stablecoins are worth pursuing too

Why an *algorithmic* stablecoin?

- Demonstration of Cardano’s Plutus and Ergo’s ErgoScript capabilities
- Demonstration of an algorithmic stablecoin on UTxO-based blockchains
- An algorithmic stablecoin:
  - is autonomous and hence requires less management
  - is less susceptible to misbehaviour by a managing entity

Implementation in Cardano’s Plutus

Note: it is slightly outdated and with some minor discrepancies

```haskell
-- | The bank's state
data BankState =
    BankState
        { bsReserves             :: BC Integer -- ^ Value of the bank's reserves in base currency
        , bsStablecoins          :: SC Integer -- ^ Amount of stablecoins in circulation
        , bsReservecoins         :: RC Integer -- ^ Amount of reservecoins currently in circulation
        , bsForgingPolicyScript :: MonetaryPolicyHash -- ^ Hash of the forging policy that forwards
        }
    deriving stock (Generic, Haskell.Eq, Haskell.Show)
    deriving anyclass (ToJSON, FromJSON)
```

```haskell
{-# INLINEABLE liabilities #-}
-- | The bank's liabilities (total value of stablecoins in base currency)
liabilities ::
    BankState
    -> ConversionRate
    -> BC (Ratio Integer)
liabilities BankState{bsReserves=BC reserves,bsStablecoins=SC stablecoins} cr =
    let BC stableCoinLiabilities = convert cr (PC $ fromInteger stablecoins)
    in BC (min (fromInteger reserves) stableCoinLiabilities)

{-# INLINEABLE equity #-}
-- | The bank's equity (what's left of the reserves after subtracting
--   liabilities).
equity ::
    BankState
    -> ConversionRate
    -> BC (Ratio Integer)
equity r@BankState{bsReserves=BC reserves} cr =
    let BC l = liabilities r cr
    in BC (fromInteger reserves - l)
```

```haskell
-- | Stablecoin parameters.
data Stablecoin =
    Stablecoin
        { scOracle                 :: PubKey -- ^ Public key of the oracle that provides exchange rates
        , scFee                    :: Ratio Integer -- ^ Fee charged by bank for transactions. Calculated as a fraction of t
        , scMinReserveRatio        :: Ratio Integer -- ^ The minimum ratio of reserves to liabilities
        , scMaxReserveRatio        :: Ratio Integer -- ^ The maximum ratio of reserves to liabilities
        , scReservecoinDefaultPrice :: BC Integer -- ^ The price of a single reservecoin if no reservecoins have been issued
        , scBaseCurrency           :: (CurrencySymbol, TokenName) -- ^ The base currency. Value of this currency will be loc
        , scStablecoinTokenName    :: TokenName -- ^ 'TokenName' of the stablecoin
        , scReservecoinTokenName   :: TokenName -- ^ 'TokenName' of the reservecoin
        }
    deriving stock (Generic, Haskell.Eq, Haskell.Show)
    deriving anyclass (ToJSON, FromJSON)
```

```haskell
-- | Action that can be performed on the stablecoin contract.
data SCAction
    = MintStablecoin (SC Integer) -- ^ Create a number stablecoins, depositing the matching amount of base currency
    | MintReserveCoin (RC Integer) -- ^ Create a number of reservecoins, depositing the matching amount of base currency
    deriving stock (Generic, Haskell.Eq, Haskell.Show)
    deriving anyclass (ToJSON, FromJSON)
```

```haskell
{-# INLINEABLE applyInput #-}
-- | Given a stablecoin definition, current state and input, compute the
--   new state and tx constraints, without checking whether the new state
--   is valid.
applyInput :: forall i o. Stablecoin -> BankState -> Input -> Maybe (TxConstraints i o, BankState)
applyInput sc@Stablecoin{scOracle,scStablecoinTokenName,scReservecoinTokenName} bs@BankState{bsForgingPolicyScript} Input{inpSCAction, inpConversionRate} = do
    (Observation{obsValue=rate, obsSlot}, constraints) <- either (const Nothing) pure (verifySignedMessageConstraints scOracle inpConversionRate)
    let fees = calcFees sc bs rate inpSCAction
        (newState, newConstraints) = case inpSCAction of
            MintStablecoin sc' ->
                let scValue = stablecoinNominalPrice bs rate * (BC $ fromInteger $ unSC sc') in
                (bs
                { bsStablecoins = bsStablecoins bs + sc'
                , bsReserves = bsReserves bs + fmap round (fees + scValue)
                }, Constraints.mustForgeCurrency bsForgingPolicyScript scStablecoinTokenName (unSC sc'))
            MintReserveCoin rc ->
                let rcValue = reservecoinNominalPrice sc bs rate * (BC $ fromInteger $ unRC rc) in
                (bs
                { bsReservecoins = bsReservecoins bs + rc
                , bsReserves = bsReserves bs + fmap round (fees + rcValue)
                }, Constraints.mustForgeCurrency bsForgingPolicyScript scReservecoinTokenName (unRC rc))
    let dateConstraints = Constraints.mustValidateIn $ Interval.from obsSlot
    pure (constraints <> newConstraints <> dateConstraints, newState)
```

```haskell
{-# INLINEABLE checkValidState #-}
checkValidState :: Stablecoin -> BankState -> ConversionRate -> Either InvalidStateReason ()
checkValidState sc bs@BankState{bsReservecoins, bsReserves, bsStablecoins} cr = do
    -- TODO: Do we need a validation type in the state machine lib?
    unless (bsReservecoins >= RC 0) (Left NegativeReserveCoins)
    unless (bsReserves >= BC 0) (Left NegativeReserves)
    unless (bsStablecoins >= SC 0) (Left NegativeStablecoins)
    unless (liabilities bs cr >= zero) (Left NegativeLiabilities)
    unless (equity bs cr >= zero) (Left NegativeEquity)

    let actualReserves = fmap fromInteger bsReserves
        allowedReserves = (,) <$> minReserve sc cr bs <*> maxReserve sc cr bs

    case allowedReserves of
        Just (minReserves, maxReserves) -> do
            unless (actualReserves >= minReserves) (Left $ MinReserves minReserves actualReserves)
            unless (actualReserves <= maxReserves) (Left $ MaxReserves maxReserves actualReserves)
        Nothing -> pure ()
```

Current and Future Work

- Debt-to-Equity Swap
  - Smoother handling of balance sheet insolvency
- Continuous Pricing
  - Buying N RCs at once at the same price of buying 1 RC N times
- Dynamic Fees
  - Fees growing linearly for minting and redemption actions that move r away from r_{opt}
  - Greater robustness against oracle delays and against manipulation of the BC price

- Debt financing (bonds)
- Dividends
- Stablecoins pegged to other assets
- Reserves consisting of a basket of assets
- More Stability Simulations
- Formal Verification
- Governance and Updates
- KYC/AML
- Staking

Related Work

Seigniorage Shares

- Share holders profit from the contract like RC holders
- But profit comes from seigniorage, not market making fees
- Stability based on changing the supply of the stablecoin
- No backing

DAI

- Crypto-collateralization
  - Similar to crypto-backing
  - Collateral $\neq$ Reserve
- Could be interpreted as a bank with 0% reserves
- No DAI redemption (except by vault owner or in a vault liquidation auction)
  - SCs always redeemable by anyone, immediately
- Penalization for providing collateral ("stability fee")
  - RC holders are rewarded for providing reserves

Staticoin/Riskcoin

- Riskcoin similar to RC
- Riskcoin's focus is on the leverage that it provides over the BC
- No dilution protection for Riskcoin holders (r_{max})
- No reserve protection for Staticoin holders (r_{min})
- No fair pricing of staticoins when $r < 1$ ("bank runs")

Summary

- The presented stablecoin contract:
  - takes some of the responsibility and role of banks
  - behaves like a full-reserve ``bank-like” autonomous entity, using automatic:
    - *market making* for stabilization
    - *equity financing* for replenishing the reserve
    - *buy back* for rewarding reserve contributors

- The implementation in Ergo is, to the best of our knowledge, the first algorithmic stablecoin on a UTxO-based blockchain

Final Remark: Use with Caution!

- You are interacting with an **autonomous software entity** that *follows its rules blindly and irreversibly*.
- The best we (Ergo, Emurgo and Input Output) can do is to:
  - Make the rules be like the traditional rules that are already accepted by society and regulators
  - Explain these rules clearly to you
- Analogy with self-driving cars:
  - a self-driving car still needs to comply with traffic laws
  - but the builder/deployer of a self-driving car is not the car's driver (and ought not to need to have a driver's license)