REFERENCE IMPLEMENTATION

of the decentralized

DAI STABLECOIN

issuance system

Nikolai Mushegian
Daniel Brockman
Mikael Brockman

[draft; 2018-02-06]

Contents

1 Introduction

The **Dai stablecoin system** is a set of blockchain smart contracts designed to issue a collateral-backed token (called the dai) and subject its price to a decentralized stability mechanism.

This document is an executable technical specification of the system. It is a draft and will change before launch.

For an overview of the system, see the white paper.

For a "choose your own adventure" exploration of the system's mechanics, please wait for the interactive FAQ.

We are dedicated to providing material for new people to understand the system in depth. This will be important for successful governance in the project's future.

If you have any questions, ask on our chat or subreddit. Asking helps us work on our explanatory material, so we appreciate it.

1.1 Why a reference implementation?

The contracts that will be deployed on the Ethereum blockchain are prototyped in Solidity. This paper is a model of the system written as a Haskell program. The motivations for this include:

Comparison. Checking two free-standing implementations against each other is a well-known way of ensuring that they both behave as intended.

Verification. Haskell lets us use powerful testing tools such as QuickCheck for comprehensively verifying key properties. This is a middle ground between testing and formal verification.

Formality. The work of translating into a purely functional program opens up opportunities for formal verification. This document will be useful for modelling aspects of the system in a proof assistant like Isabelle.

Explicitness. Coding the contract behavior in Haskell, a statically typed functional language, enforces explicit description of aspects which Solidity leaves implicit.

Clarity. An implementation not intended to be deployed on the blockchain is free from concerns about optimizing for gas cost and other factors that make the Solidity implementation less ideal as an understandable specification.

Simulation. Solidity is specific to the blockchain environment and lacks facilities for interfacing with files or other programs. A reference implementation is useful for doing simulations of the system's economic, game-theoretic, or statistical aspects.

1.2 Formal verification and steps thereto

We are developing automatic test suites that generate interaction sequences for property verification.

One such property is that the reference implementation behaves like the on-chain implementation. We verify this by generating Solidity test cases with equality assertions for the entire state.

Other key properties include

- that the target price changes according to the target rate;
- that the total dai supply is fully accounted for;
- that actions are restricted with respect to CDP stage;

along with similar invariants and conditions. A future revision of this document will include formal statements of these properties.

1.3 Note on jargon

The reference implementation uses a concise vocabulary for system variables and actions.

This document has a glossary accessible through hovering over highlighted words.

Here are some of the motivations for this jargon:

- We sidestep terminological debates; for example, whether to say »rate of target price change« or »target rate«.
- With decoupled financial and technical vocabularies, we can more flexibly improve one without affecting the other.
- The ability to discuss the system formally, with the financial interpretation partly suspended, has suggested insights that would have been harder to think of inside the normal language.
- The precise and distinctive language makes the structure and logic of the implementation more apparent and easier to formalize.

- Concise names make the code less verbose and the concepts easier to handle on paper, whiteboard, etc.

2 Dai mechanics

> *Note: this section is incomplete. It is supposed to briefly and technically explain the explicit mechanics of the system with links to relevant definitions.*

The dai stablecoin system lets users lock collateral assets and issue dai in proportion to the collateral's market value. Thus they can deposit their valuable tokens in order to withdraw some quantity of stablecoin. Such a deposit account is called a "collateralized debt position" or CDP.

> *See lock, draw, and Urn.*

As long as such a deposit retains sufficient market value, the user may reclaim their deposit, partially or in whole, by paying back dai. As long as the CDP is collateralized in excess of the required ratio, the user can also decrease their collateralization by reclaiming part of the deposit without paying back dai.

> *See free and wipe.*

Governance decides which external tokens are valid as collateral, and creates different deposit classes, or "CDP types", each with different parameters such as maximum dai issuance, minimum collateral ratio, and so on.

> *See Ilk.*

For deciding collateral requirements, the system values the dai not at the market price, but at its own *target price*, which is adjusted by the stability mechanism.

See feel, which determines the lifecycle stage of a CDP.

The target price adjustment is a second order effect. Primarily, the stability mechanism reacts to market price changes by adjusting the *target rate*.

See prod, which updates the stability mechanism.

3 Preamble and data types

This program uses some symbols defined in external libraries. Most symbols should be clear in context, but our "prelude" lists and briefly explains each imported type and function.

[**TODO:** Render the prelude.]

```
module Maker where
import Prelude (); import Maker.Prelude; import Maker.Decimal
```

3.1 Numeric types

The system uses two precisions of decimal numbers, to which we have given short mnemonic names.

One is called **wad** and has 18 digits of precision. It is used for token quantities, such as amounts of ETH, DAI, or MKR.

The other is called **ray** and has 36 digits of precision. It is used for precise rates and ratios, such as the stability fee parameter.

We define these as distinct types. The type system will not allow us to combine them without explicit conversion.

```
newtype Wad = Wad (Decimal E18)
  deriving (Ord, Eq, Num, Real, Fractional, RealFrac)
newtype Ray = Ray (Decimal E36)
  deriving (Ord, Eq, Num, Real, Fractional, RealFrac)
```

We define a generic function for converting one of these types to the other.

```
cast x = fromRational (toRational x)  [Via fractional n/m form]
```

We also define a type for time durations in whole seconds.

```
newtype Sec = Sec Int
  deriving (Eq, Ord, Enum, Num, Real, Integral)
```

3.2 Identifiers and addresses

The following common Haskell idiom lets us use `Id Ilk` , `Id Urn` , and so on, as distinct identifier types.

```
newtype Id a = Id String
  deriving (Eq, Ord, Show)
```

We define another type for representing Ethereum account addresses.

```
newtype Address = Address String
  deriving (Eq, Ord, Show)
```

3.3 Gem, SIN, DAI, MKR: token identifiers

The system makes use of four basic types of tokens.

```
data Token
    -- Some collateral token approved by system governance
  = Gem (Id Tag)
    -- Fungible stablecoin, issued by CDP owners and traded publicly
  | DAI
    -- Internal anticoin whose quantity is always equal to total issued dai
  | SIN
    -- Volatile countercoin and voting token
  | MKR
  deriving (Eq, Ord, Show)
```

The system's approved collateral tokens are called "gems". We use the type `Id Tag` to denote the identity of some collateral token.

The model treats all collateral tokens as basic ERC20 tokens differing only in symbol. In reality, voters should make sure that tokens are well-behaved before approving them.

3.4 Tag: collateral token price record

The data received from price feeds is stored in `Tag` records.

```
data Tag = Tag {
  -- Latest token market price (denominated in SDR)
  · tag :: Wad,
```

```
    Timestamp after which price should be considered stale
    · zzz :: Sec

} deriving (Eq, Show)
```

3.5 Urn: CDP record

An `Urn` record keeps track of one CDP.

```
data Urn = Urn {

    CDP type identifier
    · ilk  :: Id Ilk,

    CDP owner
    · lad  :: Address,

    Amount of outstanding dai issued by this CDP, denominated in debt unit
    · art  :: Wad,

    Amount of collateral currently locked by this CDP
    · ink  :: Wad,

    Actor that triggered liquidation, if applicable
    · cat  :: Maybe Actor

} deriving (Eq, Show)
```

3.6 Ilk: CDP type record

An `Ilk` record keeps track of one CDP type.

```
data Ilk = Ilk {

    Token used as collateral for CDPs of this type
    · gem :: Id Tag,

    Total debt owed by CDPs of this type, denominated in debt unit
    · rum :: Wad,
```

Current dai value of debt unit, increasing according to stability fee

- **chi** :: Ray,

Debt ceiling: maximum total outstanding dai value that can be issued by this CDP type

- **hat** :: Wad,

Liquidation ratio (collateral value per dai value)

- **mat** :: Ray,

Liquidation penalty (fraction of dai)

- **axe** :: Ray,

Fee (per-second fraction of dai)

- **tax** :: Ray,

Grace period of price feed unavailability

- **lax** :: Sec,

Timestamp of latest debt unit adjustment

- **rho** :: Sec

} **deriving** (Eq, Show)

3.7 Vox: feedback mechanism record

The **feedback mechanism** is the aspect of the system that adjusts the target price of dai based on market price. Its data is grouped in a record called `Vox`.

data **Vox** = Vox {

Dai market price denominated in SDR

- **wut** :: Wad,

Dai target price denominated in SDR

- **par** :: Wad,

Current per-second change in target price

- **way** :: Ray,

Sensitivity parameter (set by governance)

- **how** :: Ray,

Timestamp of latest feedback iteration

- **tau** :: Sec

```
} deriving (Eq, Show)
```

3.8 Actor: account identifier

We use a data type to explicitly distinguish the different entities that can hold a token balance or invoke actions.

```
data Actor
    -- Extern address (CDP owner)
  = Account Address
    -- Collateral vault, holds all locked collateral until liquidation
  | Jar
    -- DAI and SIN are minted and burned by the "jug"
  | Jug
    -- The settler component
  | Vow
    -- The collateral auctioneer that raises DAI to cover liquidations
  | Flipper
    -- The "buy and burn" auctioneer that spends fee revenue on buying MKR
  | Flapper
    -- The "inflate and sell" auctioneer that mints MKR to cover liquidations
  | Flopper
    -- Test driver (not present in real system)
  | Toy
    -- Omnipotent actor (temporary kludge)
  | God
  deriving (Eq, Ord, Show)
```

3.9 System model

Finally we define the overall state of the model.

```
data System = System {
  Feedback mechanism data
  · vox      :: Vox,
  CDP records
  · urns     :: Map (Id Urn) Urn,
  CDP type records
  · ilks     :: Map (Id Ilk) Ilk,
  Price tags of collateral tokens
  · tags     :: Map (Id Tag) Tag,
  Token balances by actor and token
  · balances :: Map (Actor, Token) Wad,
  Current timestamp
  · era      :: Sec,
  Settler operation mode
  · mode     :: Mode,
  Sender of current action
  · sender   :: Actor,
  All user accounts (for tests)
  · accounts :: [Address]
} deriving (Eq, Show)
```

```
Settler-related work in progress
data Mode = Dummy
  deriving (Eq, Show)
```

4 Actions

The **actions** are the basic state transitions of the system.

Unless specified as **internal**, actions are accessible as public functions on the blockchain.

The `auth` modifier marks actions which can only be invoked from addresses to which the system has granted authority.

For details on the underlying »Action monad« which specifies how the action definitions behave with regard to state and rollback, see chapter \ref{chapter:monad}.

4.1 Issuance

Any user can open one or more accounts with the system using `open`, specifying a self-chosen account identifier and an ilk.

```
open id_urn id_ilk = do
  Fail if account identifier is taken
  none (urns ∘ ix id_urn)
  Fail if ilk type is not present
  _ ← look (ilks ∘ ix id_ilk)
  Create a CDP with the sender as owner
  Account id_lad ← use sender
  initialize (urns ∘ at id_urn) (emptyUrn id_ilk id_lad)
```

The owner of an urn can transfer its ownership at any time using `give`.

```
give id_urn id_lad = do
  Fail if sender is not the CDP owner
  id_sender ← use sender
  owns id_urn id_sender
  Transfer CDP ownership
  assign (urns ∘ ix id_urn ∘ lad) id_lad
```

Unless CDP is in liquidation, its owner can use **lock** to lock more collateral.

```
lock id_urn wad_gem = do
  Fail if sender is not the CDP owner
   id_lad ← use sender
   owns id_urn id_lad

  Fail if liquidation in process
   want (feel id_urn) (not ∘ oneOf [Grief, Dread])

  Identify collateral token
   id_ilk  ← look (urns ∘ ix id_urn ∘ ilk)
   id_tag  ← look (ilks ∘ ix id_ilk ∘ gem)

  Take custody of collateral
   transfer (Gem id_tag) wad_gem id_lad Jar

  Record an collateral token balance increase
   increase (urns ∘ ix id_urn ∘ ink) wad_gem
```

When a CDP has no risk problems (except that its ilk's ceiling may be exceeded), its owner can use **free** to reclaim some amount of collateral, as long as this would not take the CDP below its liquidation ratio.

```
free id_urn wad_gem = do
  Fail if sender is not the CDP owner
   id_lad ← use sender
   owns id_urn id_lad

  Record a collateral token balance decrease
   decrease (urns ∘ ix id_urn ∘ ink) wad_gem

  Roll back on any risk problem except ilk ceiling excess
   want (feel id_urn) (oneOf [Pride, Anger])

  Release custody of collateral
   id_ilk ← look (urns ∘ ix id_urn ∘ ilk)
   id_tag ← look (ilks ∘ ix id_ilk ∘ gem)
   transfer (Gem id_tag) wad_gem Jar id_lad
```

When a CDP has no risk problems, its owner can can use `draw` to issue fresh stablecoin, as long as the ilk ceiling is not exceeded and the issuance would not take the CDP below its liquidation ratio.

```
draw id_urn wad_dai = do
  Fail if sender is not the CDP owner
   id_lad ← use sender
   owns id_urn id_lad
  Update debt unit and unprocessed fee revenue
   id_ilk ← look (urns ∘ ix id_urn ∘ ilk)
   chi_1 ← drip id_ilk
  Denominate issuance quantity in debt unit
   let wad_chi = wad_dai / cast chi_1
  Record increase of CDP's stablecoin issuance
   increase (urns ∘ ix id_urn ∘ art) wad_chi
  Record increase of ilk's stablecoin issuance
   increase (ilks ∘ ix id_ilk ∘ rum) wad_chi
  Roll back on any risk problem
   want (feel id_urn) (== Pride)
  Mint both stablecoin and anticoin
   lend wad_dai
  Transfer stablecoin to CDP owner
   transfer DAI wad_dai Jug id_lad
```

An CDP owner who has previously issued stablecoin can use `wipe` to send back dai and reduce the CDP's issuance.

```
wipe id_urn wad_dai = do
  Fail if sender is not the CDP owner
   id_lad ← use sender
   owns id_urn id_lad
  Fail if CDP is in liquidation
   want (feel id_urn) (not ∘ oneOf [Grief, Dread])
  Update debt unit and unprocessed fee revenue
   id_ilk ← look (urns ∘ ix id_urn ∘ ilk)
   chi_1 ← drip id_ilk
```

```
  Denominate stablecoin amount in debt unit
  let wad_chi = wad_dai / cast chi_1

  Record decrease of CDP issuance
  decrease (urns ∘ ix id_urn ∘ art) wad_chi

  Record decrease of ilk total issuance
  decrease (ilks ∘ ix id_ilk ∘ rum) wad_chi

  Take custody of stablecoin from CDP owner
  transfer DAI wad_dai id_lad Jar

  Destroy stablecoin and anticoin
  mend wad_dai
```

An CDP owner can use `shut` to close their account, if the price feed is up to date and the CDP is not in liquidation. This reclaims all collateral and cancels all issuance plus fee.

```
shut id_urn = do
  Update debt unit and unprocessed fee revenue
  id_ilk ← look (urns ∘ ix id_urn ∘ ilk)
  chi_1 ← drip id_ilk

  Reverse all issued stablecoin plus fee
  art_0 ← look (urns ∘ ix id_urn ∘ art)
  wipe id_urn (art_0 * cast chi_1)

  Reclaim all locked collateral
  ink_0 ← look (urns ∘ ix id_urn ∘ ink)
  free id_urn ink_0

  Nullify CDP record
  assign (urns ∘ at id_urn) Nothing
```

4.2 Assessment

We define six stages of a CDP's lifecycle.

```
data Stage
```

```
    -- Overcollateralized, CDP type below debt ceiling, fresh price tag, liquidation not triggered
  = Pride

    -- Debt ceiling reached for CDP's type
  | Anger

    -- CDP type's collateral price feed in limbo
  | Worry

    -- CDP undercollateralized, or CDP type's price limbo grace period exceeded
  | Panic

    -- Liquidation triggered
  | Grief

    -- Liquidation triggered and started
  | Dread

  deriving (Eq, Show)
```

4.2.1 Lifecycle stage effects

The following table shows which CDP actions are allowed and prohibited in each stage of the CDP lifecycle.

					decrease collateral				
	give	shut	lock	wipe	free	draw	bite	grab	plop
Pride	■	■	■	■	■	■			
Anger	■	■	■	■	■				
Worry	■	■	■	■					
Panic	■	■	■	■			■		
Grief	■							■	
Dread	■								■
	give	shut	lock	wipe	free	draw	bite	grab	plop
		increase collateral					liquidation		

Some implications:

- Collateral-increasing actions are allowed until `Grief` .

- To draw is only allowed during Pride, while free is also allowed during Anger.
- To give is allowed at any time, including during liquidation.
- Each of the liquidation actions corresponds to its own stage.

4.2.2 CDP stage analysis

We define the function analyze that determines the lifecycle stage of a CDP.

```
analyze era₀ par₀ urn₀ ilk₀ tag₀ =
  let
    Value of urn's locked collateral in SDR:
    pro = view ink urn₀ * view tag tag₀
    CDP's issuance denominated in SDR:
    con = view art urn₀ * cast (view chi ilk₀) * par₀
    Required collateral value as per liquidation ratio:
    min = con * cast (view mat ilk₀)
    CDP type's total DAI issuance:
    cap = view rum ilk₀ * cast (view chi ilk₀)
  in if
    Cases checked in order:
    | has cat urn₀ && view ink urn₀ == 0      → Dread
    | has cat urn₀                            → Grief
    | pro < min                               → Panic
    | view zzz tag₀ + view lax ilk₀ < era₀    → Panic
    | view zzz tag₀ < era₀                    → Worry
    | cap > view hat ilk₀                     → Anger
    | otherwise                               → Pride
```

Now we define the internal act feel which returns the value of analyze after ensuring that the system state is updated.

```
feel id_urn = do
```

```
Adjust target price and target rate
 prod

Update debt unit and unprocessed fee revenue
 id_ilk ← look (urns ∘ ix id_urn ∘ ilk)
 drip id_ilk

Read parameters for stage analysis
 era0 ← use era
 par0 ← use (vox ∘ par)
 urn0 ← look (urns ∘ ix id_urn)
 ilk0 ← look (ilks ∘ ix (view ilk urn0))
 tag0 ← look (tags ∘ ix (view gem ilk0))

Return lifecycle stage of CDP
 return (analyze era0 par0 urn0 ilk0 tag0)
```

CDP actions use `feel` to prohibit increasing risk when already risky, and to freeze stablecoin and collateral during liquidation.

4.3 Adjustment

The feedback mechanism is updated through `prod`, which can be invoked at any time by keepers, but is also invoked as a side effect of any CDP act that uses `feel` to assess risk.

```
prod = do

 Read all parameters relevant for feedback mechanism
  era0 ← use era
  tau0 ← use (vox ∘ tau)
  wut0 ← use (vox ∘ wut)
  par0 ← use (vox ∘ par)
  how0 ← use (vox ∘ how)
  way0 ← use (vox ∘ way)

  let
   Time difference in seconds
    age  = era0 - tau0

   Current target rate applied to target price
    par1  = par0 * cast (way0 ^^ age)
```

Sensitivity parameter applied over time

```
wag   = how₀ * fromIntegral age
```

Target rate scaled up or down

```
way₁  = inj (prj way₀ +
             if wut₀ < par₀ then wag else -wag)
```

Update target price

```
assign (vox ∘ par) par₁
```

Update rate of price change

```
assign (vox ∘ way) way₁
```

Record time of update

```
assign (vox ∘ tau) era₀
```

```
where
```

Convert between multiplicative and additive form

```
prj x  = if x >= 1  then x - 1  else 1 - 1 / x
inj x  = if x >= 0  then x + 1  else 1 / (1 - x)
```

The stability fee of an ilk can change through governance. Due to the constraint that acts should run in constant time, the system cannot iterate over CDPs to effect such changes. Instead each ilk has a single »debt unit« which accumulates the stability fee. The `drip` act updates this unit. It can be called at any time by keepers, but is also called as a side effect of every act that uses `feel` to assess CDP risk.

```
drip id_ilk = do
  rho₀  ← look (ilks ∘ ix id_ilk ∘ rho)
  tax₀  ← look (ilks ∘ ix id_ilk ∘ tax)
  chi₀  ← look (ilks ∘ ix id_ilk ∘ chi)
  rum₀  ← look (ilks ∘ ix id_ilk ∘ rum)
  era₀  ← use era
  let
```

Time difference in seconds

```
age   = era₀ - rho₀
```

Value of debt unit increased according to stability fee

```
chi₁  = chi₀ * tax₀ ^^ age
```

Stability fee revenue denominated in new unit

```
dew   = (cast (chi₁ - chi₀) :: Wad) * rum₀
```

Mint stablecoin and anticoin for marginally accrued fee

```
lend dew
```

Record time of update

```
assign (ilks ∘ ix id_ilk ∘ rho) era_0
```

Record new debt unit

```
assign (ilks ∘ ix id_ilk ∘ chi) chi_1
```

Return the new debt unit

```
return chi_1
```

4.4 Price feed input

The mark act records a new market price of an collateral along with the expiration date of this price.

```
mark id_gem tag_1 zzz_1 = auth $ do
  initialize (tags ∘ at id_gem) Tag {
    · tag  = tag_1,
    · zzz  = zzz_1
  }
```

The tell act records a new market price of dai along with the expiration date of this price.

```
tell wad = auth $ do
  assign (vox ∘ wut) wad
```

4.5 Liquidation

When a CDP's stage marks it as in need of liquidation, any account can invoke the bite act to trigger the liquidation process. This enables the settler contract to grab the collateral for auctioning and take over the anticoin.

```
bite id_urn = do
  -- Fail if CDP is not in the appropriate stage
  want (feel id_urn) (== Panic)

  -- Record the sender as the liquidation initiator
  id_cat ← use sender
  assign (urns ∘ ix id_urn ∘ cat) (Just id_cat)

  -- Apply liquidation penalty to CDP issuance
  id_ilk ← look (urns ∘ ix id_urn ∘ ilk)
  axe_0 ← look (ilks ∘ ix id_ilk ∘ axe)
  art_0 ← look (urns ∘ ix id_urn ∘ art)
  let art_1 = art_0 * cast axe_0

  -- Update CDP issuance to include penalty
  assign (urns ∘ ix id_urn ∘ art) art_1
```

After liquidation has been triggered, the designated settler contract invokes `grab` to receive both the CDP's collateral and the anticoins corresponding to the CDP's issuance.

```
grab id_urn = auth $ do
  -- Fail if CDP is not marked for liquidation
  want (feel id_urn) (== Grief)

  ink_0 ← look (urns ∘ ix id_urn ∘ ink)
  art_0 ← look (urns ∘ ix id_urn ∘ art)
  id_ilk ← look (urns ∘ ix id_urn ∘ ilk)
  id_tag ← look (ilks ∘ ix id_ilk ∘ gem)

  -- Update the debt unit and unprocessed fee revenue
  chi_1 ← drip id_ilk

  -- Denominate the issuance in dai
  let con = art_0 * cast chi_1

  -- Transfer collateral and anticoin to settler
  transfer (Gem id_tag) ink_0 Jar Vow
  transfer SIN con Jug Vow

  -- Nullify CDP's collateral and anticoin quantities
  assign (urns ∘ ix id_urn ∘ ink) 0
  assign (urns ∘ ix id_urn ∘ art) 0
```

Decrease the ilk's total issuance

```
decrease (ilks ∘ ix id_ilk ∘ rum) art_0
```

When the settler has finished the liquidation of a CDP, it invokes `plop` to give back any collateral it did not need to sell and restore the CDP.

```
plop id_urn wad_dai = auth $ do
```

Fail unless CDP is in the proper stage

```
want (feel id_urn) (== Dread)
```

Forget the CDP's initiator of liquidation

```
assign (urns ∘ ix id_urn ∘ cat) Nothing
```

Take excess collateral from settler to vault

```
id_vow ← use sender
id_ilk ← look (urns ∘ ix id_urn ∘ ilk)
id_tag ← look (ilks ∘ ix id_ilk ∘ gem)
transfer (Gem id_tag) wad_dai id_vow Jar
```

Record the excess collateral as belonging to the CDP

```
assign (urns ∘ ix id_urn ∘ ink) wad_dai
```

The settler can invoke `loot` at any time to claim all uncollected stability fee revenue for use in the countercoin buy-and-burn auction.

```
loot = auth $ do
```

The dai vault's balance is the uncollected stability fee revenue

```
wad ← look (balance DAI Jug)
```

Transfer the entire dai vault balance to sender

```
transfer DAI wad Jug Vow
```

4.6 Auctioning

Note: this section is incomplete; all auctions are dummies.

```
flip id_gem wad_jam wad_tab id_urn = do
  vow ← look mode
  case vow of
    Dummy → return ()


flap = do
  vow ← look mode
  case vow of
    Dummy → return ()
flop = do
  vow ← look mode
  case vow of
    Dummy → return ()
```

4.7 Settlement

```
tidy who = auth $ do
```

Find the entity's stablecoin and anticoin balances

```
  awe ← look (balance DAI who)
  woe ← look (balance SIN who)
```

We can burn at most the smallest of the two balances

```
  let x = min awe woe
```

Transfer stablecoin and anticoin to the settler

```
  transfer DAI x who Vow
  transfer SIN x who Vow
```

Burn both stablecoin and anticoin

```
  burn DAI x Vow
  burn SIN x Vow
```

```
kick = do
```

Transfer unprocessed stability fee revenue to vow account

```
  loot
```

Cancel stablecoin against anticoin

```
  tidy Vow
```

Assign any remaining stablecoin to countercoin-deflating auction

```
transferAll DAI Vow Flapper
flap
```

Assign any remaining anticoin to countercoin-inflating auction

```
transferAll SIN Vow Flopper
flop
```

4.8 Governance

Governance uses `form` to create a new ilk. Since the new type is initialized with a zero ceiling, a separate transaction can safely set the risk parameters before any issuance occurs.

```
form id_ilk id_gem = auth $ do
  initialize (ilks ∘ at id_ilk) (defaultIlk id_gem)
```

Governance uses `frob` to alter the sensitivity factor, which is the only mutable parameter of the feedback mechanism.

```
frob how_1 = auth $ do
  assign (vox ∘ how) how_1
```

Governance can alter the five risk parameters of an ilk using `cuff` for the liquidation ratio; `chop` for the liquidation penalty; `cork` for the ilk ceiling; `calm` for the duration of price limbo; and `crop` for the stability fee.

```
cuff id_ilk mat_1 = auth $ do
  assign (ilks ∘ ix id_ilk ∘ mat) mat_1

chop id_ilk axe_1 = auth $ do
  assign (ilks ∘ ix id_ilk ∘ axe) axe_1

cork id_ilk hat_1 = auth $ do
  assign (ilks ∘ ix id_ilk ∘ hat) hat_1
```

```
calm id_ilk lax_1 = auth $ do
  assign (ilks ∘ ix id_ilk ∘ lax) lax_1
```

When altering the stability fee with `crop`, we ensure that the previous stability fee has been accounted for in the internal debt unit.

```
crop id_ilk tax_1 = auth $ do
  -- Apply the current stability fee to the internal debt unit
  drip id_ilk
  -- Change the stability fee
  assign (ilks ∘ ix id_ilk ∘ tax) tax_1
```

4.9 Token manipulation

We model the ERC20 transfer function in simplified form (omitting the concept of »allowance«).

```
transfer id_gem wad src dst =
  -- Operate in the token's balance table
  zoom balances $ do
  -- Fail if source balance insufficient
  balance ← look (ix (src, id_gem))
  aver (balance >= wad)
  -- Update balances
  decrease    (ix (src, id_gem)) wad
  initialize  (at (dst, id_gem)) 0
  increase    (ix (dst, id_gem)) wad
```

```
transferAll id_gem src dst = do
  wad ← look (balance id_gem src)
  transfer id_gem wad src dst
```

The internal act **mint** inflates the supply of a token. It is used by **lend** to create new stablecoin and anticoin, and by the settler to create new countercoin.

```
mint idgem wad dst = do
  initialize (balances ∘ at (dst, idgem)) 0
  increase   (balances ∘ ix (dst, idgem)) wad
```

The internal act **burn** deflates the supply of a token. It is used by **mend** to destroy stablecoin and anticoin, and by the settler to destroy countercoin.

```
burn idgem wad src =
  decrease (balances ∘ ix (src, idgem)) wad
```

The internal act **lend** mints identical amounts of both stablecoin and anticoin. It is used by **draw** to issue stablecoin; it is also used by **drip** to issue stablecoin representing revenue from stability fees, which stays in the vault until collected.

```
lend waddai = do
  mint DAI waddai Jug
  mint SIN waddai Jug
```

The internal act **mend** destroys identical amounts of both dai and the internal debt token. Its use via **wipe** is how the stablecoin supply is reduced.

```
mend waddai = do
  burn DAI waddai Jug
  burn SIN waddai Jug
```

5 Default data

```
defaultIlk :: Id Tag → Ilk
defaultIlk idtag = Ilk {
  · gem = idtag,
  · axe = Ray 1,
  · mat = Ray 1,
  · tax = Ray 1,
  · hat = Wad 0,
  · lax = Sec 0,
  · chi = Ray 1,
  · rum = Wad 0,
  · rho = Sec 0
}
```

```
emptyUrn :: Id Ilk → Address → Urn
emptyUrn idilk idlad = Urn {
  · cat = Nothing,
  · lad = idlad,
  · ilk = idilk,
  · art = Wad 0,
  · ink = Wad 0
}
```

```
initialTag :: Tag
initialTag = Tag {
  · tag = Wad 0,
  · zzz = 0
}
```

```
initialSystem :: Ray → System
initialSystem how0 = System {
  · balances = empty,
  · ilks     = empty,
  · urns     = empty,
  · tags     = empty,
  · era      = 0,
  · sender   = God,
  · accounts = mempty,
```

```
    · mode     = Dummy,
    · vox      = Vox {
      · tau = 0,
      · wut = Wad 1,
      · par = Wad 1,
      · how = how₀,
      · way = Ray 1
    }
  }
```

6 Action framework

The reader does not need any abstract understanding of monads to understand the code. They provide syntax (the **do** notation) for expressing exceptions and state in a way that is still purely functional. Each line of such a block is interpreted by the monad to provide the semantics we want.

6.1 The Action monad

This defines the `Action` monad as a simple composition of a state monad and an error monad:

```
type Action a = StateT System (Except Error) a
```

We divide act failure modes into general assertion failures and authentication failures.

```
data Error = AssertError Act | AuthError
  deriving (Show, Eq)
```

An act can be executed on a given initial system state using `exec` . The result is either an error or a new state. The `exec` function can also accept a sequence of actions, which will be interpreted as a single transaction.

```
exec :: System → Action () → Either Error System
exec sys m = runExcept (execStateT m sys)
```

6.2 Asserting

We now define a set of functions that fail unless some condition holds.

General assertion

```
aver x = unless x (throwError (AssertError ?act))
```

Assert that an indexed value is not present

```
none x = preuse x >>= \case
  Nothing → return ()
  Just _  → throwError (AssertError ?act)
```

Assert that an indexed value is present

```
look f = preuse f >>= \case
  Nothing → throwError (AssertError ?act)
  Just x  → return x
```

Execute an act and assert a condition on its result

```
want m p = m >>= (aver ∘ p)
```

```
has p x = view p x /= Nothing
```

We define `owns` id_{urn} id`lad` as an assertion that the given CDP is owned by the given account.

```
owns id_urn id_lad =
  want (look (urns ∘ ix id_urn ∘ lad)) ((== id_lad) ∘ Account)
```

We define `auth k` as an act modifier that executes `k` only if the sender is authorized.

```
auth continue = do
  s ← use sender
  unless (s == God) (throwError AuthError)
  continue
```

7 Glossary

Id a (for any a) is a string identifier that cannot be mixed up with some other Id b.

Urn is the data record for a CDP.

Ilk is the data record for a CDP type.

The **lad** of an Urn (type: Actor) is the account identifier of the owner of that CDP.

The **art** of an Urn (type: Wad) is the amount of outstanding dai issued by that CDP.

The **ink** of an Urn (type: Wad) is the quantity of collateral locked in the corresponding CDP.

The **cat** of an Urn is the actor which triggered the CDP's liquidation, if applicable.

The **gem** of an Ilk is the collateral token used for collateral in the corresponding CDP type.

The **tax** of an Ilk (type: Ray) is the stability fee imposed on CDPs of the corresponding CDP type, expressed as a per-second fraction of the CDP's outstanding dai.

The **lax** of an Ilk (type: Sec) is the grace period for expired collateral price tags applying to CDPs of the corresponding type.

The **hat** of an Ilk (type: Wad) is the maximum total ("ceiling") dai issuance for the corresponding CDP type.

The **rum** of an Ilk (type: Wad) is the total current issuance for the corresponding CDP type, denominated in the CDP's internal debt unit.

The **chi** of an Ilk (type: Ray) is the dai valuation for the corresponding CDP type's internal debt unit, compounding over time according to the CDP type's stability fee.

The **mat** of an Ilk (type: Ray) is the minimum required collateralization ratio (value of collateral divided by value of issued dai) for CDPs of the corresponding CDP type.

The **axe** of an Ilk (type: Ray) is the penalty imposed on liquidated CDPs of the corresponding CDP type, expressed as a fraction of the CDP's outstanding dai.

The **rho** of an Ilk (type: Sec) is the timestamp of its latest debt unit adjustment.

The **tag** of a Tag record (type: Wad) is the recorded market price of the corresponding collateral token denominated in SDR.

The **zzz** of a Tag (type: Sec) is the timestamp at which the corresponding collateral price tag will expire.

Pride is the risk stage of a non-risky CDP.

Anger is the Stage of a CDP whose type has reached its debt ceiling, but has a fresh price feed, is overcollateralized, and has not been triggered for liquidation.

Worry is the Stage of a CDP whose collateral price feed has expired yet is still within the CDP type's grace period; but the CDP is still considered overcollateralized and has not been triggered for liquidation. (The CDP's type may also have reached its debt ceiling.)

Panic is the Stage of a CDP which is undercollateralized or whose price feed is expired past the CDP type's grace period; but which has not yet been triggered for liquidation. (The CDP's type may also have reached its debt ceiling.)

Grief is the Stage of a CDP which has been triggered for liquidation.

Dread is the Stage of a CDP which is undergoing liquidation.

has k x is true if the field k of the record x is not Nothing.

Wad is the type of a decimal number with 18 decimals of precision, used for token quantities.

Ray is the type of a decimal number with 36 decimals of precision, used for precise rates and ratios.

Sec is the type of a timestamp or duration in whole seconds.

cast x converts x to whatever numeric type is required in the expression context, possibly losing precision.

Address represents an arbitrary Ethereum account address.

Token identifies an ERC20 token used by the system: either some Gem (a collateral token) or one of SIN, DAI, or MKR.

Gem is a constructor for a Token representing a collateral token.

DAI is the identifier of the dai stablecoin token.

MKR is the identifier of the MKR token (the countercoin and governance token).

SIN is the identifier of the internal "anticoin" token which is always minted and burned in the same amounts as dai, only kept within the system as an accounting quantity.

wut (type: Wad) is the feedback mechanism's latest market price of dai, denominated in SDR.

par (type: Wad) is the feedback mechanism's latest target price of dai, denominated in SDR.

way (type: Ray) is the current per-second change in target price, continuously altered by the feedback mechanism according to the sensitivity parameter.

how (type: Ray) is the sensitivity parameter of the feedback mechanism, set by governance, controlling the rate of change of the dai target price.

tau (type: Sec) is the timestamp of the latest feedback mechanism iteration.

Tag is the record of collateral price feed updates. The type Id Tag is used to identify collateral tokens (aka Gems).

Vox is the record of feedback mechanism data.

Actor represents the identity of an entity which can hold a token balance or perform system actions.

Account (type: Address → Actor) constructs an Actor identifier denoting an external Ethereum account.

Jar (type: Actor) identifies the system's collateral vault.

Jug (type: Actor) identifies the actor that mints DAI/SIN and holds SIN.

Vow (type: Actor) identifies the system's settler component.

Flipper (type: Actor) identifies the collateral auctioneer component.

Flapper (type: Actor) identifies the DAI stablecoin auctioneer component.

Flopper (type: Actor) identifies the MKR countercoin auctioneer component.

Toy (type: Actor) identifies the system's test driver (not present in production).

God (type: Actor) identifies an omnipotent actor (prototyping kludge, will be removed).

lock transfers collateral from a CDP owner to the system's token vault and records an increase of ink to the CDP's Urn.

draw mints new DAI for the owner of an overcollateralized CDP.

give transfers ownership of a CDP.

free reclaims collateral from an overcollateralized CDP.

prod updates the stability feedback mechanism. It adjusts the target price (par) according to the target rate (way), and adjusts the target rate according to the current market price (wut) and the sensitivity parameter (how).

feel calculates the Stage of a CDP. This involves deciding collateralization requirements as well as checking price feed status and liquidation progress.