presents the

REFERENCE IMPLEMENTATION

of the remarkable

DAI CREDIT SYSTEM

issuing a diversely collateralized stablecoin

with last update on March 6, 2017.

Contents

Chapter 1

Introduction

The DAI CREDIT SYSTEM, henceforth also "Maker," is a network of Ethereum contracts designed to issue the DAI currency token and automatically adjust incentives in order to keep dai market value stable relative to SDR[1] in the short and medium term.

New dai enters the money supply when a borrower takes out a loan backed by an excess of collateral locked in Maker's token vault. The debt and collateral amounts are recorded in a *collateralized debt position*, or CDP. Thus all outstanding dai represents some CDP owner's claim on their collateral.

Maker's knowledge of the market values of dai and the various tokens used as collateral comes from *price feeds*. Prices are used to continuously assess the risk of each CDP. If the value of a CDP's collateral drops below a certain multiple of its debt, it is marked for liquidation, which triggers a decentralized auction mechanism.

Another token, MKR, is also controlled by Maker, acting as a "share" in the system itself. When a CDP liquidation fails to recover the full value of debt, Maker mints more MKR and auctions it out. Thus MKR is used to fund last resort market making. The value of the MKR token is based on the *stability fee* imposed on all dai loans: stability fee revenue goes toward buying MKR for burning.

This document is an executable technical specification of the exact workings of the Maker smart contracts.

[1] "Special Drawing Rights" (ticker symbol XDR), the international reserve asset created by the International Monetary Fund, whose value is derives from a weighted basket of world currencies. In the long term, the value of dai may diverge from the value of SDR; whether in an inflationary or deflationary way will depend on market forces.

1.1 Reference implementation

The version of this system that will be deployed on the Ethereum blockchain is written in Solidity, which is a workable smart contract implementation language. This reference implementation is a precise model of the behavior of those contracts, written as a "literate" Haskell program. The motivations for such a reference implementation include:

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

2. **Testing.** Haskell lets us use flexible and powerful testing tools such as QuickCheck and SmallCheck for comprehensively verifying key properties as a middle ground between unit testing and formal verification.

3. **Explicitness.** Coding the contract behavior in Haskell, a purely functional language, enforces explicit description of aspects which Solidity leaves implicit. For example, a Solidity program can read a previously unwritten mapping and get back a value initialized with zeroed memory, whereas in Haskell we must explicitly describe default values. The state rollback behavior of failed actions is also in Haskell explicitly coded as part of the monad transformer stack.

4. **Type correctness.** While Solidity does have a static type system, it is not expressive enough to encode the distinctions made by our system. In particular, the two different decimal fixed point number types that we use are typed in Solidity with one and the same `uint128` type. In Haskell we can make this distinction explicit.

5. **Formality.** The work of translating a Solidity program into a purely functional program opens up opportunities for certain types of formal verification. In particular, this document will be useful for modelling aspects of the system in a proof assistant like Agda, Idris, Coq, or Isabelle. We can also use logical tools for Haskell, such as Liquid Haskell (which provides compile time logical property checking) and `sbv` (a toolkit for model checking and symbolic execution).

6. **Simulation.** Solidity is highly specific to the Ethereum blockchain environment and as such does not have facilities for interfacing with files or other computer programs. This makes the Solidity implementation of the system less useful for doing simulations of the system's economic, game-theoretic, or statistical aspects.

1.2 Prerequisite Haskell knowledge

Some parts of this document require specific knowledge about Haskell programming, but these parts only make up a framework for expressing the more interesting parts in a natural way free of boilerplate.

◊ *Guidelines for skipping boring chapters and so on...*

For a complete understanding of the reference implementation's source code, the reader should grasp the following Haskell patterns:

- The use of **newtype** wrappers to distinguish different types of values which have the same underlying type.
- The use of **do** notation with the standard monad transformers:
  - `StateT` for updating state,
  - `ReaderT` for the read-only environment,
  - `WriterT` for "write-only state" (namely logs), and
  - `ExceptT` for failures which roll back state changes.
- The basic use of "lenses" (via the `lens` library) for convenient reading and writing of specific parts of nested values.
- The use of "parametricity" to express type-level guarantees about how function parameters are used, especially for understanding Appendix A which uses type signatures to specify which parts of the system are used or altered by each system action.
- ◊ *Some more stuff here...*

Part I

Implementation

Chapter 2

Preamble

We declare the program's dependencies up front. The reader should probably skim this section and consult it later if unfamiliar with some type or function.

```
module Maker where
```

We use a typical composition of monad transformers from the `mtl` library to structure stateful actions. This becomes relevant in section 4.2 (*The Maker monad*).

```
import Control.Monad.State (
  MonadState,     Type class of monads with state
  StateT,         Type constructor that adds state to a monad type
  execStateT,     Runs a state monad with given initial state
  get,            Gets the state in a do block
  put)            Sets the state in a do block
import Control.Monad.Reader (
  MonadReader,    Type class of monads with "environments"
  ask,            Reads the environment in a do block
  local)          Runs a sub-computation with a modified environment
import Control.Monad.Writer (
  MonadWriter,    Type class of monads that emit logs
  WriterT,        Type constructor that adds logging to a monad type
  runWriterT)     Runs a writer monad
import Control.Monad.Except (
  MonadError,     Type class of monads that fail
  Except,         Type constructor of failing monads
  throwError,     Short-circuits the monadic computation
  runExcept)      Runs a failing monad
```

Our numeric types use decimal fixed-point arithmetic.

```
import Data.Fixed (
  Fixed,                Type constructor for fixed-point numbers of given precision
  HasResolution (..))   Type class for specifying precisions
```

We rely on the `lens` library for accessing nested values. There is no need to understand the theory behind lenses to understand this program. The notation $a \circ b \circ c$ denotes a nested accessor much like `a.b.c` in C-style languages; for more details, consult lens documentation[1].

```
import Control.Lens (
  Lens',
  lens,
  makeFields,           Defines lenses for record fields
  set,                  Writes a lens
  view, preview,        Reads a lens in a do block
  (&~),                 Lets us use a do block with setters ◊ Get rid of this.
  ix,                   Lens for map retrieval and updating
  at,                   Lens for map insertion
  alongside,

  Operators for partial state updates in do blocks:
  (:=),                 Replace
  (-=), (+=), (*=),     Update arithmetically
  (%=),                 Update according to function
  (?=))                 Insert into map
```

Where the Solidity code uses `mapping`, we use Haskell's regular tree-based map type[2].

```
import Data.Map (
  Map,          Type constructor for mappings
  ∅,            Polymorphic empty mapping
  singleton)    Creates a mapping with a single key–value pair
```

For sequences of log entries, we use a sequence structure which has better time complexity than regular lists.

```
import           Data.Sequence (Seq)
import qualified Data.Sequence as Sequence
```

Some less interesting imports are omitted from this document.

[1] Gabriel Gonzalez's 2013 article *Program imperatively using Haskell* is a good introduction.

[2] We assume the axiom that Keccak hash collisions are impossible.

Chapter 3

Types

3.1 Numeric types

Many Ethereum tokens (e.g. ETH, DAI, and MKR) are denominated with 18 decimals. That makes decimal fixed point with 18 digits of precision a natural choice for representing currency quantities. We call such quantities "wads" (as in "wad of cash").

For some quantities, such as the rate of deflation per second, we want as much precision as possible, so we use twice the number of decimals. We call such quantities "rays" (mnemonic "rate," but also imagine a very precisely aimed ray of light).

```
Dummy types for specifying precisions
data E18; data E36

Specify 10^-18 as the precision of E18
instance HasResolution E18 where
  resolution _ = 10 ↑ (18 :: Integer)

Specify 10^-36 as the precision of E36
instance HasResolution E36 where
  resolution _ = 10 ↑ (36 :: Integer)

Create the distinct wad type for currency quantities
newtype Wad = Wad (Fixed E18)
  deriving (Ord, Eq, Num, Real, Fractional)

Create the distinct ray type for precise rate quantities
newtype Ray = Ray (Fixed E36)
  deriving (Ord, Eq, Num, Real, Fractional)
```

In calculations that combine `wad`s and `ray`s, we have to convert between the number types. Haskell does not convert numbers automatically, so when we explicitly need it, we use a *cast* function.

```
Convert via fractional n/m form.
cast :: (Real a, Fractional b) ⇒ a → b
cast = fromRational ∘ toRational
```

We also define a type for non-negative integers.

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

3.2 Identifier type

There are several kinds of identifiers used in the system, and we can use types to distinguish them.

```
The type parameter a creates distinct types.
For example, Id Foo and Id Bar are incompatible.
data Id a = Id String
  deriving (Show, Eq, Ord)
```

We will often use mappings from IDs to the value type corresponding to that ID type, so we define an alias for such mappings.

```
type IdMap a = Map (Id a) a
```

3.3 Domain types

This section introduces the records stored by the Maker system. The order of presentation is by use; types further down refer to types further up, but not the other way around.

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

We also have three predefined entities:

```
  The DAI token address
id_DAI = Id "Dai"
  The CDP engine address
id_vat = Address "Vat"
  The account with ultimate authority
  ◊ Kludge until authority is modelled
id_god = Address "God"


data Gem =
  Gem {
    gemTotalSupply :: !Wad,
    gemBalanceOf   :: !(Map Address Wad),
    gemAllowance   :: !(Map (Address, Address) Wad)
  } deriving (Eq, Show)
makeFields '' Gem


data Jar = Jar {
    Collateral token
    jarGem :: !Gem,
    Market price
    jarTag  :: !Wad,
    Price expiration
    jarZzz  :: !Nat
  } deriving (Eq, Show)
makeFields '' Jar


data Ilk = Ilk {
    Collateral vault
    ilkJar  :: !(Id Jar),
    Liquidation penalty
    ilkAxe :: !Ray,
    Debt ceiling
    ilkHat :: !Wad,
```

```
    Liquidation ratio
      ilkMat :: !Ray,
    Stability fee
      ilkTax :: !Ray,
    Limbo duration
      ilkLag :: !Nat,
    Last dripped
      ilkRho :: !Nat,
    Total debt in dai
      ilkDin :: !Wad,
    Price of debt coin
      ilkChi :: !Ray
  } deriving (Eq, Show)
makeFields '' Ilk


data Urn = Urn {
    Address of biting cat
      urnCat  :: !(Maybe Address),
    Address of liquidating vow
      urnVow :: !(Maybe Address),
    Issuer
      urnLad  :: !Address,
    CDP type
      urnIlk   :: !(Id Ilk),
    Outstanding debt in debt coins
      urnArt  :: !Wad,
    Collateral amount in debt coins
      urnJam :: !Wad
  } deriving (Eq, Show)
makeFields '' Urn


data Vat = Vat {
    Market price
      vatFix  :: !Wad,
```

```
    Sensitivity
      vatHow :: !Ray,
    Target price
      vatPar  :: !Wad,
    Target rate
      vatWay :: !Ray,
    Last prodded
      vatTau  :: !Nat,
    Unprocessed revenue from stability fees
      vatPie  :: !Wad,
    Bad debt from liquidated CDPs
      vatSin  :: !Wad,
    Collateral tokens
      vatJars :: !(IdMap Jar),
    CDP types
      vatIlks  :: !(IdMap Ilk),
    CDPs
      vatUrns :: !(IdMap Urn)
  } deriving (Eq, Show)
makeFields '' Vat


data System =
  System {
    systemVat    :: Vat,
    systemEra    :: Nat,
    systemSender :: Address
  } deriving (Eq, Show)
makeFields '' System
```

3.4 Default data

```
defaultIlk :: Id Jar → Ilk
defaultIlk id_jar = Ilk {
  ilkJar  = id_jar,
```

$$
\begin{aligned}
&\quad ilkAxe = \texttt{Ray}\ 1,\\
&\quad ilkMat = \texttt{Ray}\ 1,\\
&\quad ilkTax = \texttt{Ray}\ 1,\\
&\quad ilkHat = \texttt{Wad}\ 0,\\
&\quad ilkLag = \texttt{Nat}\ 0,\\
&\quad ilkChi = \texttt{Ray}\ 1,\\
&\quad ilkDin = \texttt{Wad}\ 0,\\
&\quad ilkRho = \texttt{Nat}\ 0\\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&defaultUrn :: \text{Id}\ \texttt{Ilk} \to \text{Address} \to \texttt{Urn}\\
&defaultUrn\ id_{\texttt{ilk}}\ id_{\texttt{lad}} = \texttt{Urn}\ \{\\
&\quad urnVow = \text{Nothing},\\
&\quad urnCat = \text{Nothing},\\
&\quad urnLad = id_{\texttt{lad}},\\
&\quad urnIlk = id_{\texttt{ilk}},\\
&\quad urnArt = \texttt{Wad}\ 0,\\
&\quad urnJam = \texttt{Wad}\ 0\\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&initialVat :: \texttt{Ray} \to \texttt{Vat}\\
&initialVat\ \texttt{how}_0 = \texttt{Vat}\ \{\\
&\quad vatTau = 0,\\
&\quad vatFix = \texttt{Wad}\ 1,\\
&\quad vatPar = \texttt{Wad}\ 1,\\
&\quad vatHow = \texttt{how}_0,\\
&\quad vatWay = \texttt{Ray}\ 1,\\
&\quad vatPie = \texttt{Wad}\ 0,\\
&\quad vatSin = \texttt{Wad}\ 0,\\
&\quad vatIlks = \varnothing,\\
&\quad vatUrns = \varnothing,\\
&\quad vatJars =\\
&\qquad singleton\ id_{\text{DAI}}\ \texttt{Jar}\ \{\\
&\qquad\quad jarGem = \texttt{Gem}\ \{\\
&\qquad\qquad gemTotalSupply = 0,\\
&\qquad\qquad gemBalanceOf = \varnothing,\\
&\qquad\qquad gemAllowance = \varnothing\\
&\qquad\quad \},\\
&\qquad\quad jarTag = \texttt{Wad}\ 0,\\
&\qquad\quad jarZzz = 0
\end{aligned}
$$

```
    }
}
```

```
initialSystem :: Ray → System
initialSystem how_0 = System {
  systemVat    = initialVat how_0,
  systemEra    = 0,
  systemSender = id_god
}
```

Chapter 4

Act framework

4.1 Act descriptions

We define the Maker act vocabulary as a data type. This is used for logging and generally for representing acts.

```
data Act =
    Bite (Id Urn)
  | Draw (Id Urn) Wad
  | Form (Id Ilk) (Id Jar)
  | Free (Id Urn) Wad
  | Frob Ray
  | Give (Id Urn) Address
  | Grab (Id Urn)
  | Heal Wad
  | Lock (Id Urn) Wad
  | Loot Wad
  | Mark (Id Jar) Wad     Nat
  | Open (Id Urn) (Id Ilk)
  | Prod
  | Poke (Id Urn)
  | Pull (Id Jar) Address Wad
  | Shut (Id Urn)
  | Tell Wad
  | Warp Nat
  | Wipe (Id Urn) Wad
  deriving (Eq, Show)
```

Acts which are logged through the `note` modifier record the sender ID and the act descriptor.

```
data Log = LogNote Address Act
  deriving (Show, Eq)
```

Acts can fail. We divide the failure modes into general assertion failures and authentication failures.

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

4.2 The Maker monad

The reader does not need any abstract understanding of monads to understand the code. What they give us is a nice syntax—the **do** notation—for expressing exceptions, state, and logging in a way that is still purely functional.

```
newtype Maker a =
  Maker (StateT System
    (WriterT (Seq Log)
      (Except Error)) a)
  deriving (
    Functor, Applicative, Monad,
    MonadError  Error,
    MonadState  System,
    MonadWriter (Seq Log)
  )

exec :: System
     → Maker ()
     → Either Error (System, Seq Log)
exec sys (Maker m) =
  runExcept (runWriterT (execStateT m sys))

instance MonadReader System Maker where
  ask = Maker get
  local f (Maker m) = Maker $ do
    s ← get; put (f s)
    x ← m;  put s
    return x
```

4.3 Constraints

```
type Reads  r  m = MonadReader r m
type Writes w  m = MonadState w m
type Logs       m = MonadWriter (Seq Log) m
type Fails      m = MonadError Error m

type IsAct = ?act :: Act
type Notes      m = (IsAct, Logs m)
```

4.4 Accessor aliases

```
ilkAt  id = vat ∘ ilks ∘ ix id
urnAt  id = vat ∘ urns ∘ ix id
jarAt  id = vat ∘ jars ∘ ix id
```

4.5 Logging and asserting

```
log :: Logs m ⇒ Log → m ()
log x = Writer.tell (Sequence.singleton x)

aver :: Fails m ⇒ Bool → m ()
aver x = unless x (throwError AssertError)

need :: (Fails m, Reads r m)
   ⇒ Getting (First a) r a → m a
need f = preview f ≫= λcase
  Nothing → throwError AssertError
  Just x → return x
```

4.6 Modifiers

```
note ::
  (IsAct, Logs m,
   Reads r m,
```

```
      HasSender r Address)
   ⇒ m a → m a

note k = do
  s ← view sender
  x ← k
  log (LogNote s ?act)
  return x

auth ::
  (IsAct, Fails m,
   Reads r m,
      HasSender r Address)
   ⇒ m a → m a

auth continue = do
  s ← view sender
  unless (s ≡ id_god)
    (throwError AuthError)
  continue
```

Chapter 5

Acts

We call the basic operations of the Dai credit system "acts."

5.1 Risk assessment

We divide an urn's situation into five stages of risk. Table 5.1 shows which acts each stage allows. The stages are naturally ordered from more to less risky.

data Stage = `Dread` | `Grief` | `Panic` | `Worry` | `Anger` | `Pride`
  deriving (Eq, Ord, Show)

First we define a pure function *analyze* that determines an urn's stage.

analyze $\texttt{era}_0\ \texttt{par}_0\ \texttt{urn}_0\ \texttt{ilk}_0\ \texttt{jar}_0 =$
  if
    Undergoing liquidation?
      | $\mathit{view}\ \texttt{vow}\ \texttt{urn}_0 \not\equiv \text{Nothing}$ $\rightarrow$ `Dread`
    Liquidation triggered?
      | $\mathit{view}\ \texttt{cat}\ \texttt{urn}_0 \not\equiv \text{Nothing}$ $\rightarrow$ `Grief`
    Undercollateralized?
      | $\texttt{pro} < \texttt{min}$ $\rightarrow$ `Panic`
    Price feed expired?
      | $\texttt{era}_0 > \mathit{view}\ \texttt{zzz}\ \texttt{jar}_0 + \mathit{view}\ \texttt{lag}\ \texttt{ilk}_0$ $\rightarrow$ `Panic`
    Price feed in limbo?
      | $\mathit{view}\ \texttt{zzz}\ \texttt{jar}_0 < \texttt{era}_0$ $\rightarrow$ `Worry`
    Debt ceiling reached?
      | $\texttt{cap} > \mathit{view}\ \texttt{hat}\ \texttt{ilk}_0$ $\rightarrow$ `Anger`
    Safely overcollateralized
      | *otherwise* $\rightarrow$ `Pride`
  where
    CDP's collateral value in SDR:
      $\texttt{pro} = \mathit{view}\ \texttt{jam}\ \texttt{urn}_0 * \mathit{view}\ \texttt{tag}\ \texttt{jar}_0$
    CDP type's total debt in SDR:
      $\texttt{cap} = \mathit{view}\ \texttt{din}\ \texttt{ilk}_0 * \mathit{cast}\ (\mathit{view}\ \texttt{chi}\ \texttt{ilk}_0)$
    CDP's debt in SDR:
      $\texttt{con} = \mathit{view}\ \texttt{art}\ \texttt{urn}_0 * \mathit{cast}\ (\mathit{view}\ \texttt{chi}\ \texttt{ilk}_0) * \texttt{par}_0$
    Required collateral as per liquidation ratio:
      $\texttt{min} = \texttt{con} * \mathit{view}\ \texttt{mat}\ \texttt{ilk}_0$

Table 5.1: Urn acts in the five stages of risk

	give	shut	lock	wipe	free	draw	bite	grab	plop	
Pride	•	•	•	•	•	•				overcollateralized
Anger	•	•	•	•	•					debt ceiling reached
Worry	•	•	•	•						price feed in limbo
Panic	•	•	•	•			•			undercollateralized
Grief	•							•		liquidation initiated
Dread	•								•	liquidation in progress

Now we define the internal act `gaze` which returns the value of *analyze* after ensuring the system state is updated.

```
gaze id_urn = do
  Perform dai revaluation and rate adjustment
  prod
  Update price of specific debt coin
  id_ilk ← need (urnAt id_urn ∘ ilk)
  drip id_ilk
  Read parameters for risk analysis
  era_0 ← view era
  par_0 ← view (vat ∘ par)
  urn_0 ← need (urnAt id_urn)
  ilk_0 ← need (ilkAt  (view ilk urn_0))
  jar_0 ← need (jarAt  (view jar ilk_0))
  Return risk stage of CDP
  return (analyze era_0 par_0 urn_0 ilk_0 jar_0)
```

5.2 Lending

```
open id_urn id_ilk =
  note $ do
    id_lad ← view sender
    vat ∘ urns ∘ at id_urn ?= defaultUrn id_ilk id_lad


lock id_urn x =
  note $ do
```

Ensure CDP exists; identify collateral type

$id_{\texttt{ilk}} \leftarrow need\ (urnAt\ id_{\texttt{urn}} \circ \texttt{ilk})$

$id_{\texttt{jar}} \leftarrow need\ (ilkAt\ \ id_{\texttt{ilk}} \circ \texttt{jar})$

Record an increase in collateral

$urnAt\ id_{\texttt{urn}} \circ \texttt{jam} \mathrel{+}= x$

Take sender's tokens

$id_{\texttt{lad}} \leftarrow view\ sender$

$\texttt{pull}\ id_{\texttt{jar}}\ id_{\texttt{lad}}\ x$

$\texttt{free}\ id_{\texttt{urn}}\ \texttt{wad}_{\texttt{gem}} =$

$\texttt{note}\ \$\ \mathbf{do}$

Fail if sender is not the CDP owner.

$id_{sender} \leftarrow view\ sender$

$id_{\texttt{lad}} \leftarrow need\ (urnAt\ id_{\texttt{urn}} \circ \texttt{lad})$

$\texttt{aver}\ (id_{sender} \equiv id_{\texttt{lad}})$

Tentatively record the decreased collateral.

$urnAt\ id_{\texttt{urn}} \circ \texttt{jam} \mathrel{-}= \texttt{wad}_{\texttt{gem}}$

Fail if collateral decrease results in undercollateralization.

$\texttt{gaze}\ id_{\texttt{urn}} \gg\!= \texttt{aver} \circ (\equiv \texttt{Pride})$

Send the collateral to the CDP owner.

$id_{\texttt{ilk}} \leftarrow need\ (urnAt\ \ id_{\texttt{urn}} \circ \texttt{ilk})$

$id_{\texttt{jar}} \leftarrow need\ (ilkAt\ \ id_{\texttt{ilk}} \circ \texttt{jar})$

$\texttt{push}\ id_{\texttt{jar}}\ id_{\texttt{lad}}\ \texttt{wad}_{\texttt{gem}}$

$\texttt{draw}\ id_{\texttt{urn}}\ \texttt{wad}_{\text{DAI}} =$

$\texttt{note}\ \$\ \mathbf{do}$

Fail if sender is not the CDP owner

$id_{sender} \leftarrow view\ sender$

$id_{\texttt{lad}} \leftarrow need\ (urnAt\ id_{\texttt{urn}} \circ \texttt{lad})$

$\texttt{aver}\ (id_{sender} \equiv id_{\texttt{lad}})$

Update price of debt coin

$id_{\texttt{ilk}} \leftarrow need\ (urnAt\ id_{\texttt{urn}} \circ \texttt{ilk})$

$\texttt{chi}_1 \leftarrow \texttt{drip}\ id_{\texttt{ilk}}$

Denominate draw amount in debt coin

$\mathbf{let}\ \texttt{wad}_{\texttt{chi}} = \texttt{wad}_{\text{DAI}} \,/\, cast\ \texttt{chi}_1$

```
    Increase debt
     urnAt id_urn ∘ art += wad_chi
    Roll back unless overcollateralized
     gaze id_urn ≫= aver ∘ (≡ Pride)
    Mint dai and send to the CDP owner
     mint id_DAI wad_DAI
     push id_DAI id_lad wad_DAI

wipe id_urn wad_DAI =
  note $ do
    Fail if sender is not the CDP owner
     id_sender ← view sender
     id_lad    ← need (urnAt id_urn ∘ lad)
     aver (id_sender ≡ id_lad)
    Update price of debt coin
     id_ilk ← need (urnAt id_urn ∘ ilk)
     chi_1 ← drip id_ilk
    Denominate dai amount in debt coin
     let wad_chi = wad_DAI / cast chi_1
    Roll back if the CDP is not overcollateralized
     gaze id_urn ≫= aver ∘ (≡ Pride)
    Reduce debt
     urnAt id_urn ∘ art −= wad_chi
    Take dai from CDP owner, or roll back
     pull id_DAI id_lad wad_DAI
    Destroy dai
     burn id_DAI wad_DAI

give id_urn id_lad =
  note $ do
    x ← need (urnAt id_urn ∘ lad)
    y ← view sender
    aver (x ≡ y)
    urnAt id_urn ∘ lad := id_lad
```

```
shut id_urn =
  note $ do
    Update price of debt coin
    id_ilk ← need (urnAt id_urn ∘ ilk)
    chi_1 ← drip id_ilk
    Attempt to repay all the CDP's outstanding dai
    art0 ← need (urnAt id_urn ∘ art)
    wipe id_urn (art0 * cast chi_1)
    Reclaim all the collateral
    jam0 ← need (urnAt id_urn ∘ jam)
    free id_urn jam0
    Nullify the CDP
    vat ∘ urns ∘ at id_urn := Nothing
```

5.3 Frequent adjustments

```
prod = note $ do
  era₀ ← view era
  tau₀ ← view (vat ∘ tau)
  fix₀ ← view (vat ∘ fix)
  par₀ ← view (vat ∘ par)
  how₀ ← view (vat ∘ how)
  way₀ ← view (vat ∘ way)
  let
    Time difference in seconds
    age = era₀ − tau₀
    Current deflation rate applied to target price
    par₁ = par₀ * cast (way₀ ↑↑ age)
    Sensitivity parameter applied over time
    wag = how₀ * fromIntegral age
    Deflation rate scaled up or down
    way₁ = inj (prj way₀ +
                  if fix₀ < par₀ then wag else − wag)
  vat ∘ par := par₁
  vat ∘ way := way₁
  vat ∘ 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)
```

This internal act happens on every *poke*. It is also invoked when governance changes the `tax` of an `ilk`.

```
drip id_ilk = do
  Current time stamp
  era₀ ← view era
  rho₀ ← need (ilkAt id_ilk ∘ rho)
  Current stability fee
  tax₀ ← need (ilkAt id_ilk ∘ tax)
  Current price of debt coin
```

```
  chi0 ← need (ilkAt id_ilk ∘ chi)
  let
    age  = era0 − rho0
    chi1 = chi0 ∗ tax0 ↑↑ age
  ilkAt id_ilk ∘ chi := chi1
  ilkAt id_ilk ∘ rho := era0
  return chi1
```

5.4 Governance

```
form id_ilk id_jar =
  auth ∘ note $ do
    vat ∘ ilks ∘ at id_ilk ?= defaultIlk id_jar
```

```
frob how′ =
  auth ∘ note $ do
    vat ∘ how := how′
```

5.5 Price feedback

```
mark id_jar tag1 zzz1 =
  auth ∘ note $ do
    jarAt id_jar ∘ tag := tag1
    jarAt id_jar ∘ zzz := zzz1
```

```
tell x =
  auth ∘ note $ do
    vat ∘ fix := x
```

5.6 Liquidation and settlement

```
bite id_urn =
  note $ do
    Fail if urn is not undercollateralized
    gaze id_urn >=> aver ∘ (≡ Panic)
    Record the sender as the requester of liquidation
    id_cat              ← view sender
    urnAt id_urn ∘ cat := id_cat
    Read current debt
    art0 ← need (urnAt id_urn ∘ art)
    Update price of debt coin
    id_ilk ← need (urnAt id_urn ∘ ilk)
    chi_1 ← drip id_ilk
    Read liquidation penalty ratio
    id_ilk ← need (urnAt id_urn ∘ ilk)
    axe_0 ← need (ilkAt id_ilk  ∘ axe)
    Apply liquidation penalty to debt
    let art1 = art0 * axe_0
    Update debt and record it as in need of settlement
    urnAt id_urn ∘ art := art1
    sin                 += art1 * chi_1
```

```
grab id_urn =
  auth ∘ note $ do
    Fail if CDP is not marked for liquidation
    gaze id_urn >=> aver ∘ (≡ Grief)
    Record the sender as the CDP's settler
    id_vow ← view sender
    urnAt id_urn ∘ vow := id_vow
    Clear the CDP's requester of liquidation
    urnAt id_urn ∘ cat := Nothing
```

```
heal wad_DAI =
```

```
  auth ∘ note $ do
    vat ∘ sin −= wad_DAI


loot wad_DAI =
  auth ∘ note $ do
    vat ∘ pie −= wad_DAI
```

5.7 Minting, burning, and transferring

```
pull id_jar id_lad w = do
  g  ← need (jarAt id_jar ∘ gem)
  g' ← transferFrom id_lad id_vat w g
  jarAt id_jar ∘ gem := g'


push id_jar id_lad w = do
  g  ← need (jarAt id_jar ∘ gem)
  g' ← transferFrom id_vat id_lad w g
  jarAt id_jar ∘ gem := g'


mint id_jar wad_0 = do
  jarAt id_jar ∘ gem ∘ totalSupply          += wad_0
  jarAt id_jar ∘ gem ∘ balanceOf ∘ ix id_vat += wad_0


burn id_jar wad_0 = do
  jarAt id_jar ∘ gem ∘ totalSupply          −= wad_0
  jarAt id_jar ∘ gem ∘ balanceOf ∘ ix id_vat −= wad_0
```

5.8 Test-related manipulation

```
warp t =
  auth ∘ note $ do
    era += t
```

5.9 Other stuff

```
perform :: Act → Maker ()
perform x =
  let ?act = x in case x of
    Form id jar   → form id jar
    Mark jar tag zzz → mark jar tag zzz
    Open id ilk   → open id ilk
    Tell wad      → tell wad
    Frob ray      → frob ray
    Prod          → prod
    Warp t        → warp t
    Give urn lad → give urn lad
    Pull jar lad wad → pull jar lad wad
    Lock urn wad → lock urn wad

transferFrom
  :: (MonadError Error m)
  ⇒ Address → Address → Wad
  → Gem → m Gem

transferFrom src dst wad gem =
  case view (balanceOf ∘ at src) gem of
    Nothing →
      throwError AssertError
    Just balance → do
      aver (balance ⩾ wad)
      return $ gem &~ do
        balanceOf ∘ ix src -= wad
        balanceOf ∘ at dst %=
          (λcase
            Nothing → Just wad
            Just x  → Just (wad + x))
```

Chapter 6

Testing

Sketches for property stuff...

```
data Parameter =
  Fix | Par | Way

maintains :: Eq a ⇒ Lens' System a → Maker () → System → Bool
maintains p = λm sys0 →
  case exec sys0 m of
    On success, data must be compared for equality
      Right (sys1, _) → view p sys0 ≡ view p sys1
    On rollback, data is maintained by definition
      Left _          → True

changesOnly :: Eq a ⇒ Lens' System a → Maker () → System → Bool
changesOnly p = λm sys0 →
  case exec sys0 m of
    On success, equalize p and compare
      Right (sys1, _) → set p (view p sys1) sys0 ≡ sys1
    On rollback, data is maintained by definition
      Left _          → True

also :: Lens' s a → Lens' s b → Lens' s (a, b)
also f g = lens getter setter
  where
  getter x = (view f x, view g x)
  setter x (a, b) = set f a (set g b x)
```

$$
\begin{aligned}
&\mathit{keeps} :: \text{Parameter} \to \text{Maker}\ () \to \text{System} \to \text{Bool}\\
&\mathit{keeps}\ \texttt{Fix} = \mathit{maintains}\ (\texttt{vat} \circ \texttt{fix})\\
&\mathit{keeps}\ \texttt{Par} = \mathit{maintains}\ (\texttt{vat} \circ \texttt{par})\\
&\mathit{keeps}\ \texttt{Way} = \mathit{maintains}\ (\texttt{vat} \circ \texttt{way})
\end{aligned}
$$

Thus:

$$
\begin{aligned}
&\mathit{foo}\ \texttt{sys}_0 = \mathit{all}\ (\lambda f \to f\ \texttt{sys}_0)\\
&\quad [\mathit{changesOnly}\ ((\texttt{vat} \circ \texttt{par})\ `\mathit{also}`\\
&\qquad\qquad (\texttt{vat} \circ \texttt{way}))\\
&\qquad (\mathit{perform}\ \texttt{Prod})]
\end{aligned}
$$

Appendix A

Act type signatures

type Numbers `wad` `ray` `nat` =
  (`wad`∼`Wad`, `ray`∼`Ray`, `nat`∼`Nat`)

We see that `drip` may fail; it reads an `ilk`'s `tax`, `cow`, `rho`, and *bag*; and it writes those same parameters except `tax`.

`drip` ::
  (Fails m,
   Reads r m,
     HasEra r `Nat`,
     HasVat r $\texttt{vat}_r$,
       HasIlks $\texttt{vat}_r$ (Map (Id `Ilk`) $\texttt{ilk}_r$),
         HasTax $\texttt{ilk}_r$ `Ray`,
         HasRho $\texttt{ilk}_r$ `Nat`,
         HasChi $\texttt{ilk}_r$ `Ray`,
   Writes w m,
     HasVat w $\texttt{vat}_w$,
       HasIlks $\texttt{vat}_w$ (Map (Id `Ilk`) $\texttt{ilk}_w$),
         HasRho $\texttt{ilk}_w$ `Nat`,
         HasChi $\texttt{ilk}_w$ `Ray`)
  ⇒ Id `Ilk` → m `Ray`

`form` ::
  (IsAct, Fails m, Logs m,
   Reads r m, HasSender r Address,
   Writes w m, HasVat w $\texttt{vat}_w$,

```
                    HasIlks vat_w (IdMap Ilk))
  ⇒ Id Ilk → Id Jar → m ()


frob :: (IsAct, Fails m, Logs m,
         Reads r m,  HasSender r Address,
         Writes w m, HasVat w vat_w,
                     HasHow vat_w ray)
  ⇒ ray → m ()


open ::
  (IsAct, Logs m,
   Reads r m,  HasSender r Address,
   Writes w m, HasVat w vat_w,
               HasUrns vat_w (IdMap Urn))
  ⇒ Id Urn → Id Ilk → m ()


give ::
  (IsAct, Fails m, Logs m,
   Reads r m,  HasSender r Address,
               HasVat r vat_r,
                 HasUrns vat_r (Map (Id Urn) urn_r),
                   HasLad urn_r Address,
   Writes w m, HasVat w vat_r)
  ⇒ Id Urn → Address → m ()


lock ::
  (IsAct, Fails m, Logs m,
   Reads r m,
     HasSender r Address,
     HasVat r vat_r,
       HasUrns vat_r (Map (Id Urn) urn_r),
         HasIlk urn_r (Id Ilk),
       HasIlks vat_r (Map (Id Ilk) ilk_r),
         HasJar ilk_r (Id Jar),
       HasJars vat_r (Map (Id Jar) jar_r),
         HasGem jar_r Gem,
   Writes w m,
     HasVat w vat_w,
```

```
        HasJars vat_w (Map (Id Jar) jar_r),
        HasUrns vat_w (Map (Id Urn) urn_w),
          HasJam urn_w Wad)
   ⇒ Id Urn → Wad → m ()


mark ::
  (IsAct, Fails m, Logs m,
   Reads r m,  HasSender r Address,
   Writes w m, HasVat w vat_w,
                   HasJars vat_w (Map (Id Jar) jar_w),
                     HasTag jar_w wad,
                     HasZzz jar_w nat)
   ⇒ Id Jar → wad → nat → m ()


tell ::
  (IsAct, Fails m, Logs m,
   Reads r m,  HasSender r Address,
   Writes w m, HasVat w vat_w,
                   HasFix vat_w wad)
   ⇒ wad → m ()


prod ::
  (IsAct, Logs m,
   Reads r m,
     HasSender r Address,
     HasEra r nat,
     HasVat r vat_r,  (HasPar vat_r wad,
                       HasTau vat_r nat,
                       HasHow vat_r ray,
                       HasWay vat_r ray,
                       HasFix vat_r wad),
   Writes w m,
     HasVat w vat_w, (HasPar vat_w wad,
                       HasWay vat_w ray,
                       HasTau vat_w nat),
   Integral nat,
   Ord wad, Fractional wad,
   Fractional ray, Real ray)
   ⇒ m ()
```

```
warp ::
  (IsAct, Fails m, Logs m,
   Reads r m,  HasSender r Address,
   Writes w m, HasEra w nat,
                    Num nat)
   ⇒ nat → m ()


pull ::
  (Fails m,
   Reads r m,
     HasVat r vat_r,  HasJars vat_r (Map (Id Jar) jar_r),
                        HasGem jar_r Gem,
   Writes w m,
     HasVat w vat_w, HasJars vat_w (Map (Id Jar) jar_r))
   ⇒ Id Jar → Address → Wad → m ()


push ::
  (Fails m,
   Reads r m,
     HasVat r vat_r,  HasJars vat_r (Map (Id Jar) jar_r),
                        HasGem jar_r Gem,
   Writes w m,
     HasVat w vat_w, HasJars vat_w (Map (Id Jar) jar_r))
   ⇒ Id Jar → Address → Wad → m ()


mint ::
  (Fails m,
   Writes w m,
     HasVat w vat_w, HasJars vat_w (Map (Id Jar) jar_r),
                        HasGem jar_r gem_r,
                          HasTotalSupply gem_r Wad,
                          HasBalanceOf   gem_r (Map Address Wad))
   ⇒ Id Jar → Wad → m ()


burn ::
  (Fails m,
   Writes w m,
     HasVat w vat_w, HasJars vat_w (Map (Id Jar) jar_r),
                        HasGem jar_r gem_r,
```

```
                              HasTotalSupply gem_r Wad,
                              HasBalanceOf   gem_r (Map Address Wad))
    ⇒ Id Jar → Wad → m ()

grab ::
  (IsAct, Fails m, Logs m,
   Numbers wad ray nat,
   Reads r m,
     HasSender r Address,
     HasEra r Nat,
     HasVat r vat_r,
       HasFix vat_r wad,
       HasPar vat_r wad,
       HasHow vat_r ray,
       HasWay vat_r ray,
       HasTau vat_r nat,
       HasUrns vat_r (Map (Id Urn) urn_r),
         HasJam urn_r wad,
         HasArt urn_r wad,
         HasCat urn_r (Maybe Address), HasVow urn_r (Maybe Address),
         HasIlk urn_r (Id Ilk),
       HasIlks vat_r (Map (Id Ilk) ilk_r),
         HasHat ilk_r wad,
         HasMat ilk_r wad,
         HasDin ilk_r wad,
         HasTax ilk_r ray,
         HasLag ilk_r nat,
         HasChi ilk_r ray,  HasRho ilk_r nat,
         HasJar ilk_r (Id Jar),
       HasJars vat_r (Map (Id Jar) jar_r),
         HasGem jar_r Gem,
         HasTag jar_r wad,
         HasZzz jar_r nat,
   Writes w m,
     HasVat w vat_w,
       HasTau vat_w nat,
       HasWay vat_w ray, HasPar vat_w wad,
       HasUrns vat_w (Map (Id Urn) urn_w),
         HasJam urn_w wad, HasArt urn_w wad,
         HasVow urn_w Address,
         HasCat urn_w (Maybe Address),
```

HasIlks $\texttt{vat}_w$ (Map (Id `Ilk`) $\texttt{ilk}_w$),
HasChi $\texttt{ilk}_w$ `ray`,
HasRho $\texttt{ilk}_w$ `nat`,
HasJars $\texttt{vat}_w$ (Map (Id `Jar`) $\texttt{jar}_r$)
) $\Rightarrow$ Id `Urn` $\rightarrow m$ ()