presents the

REFERENCE IMPLEMENTATION

of the remarkable

DAI CREDIT SYSTEM

issuing a diversely collateralized stablecoin

with last update on March 9, 2017.

Contents

List of Tables

List of Figures

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 previously unwritten storage and get back a zero value, whereas in Haskell we must give explicit defaults. The state rollback behavior of failed actions is also explicit in the type of the execution function, which may return an error.

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.

Part I

Implementation

Chapter 2

Preamble

We replace the default prelude module with our own. This brings in dependencies and hides unneeded symbols. Consult Appendix A to see exactly what is brought into scope.

```
module Maker where
import Maker.Prelude  Fully import the Maker prelude
import Prelude ()     Import nothing from Prelude
import Maker.Decimal
import Debug.Trace
```

Chapter 3

Types

3.1 Numeric types

The system uses two different precisions of decimal fixed point numbers, which we call *wads* and *rays*, having respectively 18 digits of precision (used for token quantities) and 36 digits (used for precise rates and ratios).

```
Define the distinct wad type for currency quantities
newtype Wad = Wad (Decimal E18)
  deriving (Ord, Eq, Num, Real, Fractional, RealFrac)
Define the distinct ray type for precise rate quantities
newtype Ray = Ray (Decimal E36)
  deriving (Ord, Eq, Num, Real, Fractional, RealFrac)
```

See Appendix B for details on how we modify Haskell's decimal fixed point type to do more correct rounding for multiplication and division.

Haskell number types are not automatically converted, so in calculations that combine wads and rays, we convert explicitly with 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 time durations in whole seconds.

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

```
instance Epsilon Wad where ε = Wad ε
instance Epsilon Ray where ε = Ray ε
```

3.2 Identifiers and addresses

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 define another type for representing Ethereum account addresses.

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

We also have three predefined entity identifiers.

```
  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"
  The address of the test driver
id_toy = Address "TOY"
```

This section introduces the records stored by the Maker system.

3.3 `Gem` — ERC20 token model

```
data Holder = AddressHolder Address
  |           JarHolder     (Id Jar)
  deriving (Eq, Show, Ord)
```

data `Gem` = `Gem` {

gemTotalSupply :: `Wad`,
gemBalanceOf :: Map Holder `Wad`

} **deriving** (Eq, Show)

3.4 `Jar` — collateral type

data `Jar` = `Jar` {

jarGem :: `Gem`, Collateral token
jarTag :: `Wad`, Market price
jarZzz :: `Sec` Price expiration

} **deriving** (Eq, Show)

3.5 `Ilk` — CDP type

data `Ilk` = `Ilk` {

ilkJar :: Id `Jar`, Collateral vault
ilkAxe :: `Ray`, Liquidation penalty
ilkHat :: `Wad`, Debt ceiling
ilkMat :: `Ray`, Liquidation ratio
ilkTax :: `Ray`, Stability fee
ilkLag :: `Sec`, Limbo duration
ilkRho :: `Sec`, Last dripped
ilkRum :: `Wad`, Total debt in debt unit
ilkChi :: `Ray` Dai value of debt unit

} **deriving** (Eq, Show)

3.6 `Urn` — collateralized debt position (CDP)

data `Urn` = `Urn` {

urnCat :: Maybe Address, Address of liquidation initiator

$urnVow$ :: Maybe Address,		Address of liquidation contract
$urnLad$ :: Address,		Issuer
$urnIlk$ :: Id `Ilk`,		CDP type
$urnArt$ :: `Wad`,		Outstanding debt in debt unit
$urnJam$ :: `Wad`		Collateral amount in debt unit

} **deriving** (Eq, Show)

3.7 `Vat` — CDP engine

data `Vat` = `Vat` {

$vatFix$ :: `Wad`,		Market price
$vatHow$ :: `Ray`,		Sensitivity
$vatPar$ :: `Wad`,		Target price
$vatWay$ :: `Ray`,		Target rate
$vatTau$ :: `Sec`,		Last prodded
$vatJoy$ :: `Wad`,		Unprocessed stability fees
$vatSin$ :: `Wad`,		Bad debt from liquidated CDPs
$vatJars$ :: Map (Id `Jar`) `Jar`,		Collateral tokens
$vatIlks$ :: Map (Id `Ilk`) `Ilk`,		CDP types
$vatUrns$ :: Map (Id `Urn`) `Urn`		CDPs

} **deriving** (Eq, Show)

3.8 System model

data System = System {

$systemVat$	:: `Vat`,	Root Maker entity
$systemEra$	:: `Sec`,	Current time stamp
$systemSender$	:: Address,	Sender of current act
$systemAccounts$	:: [Address]	For test suites

} **deriving** (Eq, Show)

Lens fields

$$
\begin{aligned}
&\mathit{makeFields}\ \text{''}\texttt{Gem}\\
&\mathit{makeFields}\ \text{''}\texttt{Jar}\\
&\mathit{makeFields}\ \text{''}\texttt{Ilk}\\
&\mathit{makeFields}\ \text{''}\texttt{Urn}\\
&\mathit{makeFields}\ \text{''}\texttt{Vat}\\
&\mathit{makeFields}\ \text{''}\mathrm{System}
\end{aligned}
$$

3.9 Default data

$$
\begin{aligned}
&\mathit{defaultIlk} :: \mathrm{Id}\ \texttt{Jar} \to \texttt{Ilk}\\
&\mathit{defaultIlk}\ \mathit{id}_{\texttt{jar}} = \texttt{Ilk}\ \{\\
&\quad \mathit{ilkJar}\ = \mathit{id}_{\texttt{jar}},\\
&\quad \mathit{ilkAxe}\ = \texttt{Ray}\ 1,\\
&\quad \mathit{ilkMat}\ = \texttt{Ray}\ 1,\\
&\quad \mathit{ilkTax}\ = \texttt{Ray}\ 1,\\
&\quad \mathit{ilkHat}\ = \texttt{Wad}\ 0,\\
&\quad \mathit{ilkLag}\ = \texttt{Sec}\ 0,\\
&\quad \mathit{ilkChi}\ = \texttt{Ray}\ 1,\\
&\quad \mathit{ilkRum} = \texttt{Wad}\ 0,\\
&\quad \mathit{ilkRho}\ = \texttt{Sec}\ 0\\
&\}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{initialVat} :: \texttt{Ray} \to \texttt{Vat}\\
&\mathit{initialVat}\ \texttt{how}_0 = \texttt{Vat}\ \{
\end{aligned}
$$

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

$$
\begin{aligned}
&initialSystem :: \texttt{Ray} \to \text{System}\\
&initialSystem\ \texttt{how}_0 = \text{System}\ \{\\
&\quad systemVat \;&&= initialVat\ \texttt{how}_0,\\
&\quad systemEra \;&&= 0,\\
&\quad systemSender \;&&= id_{god},\\
&\quad systemAccounts \;&&= mempty\\
&\}
\end{aligned}
$$

Chapter 4

Acts

The *acts* are the basic state transitions of the credit system.

For details on the underlying "Maker monad," which specifies how the act definitions behave with regard to state and rollback thereof, see chapter 5.

4.1 Assessment

We divide an urn's situation into five stages of risk. Table 4.1 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 era₀ par₀ urn₀ ilk₀ jar₀ =
  if
    Undergoing liquidation?
      | view vow urn₀ ≢ Nothing          → Dread
    Liquidation triggered?
      | view cat urn₀ ≢ Nothing          → Grief
    Undercollateralized?
      | pro < min                        → Panic
    Price feed expired?
      | era₀ > view zzz jar₀ + view lag ilk₀ → Panic
    Price feed in limbo?
      | view zzz jar₀ < era₀             → Worry
    Debt ceiling reached?
      | cap > view hat ilk₀              → Anger
    Safely overcollateralized
      | otherwise                        → Pride
  where
    CDP's collateral value in SDR:
      pro = view jam urn₀ * view tag jar₀
    CDP type's total debt in SDR:
      cap = (view rum ilk₀ * cast (view chi ilk₀)) :: Wad
    CDP's debt in SDR:
      con = view art urn₀ * cast (view chi ilk₀) * par₀
    Required collateral as per liquidation ratio:
      min = con * view mat ilk₀
```

Table 4.1: 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 unit
  id_ilk ← look (vat ∘ urns ∘ ix id_urn ∘ ilk)
  drip id_ilk
  Read parameters for risk analysis
  era_0 ← use era
  par_0 ← use (vat ∘ par)
  urn_0 ← look (vat ∘ urns ∘ ix id_urn)
  ilk_0 ← look (vat ∘ ilks ∘ ix (view ilk urn_0))
  jar_0 ← look (vat ∘ jars ∘ ix (view jar ilk_0))
  Return risk stage of CDP
  return (analyze era_0 par_0 urn_0 ilk_0 jar_0)
```

4.2 Lending

```
open id_urn id_ilk =
  do
    id_lad ← use sender
    vat ∘ urns ∘ at id_urn ?= defaultUrn id_ilk id_lad
```

```
lock id_urn x = do
  Ensure CDP exists; identify collateral type
```

$id_{\texttt{ilk}} \leftarrow \mathit{look}\ (\texttt{vat} \circ \texttt{urn}s \circ \mathit{ix}\ \mathit{id}_{\texttt{urn}} \circ \texttt{ilk})$
$id_{\texttt{jar}} \leftarrow \mathit{look}\ (\texttt{vat} \circ \texttt{ilk}s \circ \mathit{ix}\ \mathit{id}_{\texttt{ilk}} \circ \texttt{jar})$

Record an increase in collateral

$\texttt{vat} \circ \texttt{urn}s \circ \mathit{ix}\ \mathit{id}_{\texttt{urn}} \circ \texttt{jam} \mathrel{+}= x$

Take sender's tokens

$id_{\texttt{lad}} \leftarrow \mathit{use}\ \mathit{sender}$
$\texttt{pull}\ \mathit{id}_{\texttt{jar}}\ \mathit{id}_{\texttt{lad}}\ x$

$\texttt{free}\ \mathit{id}_{\texttt{urn}}\ \texttt{wad}_{\texttt{gem}} = \mathbf{do}$

Fail if sender is not the CDP owner

$id_{\mathit{sender}} \leftarrow \mathit{use}\ \mathit{sender}$
$id_{\texttt{lad}} \leftarrow \mathit{look}\ (\texttt{vat} \circ \texttt{urn}s \circ \mathit{ix}\ \mathit{id}_{\texttt{urn}} \circ \texttt{lad})$
$\texttt{aver}\ (\mathit{id}_{\mathit{sender}} \equiv \mathit{id}_{\texttt{lad}})$

Decrease the collateral amount

$\texttt{vat} \circ \texttt{urn}s \circ \mathit{ix}\ \mathit{id}_{\texttt{urn}} \circ \texttt{jam} \mathrel{-}= \texttt{wad}_{\texttt{gem}}$

Roll back if undercollateralized

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

Send the collateral to the CDP owner

$id_{\texttt{ilk}} \leftarrow \mathit{look}\ (\texttt{vat} \circ \texttt{urn}s \circ \mathit{ix}\ \mathit{id}_{\texttt{urn}} \circ \texttt{ilk})$
$id_{\texttt{jar}} \leftarrow \mathit{look}\ (\texttt{vat} \circ \texttt{ilk}s \circ \mathit{ix}\ \mathit{id}_{\texttt{ilk}} \circ \texttt{jar})$
$\texttt{push}\ \mathit{id}_{\texttt{jar}}\ \mathit{id}_{\texttt{lad}}\ \texttt{wad}_{\texttt{gem}}$

$\texttt{draw}\ \mathit{id}_{\texttt{urn}}\ \texttt{wad}_{\text{DAI}} = \mathbf{do}$

Fail if sender is not the CDP owner

$id_{\mathit{sender}} \leftarrow \mathit{use}\ \mathit{sender}$
$id_{\texttt{lad}} \leftarrow \mathit{look}\ (\texttt{vat} \circ \texttt{urn}s \circ \mathit{ix}\ \mathit{id}_{\texttt{urn}} \circ \texttt{lad})$
$\texttt{aver}\ (\mathit{id}_{\mathit{sender}} \equiv \mathit{id}_{\texttt{lad}})$

Update value of debt unit

$id_{\texttt{ilk}} \leftarrow \mathit{look}\ (\texttt{vat} \circ \texttt{urn}s \circ \mathit{ix}\ \mathit{id}_{\texttt{urn}} \circ \texttt{ilk})$
$\texttt{chi}_1 \leftarrow \texttt{drip}\ \mathit{id}_{\texttt{ilk}}$

Denominate draw amount in debt unit

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

Increase debt

$\texttt{vat} \circ \texttt{urn}s \circ \mathit{ix}\ \mathit{id}_{\texttt{urn}} \circ \texttt{art} \mathrel{+}= \texttt{wad}_{\texttt{chi}}$

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 = do
  Fail if sender is not the CDP owner
    id_sender ← use sender
    id_lad    ← look (vat ∘ urns ∘ ix id_urn ∘ lad)
    aver (id_sender ≡ id_lad)
  Update value of debt unit
    id_ilk ← look (vat ∘ urns ∘ ix id_urn ∘ ilk)
    chi_1 ← drip id_ilk
  Roll back unless overcollateralized
    gaze id_urn >>= aver ∘ (≡ Pride)
  Denominate dai amount in debt unit
    let wad_chi = wad_DAI / cast chi_1
  Reduce debt
    vat ∘ urns ∘ ix 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 = do
  x ← look (vat ∘ urns ∘ ix id_urn ∘ lad)
  y ← use sender
  aver (x ≡ y)
  vat ∘ urns ∘ ix id_urn ∘ lad := id_lad

shut id_urn = do
  Update value of debt unit
    id_ilk ← look (vat ∘ urns ∘ ix id_urn ∘ ilk)
    chi_1 ← drip id_ilk
```

Attempt to repay all the CDP's outstanding dai

$\texttt{art}_0 \leftarrow \mathit{look}\ (\texttt{vat} \circ \texttt{urn}s \circ \mathit{ix}\ \mathit{id}_{\texttt{urn}} \circ \texttt{art})$

$\texttt{wipe}\ \mathit{id}_{\texttt{urn}}\ (\texttt{art}_0 * \mathit{cast}\ \texttt{chi}_1)$

Reclaim all the collateral

$\mathit{jam0} \leftarrow \mathit{look}\ (\texttt{vat} \circ \texttt{urn}s \circ \mathit{ix}\ \mathit{id}_{\texttt{urn}} \circ \texttt{jam})$

$\texttt{free}\ \mathit{id}_{\texttt{urn}}\ \mathit{jam0}$

Nullify the CDP

$\texttt{vat} \circ \texttt{urn}s \circ \mathit{at}\ \mathit{id}_{\texttt{urn}} := \text{Nothing}$

4.3 Adjustment

```
prod = do
  era₀ ← use era
  tau₀ ← use (vat ∘ tau)
  fix₀ ← use (vat ∘ fix)
  par₀ ← use (vat ∘ par)
  how₀ ← use (vat ∘ how)
  way₀ ← use (vat ∘ 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 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)
```

```
drip id_ilk = do
  Current time stamp
  era_0  ← use era
  Time stamp of previous drip
  rho_0  ← look (vat ∘ ilks ∘ ix id_ilk ∘ rho)
  Current stability fee
  tax_0  ← look (vat ∘ ilks ∘ ix id_ilk ∘ tax)
  Current value of debt unit
  chi_0  ← look (vat ∘ ilks ∘ ix id_ilk ∘ chi)
  Current total debt in debt unit
  rum0 ← look (vat ∘ ilks ∘ ix id_ilk ∘ rum)
  Current unprocessed stability fee revenue
  joy_0  ← look (vat ∘ ilks ∘ ix id_ilk ∘ joy)
  let
    age   = era_0 − rho_0
    chi_1 = chi_0 * tax_0 ↑↑ age
    joy_1 = joy_0 + (cast (chi_1 − chi_0) :: Wad) * rum0
  vat ∘ ilks ∘ ix id_ilk ∘ chi := chi_1
  vat ∘ ilks ∘ ix id_ilk ∘ rho := era_0
  vat ∘ ilks ∘ ix id_ilk ∘ joy := joy_1
  return chi_1
```

4.4 Feedback

```
mark id_jar tag_1 zzz_1 =
  auth $ do
    vat ∘ jars ∘ ix id_jar ∘ tag := tag_1
    vat ∘ jars ∘ ix id_jar ∘ zzz := zzz_1
```

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

4.5 Liquidation

$\texttt{bite}\ id_{\texttt{urn}} = \textbf{do}$

Fail if CDP is not in need of liquidation

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

Record the sender as the liquidation initiator

$id_{\texttt{cat}} \leftarrow \mathit{use\ sender}$

$\texttt{vat} \circ \texttt{urns} \circ \mathit{ix}\ id_{\texttt{urn}} \circ \texttt{cat} := id_{\texttt{cat}}$

Read current debt

$\texttt{art}_0 \leftarrow \mathit{look}\ (\texttt{vat} \circ \texttt{urns} \circ \mathit{ix}\ id_{\texttt{urn}} \circ \texttt{art})$

Update value of debt unit

$id_{\texttt{ilk}} \leftarrow \mathit{look}\ (\texttt{vat} \circ \texttt{urns} \circ \mathit{ix}\ id_{\texttt{urn}} \circ \texttt{ilk})$

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

Read liquidation penalty ratio

$id_{\texttt{ilk}} \leftarrow \mathit{look}\ (\texttt{vat} \circ \texttt{urns} \circ \mathit{ix}\ id_{\texttt{urn}} \circ \texttt{ilk})$

$\texttt{axe}_0 \leftarrow \mathit{look}\ (\texttt{vat} \circ \texttt{ilks} \circ \mathit{ix}\ id_{\texttt{ilk}} \circ \texttt{axe})$

Apply liquidation penalty to debt

$\textbf{let}\ \texttt{art}_1 = \texttt{art}_0 * \texttt{axe}_0$

Update CDP debt

$\texttt{vat} \circ \texttt{urns} \circ \mathit{ix}\ id_{\texttt{urn}} \circ \texttt{art} := \texttt{art}_1$

Record as bad debt

$\texttt{sin} \mathrel{+}= \texttt{art}_1 * \texttt{chi}_1$

$\texttt{grab}\ id_{\texttt{urn}} =$

$\texttt{auth}\ \$\ \textbf{do}$

Fail if CDP is not marked for liquidation

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

Record the sender as the CDP's settler

$id_{\texttt{vow}} \leftarrow \mathit{use\ sender}$

$\texttt{vat} \circ \texttt{urns} \circ \mathit{ix}\ id_{\texttt{urn}} \circ \texttt{vow} := \text{Just}\ id_{\texttt{vow}}$

Clear the CDP's requester of liquidation

$\texttt{vat} \circ \texttt{urns} \circ \mathit{ix}\ id_{\texttt{urn}} \circ \texttt{cat} := \text{Nothing}$

$\texttt{heal}\ \texttt{wad}_{\text{DAI}} =$

$\texttt{auth}\ \$\ \textbf{do}$

$\texttt{vat} \circ \texttt{sin} \mathrel{-}= \texttt{wad}_{\text{DAI}}$

```
love wad_DAI =
  auth $ do
    vat ∘ joy −= wad_DAI
```

4.6 Governance

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

```
frob how' =
  auth $ do
    vat ∘ how := how'
```

```
chop id_ilk axe_1 =
  auth $ do
    vat ∘ ilks ∘ ix id_ilk ∘ axe := axe_1
```

```
cork id_ilk hat_1 =
  auth $ do
    vat ∘ ilks ∘ ix id_ilk ∘ hat := hat_1
```

```
calm id_ilk lag_1 =
  auth $ do
    vat ∘ ilks ∘ ix id_ilk ∘ lag := lag_1
```

```
cuff id_ilk mat_1 =
  auth $ do
    vat ∘ ilks ∘ ix id_ilk ∘ mat := mat_1
```

```
crop id_ilk tax_1 =
  auth $ do
    drip id_ilk
    vat ∘ ilks ∘ ix id_ilk ∘ tax := tax_1
```

4.7 Treasury

```
pull id_jar id_lad w = do
  g  ← look (vat ∘ jars ∘ ix id_jar ∘ gem)
  g' ← transferFrom (AddressHolder id_lad)
                    (JarHolder id_jar) w g
  vat ∘ jars ∘ ix id_jar ∘ gem := g'
```

```
push id_jar id_lad w = do
  g  ← look (vat ∘ jars ∘ ix id_jar ∘ gem)
  g' ← transferFrom (JarHolder id_jar)
                    (AddressHolder id_lad) w g
  vat ∘ jars ∘ ix id_jar ∘ gem := g'
```

```
mint id_jar wad_0 =
  zoom (vat ∘ jars ∘ ix id_jar ∘ gem) $ do
    totalSupply                      += wad_0
    balanceOf ∘ ix (JarHolder id_jar) += wad_0
```

```
burn id_jar wad_0 =
  zoom (vat ∘ jars ∘ ix id_jar ∘ gem) $ do
    totalSupply                      -= wad_0
    balanceOf ∘ ix (JarHolder id_jar) -= wad_0
```

4.8 Manipulation

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

```
mine id_jar = do
  vat ∘ jars ∘ at id_jar ?= Jar {
    jarGem = Gem {
      gemTotalSupply = 1000000000000,
      gemBalanceOf   = singleton (AddressHolder id_toy) 1000000000000
    },
    jarTag = Wad 0,
    jarZzz = 0
  }
```

```
hand dst w id_jar = do
  g  ← look (vat ∘ jars ∘ ix id_jar ∘ gem)
  g' ← transferFrom (AddressHolder id_toy) (AddressHolder dst) w g
  vat ∘ jars ∘ ix id_jar ∘ gem := g'
```

```
sire lad = do
  accounts %= (lad:)
```

4.9 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
    Mine id       → mine id
    Hand lad wad jar → hand lad wad jar
    Sire lad      → sire lad
be :: Address → Act → Maker ()
be who x = do
  old ← use sender
  sender := who
  y ← perform x
  sender := old
  return y
transferFrom
  ::  ( ?act :: Act, MonadError Error m)
  ⇒  Holder → Holder → Wad
  →  Gem → m Gem
transferFrom src dst wad gem =
  case view (balanceOf ∘ at src) gem of
    Nothing →
      throwError (AssertError ?act)
    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 5

Act framework

5.1 Act descriptions

We define the Maker act vocabulary as a data type.

```
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
  | Love Wad
  | Mark (Id Jar) Wad     Sec
  | Open (Id Urn) (Id Ilk)
  | Prod
  | Pull (Id Jar) Address Wad
  | Shut (Id Urn)
  | Tell Wad
  | Warp Sec
  | Wipe (Id Urn) Wad
  | Mine (Id Jar)
  | Hand Address Wad      (Id Jar)
  | Sire Address
```

```
Test acts
  | Addr Address
  deriving (Eq, Show)
```

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

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

5.2 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 and state in a way that is still purely functional.

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

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

5.3 Asserting

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

look f = preuse f ≫= λcase
  Nothing → throwError (AssertError ?act)
  Just x → return x
```

5.4 Modifiers

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

Chapter 6

Testing

Sketches for property stuff...

$$
\begin{aligned}
&\textbf{data}\ \text{Parameter} = \\
&\quad \texttt{Fix} \mid \texttt{Par} \mid \texttt{Way}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{maintains} \\
&\quad :: \text{Eq}\ a \Rightarrow \text{Lens}'\ \text{System}\ a \rightarrow \text{Maker}\ () \\
&\qquad\quad \rightarrow \text{System} \rightarrow \text{Bool} \\
&\mathit{maintains}\ p = \lambda m\ \texttt{sys}_0 \rightarrow \\
&\quad \textbf{case}\ \mathit{exec}\ \texttt{sys}_0\ m\ \textbf{of} \\
&\qquad \text{On success, data must be compared for equality} \\
&\qquad\ \text{Right}\ \texttt{sys}_1 \rightarrow \mathit{view}\ p\ \texttt{sys}_0 \equiv \mathit{view}\ p\ \texttt{sys}_1 \\
&\qquad \text{On rollback, data is maintained by definition} \\
&\qquad\ \text{Left}\ \_ \qquad \rightarrow \text{True}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{changesOnly} \\
&\quad ::\ \ \text{Lens}'\ \text{System}\ a \rightarrow \text{Maker}\ () \\
&\quad \rightarrow \text{System} \rightarrow \text{Bool} \\
&\mathit{changesOnly}\ p = \lambda m\ \texttt{sys}_0 \rightarrow \\
&\quad \textbf{case}\ \mathit{exec}\ \texttt{sys}_0\ m\ \textbf{of} \\
&\qquad \text{On success, equalize } p \text{ and compare} \\
&\qquad\ \text{Right}\ \texttt{sys}_1 \rightarrow \mathit{set}\ p\ (\mathit{view}\ p\ \texttt{sys}_1)\ \texttt{sys}_0 \equiv \texttt{sys}_1 \\
&\qquad \text{On rollback, data is maintained by definition} \\
&\qquad\ \text{Left}\ \_ \qquad \rightarrow \text{True}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{also} :: \text{Lens}'\ s\ a \rightarrow \text{Lens}'\ s\ b \rightarrow \text{Lens}'\ s\ (a, b) \\
&\mathit{also}\ f\ g = \mathit{lens}\ \mathit{getter}\ \mathit{setter}
\end{aligned}
$$

```
  where
  getter x = (view f x, view g x)
  setter x (a, b) = set f a (set g b x)


keeps :: Parameter → Maker () → System → Bool
keeps Fix = maintains (vat ∘ fix)
keeps Par = maintains (vat ∘ par)
keeps Way = maintains (vat ∘ way)
```

Thus:

```
foo sys0 = all (λf → f sys0)
  [changesOnly ((vat ∘ par) `also`
                (vat ∘ way))
    (perform Prod)]
```

Appendix A

Prelude

```
module Maker.Prelude (
  module Maker.Prelude,
  module X
) where


import Prelude as X (
 Conversions to and from strings
  Read (..), Show (..),
 Comparisons
  Eq (..), Ord (..),
 Core abstractions
  Functor      (fmap),
  Applicative (),
  Monad        (return, (>>=)),
 Numeric classes
  Num (..), Integral (), Enum (),
 Numeric conversions
  Real (..), Fractional (..),
  RealFrac (..),
  fromIntegral,
 Simple types
  Integer, Int, String,
 Algebraic types
  Bool    (True, False),
```

Maybe (Just, Nothing),
Either (Right, Left),

Functional operators
$(\circ)$, (\$),

Numeric operators
$(+), (-), (*), (/), (\uparrow), (\uparrow\uparrow), div$,

Utilities
all,

Constants
$mempty, \bot, otherwise$)

We use a typical composition of monad transformers from the `mtl` library to structure stateful actions. See section 5.2 (*The Maker monad*).

import Control.Monad.State *as* X (
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 *as* X (
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 *as* X (
MonadWriter, Type class of monads that emit logs
WriterT, Type constructor that adds logging to a monad type
Writer, Type constructor of logging monads
runWriterT, Runs a writer monad transformer
execWriterT, Runs a writer monad transformer keeping only logs
execWriter) Runs a writer monad keeping only logs

import Control.Monad.Except *as* X (
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 *as* X (
Fixed (..), Type constructor for 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 *as* X (

Lens′,	
lens,	
makeFields,	Defines lenses for record fields
set,	Writes a lens
use, *preuse*,	
zoom,	
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

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 *as* X (

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 *as* X (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.

Appendix B

Rounding fixed point numbers

This somewhat arcane-looking code implements a wrapper around the base library's decimal fixed point type, only with $x * y$ and $x \,/\, y$ operations that do rounding instead of truncation of their intermediate results.

```
module Maker.Decimal (Decimal, E18, E36, Epsilon (..)) where
import Data.Fixed
newtype HasResolution e ⇒ Decimal e = D (Fixed e)
  deriving (Ord, Eq, Real, RealFrac)
```

We want the printed representations of these numbers to look like `"0.01"` and not `"R 0.01"`.

```
instance HasResolution e ⇒ Read (Decimal e) where
  readsPrec n s = fmap (λ(x, y) → (D x, y)) (readsPrec n s)
instance HasResolution e ⇒ Show (Decimal e) where
  show (D x) = show x
```

In the Num instance, we delegate everything except multiplication.

```
instance HasResolution e ⇒ Num (Decimal e) where
  x@(D (MkFixed a)) * D (MkFixed b) =
    D (MkFixed (div (a * b + div (resolution x) 2)
                    (resolution x)))
  D a + D b      = D (a + b)
  D a - D b      = D (a - b)
  negate (D a)   = D (negate a)
  abs    (D a)   = D (abs a)
```

```
  signum (D a) = D (signum a)
  fromInteger i = D (fromInteger i)
```

In the Fractional instance, we delegate everything except division.

```
instance HasResolution e ⇒ Fractional (Decimal e) where
  x@(D (MkFixed a)) / D (MkFixed b) =
    D (MkFixed (div (a ∗ resolution x + div b 2) b))
  recip (D a)       = D (recip a)
  fromRational r = D (fromRational r)
```

We define the E18 and E36 symbols and their fixed point multipliers.

```
data E18; data E36
instance HasResolution E18 where
  resolution _ = 10 ↑ (18 :: Integer)
instance HasResolution E36 where
  resolution _ = 10 ↑ (36 :: Integer)
```

The fixed point number types have well-defined smallest increments (denoted ϵ). This becomes useful when verifying equivalences.

```
class Epsilon t where ϵ :: t
instance HasResolution a ⇒ Epsilon (Decimal a) where
    The use of ⊥ is safe since resolution ignores the value.
  ϵ = 1 / fromIntegral (resolution (⊥ :: Fixed a))
```