MAKER

presents the

REFERENCE IMPLEMENTATION

of the remarkable

DAI CREDIT SYSTEM

issuing a diversely collateralized stablecoin

with last update on March 8, 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 GHC.Exts (Constraint)
```

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

We must 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)
```

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)
```

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"
```

This section introduces the records stored by the Maker system.

3.3 Gem — ERC20 token model

```
data Gem = Gem {
  gemTotalSupply :: Wad,
  gemBalanceOf   :: Map Address            Wad,
  gemAllowance   :: Map (Address, Address) 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      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 units
  urnJam :: Wad             Collateral amount in debt units
  } 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

```
makeFields ''Gem
makeFields ''Jar
makeFields ''Ilk
makeFields ''Urn
makeFields ''Vat
makeFields ''System
```

3.9 Default data

```
defaultIlk :: Id Jar → Ilk
defaultIlk id_jar = Ilk {
  ilkJar  = id_jar,
  ilkAxe  = Ray 1,
  ilkMat  = Ray 1,
  ilkTax  = Ray 1,
  ilkHat  = Wad 0,
  ilkLag  = Sec 0,
  ilkChi  = Ray 1,
  ilkRum  = Wad 0,
  ilkRho  = Sec 0
}
```

```
defaultUrn :: Id Ilk → Address → Urn
defaultUrn id_ilk id_lad = Urn {
  urnVow = Nothing,
  urnCat = Nothing,
  urnLad = id_lad,
  urnIlk = id_ilk,
  urnArt = Wad 0,
  urnJam = Wad 0
}
```

```
initialVat :: Ray → Vat
initialVat how_0 = Vat {
  vatTau  = 0,
  vatFix  = Wad 1,
  vatPar  = Wad 1,
  vatHow  = how_0,
  vatWay  = Ray 1,
  vatJoy  = Wad 0,
  vatSin  = Wad 0,
  vatIlks = ∅,
  vatUrns = ∅,
  vatJars =
    singleton id_DAI Jar {
      jarGem = Gem {
          $gemTotalSupply = 0$,
          $gemBalanceOf\ \ = \varnothing$,
          $gemAllowance\ \ \ = \varnothing$
        },
        $jarTag = \texttt{Wad}\ 0$,
        $jarZzz\ = 0$
      }
}
```

```
$initialSystem :: \texttt{Ray} \to \mathrm{System}$
$initialSystem\ \texttt{how}_0 = \mathrm{System}$ {
  $systemVat\qquad\ \ = initialVat\ \texttt{how}_0$,
  $systemEra\qquad\ \ = 0$,
  $systemSender\ \ \ \ = id_{god}$,
  $systemAccounts = mempty$
}
```

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 Risk 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 $id_{\mathtt{urn}}$ = do
  Perform dai revaluation and rate adjustment
    prod
  Update price of specific debt unit
    $id_{\mathtt{ilk}}$ ← $look$ (vat ∘ urns ∘ $ix$ $id_{\mathtt{urn}}$ ∘ ilk)
    drip $id_{\mathtt{ilk}}$
  Read parameters for risk analysis
    $\mathtt{era}_0$ ← $view$ era
    $\mathtt{par}_0$ ← $view$ (vat ∘ par)
    $\mathtt{urn}_0$ ← $look$ (vat ∘ urns ∘ $ix$ $id_{\mathtt{urn}}$)
    $\mathtt{ilk}_0$ ← $look$ (vat ∘ ilks ∘ $ix$ ($view$ ilk $\mathtt{urn}_0$))
    $\mathtt{jar}_0$ ← $look$ (vat ∘ jars ∘ $ix$ ($view$ jar $\mathtt{ilk}_0$))
  Return risk stage of CDP
    $return$ ($analyze$ $\mathtt{era}_0$ $\mathtt{par}_0$ $\mathtt{urn}_0$ $\mathtt{ilk}_0$ $\mathtt{jar}_0$)
```

4.2 Lending

```
open $id_{\mathtt{urn}}$ $id_{\mathtt{ilk}}$ =
  do
    $id_{\mathtt{lad}}$ ← $view$ $sender$
    vat ∘ urns ∘ $at$ $id_{\mathtt{urn}}$ ?= $defaultUrn$ $id_{\mathtt{ilk}}$ $id_{\mathtt{lad}}$
```

```
lock $id_{\mathtt{urn}}$ $x$ = do
    Ensure CDP exists; identify collateral type
    $id_{\texttt{ilk}} \leftarrow look\ (\texttt{vat} \circ \texttt{urn}s \circ ix\ id_{\texttt{urn}} \circ \texttt{ilk})$
    $id_{\texttt{jar}} \leftarrow look\ (\texttt{vat} \circ \texttt{ilk}s \circ ix\ id_{\texttt{ilk}} \circ \texttt{jar})$
  Record an increase in collateral
    $\texttt{vat} \circ \texttt{urn}s \circ ix\ id_{\texttt{urn}} \circ \texttt{jam} \mathrel{+}= x$
  Take sender's tokens
    $id_{\texttt{lad}} \leftarrow view\ sender$
    $\texttt{pull}\ id_{\texttt{jar}}\ id_{\texttt{lad}}\ x$
```

```
$\texttt{free}\ id_{\texttt{urn}}\ \texttt{wad}_{\texttt{gem}} = \textbf{do}$
  Fail if sender is not the CDP owner
    $id_{sender} \leftarrow view\ sender$
    $id_{\texttt{lad}}\quad \leftarrow look\ (\texttt{vat} \circ \texttt{urn}s \circ ix\ id_{\texttt{urn}} \circ \texttt{lad})$
    $\texttt{aver}\ (id_{sender} \equiv id_{\texttt{lad}})$
  Decrease the collateral amount
    $\texttt{vat} \circ \texttt{urn}s \circ ix\ id_{\texttt{urn}} \circ \texttt{jam} \mathrel{-}= \texttt{wad}_{\texttt{gem}}$
  Roll back if undercollateralized
    $\texttt{gaze}\ id_{\texttt{urn}} \ggg \texttt{aver} \circ (\equiv \texttt{Pride})$
  Send the collateral to the CDP owner
    $id_{\texttt{ilk}} \leftarrow look\ (\texttt{vat} \circ \texttt{urn}s \circ ix\ id_{\texttt{urn}} \circ \texttt{ilk})$
    $id_{\texttt{jar}} \leftarrow look\ (\texttt{vat} \circ \texttt{ilk}s \circ ix\ id_{\texttt{ilk}} \circ \texttt{jar})$
    $\texttt{push}\ id_{\texttt{jar}}\ id_{\texttt{lad}}\ \texttt{wad}_{\texttt{gem}}$
```

```
$\texttt{draw}\ id_{\texttt{urn}}\ \texttt{wad}_{\text{DAI}} = \textbf{do}$
  Fail if sender is not the CDP owner
    $id_{sender} \leftarrow view\ sender$
    $id_{\texttt{lad}}\quad \leftarrow look\ (\texttt{vat} \circ \texttt{urn}s \circ ix\ id_{\texttt{urn}} \circ \texttt{lad})$
    $\texttt{aver}\ (id_{sender} \equiv id_{\texttt{lad}})$
  Update value of debt unit
    $id_{\texttt{ilk}}\quad \leftarrow look\ (\texttt{vat} \circ \texttt{urn}s \circ ix\ id_{\texttt{urn}} \circ \texttt{ilk})$
    $\texttt{chi}_1\quad \leftarrow \texttt{drip}\ id_{\texttt{ilk}}$
  Denominate draw amount in debt unit
    $\textbf{let}\ \texttt{wad}_{\texttt{chi}} = \texttt{wad}_{\text{DAI}}\ /\ cast\ \texttt{chi}_1$
  Increase debt
    $\texttt{vat} \circ \texttt{urn}s \circ ix\ id_{\texttt{urn}} \circ \texttt{art} \mathrel{+}= \texttt{wad}_{\texttt{chi}}$
  Roll back unless overcollateralized
```

```
    gaze $id_{\mathtt{urn}}$ $\ggg$= aver $\circ$ ($\equiv$ Pride)
  Mint dai and send to the CDP owner
    mint $id_{\text{DAI}}$ $\mathtt{wad}_{\text{DAI}}$
    push $id_{\text{DAI}}$ $id_{\mathtt{lad}}$ $\mathtt{wad}_{\text{DAI}}$

wipe $id_{\mathtt{urn}}$ $\mathtt{wad}_{\text{DAI}}$ = do
  Fail if sender is not the CDP owner
    $id_{sender}$ $\leftarrow$ $view\ sender$
    $id_{\mathtt{lad}}$     $\leftarrow$ $look$ (vat $\circ$ urn$s$ $\circ$ $ix$ $id_{\mathtt{urn}}$ $\circ$ lad)
    aver ($id_{sender} \equiv id_{\mathtt{lad}}$)
  Update value of debt unit
    $id_{\mathtt{ilk}}$ $\leftarrow$ $look$ (vat $\circ$ urn$s$ $\circ$ $ix$ $id_{\mathtt{urn}}$ $\circ$ ilk)
    $\mathtt{chi}_1$ $\leftarrow$ drip $id_{\mathtt{ilk}}$
  Roll back unless overcollateralized
    gaze $id_{\mathtt{urn}}$ $\ggg$= aver $\circ$ ($\equiv$ Pride)
  Denominate dai amount in debt unit
    let $\mathtt{wad}_{\mathtt{chi}}$ = $\mathtt{wad}_{\text{DAI}}$ / $cast$ $\mathtt{chi}_1$
  Reduce debt
    vat $\circ$ urn$s$ $\circ$ $ix$ $id_{\mathtt{urn}}$ $\circ$ art $-$= $\mathtt{wad}_{\mathtt{chi}}$
  Take dai from CDP owner, or roll back
    pull $id_{\text{DAI}}$ $id_{\mathtt{lad}}$ $\mathtt{wad}_{\text{DAI}}$
  Destroy dai
    burn $id_{\text{DAI}}$ $\mathtt{wad}_{\text{DAI}}$

give $id_{\mathtt{urn}}$ $id_{\mathtt{lad}}$ = do
  $x$ $\leftarrow$ $look$ (vat $\circ$ urn$s$ $\circ$ $ix$ $id_{\mathtt{urn}}$ $\circ$ lad)
  $y$ $\leftarrow$ $view\ sender$
  aver ($x \equiv y$)
  vat $\circ$ urn$s$ $\circ$ $ix$ $id_{\mathtt{urn}}$ $\circ$ lad := $id_{\mathtt{lad}}$

shut $id_{\mathtt{urn}}$ = do
  Update value of debt unit
    $id_{\mathtt{ilk}}$ $\leftarrow$ $look$ (vat $\circ$ urn$s$ $\circ$ $ix$ $id_{\mathtt{urn}}$ $\circ$ ilk)
    $\mathtt{chi}_1$ $\leftarrow$ drip $id_{\mathtt{ilk}}$
```

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

4.3 Frequent adjustments

```
prod = 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 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)
```

```
$\texttt{drip}\ id_{\texttt{ilk}} = \textbf{do}$
  Current time stamp
   $\texttt{era}_0 \leftarrow view\ \texttt{era}$
  Time stamp of previous drip
   $\texttt{rho}_0 \leftarrow look\ (\texttt{vat} \circ \texttt{ilk}s \circ ix\ id_{\texttt{ilk}} \circ \texttt{rho})$
  Current stability fee
   $\texttt{tax}_0 \leftarrow look\ (\texttt{vat} \circ \texttt{ilk}s \circ ix\ id_{\texttt{ilk}} \circ \texttt{tax})$
  Current value of debt unit
   $\texttt{chi}_0 \leftarrow look\ (\texttt{vat} \circ \texttt{ilk}s \circ ix\ id_{\texttt{ilk}} \circ \texttt{chi})$
  Current total debt in debt units
   $rum0 \leftarrow look\ (\texttt{vat} \circ \texttt{ilk}s \circ ix\ id_{\texttt{ilk}} \circ \texttt{rum})$
  Current unprocessed stability fee revenue
   $\texttt{joy}_0 \leftarrow look\ (\texttt{vat} \circ \texttt{ilk}s \circ ix\ id_{\texttt{ilk}} \circ \texttt{joy})$
   $\textbf{let}$
     $age = \texttt{era}_0 - \texttt{rho}_0$
     $\texttt{chi}_1 = \texttt{chi}_0 * \texttt{tax}_0 \uparrow\uparrow age$
     $\texttt{joy}_1 = \texttt{joy}_0 + (cast\ (\texttt{chi}_1 - \texttt{chi}_0) :: \texttt{Wad}) * rum0$
   $\texttt{vat} \circ \texttt{ilk}s \circ ix\ id_{\texttt{ilk}} \circ \texttt{chi} := \texttt{chi}_1$
   $\texttt{vat} \circ \texttt{ilk}s \circ ix\ id_{\texttt{ilk}} \circ \texttt{rho} := \texttt{era}_0$
   $\texttt{vat} \circ \texttt{ilk}s \circ ix\ id_{\texttt{ilk}} \circ \texttt{joy} := \texttt{joy}_1$
   $return\ \texttt{chi}_1$
```

4.4 Price feedback

```
$\texttt{mark}\ id_{\texttt{jar}}\ \texttt{tag}_1\ \texttt{zzz}_1 =$
  $\texttt{auth}\ \$\ \textbf{do}$
    $jarAt\ id_{\texttt{jar}} \circ \texttt{tag} := \texttt{tag}_1$
    $jarAt\ id_{\texttt{jar}} \circ \texttt{zzz} := \texttt{zzz}_1$
```

```
$\texttt{tell}\ x =$
  $\texttt{auth}\ \$\ \textbf{do}$
    $\texttt{vat} \circ \texttt{fix} := x$
```

4.5 Liquidation and settlement

```
bite $id_{\texttt{urn}}$ = do
    Fail if CDP is not in need of liquidation
      gaze $id_{\texttt{urn}}$ $\ggg$= aver $\circ$ ($\equiv$ Panic)
    Record the sender as the liquidation initiator
      $id_{\texttt{cat}} \leftarrow$ view sender
      vat $\circ$ urns $\circ$ ix $id_{\texttt{urn}}$ $\circ$ cat := $id_{\texttt{cat}}$
    Read current debt
      $\texttt{art}_0$      $\leftarrow$ look (vat $\circ$ urns $\circ$ ix $id_{\texttt{urn}}$ $\circ$ art)
    Update value of debt unit
      $id_{\texttt{ilk}} \leftarrow$ look (vat $\circ$ urns $\circ$ ix $id_{\texttt{urn}}$ $\circ$ ilk)
      $\texttt{chi}_1 \leftarrow$ drip $id_{\texttt{ilk}}$
    Read liquidation penalty ratio
      $id_{\texttt{ilk}}$      $\leftarrow$ look (vat $\circ$ urns $\circ$ ix $id_{\texttt{urn}}$ $\circ$ ilk)
      $\texttt{axe}_0$      $\leftarrow$ look (vat $\circ$ ilks $\circ$ ix $id_{\texttt{ilk}}$ $\circ$ axe)
    Apply liquidation penalty to debt
      let $\texttt{art}_1 = \texttt{art}_0 * \texttt{axe}_0$
    Update CDP debt
      vat $\circ$ urns $\circ$ ix $id_{\texttt{urn}}$ $\circ$ art := $\texttt{art}_1$
    Record as bad debt
      sin += $\texttt{art}_1 * \texttt{chi}_1$
```

```
grab $id_{\texttt{urn}}$ =
  auth $ do
    Fail if CDP is not marked for liquidation
      gaze $id_{\texttt{urn}}$ $\ggg$= aver $\circ$ ($\equiv$ Grief)
    Record the sender as the CDP's settler
      $id_{\texttt{vow}} \leftarrow$ view sender
      vat $\circ$ urns $\circ$ ix $id_{\texttt{urn}}$ $\circ$ vow := $id_{\texttt{vow}}$
    Clear the CDP's requester of liquidation
      vat $\circ$ urns $\circ$ ix $id_{\texttt{urn}}$ $\circ$ cat := Nothing
```

```
heal $\texttt{wad}_{\text{DAI}}$ =
  auth $ do
    vat ∘ sin −= wad_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 $id_{\texttt{ilk}}$ $\texttt{mat}_1$ =
  auth $ do
    vat ∘ ilks ∘ ix $id_{\texttt{ilk}}$ ∘ mat := $\texttt{mat}_1$
```

```
crop $id_{\texttt{ilk}}$ $\texttt{tax}_1$ =
  auth $ do
    drip $id_{\texttt{ilk}}$
    vat ∘ ilks ∘ ix $id_{\texttt{ilk}}$ ∘ tax := $\texttt{tax}_1$
```

4.7 Minting, burning, and transferring

```
pull $id_{\texttt{jar}}$ $id_{\texttt{lad}}$ $w$ = do
  $g$  ← look (jarAt $id_{\texttt{jar}}$ ∘ gem)
  $g'$ ← transferFrom $id_{\texttt{lad}}$ $id_{\texttt{vat}}$ $w$ $g$
  jarAt $id_{\texttt{jar}}$ ∘ gem := $g'$
```

```
push $id_{\texttt{jar}}$ $id_{\texttt{lad}}$ $w$ = do
  $g$  ← look (jarAt $id_{\texttt{jar}}$ ∘ gem)
  $g'$ ← transferFrom $id_{\texttt{vat}}$ $id_{\texttt{lad}}$ $w$ $g$
  jarAt $id_{\texttt{jar}}$ ∘ gem := $g'$
```

```
mint $id_{\texttt{jar}}$ $\texttt{wad}_0$ = do
  jarAt $id_{\texttt{jar}}$ ∘ gem ∘ totalSupply                  += $\texttt{wad}_0$
  jarAt $id_{\texttt{jar}}$ ∘ gem ∘ balanceOf ∘ ix $id_{\texttt{vat}}$ += $\texttt{wad}_0$
```

```
burn $id_{\texttt{jar}}$ $\texttt{wad}_0$ = do
  jarAt $id_{\texttt{jar}}$ ∘ gem ∘ totalSupply                  −= $\texttt{wad}_0$
  jarAt $id_{\texttt{jar}}$ ∘ gem ∘ balanceOf ∘ ix $id_{\texttt{vat}}$ −= $\texttt{wad}_0$
```

4.8 Test-related manipulation

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

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
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 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
 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 | 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)
```

The following instance makes the mutable state also available as read-only state.

```
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
```

5.3 Accessor aliases

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

5.4 Asserting

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

5.5 Modifiers

```
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 6

Testing

Sketches for property stuff...

```
data Parameter =
  Fix | Par | Way

maintains
  :: Eq a ⇒ Lens′ System a → Maker ()
           → System → Bool
maintains p = λm sys_0 →
  case exec sys_0 m of
    On success, data must be compared for equality
    Right sys_1 → view p sys_0 ≡ view p sys_1
    On rollback, data is maintained by definition
    Left _      → True

changesOnly
  ::  Lens′ System a → Maker ()
  → System → Bool
changesOnly p = λm sys_0 →
  case exec sys_0 m of
    On success, equalize p and compare
    Right sys_1 → set p (view p sys_1) sys_0 ≡ sys_1
    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
    $getter\ x = (view\ f\ x, view\ g\ x)$
    $setter\ x\ (a, b) = set\ f\ a\ (set\ g\ b\ x)$
```

```
$keeps :: \text{Parameter} \to \text{Maker}\ () \to \text{System} \to \text{Bool}$
$keeps$ Fix $= maintains$ (vat $\circ$ fix)
$keeps$ Par $= maintains$ (vat $\circ$ par)
$keeps$ Way $= maintains$ (vat $\circ$ way)
```

Thus:

```
$foo$ sys$_0$ $= all\ (\lambda f \to f$ sys$_0)$
    [$changesOnly$ ((vat $\circ$ par) `$also$`
                    (vat $\circ$ 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 (toRational), Fractional (fromRational),
  RealFrac (truncate),
  fromIntegral,
 Simple types
  Integer, Int, String,
 Algebraic types
  Bool   (True, False),
    Maybe (Just, Nothing),
    Either  (Right, Left),
  Functional operators
    (∘), ($),
  Numeric operators
    (+), (−), (∗), (/), (↑), (↑↑),
  Utilities
    all,
  Constants
    mempty, ⊥, 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
  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

Act type signatures

```
type Reads  r  m = MonadReader r m
type Writes w  m = MonadState w m
type Fails     m = MonadError Error m
type IsAct = ?act :: Act
```

```
type Numbers wad ray sec =
  (wad~Wad, ray~Ray, sec~Sec)
```

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 Sec,
     HasVat r vat_r,
       HasIlks vat_r (Map (Id Ilk) ilk_r),
         HasTax ilk_r Ray,
         HasRho ilk_r Sec,
         HasChi ilk_r Ray,
         HasRum ilk_r Wad,
         HasJoy ilk_r Wad,
   Writes w m,
     HasVat w vat_w,
       HasIlks vat_w (Map (Id Ilk) ilk_w),
         HasRho ilk_w Sec,
```

$$\begin{aligned}
&\qquad\qquad \text{HasJoy}\ \texttt{ilk}_w\ \texttt{Wad},\\
&\qquad\qquad \text{HasChi}\ \texttt{ilk}_w\ \texttt{Ray})\\
&\quad \Rightarrow \text{Id}\ \texttt{Ilk} \to m\ \texttt{Ray}
\end{aligned}$$

$$\begin{aligned}
&\texttt{form} ::\\
&\quad (\text{IsAct}, \text{Fails}\ m,\\
&\quad\ \ \text{Reads}\ r\ m,\ \ \text{HasSender}\ r\ \text{Address},\\
&\quad\ \ \text{Writes}\ w\ m, \text{HasVat}\ w\ \texttt{vat}_w,\\
&\qquad\qquad\qquad \text{HasIlks}\ \texttt{vat}_w\ (\text{Map}\ (\text{Id}\ \texttt{Ilk})\ \texttt{Ilk}))\\
&\quad \Rightarrow \text{Id}\ \texttt{Ilk} \to \text{Id}\ \texttt{Jar} \to m\ ()
\end{aligned}$$

$$\begin{aligned}
&\texttt{frob} :: (\text{IsAct}, \text{Fails}\ m,\\
&\qquad\quad \text{Reads}\ r\ m,\ \ \text{HasSender}\ r\ \text{Address},\\
&\qquad\quad \text{Writes}\ w\ m, \text{HasVat}\ w\ \texttt{vat}_w,\\
&\qquad\qquad\qquad\quad \text{HasHow}\ \texttt{vat}_w\ \texttt{ray})\\
&\quad \Rightarrow \texttt{ray} \to m\ ()
\end{aligned}$$

$$\begin{aligned}
&\texttt{open} ::\\
&\quad (\text{IsAct},\\
&\quad\ \ \text{Reads}\ r\ m,\ \ \text{HasSender}\ r\ \text{Address},\\
&\quad\ \ \text{Writes}\ w\ m, \text{HasVat}\ w\ \texttt{vat}_w,\\
&\qquad\qquad\qquad \text{HasUrns}\ \texttt{vat}_w\ (\text{Map}\ (\text{Id}\ \texttt{Urn})\ \texttt{Urn}))\\
&\quad \Rightarrow \text{Id}\ \texttt{Urn} \to \text{Id}\ \texttt{Ilk} \to m\ ()
\end{aligned}$$

$$\begin{aligned}
&\texttt{give} ::\\
&\quad (\text{IsAct}, \text{Fails}\ m,\\
&\quad\ \ \text{Reads}\ r\ m,\ \ \text{HasSender}\ r\ \text{Address},\\
&\qquad\qquad\quad \text{HasVat}\ r\ \texttt{vat}_r,\\
&\qquad\qquad\qquad \text{HasUrns}\ \texttt{vat}_r\ (\text{Map}\ (\text{Id}\ \texttt{Urn})\ \texttt{urn}_r),\\
&\qquad\qquad\qquad\quad \text{HasLad}\ \texttt{urn}_r\ \text{Address},\\
&\quad\ \ \text{Writes}\ w\ m, \text{HasVat}\ w\ \texttt{vat}_r)\\
&\quad \Rightarrow \text{Id}\ \texttt{Urn} \to \text{Address} \to m\ ()
\end{aligned}$$

$$\begin{aligned}
&\texttt{lock} ::\\
&\quad (\text{IsAct}, \text{Fails}\ m,\\
&\quad\ \ \text{Reads}\ r\ m,\\
&\qquad \text{HasSender}\ r\ \text{Address},\\
&\qquad \text{HasVat}\ r\ \texttt{vat}_r,
\end{aligned}$$

```
          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,
    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 sec)
   ⇒ Id Jar → wad → sec → m ()


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


prod ::
   (IsAct,
    Reads r m,
      HasSender r Address,
      HasEra r sec,
      HasVat r vat_r,  (HasPar vat_r wad,
                        HasTau vat_r sec,
                        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 sec),
    Integral sec,
    Ord wad, Fractional wad,
    Fractional ray, Real ray)
    ⇒ m ()


warp ::
  (IsAct, Fails m,
    Reads r m,  HasSender r Address,
    Writes w m, HasEra w sec,
                    Num sec)
    ⇒ sec → 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,
   Numbers wad ray sec,
   Reads r m,
     HasSender r Address,
     HasEra r Sec,
     HasVat r vat_r,
       HasFix vat_r wad,
       HasPar vat_r wad,
       HasHow vat_r ray,
       HasWay vat_r ray,
       HasTau vat_r sec,
       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,
         HasRum ilk_r wad,
         HasJoy ilk_r wad,
         HasTax ilk_r ray,
         HasLag ilk_r sec,
         HasChi ilk_r ray,  HasRho ilk_r sec,
         HasJar ilk_r (Id Jar),
       HasJars vat_r (Map (Id Jar) jar_r),
            HasGem jar_r Gem,
            HasTag jar_r wad,
            HasZzz jar_r sec,
  Writes w m,
    HasVat w vat_w,
      HasTau vat_w sec,
      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 vat_w (Map (Id Ilk) ilk_w),
        HasJoy ilk_w wad,
        HasChi ilk_w ray,
        HasRho ilk_w sec,
      HasJars vat_w (Map (Id Jar) jar_r)
) ⇒ Id Urn → m ()
```