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**
  $resolution\ \_ = 10 \uparrow (18 :: \text{Integer})$
instance HasResolution E36 **where**
  $resolution\ \_ = 10 \uparrow (36 :: \text{Integer})$

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.
$cast :: (\text{Real } a, \text{Fractional } b) \Rightarrow a \rightarrow b$
$cast = fromRational \circ toRational$

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 biting cat
  urnVow :: Maybe Address,   Address of liquidating vow
  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

$defaultIlk :: \text{Id Jar} \rightarrow \text{Ilk}$
$defaultIlk\ id_{\texttt{jar}} = \text{Ilk \{}$
   $ilkJar\quad = id_{\texttt{jar}},$
   $ilkAxe\ = \text{Ray } 1,$
   $ilkMat\ = \text{Ray } 1,$
   $ilkTax\ = \text{Ray } 1,$
   $ilkHat\ = \text{Wad } 0,$
   $ilkLag\ = \text{Sec } 0,$
   $ilkChi\ = \text{Ray } 1,$
   $ilkRum = \text{Wad } 0,$
   $ilkRho\ = \text{Sec } 0$
}

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

$initialVat :: \text{Ray} \rightarrow \text{Vat}$
$initialVat\ \texttt{how}_0 = \text{Vat \{}$
   $vatTau\ = 0,$
   $vatFix\quad = \text{Wad } 1,$
   $vatPar\ = \text{Wad } 1,$
   $vatHow\ = \texttt{how}_0,$
   $vatWay\ = \text{Ray } 1,$
   $vatJoy\quad = \text{Wad } 0,$
   $vatSin\quad = \text{Wad } 0,$
   $vatIlks\quad = \varnothing,$
   $vatUrns = \varnothing,$
   $vatJars\ =$
     $singleton\ id_{\text{DAI}}\ \text{Jar \{}$
       $jarGem = \text{Gem \{}$

$$
\begin{aligned}
&\qquad gemTotalSupply = 0, \\
&\qquad gemBalanceOf \;\; = \varnothing, \\
&\qquad gemAllowance \;\; = \varnothing \\
&\quad \}, \\
&\quad jarTag = \texttt{Wad}\ 0, \\
&\quad jarZzz \;\; = 0 \\
&\;\} \\
\}&
\end{aligned}
$$

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

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   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.

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

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_{\mathrm{urn}} = $ **do**

  Perform dai revaluation and rate adjustment
   prod

  Update price of specific debt unit
   $id_{\mathrm{ilk}} \leftarrow look \ (\mathrm{vat} \circ \mathrm{urn}s \circ ix \ id_{\mathrm{urn}} \circ \mathrm{ilk})$
   drip id_{ilk}

  Read parameters for risk analysis
   $\mathrm{era}_0 \leftarrow view \ \mathrm{era}$
   $\mathrm{par}_0 \leftarrow view \ (\mathrm{vat} \circ \mathrm{par})$
   $\mathrm{urn}_0 \leftarrow look \ (\mathrm{vat} \circ \mathrm{urn}s \circ ix \ id_{\mathrm{urn}})$
   $\mathrm{ilk}_0 \leftarrow look \ (\mathrm{vat} \circ \mathrm{ilk}s \circ ix \ (view \ \mathrm{ilk} \ \mathrm{urn}_0))$
   $\mathrm{jar}_0 \leftarrow look \ (\mathrm{vat} \circ \mathrm{jar}s \circ ix \ (view \ \mathrm{jar} \ \mathrm{ilk}_0))$

  Return risk stage of CDP
   $return \ (analyze \ \mathrm{era}_0 \ \mathrm{par}_0 \ \mathrm{urn}_0 \ \mathrm{ilk}_0 \ \mathrm{jar}_0)$

4.2 Lending

open $id_{\mathrm{urn}} \ id_{\mathrm{ilk}} = $
  do
   $id_{\mathrm{lad}} \leftarrow view \ sender$
   $\mathrm{vat} \circ \mathrm{urn}s \circ at \ id_{\mathrm{urn}} \ ?= \ defaultUrn \ id_{\mathrm{ilk}} \ id_{\mathrm{lad}}$

lock $id_{\mathrm{urn}} \ x = $ **do**

  Ensure CDP exists; identify collateral type

17

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

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

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

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
$$\textbf{let}\ \texttt{wad}_{\texttt{chi}} = \texttt{wad}_{\texttt{DAI}}\ /\ cast\ \texttt{chi}_1$$

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

$\quad$ gaze $id_\mathrm{urn} \ggeq$ aver $\circ\ (\equiv \mathtt{Pride})$

$\quad$ Mint dai and send to the CDP owner
$\quad\quad$ mint id_DAI wad$_\mathrm{DAI}$
$\quad\quad$ push $id_\mathrm{DAI}\ id_\mathrm{lad}$ wad$_\mathrm{DAI}$

wipe id_urn wad$_\mathrm{DAI} = \mathbf{do}$

$\quad$ Fail if sender is not the CDP owner
$\quad\quad id_{sender} \leftarrow view\ sender$
$\quad\quad id_\mathrm{lad}\quad \leftarrow look\ (\mathtt{vat} \circ \mathrm{urn}s \circ ix\ id_\mathrm{urn} \circ \mathtt{lad})$
$\quad\quad$ aver $(id_{sender} \equiv id_\mathrm{lad})$

$\quad$ Update value of debt unit
$\quad\quad id_\mathrm{ilk} \leftarrow look\ (\mathtt{vat} \circ \mathrm{urn}s \circ ix\ id_\mathrm{urn} \circ \mathtt{ilk})$
$\quad\quad$ chi$_1 \leftarrow$ drip id_ilk

$\quad$ Roll back unless overcollateralized
$\quad\quad$ gaze $id_\mathrm{urn} \ggeq$ aver $\circ\ (\equiv \mathtt{Pride})$

$\quad$ Denominate dai amount in debt unit
$\quad\quad \mathbf{let}$ wad$_\mathrm{chi}$ = wad$_\mathrm{DAI}\ /\ cast$ chi$_1$

$\quad$ Reduce debt
$\quad\quad$ vat $\circ \mathrm{urn}s \circ ix\ id_\mathrm{urn} \circ$ art $-\!\!= $ wad$_\mathrm{chi}$

$\quad$ Take dai from CDP owner, or roll back
$\quad\quad$ pull $id_\mathrm{DAI}\ id_\mathrm{lad}$ wad$_\mathrm{DAI}$

$\quad$ Destroy dai
$\quad\quad$ burn id_DAI wad$_\mathrm{DAI}$

give $id_\mathrm{urn}\ id_\mathrm{lad} = \mathbf{do}$

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

shut $id_\mathrm{urn} = \mathbf{do}$

$\quad$ Update value of debt unit
$\quad\quad id_\mathrm{ilk} \leftarrow look\ (\mathtt{vat} \circ \mathrm{urn}s \circ ix\ id_\mathrm{urn} \circ \mathtt{ilk})$
$\quad\quad$ chi$_1 \leftarrow$ drip id_ilk

Attempt to repay all the CDP's outstanding dai

$\text{art}_0 \leftarrow look \ (\text{vat} \circ \text{urn}s \circ ix \ id_{\text{urn}} \circ \text{art})$

$\text{wipe} \ id_{\text{urn}} \ (\text{art}_0 * cast \ \text{chi}_1)$

Reclaim all the collateral

$jam0 \leftarrow look \ (\text{vat} \circ \text{urn}s \circ ix \ id_{\text{urn}} \circ \text{jam})$

$\text{free} \ id_{\text{urn}} \ jam0$

Nullify the CDP

$\text{vat} \circ \text{urn}s \circ at \ id_{\text{urn}} := \text{Nothing}$

4.3   Frequent adjustments

$\text{prod} = \textbf{do}$

$\quad \text{era}_0 \leftarrow \textit{view } \text{era}$

$\quad \text{tau}_0 \leftarrow \textit{view } (\text{vat} \circ \text{tau})$

$\quad \text{fix}_0 \leftarrow \textit{view } (\text{vat} \circ \text{fix})$

$\quad \text{par}_0 \leftarrow \textit{view } (\text{vat} \circ \text{par})$

$\quad \text{how}_0 \leftarrow \textit{view } (\text{vat} \circ \text{how})$

$\quad \text{way}_0 \leftarrow \textit{view } (\text{vat} \circ \text{way})$

$\quad \textbf{let}$

Time difference in seconds

$\quad\quad age = \text{era}_0 - \text{tau}_0$

Current target rate applied to target price

$\quad\quad \text{par}_1 = \text{par}_0 * \textit{cast } (\text{way}_0 \uparrow\uparrow age)$

Sensitivity parameter applied over time

$\quad\quad wag = \text{how}_0 * \textit{fromIntegral } age$

Target rate scaled up or down

$\quad\quad \text{way}_1 = \textit{inj } (\textit{prj } \text{way}_0 +$
$\quad\quad\quad\quad\quad\quad \textbf{if } \text{fix}_0 < \text{par}_0 \textbf{ then } \textit{wag} \textbf{ else } - \textit{wag})$

$\quad \text{vat} \circ \text{par} := \text{par}_1$

$\quad \text{vat} \circ \text{way} := \text{way}_1$

$\quad \text{vat} \circ \text{tau} := \text{era}_0$

$\quad \textbf{where}$

Convert between multiplicative and additive form

$\quad\quad \textit{prj } x \quad = \textbf{if } x \geqslant 1 \textbf{ then } x - 1 \textbf{ else } 1 - 1 \mathbin{/} x$

$\quad\quad \textit{inj } x \quad = \textbf{if } x \geqslant 0 \textbf{ then } x + 1 \textbf{ else } 1 \mathbin{/} (1 - x)$

```
drip id_ilk = do
  Current time stamp
    era_0  ← view 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 units
    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 Governance

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



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

4.5 Price feedback

```
mark id_jar tag_1 zzz_1 =
   auth $ do
      jarAt id_jar ∘ tag := tag_1
      jarAt id_jar ∘ zzz := zzz_1
```

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

4.6 Liquidation and settlement

```
bite id_urn = do
```

Fail if urn is not undercollateralized
$$\text{gaze } id_{\text{urn}} \ggg \text{aver} \circ (\equiv \text{Panic})$$

Record the sender as the requester of liquidation
$$id_{\text{cat}} \leftarrow view\ sender$$
$$\text{vat} \circ \text{urn}s \circ ix\ id_{\text{urn}} \circ \text{cat} := id_{\text{cat}}$$

Read current debt
$$\text{art}_0 \leftarrow look\ (\text{vat} \circ \text{urn}s \circ ix\ id_{\text{urn}} \circ \text{art})$$

Update value of debt unit
$$id_{\text{ilk}} \leftarrow look\ (\text{vat} \circ \text{urn}s \circ ix\ id_{\text{urn}} \circ \text{ilk})$$
$$\text{chi}_1 \leftarrow \text{drip}\ id_{\text{ilk}}$$

Read liquidation penalty ratio
$$id_{\text{ilk}} \leftarrow look\ (\text{vat} \circ \text{urn}s \circ ix\ id_{\text{urn}} \circ \text{ilk})$$
$$\text{axe}_0 \leftarrow look\ (\text{vat} \circ \text{ilk}s \circ ix\ id_{\text{ilk}} \circ \text{axe})$$

Apply liquidation penalty to debt
$$\textbf{let } \text{art}_1 = \text{art}_0 * \text{axe}_0$$

Update debt and record it as in need of settlement
$$\text{vat} \circ \text{urn}s \circ ix\ id_{\text{urn}} \circ \text{art} := \text{art}_1$$
$$\text{sin} \mathrel{+}= \text{art}_1 * \text{chi}_1$$

```
grab id_urn =
  auth $ 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
      vat ∘ urns ∘ ix id_urn ∘ vow := id_vow
    Clear the CDP's requester of liquidation
      vat ∘ urns ∘ ix id_urn ∘ cat := Nothing



heal wad_DAI =
  auth $ do
    vat ∘ sin −= wad_DAI



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

4.7   Minting, burning, and transferring

```
pull id_jar id_lad w = do
  g  ← look (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  ← look (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
```

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)
```

27

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
    $\rightarrow$ Maker ()
    $\rightarrow$ 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**
      $s \leftarrow get; put\ (f\ s)$
      $x \leftarrow m;\ put\ s$
      $return\ x$

5.3 Accessor aliases

$ilkAt$ id = `vat` $\circ$ `ilk`$s \circ ix\ id$
$urnAt$ id = `vat` $\circ$ `urn`$s \circ ix\ id$
$jarAt$ id = `vat` $\circ$ `jar`$s \circ ix\ id$

5.4   Asserting

$\text{aver} :: \text{Fails } m \Rightarrow \text{Bool} \rightarrow m\ ()$
$\text{aver } x = unless\ x\ (throwError\ \text{AssertError})$

$look :: (\text{Fails } m, \text{Reads } r\ m)$
$\quad \Rightarrow \text{Getting (First } a)\ r\ a \rightarrow m\ a$
$look\ f = preview\ f \ggg \lambda\textbf{case}$
$\quad \text{Nothing} \rightarrow throwError\ \text{AssertError}$
$\quad \text{Just } x \rightarrow return\ x$

5.5   Modifiers

$\text{auth} ::$
$\quad (\text{IsAct}, \text{Fails } m,$
$\quad\ \text{Reads } r\ m,$
$\quad\quad \text{HasSender } r\ \text{Address})$
$\quad \Rightarrow m\ a \rightarrow m\ a$

$\text{auth } continue = \textbf{do}$
$\quad s \leftarrow view\ sender$
$\quad unless\ (s \equiv id_{god})$
$\quad\quad (throwError\ \text{AuthError})$
$\quad continue$

Chapter 6

Testing

Sketches for property stuff...

> **data** Parameter =
>   Fix | Par | Way
>
> *maintains*
>   :: Eq a $\Rightarrow$ Lens$'$ System a $\rightarrow$ Maker ()
> $\rightarrow$ System $\rightarrow$ Bool
> *maintains* p = λm $\mathbf{sys}_0$ $\rightarrow$
>   **case** *exec* $\mathbf{sys}_0$ m **of**
>   On success, data must be compared for equality
>   Right $\mathbf{sys}_1$ $\rightarrow$ *view* p $\mathbf{sys}_0$ $\equiv$ *view* p $\mathbf{sys}_1$
>   On rollback, data is maintained by definition
>   Left $\_$ $\rightarrow$ True
>
> *changesOnly*
>   :: Lens$'$ System a $\rightarrow$ Maker ()
> $\rightarrow$ System $\rightarrow$ Bool
> *changesOnly* p = λm $\mathbf{sys}_0$ $\rightarrow$
>   **case** *exec* $\mathbf{sys}_0$ m **of**
>   On success, equalize p and compare
>   Right $\mathbf{sys}_1$ $\rightarrow$ *set* p (*view* p $\mathbf{sys}_1$) $\mathbf{sys}_0$ $\equiv$ $\mathbf{sys}_1$
>   On rollback, data is maintained by definition
>   Left $\_$ $\rightarrow$ True
>
> *also* :: Lens$'$ s a $\rightarrow$ Lens$'$ s b $\rightarrow$ Lens$'$ s (a, b)
> *also f g = lens getter setter*

30

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

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

Thus:

$foo\ \text{sys}_0 = all\ (\lambda f \rightarrow f\ \text{sys}_0)$
$\quad [\,changesOnly\ ((\text{vat} \circ \text{par})\ `also`$
$\qquad\qquad\qquad (\text{vat} \circ \text{way}))$
$\qquad (perform\ \text{Prod})\,]$

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
$(\circ), (\$),$

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

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

33

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,
```

$\qquad\qquad$ HasJoy $\mathtt{ilk}_w$ Wad,
$\qquad\qquad$ HasChi $\mathtt{ilk}_w$ Ray)
$\quad \Rightarrow$ Id $\mathtt{Ilk} \to m$ Ray

`form` ::
$\quad$ (IsAct, Fails m,
$\quad$ Reads $r\ m$, HasSender r Address,
$\quad$ Writes $w\ m$, HasVat w $\mathtt{vat}_w$,
$\qquad\qquad\qquad$ HasIlks $\mathtt{vat}_w$ (Map (Id $\mathtt{Ilk}$) $\mathtt{Ilk}$))
$\quad \Rightarrow$ Id $\mathtt{Ilk} \to$ Id $\mathtt{Jar} \to m$ ()

`frob` :: (IsAct, Fails m,
$\qquad$ Reads $r\ m$, HasSender r Address,
$\qquad$ Writes $w\ m$, HasVat w $\mathtt{vat}_w$,
$\qquad\qquad\qquad$ HasHow $\mathtt{vat}_w$ $\mathtt{ray}$)
$\quad \Rightarrow \mathtt{ray} \to m$ ()

`open` ::
$\quad$ (IsAct,
$\quad$ Reads $r\ m$, HasSender r Address,
$\quad$ Writes $w\ m$, HasVat w $\mathtt{vat}_w$,
$\qquad\qquad$ HasUrns $\mathtt{vat}_w$ (Map (Id $\mathtt{Urn}$) $\mathtt{Urn}$))
$\quad \Rightarrow$ Id $\mathtt{Urn} \to$ Id $\mathtt{Ilk} \to m$ ()

`give` ::
$\quad$ (IsAct, Fails m,
$\quad$ Reads $r\ m$, HasSender r Address,
$\qquad\qquad$ HasVat r $\mathtt{vat}_r$,
$\qquad\qquad\qquad$ HasUrns $\mathtt{vat}_r$ (Map (Id $\mathtt{Urn}$) $\mathtt{urn}_r$),
$\qquad\qquad\qquad$ HasLad $\mathtt{urn}_r$ Address,
$\quad$ Writes $w\ m$, HasVat w $\mathtt{vat}_r$)
$\quad \Rightarrow$ Id $\mathtt{Urn} \to$ Address $\to m$ ()

`lock` ::
$\quad$ (IsAct, Fails m,
$\quad$ Reads $r\ m$,
$\qquad$ HasSender r Address,
$\qquad$ HasVat r $\mathtt{vat}_r$,

$$\text{HasUrns vat}_r \ (\text{Map (Id Urn) urn}_r),$$
$$\text{HasIlk urn}_r \ (\text{Id Ilk}),$$
$$\text{HasIlks vat}_r \ (\text{Map (Id Ilk) ilk}_r),$$
$$\text{HasJar ilk}_r \ (\text{Id Jar}),$$
$$\text{HasJars vat}_r \ (\text{Map (Id Jar) jar}_r),$$
$$\text{HasGem jar}_r \ \text{Gem},$$
$$\text{Writes } w \ m,$$
$$\text{HasVat } w \ \text{vat}_w,$$
$$\text{HasJars vat}_w \ (\text{Map (Id Jar) jar}_r),$$
$$\text{HasUrns vat}_w \ (\text{Map (Id Urn) urn}_w),$$
$$\text{HasJam urn}_w \ \text{Wad})$$
$$\Rightarrow \text{Id Urn} \rightarrow \text{Wad} \rightarrow m \ ()$$

`mark ::`
$$(\text{IsAct}, \text{Fails } m,$$
$$\text{Reads } r \ m, \ \ \text{HasSender } r \ \text{Address},$$
$$\text{Writes } w \ m, \text{HasVat } w \ \text{vat}_w,$$
$$\text{HasJars vat}_w \ (\text{Map (Id Jar) jar}_w),$$
$$\text{HasTag jar}_w \ \text{wad},$$
$$\text{HasZzz jar}_w \ \text{sec})$$
$$\Rightarrow \text{Id Jar} \rightarrow \text{wad} \rightarrow \text{sec} \rightarrow m \ ()$$

`tell ::`
$$(\text{IsAct}, \text{Fails } m,$$
$$\text{Reads } r \ m, \ \ \text{HasSender } r \ \text{Address},$$
$$\text{Writes } w \ m, \text{HasVat } w \ \text{vat}_w,$$
$$\text{HasFix vat}_w \ \text{wad})$$
$$\Rightarrow \text{wad} \rightarrow m \ ()$$

`prod ::`
$$(\text{IsAct},$$
$$\text{Reads } r \ m,$$
$$\text{HasSender } r \ \text{Address},$$
$$\text{HasEra } r \ \text{sec},$$
$$\text{HasVat } r \ \text{vat}_r, \ \ (\text{HasPar vat}_r \ \text{wad},$$
$$\text{HasTau vat}_r \ \text{sec},$$
$$\text{HasHow vat}_r \ \text{ray},$$
$$\text{HasWay vat}_r \ \text{ray},$$
$$\text{HasFix vat}_r \ \text{wad}),$$

Writes w m,
 HasVat w $\mathtt{vat}_w$, (HasPar $\mathtt{vat}_w$ $\mathtt{wad}$,
        HasWay $\mathtt{vat}_w$ $\mathtt{ray}$,
        HasTau $\mathtt{vat}_w$ $\mathtt{sec}$),
Integral $\mathtt{sec}$,
Ord $\mathtt{wad}$, Fractional $\mathtt{wad}$,
Fractional $\mathtt{ray}$, Real $\mathtt{ray}$)
$\Rightarrow m$ ()

$\mathtt{warp} ::$
 (IsAct, Fails m,
 Reads r m, HasSender r Address,
 Writes w m, HasEra w $\mathtt{sec}$,
      Num $\mathtt{sec}$)
 $\Rightarrow \mathtt{sec} \rightarrow m$ ()

$\mathtt{pull} ::$
 (Fails m,
 Reads r m,
  HasVat r $\mathtt{vat}_r$, HasJars $\mathtt{vat}_r$ (Map (Id Jar) $\mathtt{jar}_r$),
        HasGem $\mathtt{jar}_r$ Gem,
 Writes w m,
  HasVat w $\mathtt{vat}_w$, HasJars $\mathtt{vat}_w$ (Map (Id Jar) $\mathtt{jar}_r$))
 $\Rightarrow$ Id Jar $\rightarrow$ Address $\rightarrow$ Wad $\rightarrow m$ ()

$\mathtt{push} ::$
 (Fails m,
 Reads r m,
  HasVat r $\mathtt{vat}_r$, HasJars $\mathtt{vat}_r$ (Map (Id Jar) $\mathtt{jar}_r$),
        HasGem $\mathtt{jar}_r$ Gem,
 Writes w m,
  HasVat w $\mathtt{vat}_w$, HasJars $\mathtt{vat}_w$ (Map (Id Jar) $\mathtt{jar}_r$))
 $\Rightarrow$ Id Jar $\rightarrow$ Address $\rightarrow$ Wad $\rightarrow m$ ()

$\mathtt{mint} ::$
 (Fails m,
 Writes w m,
  HasVat w $\mathtt{vat}_w$, HasJars $\mathtt{vat}_w$ (Map (Id Jar) $\mathtt{jar}_r$),
        HasGem $\mathtt{jar}_r$ gem_r,

$$\text{HasTotalSupply } gem\_r \text{ Wad,}$$
$$\text{HasBalanceOf} \quad gem\_r \text{ (Map Address Wad))}$$
$$\Rightarrow \text{Id Jar} \rightarrow \text{Wad} \rightarrow m\ ()$$

```
burn ::
```
$$(\text{Fails } m,$$
$$\text{Writes } w\ m,$$
$$\text{HasVat } w \text{ vat}_w, \text{HasJars vat}_w \text{ (Map (Id Jar) jar}_r),$$
$$\text{HasGem jar}_r\ gem\_r,$$
$$\text{HasTotalSupply } gem\_r \text{ Wad,}$$
$$\text{HasBalanceOf} \quad gem\_r \text{ (Map Address Wad))}$$
$$\Rightarrow \text{Id Jar} \rightarrow \text{Wad} \rightarrow m\ ()$$

```
grab ::
```
$$(\text{IsAct}, \text{Fails } m,$$
$$\text{Numbers wad ray sec,}$$
$$\text{Reads } r\ m,$$
$$\text{HasSender } r \text{ Address,}$$
$$\text{HasEra } r \text{ Sec,}$$
$$\text{HasVat } r \text{ vat}_r,$$
$$\text{HasFix vat}_r \text{ wad,}$$
$$\text{HasPar vat}_r \text{ wad,}$$
$$\text{HasHow vat}_r \text{ ray,}$$
$$\text{HasWay vat}_r \text{ ray,}$$
$$\text{HasTau vat}_r \text{ sec,}$$
$$\text{HasUrns vat}_r \text{ (Map (Id Urn) urn}_r),$$
$$\text{HasJam urn}_r \text{ wad,}$$
$$\text{HasArt urn}_r \text{ wad,}$$
$$\text{HasCat urn}_r \text{ (Maybe Address)}, \text{HasVow urn}_r \text{ (Maybe Address)},$$
$$\text{HasIlk urn}_r \text{ (Id Ilk),}$$
$$\text{HasIlks vat}_r \text{ (Map (Id Ilk) ilk}_r),$$
$$\text{HasHat ilk}_r \text{ wad,}$$
$$\text{HasMat ilk}_r \text{ wad,}$$
$$\text{HasRum ilk}_r \text{ wad,}$$
$$\text{HasJoy ilk}_r \text{ wad,}$$
$$\text{HasTax ilk}_r \text{ ray,}$$
$$\text{HasLag ilk}_r \text{ sec,}$$
$$\text{HasChi ilk}_r \text{ ray,} \quad \text{HasRho ilk}_r \text{ sec,}$$
$$\text{HasJar ilk}_r \text{ (Id Jar),}$$
$$\text{HasJars vat}_r \text{ (Map (Id Jar) jar}_r),$$

$\quad$ HasGem $\mathtt{jar}_r$ Gem,

$\quad$ HasTag $\mathtt{jar}_r$ wad,

$\quad$ HasZzz $\mathtt{jar}_r$ sec,

Writes $w\ m$,

$\quad$ HasVat $w\ \mathtt{vat}_w$,

$\quad$ HasTau $\mathtt{vat}_w$ sec,

$\quad$ HasWay $\mathtt{vat}_w$ ray, HasPar $\mathtt{vat}_w$ wad,

$\quad$ HasUrns $\mathtt{vat}_w$ (Map (Id Urn) $\mathtt{urn}_w$),

$\quad\quad$ HasJam $\mathtt{urn}_w$ wad, HasArt $\mathtt{urn}_w$ wad,

$\quad\quad$ HasVow $\mathtt{urn}_w$ Address,

$\quad\quad$ HasCat $\mathtt{urn}_w$ (Maybe Address),

$\quad$ HasIlks $\mathtt{vat}_w$ (Map (Id Ilk) $\mathtt{ilk}_w$),

$\quad\quad$ HasJoy $\mathtt{ilk}_w$ wad,

$\quad\quad$ HasChi $\mathtt{ilk}_w$ ray,

$\quad\quad$ HasRho $\mathtt{ilk}_w$ sec,

$\quad$ HasJars $\mathtt{vat}_w$ (Map (Id Jar) $\mathtt{jar}_r$)

$) \Rightarrow$ Id Urn $\rightarrow m$ ()