MAKER

presents the

REFERENCE IMPLEMENTATION

of the remarkable

DAI CREDIT SYSTEM

issuing a diversely collateralized stablecoin

with last update on February 28, 2017.

Contents

Chapter 1

Introduction

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

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

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

Maker also controls the supply of another token, MKR. This token acts as a "share" in Maker. 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. On the other hand, Maker imposes a *stability fee* on all dai loans, and revenue from these fees 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 include:

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

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

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

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

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

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

1.2 Prerequisite Haskell knowledge

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

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

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

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

Part I

Implementation

Chapter 2

Preamble

```
module Maker where
```

We import types for the decimal fixed-point arithmetic which we use for amounts and rates.

```
import Data.Fixed
```

We rely on the `lens` library for defining and using accessors which otherwise tend to become long-winded in Haskell. Since our program has several nested records, this makes the code much clearer. There is no need to understand the theory behind lenses to understand this program. All the reader needs to know is that $a \circ b \circ c$ denotes a nested accessor much like `a.b.c` in C-style languages. The rest should be obvious from context.

```
import Control.Lens
```

We use a typical stack of monad transformers from the *mtl* library to structure state-modifying actions. Again, the reader does not need any abstract understanding of monads. They make our code clear and simple by enabling **do** blocks to express exceptions, state, and logging.

```
import Control.Monad.Except
  (MonadError, Except, throwError, runExcept)
import Control.Monad.Reader
  (MonadReader (..))
import Control.Monad.State
  (MonadState, StateT, execStateT, get, put)
import Control.Monad.Writer
  (MonadWriter, WriterT, runWriterT)
```

Some less interesting imports are omitted from this document.

Chapter 3

Types

3.1 Numeric types

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

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

Phantom types encode precision at compile time.

```
data E18; data E36
```

Specify 10^{-18} as the precision of E18.

```
instance HasResolution E18 where
  resolution _ = 10 ↑ (18 :: Integer)
```

Specify 10^{-36} as the precision of E36.

```
instance HasResolution E36 where
  resolution _ = 10 ↑ (36 :: Integer)
```

Create the distinct WAD type for currency quantities.

```
newtype Wad = Wad (Fixed E18)
  deriving (Ord, Eq, Num, Real, Fractional)
```

Create the distinct RAY type for precise rate quantities.

```
newtype Ray = Ray (Fixed E36)
  deriving (Ord, Eq, Num, Real, Fractional)
```

In calculations where a WAD is multiplied by a RAY, for example in the deflation mechanism, we have to downcast in a way that loses precision. Haskell does not cast automatically, so unless you see the following *cast* function applied, you can assume that precision is unchanged.

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

We also define a type for non-negative integers.

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

3.1.1 Epsilon values

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 (Fixed a) where
    The use of ⊥ is safe since resolution ignores the value.
  ϵ = 1 / fromIntegral (resolution (⊥ :: Fixed a))
instance Epsilon Wad where ϵ = Wad ϵ
instance Epsilon Ray  where ϵ = Ray ϵ
```

3.2 Identifier type

There are several types of identifiers used in the system, and we can use Haskell's type system to distinguish them.

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

It turns out that we will in several places use mappings from IDs to the value type corresponding to that ID type, so we define an alias for such mapping types.

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

We also have three predefined entities:

The DAI token address

id_{DAI} = Id "Dai"

The CDP engine address

id_{VAT} = Id "Vat"

The account with ultimate authority

id_{god} = Id "God"

3.3 Structures

[XXX: describe structures]

```
data Lad = Lad deriving (Eq, Show)
```

3.3.1 Gem — Collateral token model

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

3.3.2 Jar — Collateral token

```
data Jar = Jar {
    Collateral token
```

```
      jarGem :: !Gem,
    Market price
      jarTag  :: !Wad,
    Price expiration
      jarZzz  :: !Nat
  } deriving (Eq, Show, Read)
makeFields ' ' Jar
```

3.3.3 Ilk — cdp type

```
data Ilk = Ilk {
    Collateral vault
      ilkJar  :: !(Id Jar),
    Liquidation penalty
      ilkAxe  :: !Ray,
    Debt ceiling
      ilkHat  :: !Wad,
    Liquidation ratio
      ilkMat  :: !Ray,
    Stability fee
      ilkTax  :: !Ray,
    Limbo duration
      ilkLag  :: !Nat,
    Last dripped
      ilkRho  :: !Nat,
    ???
      ilkCow :: !Ray,
    Stability fee accumulator
      ilkBag  :: !(Map Nat Ray)
  } deriving (Eq, Show)
makeFields ' ' Ilk
```

3.3.4 Urn — collateralized debt position (cdp)

```
data Urn = Urn {
    Address of biting cat
      urnCat  :: !(Maybe (Id Lad)),
    Address of liquidating vow
      urnVow :: !(Maybe (Id Lad)),
    Issuer
      urnLad  :: !(Id Lad),
    cdp type
      urnIlk   :: !(Id Ilk),
    Outstanding dai debt
      urnCon :: !Wad,
    Collateral amount
      urnPro  :: !Wad,
    Last poked
      urnPhi  :: !Nat
  } deriving (Eq, Show)
makeFields ''Urn
```

3.3.5 Vat — Dai creditor

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

3.3.6 System model

```
data System =
  System {
    systemVat    :: Vat,
    systemEra    :: !Nat,
    systemLads   :: IdMap Lad,  System users
    systemSender :: Id Lad
  } deriving (Eq, Show)
makeFields ''System
```

3.3.7 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  = Nat 0,
  ilkBag  = ∅,
  ilkCow  = Ray 1,
  ilkRho  = Nat 0
}

defaultUrn :: Id Ilk → Id Lad → Urn
defaultUrn id_ILK id_LAD = Urn {
  urnVow = Nothing,
  urnCat = Nothing,
  urnLad = id_LAD,
  urnIlk = id_ILK,
  urnCon = Wad 0,
  urnPro = Wad 0,
  urnPhi = Nat 0
}

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

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

Chapter 4

Act framework

4.1 Act descriptions

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

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

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

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

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

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

Now we can define the type of a

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

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

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

4.2 Constraints

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

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

4.3 Accessor aliases

```
ilkAt  id = VAT ∘ ILKs  ∘ ix id
urnAt id = VAT ∘ URNs ∘ ix id
jarAt  id = VAT ∘ JARs  ∘ ix id
```

4.4 Logging and asserting

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

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

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

4.5 Modifiers

```
note ::
  (IsAct, Logs m,
   Reads r m,
      HasSender r (Id Lad))
   ⇒ m a → m a
```

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

```
auth ::
   (IsAct, Fails m,
    Reads r m,
      HasSender r (Id Lad))
   ⇒ m a → m a
```

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

Chapter 5

Acts

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

Table 5.1: Urn acts in the five stages of risk

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

5.1 Risk assessment

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

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

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

```
analyze ERA0 PAR0 URN0 ILK0 JAR0 =
  let
    Market value of collateral
     PRO_SDR = view PRO URN0 * view TAG JAR0
    Debt at DAI target price
     CON_SDR = view CON URN0 * PAR0
  in if
    Undergoing liquidation?
     | view VOW URN0 ≢ Nothing                 → Dread
    Liquidation triggered?
     | view CAT  URN0 ≢ Nothing                → Grief
    Undercollateralized?
     | PRO_SDR < CON_SDR * view MAT ILK0       → Panic
    Price feed expired?
     | ERA0 > view ZZZ JAR0 + view LAG ILK0    → Panic
    Price feed in limbo?
     | view ZZZ  JAR0 < ERA0                   → Worry
    Debt ceiling reached?
     | view COW ILK0  > view HAT ILK0          → Anger
    Safely overcollateralized.
     | otherwise                               → Pride
```

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

```
gaze $id_{\text{URN}}$ = $\mathbf{do}$
  prod
  poke $id_{\text{URN}}$
  $\text{ERA}_0 \leftarrow \textit{view}\ \text{ERA}$
  $\text{PAR}_0 \leftarrow \textit{view}\ (\text{VAT} \circ \text{PAR})$
  $\text{URN}_0 \leftarrow \textit{need}\ (\textit{urnAt}\ id_{\text{URN}})$
  $\text{ILK}_0\ \ \leftarrow \textit{need}\ (\textit{ilkAt}\ \ (\textit{view}\ \text{ILK}\ \text{URN}_0))$
  $\text{JAR}_0 \leftarrow \textit{need}\ (\textit{jarAt}\ \ (\textit{view}\ \text{JAR}\ \text{ILK}_0\ ))$
  $\textit{return}\ (\textit{analyze}\ \text{ERA}_0\ \text{PAR}_0\ \text{URN}_0\ \text{ILK}_0\ \text{JAR}_0)$
```

5.2 Lending

```
open $id_{\text{URN}}$ $id_{\text{ILK}}$ =
  note \$ $\mathbf{do}$
    $id_{\text{LAD}} \leftarrow \textit{view sender}$
    $\text{VAT} \circ \text{URN}s \circ \textit{at}\ id_{\text{URN}}$ ?= $\textit{defaultUrn}\ id_{\text{ILK}}\ id_{\text{LAD}}$
```

```
lock $id_{\text{URN}}$ $x$ =
  note \$ $\mathbf{do}$
   Ensure CDP exists; identify collateral type
    $id_{\text{ILK}}\ \leftarrow \textit{need}\ (\textit{urnAt}\ id_{\text{URN}} \circ \text{ILK})$
    $id_{\text{JAR}} \leftarrow \textit{need}\ (\textit{ilkAt}\ \ id_{\text{ILK}}\ \circ \text{JAR})$
   Record an increase in collateral
    $\textit{urnAt}\ id_{\text{URN}} \circ \text{PRO}$ += $x$
   Take sender's tokens
    $id_{\text{LAD}} \leftarrow \textit{view sender}$
    pull $id_{\text{JAR}}$ $id_{\text{LAD}}$ $x$
```

```
free $id_{\text{URN}}$ $\text{WAD}_{\text{GEM}}$ =
  note \$ $\mathbf{do}$
```

```
    Fail if sender is not the CDP owner.
     $id_{sender} \leftarrow \mathit{view\ sender}$
     $id_{\mathrm{LAD}} \quad \leftarrow \mathit{need}\ (\mathit{urnAt}\ id_{\mathrm{URN}} \circ \mathrm{LAD})$
     $\mathit{sure}\ (id_{sender} \equiv id_{\mathrm{LAD}})$
    Tentatively record the decreased collateral.
     $\mathit{urnAt}\ id_{\mathrm{URN}} \circ \mathrm{PRO}\ {-}{=}\ \mathrm{WAD}_{\mathrm{GEM}}$
    Fail if collateral decrease results in undercollateralization.
     $\mathtt{gaze}\ id_{\mathrm{URN}} \ggg\!= \mathit{sure} \circ (\equiv \mathtt{Pride})$
    Send the collateral to the CDP owner.
     $id_{\mathrm{ILK}} \leftarrow \mathit{need}\ (\mathit{urnAt} \quad id_{\mathrm{URN}} \circ \mathrm{ILK})$
     $id_{\mathrm{JAR}} \leftarrow \mathit{need}\ (\mathit{ilkAt} \quad id_{\mathrm{ILK}} \circ \mathrm{JAR})$
     $\mathtt{push}\ id_{\mathrm{JAR}}\ id_{\mathrm{LAD}}\ \mathrm{WAD}_{\mathrm{GEM}}$
```

```
$\mathtt{draw}\ id_{\mathrm{URN}}\ \mathrm{WAD}_{\mathrm{DAI}} =$
  $\mathtt{note}\ \$\ \mathbf{do}$
    Fail if sender is not the CDP owner.
     $id_{sender} \leftarrow \mathit{view\ sender}$
     $id_{\mathrm{LAD}} \quad \leftarrow \mathit{need}\ (\mathit{urnAt}\ id_{\mathrm{URN}} \circ \mathrm{LAD})$
     $\mathit{sure}\ (id_{sender} \equiv id_{\mathrm{LAD}})$
    Tentatively record DAI debt.
     $\mathit{urnAt}\ id_{\mathrm{URN}} \circ \mathrm{CON}\ {+}{=}\ \mathrm{WAD}_{\mathrm{DAI}}$
    Fail if CDP with new debt is not overcollateralized.
     $\mathtt{gaze}\ id_{\mathrm{URN}} \ggg\!= \mathit{sure} \circ (\equiv \mathtt{Pride})$
    Mint DAI and send it to the CDP owner.
     $\mathtt{mint}\ id_{\mathrm{DAI}}\ \mathrm{WAD}_{\mathrm{DAI}}$
     $\mathtt{push}\ id_{\mathrm{DAI}}\ id_{\mathrm{LAD}}\ \mathrm{WAD}_{\mathrm{DAI}}$
```

```
$\mathtt{wipe}\ id_{\mathrm{URN}}\ \mathrm{WAD}_{\mathrm{DAI}} =$
  $\mathtt{note}\ \$\ \mathbf{do}$
    Fail if sender is not the CDP owner.
     $id_{sender} \leftarrow \mathit{view\ sender}$
     $id_{\mathrm{LAD}} \quad \leftarrow \mathit{need}\ (\mathit{urnAt}\ id_{\mathrm{URN}} \circ \mathrm{LAD})$
     $\mathit{sure}\ (id_{sender} \equiv id_{\mathrm{LAD}})$
    Fail if the CDP is not currently overcollateralized.
     $\mathtt{gaze}\ id_{\mathrm{URN}} \ggg\!= \mathit{sure} \circ (\equiv \mathtt{Pride})$
```

```
      Preliminarily reduce the CDP debt.
       $urnAt\ id_{\mathrm{URN}} \circ \mathrm{CON}$ −= $\mathrm{WAD}_{\mathrm{DAI}}$
      Attempt to get back DAI from CDP owner and destroy it.
       pull $id_{\mathrm{DAI}}\ id_{\mathrm{LAD}}\ \mathrm{WAD}_{\mathrm{DAI}}$
       burn $id_{\mathrm{DAI}}\ \mathrm{WAD}_{\mathrm{DAI}}$
```

```
give $id_{\mathrm{URN}}\ id_{\mathrm{LAD}}$ =
  note $ do
     $x \leftarrow need\ (urnAt\ id_{\mathrm{URN}} \circ \mathrm{LAD})$
     $y \leftarrow view\ sender$
     $sure\ (x \equiv y)$
     $urnAt\ id_{\mathrm{URN}} \circ \mathrm{LAD} := id_{\mathrm{LAD}}$
```

```
shut $id_{\mathrm{URN}}$ =
  note $ do
    Update the CDP's debt (prorating the stability fee).
     poke $id_{\mathrm{URN}}$
    Attempt to repay all the CDP's outstanding DAI.
     $\mathrm{CON}_0 \leftarrow need\ (urnAt\ id_{\mathrm{URN}} \circ \mathrm{CON})$
     wipe $id_{\mathrm{URN}}\ \mathrm{CON}_0$
    Reclaim all the collateral.
     $\mathrm{PRO}_0 \leftarrow need\ (urnAt\ id_{\mathrm{URN}} \circ \mathrm{PRO})$
     free $id_{\mathrm{URN}}\ \mathrm{PRO}_0$
    Nullify the CDP.
     $\mathrm{VAT} \circ \mathrm{URN}s \circ at\ id_{\mathrm{URN}} := \mathrm{Nothing}$
```

5.3 Frequent adjustments

```
prod = note $ do
  ERA0 ← view ERA
  TAU0 ← view (VAT ∘ TAU)
  FIX0 ← view (VAT ∘ FIX)
  PAR0 ← view (VAT ∘ PAR)
  HOW0 ← view (VAT ∘ HOW)
  WAY0 ← view (VAT ∘ WAY)
  let
    Time difference in seconds
    fan  = ERA0 − TAU0
    Current deflation rate applied to target price
    PAR1 = PAR0 ∗ cast (WAY0 ↑↑ fan)
    Sensitivity parameter applied over time
    wag = HOW0 ∗ fromIntegral fan
    Deflation rate scaled up or down
    WAY1 = inj (prj WAY0 +
                 if FIX0 < PAR0 then wag else − wag)
  VAT ∘ PAR  := PAR1
  VAT ∘ WAY  := WAY1
  VAT ∘ TAU  := ERA0
  where
    Convert between multiplicative and additive form
    prj x    = if x ⩾ 1 then x − 1 else 1 − 1 / x
    inj x    = if x ⩾ 0 then x + 1 else 1 / (1 − x)
```

This internal act happens on every `poke`. It is also invoked when governance changes the TAX of an ILK.

```
drip id_ILK = do
  Current time stamp
  ERA0 ← view ERA
  Current stability fee
  TAX0 ← need (ilkAt id_ILK ∘ TAX)
  COW0 ← need (ilkAt id_ILK ∘ COW)
  Previous time and stability fee thus far
```

```
    $\mathrm{RHO}_0 \leftarrow need\ (ilkAt\ id_{\mathrm{ILK}} \circ \mathrm{RHO})$
    $ice \quad\ \leftarrow need\ (ilkAt\ id_{\mathrm{ILK}} \circ \mathrm{BAG} \circ ix\ \mathrm{RHO}_0)$
    let
      Seconds passed
        $age \quad = \mathrm{ERA}_0 - \mathrm{RHO}_0$
      Stability fee accrued since last drip
        $dew \quad = ice * \mathrm{TAX}_0 \uparrow\uparrow age$
      I don't understand this calculation
        $\mathrm{COW}_1 = \mathrm{COW}_0 * (dew\ /\ ice)$
    $ilkAt\ id_{\mathrm{ILK}} \circ \mathrm{BAG} \circ at\ \mathrm{ERA}_0 \mathrel{?}= dew$
    $ilkAt\ id_{\mathrm{ILK}} \circ \mathrm{COW} \qquad\qquad := \mathrm{COW}_1$
    $ilkAt\ id_{\mathrm{ILK}} \circ \mathrm{RHO} \qquad\qquad := \mathrm{ERA}_0$
    $return\ dew$
```

```
  poke $id_{\mathrm{URN}}$ =
    note $ do
      Read previous stability fee accumulator.
        $id_{\mathrm{ILK}} \leftarrow need\ (urnAt\ id_{\mathrm{URN}} \circ \mathrm{ILK})$
        $phi0 \leftarrow need\ (urnAt\ id_{\mathrm{URN}} \circ \mathrm{PHI})$
        $ice \quad \leftarrow need\ (ilkAt\ id_{\mathrm{ILK}} \quad \circ \mathrm{BAG} \circ ix\ phi0)$
      Update the stability fee accumulator.
        $\mathrm{CON}_0 \leftarrow need\ (urnAt\ id_{\mathrm{URN}} \circ \mathrm{CON})$
        $dew \quad \leftarrow$ drip $id_{\mathrm{ILK}}$
      Apply new stability fee to CDP debt.
        $urnAt\ id_{\mathrm{URN}} \circ \mathrm{CON} *= cast\ (dew\ /\ ice)$
      Record the poke time.
        $\mathrm{ERA}_0 \leftarrow view\ \mathrm{ERA}$
        $urnAt\ id_{\mathrm{URN}} \circ \mathrm{PHI} := \mathrm{ERA}_0$
```

5.4 Governance

```
  form $id_{\mathrm{ILK}}\ id_{\mathrm{JAR}}$ =
    auth ∘ note $ do
      $\mathrm{VAT} \circ \mathrm{ILK}s \circ at\ id_{\mathrm{ILK}} \mathrel{?}= defaultIlk\ id_{\mathrm{JAR}}$
```

```
frob how′ =
  auth ∘ note $ do
    VAT ∘ HOW := how′
```

5.5 Price feedback

```
mark id_JAR TAG_1 ZZZ_1 =
  auth ∘ note $ do
    jarAt id_JAR ∘ TAG := TAG_1
    jarAt id_JAR ∘ ZZZ  := ZZZ_1
```

```
tell x =
  auth ∘ note $ do
    VAT ∘ FIX := x
```

5.6 Liquidation and settlement

```
bite id_URN =
  note $ do
   Fail if urn is not undercollateralized.
    gaze id_URN ≫= sure ∘ (≡ Panic)
   Record the sender as the liquidation initiator.
    id_CAT                 ← view sender
    urnAt id_URN ∘ CAT := id_CAT
   Read current debt.
    CON_0 ← need (urnAt id_URN ∘ CON)
   Read liquidation penalty ratio.
    id_ILK  ← need (urnAt id_URN ∘ ILK)
    AXE_0 ← need (ilkAt id_ILK   ∘ AXE)
   Apply liquidation penalty to debt.
    let CON_1 = CON_0 ∗ AXE_0
    Update debt and record it as in need of settlement.
      $\mathit{urnAt}\ id_{\text{URN}} \circ \text{CON} := \text{CON}_1$
      $\text{SIN} \qquad\qquad\quad += \text{CON}_1$
```

```
grab $id_{\text{URN}}$ =
  auth ∘ note $ do
    Fail if CDP liquidation is not initiated.
      gaze $id_{\text{URN}} \ggg\!= \mathit{sure} \circ (\equiv$ Grief$)$
    Record the sender as the CDP's settler.
      $id_{\text{VOW}} \leftarrow \mathit{view\ sender}$
      $\mathit{urnAt}\ id_{\text{URN}} \circ \text{VOW} := id_{\text{VOW}}$
    Nullify the CDP's debt and collateral.
      $\text{PRO}_0 \leftarrow \mathit{need}\ (\mathit{urnAt}\ id_{\text{URN}} \circ \text{PRO})$
      $\mathit{urnAt}\ id_{\text{URN}} \circ \text{CON} := 0$
      $\mathit{urnAt}\ id_{\text{URN}} \circ \text{PRO} := 0$
    Send the collateral to the settler for auctioning.
      $id_{\text{ILK}} \leftarrow \mathit{need}\ (\mathit{urnAt}\ id_{\text{URN}} \circ \text{ILK})$
      $id_{\text{JAR}} \leftarrow \mathit{need}\ (\mathit{ilkAt}\ id_{\text{ILK}} \circ \text{JAR})$
      push $id_{\text{JAR}}\ id_{\text{VOW}}\ \text{PRO}_0$
```

```
heal $\text{WAD}_{\text{DAI}}$ =
  auth ∘ note $ do
    $\text{VAT} \circ \text{SIN} -= \text{WAD}_{\text{DAI}}$
```

```
loot $\text{WAD}_{\text{DAI}}$ =
  auth ∘ note $ do
    $\text{VAT} \circ \text{PIE} -= \text{WAD}_{\text{DAI}}$
```

5.7 Minting, burning, and transferring

```
pull $id_{\text{JAR}}\ id_{\text{LAD}}\ w$ = do
  $g \leftarrow \mathit{need}\ (\mathit{jarAt}\ id_{\text{JAR}} \circ \text{GEM})$
  $g' \leftarrow \mathit{transferFrom}\ \mathit{id}_{\text{LAD}}\ \mathit{id}_{\text{VAT}}\ w\ g$
  $\mathit{jarAt}\ \mathit{id}_{\text{JAR}} \circ \text{GEM} := g'$
```

```
push $\mathit{id}_{\text{JAR}}\ \mathit{id}_{\text{LAD}}\ w =$ do
  $g\ \leftarrow \mathit{need}\ (\mathit{jarAt}\ \mathit{id}_{\text{JAR}} \circ \text{GEM})$
  $g' \leftarrow \mathit{transferFrom}\ \mathit{id}_{\text{VAT}}\ \mathit{id}_{\text{LAD}}\ w\ g$
  $\mathit{jarAt}\ \mathit{id}_{\text{JAR}} \circ \text{GEM} := g'$
```

```
mint $\mathit{id}_{\text{JAR}}\ \text{WAD}_0 =$ do
  $\mathit{jarAt}\ \mathit{id}_{\text{JAR}} \circ \text{GEM} \circ \mathit{totalSupply}$              $\mathrel{+}= \text{WAD}_0$
  $\mathit{jarAt}\ \mathit{id}_{\text{JAR}} \circ \text{GEM} \circ \mathit{balanceOf} \circ \mathit{ix}\ \mathit{id}_{\text{VAT}}$ $\mathrel{+}= \text{WAD}_0$
```

```
burn $\mathit{id}_{\text{JAR}}\ \text{WAD}_0 =$ do
  $\mathit{jarAt}\ \mathit{id}_{\text{JAR}} \circ \text{GEM} \circ \mathit{totalSupply}$              $\mathrel{-}= \text{WAD}_0$
  $\mathit{jarAt}\ \mathit{id}_{\text{JAR}} \circ \text{GEM} \circ \mathit{balanceOf} \circ \mathit{ix}\ \mathit{id}_{\text{VAT}}$ $\mathrel{-}= \text{WAD}_0$
```

5.8 Test-related manipulation

```
warp $t$ =
  auth ∘ note $ do
    $\text{ERA} \mathrel{+}= t$
```

5.9 System modelling

```
$\mathit{newLad}\ \mathit{id}_{\text{LAD}} = \mathit{lads} \circ \mathit{at}\ \mathit{id}_{\text{LAD}}\ ?= \text{LAD}$
```

```
$\mathit{newLad} ::$
  $(\text{Writes}\ w\ m, \text{HasLads}\ w\ (\text{IdMap LAD}))$
  $\Rightarrow \text{Id LAD} \rightarrow m\ ()$
```

```
newJar id id_JAR =
  auth ∘ note $ do
    VAT ∘ JARs ∘ at id ?= id_JAR
```

```
newJar ::
    ( IsAct, Fails m, Logs m,
      Reads r m,  HasSender r (Id LAD),
      Writes w m, HasVat w VAT_w,
                         HasJars VAT_w (IdMap JAR))
   ⇒
    Id JAR → JAR → m ()
```

5.10 Other stuff

```
perform :: Act → Maker ()
perform x =
  let ?act = x in case x of
    NewLad id        → newLad id
    NewJar id JAR → newJar id JAR
    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)
  ⇒  Id LAD → Id LAD → WAD
  →  GEM → m GEM
transferFrom src dst WAD GEM =
  case GEM ^. balanceOf ∘ (at src) of
    Nothing →
      throwError AssertError
```

```
Just balance → do
  sure (balance ⩾ WAD)
  return $ GEM
    & balanceOf ∘ ix src −~ WAD
    & balanceOf ∘ at dst %~
      (λcase
        Nothing → Just WAD
        Just x  → Just (WAD + x))
```

Chapter 6

Testing

Appendix A

Act type signatures

We see that `drip` may fail; it reads an ILK's TAX, COW, RHO, and BAG; and it writes those same parameters except TAX.

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

```
form ::
  (IsAct, Fails $m$, Logs $m$,
   Reads $r$ $m$,  HasSender $r$ (Id Lad),
   Writes $w$ $m$, HasVat $w$ $\textsc{vat}_w$,
                   HasIlks $\textsc{vat}_w$ (IdMap Ilk))
```

$$
\begin{array}{l}
\quad \Rightarrow \text{Id } \textsc{Ilk} \rightarrow \text{Id } \textsc{Jar} \rightarrow m\ ()
\end{array}
$$

$$
\begin{array}{l}
\texttt{frob} :: (\text{IsAct}, \text{Fails } m, \text{Logs } m, \\
\qquad\qquad \text{Reads } r\ m,\ \ \text{HasSender } r\ (\text{Id } \textsc{Lad}), \\
\qquad\qquad \text{Writes } w\ m, \text{HasVat } w\ \textsc{vat}_w, \\
\qquad\qquad\qquad\qquad\qquad\quad \text{HasHow } \textsc{vat}_w\ \textsc{ray}) \\
\quad \Rightarrow \textsc{ray} \rightarrow m\ ()
\end{array}
$$

$$
\begin{array}{l}
\texttt{open} :: \\
\quad (\text{IsAct}, \text{Logs } m, \\
\quad\ \ \text{Reads } r\ m,\ \ \text{HasSender } r\ (\text{Id } \textsc{Lad}), \\
\quad\ \ \text{Writes } w\ m, \text{HasVat } w\ \textsc{vat}_w, \\
\qquad\qquad\qquad\quad \text{HasUrns } \textsc{vat}_w\ (\text{IdMap } \textsc{Urn})) \\
\quad \Rightarrow \text{Id } \textsc{Urn} \rightarrow \text{Id } \textsc{Ilk} \rightarrow m\ ()
\end{array}
$$

$$
\begin{array}{l}
\texttt{give} :: \\
\quad (\text{IsAct}, \text{Fails } m, \text{Logs } m, \\
\quad\ \ \text{Reads } r\ m,\ \ \text{HasSender } r\ (\text{Id } \textsc{Lad}), \\
\qquad\qquad\qquad \text{HasVat } r\ \textsc{vat}_r, \\
\qquad\qquad\qquad\quad \text{HasUrns } \textsc{vat}_r\ (\text{Map } (\text{Id } \textsc{Urn})\ \textsc{urn}_r), \\
\qquad\qquad\qquad\qquad \text{HasLad } \textsc{urn}_r\ (\text{Id } \textsc{Lad}), \\
\quad\ \ \text{Writes } w\ m, \text{HasVat } w\ \textsc{vat}_r) \\
\quad \Rightarrow \text{Id } \textsc{Urn} \rightarrow \text{Id } \textsc{Lad} \rightarrow m\ ()
\end{array}
$$

$$
\begin{array}{l}
\texttt{lock} :: \\
\quad (\text{IsAct}, \text{Fails } m, \text{Logs } m, \\
\quad\ \ \text{Reads } r\ m, \\
\qquad \text{HasSender } r\ (\text{Id } \textsc{Lad}), \\
\qquad \text{HasVat } r\ \textsc{vat}_r, \\
\qquad\quad \text{HasUrns } \textsc{vat}_r\ (\text{Map } (\text{Id } \textsc{Urn})\ \textsc{urn}_r), \\
\qquad\qquad \text{HasIlk } \textsc{urn}_r\ (\text{Id } \textsc{Ilk}), \\
\qquad\quad \text{HasIlks } \textsc{vat}_r\ (\text{Map } (\text{Id } \textsc{Ilk})\ \textsc{ilk}_r), \\
\qquad\qquad \text{HasJar } \textsc{ilk}_r\ (\text{Id } \textsc{Jar}), \\
\qquad\quad \text{HasJars } \textsc{vat}_r\ (\text{Map } (\text{Id } \textsc{Jar})\ \textsc{jar}_r), \\
\qquad\qquad \text{HasGem } \textsc{jar}_r\ \textsc{Gem}, \\
\quad\ \ \text{Writes } w\ m, \\
\qquad \text{HasVat } w\ \textsc{vat}_w, \\
\qquad\quad \text{HasJars } \textsc{vat}_w\ (\text{Map } (\text{Id } \textsc{Jar})\ \textsc{jar}_r), \\
\qquad\quad \text{HasUrns } \textsc{vat}_w\ (\text{Map } (\text{Id } \textsc{Urn})\ \textsc{urn}_w),
\end{array}
$$

```
            HasPro URN_w WAD)
  ⇒ Id URN → WAD → m ()

mark ::
  (IsAct, Fails m, Logs m,
   Reads r m,  HasSender r (Id LAD),
   Writes w m, HasVat w VAT_w,
                  HasJars VAT_w (Map (Id JAR) JAR_w),
                     HasTag JAR_w WAD,
                     HasZzz JAR_w NAT)
  ⇒ Id JAR → WAD → NAT → m ()

tell ::
  (IsAct, Fails m, Logs m,
   Reads r m,  HasSender r (Id LAD),
   Writes w m, HasVat w VAT_w,
                  HasFix VAT_w WAD)
  ⇒ WAD → m ()

prod ::
  (IsAct, Logs m,
   Reads r m,
     HasSender r (Id LAD),
     HasEra r NAT,
     HasVat r VAT_r,  (HasPar VAT_r WAD,
                       HasTau VAT_r NAT,
                       HasHow VAT_r RAY,
                       HasWay VAT_r RAY,
                       HasFix VAT_r WAD),
   Writes w m,
     HasVat w VAT_w, (HasPar VAT_w WAD,
                       HasWay VAT_w RAY,
                       HasTau VAT_w NAT),
   Integral NAT,
   Ord WAD, Fractional WAD,
   Fractional RAY, Real RAY)
  ⇒ m ()

warp ::
  (IsAct, Fails m, Logs m,
```

    Reads r m, HasSender r (Id LAD),
    Writes w m, HasEra w NAT,
                    Num NAT)
    ⇒ NAT → m ()

`pull` ::
  (Fails m,
   Reads r m,
     HasVat r VAT$_r$, HasJars VAT$_r$ (Map (Id JAR) JAR$_r$),
                          HasGem JAR$_r$ GEM,
   Writes w m,
     HasVat w VAT$_w$, HasJars VAT$_w$ (Map (Id JAR) JAR$_r$))
   ⇒ Id JAR → Id LAD → 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 → Id LAD → 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 (Id LAD) 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 (Id LAD) WAD))
   ⇒ Id JAR → WAD → m ()