MAKER

presents the

REFERENCE IMPLEMENTATION

of the remarkable

DAI CREDIT SYSTEM

issuing a diversely collateralized stablecoin

with last update on March 10, 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. **Typing.** 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 Limitations

This implementation has a simplified model of Maker's governance authorization. Instead of the "access control list" approach of the `DSGuard` component, we give full authority to one single address. A future iteration will include the full authorization model.

We also do not currently model the EVM's 256 bit word size, but allow all quantities to grow arbitrarily large. This will also be modelled in a future iteration.

Finally, our model of ERC20 tokens is simplified, and for example does not include the concept of "allowances."

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 Prelude ()          Import nothing from Prelude
import Maker.Prelude       Import everything from Maker Prelude
```

We also import our definition of decimal fixed point numbers, listed in Appendix B.

```
import Maker.Decimal
```

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). See Appendix B for details on decimal fixed point numbers and rounding.

Define the distinct type for currency quantities

$\textbf{newtype}\ \texttt{Wad} = \texttt{Wad}\ (\text{Decimal E18})$
$\quad\textbf{deriving}\ (\text{Ord}, \text{Eq}, \text{Num}, \text{Real}, \text{Fractional}, \text{RealFrac})$

Define the distinct type for rates and ratios

$\textbf{newtype}\ \texttt{Ray} = \texttt{Ray}\ (\text{Decimal E36})$
$\quad\textbf{deriving}\ (\text{Ord}, \text{Eq}, \text{Num}, \text{Real}, \text{Fractional}, \text{RealFrac})$

We also define a type for time durations in whole seconds.

$\textbf{newtype}\ \texttt{Sec} = \texttt{Sec}\ \text{Int}$
$\quad\textbf{deriving}\ (\text{Eq}, \text{Ord}, \text{Enum}, \text{Num}, \text{Real}, \text{Integral})$

Haskell number types are not automatically converted, so we convert explicitly with a *cast* function.

Convert via fractional n/m form.

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

3.2 Identifiers and addresses

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

The type parameter a creates distinct types.
For example, Id Foo and Id Bar are incompatible.

$$\textbf{data}\ \mathrm{Id}\ a = \mathrm{Id}\ \mathrm{String}$$
$$\quad \textbf{deriving}\ (\mathrm{Show}, \mathrm{Eq}, \mathrm{Ord})$$

We define another type for representing Ethereum account addresses.

$$\textbf{data}\ \mathrm{Address} = \mathrm{Address}\ \mathrm{String}$$
$$\quad \textbf{deriving}\ (\mathrm{Ord}, \mathrm{Eq}, \mathrm{Show})$$

We also have three predefined entity identifiers.

The DAI token address
$$id_{\mathrm{DAI}} = \mathrm{Id}\ \texttt{"DAI"}$$
The CDP engine address
$$id_{\texttt{vat}} = \mathrm{Address}\ \texttt{"VAT"}$$
A test account with ultimate authority
$$id_{god} = \mathrm{Address}\ \texttt{"GOD"}$$

3.3 `Gem` — token model

In this model, all tokens behave in the same simple way.[1] We omit the ERC20 concept of "allowances."

$$\textbf{data}\ \texttt{Gem} = \texttt{Gem}\ \{$$

- $\texttt{balanceOf} :: \mathrm{Map}\ \mathrm{Holder}\ \texttt{Wad}$

$$\}\ \textbf{deriving}\ (\mathrm{Eq}, \mathrm{Show})$$

We distinguish between tokens held by vaults, tokens held by the test driver, and tokens held by CDP owners.

[1] In the real world, token semantics can differ, despite nominally following the ERC20 interface. Maker governance therefore involves due diligence on collateral token contracts.

data Holder = InAccount Address
  | InVault (Id `Jar`)
  | InToy
  deriving (Eq, Show, Ord)

3.4 `Jar` — collateral vaults

data `Jar` = `Jar` {

- `gem` :: `Gem`, Collateral token
- `tag` :: `Wad`, Market price
- `zzz` :: `Sec` Price expiration

} **deriving** (Eq, Show)

3.5 `Ilk` — CDP type

data `Ilk` = `Ilk` {

- `jar` :: Id `Jar`, Collateral vault
- `axe` :: `Ray`, Liquidation penalty
- `hat` :: `Wad`, Debt ceiling
- `mat` :: `Ray`, Liquidation ratio
- `tax` :: `Ray`, Stability fee
- `lag` :: `Sec`, Limbo duration
- `rho` :: `Sec`, Last dripped
- `rum` :: `Wad`, Total debt in debt unit
- `chi` :: `Ray` Dai value of debt unit

} **deriving** (Eq, Show)

3.6 `Urn` — collateralized debt position (CDP)

data `Urn` = `Urn` {

- `cat` :: Maybe Address, Address of liquidation initiator

- `vow` :: Maybe Address, Address of liquidation contract
- `lad` :: Address, Issuer
- `ilk` :: Id `Ilk`, CDP type
- `art` :: `Wad`, Outstanding debt in debt unit
- `jam` :: `Wad` Collateral amount in debt unit

} **deriving** (Eq, Show)

3.7 `Vat` — CDP engine

data `Vat` = `Vat` {

- `fix` :: `Wad`, Market price
- `how` :: `Ray`, Sensitivity
- `par` :: `Wad`, Target price
- `way` :: `Ray`, Target rate
- `tau` :: `Sec`, Last prodded
- `joy` :: `Wad`, Unprocessed stability fees
- `sin` :: `Wad`, Bad debt from liquidated CDPs
- `jars` :: Map (Id `Jar`) `Jar`, Collateral tokens
- `ilks` :: Map (Id `Ilk`) `Ilk`, CDP types
- `urns` :: Map (Id `Urn`) `Urn` CDPs

} **deriving** (Eq, Show)

3.8 System model

data System = System {

- `vat` :: `Vat`, Root Maker entity
- `era` :: `Sec`, Current time stamp
- `sender` :: Address, Sender of current act
- `accounts` :: [Address] For test suites

} **deriving** (Eq, Show)

Lens fields

$$\begin{array}{l}
\mathit{makeLenses}\ \text{''}\ \texttt{Gem} \\
\mathit{makeLenses}\ \text{''}\ \texttt{Jar} \\
\mathit{makeLenses}\ \text{''}\ \texttt{Ilk} \\
\mathit{makeLenses}\ \text{''}\ \texttt{Urn} \\
\mathit{makeLenses}\ \text{''}\ \texttt{Vat} \\
\mathit{makeLenses}\ \text{''}\ \text{System}
\end{array}$$

3.9 Default data

$\mathit{defaultIlk} :: \text{Id}\ \texttt{Jar} \to \texttt{Ilk}$
$\mathit{defaultIlk}\ id_{\texttt{jar}} = \texttt{Ilk}\ \{$

- $\texttt{jar} = id_{\texttt{jar}}$,
- $\texttt{axe} = \texttt{Ray}\ 1$,
- $\texttt{mat} = \texttt{Ray}\ 1$,
- $\texttt{tax} = \texttt{Ray}\ 1$,
- $\texttt{hat} = \texttt{Wad}\ 0$,
- $\texttt{lag} = \texttt{Sec}\ 0$,
- $\texttt{chi} = \texttt{Ray}\ 1$,
- $\texttt{rum} = \texttt{Wad}\ 0$,
- $\texttt{rho} = \texttt{Sec}\ 0$

$\}$

$\mathit{defaultUrn} :: \text{Id}\ \texttt{Ilk} \to \text{Address} \to \texttt{Urn}$
$\mathit{defaultUrn}\ id_{\texttt{ilk}}\ id_{\texttt{lad}} = \texttt{Urn}\ \{$

- $\texttt{vow} = \text{Nothing}$,
- $\texttt{cat} = \text{Nothing}$,
- $\texttt{lad} = id_{\texttt{lad}}$,
- $\texttt{ilk} = id_{\texttt{ilk}}$,
- $\texttt{art} = \texttt{Wad}\ 0$,
- $\texttt{jam} = \texttt{Wad}\ 0$

$\}$

$\mathit{initialVat} :: \texttt{Ray} \to \texttt{Vat}$
$\mathit{initialVat}\ \texttt{how}_0 = \texttt{Vat}\ \{$

- tau $= 0$,
- fix $=$ Wad 1,
- par $=$ Wad 1,
- how $=$ $\texttt{how}_0$,
- way $=$ Ray 1,
- joy $=$ Wad 0,
- sin $=$ Wad 0,
- ilks $= \varnothing$,
- urns $= \varnothing$,
- jars $=$
  singleton id_{DAI} Jar {
  - gem $=$ Gem {
    - balanceOf $= \varnothing$

    },
  - tag $=$ Wad 0,
  - zzz $= 0$

  }

}

$\mathit{initialSystem} :: \texttt{Ray} \to \text{System}$
$\mathit{initialSystem}\ \texttt{how}_0 = \text{System}\ \{$

- vat $= \mathit{initialVat}\ \texttt{how}_0$,
- era $= 0$,
- sender $= id_{god}$,
- accounts $= \mathit{mempty}$

}

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 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.

$$\begin{array}{l}
\textbf{data}\ \text{Stage} = \texttt{Dread} \mid \texttt{Grief} \mid \texttt{Panic} \mid \texttt{Worry} \mid \texttt{Anger} \mid \texttt{Pride} \\
\quad \textbf{deriving}\ (\text{Eq}, \text{Ord}, \text{Show})
\end{array}$$

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

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{analyze}\ \texttt{era}_0\ \texttt{par}_0\ \texttt{urn}_0\ \texttt{ilk}_0\ \texttt{jar}_0 =} \\
\quad \textbf{if} & & \\
\quad\quad \multicolumn{2}{l}{\textsf{Undergoing liquidation?}} & \\
\quad\quad\quad \mid \mathit{view}\ \texttt{vow}\ \texttt{urn}_0 \not\equiv \text{Nothing} & \rightarrow \texttt{Dread} \\
\quad\quad \multicolumn{2}{l}{\textsf{Liquidation triggered?}} & \\
\quad\quad\quad \mid \mathit{view}\ \texttt{cat}\ \texttt{urn}_0 \not\equiv \text{Nothing} & \rightarrow \texttt{Grief} \\
\quad\quad \multicolumn{2}{l}{\textsf{Undercollateralized?}} & \\
\quad\quad\quad \mid \texttt{pro} < \texttt{min} & \rightarrow \texttt{Panic} \\
\quad\quad \multicolumn{2}{l}{\textsf{Price feed expired?}} & \\
\quad\quad\quad \mid \texttt{era}_0 > \mathit{view}\ \texttt{zzz}\ \texttt{jar}_0 + \mathit{view}\ \texttt{lag}\ \texttt{ilk}_0 & \rightarrow \texttt{Panic} \\
\quad\quad \multicolumn{2}{l}{\textsf{Price feed in limbo?}} & \\
\quad\quad\quad \mid \mathit{view}\ \texttt{zzz}\ \texttt{jar}_0 < \texttt{era}_0 & \rightarrow \texttt{Worry} \\
\quad\quad \multicolumn{2}{l}{\textsf{Debt ceiling reached?}} & \\
\quad\quad\quad \mid \texttt{cap} > \mathit{view}\ \texttt{hat}\ \texttt{ilk}_0 & \rightarrow \texttt{Anger} \\
\quad\quad \multicolumn{2}{l}{\textsf{Safely overcollateralized}} & \\
\quad\quad\quad \mid \mathit{otherwise} & \rightarrow \texttt{Pride} \\
\quad \textbf{where} & & \\
\quad\quad \multicolumn{2}{l}{\textsf{CDP's collateral value in SDR:}} & \\
\quad\quad \multicolumn{2}{l}{\texttt{pro} = \mathit{view}\ \texttt{jam}\ \texttt{urn}_0 * \mathit{view}\ \texttt{tag}\ \texttt{jar}_0} & \\
\quad\quad \multicolumn{2}{l}{\textsf{CDP type's total debt in SDR:}} & \\
\quad\quad \multicolumn{2}{l}{\texttt{cap} = (\mathit{view}\ \texttt{rum}\ \texttt{ilk}_0 * \mathit{cast}\ (\mathit{view}\ \texttt{chi}\ \texttt{ilk}_0)) :: \texttt{Wad}} & \\
\quad\quad \multicolumn{2}{l}{\textsf{CDP's debt in SDR:}} & \\
\quad\quad \multicolumn{2}{l}{\texttt{con} = \mathit{view}\ \texttt{art}\ \texttt{urn}_0 * \mathit{cast}\ (\mathit{view}\ \texttt{chi}\ \texttt{ilk}_0) * \texttt{par}_0} & \\
\quad\quad \multicolumn{2}{l}{\textsf{Required collateral as per liquidation ratio:}} & \\
\quad\quad \multicolumn{2}{l}{\texttt{min} = \texttt{con} * \mathit{cast}\ (\mathit{view}\ \texttt{mat}\ \texttt{ilk}_0)} &
\end{array}$$

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.

$\texttt{gaze}\ id_{\texttt{urn}} = \mathbf{do}$

Perform dai revaluation and rate adjustment

$\quad\texttt{prod}$

Update price of specific debt unit

$\quad id_{\texttt{ilk}} \leftarrow look\ (\texttt{vat}\,.\,\texttt{urn}s\,.\,ix\ id_{\texttt{urn}}\,.\,\texttt{ilk})$

$\quad\texttt{drip}\ id_{\texttt{ilk}}$

Read parameters for risk analysis

$\quad\texttt{era}_0 \leftarrow use\ \texttt{era}$

$\quad\texttt{par}_0 \leftarrow use\ (\texttt{vat}\,.\,\texttt{par})$

$\quad\texttt{urn}_0 \leftarrow look\ (\texttt{vat}\,.\,\texttt{urn}s\,.\,ix\ id_{\texttt{urn}})$

$\quad\texttt{ilk}_0 \leftarrow look\ (\texttt{vat}\,.\,\texttt{ilk}s\,.\,ix\ (view\ \texttt{ilk}\ \texttt{urn}_0))$

$\quad\texttt{jar}_0 \leftarrow look\ (\texttt{vat}\,.\,\texttt{jar}s\,.\,ix\ (view\ \texttt{jar}\ \texttt{ilk}_0))$

Return risk stage of CDP

$\quad return\ (analyze\ \texttt{era}_0\ \texttt{par}_0\ \texttt{urn}_0\ \texttt{ilk}_0\ \texttt{jar}_0)$

4.2 Lending

Any Ethereum address can open one or more accounts with the system using `open`, specifying an account identifier (self-chosen) and a CDP type.

$\texttt{open}\ id_{\texttt{urn}}\ id_{\texttt{ilk}} = \textbf{do}$

Fail if account identifier is taken

$\quad none\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ at\ id_{\texttt{urn}})$

Create a CDP record with the sender as owner

$\quad id_{\texttt{lad}} \leftarrow use\ sender$

$\quad initializeTo\ (defaultUrn\ id_{\texttt{ilk}}\ id_{\texttt{lad}})$

$\qquad (\texttt{vat}\ .\ \texttt{urn}s\ .\ at\ id_{\texttt{urn}})$

The owner of a CDP can transfer its ownership at any time using `give`.

$\texttt{give}\ id_{\texttt{urn}}\ id_{\texttt{lad}} = \textbf{do}$

Fail if sender is not the CDP owner

$\quad owns\ id_{\texttt{urn}}\ id_{\texttt{lad}}$

Transfer ownership

$\quad \texttt{vat}\ .\ \texttt{urn}s\ .\ ix\ id_{\texttt{urn}}\ .\ \texttt{lad} := id_{\texttt{lad}}$

$\texttt{lock}\ id_{\texttt{urn}}\ \texttt{wad}_{\texttt{gem}} = \textbf{do}$

Fail if sender is not the CDP owner

$\quad id_{\texttt{lad}} \leftarrow use\ sender$

$\quad owns\ id_{\texttt{urn}}\ id_{\texttt{lad}}$

Ensure CDP exists; identify collateral type

$\quad id_{\texttt{ilk}} \leftarrow look\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ ix\ id_{\texttt{urn}}\ .\ \texttt{ilk})$

$\quad id_{\texttt{jar}} \leftarrow look\ (\texttt{vat}\ .\ \texttt{ilk}s\ .\ ix\ id_{\texttt{ilk}}\ .\ \texttt{jar})$

Record an increase in collateral

$\quad increaseBy\ \texttt{wad}_{\texttt{gem}}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ ix\ id_{\texttt{urn}}\ .\ \texttt{jam})$

Take sender's tokens

$\quad id_{\texttt{lad}} \leftarrow use\ sender$

$\quad \texttt{pull}\ id_{\texttt{jar}}\ id_{\texttt{lad}}\ \texttt{wad}_{\texttt{gem}}$

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

Fail if sender is not the CDP owner

$id_{\mathtt{lad}} \leftarrow \mathit{use\ sender}$
$\mathit{owns}\ id_{\mathtt{urn}}\ id_{\mathtt{lad}}$

Decrease the collateral amount
$\mathit{decreaseBy}\ \mathtt{wad}_{\mathtt{gem}}\ (\mathtt{vat}\ .\ \mathtt{urn}s\ .\ \mathit{ix}\ id_{\mathtt{urn}}\ .\ \mathtt{jam})$

Roll back if undercollateralized
$\mathtt{gaze}\ id_{\mathtt{urn}} \ggg= \mathtt{aver}\ .\ (\equiv \mathtt{Pride})$

Send the collateral to the CDP owner
$id_{\mathtt{ilk}} \leftarrow \mathit{look}\ (\mathtt{vat}\ .\ \mathtt{urn}s\ .\ \mathit{ix}\ id_{\mathtt{urn}}\ .\ \mathtt{ilk})$
$id_{\mathtt{jar}} \leftarrow \mathit{look}\ (\mathtt{vat}\ .\ \mathtt{ilk}s\ .\ \mathit{ix}\ id_{\mathtt{ilk}}\ .\ \mathtt{jar})$
$\mathtt{push}\ id_{\mathtt{jar}}\ id_{\mathtt{lad}}\ \mathtt{wad}_{\mathtt{gem}}$

$\mathtt{draw}\ id_{\mathtt{urn}}\ \mathtt{wad}_{\text{DAI}} = \mathbf{do}$

Fail if sender is not the CDP owner
$id_{\mathtt{lad}} \leftarrow \mathit{use\ sender}$
$\mathit{owns}\ id_{\mathtt{urn}}\ id_{\mathtt{lad}}$

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

Denominate draw amount in debt unit
$\mathbf{let}\ \mathtt{wad}_{\mathtt{chi}} = \mathtt{wad}_{\text{DAI}}\ /\ \mathit{cast}\ \mathtt{chi}_1$

Increase debt
$\mathit{increaseBy}\ \mathtt{wad}_{\mathtt{chi}}\ (\mathtt{vat}\ .\ \mathtt{urn}s\ .\ \mathit{ix}\ id_{\mathtt{urn}}\ .\ \mathtt{art})$

Roll back unless overcollateralized
$\mathtt{gaze}\ id_{\mathtt{urn}} \ggg= \mathtt{aver}\ .\ (\equiv \mathtt{Pride})$

Mint dai and send to the CDP owner
$\mathtt{mint}\ id_{\text{DAI}}\ \mathtt{wad}_{\text{DAI}}$
$\mathtt{push}\ id_{\text{DAI}}\ id_{\mathtt{lad}}\ \mathtt{wad}_{\text{DAI}}$

$\mathtt{wipe}\ id_{\mathtt{urn}}\ \mathtt{wad}_{\text{DAI}} = \mathbf{do}$

Fail if sender is not the CDP owner
$id_{\mathtt{lad}} \leftarrow \mathit{use\ sender}$
$\mathit{owns}\ id_{\mathtt{urn}}\ id_{\mathtt{lad}}$

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

Roll back unless overcollateralized

$\texttt{gaze}\ id_{\texttt{urn}} \ggg\!= \texttt{aver} \ .\ (\equiv \texttt{Pride})$

Denominate dai amount in debt unit

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

Reduce debt

$\mathit{decreaseBy}\ \texttt{wad}_{\texttt{chi}}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ \mathit{ix}\ id_{\texttt{urn}}\ .\ \texttt{art})$

Take dai from CDP owner, or roll back

$\texttt{pull}\ id_{\text{DAI}}\ id_{\texttt{lad}}\ \texttt{wad}_{\text{DAI}}$

Destroy dai

$\texttt{burn}\ id_{\text{DAI}}\ \texttt{wad}_{\text{DAI}}$

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

Update value of debt unit

$id_{\texttt{ilk}} \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ \mathit{ix}\ id_{\texttt{urn}}\ .\ \texttt{ilk})$

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

Attempt to repay all the CDP's outstanding dai

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

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

Reclaim all the collateral

$\mathit{jam0} \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ \mathit{ix}\ id_{\texttt{urn}}\ .\ \texttt{jam})$

$\texttt{free}\ id_{\texttt{urn}}\ \mathit{jam0}$

Nullify the CDP

$\texttt{vat}\ .\ \texttt{urn}s\ .\ \mathit{at}\ id_{\texttt{urn}} := \text{Nothing}$

4.3 Adjustment

$$
\begin{array}{l}
\texttt{prod} = \textbf{do} \\
\quad \texttt{era}_0 \leftarrow \mathit{use}\ \texttt{era} \\
\quad \texttt{tau}_0 \leftarrow \mathit{use}\ (\texttt{vat} \mathbin{.} \texttt{tau}) \\
\quad \texttt{fix}_0 \leftarrow \mathit{use}\ (\texttt{vat} \mathbin{.} \texttt{fix}) \\
\quad \texttt{par}_0 \leftarrow \mathit{use}\ (\texttt{vat} \mathbin{.} \texttt{par}) \\
\quad \texttt{how}_0 \leftarrow \mathit{use}\ (\texttt{vat} \mathbin{.} \texttt{how}) \\
\quad \texttt{way}_0 \leftarrow \mathit{use}\ (\texttt{vat} \mathbin{.} \texttt{way}) \\
\quad \textbf{let}
\end{array}
$$

Time difference in seconds

$$\mathit{age}\ = \texttt{era}_0 - \texttt{tau}_0$$

Current target rate applied to target price

$$\texttt{par}_1 = \texttt{par}_0 * \mathit{cast}\ (\texttt{way}_0 \uparrow\uparrow \mathit{age})$$

Sensitivity parameter applied over time

$$\mathit{wag} = \texttt{how}_0 * \mathit{fromIntegral}\ \mathit{age}$$

Target rate scaled up or down

$$
\begin{array}{l}
\texttt{way}_1 = \mathit{inj}\ (\mathit{prj}\ \texttt{way}_0 + \\
\qquad\qquad \textbf{if}\ \texttt{fix}_0 < \texttt{par}_0\ \textbf{then}\ \mathit{wag}\ \textbf{else} - \mathit{wag})
\end{array}
$$

$$
\begin{array}{l}
\quad \texttt{vat} \mathbin{.} \texttt{par} := \texttt{par}_1 \\
\quad \texttt{vat} \mathbin{.} \texttt{way} := \texttt{way}_1 \\
\quad \texttt{vat} \mathbin{.} \texttt{tau} := \texttt{era}_0 \\
\quad \textbf{where}
\end{array}
$$

Convert between multiplicative and additive form

$$
\begin{array}{ll}
\mathit{prj}\ x & = \textbf{if}\ x \geqslant 1\ \textbf{then}\ x - 1\ \textbf{else}\ 1 - 1 \mathbin{/} x \\
\mathit{inj}\ x & = \textbf{if}\ x \geqslant 0\ \textbf{then}\ x + 1\ \textbf{else}\ 1 \mathbin{/} (1 - x)
\end{array}
$$

```
drip id_ilk = do
  rho0 ← look (vat . ilks . ix id_ilk . rho)   Time stamp of previous drip
  tax0 ← look (vat . ilks . ix id_ilk . tax)   Current stability fee
  chi0 ← look (vat . ilks . ix id_ilk . chi)   Current value of debt unit
  rum0 ← look (vat . ilks . ix id_ilk . rum)   Current total debt in debt unit
  joy0 ← look (vat . joy)                      Current unprocessed stability fee revenue
  era0 ← use era                               Current time stamp
  let
    age  = era0 − rho0
    chi1 = chi0 ∗ tax0 ↑↑ age
    joy1 = joy0 + (cast (chi1 − chi0) :: Wad) ∗ rum0
  vat . ilks . ix id_ilk . chi := chi1
  vat . ilks . ix id_ilk . rho := era0
  vat . joy := joy1
  return chi1
```

4.4 Feedback

```
mark id_jar tag1 zzz1 =
  auth $ do
    vat . jars . ix id_jar . tag := tag1
    vat . jars . ix id_jar . zzz := zzz1
```

```
tell wad_gem =
  auth $ do
    vat . fix := wad_gem
```

4.5 Liquidation

```
bite id_urn = do
   Fail if CDP is not in need of liquidation
    gaze id_urn ≫= aver . (≡ Panic)
   Record the sender as the liquidation initiator
```

$id_{\texttt{cat}} \leftarrow use\ sender$
$\texttt{vat . urn}s\texttt{ . }ix\ id_{\texttt{urn}}\texttt{ . cat} := \text{Just}\ id_{\texttt{cat}}$

Read current debt

$\texttt{art}_0 \leftarrow look\ (\texttt{vat . urn}s\texttt{ . }ix\ id_{\texttt{urn}}\texttt{ . art})$

Update value of debt unit

$id_{\texttt{ilk}} \leftarrow look\ (\texttt{vat . urn}s\texttt{ . }ix\ id_{\texttt{urn}}\texttt{ . ilk})$
$\texttt{chi}_1 \leftarrow \texttt{drip}\ id_{\texttt{ilk}}$

Read liquidation penalty ratio

$id_{\texttt{ilk}} \leftarrow look\ (\texttt{vat . urn}s\texttt{ . }ix\ id_{\texttt{urn}}\texttt{ . ilk})$
$\texttt{axe}_0 \leftarrow look\ (\texttt{vat . ilk}s\texttt{ . }ix\ id_{\texttt{ilk}}\texttt{ . axe})$

Apply liquidation penalty to debt

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

Update CDP debt

$\texttt{vat . urn}s\texttt{ . }ix\ id_{\texttt{urn}}\texttt{ . art} := \texttt{art}_1$

Record as bad debt

$increaseBy\ (\texttt{art}_1 * cast\ \texttt{chi}_1)\ (\texttt{vat . sin})$

$\texttt{grab}\ id_{\texttt{urn}} =$
$\quad \texttt{auth}\ \$\ \textbf{do}$

Fail if CDP is not marked for liquidation

$\texttt{gaze}\ id_{\texttt{urn}} \ggg \texttt{aver . } (\equiv \texttt{Grief})$

Record the sender as the CDP's settler

$id_{\texttt{vow}} \leftarrow use\ sender$
$\texttt{vat . urn}s\texttt{ . }ix\ id_{\texttt{urn}}\texttt{ . vow} := \text{Just}\ id_{\texttt{vow}}$

Forget the CDP's requester of liquidation

$\texttt{vat . urn}s\texttt{ . }ix\ id_{\texttt{urn}}\texttt{ . cat} := \text{Nothing}$

$\texttt{plop}\ id_{\texttt{urn}}\ \texttt{wad}_{\text{DAI}} =$
$\quad \texttt{auth}\ \$\ \textbf{do}$

Fail unless CDP is in liquidation

$\texttt{gaze}\ id_{\texttt{urn}} \ggg \texttt{aver . } (\equiv \texttt{Dread})$

Forget the CDP's settler

$\texttt{vat . urn}s\texttt{ . }ix\ id_{\texttt{urn}}\texttt{ . vow} := \text{Nothing}$

Return some amount of excess auction gains

$\texttt{vat . urn}s\texttt{ . }ix\ id_{\texttt{urn}}\texttt{ . jam} := \texttt{wad}_{\text{DAI}}$

$\texttt{heal}\ \texttt{wad}_{\text{DAI}} =$
$\quad\texttt{auth}\ \$\ \textbf{do}$
$\qquad \textit{decreaseBy}\ \texttt{wad}_{\text{DAI}}\ (\texttt{vat} \mathbin{.} \texttt{sin})$

$\texttt{love}\ \texttt{wad}_{\text{DAI}} =$
$\quad\texttt{auth}\ \$\ \textbf{do}$
$\qquad \textit{decreaseBy}\ \texttt{wad}_{\text{DAI}}\ (\texttt{vat} \mathbin{.} \texttt{joy})$

4.6 Governance

$\texttt{form}\ id_{\texttt{ilk}}\ id_{\texttt{jar}} =$
$\quad\texttt{auth}\ \$\ \textbf{do}$
$\qquad \textit{initializeTo}\ (\textit{defaultIlk}\ id_{\texttt{jar}})$
$\qquad\quad (\texttt{vat} \mathbin{.} \texttt{ilk}s \mathbin{.} \textit{at}\ id_{\texttt{ilk}})$

$\texttt{frob}\ \texttt{how}_1 =$
$\quad\texttt{auth}\ \$\ \textbf{do}$
$\qquad \texttt{vat} \mathbin{.} \texttt{how} := \texttt{how}_1$

$\texttt{chop}\ id_{\texttt{ilk}}\ \texttt{axe}_1 =$
$\quad\texttt{auth}\ \$\ \textbf{do}$
$\qquad \texttt{vat} \mathbin{.} \texttt{ilk}s \mathbin{.} \textit{ix}\ id_{\texttt{ilk}} \mathbin{.} \texttt{axe} := \texttt{axe}_1$

$\texttt{cork}\ id_{\texttt{ilk}}\ \texttt{hat}_1 =$
$\quad\texttt{auth}\ \$\ \textbf{do}$
$\qquad \texttt{vat} \mathbin{.} \texttt{ilk}s \mathbin{.} \textit{ix}\ id_{\texttt{ilk}} \mathbin{.} \texttt{hat} := \texttt{hat}_1$

$\texttt{calm}\ id_{\texttt{ilk}}\ \texttt{lag}_1 =$
$\quad\texttt{auth}\ \$\ \textbf{do}$
$\qquad \texttt{vat} \mathbin{.} \texttt{ilk}s \mathbin{.} \textit{ix}\ id_{\texttt{ilk}} \mathbin{.} \texttt{lag} := \texttt{lag}_1$

$\texttt{cuff}\ id_{\texttt{ilk}}\ \texttt{mat}_1 =$
$\quad \texttt{auth}\ \$\ \textbf{do}$
$\qquad \texttt{vat} \,.\, \texttt{ilk}s \,.\, ix\ id_{\texttt{ilk}} \,.\, \texttt{mat} := \texttt{mat}_1$

$\texttt{crop}\ id_{\texttt{ilk}}\ \texttt{tax}_1 =$
$\quad \texttt{auth}\ \$\ \textbf{do}$
$\qquad \texttt{drip}\ id_{\texttt{ilk}}$
$\qquad \texttt{vat} \,.\, \texttt{ilk}s \,.\, ix\ id_{\texttt{ilk}} \,.\, \texttt{tax} := \texttt{tax}_1$

4.7 Treasury

$\texttt{pull}\ id_{\texttt{jar}}\ id_{\texttt{lad}}\ \texttt{wad}_{\texttt{gem}} =$
$\quad transfer\ id_{\texttt{jar}}\ \texttt{wad}_{\texttt{gem}}$
$\qquad (\text{InAccount}\ id_{\texttt{lad}})$
$\qquad (\text{InVault}\quad id_{\texttt{jar}})$

$\texttt{push}\ id_{\texttt{jar}}\ id_{\texttt{lad}}\ \texttt{wad}_{\texttt{gem}} =$
$\quad transfer\ id_{\texttt{jar}}\ \texttt{wad}_{\texttt{gem}}$
$\qquad (\text{InVault}\quad id_{\texttt{jar}})$
$\qquad (\text{InAccount}\ id_{\texttt{lad}})$

$\texttt{mint}\ id_{\texttt{jar}}\ \texttt{wad}_0 =$
$\quad zoom\ (\texttt{vat} \,.\, \texttt{jar}s \,.\, ix\ id_{\texttt{jar}} \,.\, \texttt{gem})\ \$\ \textbf{do}$
$\qquad increaseBy\ \texttt{wad}_0\ (balanceOf \,.\, ix\ (\text{InVault}\ id_{\texttt{jar}}))$

$\texttt{burn}\ id_{\texttt{jar}}\ \texttt{wad}_0 =$
$\quad zoom\ (\texttt{vat} \,.\, \texttt{jar}s \,.\, ix\ id_{\texttt{jar}} \,.\, \texttt{gem})\ \$\ \textbf{do}$
$\qquad decreaseBy\ \texttt{wad}_0\ (balanceOf \,.\, ix\ (\text{InVault}\ id_{\texttt{jar}}))$

4.8 Manipulation

$\texttt{warp}\ t = \texttt{auth}\ (\textbf{do}\ \mathit{increaseBy}\ t\ \texttt{era})$

$\texttt{mine}\ \mathit{id}_{\texttt{jar}} = \textbf{do}$
$\quad \mathit{initializeTo}$
$\quad\quad (\texttt{Jar}\ \{$
- $\texttt{gem} = \texttt{Gem}\ (\mathit{singleton}\ \text{InToy}\ 1000000000000),$
- $\texttt{tag} = \texttt{Wad}\ 0,$
- $\texttt{zzz} = 0\})$

$\quad\quad (\texttt{vat}\ .\ \texttt{jar}s\ .\ \mathit{at}\ \mathit{id}_{\texttt{jar}})$

$\texttt{hand}\ \mathit{dst}\ \texttt{wad}_{\texttt{gem}}\ \mathit{id}_{\texttt{jar}} = \textbf{do}$
$\quad \mathit{transfer}\ \mathit{id}_{\texttt{jar}}\ \texttt{wad}_{\texttt{gem}}$
$\quad\quad \text{InToy}\ (\text{InAccount}\ \mathit{dst})$

$\texttt{sire}\ \texttt{lad} = \textbf{do}\ \mathit{prepend}\ \texttt{lad}\ \mathit{accounts}$

4.9 Other stuff

$$
\begin{array}{l}
\mathit{perform} :: \text{Act} \to \text{Maker}\ () \\
\mathit{perform}\ x = \\
\quad \textbf{let}\ ?\mathit{act} = x\ \textbf{in case}\ x\ \textbf{of} \\
\quad\quad \begin{array}{ll}
\texttt{Form}\ \mathit{id}\ \texttt{jar} & \to \texttt{form}\ \mathit{id}\ \texttt{jar} \\
\texttt{Mark}\ \texttt{jar}\ \texttt{tag}\ \texttt{zzz} & \to \texttt{mark}\ \texttt{jar}\ \texttt{tag}\ \texttt{zzz} \\
\texttt{Open}\ \mathit{id}\ \texttt{ilk} & \to \texttt{open}\ \mathit{id}\ \texttt{ilk} \\
\texttt{Tell}\ \texttt{wad} & \to \texttt{tell}\ \texttt{wad} \\
\texttt{Frob}\ \texttt{ray} & \to \texttt{frob}\ \texttt{ray} \\
\texttt{Prod} & \to \texttt{prod} \\
\texttt{Warp}\ t & \to \texttt{warp}\ t \\
\texttt{Give}\ \texttt{urn}\ \texttt{lad} & \to \texttt{give}\ \texttt{urn}\ \texttt{lad} \\
\texttt{Pull}\ \texttt{jar}\ \texttt{lad}\ \texttt{wad} & \to \texttt{pull}\ \texttt{jar}\ \texttt{lad}\ \texttt{wad} \\
\texttt{Lock}\ \texttt{urn}\ \texttt{wad} & \to \texttt{lock}\ \texttt{urn}\ \texttt{wad}
\end{array}
\end{array}
$$

$$
\begin{array}{ll}
\texttt{Mine}\ id & \to \texttt{mine}\ id \\
\texttt{Hand}\ \texttt{lad}\ \texttt{wad}\ \texttt{jar} & \to \texttt{hand}\ \texttt{lad}\ \texttt{wad}\ \texttt{jar} \\
\texttt{Sire}\ \texttt{lad} & \to \texttt{sire}\ \texttt{lad}
\end{array}
$$

$$
\begin{array}{l}
being :: \mathrm{Act} \to \mathrm{Address} \to \mathrm{Maker}\ () \\
being\ x\ who = \mathbf{do} \\
\quad old \quad\ \ \leftarrow use\ sender \\
\quad sender := who \\
\quad y \qquad\ \ \leftarrow perform\ x \\
\quad sender := old \\
\quad return\ y
\end{array}
$$

$transfer\ id_{\texttt{jar}}\ \texttt{wad}\ src\ dst =$

Operate in the token's balance table

$zoom\ (\texttt{vat}\ .\ \texttt{jar}s\ .\ ix\ id_{\texttt{jar}}\ .\ \texttt{gem}\ .\ balanceOf)\ \$\ \mathbf{do}$

Fail if source balance insufficient

$balance \leftarrow look\ (ix\ src)$
$\texttt{aver}\ (balance \geqslant \texttt{wad})$

Decrease source balance

$decreaseBy\ \texttt{wad}\ (ix\ src)$

Increase destination balance

$initializeTo\ 0 \quad (at\ dst)$
$increaseBy\ \texttt{wad}\ (ix\ dst)$

Chapter 5

Act framework

5.1 Act descriptions

We define the Maker act vocabulary as a data type.

```
data Act =
    Bite (Id Urn)
  | Draw (Id Urn) Wad
  | Form (Id Ilk) (Id Jar)
  | Free (Id Urn) Wad
  | Frob Ray
  | Give (Id Urn) Address
  | Grab (Id Urn)
  | Heal Wad
  | Lock (Id Urn) Wad
  | Love Wad
  | Mark (Id Jar) Wad     Sec
  | Open (Id Urn) (Id Ilk)
  | Prod
  | Pull (Id Jar) Address Wad
  | Shut (Id Urn)
  | Tell Wad
  | Warp Sec
  | Wipe (Id Urn) Wad
  | Mine (Id Jar)
  | Hand Address Wad      (Id Jar)
  | Sire Address
```

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

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

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

5.2 The Maker monad

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

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

```
type Maker a = Maker' System a
```

```
type instance Zoomed (Maker' s) = Focusing (Except Error)
instance Zoom (Maker' s) (Maker' t) s t where
  zoom l (Maker m) = Maker (zoom l m)
```

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

5.3 Asserting

$$\texttt{aver}\ x = \mathit{unless}\ x\ (\mathit{throwError}\ (\mathrm{AssertError}\ ?\mathit{act}))$$

$$\begin{array}{l} \mathit{none}\ x = \mathit{preuse}\ x \ggg \lambda\mathbf{case} \\ \quad \mathrm{Nothing} \to \mathit{return}\ () \\ \quad \mathrm{Just}\ \_ \to \mathit{throwError}\ (\mathrm{AssertError}\ ?\mathit{act}) \end{array}$$

$$\begin{array}{l} \mathit{look}\ f = \mathit{preuse}\ f \ggg \lambda\mathbf{case} \\ \quad \mathrm{Nothing} \to \mathit{throwError}\ (\mathrm{AssertError}\ ?\mathit{act}) \\ \quad \mathrm{Just}\ x \to \mathit{return}\ x \end{array}$$

We define $\mathit{owns}\ \mathit{id}_{\texttt{urn}}\ \mathit{id}_{\texttt{lad}}$ as an assertion that the given CDP is owned by the given account.

$$\begin{array}{l} \mathit{owns}\ \mathit{id}_{\texttt{urn}}\ \mathit{id}_{\texttt{lad}} = \mathbf{do} \\ \quad \mathit{id}_{\mathit{sender}} \leftarrow \mathit{use}\ \mathit{sender} \\ \quad \texttt{aver}\ (\mathit{id}_{\mathit{sender}} \equiv \mathit{id}_{\texttt{lad}}) \\ \quad \mathit{return}\ \mathit{id}_{\mathit{sender}} \end{array}$$

5.4 Modifiers

$$\begin{array}{l} \texttt{auth}\ \mathit{continue} = \mathbf{do} \\ \quad s \leftarrow \mathit{use}\ \mathit{sender} \\ \quad \mathit{unless}\ (s \equiv \mathit{id}_{\mathit{god}}) \\ \quad\quad (\mathit{throwError}\ \mathrm{AuthError}) \\ \quad \mathit{continue} \end{array}$$

Chapter 6

Testing

Sketches for property stuff...

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

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

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

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

$$
\begin{array}{l}
\quad \textbf{where} \\
\quad \textit{getter}\ x = (\textit{view}\ f\ x, \textit{view}\ g\ x) \\
\quad \textit{setter}\ x\ (a, b) = \textit{set}\ f\ a\ (\textit{set}\ g\ b\ x)
\end{array}
$$

$$
\begin{array}{l}
\textit{keeps} :: \text{Parameter} \rightarrow \text{Maker}\ () \rightarrow \text{System} \rightarrow \text{Bool} \\
\textit{keeps}\ \texttt{Fix} = \textit{maintains}\ (\texttt{vat}\ .\ \texttt{fix}) \\
\textit{keeps}\ \texttt{Par} = \textit{maintains}\ (\texttt{vat}\ .\ \texttt{par}) \\
\textit{keeps}\ \texttt{Way} = \textit{maintains}\ (\texttt{vat}\ .\ \texttt{way})
\end{array}
$$

Thus:

$$
\begin{array}{l}
\textit{foo}\ \texttt{sys}_0 = \textit{all}\ (\lambda f \rightarrow f\ \texttt{sys}_0) \\
\quad [\,\textit{changesOnly}\ ((\texttt{vat}\ .\ \texttt{par})\ `\textit{also}` \\
\qquad\qquad\qquad\quad (\texttt{vat}\ .\ \texttt{way})) \\
\qquad (\textit{perform}\ \texttt{Prod})\,]
\end{array}
$$

Appendix A

Prelude

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

import Prelude *as* X (

Conversions to and from strings

$\text{Read}\ (..), \text{Show}\ (..),$

Comparisons

$\text{Eq}\ (..), \text{Ord}\ (..),$

Core abstractions

$\text{Functor}\quad (\mathit{fmap}),$
$\text{Applicative}\ (),$
$\text{Monad}\quad (\mathit{return}, (\ggg\!=)),$

Numeric classes

$\text{Num}\ (..), \text{Integral}\ (), \text{Enum}\ (),$

Numeric conversions

$\text{Real}\ (..), \text{Fractional}\ (..),$
$\text{RealFrac}\ (..),$
$\mathit{fromIntegral},$

Simple types

$\text{Integer}, \text{Int}, \text{String},$

Algebraic types

$\text{Bool}\quad (\text{True}, \text{False}),$

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

Functional operators
(.), ($),

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

Utilities
all,

Constants
$mempty, \bot, otherwise)$

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

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

Our numeric types use decimal fixed-point arithmetic.

import Data.Fixed *as* X (	
Fixed (..),	Type constructor for numbers of given precision
HasResolution (..))	Type class for specifying precisions

We rely on the `lens` library for accessing nested values. There is no need to understand the theory behind lenses to understand this program. The notation $a\ .\ b\ .\ c$ 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,
  makeLenses,       Defines lenses for record fields
  makeFields,       Defines lenses for record fields
  set,              Writes a lens
  use, preuse,
  Zoom (..),
  view, preview,    Reads a lens in a do block
  (&~),             Lets us use a do block with setters ◊ Get rid of this.
  ix,               Lens for map retrieval and updating
  at,               Lens for map insertion
 Operators for partial state updates in do blocks:
  (:=),             Replace
  (-=), (+=),       Update arithmetically
  (%=),             Update according to function
  (?=))             Insert into map
import Control.Lens.Zoom as X
import Control.Lens.Internal.Zoom as X
```

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

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

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

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

Some less interesting imports are omitted from this document.

[1] Gabriel Gonzalez's 2013 article *Program imperatively using Haskell* is a good introduction.
[2] We assume the axiom that Keccak hash collisions are impossible.

Appendix B

Rounding fixed point numbers

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

```
module Maker.Decimal (Decimal, E18, E36, Epsilon (..)) where

import Data.Fixed

newtype HasResolution e ⇒ Decimal e = D (Fixed e)
  deriving (Ord, Eq, Real, RealFrac)
```

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

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

In the Num instance, we delegate everything except multiplication.

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

$signum\ (\mathrm{D}\ a) = \mathrm{D}\ (signum\ a)$
$fromInteger\ i = \mathrm{D}\ (fromInteger\ i)$

In the Fractional instance, we delegate everything except division.

instance HasResolution $e \Rightarrow$ Fractional (Decimal e) **where**
$x@(\mathrm{D}\ (\mathrm{MkFixed}\ a))\ /\ \mathrm{D}\ (\mathrm{MkFixed}\ b) =$
$\mathrm{D}\ (\mathrm{MkFixed}\ (div\ (a * resolution\ x + div\ b\ 2)\ b))$
$recip\ (\mathrm{D}\ a) \quad = \mathrm{D}\ (recip\ a)$
$fromRational\ r = \mathrm{D}\ (fromRational\ r)$

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

data E18; **data** E36
instance HasResolution E18 **where**
$resolution\ \_ = 10 \uparrow (18 :: \mathrm{Integer})$
instance HasResolution E36 **where**
$resolution\ \_ = 10 \uparrow (36 :: \mathrm{Integer})$

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

class Epsilon t **where** $\epsilon :: t$
instance HasResolution $a \Rightarrow$ Epsilon (Decimal a) **where**
The use of $\perp$ is safe since *resolution* ignores the value.
$\epsilon = 1\ /\ fromIntegral\ (resolution\ (\perp :: \mathrm{Fixed}\ a))$