MAKER

presents the

REFERENCE
IMPLEMENTATION

also known as the
PURPLE PAPER

of the remarkable

DAI CREDIT SYSTEM

issuing a diversely collateralized stablecoin

formulated by

Daniel Brockman
Mikael Brockman
Nikolai Mushegian

with last update on March 16, 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 credit incentives in order to keep its market value stable relative to SDR[1] in the short and medium term.

New dai enters the money supply when a borrower locks an excess of collateral in the system and takes out a loan. 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—until risk provokes a liquidation.

Off-chain *price feeds* give Maker knowledge of the market values of dai and the various tokens used as collateral, enabling the system to assess credit risk. If the value of a CDP's collateral drops below a certain multiple of its debt, a decentralized auction is triggered which liquidates the collateral for dai in order to settle the debt.

The system issues a separate token with symbol MKR. Since collateral auctions may fail to recover the full value of liquidated debt, the MKR token can be diluted to back emergency debt. The value of MKR, though volatile by design, is backed by the revenue from *stability fees* imposed on all dai loans. The DAI raised from stability fees is used to buy MKR tokens from the market and destroy them.

For more details on the economics of the system, as well as descriptions of governance, off-chain mechanisms that provide efficiency, and so on, see the whitepaper.

This document is an executable technical specification of the of the Maker smart contracts. It is a draft; be aware that the contents will certainly change before launch.

[1] "Special Drawing Rights" (ticker symbol XDR), the international reserve asset created by the International Monetary Fund, whose value is derived from a weighted basket of world currencies.

1.1 Motivation

The version of this system that will be deployed on the blockchain is written in Solidity, which is a workable smart contract implementation language. This reference implementation is a 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 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. **Clarity**. An implementation not intended to be deployed on the blockchain is free from concerns about optimizing for gas cost and other factors that make the Solidity implementation less ideal as an understandable specification.

7. **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 model is limited in that it has

1. a simplified version of authorization for governance;
2. a simplified version of `ERC20` token semantics;
3. no implementation of the decentralized auction contracts; and
4. no 256-bit word limits.

These limitations will be addressed in future revisions.

1.3 Verification

Separately from this document, we are developing automatic test suites that generate many, large, and diverse action sequences for property verification. One such property is that the reference implementation exactly matches the on-chain implementation; this is verified through the generation of Solidity test cases with assertions covering the entire state. Other key properties include

- that the target price changes only according to the target rate;
- that the total dai supply is fully accounted for by CDP debts;
- that CDP acts are restricted with respect to risk stage;

along with similar invariants and conditions. A future revision of this document will include formal statements of these properties.

Part I

Implementation

Preamble

This is a Haskell program, and as such makes reference to a background of symbols defined in libraries, as a mathematical paper depends on preestablished theories.

Context should allow the reader to understand most symbols without further reading, but Appendix A lists and briefly explains each imported type and function.

We replace the default prelude module with our own.

```
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
```

Now we proceed to define the specifics of the Maker system.

Chapter 2

Types

This chapter defines the data types used by Maker: numeric types, identifiers, on-chain records, and test model data.

Haskell syntax note: **newtype** defines a type synonym with distinct type identity; **data** creates a record type; and **deriving** creates automatic instances of common functionality.

2.1 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 of currency quantities
newtype Wad = Wad (Decimal E18)
  deriving (Ord, Eq, Num, Real, Fractional, RealFrac)

-- Define the distinct type of rates and ratios
newtype Ray = Ray (Decimal E36)
  deriving (Ord, Eq, Num, Real, Fractional, RealFrac)
```

We also define a type for time durations in whole seconds, as this is the maximum precision allowed by the Ethereum virtual machine.

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

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

Convert via fractional n/m form.

$$cast :: (\mathrm{Real}\ a, \mathrm{Fractional}\ b) \Rightarrow a \to b$$
$$cast = fromRational\ .\ toRational$$

2.2 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; e.g., Id Foo and Id Bar are incompatible.

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

We define another type for representing Ethereum account addresses.

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

We also have predefined entity identifiers.

The DAI token identifier

$$id_{\mathrm{DAI}} = \mathrm{Id}\ \texttt{"DAI"}$$

The internal debt token identifier

$$id_{\texttt{sin}} = \mathrm{Id}\ \texttt{"SIN"}$$

A test account with ultimate authority

$$id_{god} = \mathrm{Address}\ \texttt{"GOD"}$$

2.3 `Gem` — collateral price feed entry

The data received from price feeds is categorized by token and stored in `Gem` records. Our model also has the token balances embedded in these records; in reality[1], the balances are in separate `ERC20` contracts.

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

[1] We use “reality” to denote the actual state of the consensus Ethereum blockchain.

- `erc20` :: `ERC20`, Token balances
- `tag` :: `Wad`, Market price denominated in SDR
- `zzz` :: `Sec` Time of price expiration

} **deriving** (Eq, Show)

2.4 `ERC20` — token model

In reality, token semantics can differ, despite nominally following the `ERC20` interface. Governance therefore involves reviewing the behaviors of collateral tokens. In our model, tokens behave in the same simple way. We also omit the notion of "allowance."

Tokens can be held by CDP owners, by token vaults, or by the test driver. We model this distinction with a data type.

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

We now define an `ERC20` instance as a map tracking the token quantity held by each holder.

data `ERC20` = `ERC20` { · `balanceOf` :: Map Holder `Wad`}
  deriving (Eq, Show)

2.5 `Ilk` — CDP type

Each CDP belongs to a CDP type, specified by an `Ilk` record. Five parameters, `mat`, `axe`, `hat`, `tax` and `lax`, are set by governance and are known as the Risk Parameters. The rest of the values are used by the system to keep track of the current state. The meaning of each `ilk` parameter is defined by its interactions in the act definitions of Chapter 3; see the whitepaper for an overview.

data `Ilk` = `Ilk` {

- `jar` :: Id `Gem`, Collateral token identifier
- `tax` :: `Sec`, Grace period after price feed becomes unavailable
- `mat` :: `Ray`, Collateral-to-debt ratio at which liquidation can be triggered
- `axe` :: `Ray`, Penalty on liquidation as fraction of debt
- `hat` :: `Wad`, Limit on total debt for CDP type ("debt ceiling")

- `tax` :: `Ray`, Stability fee as per-second fraction of debt value
- `chi` :: `Ray`, Value of internal debt unit in dai
- `rho` :: `Sec`, Time of latest debt unit adjustment
- `rum` :: `Wad` Total debt in debt units

} **deriving** (Eq, Show)

2.6 `Urn` — collateralized debt position (CDP)

For each CDP we maintain an `Urn` record identifying its type and specifying ownership, quantities of debt and collateral denominated in the CDP type's debt unit, along with who triggered liquidation (if applicable).

data `Urn` = `Urn` {

- `ilk` :: Id `Ilk`, Identifier of CDP type
- `lad` :: Address, Owner of CDP
- `art` :: `Wad`, Outstanding debt in debt unit
- `jam` :: `Wad`, Collateral amount in debt unit
- `cat` :: Maybe Address Address that triggered liquidation, if applicable

} **deriving** (Eq, Show)

2.7 `Vox` — feedback mechanism data

The *feedback mechanism* is the aspect of the CDP engine that adjusts the target price of dai based on market price, and its data is kept in a singleton record called `Vox`.

data `Vox` = `Vox` {

- `wut` :: `Wad`, Market price of dai denominated in SDR
- `par` :: `Wad`, Target price of dai denominated in SDR
- `way` :: `Ray`, Current per-second change in target price
- `how` :: `Ray`, Sensitivity parameter set by governance
- `tau` :: `Sec` Time of latest feedback cycle

} **deriving** (Eq, Show)

Keeping the feedback data separate allows us to more easily upgrade the mechanism in the future.

2.8 `Vat` — CDP engine aggregate

The `Vat` record aggregates the records of tokens, CDPs, CDP types, and price feeds, along with the data of the feedback mechanism.

```
data Vat = Vat {
  · gems :: Map (Id Gem) Gem,   Price feed data
  · ilks :: Map (Id Ilk) Ilk,   CDP type records
  · urns :: Map (Id Urn) Urn,   CDP records
  · vox  :: Vox                 Data of feedback mechanism
  } deriving (Eq, Show)
```

2.9 System model

Finally we define a record with no direct counterpart in the Solidity contracts, which has the `Vat` record along with model state.

```
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)
```

2.10 Default data

```
defaultIlk :: Id Gem → Ilk
defaultIlk id_gem = Ilk {
  · jar = id_gem,
  · axe = Ray 1,
  · mat = Ray 1,
  · tax = Ray 1,
  · hat = Wad 0,
  · tax = Sec 0,
  · chi = Ray 1,
```

$\cdot\ \texttt{rum} = \texttt{Wad}\ 0,$
$\cdot\ \texttt{rho} = \texttt{Sec}\ 0$
$\}$

$\mathit{emptyUrn} :: \mathrm{Id}\ \texttt{Ilk} \to \mathrm{Address} \to \texttt{Urn}$
$\mathit{emptyUrn}\ \mathit{id}_{\texttt{ilk}}\ \mathit{id}_{\texttt{lad}} = \texttt{Urn}\ \{$
$\cdot\ \texttt{cat} = \mathrm{Nothing},$
$\cdot\ \texttt{lad} = \mathit{id}_{\texttt{lad}},$
$\cdot\ \texttt{ilk} = \mathit{id}_{\texttt{ilk}},$
$\cdot\ \texttt{art} = \texttt{Wad}\ 0,$
$\cdot\ \texttt{jam} = \texttt{Wad}\ 0$
$\}$

$\mathit{initialVat} :: \texttt{Ray} \to \texttt{Vat}$
$\mathit{initialVat}\ \texttt{how}_0 = \texttt{Vat}\ \{$
$\cdot\ \texttt{vox} = \texttt{Vox}\ \{$
$\cdot\ \texttt{tau} = 0,$
$\cdot\ \texttt{wut} = \texttt{Wad}\ 1,$
$\cdot\ \texttt{par} = \texttt{Wad}\ 1,$
$\cdot\ \texttt{how} = \texttt{how}_0,$
$\cdot\ \texttt{way} = \texttt{Ray}\ 1$
$\},$
$\cdot\ \texttt{ilks} = \varnothing,$
$\cdot\ \texttt{urns} = \varnothing,$
$\cdot\ \texttt{gems} = \mathit{singleton}\ \mathit{id}_{\mathrm{DAI}}\ \texttt{Gem}\ \{$
$\cdot\ \texttt{erc20} = \texttt{ERC20}\ \{$
$\cdot\ \texttt{balanceOf} = \mathit{fromList}\ [(\mathrm{InVault}\ \mathit{id}_{\mathrm{DAI}}, 0),$
$(\mathrm{InVault}\ \mathit{id}_{\texttt{sin}}, 0)]$
$\},$
$\cdot\ \texttt{tag} = \texttt{Wad}\ 0,$
$\cdot\ \texttt{zzz} = 0$
$\}$
$\}$

$\mathit{initialSystem} :: \texttt{Ray} \to \mathrm{System}$
$\mathit{initialSystem}\ \texttt{how}_0 = \mathrm{System}\ \{$
$\cdot\ \texttt{vat} = \mathit{initialVat}\ \texttt{how}_0,$
$\cdot\ \texttt{era} = 0,$
$\cdot\ \texttt{sender} = \mathit{id}_{\mathit{god}},$
$\cdot\ \texttt{accounts} = \mathit{mempty}$
$\}$

Chapter 3

Acts

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

Unless specified as *internal*, acts are accessible as public functions on the blockchain.

The `auth` modifier marks acts which can only be invoked from addresses to which the system has granted authority.

For details on the underlying “Maker monad,” which specifies how the act definitions behave with regard to state and rollback, see chapter 4.

3.1 Assessment

In order to prohibit CDP acts based on risk situation, we define five stages of risk.

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

We define the function *analyze* that determines the risk stage of a CDP.

```
analyze era0 par0 urn0 ilk0 jar0 =
  if | view cat urn0 ≢ Nothing ∧ view jam urn0 ≡ 0
         CDP liquidation triggered and started
           → Dread
     | view cat urn0 ≢ Nothing
         CDP liquidation triggered
           → Grief
     | pro < min
         CDP's collateralization below liquidation ratio
           → Panic
     | view zzz jar0 + view lax ilk0 < era0
         CDP type's price limbo exceeded limit
           → Panic
     | view zzz jar0 < era0
         CDP type's price feed in limbo
           → Worry
     | cap > view hat ilk0
         CDP type's debt ceiling exceeded
           → Anger
     | otherwise
         No problems
           → Pride
  where
   CDP's collateral value in SDR:
     pro = view jam urn0 * view tag jar0
   CDP type's total debt in SDR:
     cap = view rum ilk0 * cast (view chi ilk0)
   CDP's debt in SDR:
     con = view art urn0 * cast (view chi ilk0) * par0
   Required collateral as per liquidation ratio:
     min = con * cast (view mat ilk0)
```

Table 3.1: CDP acts in the five stages of risk

	`give`	`shut`	`lock`	`wipe`	`free`	`draw`	`bite`	`grab`	`plop`
`Pride`	✌	🤞	🤞	🤞	☝	☝	—	—	—
`Anger`	✌	🤞	🤞	🤞	☝	—	—	—	—
`Worry`	✌	🤞	🤞	🤞	—	—	—	—	—
`Panic`	✌	🤞	🤞	🤞	—	—	✋	—	—
`Grief`	✌	—	—	—	—	—	—	✊	—
`Dread`	✌	—	—	—	—	—	—	—	✊
			decrease risk		increase risk			unwind risk	

✋	allowed for anyone
✌	allowed for owner unconditionally
🤞	allowed for owner if able to repay
☝	allowed for owner if collateralized
✊	allowed for settler contract

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

$$
\begin{array}{l}
\texttt{feel}\ id_{\texttt{urn}} = \textbf{do} \\
\quad \text{Adjust target price and target rate} \\
\quad\quad \texttt{prod} \\
\quad \text{Update debt unit and unprocessed fee revenue} \\
\quad\quad id_{\texttt{ilk}} \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \mathit{ix}\ id_{\texttt{urn}}\ .\ \texttt{ilk}) \\
\quad\quad \texttt{drip}\ id_{\texttt{ilk}} \\
\quad \text{Read parameters for risk analysis} \\
\quad\quad \texttt{era}_0 \leftarrow \mathit{use}\ \ \texttt{era} \\
\quad\quad \texttt{par}_0 \leftarrow \mathit{use}\ \ (\texttt{vat}\ .\ \texttt{vox}\ .\ \texttt{par}) \\
\quad\quad \texttt{urn}_0 \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \mathit{ix}\ id_{\texttt{urn}}) \\
\quad\quad \texttt{ilk}_0 \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ \mathit{ix}\ (\mathit{view}\ \texttt{ilk}\ \texttt{urn}_0)) \\
\quad\quad \texttt{jar}_0 \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{gems}\ .\ \mathit{ix}\ (\mathit{view}\ \texttt{jar}\ \texttt{ilk}_0)) \\
\quad \text{Return risk stage of CDP} \\
\quad\quad \mathit{return}\ (\mathit{analyze}\ \texttt{era}_0\ \texttt{par}_0\ \texttt{urn}_0\ \texttt{ilk}_0\ \texttt{jar}_0)
\end{array}
$$

Acts on CDPs use `feel` to prohibit increasing risk when already risky, and to freeze debt and collateral during liquidation; see Table 3.1.

3.2 Lending

Any user can open one or more accounts with the system using open, specifying a self-chosen account identifier and a CDP type.

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

Fail if account identifier is taken

$\quad none\ (\texttt{vat}\ .\ \texttt{urns}\ .\ ix\ id_{\texttt{urn}})$

Create a CDP record with the sender as owner

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

$\quad initialize\ (\texttt{vat}\ .\ \texttt{urns}\ .\ at\ id_{\texttt{urn}})\ (emptyUrn\ id_{\texttt{ilk}}\ id_{\texttt{lad}})$

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

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

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

Transfer ownership

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

Unless liquidation has been triggered for a CDP, its owner can use `lock` to deposit more collateral.

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

Fail if sender is not the CDP owner

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

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

Fail if liquidation triggered or initiated

$\quad want\ (\texttt{feel}\ id_{\texttt{urn}})\ (\notin [\texttt{Grief}, \texttt{Dread}])$

Identify collateral type

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

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

Transfer tokens from owner to collateral vault

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

Record an increase in collateral

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

When a CDP has no risk problems (except that its CDP type's debt ceiling may be exceeded), its owner can use `free` to withdraw some amount of collateral, as long as the withdrawal would not reduce collateralization below the liquidation ratio.

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

Fail if sender is not the CDP owner

$\mathit{id}_{\texttt{lad}} \leftarrow \mathit{use\ sender}$
$\mathit{owns}\ \mathit{id}_{\texttt{urn}}\ \mathit{id}_{\texttt{lad}}$

Record a decrease in collateral

$\mathit{decrease}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{urn}}\ .\ \texttt{jam})\ \texttt{wad}_{\texttt{gem}}$

Roll back on any risk problem except debt ceiling excess

$\mathit{want}\ (\texttt{feel}\ \mathit{id}_{\texttt{urn}})\ (\in [\texttt{Pride}, \texttt{Anger}])$

Transfer tokens from collateral vault to owner

$\mathit{id}_{\texttt{ilk}} \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{urn}}\ .\ \texttt{ilk})$
$\mathit{id}_{\texttt{gem}} \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{ilk}}\ .\ \texttt{jar})$
$\texttt{push}\ \mathit{id}_{\texttt{gem}}\ \mathit{id}_{\texttt{lad}}\ \texttt{wad}_{\texttt{gem}}$

When a CDP has no risk problems, its owner can can use `draw` to take out a loan of newly minted dai, as long as the CDP type's debt ceiling is not reached and the loan would not result in undercollateralization.

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

Fail if sender is not the CDP owner

$\mathit{id}_{\texttt{lad}} \leftarrow \mathit{use\ sender}$
$\mathit{owns}\ \mathit{id}_{\texttt{urn}}\ \mathit{id}_{\texttt{lad}}$

Update debt unit and unprocessed fee revenue

$\mathit{id}_{\texttt{ilk}} \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{urn}}\ .\ \texttt{ilk})$
$\texttt{chi}_1 \leftarrow \texttt{drip}\ \mathit{id}_{\texttt{ilk}}$

Denominate loan in debt unit

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

Increase CDP debt

$\mathit{increase}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{urn}}\ .\ \texttt{art})\ \texttt{wad}_{\texttt{chi}}$

Increase total debt of CDP type

$\mathit{increase}\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{ilk}}\ .\ \texttt{rum})\ \texttt{wad}_{\texttt{chi}}$

Roll back on any risk problem

$\mathit{want}\ (\texttt{feel}\ \mathit{id}_{\texttt{urn}})\ (\equiv \texttt{Pride})$

Mint both dai and debt tokens

$\texttt{lend}\ \texttt{wad}_{\text{DAI}}$

Transfer dai to CDP owner

$\texttt{push}\ \mathit{id}_{\text{DAI}}\ \mathit{id}_{\texttt{lad}}\ \texttt{wad}_{\text{DAI}}$

A CDP owner who has previously loaned dai can use `wipe` to repay part of their debt as long as liquidation has not been triggered.

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

Fail if sender is not the CDP owner

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

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

Fail if liquidation triggered or initiated

$\textit{want}\ (\texttt{feel}\ id_{\texttt{urn}})\ (\notin [\texttt{Grief}, \texttt{Dread}])$

Update debt unit and unprocessed fee revenue

$id_{\texttt{ilk}} \leftarrow \textit{look}\ (\texttt{vat} \mathbin{.} \texttt{urns} \mathbin{.} \textit{ix}\ id_{\texttt{urn}} \mathbin{.} \texttt{ilk})$

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

Denominate dai amount in debt unit

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

Decrease CDP debt

$\textit{decrease}\ (\texttt{vat} \mathbin{.} \texttt{urns} \mathbin{.} \textit{ix}\ id_{\texttt{urn}} \mathbin{.} \texttt{art})\ \texttt{wad}_{\texttt{chi}}$

Decrease total CDP type debt

$\textit{decrease}\ (\texttt{vat} \mathbin{.} \texttt{ilks} \mathbin{.} \textit{ix}\ id_{\texttt{ilk}} \mathbin{.} \texttt{rum})\ \texttt{wad}_{\texttt{chi}}$

Transfer dai from CDP owner to dai vault

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

Destroy dai and corresponding debt tokens

$\texttt{mend}\ \texttt{wad}_{\text{DAI}}$

A CDP owner can use `shut` to close their account—repaying all debt and reclaiming all collateral—if the price feed is up to date and liquidation has not been initiated.

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

Update debt unit and unprocessed fee revenue

$id_{\texttt{ilk}} \leftarrow \textit{look}\ (\texttt{vat} \mathbin{.} \texttt{urns} \mathbin{.} \textit{ix}\ id_{\texttt{urn}} \mathbin{.} \texttt{ilk})$

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

Reclaim all outstanding dai

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

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

Reclaim all collateral

$\texttt{jam}_0 \leftarrow \textit{look}\ (\texttt{vat} \mathbin{.} \texttt{urns} \mathbin{.} \textit{ix}\ id_{\texttt{urn}} \mathbin{.} \texttt{jam})$

$\texttt{free}\ id_{\texttt{urn}}\ \texttt{jam}_0$

Nullify CDP record

$\texttt{vat} \mathbin{.} \texttt{urns} \mathbin{.} \textit{at}\ id_{\texttt{urn}} := \text{Nothing}$

3.3 Adjustment

The feedback mechanism is updated through `prod`, which can be invoked at any time by keepers, but is also invoked as a side effect of any CDP act that uses `feel` to assess the CDP risk.

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

Read all parameters relevant for feedback mechanism

$\texttt{era}_0 \leftarrow \textit{use}\ \texttt{era}$
$\texttt{tau}_0 \leftarrow \textit{use}\ (\texttt{vat}\ .\ \texttt{vox}\ .\ \texttt{tau})$
$\texttt{wut}_0 \leftarrow \textit{use}\ (\texttt{vat}\ .\ \texttt{vox}\ .\ \texttt{wut})$
$\texttt{par}_0 \leftarrow \textit{use}\ (\texttt{vat}\ .\ \texttt{vox}\ .\ \texttt{par})$
$\texttt{how}_0 \leftarrow \textit{use}\ (\texttt{vat}\ .\ \texttt{vox}\ .\ \texttt{how})$
$\texttt{way}_0 \leftarrow \textit{use}\ (\texttt{vat}\ .\ \texttt{vox}\ .\ \texttt{way})$

$\textbf{let}$

Time difference in seconds

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

Current target rate applied to target price

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

Sensitivity parameter applied over time

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

Target rate scaled up or down

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

Update target price

$\texttt{vat}\ .\ \texttt{vox}\ .\ \texttt{par} := \texttt{par}_1$

Update rate of price change

$\texttt{vat}\ .\ \texttt{vox}\ .\ \texttt{way} := \texttt{way}_1$

Record time of update

$\texttt{vat}\ .\ \texttt{vox}\ .\ \texttt{tau} := \texttt{era}_0$

$\textbf{where}$

Convert between multiplicative and additive form

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

The stability fee of a CDP type can change through governance. Due to the constraint that acts should run in constant time, the system cannot iterate over CDP records to effect such changes. Instead each CDP type has a single "debt unit" which accumulates the stability fee. The `drip` act updates this unit. It can be called at any time by keepers, but is also called as a side effect of every act that uses `feel` to assess CDP risk.

$\texttt{drip}\ id_{\texttt{ilk}} = \textbf{do}$

Time stamp of previous `drip`

$\texttt{rho}_0 \leftarrow look\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ ix\ id_{\texttt{ilk}}\ .\ \texttt{rho})$

Current stability fee

$\texttt{tax}_0 \leftarrow look\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ ix\ id_{\texttt{ilk}}\ .\ \texttt{tax})$

Current debt unit value

$\texttt{chi}_0 \leftarrow look\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ ix\ id_{\texttt{ilk}}\ .\ \texttt{chi})$

Current total debt in debt unit

$\texttt{rum}_0 \leftarrow look\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ ix\ id_{\texttt{ilk}}\ .\ \texttt{rum})$

Current time stamp

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

let

Time difference in seconds

$age\ \ = \texttt{era}_0 - \texttt{rho}_0$

Value of debt unit increased according to stability fee

$\texttt{chi}_1 = \texttt{chi}_0 * \texttt{tax}_0 \uparrow\uparrow age$

Stability fee revenue denominated in new unit

$dew\ \ = (cast\ (\texttt{chi}_1 - \texttt{chi}_0) :: \texttt{Wad}) * \texttt{rum}_0$

Mint dai and internal debt tokens for marginal stability fee

$\texttt{lend}\ dew$

Record time of update

$\texttt{vat}\ .\ \texttt{ilks}\ .\ ix\ id_{\texttt{ilk}}\ .\ \texttt{rho} := \texttt{era}_0$

Record new debt unit

$\texttt{vat}\ .\ \texttt{ilks}\ .\ ix\ id_{\texttt{ilk}}\ .\ \texttt{chi} := \texttt{chi}_1$

Return the new debt unit

$return\ \texttt{chi}_1$

3.4 Price feed input

The `mark` act records a new market price of a collateral token along with the expiration date of this price.

$$
\begin{array}{l}
\texttt{mark}\ \mathit{id}_{\texttt{gem}}\ \texttt{tag}_1\ \texttt{zzz}_1 = \texttt{auth}\ \$\ \textbf{do} \\
\quad \texttt{vat} \,.\, \texttt{gems} \,.\, \mathit{ix}\ \mathit{id}_{\texttt{gem}} \,.\, \texttt{tag} := \texttt{tag}_1 \\
\quad \texttt{vat} \,.\, \texttt{gems} \,.\, \mathit{ix}\ \mathit{id}_{\texttt{gem}} \,.\, \texttt{zzz} := \texttt{zzz}_1
\end{array}
$$

The `tell` act records a new market price of the DAI token along with the expiration date of this price.

$$
\texttt{tell}\ \texttt{wad}_{\texttt{gem}} = \texttt{auth}\ \$\ \textbf{do}\ \texttt{vat} \,.\, \texttt{vox} \,.\, \texttt{wut} := \texttt{wad}_{\texttt{gem}}
$$

3.5 Liquidation

When a CDP's risk stage marks it as in need of liquidation, any account can invoke the `bite` act to trigger the liquidation process. This enables the settler contract to grab the collateral for auctioning and take over the debt tokens representing "bad debt."

$$
\texttt{bite}\ \mathit{id}_{\texttt{urn}} = \textbf{do}
$$

Fail if CDP is not in the appropriate risk stage

$$
\mathit{want}\ (\texttt{feel}\ \mathit{id}_{\texttt{urn}})\ (\equiv \texttt{Panic})
$$

Record the sender as the liquidation initiator

$$
\begin{array}{l}
\mathit{id}_{\texttt{cat}} \leftarrow \mathit{use\ sender} \\
\texttt{vat} \,.\, \texttt{urns} \,.\, \mathit{ix}\ \mathit{id}_{\texttt{urn}} \,.\, \texttt{cat} := \text{Just}\ \mathit{id}_{\texttt{cat}}
\end{array}
$$

Apply liquidation penalty to debt

$$
\begin{array}{l}
\mathit{id}_{\texttt{ilk}} \leftarrow \mathit{look}\ (\texttt{vat} \,.\, \texttt{urns} \,.\, \mathit{ix}\ \mathit{id}_{\texttt{urn}} \,.\, \texttt{ilk}) \\
\texttt{axe}_0 \leftarrow \mathit{look}\ (\texttt{vat} \,.\, \texttt{ilks} \,.\, \mathit{ix}\ \mathit{id}_{\texttt{ilk}} \,.\, \texttt{axe}) \\
\texttt{art}_0 \leftarrow \mathit{look}\ (\texttt{vat} \,.\, \texttt{urns} \,.\, \mathit{ix}\ \mathit{id}_{\texttt{urn}} \,.\, \texttt{art}) \\
\textbf{let}\ \texttt{art}_1 = \texttt{art}_0 * \mathit{cast}\ \texttt{axe}_0
\end{array}
$$

Update debt

$$
\texttt{vat} \,.\, \texttt{urns} \,.\, \mathit{ix}\ \mathit{id}_{\texttt{urn}} \,.\, \texttt{art} := \texttt{art}_1
$$

After liquidation has been triggered, the designated settler contract invokes grab to receive both the CDP's collateral tokens and the internal debt tokens corresponding to the CDP's debt.

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

Fail if CDP is not marked for liquidation

$\quad want\ (\texttt{feel}\ id_{\texttt{urn}})\ (\equiv \texttt{Grief})$

Transfer the collateral to the settler

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

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

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

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

$\quad \texttt{push}\ id_{\texttt{gem}}\ id_{\texttt{vow}}\ \texttt{jam}_0$

Update the debt unit and stability fee

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

Denominate the debt in dai

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

$\quad \textbf{let}\ \texttt{con} = \texttt{art}_0 * cast\ \texttt{chi}_1$

Transfer the debt tokens to the settler

$\quad \texttt{push}\ id_{\text{DAI}}\ id_{\texttt{vow}}\ \texttt{con}$

When the settler has finished the process of liquidating a CDP's collateral, it invokes plop on the CDP to give back any excess collateral gains.

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

Fail unless CDP is in liquidation

$\quad want\ (\texttt{feel}\ id_{\texttt{urn}})\ (\equiv \texttt{Dread})$

Forget the CDP's requester of liquidation

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

Return some amount of excess auction gains

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

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

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

$\quad \texttt{pull}\ id_{\texttt{gem}}\ id_{\texttt{vow}}\ \texttt{wad}_{\text{DAI}}$

Record the gains as the CDP's collateral

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

The settler can invoke `loot` at any time to claim all uncollected stability fee revenue (for use in the MKR buy and burn auction).

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

The dai vault's balance is the uncollected stability fee revenue

$\texttt{wad}_{\text{DAI}} \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{gems}\ .\ \mathit{ix}\ \mathit{id}_{\text{DAI}}\ .\ \text{ERC20}\ .\ \mathit{balanceOf}\ .\ \mathit{ix}\ (\text{InVault}\ \mathit{id}_{\text{DAI}}))$

Transfer the entire dai vault balance to sender

$\mathit{id}_{\texttt{vow}} \leftarrow \mathit{use}\ \mathit{sender}$

$\texttt{push}\ \mathit{id}_{\text{DAI}}\ \mathit{id}_{\texttt{vow}}\ \texttt{wad}_{\text{DAI}}$

3.6 Governance

Governance uses `form` to create a new CDP type. Since the new type is initialized with a zero debt ceiling, a separate transaction can safely set the risk parameters before any lending occurs.

$$\begin{aligned}&\texttt{form}\ \mathit{id}_{\texttt{ilk}}\ \mathit{id}_{\texttt{gem}} = \texttt{auth}\ \$\ \textbf{do}\\&\quad \mathit{initialize}\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ \mathit{at}\ \mathit{id}_{\texttt{ilk}})\ (\mathit{defaultIlk}\ \mathit{id}_{\texttt{gem}})\end{aligned}$$

Governance uses `frob` to alter the sensitivity factor, which is the only mutable parameter of the feedback mechanism.

$$\texttt{frob}\ \texttt{how}_1 = \texttt{auth}\ \$\ \textbf{do}\ \texttt{vat}\ .\ \texttt{vox}\ .\ \texttt{how} := \texttt{how}_1$$

Governance can alter the five risk parameters of a CDP type using `cuff` for the liquidation ratio; `chop` for the liquidation penalty; `cork` for the debt ceiling; `calm` for the duration of price limbo; and `crop` for the stability fee.

$$\begin{aligned}&\texttt{cuff}\ \mathit{id}_{\texttt{ilk}}\ \texttt{mat}_1 = \texttt{auth}\ \$\ \textbf{do}\ \texttt{vat}\ .\ \texttt{ilks}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{ilk}}\ .\ \texttt{mat} := \texttt{mat}_1\\&\texttt{chop}\ \mathit{id}_{\texttt{ilk}}\ \texttt{axe}_1 = \texttt{auth}\ \$\ \textbf{do}\ \texttt{vat}\ .\ \texttt{ilks}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{ilk}}\ .\ \texttt{axe} := \texttt{axe}_1\\&\texttt{cork}\ \mathit{id}_{\texttt{ilk}}\ \texttt{hat}_1 = \texttt{auth}\ \$\ \textbf{do}\ \texttt{vat}\ .\ \texttt{ilks}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{ilk}}\ .\ \texttt{hat} := \texttt{hat}_1\\&\texttt{calm}\ \mathit{id}_{\texttt{ilk}}\ \mathit{lax1} = \texttt{auth}\ \$\ \textbf{do}\ \texttt{vat}\ .\ \texttt{ilks}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{ilk}}\ .\ \texttt{lax} := \mathit{lax1}\end{aligned}$$

When altering the stability fee with `crop`, we ensure that the previous stability fee has been accounted for in the internal debt unit.

$\texttt{crop}\ \mathit{id}_{\texttt{ilk}}\ \texttt{tax}_1 =$

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

Apply the current stability fee to the internal debt unit

$\texttt{drip}\ \mathit{id}_{\texttt{ilk}}$

Change the stability fee

$\texttt{vat}\ .\ \texttt{ilks}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{ilk}}\ .\ \texttt{tax} := \texttt{tax}_1$

3.7 Vaults

The internal act `pull` transfers tokens into a vault. It is used by `lock` to acquire collateral from a CDP owner; by `wipe` to acquire dai from a CDP owner; and by `plop` to acquire collateral from the settler contract.

$$
\begin{array}{l}
\texttt{pull}\ id_{\texttt{gem}}\ id_{\texttt{lad}}\ \texttt{wad}_{\texttt{gem}} = \\
\quad \textit{transfer}\ id_{\texttt{gem}}\ \texttt{wad}_{\texttt{gem}}\ (\text{InAccount}\ id_{\texttt{lad}})\ (\text{InVault}\ id_{\texttt{gem}})
\end{array}
$$

The internal act `push` transfers tokens out from a collateral vault. It is used by `draw` to send dai to a CDP owner; by `free` to send collateral to a CDP owner; and by `grab` to send collateral to the settler contract.

$$
\begin{array}{l}
\texttt{push}\ id_{\texttt{gem}}\ id_{\texttt{lad}}\ \texttt{wad}_{\texttt{gem}} = \\
\quad \textit{transfer}\ id_{\texttt{gem}}\ \texttt{wad}_{\texttt{gem}}\ (\text{InVault}\ id_{\texttt{gem}})\ (\text{InAccount}\ id_{\texttt{lad}})
\end{array}
$$

3.8 Token manipulation

We model the `ERC20` transfer function in simplified form (omitting the concept of "allowance").

$$
\begin{array}{l}
\textit{transfer}\ id_{\texttt{gem}}\ \texttt{wad}\ \textit{src}\ \textit{dst} = \\
\quad \text{Operate in the token's balance table} \\
\quad \textit{zoom}\ (\texttt{vat}\ .\ \texttt{gems}\ .\ \textit{ix}\ id_{\texttt{gem}}\ .\ \texttt{ERC20}\ .\ \textit{balanceOf})\ \$\ \textbf{do} \\
\qquad \text{Fail if source balance insufficient} \\
\qquad \textit{balance} \leftarrow \textit{look}\ (\textit{ix}\ \textit{src}) \\
\qquad \texttt{aver}\ (\textit{balance} \geqslant \texttt{wad}) \\
\qquad \text{Update balances} \\
\qquad \textit{decrease}\ \ (\textit{ix}\ \textit{src})\ \texttt{wad} \\
\qquad \textit{initialize}\ (\textit{at}\ \textit{dst})\ 0 \\
\qquad \textit{increase}\ \ (\textit{ix}\ \textit{dst})\ \texttt{wad}
\end{array}
$$

The internal act `mint` inflates the supply of a token. It is used by `lend` to create new DAI and debt tokens, and by the settler to create new MKR.

$$
\begin{array}{l}
\texttt{mint}\ id_{\texttt{gem}}\ \texttt{wad}_0 = \\
\quad \textit{zoom}\ (\texttt{vat}\ .\ \texttt{gems}\ .\ \textit{ix}\ id_{\texttt{gem}}\ .\ \texttt{ERC20})\ \$\ \textbf{do} \\
\qquad \textit{increase}\ (\textit{balanceOf}\ .\ \textit{ix}\ (\text{InVault}\ id_{\texttt{gem}}))\ \texttt{wad}_0
\end{array}
$$

The internal act `burn` deflates the supply of a token. It is used by `mend` to destroy DAI and debt tokens, and by the settler to destroy MKR.

$$\begin{aligned}&\texttt{burn}\ id_{\texttt{gem}}\ \texttt{wad}_0 = \\ &\quad \textit{zoom}\ (\texttt{vat}\ .\ \texttt{gems}\ .\ \textit{ix}\ id_{\texttt{gem}}\ .\ \texttt{ERC20})\ \$\ \textbf{do} \\ &\qquad \textit{decrease}\ (\textit{balanceOf}\ .\ \textit{ix}\ (\text{InVault}\ id_{\texttt{gem}}))\ \texttt{wad}_0\end{aligned}$$

The internal act `lend` mints identical amounts of both dai and the internal debt token. It is used by `draw` to issue dai to a lender; it is also used by `drip` to issue dai representing revenue from stability fees, which stays in the dai vault until collected.

$$\begin{aligned}&\texttt{lend}\ \texttt{wad}_{\text{DAI}} = \texttt{auth}\ \$\ \textbf{do} \\ &\quad \texttt{mint}\ id_{\text{DAI}}\ \texttt{wad}_{\text{DAI}} \\ &\quad \texttt{mint}\ id_{\texttt{sin}}\ \texttt{wad}_{\text{DAI}}\end{aligned}$$

The internal act `mend` destroys identical amounts of both dai and the internal debt token. Its use via `wipe` is how the dai supply is reduced.

$$\begin{aligned}&\texttt{mend}\ \texttt{wad}_{\text{DAI}} = \texttt{auth}\ \$\ \textbf{do} \\ &\quad \texttt{burn}\ id_{\text{DAI}}\ \texttt{wad}_{\text{DAI}} \\ &\quad \texttt{burn}\ id_{\texttt{sin}}\ \texttt{wad}_{\text{DAI}}\end{aligned}$$

Chapter 4

Act framework

The reader does not need any abstract understanding of monads to understand the code. They give us a nice syntax—the **do** block notation—for expressing exceptions and state in a way that is still purely functional. Each line of such a block is interpreted by the monad to provide the semantics we want.

4.1 The Maker monad

This defines the Maker monad as a simple composition of a state monad and an error monad:

$$\textbf{type}\ \text{Maker}\ a = \text{StateT System (Except Error)}\ a$$

We divide act failure modes into general assertion failures and authentication failures.

$$\begin{aligned}&\textbf{data}\ \text{Error} = \text{AssertError Act} \mid \text{AuthError}\\&\quad \textbf{deriving}\ (\text{Show}, \text{Eq})\end{aligned}$$

An act can be executed on a given initial system state using *exec*. The result is either an error or a new state. The *exec* function can also accept a sequence of acts, which will be interpreted as a single transaction.

$$\begin{aligned}&\mathit{exec} :: \text{System} \to \text{Maker}\ () \to \text{Either Error System}\\&\mathit{exec}\ \mathtt{sys}\ m = \mathit{runExcept}\ (\mathit{execStateT}\ m\ \mathtt{sys})\end{aligned}$$

4.2 Asserting

We now define a set of functions that fail unless some condition holds.

General assertion

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

Assert that an indexed value is not present

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

Assert that an indexed value is present

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

Execute an act and assert a condition on its result

$\mathit{want}\ m\ p = m \ggg\!= (\texttt{aver}\ .\ p)$

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

$\mathit{owns}\ id_{\texttt{urn}}\ id_{\texttt{lad}} = \mathbf{do}$
$\quad \mathit{want}\ (\mathit{look}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \mathit{ix}\ id_{\texttt{urn}}\ .\ \texttt{lad}))\ (\equiv id_{\texttt{lad}})$

We define `auth` k as an act modifier that executes k only if the sender is authorized.

$\texttt{auth}\ \mathit{continue} = \mathbf{do}$
$\quad s \leftarrow \mathit{use}\ \mathit{sender}$
$\quad \mathit{unless}\ (s \equiv id_{god})\ (\mathit{throwError}\ \mathrm{AuthError})$
$\quad \mathit{continue}$

Appendix A

Prelude

This module reexports symbols from other packages and exports a few new symbols of its own.

```
module Maker.Prelude (module Maker.Prelude, module X) where
import Prelude as X (
  Conversions to and from strings
    Read (..), Show (..),
  Comparisons
    Eq (..), Ord (..),
  Core abstractions
    Functor     (fmap),
    Applicative (),
    Monad       (return, (>>=)),
  Numeric classes
    Num (..), Integral (), Enum (),
  Numeric conversions
    Real (..), Fractional (..),
    RealFrac (..),
    fromIntegral,
  Simple types
    Integer, Int, String,
  Algebraic types
    Bool  (True, False),
    Maybe (Just, Nothing),
    Either (Right, Left),
```

Functional operators
  $(.), (\$)$,
Numeric operators
  $(+), (-), (*), (/), (\uparrow), (\uparrow\uparrow)$, *div*,
Utilities
  all, $\neg$, *elem*,
Constants
  mempty, $\bot$, *otherwise*)

We use a typical composition of monad transformers from the `mtl` library to structure stateful actions. See section 4.1 (*The Maker monad*).

import Control.Monad.State *as* X (	
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.Writer *as* X (	
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].

[1] Gabriel Gonzalez's 2013 article *Program imperatively using Haskell* is a good introduction.

$\textbf{import}$ Control.Lens as X (

Lens′, *lens*,

makeLenses,	Defines lenses for record fields
makeFields,	Defines lenses for record fields
set,	Writes a lens
use, *preuse*,	Reads a lens from a state value
view,	Reads a lens from a value
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

$\textbf{import}$ Control.Lens.Zoom as X (zoom)

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

$\textbf{import}$ Data.Map as X (

Map,	Type constructor for mappings
$\varnothing$,	Polymorphic empty mapping
singleton,	Creates a mapping with a single key–value pair
fromList)	Creates a mapping with several key–value pairs

Finally we define some of our own convenience functions.

$$
\begin{aligned}
&\mathit{decrease}\ \ a\ x = a \mathrel{-=} x\\
&\mathit{increase}\ \ a\ x = a \mathrel{+=} x\\
&\mathit{initialize}\ a\ x = a \mathrel{\%=} (\lambda\textbf{case}\ \text{Nothing} \rightarrow \text{Just}\ x;\ y \rightarrow y)\\
&\mathit{prepend}\ \ a\ x = a \mathrel{\%=} (x{:})\\
&x \notin xs = \neg\,(\mathit{elem}\ x\ xs)
\end{aligned}
$$

[2]We assume the axiom that Keccak hash collisions are impossible.

Appendix B

Fixed point numbers with rounding

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

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

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

```
instance HasResolution e => Read (Decimal e) where
  readsPrec n s = fmap (\(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)
```

$$
\begin{aligned}
&\mathit{signum}\ (\mathrm{D}\ a) = \mathrm{D}\ (\mathit{signum}\ a)\\
&\mathit{fromInteger}\ i\ \ = \mathrm{D}\ (\mathit{fromInteger}\ i)
\end{aligned}
$$

In the Fractional instance, we delegate everything except division.

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

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

$$
\begin{aligned}
&\textbf{data}\ \mathrm{E18};\ \textbf{data}\ \mathrm{E36}\\
&\textbf{instance}\ \mathrm{HasResolution}\ \mathrm{E18}\ \textbf{where}\\
&\quad \mathit{resolution}\ \_ = 10 \uparrow (18 :: \mathrm{Integer})\\
&\textbf{instance}\ \mathrm{HasResolution}\ \mathrm{E36}\ \textbf{where}\\
&\quad \mathit{resolution}\ \_ = 10 \uparrow (36 :: \mathrm{Integer})
\end{aligned}
$$

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

$$
\begin{aligned}
&\textbf{class}\ \mathrm{Epsilon}\ t\ \textbf{where}\ \epsilon :: t\\
&\textbf{instance}\ \mathrm{HasResolution}\ a \Rightarrow \mathrm{Epsilon}\ (\mathrm{Decimal}\ a)\ \textbf{where}
\end{aligned}
$$

The use of $\perp$ is safe since *resolution* ignores the value.

$$
\quad \epsilon = 1\ /\ \mathit{fromIntegral}\ (\mathit{resolution}\ (\perp :: \mathrm{Fixed}\ a))
$$