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 12, 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 Maker's token vault 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 to liquidate the collateral for dai to be burned thus settling the debt.

The system issues a separate token with symbol MKR, which behaves like a "share" in Maker itself. 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 a *stability fee* imposed on all dai loans and used to buy MKR for burning.

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 derives 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.

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

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.

newtype Id $a =$ Id String **deriving** (Eq, Ord, Show)

We define another type for representing Ethereum account addresses.

newtype Address $=$ Address String **deriving** (Eq, Ord, Show)

We also have two predefined entity identifiers.

The DAI token vault address

$id_{\text{DAI}} =$ Id `"DAI"`

A test account with ultimate authority

$id_{god} =$ Address `"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.

data `Gem` $=$ `Gem` $\{$

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

$\}$ **deriving** (Eq, Show)

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

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 containing parameters for lending: the collateral token, the limit for the debt-to-collateral ratio, the penalty upon forced liquidation, and various other parameters. These are defined by governance. The meaning of each 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
  _lax  :: Sec,       Maximum duration of price feed limbo
  • mat :: Ray,       Collateral-to-debt ratio at which liquidation can be triggered
  • axe :: Ray,       Penalty on liquidation as fraction of collateral value
  • hat :: Wad,       Limit on total debt for CDP type ("debt ceiling")
  • tax :: Ray,       Stability fee as per-second fraction of debt value
  • chi :: Ray,       Dai value of internal debt unit
  • rho :: Sec,       Time of latest debt unit adjustment
  • rum :: Wad        Total debt denominated in debt unit
  } 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 the progress of liquidation (if relevant).

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 relevant
- `vow` :: Maybe Address Address of settling contract, if relevant

} **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` {

- `fix` :: `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 update

} **deriving** (Eq, Show)

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

2.8 `Vat` — CDP engine data

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

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
- `joy` :: `Wad`, Unprocessed stability fees
- `sin` :: `Wad` Bad debt from liquidated CDPs

} **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 $id_{\texttt{gem}}$ = `Ilk` {

- `jar` = $id_{\texttt{gem}}$,
- `axe` = `Ray 1`,
- `mat` = `Ray 1`,
- `tax` = `Ray 1`,
- `hat` = `Wad 0`,

_lax = `Sec 0`,

- `chi` = `Ray 1`,
- `rum` = `Wad 0`,
- `rho` = `Sec 0`

}

$emptyUrn :: \text{Id } \texttt{Ilk} \to \text{Address} \to \texttt{Urn}$
$emptyUrn\ id_{\texttt{ilk}}\ id_{\texttt{lad}} = \texttt{Urn}\ \{$
- $\texttt{vow} = \text{Nothing},$
- $\texttt{cat} = \text{Nothing},$
- $\texttt{lad} = id_{\texttt{lad}},$
- $\texttt{ilk} = id_{\texttt{ilk}},$
- $\texttt{art} = \texttt{Wad}\ 0,$
- $\texttt{jam} = \texttt{Wad}\ 0$

$\}$

$initialVat :: \texttt{Ray} \to \texttt{Vat}$
$initialVat\ \texttt{how}_0 = \texttt{Vat}\ \{$
- $\texttt{vox} = \texttt{Vox}\ \{$
  - $\texttt{tau} = 0,$
  - $\texttt{fix} = \texttt{Wad}\ 1,$
  - $\texttt{par} = \texttt{Wad}\ 1,$
  - $\texttt{how} = \texttt{how}_0,$
  - $\texttt{way} = \texttt{Ray}\ 1$

  $\},$
- $\texttt{joy} = \texttt{Wad}\ 0,$
- $\texttt{sin} = \texttt{Wad}\ 0,$
- $\texttt{ilks} = \varnothing,$
- $\texttt{urns} = \varnothing,$
- $\texttt{gems} = singleton\ id_{\text{DAI}}\ \texttt{Gem}\ \{$
  - $\texttt{erc20} = \texttt{ERC20}\ \{\bullet\ \texttt{balanceOf} = \varnothing\},$
  - $\texttt{tag} = \texttt{Wad}\ 0,$
  - $\texttt{zzz} = 0$

  $\}$

$\}$

$initialSystem :: \texttt{Ray} \to \text{System}$
$initialSystem\ \texttt{how}_0 = \text{System}\ \{$
- $\texttt{vat} = initialVat\ \texttt{how}_0,$
- $\texttt{era} = 0,$
- $\texttt{sender} = id_{god},$
- $\texttt{accounts} = mempty$

$\}$

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.

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

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

$$\begin{aligned}&\mathit{analyze}\ \texttt{era}_0\ \texttt{par}_0\ \texttt{urn}_0\ \texttt{ilk}_0\ \texttt{jar}_0 =\\ &\quad \textbf{if}\ \mid \mathit{view}\ \texttt{vow}\ \texttt{urn}_0 \not\equiv \text{Nothing}\\ &\qquad \text{CDP liquidation in progress}\\ &\qquad\quad \to \texttt{Dread}\\ &\qquad \mid \mathit{view}\ \texttt{cat}\ \texttt{urn}_0 \not\equiv \text{Nothing}\\ &\qquad\quad \text{CDP liquidation triggered}\\ &\qquad\quad\quad \to \texttt{Grief}\\ &\qquad \mid \texttt{pro} < \texttt{min}\\ &\qquad\quad \text{CDP's collateralization below liquidation ratio}\\ &\qquad\quad\quad \to \texttt{Panic}\\ &\qquad \mid \mathit{view}\ \texttt{zzz}\ \texttt{jar}_0 + \mathit{view}\ \mathit{lax}\ \texttt{ilk}_0 < \texttt{era}_0\\ &\qquad\quad \text{CDP type's price limbo exceeded limit}\\ &\qquad\quad\quad \to \texttt{Panic}\\ &\qquad \mid \mathit{view}\ \texttt{zzz}\ \texttt{jar}_0 < \texttt{era}_0\\ &\qquad\quad \text{CDP type's price feed in limbo}\\ &\qquad\quad\quad \to \texttt{Worry}\\ &\qquad \mid \texttt{cap} > \mathit{view}\ \texttt{hat}\ \texttt{ilk}_0\\ &\qquad\quad \text{CDP type's debt ceiling exceeded}\\ &\qquad\quad\quad \to \texttt{Anger}\\ &\qquad \mid \mathit{otherwise}\\ &\qquad\quad \text{No problems}\\ &\qquad\quad\quad \to \texttt{Pride}\\ &\quad \textbf{where}\\ &\qquad \text{CDP's collateral value in SDR:}\\ &\qquad\quad \texttt{pro} = \mathit{view}\ \texttt{jam}\ \texttt{urn}_0 * \mathit{view}\ \texttt{tag}\ \texttt{jar}_0\\ &\qquad \text{CDP type's total debt in SDR:}\\ &\qquad\quad \texttt{cap} = \mathit{view}\ \texttt{rum}\ \texttt{ilk}_0 * \mathit{cast}\ (\mathit{view}\ \texttt{chi}\ \texttt{ilk}_0)\\ &\qquad \text{CDP's debt in SDR:}\\ &\qquad\quad \texttt{con} = \mathit{view}\ \texttt{art}\ \texttt{urn}_0 * \mathit{cast}\ (\mathit{view}\ \texttt{chi}\ \texttt{ilk}_0) * \texttt{par}_0\\ &\qquad \text{Required collateral as per liquidation ratio:}\\ &\qquad\quad \texttt{min} = \texttt{con} * \mathit{cast}\ (\mathit{view}\ \texttt{mat}\ \texttt{ilk}_0)\end{aligned}$$

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.

```
feel id_urn = do
  Adjust target price and target rate
  prod
  Update debt unit and unprocessed fee revenue
  id_ilk ← look (vat . urns . ix id_urn . ilk)
  drip id_ilk
  Read parameters for risk analysis
  era_0 ← use  era
  par_0 ← use  (vat . vox . par)
  urn_0 ← look (vat . urns . ix id_urn)
  ilk_0 ← look (vat . ilks . ix (view ilk urn_0))
  jar_0 ← look (vat . gems . ix (view jar ilk_0))
  Return risk stage of CDP
  return (analyze era_0 par_0 urn_0 ilk_0 jar_0)
```

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

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

Create a CDP record with the sender as owner

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

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

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

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

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

Transfer ownership

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

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

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

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

Fail if liquidation triggered or initiated

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

Identify collateral type

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

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

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

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

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.

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

Fail if sender is not the CDP owner

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

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

Record a decrease in collateral

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

Roll back on any risk problem except debt ceiling excess

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

Transfer tokens from collateral vault to owner

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

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

$\texttt{push}\ id_{\texttt{gem}}\ id_{\texttt{lad}}\ \texttt{wad}_{\texttt{gem}}$

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.

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

Fail if sender is not the CDP owner

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

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

Update debt unit and unprocessed fee revenue

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

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

Denominate loan in debt unit

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

Increase CDP debt

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

Increase total debt of CDP type

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

Roll back on any risk problem

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

Mint dai and transfer to CDP owner

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

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

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

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

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

Fail if liquidation triggered or initiated

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

Update debt unit and unprocessed fee revenue

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

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

Denominate dai amount in debt unit

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

Decrease CDP debt

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

Decrease total CDP type debt

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

Transfer dai from CDP owner to dai vault

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

Destroy reclaimed dai

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

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

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

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

Reclaim all outstanding dai

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

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

Reclaim all collateral

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

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

Nullify CDP record

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

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.

```
prod = do
  Read all parameters relevant for feedback mechanism
  era₀ ← use era
  tau₀ ← use (vat . vox . tau)
  fix₀ ← use (vat . vox . fix)
  par₀ ← use (vat . vox . par)
  how₀ ← use (vat . vox . how)
  way₀ ← use (vat . vox . way)
  let
    Time difference in seconds
    age  = era₀ − tau₀
    Current target rate applied to target price
    par₁ = par₀ * cast (way₀ ↑↑ age)
    Sensitivity parameter applied over time
    wag = how₀ * fromIntegral age
    Target rate scaled up or down
    way₁ = inj (prj way₀ +
                  if fix₀ < par₀ then wag else − wag)
  Update target price
  vat . vox . par := par₁
  Update rate of price change
  vat . vox . way := way₁
  Record time of update
  vat . vox . tau := era₀
  where
    Convert between multiplicative and additive form
    prj x = if x ⩾ 1 then x − 1 else 1 − 1 / x
    inj x = if x ⩾ 0 then x + 1 else 1 / (1 − x)
```

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.

$$
\begin{array}{l}
\texttt{drip}\ id_{\texttt{ilk}} = \textbf{do} \\
\quad \text{Time stamp of previous } \texttt{drip} \\
\quad\quad \texttt{rho}_0 \leftarrow \mathit{look}\ (\texttt{vat} \mathbin{.} \texttt{ilks} \mathbin{.} \mathit{ix}\ id_{\texttt{ilk}} \mathbin{.} \texttt{rho}) \\
\quad \text{Current stability fee} \\
\quad\quad \texttt{tax}_0 \leftarrow \mathit{look}\ (\texttt{vat} \mathbin{.} \texttt{ilks} \mathbin{.} \mathit{ix}\ id_{\texttt{ilk}} \mathbin{.} \texttt{tax}) \\
\quad \text{Current debt unit value} \\
\quad\quad \texttt{chi}_0 \leftarrow \mathit{look}\ (\texttt{vat} \mathbin{.} \texttt{ilks} \mathbin{.} \mathit{ix}\ id_{\texttt{ilk}} \mathbin{.} \texttt{chi}) \\
\quad \text{Current total debt in debt unit} \\
\quad\quad \texttt{rum}_0 \leftarrow \mathit{look}\ (\texttt{vat} \mathbin{.} \texttt{ilks} \mathbin{.} \mathit{ix}\ id_{\texttt{ilk}} \mathbin{.} \texttt{rum}) \\
\quad \text{Current unprocessed stability fee revenue} \\
\quad\quad \texttt{joy}_0 \leftarrow \mathit{look}\ (\texttt{vat} \mathbin{.} \texttt{joy}) \\
\quad \text{Current time stamp} \\
\quad\quad \texttt{era}_0 \leftarrow \mathit{use}\ \texttt{era} \\
\quad\quad \textbf{let} \\
\quad\quad\quad \text{Time difference in seconds} \\
\quad\quad\quad\quad \mathit{age} \quad = \texttt{era}_0 - \texttt{rho}_0 \\
\quad\quad\quad \text{Value of debt unit increased according to stability fee} \\
\quad\quad\quad\quad \texttt{chi}_1 = \texttt{chi}_0 * \texttt{tax}_0 \uparrow\uparrow \mathit{age} \\
\quad\quad\quad \text{Denominate stability fee revenue in new unit} \\
\quad\quad\quad\quad \texttt{joy}_1 = \texttt{joy}_0 + (\mathit{cast}\ (\texttt{chi}_1 - \texttt{chi}_0) :: \texttt{Wad}) * \texttt{rum}_0 \\
\quad \text{Record time of update} \\
\quad\quad \texttt{vat} \mathbin{.} \texttt{ilks} \mathbin{.} \mathit{ix}\ id_{\texttt{ilk}} \mathbin{.} \texttt{rho} := \texttt{era}_0 \\
\quad \text{Record new debt unit} \\
\quad\quad \texttt{vat} \mathbin{.} \texttt{ilks} \mathbin{.} \mathit{ix}\ id_{\texttt{ilk}} \mathbin{.} \texttt{chi} := \texttt{chi}_1 \\
\quad \text{Record fee revenue denominated in new debt unit} \\
\quad\quad \texttt{vat} \mathbin{.} \texttt{joy} := \texttt{joy}_1 \\
\quad \text{Return the new debt unit} \\
\quad\quad \mathit{return}\ \texttt{chi}_1
\end{array}
$$

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.

```
mark id_gem tag_1 zzz_1 = auth $ do
  vat . gems . ix id_gem . tag := tag_1
  vat . gems . ix id_gem . zzz := zzz_1
```

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

```
tell wad_gem = auth $ do vat . vox . fix := wad_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 records the CDP's debt as bad debt in the CDP engine and enables the settler contract to later grab the collateral and begin auctioning.

```
bite id_urn = do
  Fail if CDP is not in the appropriate risk stage
    want (feel id_urn) (≡ Panic)
  Record the sender as the liquidation initiator
    id_cat ← use sender
    vat . urns . ix id_urn . cat := Just id_cat
  Update debt unit
    id_ilk ← look (vat . urns . ix id_urn . ilk)
    chi_1 ← drip id_ilk
  Apply liquidation penalty to debt
    art_0 ← look (vat . urns . ix id_urn . art)
    axe_0 ← look (vat . ilks . ix id_ilk . axe)
    let art_1 = art_0 * cast axe_0
  Update CDP debt
    vat . urns . ix id_urn . art := art_1
  Record as bad debt
    increase (vat . sin) (art_1 * cast chi_1)
```

After liquidation has been triggered, the designated settler contract invokes `grab` to receive the collateral tokens.

```
grab id_urn = auth $ do
  Fail if CDP is not marked for liquidation
  want (feel id_urn) (≡ Grief)
  Record the sender as the CDP's settler
  id_vow ← use sender
  vat . urns . ix id_urn . vow := Just id_vow
  Forget the CDP's requester of liquidation
  vat . urns . ix id_urn . cat := Nothing
  Transfer the tokens to the settler
  jam_0 ← look (vat . urns . ix id_urn . jam)
  id_ilk ← look (vat . urns . ix id_urn . ilk)
  id_gem ← look (vat . ilks . ix id_ilk . jar)
  push id_gem id_vow jam_0
```

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.

```
plop id_urn wad_DAI = auth $ do
  Fail unless CDP is in liquidation
  want (feel id_urn) (≡ Dread)
  Forget the CDP's settler
  vat . urns . ix id_urn . vow := Nothing
  Return some amount of excess auction gains
  id_vow ← use sender
  id_ilk ← look (vat . urns . ix id_urn . ilk)
  id_gem ← look (vat . ilks . ix id_ilk . jar)
  pull id_gem id_vow wad_DAI
  Record the gains as the CDP's collateral
  vat . urns . ix id_urn . jam := wad_DAI
```

When the settler has processed all the bad debt due to CDP liquidation, it invokes `heal` to reset the bad debt quantity.

```
heal wad_DAI = auth $ do vat . sin := 0
```

When the settler has processed all the accrued stability fee revenue, it invokes `love` to reset the unprocessed stability fee quantity.

```
love wad_DAI = auth $ do vat . joy := 0
```

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.

```
form id_ilk id_gem = auth $ do
  initialize (vat . ilks . at id_ilk) (defaultIlk id_gem)
```

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

```
frob how_1 = auth $ do vat . vox . how := 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.

```
cuff id_ilk mat_1 = auth $ do vat . ilks . ix id_ilk . mat := mat_1
chop id_ilk axe_1 = auth $ do vat . ilks . ix id_ilk . axe := axe_1
cork id_ilk hat_1 = auth $ do vat . ilks . ix id_ilk . hat := hat_1
calm id_ilk lax1  = auth $ do vat . ilks . ix id_ilk . lax  := lax1
```

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

```
crop id_ilk tax_1 =
  auth $ do
    Apply the current stability fee to the internal debt unit
      drip id_ilk
    Change the stability fee
      vat . ilks . ix id_ilk . tax := 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.

```
pull id_gem id_lad wad_gem =
  transfer id_gem wad_gem (InAccount id_lad) (InVault id_gem)
```

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.

```
push id_gem id_lad wad_gem =
  transfer id_gem wad_gem (InVault id_gem) (InAccount id_lad)
```

3.8 Token manipulation

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

```
transfer id_gem wad src dst =
  Operate in the token's balance table
  zoom (vat . gems . ix id_gem . ERC20 . balanceOf) $ do
    Fail if source balance insufficient
    balance ← look (ix src)
    aver (balance ⩾ wad)
    Update balances
    decrease  (ix src) wad
    initialize (at dst) 0
    increase  (ix dst) wad
```

The internal act `mint` inflates the supply of a token. It is used by `draw` to create new DAI, and by settlers to create new MKR.

```
mint id_gem wad_0 =
  zoom (vat . gems . ix id_gem . ERC20) $ do
    increase (balanceOf . ix (InVault id_gem)) wad_0
```

The internal act `burn` deflates the supply of a token. It is used by `wipe` to destroy DAI, and by settlers to destroy MKR.

```
burn id_gem wad_0 =
  zoom (vat . gems . ix id_gem . ERC20) $ do
    decrease (balanceOf . ix (InVault id_gem)) wad_0
```

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:

```
type Maker a = StateT System (Except Error) a
```

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

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

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.

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

4.2 Asserting

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

```
General assertion
aver x = unless x (throwError (AssertError ?act))

Assert that an indexed value is not present
none x = preuse x >>= λcase
  Nothing → return ()
  Just _ → throwError (AssertError ?act)

Assert that an indexed value is present
look f = preuse f >>= λcase
  Nothing → throwError (AssertError ?act)
  Just x → return x

Execute an act and assert a condition on its result
want m p = m >>= (aver . p)
```

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

```
owns id_urn id_lad = do
  want (look (vat . urns . ix id_urn . lad)) (≡ id_lad)
```

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

```
auth continue = do
  s ← use sender
  unless (s ≡ id_god) (throwError AuthError)
  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
(+), (−), (∗), (/), (↑), (↑↑), *div*,
Utilities
all, ¬, *elem*,
Constants
mempty, ⊥, *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.

```
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
import Control.Lens.Zoom as X (zoom)
```

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
```

Finally we define some of our own convenience functions.

```
decrease  a x = a -= x
increase  a x = a += x
initialize a x = a %= (λcase Nothing → Just x; y → y)
prepend   a x = a %= (x:)
x ∉ xs = ¬ (elem x xs)
```

[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 (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 (D a) = D (signum a)
  fromInteger i = D (fromInteger i)
```

In the Fractional instance, we delegate everything except division.

```
instance HasResolution e ⇒ Fractional (Decimal e) where
  x@(D (MkFixed a)) / D (MkFixed b) =
    D (MkFixed (div (a * resolution x + div b 2) b))
  recip (D a)      = D (recip a)
  fromRational r = 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 ↑ (18 :: Integer)
instance HasResolution E36 where
  resolution _ = 10 ↑ (36 :: Integer)
```

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

```
class Epsilon t where ϵ :: t
instance HasResolution a ⇒ Epsilon (Decimal a) where
    The use of ⊥ is safe since resolution ignores the value.
  ϵ = 1 / fromIntegral (resolution (⊥ :: Fixed a))
```