MAKER

presents the

REFERENCE IMPLEMENTATION

also known as the

PURPLE PAPER

of the remarkable

DAI SYSTEM

a decentralized stability mechanism

formulated by

Daniel Brockman
Mikael Brockman
Nikolai Mushegian

with last update on April 17, 2017.

◈

Contents

Chapter 1

Introduction

The DAI SYSTEM, henceforth also "Maker," is a network of Ethereum contracts defining a token that is subject to a decentralized price stability mechanism. The token is named DAI.

For an overview of the economics of the system, as well as descriptions of governance, off-chain mechanisms, and so on, see the whitepaper.

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

1.1 Naming

The implementation is formulated in terms of a parallel vocabulary whose concise words can seem meaningless at first glance (e.g., `Urn`, `par`, `ink`). These words are in fact carefully selected for metaphoric resonance and evocative qualities. Definitions of the words along with mnemonic reminders can be found in the glossary.

We have found that though it requires some initial indoctrination, the Maker jargon is good for development and helps when thinking and talking about the structure and mechanics of the system. Here are some of the reasons:

- The parallel jargon lets us sidestep terminological debates; for example, whether to say "rate of target price change" or "target rate."
- With decoupled financial and technical vocabularies, we can more flexibly improve one without affecting the other.

- The ability to discuss the system formally, with the financial interpretation partly suspended, has suggested insights that would have been harder to think of inside the normal language.
- The precise and distinctive language makes the structure and logic of the implementation more apparent and easier to formalize.

Some readers may perceive the Maker terminology as unnecessarily obscure despite our apologetics. In that case, we recommend a contrasting look at the Ethereum "yellow paper," after which this document should appear highly legible.

1.2 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.

1.3 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.

1.4 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;
- that 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

Chapter 2

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 3

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.

3.1 Numeric types

The system uses two different precisions of decimal fixed point numbers, which we call *wads* and *rays*, having respectively 18 digits of precision (used for token quantities) and 36 digits (used for precise rates and ratios). See Appendix B for details on decimal fixed point numbers and rounding.

```
 Define the distinct type 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 :: (Real a, Fractional b) ⇒ a → b
cast = fromRational . toRational
```

3.2 Identifiers and addresses

There are several kinds of identifiers used in the system, and we use types to distinguish them. The type parameter *a* creates distinct types; 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 define the different kinds of tokens used by the system.

```
data Gem = Gem String  External token
         | DAI         Stablecoin
         | SIN         Anticoin
         | MKR         Countercoin
  deriving (Eq, Ord, Show)
```

3.3 Tag — external token price data

The data received from price feeds is categorized by token and stored in Tag records.

```
data Tag = Tag {
  · tag :: Wad,  Market price denominated in SDR
  · zzz :: Sec   Time of price expiration
  } deriving (Eq, Show)
```

3.4 Entity — token balance holder

We use a data type to explicitly distinguish the different entities that can hold a token balance or invoke acts.

```
data Entity = Account Address  External holder
            | Jar              Token vault

            | Joy              Spawning account for stablecoin
            | Woe              Spawning account for anticoin
            | Ice              Holding account for anticoin

            | Vow              Settler
            | Flipper          Assetcoin auctioneer
            | Flapper          Stablecoin auctioneer
            | Flopper          Countercoin auctioneer

            | Toy              Test driver
            | God              Omnipotent actor
  deriving (Eq, Ord, Show)
```

3.5 `Ilk` — urn type

Each urn belongs to an urn 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 4; see the whitepaper for an overview.

```
data Ilk = Ilk {
  · gem :: Gem,  Assetcoin identifier

  · tax :: Sec,  Grace period after price feed becomes unavailable
  · mat :: Ray,  Riddance ratio of assetcoin to stablecoin
  · axe :: Ray,  Riddance penalty as fraction of art
  · hat :: Wad,  Maximum total stablecoin for ilk ("ceiling")
  · tax :: Ray,  Stability fee as per-second fraction of an urn's stablecoin

  · chi :: Ray,  Value of fee unit in dai
  · rho :: Sec,  Time of latest fee unit adjustment
  · rum :: Wad   Total stablecoin for ilk denominated in fee unit
  } deriving (Eq, Show)
```

3.6 `Urn` — stablecoin issuance account

An urn record defines a basic entity through which users interact with the system to issue stablecoin. Each urn belongs to an ilk. The urn records an amount of locked assetcoin along with the amount of stablecoin created for this particular urn. When riddance is initiated on an urn, the identity of the triggering entity is also recorded.

data `Urn` = `Urn` {

· `ilk` :: Id `Ilk`,	Urn type
· `lad` :: Entity,	Urn owner
· `art` :: `Wad`,	Issued stablecoin in fee unit
· `ink` :: `Wad`,	Locked assetcoin in fee unit
· `cat` :: Maybe Entity	Entity that triggered riddance, if applicable

} **deriving** (Eq, Show)

3.7 `Vox` — feedback mechanism data

The *feedback mechanism* is the aspect of the system that adjusts the target price of dai based on market price. Its data is grouped in a 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)

3.8 `Vat` — system root

The `Vat` record aggregates the records of tokens, urns, ilks, and price feeds, along with the data of the feedback mechanism.

data `Vat` = `Vat` {

· `tags` :: Map `Gem` `Tag`,	Token price feeds

```
  · ilks :: Map (Id Ilk) Ilk,   Urn type records
  · urns :: Map (Id Urn) Urn,   Urn records
  · vox  :: Vox                 Feedback mechanism data
  } deriving (Eq, Show)
```

3.9 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 {
  · balances :: Map (Entity, Gem) Wad,   Token balances
  · vat      :: Vat,                     Root Maker entity
  · era      :: Sec,                     Current time stamp
  · sender   :: Entity,                  Sender of current act
  · accounts :: [Address],               For test suites
  · mode     :: Mode                     Vow operation mode
  } deriving (Eq, Show)
```

```
data Mode = Dummy
  deriving (Eq, Show)
```

3.10 Default data

```
defaultIlk :: Gem → Ilk
defaultIlk id_gem = Ilk {
  · gem = id_gem,
  · axe = Ray 1,
  · mat = Ray 1,
  · tax = Ray 1,
  · hat = Wad 0,
  · tax = Sec 0,
  · chi = Ray 1,
  · rum = Wad 0,
  · rho = Sec 0
}

emptyUrn :: Id Ilk → Entity → Urn
emptyUrn id_ilk id_lad = Urn {
  · cat = Nothing,
  · lad = id_lad,
  · ilk = id_ilk,
  · art = Wad 0,
  · ink = Wad 0
}

initialTag :: Tag
initialTag = Tag {
  · tag = Wad 0,
  · zzz = 0
}

initialVat :: Ray → Vat
initialVat how₀ = Vat {
  · vox  = Vox {
    · tau = 0,
    · wut = Wad 1,
    · par = Wad 1,
    · how = how₀,
    · way = Ray 1
  },
  · ilks = ∅,
  · urns = ∅,
  · tags = ∅
}

initialSystem :: Ray → System
initialSystem how₀ = System {
  · balances = ∅,
  · vat      = initialVat how₀,
  · era      = 0,
  · sender   = God,
  · accounts = mempty,
  · mode     = Dummy
}
```

Chapter 4

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 5.

4.1 Assessment

In order to prohibit urn acts based on risk situation, we define these 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 an urn.

```
analyze $era_0$ $par_0$ $urn_0$ $ilk_0$ $tag_0$ =
  if | view cat $urn_0$ $\not\equiv$ Nothing $\wedge$ view ink $urn_0$ $\equiv 0$
         Riddance triggered and started
           $\rightarrow$ Dread
     | view cat $urn_0$ $\not\equiv$ Nothing
         Riddance triggered
           $\rightarrow$ Grief
     | pro < min
         Value ratio of assetcoin to stablecoin below minimum
           $\rightarrow$ Panic
     | view zzz $tag_0$ + view lax $ilk_0$ < $era_0$
         Gem price limbo exceeded limit
           $\rightarrow$ Panic
     | view zzz $tag_0$ < $era_0$
         Gem price feed in limbo
           $\rightarrow$ Worry
     | cap > view hat $ilk_0$
         Ilk ceiling exceeded
           $\rightarrow$ Anger
     | otherwise
         No problems
           $\rightarrow$ Pride
  where
   Urn's assetcoin value in SDR:
    pro = view ink $urn_0$ * view tag $tag_0$
   Ilk's total stablecoin issuance in DAI:
    cap = view rum $ilk_0$ * cast (view chi $ilk_0$)
   Urn's stablecoin issuance denominated in SDR:
    con = view art $urn_0$ * cast (view chi $ilk_0$) * $par_0$
   Required assetcoin as per riddance ratio:
    min = con * cast (view mat $ilk_0$)
```

Table 4.1: Urn acts and risk stages

	`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 pay
☝	allowed for owner if above riddance ratio
✊	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 $id_{\texttt{urn}}$ = do
  Adjust target price and target rate
    prod
  Update fee unit and unprocessed fee revenue
    $id_{\texttt{ilk}}$ ← $look$ (vat . urns . $ix$ $id_{\texttt{urn}}$ . ilk)
    drip $id_{\texttt{ilk}}$
  Read parameters for risk analysis
    $\texttt{era}_0$ ← $use$ era
    $\texttt{par}_0$ ← $use$ (vat . vox . par)
    $\texttt{urn}_0$ ← $look$ (vat . urns . $ix$ $id_{\texttt{urn}}$)
    $\texttt{ilk}_0$ ← $look$ (vat . ilks . $ix$ ($view$ ilk $\texttt{urn}_0$))
    $\texttt{tag}_0$ ← $look$ (vat . tags . $ix$ ($view$ gem $\texttt{ilk}_0$))
  Return risk stage of urn
    $return$ ($analyze$ $\texttt{era}_0$ $\texttt{par}_0$ $\texttt{urn}_0$ $\texttt{ilk}_0$ $\texttt{tag}_0$)
```

Urn acts use `feel` to prohibit increasing risk when already risky, and to freeze stablecoin and assetcoin during riddance; see Table 4.1.

4.2 Issuance

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

```
$\mathtt{open}\ id_{\mathtt{urn}}\ id_{\mathtt{ilk}} = \mathbf{do}$
  Fail if account identifier is taken
    $none\ (\mathtt{vat}\ .\ \mathtt{urns}\ .\ ix\ id_{\mathtt{urn}})$
  Fail if ilk type is not present
    $\_ \leftarrow look\ (\mathtt{vat}\ .\ \mathtt{ilks}\ .\ ix\ id_{\mathtt{ilk}})$
  Create an urn with the sender as owner
    $id_{\mathtt{lad}} \leftarrow use\ sender$
    $initialize\ (\mathtt{vat}\ .\ \mathtt{urns}\ .\ at\ id_{\mathtt{urn}})\ (emptyUrn\ id_{\mathtt{ilk}}\ id_{\mathtt{lad}})$
```

The owner of an urn can transfer its ownership at any time using `give`.

```
$\mathtt{give}\ id_{\mathtt{urn}}\ id_{\mathtt{lad}} = \mathbf{do}$
  Fail if sender is not the urn owner
    $id_{sender} \leftarrow use\ sender$
    $owns\ id_{\mathtt{urn}}\ id_{sender}$
  Transfer urn ownership
    $\mathtt{vat}\ .\ \mathtt{urns}\ .\ ix\ id_{\mathtt{urn}}\ .\ \mathtt{lad} := id_{\mathtt{lad}}$
```

Unless urn is in riddance, its owner can use `lock` to lock more assetcoin.

```
$\mathtt{lock}\ id_{\mathtt{urn}}\ \mathtt{wad}_{\mathtt{gem}} = \mathbf{do}$
  Fail if sender is not the urn owner
    $id_{\mathtt{lad}} \leftarrow use\ sender$
    $owns\ id_{\mathtt{urn}}\ id_{\mathtt{lad}}$
  Fail if riddance in process
    $want\ (\mathtt{feel}\ id_{\mathtt{urn}})\ (\notin [\mathtt{Grief}, \mathtt{Dread}])$
  Identify assetcoin
    $id_{\mathtt{ilk}} \leftarrow look\ (\mathtt{vat}\ .\ \mathtt{urns}\ .\ ix\ id_{\mathtt{urn}}\ .\ \mathtt{ilk})$
    $id_{\mathtt{gem}} \leftarrow look\ (\mathtt{vat}\ .\ \mathtt{ilks}\ .\ ix\ id_{\mathtt{ilk}}\ .\ \mathtt{gem})$
  Take custody of assetcoin
    $transfer\ id_{\mathtt{gem}}\ \mathtt{wad}_{\mathtt{gem}}\ id_{\mathtt{lad}}\ \mathtt{Jar}$
  Record an assetcoin balance increase
    $increase\ (\mathtt{vat}\ .\ \mathtt{urns}\ .\ ix\ id_{\mathtt{urn}}\ .\ \mathtt{ink})\ \mathtt{wad}_{\mathtt{gem}}$
```

When an urn has no risk problems (except that its ilk's ceiling may be exceeded), its owner can use `free` to reclaim some amount of assetcoin, as long as this would not take the urn below its riddance ratio.

```
$\texttt{free}\ \mathit{id}_{\texttt{urn}}\ \texttt{wad}_{\texttt{gem}} = \textbf{do}$
  Fail if sender is not the urn owner
    $\mathit{id}_{\texttt{lad}} \leftarrow \mathit{use\ sender}$
    $\mathit{owns}\ \mathit{id}_{\texttt{urn}}\ \mathit{id}_{\texttt{lad}}$
  Record an assetcoin balance decrease
    $\mathit{decrease}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{urn}}\ .\ \texttt{ink})\ \texttt{wad}_{\texttt{gem}}$
  Roll back on any risk problem except ilk ceiling excess
    $\mathit{want}\ (\texttt{feel}\ \mathit{id}_{\texttt{urn}})\ (\in [\texttt{Pride}, \texttt{Anger}])$
  Release custody of assetcoin quantity
    $\mathit{id}_{\texttt{ilk}} \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{urn}}\ .\ \texttt{ilk})$
    $\mathit{id}_{\texttt{gem}} \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{ilk}}\ .\ \texttt{gem})$
    $\mathit{transfer}\ \mathit{id}_{\texttt{gem}}\ \texttt{wad}_{\texttt{gem}}\ \texttt{Jar}\ \mathit{id}_{\texttt{lad}}$
```

When an urn has no risk problems, its owner can can use `draw` to issue fresh stablecoin, as long as the ilk ceiling is not exceeded and the issuance would not take the urn below its riddance ratio.

```
$\texttt{draw}\ \mathit{id}_{\texttt{urn}}\ \texttt{wad}_{\text{DAI}} = \textbf{do}$
  Fail if sender is not the urn owner
    $\mathit{id}_{\texttt{lad}} \leftarrow \mathit{use\ sender}$
    $\mathit{owns}\ \mathit{id}_{\texttt{urn}}\ \mathit{id}_{\texttt{lad}}$
  Update fee 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 issuance quantity in fee unit
    $\textbf{let}\ \texttt{wad}_{\texttt{chi}} = \texttt{wad}_{\text{DAI}}\ /\ \mathit{cast}\ \texttt{chi}_1$
  Record increase of urn's stablecoin issuance
    $\mathit{increase}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{urn}}\ .\ \texttt{art})\ \texttt{wad}_{\texttt{chi}}$
  Record increase of ilk's stablecoin issuance
    $\mathit{increase}\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ \mathit{ix}\ \mathit{id}_{\texttt{ilk}}\ .\ \texttt{rum})\ \texttt{wad}_{\texttt{chi}}$
  Roll back on any risk problem
    $\mathit{want}\ (\texttt{feel}\ \mathit{id}_{\texttt{urn}})\ (\equiv \texttt{Pride})$
  Mint both stablecoin and anticoin
    $\texttt{lend}\ \texttt{wad}_{\text{DAI}}$
  Transfer stablecoin to urn owner
```

```
    $\textit{transfer}\ \texttt{DAI}\ \texttt{wad}_{\text{DAI}}\ \texttt{Joy}\ \textit{id}_{\texttt{lad}}$
  Transfer anticoin to anticoin holding account
    $\textit{transfer}\ \texttt{SIN}\ \texttt{wad}_{\text{DAI}}\ \texttt{Woe Ice}$
```

An urn owner who has previously issued stablecoin can use `wipe` to send back dai and reduce the urn's issuance.

```
$\texttt{wipe}\ \textit{id}_{\texttt{urn}}\ \texttt{wad}_{\text{DAI}} = \textbf{do}$
  Fail if sender is not the urn owner
    $\textit{id}_{\texttt{lad}} \leftarrow \textit{use sender}$
    $\textit{owns}\ \textit{id}_{\texttt{urn}}\ \textit{id}_{\texttt{lad}}$
  Fail if urn is in riddance
    $\textit{want}\ (\texttt{feel}\ \textit{id}_{\texttt{urn}})\ (\notin [\texttt{Grief}, \texttt{Dread}])$
  Update fee unit and unprocessed fee revenue
    $\textit{id}_{\texttt{ilk}} \leftarrow \textit{look}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \textit{ix}\ \textit{id}_{\texttt{urn}}\ .\ \texttt{ilk})$
    $\texttt{chi}_1 \leftarrow \texttt{drip}\ \textit{id}_{\texttt{ilk}}$
  Denominate stablecoin amount in fee unit
    $\textbf{let}\ \texttt{wad}_{\texttt{chi}} = \texttt{wad}_{\text{DAI}}\ /\ \textit{cast}\ \texttt{chi}_1$
  Record decrease of urn issuance
    $\textit{decrease}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \textit{ix}\ \textit{id}_{\texttt{urn}}\ .\ \texttt{art})\ \texttt{wad}_{\texttt{chi}}$
  Record decrease of ilk total issuance
    $\textit{decrease}\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ \textit{ix}\ \textit{id}_{\texttt{ilk}}\ .\ \texttt{rum})\ \texttt{wad}_{\texttt{chi}}$
  Take custody of stablecoin from urn owner
    $\textit{transfer}\ \texttt{DAI}\ \texttt{wad}_{\text{DAI}}\ \textit{id}_{\texttt{lad}}\ \texttt{Jar}$
  Destroy stablecoin and anticoin
    $\texttt{mend}\ \texttt{wad}_{\text{DAI}}$
```

An urn owner can use `shut` to close their account—reversing all issuance plus fee and reclaiming all assetcoin—if the price feed is up to date and the urn is not in riddance.

```
$\texttt{shut}\ \textit{id}_{\texttt{urn}} = \textbf{do}$
    Update fee unit and unprocessed fee revenue
      $\textit{id}_{\texttt{ilk}} \leftarrow \textit{look}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \textit{ix}\ \textit{id}_{\texttt{urn}}\ .\ \texttt{ilk})$
      $\texttt{chi}_1 \leftarrow \texttt{drip}\ \textit{id}_{\texttt{ilk}}$
    Reverse all issued stablecoin plus fee
      $\texttt{art}_0 \leftarrow \textit{look}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \textit{ix}\ \textit{id}_{\texttt{urn}}\ .\ \texttt{art})$
      $\texttt{wipe}\ \textit{id}_{\texttt{urn}}\ (\texttt{art}_0 * \textit{cast}\ \texttt{chi}_1)$
    Reclaim all locked assetcoin
      $\texttt{ink}_0 \leftarrow \textit{look}\ (\texttt{vat}\ .\ \texttt{urns}\ .\ \textit{ix}\ \textit{id}_{\texttt{urn}}\ .\ \texttt{ink})$
```

```
  free $id_{\texttt{urn}}$ $\texttt{ink}_0$
Nullify urn record
  vat . urns . $at\ id_{\texttt{urn}} := \text{Nothing}$
```

4.3 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 urn act that uses `feel` to assess risk.

```
prod = do
  Read all parameters relevant for feedback mechanism
    era_0 ← use era
    tau_0 ← use (vat . vox . tau)
    wut_0 ← use (vat . vox . wut)
    par_0 ← use (vat . vox . par)
    how_0 ← use (vat . vox . how)
    way_0 ← use (vat . vox . way)
    let
      Time difference in seconds
        age  = era_0 − tau_0
      Current target rate applied to target price
        par_1 = par_0 ∗ cast (way_0 ↑↑ age)
      Sensitivity parameter applied over time
        wag = how_0 ∗ fromIntegral age
      Target rate scaled up or down
        way_1 = inj (prj way_0 +
                     if wut_0 < par_0 then wag else − wag)
  Update target price
    vat . vox . par := par_1
  Update rate of price change
    vat . vox . way := way_1
  Record time of update
    vat . vox . tau := era_0
    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 an ilk can change through governance. Due to the constraint that acts should run in constant time, the system cannot iterate over urns to effect such changes. Instead each ilk has a single “fee 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 urn risk.

```
drip $id_{\texttt{ilk}}$ = do
  Time stamp of previous drip
    $\texttt{rho}_0 \leftarrow \textit{look}\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ \textit{ix}\ id_{\texttt{ilk}}\ .\ \texttt{rho})$
  Current stability fee
    $\texttt{tax}_0 \leftarrow \textit{look}\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ \textit{ix}\ id_{\texttt{ilk}}\ .\ \texttt{tax})$
  Current fee unit value
    $\texttt{chi}_0 \leftarrow \textit{look}\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ \textit{ix}\ id_{\texttt{ilk}}\ .\ \texttt{chi})$
  Current total issuance in fee unit
    $\texttt{rum}_0 \leftarrow \textit{look}\ (\texttt{vat}\ .\ \texttt{ilks}\ .\ \textit{ix}\ id_{\texttt{ilk}}\ .\ \texttt{rum})$
  Current time stamp
    $\texttt{era}_0 \leftarrow \textit{use}\ \texttt{era}$
    let
      Time difference in seconds
        $\textit{age} = \texttt{era}_0 - \texttt{rho}_0$
      Value of fee unit increased according to stability fee
        $\texttt{chi}_1 = \texttt{chi}_0 * \texttt{tax}_0 \uparrow\uparrow \textit{age}$
      Stability fee revenue denominated in new unit
        $\textit{dew} = (\textit{cast}\ (\texttt{chi}_1 - \texttt{chi}_0) :: \texttt{Wad}) * \texttt{rum}_0$
  Mint stablecoin and anticoin for marginally accrued fee
    $\texttt{lend}\ \textit{dew}$
  Record time of update
    $\texttt{vat}\ .\ \texttt{ilks}\ .\ \textit{ix}\ id_{\texttt{ilk}}\ .\ \texttt{rho} := \texttt{era}_0$
  Record new fee unit
    $\texttt{vat}\ .\ \texttt{ilks}\ .\ \textit{ix}\ id_{\texttt{ilk}}\ .\ \texttt{chi} := \texttt{chi}_1$
  Return the new fee unit
    $\textit{return}\ \texttt{chi}_1$
```

4.4 Price feed input

The mark act records a new market price of an assetcoin along with the expiration date of this price.

```
mark $id_{\texttt{gem}}$ $\texttt{tag}_1$ $\texttt{zzz}_1$ = auth $ do
   initialize (vat . tags . at $id_{\texttt{gem}}$) Tag {
     · tag = $\texttt{tag}_1$,
     · zzz = $\texttt{zzz}_1$
       }
```

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

```
tell wad = auth $ do vat . vox . wut := wad
```

4.5 Riddance

When an urn's stage marks it as in need of riddance, any account can invoke the bite act to trigger the riddance process. This enables the settler contract to grab the assetcoin for auctioning and take over the anticoin.

```
bite $id_{\texttt{urn}}$ = do
    Fail if urn is not in the appropriate stage
      want (feel $id_{\texttt{urn}}$) (≡ Panic)
    Record the sender as the riddance initiator
      $id_{\texttt{cat}}$ ← use sender
      vat . urns . ix $id_{\texttt{urn}}$ . cat := Just $id_{\texttt{cat}}$
    Apply riddance penalty to urn issuance
      $id_{\texttt{ilk}}$ ← look (vat . urns . ix $id_{\texttt{urn}}$ . ilk)
      $\texttt{axe}_0$ ← look (vat . ilks . ix $id_{\texttt{ilk}}$ . axe)
      $\texttt{art}_0$ ← look (vat . urns . ix $id_{\texttt{urn}}$ . art)
      let $\texttt{art}_1$ = $\texttt{art}_0$ ∗ cast $\texttt{axe}_0$
    Update urn issuance to include penalty
      vat . urns . ix $id_{\texttt{urn}}$ . art := $\texttt{art}_1$
```

After riddance has been triggered, the designated settler contract invokes grab to receive both the urn's assetcoin and the anticoins corresponding to the urn's issuance.

```
grab $id_{\texttt{urn}}$ = auth $ do
    Fail if urn is not marked for riddance
      $want$ (feel $id_{\texttt{urn}}$) ($\equiv$ Grief)
      $\texttt{ink}_0 \leftarrow look$ (vat . urns . $ix\ id_{\texttt{urn}}$ . ink)
      $\texttt{art}_0 \leftarrow look$ (vat . urns . $ix\ id_{\texttt{urn}}$ . art)
      $id_{\texttt{ilk}} \leftarrow look$ (vat . urns . $ix\ id_{\texttt{urn}}$ . ilk)
      $id_{\texttt{gem}} \leftarrow look$ (vat . ilks . $ix\ id_{\texttt{ilk}}$ . gem)
    Update the fee unit and unprocessed fee revenue
      $\texttt{chi}_1 \leftarrow$ drip $id_{\texttt{ilk}}$
    Denominate the issuance in dai
      let con = $\texttt{art}_0 * cast\ \texttt{chi}_1$
    Transfer assetcoin and anticoin to settler
      $transfer\ id_{\texttt{gem}}\ \texttt{ink}_0$ Jar Vow
      $transfer$ SIN  con  Jar Vow
    Nullify urn's assetcoin and anticoin quantities
      vat . urns . $ix\ id_{\texttt{urn}}$ . ink := 0
      vat . urns . $ix\ id_{\texttt{urn}}$ . art := 0
    Decrease the ilk's total issuance
      $decrease$ (vat . ilks . $ix\ id_{\texttt{ilk}}$ . rum) $\texttt{art}_0$
```

When the settler has finished the riddance of an urn, it invokes plop to give back any assetcoin it did not need to sell and restore the urn.

```
plop $id_{\texttt{urn}}$ $\texttt{wad}_{\text{DAI}}$ = auth $ do
    Fail unless urn is in the proper stage
      $want$ (feel $id_{\texttt{urn}}$) ($\equiv$ Dread)
    Forget the urn's initiator of riddance
      vat . urns . $ix\ id_{\texttt{urn}}$ . cat := Nothing
    Take excess assetcoin from settler to vault
      $id_{\texttt{vow}} \leftarrow use\ sender$
      $id_{\texttt{ilk}} \leftarrow look$ (vat . urns . $ix\ id_{\texttt{urn}}$ . ilk)
      $id_{\texttt{gem}} \leftarrow look$ (vat . ilks . $ix\ id_{\texttt{ilk}}$ . gem)
      $transfer\ id_{\texttt{gem}}\ \texttt{wad}_{\text{DAI}}\ id_{\texttt{vow}}$ Jar
    Record the excess assetcoin as belonging to the urn
      vat . urns . $ix\ id_{\texttt{urn}}$ . ink := $\texttt{wad}_{\text{DAI}}$
```

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

```
loot = auth $ do
  The dai vault's balance is the uncollected stability fee revenue
  wad ← look (balance DAI Jar)
  Transfer the entire dai vault balance to sender
  $id_{vow}$ ← use sender
  transfer DAI wad Jar Vow
```

4.6 Auctioning

```
flip $id_{gem}$ wad_jam wad_tab $id_{urn}$ = do
  vow ← look mode
  case vow of
    Dummy → return ()
```

```
flap = do
  vow ← look mode
  case vow of
    Dummy → return ()
```

```
flop = do
  vow ← look mode
  case vow of
    Dummy → return ()
```

4.7 Settlement

```
tidy who = auth $ do
  Find the entity's stablecoin and anticoin balances
    awe ← look (balance DAI who)
    woe ← look (balance SIN who)
  We can burn at most the smallest of the two balances
    let x = min awe woe
  Transfer stablecoin and anticoin to the settler
    transfer DAI x who Vow
    transfer SIN x who Vow
  Burn both stablecoin and anticoin
    burn   DAI x    Vow
    burn   SIN x    Vow
```

```
kick = do
  Transfer unprocessed stability fee revenue to vow account
    loot
  Cancel stablecoin against anticoin
    tidy Vow
  Assign any remaining stablecoin to countercoin-deflating auction
    transferAll DAI Vow Flapper
    flap
  Assign any remaining anticoin to countercoin-inflating auction
    transferAll SIN Vow Flopper
    flop
```

4.8 Governance

Governance uses `form` to create a new ilk. Since the new type is initialized with a zero ceiling, a separate transaction can safely set the risk parameters before any issuance 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 $\mathtt{how}_1$ = auth $ do vat . vox . how := $\mathtt{how}_1$
```

Governance can alter the five risk parameters of an ilk using `cuff` for the riddance ratio; `chop` for the riddance penalty; `cork` for the ilk ceiling; `calm` for the duration of price limbo; and `crop` for the stability fee.

```
cuff $id_{\mathtt{ilk}}$ $\mathtt{mat}_1$ = auth $ do vat . ilks . ix $id_{\mathtt{ilk}}$ . mat := $\mathtt{mat}_1$
chop $id_{\mathtt{ilk}}$ $\mathtt{axe}_1$ = auth $ do vat . ilks . ix $id_{\mathtt{ilk}}$ . axe := $\mathtt{axe}_1$
cork $id_{\mathtt{ilk}}$ $\mathtt{hat}_1$ = auth $ do vat . ilks . ix $id_{\mathtt{ilk}}$ . hat := $\mathtt{hat}_1$
calm $id_{\mathtt{ilk}}$ lax1  = auth $ do vat . ilks . ix $id_{\mathtt{ilk}}$ . lax := lax1
```

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

```
crop $id_{\mathtt{ilk}}$ $\mathtt{tax}_1$ = auth $ do
    Apply the current stability fee to the internal fee unit
      drip $id_{\mathtt{ilk}}$
    Change the stability fee
      vat . ilks . ix $id_{\mathtt{ilk}}$ . tax := $\mathtt{tax}_1$
```

4.9 Token manipulation

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

```
transfer $id_{\mathtt{gem}}$ wad src dst =
  Operate in the token's balance table
    zoom balances $ do
      Fail if source balance insufficient
        balance ← look (ix (src, $id_{\mathtt{gem}}$))
        aver (balance ⩾ wad)
      Update balances
        decrease  (ix (src, $id_{\mathtt{gem}}$)) wad
        initialize (at (dst, $id_{\mathtt{gem}}$)) 0
        increase  (ix (dst, $id_{\mathtt{gem}}$)) wad
```

```
transferAll id_gem src dst = do
  wad ← look (balance id_gem src)
  transfer id_gem wad src dst
```

The internal act `mint` inflates the supply of a token. It is used by `lend` to create new stablecoin and anticoin, and by the settler to create new countercoin.

```
mint id_gem wad dst = do
  initialize (balances . at (dst, id_gem)) 0
  increase  (balances . ix (dst, id_gem)) wad
```

The internal act `burn` deflates the supply of a token. It is used by `mend` to destroy stablecoin and anticoin, and by the settler to destroy countercoin.

```
burn id_gem wad src =
  decrease (balances . ix (src, id_gem)) wad
```

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

```
lend wad_DAI = do
  mint DAI wad_DAI Joy
  mint SIN wad_DAI Woe
```

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

```
mend wad_DAI = do
  burn DAI wad_DAI Jar
  burn SIN wad_DAI Ice
```

Chapter 5

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.

5.1 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)
```

5.2 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 ≡ 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 (..), read,
  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
$(+), (-), (*), (/), (\uparrow), (\uparrow\uparrow), \mathit{div},$

Utilities
$\mathit{all}, \neg, \mathit{elem}, (\wedge),$

Constants
$\mathit{mempty}, \bot, \mathit{otherwise})$

We use a typical composition of monad transformers from the `mtl` library to structure stateful actions. See section 5.1 (*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
  fromList)     Creates a mapping with several key–value pairs
```

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))
```