



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 April 6, 2017.



Contents

1	Introduction	5
1.1	Naming	6
1.2	Motivation	6
1.3	Limitations	7
1.4	Verification	8
I	Implementation	9
2	Preamble	10
3	Types	11
3.1	Numeric types	11
3.2	Identifiers and addresses	12
3.3	Jar — collateral vault	12
	gem — collateral token	12
	tag — market price of token	12
	zzz — expiration time of token price feed	12
3.4	Gem — token model	13
3.5	Ilk — CDP type	13
	jar — collateral token vault	13
	mat — liquidation ratio	13
	axe — liquidation penalty	13
	hat — debt ceiling	13
	tax — stability fee	13
	lax — price feed limbo duration	13
	rho — time of debt unit adjustment	13
	rum — total outstanding debt units	13
	chi — value of debt unit in DAI	13
3.6	Urn — collateralized debt position (CDP)	14

	cat — address of liquidation initiator	14
	lad — CDP owner	14
	ilk — CDP type	14
	art — debt denominated in debt unit	14
	ink — collateral denominated in debt unit	14
3.7	Vox — feedback mechanism data	14
	wut — market price of DAI denominated in SDR	14
	par — target price of DAI denominated in SDR	14
	how — sensitivity parameter	14
	way — rate of target price change	14
	tau — time of latest feedback cycle	14
3.8	Vat — CDP engine aggregate	15
3.9	System model	15
	era — current time	15
3.10	Default data	16
4	Acts	18
4.1	Assessment	19
	feel — identify CDP risk stage	19
4.2	Lending	21
	open — create CDP	21
	give — transfer CDP account	21
	lock — deposit collateral	21
	free — withdraw collateral	22
	draw — issue dai as debt	22
	wipe — repay debt and burn dai	23
	shut — wipe, free, and delete CDP	23
4.3	Adjustment	25
	prod — adjust target price and target rate	25
	drip — update debt unit and unprocessed fee revenue	26
4.4	Price feed input	27
	mark — update market price of collateral token	27
	tell — update market price of dai	27
4.5	Liquidation	27
	bite — mark for liquidation	27
	grab — take tokens for liquidation	28
	plop — finish liquidation returning profit	28
	loot — take unprocessed stability fees	29
4.6	Auctioning	29
	flip — put collateral up for auction	29
	flap — put fee revenue up for auction	29

	flop — put MKR up for auction	29
4.7	Settlement	30
	tidy — burn equal quantities of DAI and SIN	30
	kick — flap, flop, and whatnot	30
4.8	Governance	30
	form — create a new CDP type	30
	frob — set the sensitivity parameter	31
	chop — set liquidation penalty	31
	cork — set debt ceiling	31
	calm — set limbo duration	31
	cuff — set liquidation ratio	31
	crop — set stability fee	31
4.9	Vaults	31
	pull — transfer tokens to vault	31
	push — transfer tokens from vault	31
4.10	Token manipulation	32
	mint — inflate token	32
	burn — deflate token	32
	lend — mint dai and debt token	32
	mend — burn dai and debt token	32
5	Act framework	33
5.1	The Maker monad	33
5.2	Asserting	34
A	Prelude	35
B	Fixed point numbers with rounding	38

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 SDR¹ 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 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 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

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

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 have predefined entity identifiers.

The `DAI` token identifier
 $id_{\text{DAI}} = \text{Id "DAI"}$
The internal debt token identifier
 $id_{\text{SIN}} = \text{Id "SIN"}$
The `MKR` token identifier
 $id_{\text{MKR}} = \text{Id "MKR"}$
A test account with ultimate authority
 $id_{\text{god}} = \text{Address "GOD"}$

3.3 Jar — collateral vault

The data received from price feeds is categorized by token and stored in Jar records. Our model also has the token balances embedded in these records; in reality¹, the balances are in separate ERC20 contracts.

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

```

data Jar = Jar {
  · gem :: Gem,  Token balances
  · tag :: Wad,  Market price denominated in SDR
  · zzz :: Sec   Time of price expiration
} deriving (Eq, Show)

```

3.4 Gem — 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.”

We define a Gem record as a map tracking the token quantity held by each holding entity.

```

data Gem = Gem { · balanceOf :: Map Entity Wad }
  deriving (Eq, Show)

```

For clarity, we use a data type to distinguish the different entities that can hold a token balance.

```

data Entity = Account Address External account or contract
            | Vault (Id Jar)   Vault for collateral token
            | Joy              Spawning account for dai
            | Woe              Spawning account for debt
            | Ice              Holding account for debt
            | Vow              Settler
            | Toy              Test driver
  deriving (Eq, Ord, Show)

```

3.5 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 4; see the whitepaper for an overview.

```

data Ilk = Ilk {
  · jar :: Id Jar,   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 dai 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)

```

3.6 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 :: Entity,      Owner of CDP
  · art :: Wad,         Outstanding debt in debt unit
  · ink :: Wad,         Collateral amount in debt unit
  · cat :: Maybe Entity Entity that triggered liquidation, if applicable
} deriving (Eq, Show)

```

3.7 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.

3.8 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 {
  · jars :: Map (Id Jar) Jar, Collateral vaults
  · ilks :: Map (Id Ilk) Ilk, CDP type records
  · urns :: Map (Id Urn) Urn, CDP records
  · vox :: Vox Data of feedback mechanism
} 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 {
  · 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 :: Id Jar → Ilk

defaultIlk id_{jar} = Ilk {
 · jar = *id_{jar}*,
 · 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
}

initialJar :: Id Jar → Jar

initialJar id_{jar} = Jar {
 · gem = Gem { · balanceOf = *singleton* (Vault *id_{jar}*) 0 },
 · 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 =  $\emptyset$ ,
    · urns =  $\emptyset$ ,
    · jars = fromList [
      ( $id_{\text{DAI}}$ , initialJar  $id_{\text{DAI}}$ ),
      ( $id_{\text{SIN}}$ , initialJar  $id_{\text{SIN}}$ ),
      ( $id_{\text{MKR}}$ , initialJar  $id_{\text{MKR}}$ )
    ]
  }

```

```

initialSystem :: Ray → System
initialSystem how0 = System {
  · vat      = initialVat how0,
  · era      = 0,
  · sender   = Account  $id_{\text{god}}$ ,
  · accounts = mempty
}

```

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 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 ink 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 ink urn0 * view tag jar0
```

CDP type's total debt in DAI:

```
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 4.1: 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
   $id_{ilk} \leftarrow look(vat.urns.ix\ id_{urn}.ilk)$ 
  drip  $id_{ilk}$ 
  Read parameters for risk analysis
   $era_0 \leftarrow use\ era$ 
   $par_0 \leftarrow use\ (vat.vox.par)$ 
   $urn_0 \leftarrow look(vat.urns.ix\ id_{urn})$ 
   $ilk_0 \leftarrow look(vat.ilks.ix\ (view\ ilk\ urn_0))$ 
   $jar_0 \leftarrow look(vat.jars.ix\ (view\ jar\ ilk_0))$ 
  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 4.1.

4.2 Lending

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

```
open  $id_{urn}$   $id_{ilk}$  = do  
  Fail if account identifier is taken  
   $none$  ( $vat . urns . ix\ id_{urn}$ )  
  Create a CDP record with the sender as owner  
   $id_{lad} \leftarrow use\ sender$   
   $initialize$  ( $vat . urns . at\ id_{urn}$ ) ( $emptyUrn\ id_{ilk}\ id_{lad}$ )
```

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

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

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

```
lock  $id_{urn}$   $wad_{gem}$  = do  
  Fail if sender is not the CDP owner  
   $id_{lad} \leftarrow use\ sender$   
   $owns\ id_{urn}\ id_{lad}$   
  Fail if liquidation triggered or initiated  
   $want$  ( $feel\ id_{urn}$ ) ( $\notin [Grief, Dread]$ )  
  Identify collateral type  
   $id_{ilk} \leftarrow look$  ( $vat . urns . ix\ id_{urn} . ilk$ )  
   $id_{jar} \leftarrow look$  ( $vat . ilks . ix\ id_{ilk} . jar$ )  
  Transfer tokens from owner to collateral vault  
   $pull\ id_{jar}\ id_{lad}\ wad_{gem}$   
  Record an increase in collateral  
   $increase$  ( $vat . urns . ix\ id_{urn} . ink$ )  $wad_{gem}$ 
```

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.

```

free  $id_{urn}$   $wad_{gem}$  = do
  Fail if sender is not the CDP owner
   $id_{lad} \leftarrow use\ sender$ 
  owns  $id_{urn}$   $id_{lad}$ 
  Record a decrease in collateral
  decrease ( $vat . urns . ix\ id_{urn} . ink$ )  $wad_{gem}$ 
  Roll back on any risk problem except debt ceiling excess
  want ( $feel\ id_{urn}$ ) ( $\in [Pride, Anger]$ )
  Transfer tokens from collateral vault to owner
   $id_{ilk} \leftarrow look\ (vat . urns . ix\ id_{urn} . ilk)$ 
   $id_{jar} \leftarrow look\ (vat . ilks . ix\ id_{ilk} . jar)$ 
  push  $id_{jar}$   $id_{lad}$   $wad_{gem}$ 

```

When a CDP has no risk problems, its owner 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.

```

draw  $id_{urn}$   $wad_{dai}$  = do
  Fail if sender is not the CDP owner
   $id_{lad} \leftarrow use\ sender$ 
  owns  $id_{urn}$   $id_{lad}$ 
  Update debt unit and unprocessed fee revenue
   $id_{ilk} \leftarrow look\ (vat . urns . ix\ id_{urn} . ilk)$ 
   $chi_1 \leftarrow drip\ id_{ilk}$ 
  Denominate loan in debt unit
  let  $wad_{chi} = wad_{dai} / cast\ chi_1$ 
  Increase CDP debt
  increase ( $vat . urns . ix\ id_{urn} . art$ )  $wad_{chi}$ 
  Increase total debt of CDP type
  increase ( $vat . ilks . ix\ id_{ilk} . rum$ )  $wad_{chi}$ 
  Roll back on any risk problem
  want ( $feel\ id_{urn}$ ) ( $\equiv Pride$ )
  Mint both dai and debt tokens
  lend  $wad_{dai}$ 
  Transfer dai to CDP owner

```

transfer id_{DAI} wad_{DAI} Joy id_{lad}

Transfer sin into debt vault

transfer id_{SIN} wad_{DAI} Woe Ice

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.

wipe id_{urn} wad_{DAI} = **do**

Fail if sender is not the CDP owner

$id_{\text{lad}} \leftarrow \text{use sender}$

$\text{owns } id_{\text{urn}} id_{\text{lad}}$

Fail if liquidation triggered or initiated

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

Update debt unit and unprocessed fee revenue

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

$chi_1 \leftarrow \text{drip } id_{\text{ilk}}$

Denominate dai amount in debt unit

let $wad_{\text{chi}} = wad_{\text{DAI}} / \text{cast } chi_1$

Decrease CDP debt

$\text{decrease}(\text{vat} . \text{urns} . \text{ix } id_{\text{urn}} . \text{art}) wad_{\text{chi}}$

Decrease total CDP type debt

$\text{decrease}(\text{vat} . \text{ilks} . \text{ix } id_{\text{ilk}} . \text{rum}) wad_{\text{chi}}$

Transfer dai from CDP owner to dai vault

pull id_{DAI} id_{lad} wad_{DAI}

Destroy dai and corresponding debt tokens

mend wad_{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.

shut id_{urn} = **do**

Update debt unit and unprocessed fee revenue

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

$chi_1 \leftarrow \text{drip } id_{\text{ilk}}$

Reclaim all outstanding dai

$art_0 \leftarrow \text{look}(\text{vat} . \text{urns} . \text{ix } id_{\text{urn}} . \text{art})$

wipe id_{urn} ($art_0 * \text{cast } chi_1$)

Reclaim all collateral

$ink_0 \leftarrow \text{look}(\text{vat} . \text{urns} . \text{ix } id_{\text{urn}} . \text{ink})$

```
    free  $id_{\text{urn}}$  ink0  
Nullify CDP record  
    vat . urns . at  $id_{\text{urn}}$  := 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 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)
  wut0 ← use (vat . vox . wut)
  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 wut0 < 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.

```

drip  $id_{ilk}$  = do
  Time stamp of previous drip
   $\rho_0 \leftarrow \text{look } (\text{vat} . \text{ilks} . ix\ id_{ilk} . \rho)$ 
  Current stability fee
   $\text{tax}_0 \leftarrow \text{look } (\text{vat} . \text{ilks} . ix\ id_{ilk} . \text{tax})$ 
  Current debt unit value
   $\text{chi}_0 \leftarrow \text{look } (\text{vat} . \text{ilks} . ix\ id_{ilk} . \text{chi})$ 
  Current total debt in debt unit
   $\text{rum}_0 \leftarrow \text{look } (\text{vat} . \text{ilks} . ix\ id_{ilk} . \text{rum})$ 
  Current time stamp
   $\text{era}_0 \leftarrow \text{use era}$ 
  let
    Time difference in seconds
     $\text{age} = \text{era}_0 - \rho_0$ 
    Value of debt unit increased according to stability fee
     $\text{chi}_1 = \text{chi}_0 * \text{tax}_0 \uparrow \uparrow \text{age}$ 
    Stability fee revenue denominated in new unit
     $\text{dew} = (\text{cast } (\text{chi}_1 - \text{chi}_0) :: \text{Wad}) * \text{rum}_0$ 
  Mint dai and internal debt tokens for marginal stability fee
  lend  $\text{dew}$ 
  Record time of update
   $\text{vat} . \text{ilks} . ix\ id_{ilk} . \rho := \text{era}_0$ 
  Record new debt unit
   $\text{vat} . \text{ilks} . ix\ id_{ilk} . \text{chi} := \text{chi}_1$ 
  Return the new debt unit
  return  $\text{chi}_1$ 

```

4.4 Price feed input

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

```
mark  $id_{\text{jar}}$  tag1 zzz1 = auth $ do  
  vat . jars . ix  $id_{\text{jar}}$  . tag := tag1  
  vat . jars . ix  $id_{\text{jar}}$  . zzz := zzz1
```

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

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

4.5 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.”

```
bite  $id_{\text{urn}}$  = do  
  Fail if CDP is not in the appropriate risk stage  
  want (feel  $id_{\text{urn}}$ ) ( $\equiv$  Panic)  
  Record the sender as the liquidation initiator  
   $id_{\text{cat}} \leftarrow \text{use sender}$   
  vat . urns . ix  $id_{\text{urn}}$  . cat := Just  $id_{\text{cat}}$   
  Apply liquidation penalty to debt  
   $id_{\text{ilk}} \leftarrow \text{look (vat . urns . ix } id_{\text{urn}} . \text{ilk)}$   
   $axe_0 \leftarrow \text{look (vat . ilks . ix } id_{\text{ilk}} . \text{axe)}$   
   $art_0 \leftarrow \text{look (vat . urns . ix } id_{\text{urn}} . \text{art)}$   
  let  $art_1 = art_0 * \text{cast } axe_0$   
  Update debt  
  vat . urns . ix  $id_{\text{urn}}$  . art :=  $art_1$ 
```

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.

```

grab  $id_{urn}$  = auth $ do
  Fail if CDP is not marked for liquidation
  want (feel  $id_{urn}$ ) ( $\equiv$  Grief)
   $ink_0 \leftarrow look(vat.urns.ix\ id_{urn}.ink)$ 
   $art_0 \leftarrow look(vat.urns.ix\ id_{urn}.art)$ 
   $id_{ilk} \leftarrow look(vat.urns.ix\ id_{urn}.ilk)$ 
   $id_{jar} \leftarrow look(vat.ilks.ix\ id_{ilk}.jar)$ 
  Update the debt unit and stability fee
   $chi_1 \leftarrow drip\ id_{ilk}$ 
  Denominate the debt in dai
  let con =  $art_0 * cast\ chi_1$ 
  Transfer debt to settler
  push  $id_{sin}$  Vow con
  Transfer collateral to settler
  push  $id_{jar}$  Vow  $ink_0$ 
  Nullify CDP's collateral and debt quantities
   $vat.urns.ix\ id_{urn}.ink := 0$ 
   $vat.urns.ix\ id_{urn}.art := 0$ 
  Decrease the CDP type's total debt quantity
  decrease (vat.ilks.ix  $id_{ilk}.rum$ )  $art_0$ 

```

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  $id_{urn}$  wadDAI = auth $ do
  Fail unless CDP is in liquidation
  want (feel  $id_{urn}$ ) ( $\equiv$  Dread)
  Forget the CDP's requester of liquidation
   $vat.urns.ix\ id_{urn}.cat := Nothing$ 
  Return some amount of excess auction gains
   $id_{vow} \leftarrow use\ sender$ 
   $id_{ilk} \leftarrow look(vat.urns.ix\ id_{urn}.ilk)$ 
   $id_{jar} \leftarrow look(vat.ilks.ix\ id_{ilk}.jar)$ 
  pull  $id_{jar}\ id_{vow}\ wad_{DAI}$ 
  Record the gains as the CDP's collateral
   $vat.urns.ix\ id_{urn}.ink := wad_{DAI}$ 

```

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

```
loot = auth $ do
```

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

```
wadDAI ← look (balance idDAI (Vault idDAI))
```

Transfer the entire dai vault balance to sender

```
idvow ← use sender
```

```
transfer idDAI wadDAI (Vault idDAI) Vow
```

4.6 Auctioning

```
flip idgem wadjam wadtab idurn = 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 DAI and SIN balances of the entity

`awe ← look (balance idDAI who)`

`woe ← look (balance idSIN who)`

We can burn at most the smallest of the two balances

`let x = min awe woe`

Transfer both DAI and SIN into the vow accounts

`transfer idDAI x who Vow`

`transfer idSIN x who Vow`

Burn both DAI and SIN

`burn idDAI x Vow`

`burn idSIN x Vow`

`kick = do`

Transfer unprocessed stability fee revenue to vow account

`loot`

Cancel fee revenue against bad debt; vow keeps either a DAI balance or a SIN balance.

`tidy Vow`

Assign any remaining revenue to the MKR-deflating fee auction

`transferAll idDAI Vow Flapper`

`flap`

Assign any remaining debt to the MKR-inflating debt auction

`transferAll idSIN Vow Flopper`

`flop`

4.8 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 idilk idjar = auth $ do`

`initialize (vat . ilks . at idilk) (defaultIlk idjar)`

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

```
frob how1 = auth $ do vat . vox . how := how1
```

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 idilk mat1 = auth $ do vat . ilks . ix idilk . mat := mat1
chop idilk axe1 = auth $ do vat . ilks . ix idilk . axe := axe1
cork idilk hat1 = auth $ do vat . ilks . ix idilk . hat := hat1
calm idilk lax1 = auth $ do vat . ilks . ix idilk . 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 idilk tax1 =
  auth $ do
    Apply the current stability fee to the internal debt unit
    drip idilk
    Change the stability fee
    vat . ilks . ix idilk . tax := tax1
```

4.9 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 idjar src wadgem =
  transfer idjar wadgem src (Vault idjar)
```

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 idjar dst wadgem =
  transfer idjar wadgem (Vault idjar) dst
```

4.10 Token manipulation

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

```
transfer idjar wad src dst =  
  Operate in the token's balance table  
  zoom (vat . jars . ix idjar . gem . 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 lend to create new DAI and debt tokens, and by the settler to create new MKR.

```
mint idjar wad0 dst =  
  zoom (vat . jars . ix idjar . gem) $ do  
    increase (balanceOf . ix dst) wad0
```

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.

```
burn idjar wad0 src =  
  zoom (vat . jars . ix idjar . gem) $ do  
    decrease (balanceOf . ix src) wad0
```

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 borrower; it is also used by drip to issue dai representing revenue from stability fees, which stays in the dai vault until collected.

```
lend wadDAI = do  
  mint idDAI wadDAI Joy  
  mint idSIN wadDAI Woe
```

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.

```
mend wadDAI = do  
  burn idDAI wadDAI (Vault idDAI)  
  burn idSIN wadDAI 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
 $\text{aver } x = \text{unless } x \text{ (throwError (AssertError ?act))}$
Assert that an indexed value is not present
 $\text{none } x = \text{preuse } x \gg= \lambda \text{case}$
 $\text{Nothing} \rightarrow \text{return } ()$
 $\text{Just } _ \rightarrow \text{throwError (AssertError ?act)}$
Assert that an indexed value is present
 $\text{look } f = \text{preuse } f \gg= \lambda \text{case}$
 $\text{Nothing} \rightarrow \text{throwError (AssertError ?act)}$
 $\text{Just } x \rightarrow \text{return } x$
Execute an act and assert a condition on its result
 $\text{want } m \text{ } p = m \gg= (\text{aver } . p)$

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

$\text{owns } id_{\text{urn}} \text{ } id_{\text{lad}} = \text{do}$
 $\text{want } (\text{look } (\text{vat} . \text{urns} . \text{ix } id_{\text{urn}} . \text{lad})) (\equiv id_{\text{lad}})$

We define $\text{auth } k$ as an act modifier that executes k only if the sender is authorized.

$\text{auth } \text{continue} = \text{do}$
 $s \leftarrow \text{use sender}$
 $\text{unless } (s \equiv \text{Account } id_{\text{god}}) (\text{throwError AuthError})$
 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
 (\cdot), ($\$$),
 Numeric operators
 ($+$), ($-$), ($*$), ($/$), (\uparrow), ($\uparrow\uparrow$), *div*,
 Utilities
all, \neg , *elem*, (\wedge),
 Constants
empty, \perp , *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` documentation¹.

¹Gabriel 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 type².

```

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)

```

²We 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  $e \Rightarrow$  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  $e \Rightarrow$  Read (Decimal  $e$ ) where  
  readsPrec  $n\ s = fmap (\lambda(x, y) \rightarrow (D\ x, y)) (readsPrec\ n\ s)  
instance HasResolution  $e \Rightarrow$  Show (Decimal  $e$ ) where  
  show (D  $x$ ) = show  $x$$ 
```

In the Num instance, we delegate everything except multiplication.

```
instance HasResolution  $e \Rightarrow$  Num (Decimal  $e$ ) where  
   $x@(D\ (MkFixed\ a)) * D\ (MkFixed\ b) =$   
    D (MkFixed (div ( $a * b + \text{div}\ (resolution\ x)\ 2$ )  
                  (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  $\Rightarrow$  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  $\uparrow$  (18 :: Integer)
instance HasResolution E36 where
  resolution _ = 10  $\uparrow$  (36 :: Integer)

```

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

```

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

```