



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
Rune Christensen
Rain Clever
}

with last update on March 12, 2017.



Contents

1	Introduction	5
1.1	Motivation	6
1.2	Limitations	7
1.3	Verification	7
I	Implementation	8
2	Types	10
2.1	Numeric types	10
2.2	Identifiers and addresses	11
2.3	Gem — collateral price feed entry	11
	jar — collateral token	11
	tag — market price of token	11
	zzz — expiration time of token price feed	11
2.4	ERC20 — token model	12
2.5	Ilk — CDP type	12
	jar — collateral token vault	12
	mat — liquidation ratio	12
	axe — liquidation penalty ratio	12
	hat — debt ceiling	12
	tax — stability fee	12
	lag — price feed limbo duration	12
	rho — time of debt unit adjustment	12
	din — total outstanding dai	12
	chi — dai value of debt unit	12
2.6	Urn — collateralized debt position (CDP)	13
	cat — address of liquidation initiator	13
	vow — address of liquidation contract	13
	lad — CDP owner	13

	ilk — CDP type	13
	art — debt denominated in debt unit	13
	jam — collateral denominated in debt unit	13
2.7	Vox — feedback mechanism data	13
	fix — market price of DAI denominated in SDR	13
	par — target price of DAI denominated in SDR	13
	how — sensitivity parameter	13
	way — rate of target price change	13
	tau — time of latest target update	13
2.8	Vat — CDP engine data	13
	joy — unprocessed stability fee revenue	13
	sin — bad debt from liquidated CDPs	13
2.9	System model	14
	era — current time	14
2.10	Default data	14
3	Acts	16
3.1	Assessment	17
	feel — identify CDP risk stage	17
3.2	Lending	19
	open — create CDP	19
	give — transfer CDP account	19
	lock — deposit collateral	19
	free — withdraw collateral	20
	draw — issue dai as debt	20
	wipe — repay debt and burn dai	21
	shut — wipe, free, and delete CDP	21
3.3	Adjustment	22
	prod — adjust target price and target rate	22
	drip — update debt unit and unprocessed fee revenue	23
3.4	Price feed input	24
	mark — update market price of collateral token	24
	tell — update market price of dai	24
3.5	Liquidation	24
	bite — mark for liquidation	24
	grab — take tokens for liquidation	25
	plop — finish liquidation returning profit	25
	heal — record bad debt as processed	25
	love — record stability fee revenue as processed	25
3.6	Governance	26
	form — create a new CDP type	26

	frob — set the sensitivity parameter	26
	chop — set liquidation penalty	26
	cork — set debt ceiling	26
	calm — set limbo duration	26
	cuff — set liquidation ratio	26
	crop — set stability fee	26
3.7	Vaults	26
	pull — transfer tokens to vault	26
	push — transfer tokens from vault	27
3.8	Token manipulation	27
	mint — inflate token	27
	burn — deflate token	27
4	Act framework	28
4.1	The Maker monad	28
4.2	Asserting	29
A	Prelude	30
B	Fixed point numbers with rounding	33

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 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 :: (Real a, Fractional b) => a -> 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_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 reality¹, 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)

```

¹We 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
  • lag :: 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 idgem = Ilk {
  • jar = idgem,
  • axe = Ray 1,
  • mat = Ray 1,
  • tax = Ray 1,
  • hat = Wad 0,
  • lag = Sec 0,
  • chi = Ray 1,
  • rum = Wad 0,
  • rho = Sec 0
}

```

emptyUrn :: Id Ilk → Address → Urn

emptyUrn *id_{ilk}* *id_{lad}* = Urn {

- vow = Nothing,
- cat = Nothing,
- lad = *id_{lad}*,
- ilk = *id_{ilk}*,
- art = Wad 0,
- jam = Wad 0

}

initialVat :: Ray → Vat

initialVat how₀ = Vat {

- vox = Vox {

- tau = 0,
- fix = Wad 1,
- par = Wad 1,
- how = how₀,
- way = Ray 1

},

- joy = Wad 0,
- sin = Wad 0,
- ilks = ∅,
- urns = ∅,
- gems = *singleton id_{DAI}* Gem {
- erc20 = ERC20 {• balanceOf = ∅},
- tag = Wad 0,
- zzz = 0

}

}

initialSystem :: Ray → System

initialSystem how₀ = System {

- vat = *initialVat* how₀,
- era = 0,
- sender = *id_{god}*,
- 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.

```
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 vow urn0 ≠ Nothing
    CDP liquidation in progress
    → Dread
  | view cat urn0 ≠ Nothing
    CDP liquidation triggered
    → Grief
  | pro < min
    CDP's collateralization below liquidation ratio
    → Panic
  | view zzz jar0 + view lag ilk0 < era0
    CDP type's price limbo exceeded limit
    → Panic
  | view zzz jar0 < era0
    CDP type's price feed in limbo
    → Worry
  | cap > view hat ilk0
    CDP type's debt ceiling exceeded
    → Anger
  | otherwise
    No problems
    → Pride
```

where

CDP's collateral value in SDR:

```
pro = view jam urn0 * view tag jar0
```

CDP type's total debt in SDR:

```
cap = view rum ilk0 * cast (view chi ilk0)
```

























CDP's debt in SDR:






```
con = view art urn0 * cast (view chi ilk0) * par0
```

Required collateral as per liquidation ratio:

```
min = con * cast (view mat ilk0)
```

Table 3.1: CDP acts in the five stages of risk

	give	shut	lock	wipe	free	draw	bite	grab	plop
Pride							—	—	—
Anger						—	—	—	—
Worry					—	—	—	—	—
Panic					—	—		—	—
Grief		—	—	—	—	—	—		—
Dread		—	—	—	—	—	—	—	
	decrease risk			increase risk			unwind risk		

-  allowed for anyone
-  allowed for owner unconditionally
-  allowed for owner if able to repay
-  allowed for owner if collateralized
-  allowed for settler contract

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

```

feel  $id_{urn} = \mathbf{do}$ 
  Adjust target price and target rate
  prod
  Update debt unit and unprocessed fee revenue
   $id_{ilk} \leftarrow \text{look}(\text{vat} . \text{urns} . ix\ id_{urn} . ilk)$ 
  drip  $id_{ilk}$ 
  Read parameters for risk analysis
   $era_0 \leftarrow \text{use}\ era$ 
   $par_0 \leftarrow \text{use}\ (\text{vat} . \text{vox} . par)$ 
   $urn_0 \leftarrow \text{look}(\text{vat} . \text{urns} . ix\ id_{urn})$ 
   $ilk_0 \leftarrow \text{look}(\text{vat} . \text{ilks} . ix\ (\text{view}\ ilk\ urn_0))$ 
   $jar_0 \leftarrow \text{look}(\text{vat} . \text{gems} . ix\ (\text{view}\ jar\ ilk_0))$ 
  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.

```
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  
   $id_{ilk} \leftarrow look$  ( $vat . urns . ix$   $id_{urn} . ilk$ )  
   $id_{gem} \leftarrow look$  ( $vat . ilks . ix$   $id_{ilk} . jar$ )  
  Transfer tokens from owner to collateral vault  
   $pull$   $id_{gem}$   $id_{lad}$   $wad_{gem}$   
  Record an increase in collateral  
   $increase$  ( $vat . urns . ix$   $id_{urn} . jam$ )  $wad_{gem}$ 
```

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  $id_{urn}$  . jam)  $wad_{gem}$ 
  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)
   $id_{gem} \leftarrow look$  (vat . ilks . ix  $id_{ilk}$  . jar)
  push  $id_{gem}$   $id_{lad}$   $wad_{gem}$ 

```

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 dai and transfer to CDP owner
  mint  $id_{DAI}$   $wad_{DAI}$ 
  push  $id_{DAI}$   $id_{lad}$   $wad_{DAI}$ 

```

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.

```

wipe  $id_{urn}$  wadDAI = 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])
  Update debt unit and unprocessed fee revenue
   $id_{ilk} \leftarrow look\ (vat.\ urns.\ ix\ id_{urn}.\ ilk)$ 
   $chi_1 \leftarrow drip\ id_{ilk}$ 
  Denominate dai amount in debt unit
  let wadchi = wadDAI / cast  $chi_1$ 
  Decrease CDP debt
  decrease (vat . urns . ix  $id_{urn}$  . art) wadchi
  Decrease total CDP type debt
  decrease (vat . ilks . ix  $id_{ilk}$  . rum) wadchi
  Transfer dai from CDP owner to dai vault
  pull  $id_{DAI}$   $id_{lad}$  wadDAI
  Destroy reclaimed dai
  burn  $id_{DAI}$  wadDAI

```

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
   $id_{ilk} \leftarrow look\ (vat.\ urns.\ ix\ id_{urn}.\ ilk)$ 
   $chi_1 \leftarrow drip\ id_{ilk}$ 
  Reclaim all outstanding dai
   $art_0 \leftarrow look\ (vat.\ urns.\ ix\ id_{urn}.\ art)$ 
  wipe  $id_{urn}$  ( $art_0 * cast\ chi_1$ )
  Reclaim all collateral
   $jam_0 \leftarrow look\ (vat.\ urns.\ ix\ id_{urn}.\ jam)$ 
  free  $id_{urn}$   $jam_0$ 
  Nullify CDP record
  vat . urns . at  $id_{urn}$  := 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.

```

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 unprocessed stability fee revenue
   $\text{joy}_0 \leftarrow \text{look } (\text{vat} . \text{joy})$ 
  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
     $\text{chi}_1 = \text{chi}_0 * \text{tax}_0 \uparrow \uparrow \text{age}$ 
    Denominate stability fee revenue in new unit
     $\text{joy}_1 = \text{joy}_0 + (\text{cast } (\text{chi}_1 - \text{chi}_0) :: \text{Wad}) * \text{rum}_0$ 
  Record time of update
   $\text{vat} . \text{ilks} . ix\ id_{ilk} . \rho := \text{era}_0$ 
  Record new debt unit
   $\text{vat} . \text{ilks} . ix\ id_{ilk} . \text{chi} := \text{chi}_1$ 
  Record fee revenue denominated in new debt unit
   $\text{vat} . \text{joy} := \text{joy}_1$ 
  Return the new debt unit
  return  $\text{chi}_1$ 

```

3.4 Price feed input

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

```
mark  $id_{gem}$  tag1 zzz1 = auth $ do  
  vat.gems.ix  $id_{gem}$ .tag := tag1  
  vat.gems.ix  $id_{gem}$ .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.fix := wadgem
```

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  $id_{urn}$ ) ( $\equiv$  Panic)  
  Record the sender as the liquidation initiator  
   $id_{cat} \leftarrow use\ sender$   
  vat.urns.ix  $id_{urn}$ .cat := Just  $id_{cat}$   
  Update debt unit  
   $id_{ilk} \leftarrow look\ (vat.urns.ix\ id_{urn}.ilk)$   
   $chi_1 \leftarrow drip\ id_{ilk}$   
  Apply liquidation penalty to debt  
   $art_0 \leftarrow look\ (vat.urns.ix\ id_{urn}.art)$   
   $axe_0 \leftarrow look\ (vat.ilks.ix\ id_{ilk}.axe)$   
  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  $id_{urn}$ ) ( $\equiv$  Grief)
  Record the sender as the CDP's settler
   $id_{vow} \leftarrow use\ sender$ 
   $vat.urns.ix\ id_{urn}.vow := Just\ id_{vow}$ 
  Forget the CDP's requester of liquidation
   $vat.urns.ix\ id_{urn}.cat := Nothing$ 
  Transfer the tokens to the settler
   $jam_0 \leftarrow look\ (vat.urns.ix\ id_{urn}.jam)$ 
   $id_{ilk} \leftarrow look\ (vat.urns.ix\ id_{urn}.ilk)$ 
   $id_{gem} \leftarrow look\ (vat.ilks.ix\ id_{ilk}.jar)$ 
  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  $id_{urn}$ ) ( $\equiv$  Dread)
  Forget the CDP's settler
   $vat.urns.ix\ id_{urn}.vow := Nothing$ 
  Return some amount of excess auction gains
   $id_{vow} \leftarrow use\ sender$ 
   $id_{ilk} \leftarrow look\ (vat.urns.ix\ id_{urn}.ilk)$ 
   $id_{gem} \leftarrow look\ (vat.ilks.ix\ id_{ilk}.jar)$ 
  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  $id_{ilk}$ ) (defaultIllk  $id_{gem}$ )
```

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  $id_{ilk}$  mat1 = auth $ do vat . ilks . ix  $id_{ilk}$  . mat := mat1  
chop  $id_{ilk}$  axe1 = auth $ do vat . ilks . ix  $id_{ilk}$  . axe := axe1  
cork  $id_{ilk}$  hat1 = auth $ do vat . ilks . ix  $id_{ilk}$  . hat := hat1  
calm  $id_{ilk}$  lag1 = auth $ do vat . ilks . ix  $id_{ilk}$  . lag := lag1
```

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

```
crop  $id_{ilk}$  tax1 =  
  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 := tax1
```

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  $id_{gem}$   $id_{lad}$  wadgem =  
  transfer  $id_{gem}$  wadgem (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  
     $balance \leftarrow look (ix\ src)$   
    aver ( $balance \geq wad$ )  
  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
 $\text{aver } x = \text{unless } x \text{ (throwError (AssertError ?act))}$
Assert that an indexed value is not present
 $\text{none } x = \text{preuse } x \gg= \lambda \text{case}$
 Nothing $\rightarrow \text{return } ()$
 Just $_ \rightarrow \text{throwError (AssertError ?act)}$
Assert that an indexed value is present
 $\text{look } f = \text{preuse } f \gg= \lambda \text{case}$
 Nothing $\rightarrow \text{throwError (AssertError ?act)}$
 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.

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

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

$\text{auth } \text{continue} = \text{do}$
 $s \leftarrow \text{use sender}$
 $\text{unless } (s \equiv id_{\text{god}}) \text{ (throwError AuthError)}$
 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

$(.)$, $(\$)$,

Numeric operators

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

Utilities

all, \neg , *elem*,

Constants

mempty, \perp , *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.

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

```

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 ( $div\ (a * b + div\ (resolution\ x)\ 2)$   
              (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.
   $\epsilon = 1 / \text{fromIntegral } (\text{resolution } (\perp :: \text{Fixed } a))$ 

```