



*presents the*

REFERENCE IMPLEMENTATION

*of the remarkable*

**DAI CREDIT SYSTEM**

issuing a diversely collateralized stablecoin

*with last update on March 9, 2017.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Reference implementation . . . . .	7
<b>I</b>	<b>Implementation</b>	<b>8</b>
<b>2</b>	<b>Preamble</b>	<b>9</b>
<b>3</b>	<b>Types</b>	<b>10</b>
3.1	Numeric types . . . . .	10
3.2	Identifiers and addresses . . . . .	11
3.3	Gem — ERC20 token model . . . . .	11
3.4	Jar — collateral type . . . . .	12
	gem — collateral token . . . . .	12
	tag — market price of token . . . . .	12
	zzz — expiration date of token price feed . . . . .	12
3.5	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 — timestamp of last drip . . . . .	12
	din — total outstanding dai . . . . .	12
	chi — price of debt coin for CDP type . . . . .	12
3.6	Urn — collateralized debt position (CDP) . . . . .	12
	cat — address of liquidation initiator . . . . .	12
	vow — address of liquidation contract . . . . .	12
	lad — DAI issuer / CDP owner . . . . .	12
	ilk — CDP type . . . . .	12
	art — debt denominated in debt units . . . . .	12

	jam — collateral denominated in debt units . . . . .	12
3.7	Vat — CDP engine . . . . .	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 — timestamp of last revaluation . . . . .	13
	joy — unprocessed stability fee revenue . . . . .	13
	sin — bad debt from liquidated CDPs . . . . .	13
3.8	System model . . . . .	13
	era — Current timestamp . . . . .	13
3.9	Default data . . . . .	14
<b>4</b>	<b>Acts</b>	<b>16</b>
4.1	Risk assessment . . . . .	17
	gaze — identify CDP risk stage . . . . .	17
4.2	Lending . . . . .	18
	open — create CDP account . . . . .	18
	lock — deposit collateral . . . . .	18
	free — withdraw collateral . . . . .	19
	draw — issue dai as debt . . . . .	19
	wipe — repay debt and burn dai . . . . .	20
	give — transfer CDP account . . . . .	20
	shut — wipe, free, and delete CDP . . . . .	20
4.3	Frequent adjustments . . . . .	22
	prod — perform revaluation and rate adjustment . . . . .	22
	drip — update value of debt unit . . . . .	23
4.4	Price feedback . . . . .	23
	mark — update market price of dai . . . . .	23
	tell — update market price of collateral token . . . . .	23
4.5	Liquidation and settlement . . . . .	24
	bite — mark CDP for liquidation . . . . .	24
	grab — take tokens to begin CDP liquidation . . . . .	24
	heal — process bad debt . . . . .	24
	love — process stability fee revenue . . . . .	25
4.6	Governance . . . . .	25
	form — create a new CDP type . . . . .	25
	frob — set the sensitivity parameter . . . . .	25
	chop — set liquidation penalty . . . . .	25
	cork — set debt ceiling . . . . .	25
	calm — set limbo duration . . . . .	25

	<i>cuff</i> — set liquidation ratio . . . . .	25
	<i>crop</i> — set stability fee . . . . .	26
4.7	Minting, burning, and transferring . . . . .	26
	<i>pull</i> — take tokens to vat . . . . .	26
	<i>push</i> — send tokens from vat . . . . .	26
	<i>mint</i> — increase supply . . . . .	26
	<i>burn</i> — decrease supply . . . . .	26
4.8	Test-related manipulation . . . . .	27
	<i>warp</i> — travel in time . . . . .	27
	<i>mine</i> — create new collateral token . . . . .	27
	<i>hand</i> — give collateral tokens to account . . . . .	27
	<i>sire</i> — create a new account . . . . .	27
4.9	Other stuff . . . . .	27
<b>5</b>	<b>Act framework</b>	<b>29</b>
5.1	Act descriptions . . . . .	29
5.2	The Maker monad . . . . .	30
5.3	Accessor aliases . . . . .	30
5.4	Asserting . . . . .	31
5.5	Modifiers . . . . .	31
	<i>auth</i> — authenticating actions . . . . .	31
<b>6</b>	<b>Testing</b>	<b>32</b>
<b>A</b>	<b>Prelude</b>	<b>34</b>
<b>B</b>	<b>Rounding fixed point numbers</b>	<b>37</b>

## List of Tables

4.1	Urn acts in the five stages of risk . . . . .	18
-----	---	----

# List of Figures

# 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 incentives in order to keep dai market value stable relative to SDR<sup>1</sup> in the short and medium term.

New dai enters the money supply when a borrower takes out a loan backed by an excess of collateral locked in Maker’s token vault. 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.

Maker’s knowledge of the market values of dai and the various tokens used as collateral comes from *price feeds*. Prices are used to continuously assess the risk of each CDP. If the value of a CDP’s collateral drops below a certain multiple of its debt, it is marked for liquidation, which triggers a decentralized auction mechanism.

Another token, MKR, is also controlled by Maker, acting as a “share” in the system itself. When a CDP liquidation fails to recover the full value of debt, Maker mints more MKR and auctions it out. Thus MKR is used to fund last resort market making. The value of the MKR token is based on the *stability fee* imposed on all dai loans: stability fee revenue goes toward buying MKR for burning.

This document is an executable technical specification of the exact workings of the Maker smart contracts.

---

<sup>1</sup>“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. In the long term, the value of dai may diverge from the value of SDR; whether in an inflationary or deflationary way will depend on market forces.

## 1.1 Reference implementation

The version of this system that will be deployed on the Ethereum blockchain is written in Solidity, which is a workable smart contract implementation language. This reference implementation is a precise 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 flexible and 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. **Type correctness.** 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. **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.

# Part I

## Implementation



# Chapter 2

## Preamble

We replace the default prelude module with our own. This brings in dependencies and hides unneeded symbols. Consult [Appendix A](#) to see exactly what is brought into scope.

```
module Maker where  
import Maker.Prelude  Fully import the Maker prelude  
import Prelude ()      Import nothing from Prelude  
import Maker.Decimal  
import Debug.Trace
```

# Chapter 3

## Types

### 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).

```
Define the distinct wad type for currency quantities
newtype Wad = Wad (Decimal E18)
  deriving (Ord, Eq, Num, Real, Fractional, RealFrac)

Define the distinct ray type for precise rate quantities
newtype Ray = Ray (Decimal E36)
  deriving (Ord, Eq, Num, Real, Fractional, RealFrac)
```

See Appendix B for details on how we modify Haskell’s decimal fixed point type to do more correct rounding for multiplication and division.

Haskell number types are not automatically converted, so in calculations that combine wads and rays, we convert explicitly with a *cast* function.

```
Convert via fractional  $n/m$  form.
 $cast :: (Real\ a, Fractional\ b) \Rightarrow a \rightarrow b$ 
 $cast = fromRational \circ toRational$ 
```

We also define a type for time durations in whole seconds.

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

```

instance Epsilon Wad where  $\epsilon = \text{Wad } \epsilon$ 
instance Epsilon Ray where  $\epsilon = \text{Ray } \epsilon$ 

```

## 3.2 Identifiers and addresses

There are several kinds of identifiers used in the system, and we can use types to distinguish them.

The type parameter  $a$  creates distinct types.  
 For example, `Id Foo` and `Id Bar` are incompatible.

```

data Id  $a = \text{Id String}$ 
deriving (Show, Eq, Ord)

```

We define another type for representing Ethereum account addresses.

```

data Address = Address String
deriving (Ord, Eq, Show)

```

We also have three predefined entity identifiers.

```

The DAI token address
 $id_{\text{DAI}} = \text{Id "DAI"}$ 

The CDP engine address
 $id_{\text{vat}} = \text{Address "VAT"}$ 

The account with ultimate authority
 $\diamond$  Kludge until authority is modelled
 $id_{\text{god}} = \text{Address "GOD"}$ 

The address of the test driver
 $id_{\text{toy}} = \text{Address "TOY"}$ 

```

This section introduces the records stored by the Maker system.

## 3.3 Gem — ERC20 token model

```

data Gem = Gem {
  gemTotalSupply :: Wad,
  gemBalanceOf   :: Map Address      Wad,
  gemAllowance   :: Map (Address, Address) Wad
} deriving (Eq, Show)

```

### 3.4 Jar — collateral type

```
data Jar = Jar {  
  jarGem :: Gem,   Collateral token  
  jarTag  :: Wad,   Market price  
  jarZzz  :: Sec    Price expiration  
} deriving (Eq, Show)
```

### 3.5 Ilk — CDP type

```
data Ilk = Ilk {  
  ilkJar  :: Id Jar,   Collateral vault  
  ilkAxe  :: Ray,      Liquidation penalty  
  ilkHat  :: Wad,      Debt ceiling  
  ilkMat  :: Ray,      Liquidation ratio  
  ilkTax  :: Ray,      Stability fee  
  ilkLag  :: Sec,      Limbo duration  
  ilkRho  :: Sec,      Last dripped  
  ilkRum  :: Wad,      Total debt in debt unit  
  ilkChi  :: Ray       Value of debt unit  
} deriving (Eq, Show)
```

### 3.6 Urn — collateralized debt position (CDP)

```
data Urn = Urn {  
  urnCat  :: Maybe Address, Address of liquidation initiator  
  urnVow  :: Maybe Address, Address of liquidation contract  
  urnLad  :: Address,       Issuer  
  urnIlk  :: Id Ilk,        CDP type  
  urnArt  :: Wad,           Outstanding debt in debt units  
  urnJam  :: Wad            Collateral amount in debt units  
} deriving (Eq, Show)
```

### 3.7 Vat — CDP engine

```
data Vat = Vat {  
  vatFix   :: Wad,           Market price  
  vatHow   :: Ray,           Sensitivity  
  vatPar   :: Wad,           Target price  
  vatWay   :: Ray,           Target rate  
  vatTau   :: Sec,           Last prodded  
  vatJoy   :: Wad,           Unprocessed stability fees  
  vatSin   :: Wad,           Bad debt from liquidated CDPs  
  vatJars  :: Map (Id Jar) Jar, Collateral tokens  
  vatIlks  :: Map (Id Ilk) Ilk,  CDP types  
  vatUrns  :: Map (Id Urn) Urn   CDPs  
} deriving (Eq, Show)
```

### 3.8 System model

```
data System = System {  
  systemVat      :: Vat,      Root Maker entity  
  systemEra      :: Sec,      Current time stamp  
  systemSender   :: Address,   Sender of current act  
  systemAccounts :: [Address]  For test suites  
} deriving (Eq, Show)
```

## Lens fields

```
makeFields '' Gem  
makeFields '' Jar  
makeFields '' Ilk  
makeFields '' Urn  
makeFields '' Vat  
makeFields '' System
```

### 3.9 Default data

```

defaultIlk :: Id Jar → Ilk
defaultIlk idjar = Ilk {
  ilkJar   = idjar,
  ilkAxe  = Ray 1,
  ilkMat  = Ray 1,
  ilkTax  = Ray 1,
  ilkHat  = Wad 0,
  ilkLag  = Sec 0,
  ilkChi  = Ray 1,
  ilkRum  = Wad 0,
  ilkRho  = Sec 0
}

```

```

defaultUrn :: Id Ilk → Address → Urn
defaultUrn idilk idlad = Urn {
  urnVow = Nothing,
  urnCat = Nothing,
  urnLad = idlad,
  urnIlk = idilk,
  urnArt = Wad 0,
  urnJam = Wad 0
}

```

```

initialVat :: Ray → Vat
initialVat how0 = Vat {
  vatTau   = 0,
  vatFix   = Wad 1,
  vatPar   = Wad 1,
  vatHow   = how0,
  vatWay   = Ray 1,
  vatJoy   = Wad 0,
  vatSin   = Wad 0,
  vatIlks  = ∅,
  vatUrns  = ∅,
  vatJars  =
    singleton idDAI Jar {
      jarGem = Gem {

```

```

      gemTotalSupply = 0,
      gemBalanceOf  =  $\emptyset$ ,
      gemAllowance  =  $\emptyset$ 
    },
    jarTag = Wad 0,
    jarZzz = 0
  }
}

```

```

initialSystem :: Ray → System
initialSystem how0 = System {
  systemVat      = initialVat how0,
  systemEra      = 0,
  systemSender   = idgod,
  systemAccounts = mempty
}

```

# Chapter 4

## Acts

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

For details on the underlying “Maker monad,” which specifies how the act definitions behave with regard to state and rollback thereof, see [chapter 5](#).



## 4.1 Risk assessment

We divide an urn's situation into five stages of risk. Table 4.1 shows which acts each stage allows. The stages are naturally ordered from more to less risky.

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

First we define a pure function *analyze* that determines an urn's stage.

```
analyze era0 par0 urn0 ilk0 jar0 =
  if
    Undergoing liquidation?
      | view vow urn0 ≠ Nothing → Dread
    Liquidation triggered?
      | view cat urn0 ≠ Nothing → Grief
    Undercollateralized?
      | pro < min → Panic
    Price feed expired?
      | era0 > view zzz jar0 + view lag ilk0 → Panic
    Price feed in limbo?
      | view zzz jar0 < era0 → Worry
    Debt ceiling reached?
      | cap > view hat ilk0 → Anger
    Safely overcollateralized
      | otherwise → 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)) :: Wad
    CDP's debt in SDR:
      con = view art urn0 * cast (view chi ilk0) * par0
    Required collateral as per liquidation ratio:
      min = con * view mat ilk0
```

Table 4.1: Urn acts in the five stages of risk

	give	shut	lock	wipe	free	draw	bite	grab	plop	
Pride	•	•	•	•	•	•				overcollateralized
Anger	•	•	•	•	•					debt ceiling reached
Worry	•	•	•	•						price feed in limbo
Panic	•	•	•	•			•			undercollateralized
Grief	•							•		liquidation initiated
Dread	•								•	liquidation in progress

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

```

gaze  $id_{urn}$  = do
  Perform dai revaluation and rate adjustment
  prod
  Update price of specific debt unit
   $id_{ilk} \leftarrow look\ (vat \circ urns \circ ix\ id_{urn} \circ ilk)$ 
  drip  $id_{ilk}$ 
  Read parameters for risk analysis
   $era_0 \leftarrow use\ era$ 
   $par_0 \leftarrow use\ (vat \circ par)$ 
   $urn_0 \leftarrow look\ (vat \circ urns \circ ix\ id_{urn})$ 
   $ilk_0 \leftarrow look\ (vat \circ ilks \circ ix\ (view\ ilk\ urn_0))$ 
   $jar_0 \leftarrow look\ (vat \circ jars \circ ix\ (view\ jar\ ilk_0))$ 
  Return risk stage of CDP
  return (analyze  $era_0\ par_0\ urn_0\ ilk_0\ jar_0$ )

```

## 4.2 Lending

```

open  $id_{urn}\ id_{ilk}$  =
  do
     $id_{lad} \leftarrow use\ sender$ 
     $vat \circ urns \circ at\ id_{urn} \text{ ?= } defaultUrn\ id_{ilk}\ id_{lad}$ 

```

```

lock  $id_{urn}\ x$  = do
  Ensure CDP exists; identify collateral type

```

$id_{ilk} \leftarrow look (vat \circ urns \circ ix \ id_{urn} \circ ilk)$   
 $id_{jar} \leftarrow look (vat \circ ilks \circ ix \ id_{ilk} \circ jar)$

Record an increase in collateral

$vat \circ urns \circ ix \ id_{urn} \circ jam \ += \ x$

Take sender's tokens

$id_{lad} \leftarrow use \ sender$

$pull \ id_{jar} \ id_{lad} \ x$

**free  $id_{urn} \ wad_{gem} = do$**

Fail if sender is not the CDP owner

$id_{sender} \leftarrow use \ sender$

$id_{lad} \leftarrow look (vat \circ urns \circ ix \ id_{urn} \circ lad)$

$aver (id_{sender} \equiv id_{lad})$

Decrease the collateral amount

$vat \circ urns \circ ix \ id_{urn} \circ jam \ -= \ wad_{gem}$

Roll back if undercollateralized

$gaze \ id_{urn} \gg= aver \circ (\equiv \text{Pride})$

Send the collateral to the CDP owner

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

$id_{jar} \leftarrow look (vat \circ ilks \circ ix \ id_{ilk} \circ jar)$

$push \ id_{jar} \ id_{lad} \ wad_{gem}$

**draw  $id_{urn} \ wad_{DAI} = do$**

Fail if sender is not the CDP owner

$id_{sender} \leftarrow use \ sender$

$id_{lad} \leftarrow look (vat \circ urns \circ ix \ id_{urn} \circ lad)$

$aver (id_{sender} \equiv id_{lad})$

Update value of debt unit

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

$chi_1 \leftarrow drip \ id_{ilk}$

Denominate draw amount in debt unit

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

Increase debt

$vat \circ urns \circ ix \ id_{urn} \circ art \ += \ wad_{chi}$

Roll back unless overcollateralized

$\text{gaze } id_{\text{urn}} \gg \text{aver} \circ (\equiv \text{Pride})$

Mint dai and send to the CDP owner

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

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

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

Fail if sender is not the CDP owner

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

$id_{\text{lad}} \leftarrow \text{look } (\text{vat} \circ \text{urns} \circ ix \ id_{\text{urn}} \circ \text{lad})$

$\text{aver } (id_{\text{sender}} \equiv id_{\text{lad}})$

Update value of debt unit

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

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

Roll back unless overcollateralized

$\text{gaze } id_{\text{urn}} \gg \text{aver} \circ (\equiv \text{Pride})$

Denominate dai amount in debt unit

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

Reduce debt

$\text{vat} \circ \text{urns} \circ ix \ id_{\text{urn}} \circ \text{art} -= \text{wad}_{\text{chi}}$

Take dai from CDP owner, or roll back

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

Destroy dai

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

$\text{give } id_{\text{urn}} id_{\text{lad}} = \text{do}$

$x \leftarrow \text{look } (\text{vat} \circ \text{urns} \circ ix \ id_{\text{urn}} \circ \text{lad})$

$y \leftarrow \text{use sender}$

$\text{aver } (x \equiv y)$

$\text{vat} \circ \text{urns} \circ ix \ id_{\text{urn}} \circ \text{lad} := id_{\text{lad}}$

$\text{shut } id_{\text{urn}} = \text{do}$

Update value of debt unit

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

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

Attempt to repay all the CDP's outstanding dai

```

 $\mathbf{art}_0 \leftarrow \text{look} (\mathbf{vat} \circ \mathbf{urns} \circ \mathbf{ix} \ id_{\mathbf{urn}} \circ \mathbf{art})$ 
 $\text{wipe } id_{\mathbf{urn}} (\mathbf{art}_0 * \text{cast } \mathbf{chi}_1)$ 

```

Reclaim all the collateral

```

 $\mathbf{jam0} \leftarrow \text{look} (\mathbf{vat} \circ \mathbf{urns} \circ \mathbf{ix} \ id_{\mathbf{urn}} \circ \mathbf{jam})$ 
 $\text{free } id_{\mathbf{urn}} \mathbf{jam0}$ 

```

Nullify the CDP

```

 $\mathbf{vat} \circ \mathbf{urns} \circ \mathbf{at} \ id_{\mathbf{urn}} := \text{Nothing}$ 

```

### 4.3 Frequent adjustments

```
prod = do
  era0 ← use era
  tau0 ← use (vat ∘ tau)
  fix0 ← use (vat ∘ fix)
  par0 ← use (vat ∘ par)
  how0 ← use (vat ∘ how)
  way0 ← use (vat ∘ 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)
  vat ∘ par := par1
  vat ∘ way := way1
  vat ∘ 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)
```

```

drip  $id_{ilk}$  = do
  Current time stamp
   $era_0 \leftarrow use\ era$ 
  Time stamp of previous drip
   $\rho_0 \leftarrow look\ (vat \circ ilks \circ ix\ id_{ilk} \circ \rho)$ 
  Current stability fee
   $tax_0 \leftarrow look\ (vat \circ ilks \circ ix\ id_{ilk} \circ tax)$ 
  Current value of debt unit
   $chi_0 \leftarrow look\ (vat \circ ilks \circ ix\ id_{ilk} \circ chi)$ 
  Current total debt in debt units
   $rum0 \leftarrow look\ (vat \circ ilks \circ ix\ id_{ilk} \circ rum)$ 
  Current unprocessed stability fee revenue
   $joy_0 \leftarrow look\ (vat \circ ilks \circ ix\ id_{ilk} \circ joy)$ 
  let
     $age = era_0 - \rho_0$ 
     $chi_1 = chi_0 * tax_0 \uparrow\uparrow age$ 
     $joy_1 = joy_0 + (cast\ (chi_1 - chi_0) :: Wad) * rum0$ 
   $vat \circ ilks \circ ix\ id_{ilk} \circ chi := chi_1$ 
   $vat \circ ilks \circ ix\ id_{ilk} \circ \rho := era_0$ 
   $vat \circ ilks \circ ix\ id_{ilk} \circ joy := joy_1$ 
  return  $chi_1$ 

```

## 4.4 Price feedback

```

mark  $id_{jar}$  tag1 zzz1 =
  auth $ do
     $vat \circ jars \circ ix\ id_{jar} \circ tag := tag_1$ 
     $vat \circ jars \circ ix\ id_{jar} \circ zzz := zzz_1$ 

tell  $x =$ 
  auth $ do
     $vat \circ fix := x$ 

```

## 4.5 Liquidation and settlement

```

bite  $id_{urn}$  = do
  Fail if CDP is not in need of liquidation
  gaze  $id_{urn} \gg=$  aver  $\circ (\equiv \text{Panic})$ 
  Record the sender as the liquidation initiator
   $id_{cat} \leftarrow use\ sender$ 
   $vat \circ urns \circ ix\ id_{urn} \circ cat := id_{cat}$ 
  Read current debt
   $art_0 \leftarrow look\ (vat \circ urns \circ ix\ id_{urn} \circ art)$ 
  Update value of debt unit
   $id_{ilk} \leftarrow look\ (vat \circ urns \circ ix\ id_{urn} \circ ilk)$ 
   $chi_1 \leftarrow drip\ id_{ilk}$ 
  Read liquidation penalty ratio
   $id_{ilk} \leftarrow look\ (vat \circ urns \circ ix\ id_{urn} \circ ilk)$ 
   $axe_0 \leftarrow look\ (vat \circ ilks \circ ix\ id_{ilk} \circ axe)$ 
  Apply liquidation penalty to debt
  let  $art_1 = art_0 * axe_0$ 
  Update CDP debt
   $vat \circ urns \circ ix\ id_{urn} \circ art := art_1$ 
  Record as bad debt
   $sin += art_1 * chi_1$ 

```

```

grab  $id_{urn}$  =
  auth $ do
    Fail if CDP is not marked for liquidation
    gaze  $id_{urn} \gg=$  aver  $\circ (\equiv \text{Grief})$ 
    Record the sender as the CDP's settler
     $id_{vow} \leftarrow use\ sender$ 
     $vat \circ urns \circ ix\ id_{urn} \circ vow := Just\ id_{vow}$ 
    Clear the CDP's requester of liquidation
     $vat \circ urns \circ ix\ id_{urn} \circ cat := Nothing$ 

```

```

heal wadDAI =

```



```

auth $ do
  vat ∘ sin == wadDAI

```

```

love wadDAI =
  auth $ do
    vat ∘ joy == wadDAI

```

## 4.6 Governance

```

form idilk idjar =
  auth $ do
    vat ∘ ilks ∘ at idilk ?= defaultIlk idjar

```

```

frob how' =
  auth $ do
    vat ∘ how := how'

```

```

chop idilk axe1 =
  auth $ do
    vat ∘ ilks ∘ ix idilk ∘ axe := axe1

```

```

cork idilk hat1 =
  auth $ do
    vat ∘ ilks ∘ ix idilk ∘ hat := hat1

```

```

calm idilk lag1 =
  auth $ do
    vat ∘ ilks ∘ ix idilk ∘ lag := lag1

```

```

cuff  $id_{ilk}$   $mat_1$  =
  auth $ do
     $vat \circ ilks \circ ix \ id_{ilk} \circ mat := mat_1$ 

```

```

crop  $id_{ilk}$   $tax_1$  =
  auth $ do
    drip  $id_{ilk}$ 
     $vat \circ ilks \circ ix \ id_{ilk} \circ tax := tax_1$ 

```

## 4.7 Minting, burning, and transferring

```

pull  $id_{jar}$   $id_{lad}$   $w$  = do
   $g \leftarrow look \ (jarAt \ id_{jar} \circ gem)$ 
   $g' \leftarrow transferFrom \ id_{lad} \ id_{vat} \ w \ g$ 
   $jarAt \ id_{jar} \circ gem := g'$ 

```

```

push  $id_{jar}$   $id_{lad}$   $w$  = do
   $g \leftarrow look \ (jarAt \ id_{jar} \circ gem)$ 
   $g' \leftarrow transferFrom \ id_{vat} \ id_{lad} \ w \ g$ 
   $jarAt \ id_{jar} \circ gem := g'$ 

```

```

mint  $id_{jar}$   $wad_0$  = do
   $jarAt \ id_{jar} \circ gem \circ totalSupply \quad += wad_0$ 
   $jarAt \ id_{jar} \circ gem \circ balanceOf \circ ix \ id_{vat} \quad += wad_0$ 

```

```

burn  $id_{jar}$   $wad_0$  = do
   $jarAt \ id_{jar} \circ gem \circ totalSupply \quad -= wad_0$ 
   $jarAt \ id_{jar} \circ gem \circ balanceOf \circ ix \ id_{vat} \quad -= wad_0$ 

```

## 4.8 Test-related manipulation

```
warp  $t$  =
  auth $ do
    era +=  $t$ 
```

```
mine  $id_{\text{jar}}$  = do
  vat  $\circ$  jars  $\circ$  at  $id_{\text{jar}}$  ?= Jar {
    jarGem = Gem {
      gemTotalSupply = 10000000000000,
      gemBalanceOf   = singleton  $id_{\text{toy}}$  10000000000000,
      gemAllowance   =  $\emptyset$ 
    },
    jarTag = Wad 0,
    jarZzz = 0
  }
```

```
hand  $dst$  w  $id_{\text{jar}}$  = do
   $g \leftarrow \text{look } (\text{jarAt } id_{\text{jar}} \circ \text{gem})$ 
   $g' \leftarrow \text{transferFrom } id_{\text{toy}} \text{ } dst \text{ } w \text{ } g$ 
  jarAt  $id_{\text{jar}} \circ \text{gem} := g'$ 
```

```
sire lad = do
  accounts %= (lad:)
```

## 4.9 Other stuff

```
perform :: Act  $\rightarrow$  Maker ()
perform  $x$  =
  let ?act =  $x$  in case  $x$  of
    Form  $id$  jar    $\rightarrow$  form  $id$  jar
    Mark jar tag zzz  $\rightarrow$  mark jar tag zzz
    Open  $id$  ilk    $\rightarrow$  open  $id$  ilk
```

```

Tell wad      → tell wad
Frob ray      → frob ray
Prod          → prod
Warp t        → warp t
Give urn lad  → give urn lad
Pull jar lad wad → pull jar lad wad
Lock urn wad  → lock urn wad
Mine id       → mine id
Hand lad wad jar → hand lad wad jar
Sire lad      → sire lad

be :: Address → Act → Maker ()
be who x = do
  old ← use sender
  sender := who
  y ← perform x
  sender := old
  return y

transferFrom
  :: (MonadError Error m)
  ⇒ Address → Address → Wad
  → Gem → m Gem

transferFrom src dst wad gem =
  case view (balanceOf ∘ at src) gem of
    Nothing →
      throwError ∘ AssertError $
        "transferFrom " <> show (src, dst, wad)
    Just balance → do
      aver (balance ≥ wad)
      return $ gem &~ do
        balanceOf ∘ ix src -= wad
        balanceOf ∘ at dst %=
          (λcase
            Nothing → Just wad
            Just x   → Just (wad + x))

```

# Chapter 5

## Act framework

### 5.1 Act descriptions

We define the Maker act vocabulary as a data type.

```
data Act =  
  Bite (Id Urn)  
| Draw (Id Urn) Wad  
| Form (Id Ilk) (Id Jar)  
| Free (Id Urn) Wad  
| Frob Ray  
| Give (Id Urn) Address  
| Grab (Id Urn)  
| Heal Wad  
| Lock (Id Urn) Wad  
| Love Wad  
| Mark (Id Jar) Wad      Sec  
| Open (Id Urn) (Id Ilk)  
| Prod  
| Pull (Id Jar) Address Wad  
| Shut (Id Urn)  
| Tell Wad  
| Warp Sec  
| Wipe (Id Urn) Wad  
| Mine (Id Jar)  
| Hand Address Wad      (Id Jar)  
| Sire Address
```

```

Test acts
  | Addr Address
deriving (Eq, Show)

```

Acts can fail. We divide the failure modes into general assertion failures and authentication failures.

```

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

```

## 5.2 The Maker monad

The reader does not need any abstract understanding of monads to understand the code. What they give us is a nice syntax—the **do** notation—for expressing exceptions and state in a way that is still purely functional.

```

newtype Maker a =
  Maker (StateT System (Except Error) a)
deriving
  (Functor, Applicative, Monad,
   MonadError Error,
   MonadState System)

```

```

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

```

The following instance makes the mutable state also available as read-only state.

## 5.3 Accessor aliases

```

ilkAt id = vat ∘ ilks ∘ ix id
urnAt id = vat ∘ urns ∘ ix id
jarAt id = vat ∘ jars ∘ ix id

```

## 5.4 Asserting

```
aver :: Fails  $m \Rightarrow$  Bool  $\rightarrow m ()$   
aver  $x = unless\ x\ (throwError\ (AssertError\ "aver"))$ 
```

```
look  $f = preuse\ f\ \gg\ \lambda case$   
  Nothing  $\rightarrow throwError\ (AssertError\ "look")$   
  Just  $x \rightarrow return\ x$ 
```

## 5.5 Modifiers

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

# Chapter 6

## Testing

Sketches for property stuff...

```
data Parameter =  
  Fix | Par | Way
```

*maintains*

```
:: Eq a => Lens' System a -> Maker ()  
  -> System -> Bool
```

*maintains*  $p = \lambda m \text{ sys}_0 \rightarrow$

```
case exec sys0 m of
```

On success, data must be compared for equality

```
  Right sys1 -> view p sys0 ≡ view p sys1
```

On rollback, data is maintained by definition

```
  Left _ -> True
```

*changesOnly*

```
:: Lens' System a -> Maker ()  
  -> System -> Bool
```

*changesOnly*  $p = \lambda m \text{ sys}_0 \rightarrow$

```
case exec sys0 m of
```

On success, equalize  $p$  and compare

```
  Right sys1 -> set p (view p sys1) sys0 ≡ sys1
```

On rollback, data is maintained by definition

```
  Left _ -> True
```

*also* :: Lens'  $s$   $a \rightarrow$  Lens'  $s$   $b \rightarrow$  Lens'  $s$   $(a, b)$

*also*  $f$   $g = \text{lens getter setter}$



**where**

*getter*  $x = (\text{view } f \ x, \text{view } g \ x)$

*setter*  $x \ (a, b) = \text{set } f \ a \ (\text{set } g \ b \ x)$

*keeps*  $:: \text{Parameter} \rightarrow \text{Maker } () \rightarrow \text{System} \rightarrow \text{Bool}$

*keeps* **Fix** = *maintains* (**vat**  $\circ$  **fix**)

*keeps* **Par** = *maintains* (**vat**  $\circ$  **par**)

*keeps* **Way** = *maintains* (**vat**  $\circ$  **way**)

Thus:

*foo* **sys**<sub>0</sub> = *all* ( $\lambda f \rightarrow f \ \text{sys}_0$ )  
  [*changesOnly* ((**vat**  $\circ$  **par**) ‘*also*’  
                  (**vat**  $\circ$  **way**))  
  (*perform* **Prod**)]

# Appendix A

## Prelude

```
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  
 ( $\circ$ ), ( $\$$ ),  
 Numeric operators  
 (+), (−), (\*), (/), ( $\uparrow$ ), ( $\uparrow\uparrow$ ), *div*,  
 Utilities  
*all*,  
 Constants  
*mempty*,  $\perp$ , *otherwise*)

We use a typical composition of monad transformers from the `mtl` library to structure stateful actions. See section 5.2 (*The Maker monad*).

```

import Control.Monad.State as X (
  MonadState,    Type class of monads with state
  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.Reader as X (
  MonadReader,   Type class of monads with “environments”
  ask,           Reads the environment in a do block
  local)         Runs a sub-computation with a modified environment

import Control.Monad.Writer as X (
  MonadWriter,   Type class of monads that emit logs
  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  $a \circ b \circ c$  denotes a nested accessor much like `a.b.c` in C-style languages; for more details, consult lens documentation<sup>1</sup>.

```
import Control.Lens as X (
    Lens',
    lens,
    makeFields,    Defines lenses for record fields
    set,           Writes a lens
    use, preuse,
    view, preview, Reads a lens in a do block
    (&~),          Lets us use a do block with setters  $\diamond$  Get rid of this.
    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
```

Where the Solidity code uses `mapping`, we use Haskell’s regular tree-based map type<sup>2</sup>.

```
import Data.Map as X (
    Map,      Type constructor for mappings
    Ø,        Polymorphic empty mapping
    singleton) Creates a mapping with a single key–value pair
```

For sequences of log entries, we use a sequence structure which has better time complexity than regular lists.

```
import          Data.Sequence as X (Seq)
import qualified Data.Sequence as Sequence
```

Some less interesting imports are omitted from this document.

---

<sup>1</sup>Gabriel Gonzalez’s 2013 article *Program imperatively using Haskell* is a good introduction.

<sup>2</sup>We assume the axiom that Keccak hash collisions are impossible.

# Appendix B

## Rounding fixed point numbers

This somewhat arcane-looking code implements a wrapper around the base library's decimal fixed point type, only using a more precise rounding method for the  $x * y$  and  $x / y$  operations.

```
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  $\epsilon$ ). 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.
   $\epsilon = 1 / \text{fromIntegral } (\text{resolution } (\perp :: \text{Fixed } a))$ 

```