



presents the

REFERENCE IMPLEMENTATION

of the remarkable

DAI CREDIT SYSTEM

issuing a diversely collateralized stablecoin

with last update on February 28, 2017.

Contents

1	Introduction	5
1.1	Reference implementation	6
I	Implementation	7
2	Preamble	9
3	Types	11
3.1	Numeric types	11
3.1.1	Epsilon values	12
3.2	Identifier type	12
3.3	Structures	13
3.3.1	GEM — Collateral token model	13
3.3.2	JAR — Collateral token	13
3.3.3	ILK — CDP type	14
3.3.4	URN — collateralized debt position (CDP)	14
3.3.5	VAT — Dai creditor	15
3.3.6	System model	16
3.3.7	Default data	16
4	Act framework	19
4.1	Act descriptions	19
4.2	Constraints	21
4.3	Accessor aliases	21
4.4	Logging and asserting	21
4.5	Modifiers	21
	note — logging actions	21
	auth — authenticating actions	22
5	Acts	23

5.1	Risk assessment	24
	gaze — URN: identify CDP risk stage	24
5.2	Lending	25
	open — VAT: create CDP account	25
	lock — URN: deposit collateral	25
	free — URN: withdraw collateral	25
	draw — URN: issue DAI as debt	26
	wipe — URN: repay debt and burn DAI	26
	give — URN: transfer CDP account	27
	shut — URN: wipe, free, and delete CDP	27
5.3	Frequent adjustments	28
	prod — VAT: perform revaluation and rate adjustment	28
	drip — ILK: update stability fee accumulator	28
	poke — URN: add stability fee to CDP debt	29
5.4	Governance	29
	form — VAT: create a new CDP type	29
	frob — VAT: set the sensitivity parameter	30
5.5	Price feedback	30
	mark — VAT: update market price of DAI	30
	tell — GEM: update market price of collateral token	30
5.6	Liquidation and settlement	30
	bite — URN: trigger CDP liquidation	30
	grab — URN: promise that liquidation is in process	31
	heal — VAT: process bad debt	31
	loot — VAT: process stability fee revenue	31
5.7	Minting, burning, and transferring	31
	pull — GEM: receive tokens to VAT	31
	push — GEM: send tokens from VAT	32
	mint — DAI: increase supply	32
	burn — DAI: decrease supply	32
5.8	Test-related manipulation	32
	warp — travel in time	32
5.9	System modelling	32
5.10	Other stuff	33
6	Testing	35
A	Act type signatures	37

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* 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.

Maker also controls the supply of another token, MKR. This token acts as a “share” in Maker. 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. On the other hand, Maker imposes a *stability fee* on all dai loans, and revenue from these fees goes toward buying MKR for burning.

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

* “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 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 a previously unwritten mapping and get back a value initialized with zeroed memory, whereas in Haskell we must explicitly describe default values. The state rollback behavior of failed actions is also in Haskell explicitly coded as part of the monad transformer stack.
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

```
module Maker where
```

We import types for the decimal fixed-point arithmetic which we use for amounts and rates.

```
import Data.Fixed
```

We rely on the `lens` library for defining and using accessors which otherwise tend to become long-winded in Haskell. Since our program has several nested records, this makes the code much clearer. There is no need to understand the theory behind lenses to understand this program. All the reader needs to know is that $a \circ b \circ c$ denotes a nested accessor much like `a.b.c` in C-style languages. The rest should be obvious from context.

```
import Control.Lens
```

We use a typical stack of monad transformers from the `mtl` library to structure state-modifying actions. Again, the reader does not need any abstract understanding of monads. They make our code clear and simple by enabling `do` blocks to express exceptions, state, and logging.

```
import Control.Monad.Except
  (MonadError, Except, throwError, runExcept)
import Control.Monad.Reader
  (MonadReader (..))
import Control.Monad.State
  (MonadState, StateT, execStateT, get, put)
import Control.Monad.Writer
  (MonadWriter, WriterT, runWriterT)
```

Some less interesting imports are omitted from this document.

Chapter 3

Types

3.1 Numeric types

Many Ethereum tokens (e.g. ETH, DAI, and MKR) are denominated with 18 decimals. That makes decimal fixed point with 18 digits of precision a natural choice for representing currency quantities. We call such quantities "wads" (as in "wad of cash").

For some quantities, such as the rate of deflation per second, we want as much precision as possible, so we use twice the number of decimals. We call such quantities "rays" (mnemonic "rate," but also imagine a very precisely aimed ray of light).

Phantom types encode precision at compile time.

data E18; **data** E36

Specify 10^{-18} as the precision of E18.

instance HasResolution E18 **where**

resolution _ = $10 \uparrow (18 :: \text{Integer})$

Specify 10^{-36} as the precision of E36.

instance HasResolution E36 **where**

resolution _ = $10 \uparrow (36 :: \text{Integer})$

Create the distinct WAD type for currency quantities.

newtype WAD = WAD (Fixed E18)

deriving (Ord, Eq, Num, Real, Fractional)

Create the distinct RAY type for precise rate quantities.

newtype RAY = RAY (Fixed E36)

deriving (Ord, Eq, Num, Real, Fractional)

In calculations where a WAD is multiplied by a RAY, for example in the deflation mechanism, we have to downcast in a way that loses precision. Haskell does not cast automatically, so unless you see the following *cast* function applied, you can assume that precision is unchanged.

```
cast :: (Real a, Fractional b) => a -> b
cast =
  Convert via fractional n/m form.
  fromRational o toRational
```

We also define a type for non-negative integers.

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

3.1.1 Epsilon values

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 => Epsilon (Fixed a) where
  The use of  $\perp$  is safe since resolution ignores the value.
   $\epsilon = 1 / \text{fromIntegral } (\text{resolution } (\perp :: \text{Fixed } a))$ 
instance Epsilon WAD where  $\epsilon = \text{WAD } \epsilon$ 
instance Epsilon RAY where  $\epsilon = \text{RAY } \epsilon$ 
```

3.2 Identifier type

There are several types of identifiers used in the system, and we can use Haskell's type system to distinguish them.

```
The type parameter is only used to create distinct types.
For example, Id Foo and Id Bar are incompatible.
data Id a = Id String
deriving (Show, Eq, Ord)
```

It turns out that we will in several places use mappings from IDs to the value type corresponding to that ID type, so we define an alias for such mapping types.

```
type IdMap a = Map (Id a) a
```

We also have three predefined entities:

```

    The DAI token address
idDAI = Id "Dai"

    The CDP engine address
idVAT = Id "Vat"

    The account with ultimate authority
idgod = Id "God"
```

3.3 Structures

[XXX: describe structures]

```
data LAD = LAD deriving (Eq, Show)
```

3.3.1 Gem — Collateral token model

```

data GEM =
  GEM {
    gemTotalSupply :: !WAD,
    gemBalanceOf   :: !(Map (Id LAD) WAD),
    gemAllowance   :: !(Map (Id LAD, Id LAD) WAD)
  } deriving (Eq, Read, Show)
makeFields '' GEM
```

3.3.2 Jar — Collateral token

```

data JAR = JAR {
  Collateral token
```

```

    jarGem :: !GEM,
Market price
    jarTag  :: !WAD,
Price expiration
    jarZzz  :: !NAT
} deriving (Eq, Show, Read)
makeFields '' JAR

```

3.3.3 Ilk — cdp type

```

data ILK = ILK {
  Collateral vault
    ilkJar  :: !(Id JAR),
  Liquidation penalty
    ilkAxe  :: !RAY,
  Debt ceiling
    ilkHat  :: !WAD,
  Liquidation ratio
    ilkMat  :: !RAY,
  Stability fee
    ilkTax  :: !RAY,
  Limbo duration
    ilkLag  :: !NAT,
  Last dripped
    ilkRho  :: !NAT,
  ???
    ilkCow  :: !RAY,
  Stability fee accumulator
    ilkBag  :: !(Map NAT RAY)
} deriving (Eq, Show)
makeFields '' ILK

```

3.3.4 Urn — collateralized debt position (cdp)

```

data URN = URN {

```

```

Address of biting cat
  urnCat :: !(Maybe (Id LAD)),
Address of liquidating vow
  urnVow :: !(Maybe (Id LAD)),
Issuer
  urnLad :: !(Id LAD),
CDP type
  urnIlk  :: !(Id ILK),
Outstanding dai debt
  urnCon :: !WAD,
Collateral amount
  urnPro :: !WAD,
Last poked
  urnPhi :: !NAT
} deriving (Eq, Show)
makeFields '' URN

```

3.3.5 Vat — Dai creditor

```

data VAT = VAT {
  Market price
    vatFix  :: !WAD,
  Sensitivity
    vatHow  :: !RAY,
  Target price
    vatPar  :: !WAD,
  Target rate
    vatWay  :: !RAY,
  Last prodded
    vatTau  :: !NAT,
  Unprocessed revenue from stability fees
    vatPie  :: !WAD,
  Bad debt from liquidated CDPs
    vatSin  :: !WAD,
  Collateral tokens

```

```

    vatJars :: !(IdMap JAR),
CDP types
    vatIlks :: !(IdMap ILK),
CDPs
    vatUrns :: !(IdMap URN)
} deriving (Eq, Show)
makeFields '' VAT

```

3.3.6 System model

```

data System =
  System {
    systemVat    :: VAT,
    systemEra    :: !NAT,
    systemLads   :: IdMap LAD,  System users
    systemSender :: Id LAD
  } deriving (Eq, Show)
makeFields '' System

```

3.3.7 Default data

```

defaultIlk :: Id JAR → ILK
defaultIlk idJAR = ILK {
  ilkJar   = idJAR,
  ilkAxe   = RAY 1,
  ilkMat   = RAY 1,
  ilkTax   = RAY 1,
  ilkHat   = WAD 0,
  ilkLag   = NAT 0,
  ilkBag   = ∅,
  ilkCow   = RAY 1,
  ilkRho   = NAT 0
}

defaultUrn :: Id ILK → Id LAD → URN
defaultUrn idILK idLAD = URN {

```



```

    urnVow = Nothing,
    urnCat  = Nothing,
    urnLad  = idLAD,
    urnIlk  = idILK,
    urnCon  = WAD 0,
    urnPro  = WAD 0,
    urnPhi  = NAT 0
}

```

```

initialVat :: RAY → VAT
initialVat HOW0 = VAT {
    vatTau    = 0,
    vatFix    = WAD 1,
    vatPar    = WAD 1,
    vatHow    = HOW0,
    vatWay    = RAY 1,
    vatPie    = WAD 0,
    vatSin    = WAD 0,
    vatIlks   = ∅,
    vatUrns   = ∅,
    vatJars   =
        singleton idDAI JAR {
            jarGem = GEM {
                gemTotalSupply = 0,
                gemBalanceOf   = ∅,
                gemAllowance    = ∅
            },
            jarTag = WAD 0,
            jarZzz = 0
        }
}

```

```

initialSystem :: RAY → System
initialSystem HOW0 = System {
    systemVat    = initialVat HOW0,
    systemLads   = ∅,
    systemEra    = 0,
    systemSender = idgod
}

```


Chapter 4

Act framework

4.1 Act descriptions

We define the Maker act vocabulary as a data type. This is used for logging and generally for representing acts.

```
data Act =  
  Bite      (Id URN)  
| Draw      (Id URN) WAD  
| Form      (Id ILK) (Id JAR)  
| Free      (Id URN) WAD  
| Frob      RAY  
| Give      (Id URN) (Id LAD)  
| Grab      (Id URN)  
| Heal      WAD  
| Lock      (Id URN) WAD  
| Loot      WAD  
| Mark      (Id JAR) WAD    NAT  
| Open      (Id URN) (Id ILK)  
| Prod  
| Poke      (Id URN)  
| Pull      (Id JAR) (Id LAD) WAD  
| Shut      (Id URN)  
| Tell      WAD  
| Warp      NAT  
| Wipe      (Id URN) WAD  
| NewJar    (Id JAR) JAR
```

```

    | NewLad (Id LAD)
deriving (Eq, Show, Read)

```

Acts which are logged through the **note** modifier record the sender ID and the act descriptor.

```

data Log = LogNote (Id LAD) Act
deriving (Show, Eq)

```

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

```

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

```

Now we can define the type of a

```

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

```

```

exec :: System
    → Maker ()
    → Either Error (System, Seq Log)
exec sys (Maker m) =
  runExcept (runWriterT (execStateT m sys))

```

```

instance MonadReader System Maker where
  ask = Maker get
  local f (Maker m) = Maker $ do
    s ← get; put (f s)
    x ← m; put s
    return x

```

4.2 Constraints

```

type Reads r m = MonadReader r m
type Writes w m = MonadState w m
type Logs    m = MonadWriter (Seq Log) m
type Fails   m = MonadError Error m
type IsAct = ?act :: Act
type Notes   m = (IsAct, Logs m)

```

4.3 Accessor aliases

```

ilkAt id = VAT ∘ ILKS ∘ ix id
urnAt id = VAT ∘ URNS ∘ ix id
jarAt id = VAT ∘ JARS ∘ ix id

```

4.4 Logging and asserting

```

log :: Logs m ⇒ Log → m ()
log x = Writer.tell (Sequence.singleton x)
sure :: Fails m ⇒ Bool → m ()
sure x = unless x (throwError AssertionError)
need :: (Fails m, Reads r m)
  ⇒ Getting (First a) r a → m a
need f = preview f ≫ λcase
  Nothing → throwError AssertionError
  Just x → return x

```

4.5 Modifiers

```

note ::
  (IsAct, Logs m,
   Reads r m,

```

HasSender r (Id LAD))
 $\Rightarrow m\ a \rightarrow m\ a$

note $k = \mathbf{do}$
 $s \leftarrow \text{view sender}$
 $x \leftarrow k$
 $\text{log } (\text{LogNote } s\ ?act)$
 $\text{return } x$

auth ::
 (IsAct, Fails m ,
 Reads $r\ m$,
 HasSender r (Id LAD))
 $\Rightarrow m\ a \rightarrow m\ a$

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

Chapter 5

Acts

We call the basic operations of the Dai credit system "acts."

Table 5.1: 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

5.1 Risk assessment

We divide an urn's situation into five stages of risk. Table 5.1 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 =
  let
    Market value of collateral
    PROSDR = view PRO URN0 * view TAG JAR0
    Debt at DAI target price
    CONSDR = view CON URN0 * PAR0
  in if
    Undergoing liquidation?
    | view VOW URN0 ≠ Nothing → Dread
    Liquidation triggered?
    | view CAT URN0 ≠ Nothing → Grief
    Undercollateralized?
    | PROSDR < CONSDR * view MAT ILK0 → Panic
    Price feed expired?
    | ERA0 > view ZZZ JAR0 + view LAG ILK0 → Panic
    Price feed in limbo?
    | view ZZZ JAR0 < ERA0 → Worry
    Debt ceiling reached?
    | view COW ILK0 > view HAT ILK0 → Anger
    Safely overcollateralized.
    | otherwise → Pride

```


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

```

gaze  $id_{URN}$  = do
  prod
  poke  $id_{URN}$ 
   $ERA_0 \leftarrow view\ ERA$ 
   $PAR_0 \leftarrow view\ (VAT \circ PAR)$ 
   $URN_0 \leftarrow need\ (urnAt\ id_{URN})$ 
   $ILK_0 \leftarrow need\ (ilkAt\ (view\ ILK\ URN_0))$ 
   $JAR_0 \leftarrow need\ (jarAt\ (view\ JAR\ ILK_0))$ 
  return (analyze  $ERA_0\ PAR_0\ URN_0\ ILK_0\ JAR_0$ )

```

5.2 Lending

```

open  $id_{URN}\ id_{ILK}$  =
  note $ do
     $id_{LAD} \leftarrow view\ sender$ 
     $VAT \circ URNs \circ at\ id_{URN} \text{ ?= } defaultUrn\ id_{ILK}\ id_{LAD}$ 

```

```

lock  $id_{URN}\ x$  =
  note $ do
    Ensure CDP exists; identify collateral type
     $id_{ILK} \leftarrow need\ (urnAt\ id_{URN} \circ ILK)$ 
     $id_{JAR} \leftarrow need\ (ilkAt\ id_{ILK} \circ JAR)$ 
    Record an increase in collateral
     $urnAt\ id_{URN} \circ PRO \text{ += } x$ 
    Take sender's tokens
     $id_{LAD} \leftarrow view\ sender$ 
    pull  $id_{JAR}\ id_{LAD}\ x$ 

```

```

free  $id_{URN}\ WAD_{GEM}$  =
  note $ do

```

Fail if sender is not the CDP owner.

$id_{sender} \leftarrow view\ sender$
 $id_{LAD} \leftarrow need\ (urnAt\ id_{URN} \circ LAD)$
 $sure\ (id_{sender} \equiv id_{LAD})$

Tentatively record the decreased collateral.

$urnAt\ id_{URN} \circ PRO \dashv=\ WAD_{GEM}$

Fail if collateral decrease results in undercollateralization.

$gaze\ id_{URN} \gg= sure \circ (\equiv\ Pride)$

Send the collateral to the CDP owner.

$id_{ILK} \leftarrow need\ (urnAt\ id_{URN} \circ ILK)$
 $id_{JAR} \leftarrow need\ (ilkAt\ id_{ILK} \circ JAR)$
 $push\ id_{JAR}\ id_{LAD}\ WAD_{GEM}$

draw $id_{URN}\ WAD_{DAI} =$

note \$ **do**

Fail if sender is not the CDP owner.

$id_{sender} \leftarrow view\ sender$
 $id_{LAD} \leftarrow need\ (urnAt\ id_{URN} \circ LAD)$
 $sure\ (id_{sender} \equiv id_{LAD})$

Tentatively record DAI debt.

$urnAt\ id_{URN} \circ CON \dashv+\ WAD_{DAI}$

Fail if CDP with new debt is not overcollateralized.

$gaze\ id_{URN} \gg= sure \circ (\equiv\ Pride)$

Mint DAI and send it to the CDP owner.

mint $id_{DAI}\ WAD_{DAI}$
push $id_{DAI}\ id_{LAD}\ WAD_{DAI}$

wipe $id_{URN}\ WAD_{DAI} =$

note \$ **do**

Fail if sender is not the CDP owner.

$id_{sender} \leftarrow view\ sender$
 $id_{LAD} \leftarrow need\ (urnAt\ id_{URN} \circ LAD)$
 $sure\ (id_{sender} \equiv id_{LAD})$

Fail if the CDP is not currently overcollateralized.

$gaze\ id_{URN} \gg= sure \circ (\equiv\ Pride)$

Preliminarily reduce the CDP debt.

$urnAt\ id_{URN} \circ CON \text{ -- } WAD_{DAI}$

Attempt to get back DAI from CDP owner and destroy it.

pull $id_{DAI}\ id_{LAD}\ WAD_{DAI}$

burn $id_{DAI}\ WAD_{DAI}$

give $id_{URN}\ id_{LAD} =$

note \$ **do**

$x \leftarrow need\ (urnAt\ id_{URN} \circ LAD)$

$y \leftarrow view\ sender$

$sure\ (x \equiv y)$

$urnAt\ id_{URN} \circ LAD := id_{LAD}$

shut $id_{URN} =$

note \$ **do**

Update the CDP's debt (prorating the stability fee).

poke id_{URN}

Attempt to repay all the CDP's outstanding DAI.

$CON_0 \leftarrow need\ (urnAt\ id_{URN} \circ CON)$

wipe $id_{URN}\ CON_0$

Reclaim all the collateral.

$PRO_0 \leftarrow need\ (urnAt\ id_{URN} \circ PRO)$

free $id_{URN}\ PRO_0$

Nullify the CDP.

$VAT \circ URNs \circ at\ id_{URN} := \text{Nothing}$

5.3 Frequent adjustments

```

prod = note $ do
  ERA0 ← view ERA
  TAU0 ← view (VAT ∘ TAU)
  FIX0 ← view (VAT ∘ FIX)
  PAR0 ← view (VAT ∘ PAR)
  HOW0 ← view (VAT ∘ HOW)
  WAY0 ← view (VAT ∘ WAY)
  let
    Time difference in seconds
    fan = ERA0 − TAU0
    Current deflation rate applied to target price
    PAR1 = PAR0 * cast (WAY0 ↑↑ fan)
    Sensitivity parameter applied over time
    wag = HOW0 * fromIntegral fan
    Deflation 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)

```

This internal act happens on every **poke**. It is also invoked when governance changes the TAX of an ILK.

```

drip idILK = do
  Current time stamp
  ERA0 ← view ERA
  Current stability fee
  TAX0 ← need (ilkAt idILK ∘ TAX)
  COW0 ← need (ilkAt idILK ∘ COW)
  Previous time and stability fee thus far

```

```

RHO0 ← need (ilkAt idILK ∘ RHO)
ice    ← need (ilkAt idILK ∘ BAG ∘ ix RHO0)
let
  Seconds passed
  age    = ERA0 − RHO0
  Stability fee accrued since last drip
  dew    = ice * TAX0 ↑↑ age
  I don't understand this calculation
  COW1 = COW0 * (dew / ice)
  ilkAt idILK ∘ BAG ∘ at ERA0 ?= dew
  ilkAt idILK ∘ COW                := COW1
  ilkAt idILK ∘ RHO                := ERA0
  return dew

```

```

poke idURN =
  note $ do
    Read previous stability fee accumulator.
    idILK ← need (urnAt idURN ∘ ILK)
    phi0  ← need (urnAt idURN ∘ PHI)
    ice   ← need (ilkAt idILK ∘ BAG ∘ ix phi0)
    Update the stability fee accumulator.
    CON0 ← need (urnAt idURN ∘ CON)
    dew   ← drip idILK
    Apply new stability fee to CDP debt.
    urnAt idURN ∘ CON * = cast (dew / ice)
    Record the poke time.
    ERA0 ← view ERA
    urnAt idURN ∘ PHI := ERA0

```

5.4 Governance

```

form idILK idJAR =
  auth ∘ note $ do
    VAT ∘ ILKS ∘ at idILK ?= defaultIlk idJAR

```

```

frob how' =
  auth ◦ note $ do
    VAT ◦ HOW := how'

```

5.5 Price feedback

```

mark idJAR TAG1 ZZZ1 =
  auth ◦ note $ do
    jarAt idJAR ◦ TAG := TAG1
    jarAt idJAR ◦ ZZZ := ZZZ1

```

```

tell x =
  auth ◦ note $ do
    VAT ◦ FIX := x

```

5.6 Liquidation and settlement

```

bite idURN =
  note $ do
    Fail if urn is not undercollateralized.
    gaze idURN  $\gg$  sure ◦ ( $\equiv$  Panic)
    Record the sender as the liquidation initiator.
    idCAT  $\leftarrow$  view sender
    urnAt idURN ◦ CAT := idCAT
    Read current debt.
    CON0  $\leftarrow$  need (urnAt idURN ◦ CON)
    Read liquidation penalty ratio.
    idILK  $\leftarrow$  need (urnAt idURN ◦ ILK)
    AXE0  $\leftarrow$  need (ilkAt idILK ◦ AXE)
    Apply liquidation penalty to debt.
    let CON1 = CON0 * AXE0

```

Update debt and record it as in need of settlement.

$$\begin{array}{l} \text{urnAt } id_{URN} \circ CON := CON_1 \\ SIN \quad \quad \quad += CON_1 \end{array}$$

grab $id_{URN} =$

auth \circ **note** \$ **do**

Fail if CDP liquidation is not initiated.

$$\text{gaze } id_{URN} \gg \text{sure} \circ (\equiv \text{Grief})$$

Record the sender as the CDP's settler.

$$id_{VOW} \leftarrow \text{view sender}$$

$$\text{urnAt } id_{URN} \circ VOW := id_{VOW}$$

Nullify the CDP's debt and collateral.

$$PRO_0 \leftarrow \text{need } (\text{urnAt } id_{URN} \circ PRO)$$

$$\text{urnAt } id_{URN} \circ CON := 0$$

$$\text{urnAt } id_{URN} \circ PRO := 0$$

Send the collateral to the settler for auctioning.

$$id_{ILK} \leftarrow \text{need } (\text{urnAt } id_{URN} \circ ILK)$$

$$id_{JAR} \leftarrow \text{need } (ilkAt id_{ILK} \circ JAR)$$

$$\text{push } id_{JAR} id_{VOW} PRO_0$$

heal $WAD_{DAI} =$

auth \circ **note** \$ **do**

$$VAT \circ SIN \text{ -- } WAD_{DAI}$$

loot $WAD_{DAI} =$

auth \circ **note** \$ **do**

$$VAT \circ PIE \text{ -- } WAD_{DAI}$$

5.7 Minting, burning, and transferring

pull $id_{JAR} id_{LAD} w = \text{do}$

$$g \leftarrow \text{need } (jarAt id_{JAR} \circ GEM)$$

$$g' \leftarrow \text{transferFrom } id_{\text{LAD}} \ id_{\text{VAT}} \ w \ g$$

$$\text{jarAt } id_{\text{JAR}} \circ \text{GEM} := g'$$

$$\text{push } id_{\text{JAR}} \ id_{\text{LAD}} \ w = \mathbf{do}$$

$$g \leftarrow \text{need } (\text{jarAt } id_{\text{JAR}} \circ \text{GEM})$$

$$g' \leftarrow \text{transferFrom } id_{\text{VAT}} \ id_{\text{LAD}} \ w \ g$$

$$\text{jarAt } id_{\text{JAR}} \circ \text{GEM} := g'$$

$$\text{mint } id_{\text{JAR}} \ \text{WAD}_0 = \mathbf{do}$$

$$\text{jarAt } id_{\text{JAR}} \circ \text{GEM} \circ \text{totalSupply} \quad \quad \quad += \text{WAD}_0$$

$$\text{jarAt } id_{\text{JAR}} \circ \text{GEM} \circ \text{balanceOf} \circ ix \ id_{\text{VAT}} \quad \quad \quad += \text{WAD}_0$$

$$\text{burn } id_{\text{JAR}} \ \text{WAD}_0 = \mathbf{do}$$

$$\text{jarAt } id_{\text{JAR}} \circ \text{GEM} \circ \text{totalSupply} \quad \quad \quad -= \text{WAD}_0$$

$$\text{jarAt } id_{\text{JAR}} \circ \text{GEM} \circ \text{balanceOf} \circ ix \ id_{\text{VAT}} \quad \quad \quad -= \text{WAD}_0$$

5.8 Test-related manipulation

$$\text{warp } t =$$

$$\quad \text{auth} \circ \text{note } \$ \ \mathbf{do}$$

$$\quad \quad \text{ERA} \ += \ t$$

5.9 System modelling

$$\text{newLad } id_{\text{LAD}} = \text{lads} \circ \text{at } id_{\text{LAD}} \ ?= \text{LAD}$$

$$\text{newLad} ::$$

$$\quad (\text{Writes } w \ m, \text{HasLads } w \ (\text{IdMap } \text{LAD}))$$

$$\quad \Rightarrow \text{Id } \text{LAD} \rightarrow m \ ()$$


```

newJar id idJAR =
  auth ◦ note $ do
    VAT ◦ JARS ◦ at id ?= idJAR

newJar ::
  ( IsAct, Fails m, Logs m,
    Reads r m, HasSender r (Id LAD),
    Writes w m, HasVat w VATw,
    HasJars VATw (IdMap JAR))
⇒
  Id JAR → JAR → m ()

```

5.10 Other stuff

```

perform :: Act → Maker ()
perform x =
  let ?act = x in case x of
    NewLad id      → newLad id
    NewJar id JAR  → newJar id JAR
    Form id JAR    → form id JAR
    Mark JAR TAG ZZZ → mark JAR TAG ZZZ
    Open id ILK    → 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

transferFrom
  :: (MonadError Error m)
  ⇒ Id LAD → Id LAD → WAD
  → GEM → m GEM

transferFrom src dst WAD GEM =
  case GEM ^ . balanceOf ◦ (at src) of
    Nothing →
      throwError AssertionError

```

```

Just balance → do
  sure (balance ≥ WAD)
  return $ GEM
    & balanceOf ∘ ix src −~ WAD
    & balanceOf ∘ at dst %~
  (λcase
    Nothing → Just WAD
    Just x   → Just (WAD + x))

```

Chapter 6

Testing

Appendix A

Act type signatures

We see that `drip` may fail; it reads an ILK's TAX, COW, RHO, and BAG; and it writes those same parameters except TAX.

```
drip ::
  (Fails  $m$ ,
   Reads  $r$   $m$ ,
   HasEra  $r$  NAT,
   HasVat  $r$  VAT $_r$ ,
   HasIlks VAT $_r$  (Map (Id ILK) ILK $_r$ ),
   HasTax ILK $_r$  RAY,
   HasCow ILK $_r$  RAY,
   HasRho ILK $_r$  NAT,
   HasBag ILK $_r$  (Map NAT RAY)),
  Writes  $w$   $m$ ,
  HasVat  $w$  VAT $_w$ ,
  HasIlks VAT $_w$  (Map (Id ILK) ILK $_w$ ),
  HasCow ILK $_w$  RAY,
  HasRho ILK $_w$  NAT,
  HasBag ILK $_w$  (Map NAT RAY))
 $\Rightarrow$  Id ILK  $\rightarrow m$  RAY
```

```
form ::
  (IsAct, Fails  $m$ , Logs  $m$ ,
   Reads  $r$   $m$ , HasSender  $r$  (Id LAD),
   Writes  $w$   $m$ , HasVat  $w$  VAT $_w$ ,
   HasIlks VAT $_w$  (IdMap ILK))
```

$$\Rightarrow \text{Id ILK} \rightarrow \text{Id JAR} \rightarrow m \ ()$$

frob :: (IsAct, Fails m , Logs m ,
 Reads $r \ m$, HasSender $r \ (\text{Id LAD})$,
 Writes $w \ m$, HasVat $w \ \text{VAT}_w$,
 HasHow $\text{VAT}_w \ \text{RAY}$)
 $\Rightarrow \text{RAY} \rightarrow m \ ()$

open ::
 (IsAct, Logs m ,
 Reads $r \ m$, HasSender $r \ (\text{Id LAD})$,
 Writes $w \ m$, HasVat $w \ \text{VAT}_w$,
 HasUrns $\text{VAT}_w \ (\text{IdMap URN})$)
 $\Rightarrow \text{Id URN} \rightarrow \text{Id ILK} \rightarrow m \ ()$

give ::
 (IsAct, Fails m , Logs m ,
 Reads $r \ m$, HasSender $r \ (\text{Id LAD})$,
 HasVat $r \ \text{VAT}_r$,
 HasUrns $\text{VAT}_r \ (\text{Map} \ (\text{Id URN}) \ \text{URN}_r)$,
 HasLad $\text{URN}_r \ (\text{Id LAD})$,
 Writes $w \ m$, HasVat $w \ \text{VAT}_r$)
 $\Rightarrow \text{Id URN} \rightarrow \text{Id LAD} \rightarrow m \ ()$

lock ::
 (IsAct, Fails m , Logs m ,
 Reads $r \ m$,
 HasSender $r \ (\text{Id LAD})$,
 HasVat $r \ \text{VAT}_r$,
 HasUrns $\text{VAT}_r \ (\text{Map} \ (\text{Id URN}) \ \text{URN}_r)$,
 HasIlk $\text{URN}_r \ (\text{Id ILK})$,
 HasIlks $\text{VAT}_r \ (\text{Map} \ (\text{Id ILK}) \ \text{ILK}_r)$,
 HasJar $\text{ILK}_r \ (\text{Id JAR})$,
 HasJars $\text{VAT}_r \ (\text{Map} \ (\text{Id JAR}) \ \text{JAR}_r)$,
 HasGem $\text{JAR}_r \ \text{GEM}$,
 Writes $w \ m$,
 HasVat $w \ \text{VAT}_w$,
 HasJars $\text{VAT}_w \ (\text{Map} \ (\text{Id JAR}) \ \text{JAR}_r)$,
 HasUrns $\text{VAT}_w \ (\text{Map} \ (\text{Id URN}) \ \text{URN}_w)$,

$$\begin{aligned} & \text{HasPro URN}_w \text{ WAD}) \\ \Rightarrow \text{Id URN} \rightarrow \text{WAD} \rightarrow m \ () \end{aligned}$$

mark ::

$$\begin{aligned} & (\text{IsAct}, \text{Fails } m, \text{Logs } m, \\ & \text{Reads } r \ m, \text{HasSender } r \ (\text{Id LAD}), \\ & \text{Writes } w \ m, \text{HasVat } w \ \text{VAT}_w, \\ & \quad \text{HasJars } \text{VAT}_w \ (\text{Map } (\text{Id JAR}) \ \text{JAR}_w), \\ & \quad \text{HasTag } \text{JAR}_w \ \text{WAD}, \\ & \quad \text{HasZzz } \text{JAR}_w \ \text{NAT}) \\ \Rightarrow \text{Id JAR} \rightarrow \text{WAD} \rightarrow \text{NAT} \rightarrow m \ () \end{aligned}$$

tell ::

$$\begin{aligned} & (\text{IsAct}, \text{Fails } m, \text{Logs } m, \\ & \text{Reads } r \ m, \text{HasSender } r \ (\text{Id LAD}), \\ & \text{Writes } w \ m, \text{HasVat } w \ \text{VAT}_w, \\ & \quad \text{HasFix } \text{VAT}_w \ \text{WAD}) \\ \Rightarrow \text{WAD} \rightarrow m \ () \end{aligned}$$

prod ::

$$\begin{aligned} & (\text{IsAct}, \text{Logs } m, \\ & \text{Reads } r \ m, \\ & \quad \text{HasSender } r \ (\text{Id LAD}), \\ & \quad \text{HasEra } r \ \text{NAT}, \\ & \quad \text{HasVat } r \ \text{VAT}_r, \ (\text{HasPar } \text{VAT}_r \ \text{WAD}, \\ & \quad \quad \text{HasTau } \text{VAT}_r \ \text{NAT}, \\ & \quad \quad \text{HasHow } \text{VAT}_r \ \text{RAY}, \\ & \quad \quad \text{HasWay } \text{VAT}_r \ \text{RAY}, \\ & \quad \quad \text{HasFix } \text{VAT}_r \ \text{WAD}), \\ & \text{Writes } w \ m, \\ & \quad \text{HasVat } w \ \text{VAT}_w, \ (\text{HasPar } \text{VAT}_w \ \text{WAD}, \\ & \quad \quad \text{HasWay } \text{VAT}_w \ \text{RAY}, \\ & \quad \quad \text{HasTau } \text{VAT}_w \ \text{NAT}), \\ & \text{Integral NAT}, \\ & \text{Ord WAD}, \text{Fractional WAD}, \\ & \text{Fractional RAY}, \text{Real RAY}) \\ \Rightarrow m \ () \end{aligned}$$

warp ::

$$(\text{IsAct}, \text{Fails } m, \text{Logs } m,$$

$$\begin{aligned} \text{burn} :: & \\ & (\text{Fails } m, \\ & \text{Writes } w \ m, \\ & \quad \text{HasVat } w \ \text{VAT}_w, \text{HasJars } \text{VAT}_w \ (\text{Map } (\text{Id } \text{JAR}) \ \text{JAR}_r), \\ & \quad \text{HasGem } \text{JAR}_r \ \text{gem}_r, \\ & \quad \text{HasTotalSupply } \text{gem}_r \ \text{WAD}, \\ & \quad \text{HasBalanceOf } \text{gem}_r \ (\text{Map } (\text{Id } \text{LAD}) \ \text{WAD})) \\ \Rightarrow & \text{Id JAR} \rightarrow \text{WAD} \rightarrow m \ () \end{aligned}$$