MAKER

*presents the*

REFERENCE IMPLEMENTATION

*of the remarkable*

# DAI CREDIT SYSTEM

issuing a diversely collateralized stablecoin

*with last update on March 9, 2017.*

# Contents

# List of Tables

# 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[1] 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.

[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. 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. **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. **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.

```haskell
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). See Appendix B for details on decimal fixed point numbers and rounding.

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

Define the distinct type for 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.

**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.
$cast :: (\mathrm{Real}\ a, \mathrm{Fractional}\ b) \Rightarrow a \to b$
$cast = fromRational\ .\ toRational$

## 3.2 Identifiers and addresses

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

The type parameter $a$ creates distinct types.

For example, Id Foo and Id Bar are incompatible.

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

We define another type for representing Ethereum account addresses.

$$\textbf{data}\ \mathrm{Address} = \mathrm{Address\ String}$$
$$\quad \textbf{deriving}\ (\mathrm{Ord}, \mathrm{Eq}, \mathrm{Show})$$

We also have three predefined entity identifiers.

The DAI token address

$id_{\text{DAI}} = \mathrm{Id}$ `"DAI"`

The CDP engine address

$id_{\texttt{vat}} = \mathrm{Address}$ `"VAT"`

The address of the test driver

$id_{toy} = \mathrm{Address}$ `"TOY"`

A test account with ultimate authority

$id_{god} = \mathrm{Address}$ `"GOD"`

## 3.3 `Gem` — token model

In this model, all tokens behave in the same simple way.[1]

**data** `Gem` = `Gem` {

- `balanceOf` :: Map Holder `Wad`

} **deriving** (Eq, Show)

We distinguish between tokens held by vaults and tokens held by other addresses.

**data** Holder = InAccount Address
| InVault (Id `Jar`)
**deriving** (Eq, Show, Ord)

[1] In the real world, token semantics can differ, despite nominally following the ERC20 interface. Maker governance therefore involves due diligence on collateral token contracts.

## 3.4 Jar — collateral vaults

**data** `Jar` = `Jar` {

- `gem` :: `Gem`, Collateral token
- `tag` :: `Wad`, Market price
- `zzz` :: `Sec` Price expiration

} **deriving** (Eq, Show)

## 3.5 Ilk — CDP type

**data** `Ilk` = `Ilk` {

- `jar` :: Id `Jar`, Collateral vault
- `axe` :: `Ray`, Liquidation penalty
- `hat` :: `Wad`, Debt ceiling
- `mat` :: `Ray`, Liquidation ratio
- `tax` :: `Ray`, Stability fee
- `lag` :: `Sec`, Limbo duration
- `rho` :: `Sec`, Last dripped
- `rum` :: `Wad`, Total debt in debt unit
- `chi` :: `Ray` Dai value of debt unit

} **deriving** (Eq, Show)

## 3.6 Urn — collateralized debt position (CDP)

**data** `Urn` = `Urn` {

- `cat` :: Maybe Address, Address of liquidation initiator
- `vow` :: Maybe Address, Address of liquidation contract
- `lad` :: Address, Issuer
- `ilk` :: Id `Ilk`, CDP type
- `art` :: `Wad`, Outstanding debt in debt unit
- `jam` :: `Wad` Collateral amount in debt unit

} **deriving** (Eq, Show)

## 3.7 `Vat` — CDP engine

**data** `Vat` = `Vat` {

- `fix` :: `Wad`, — Market price
- `how` :: `Ray`, — Sensitivity
- `par` :: `Wad`, — Target price
- `way` :: `Ray`, — Target rate
- `tau` :: `Sec`, — Last prodded
- `joy` :: `Wad`, — Unprocessed stability fees
- `sin` :: `Wad`, — Bad debt from liquidated CDPs
- `jars` :: Map (Id `Jar`) `Jar`, — Collateral tokens
- `ilks` :: Map (Id `Ilk`) `Ilk`, — CDP types
- `urns` :: Map (Id `Urn`) `Urn` — CDPs

} **deriving** (Eq, Show)

## 3.8 System model

**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)

# Lens fields

*makeLenses* '' `Gem`
*makeLenses* '' `Jar`
*makeLenses* '' `Ilk`
*makeLenses* '' `Urn`
*makeLenses* '' `Vat`
*makeLenses* '' System

## 3.9 Default data

$defaultIlk :: \text{Id } \texttt{Jar} \to \texttt{Ilk}$
$defaultIlk\ id_{\texttt{jar}} = \texttt{Ilk}\ \{$

- $\texttt{jar} = id_{\texttt{jar}}$,
- $\texttt{axe} = \texttt{Ray}\ 1$,
- $\texttt{mat} = \texttt{Ray}\ 1$,
- $\texttt{tax} = \texttt{Ray}\ 1$,
- $\texttt{hat} = \texttt{Wad}\ 0$,
- $\texttt{lag} = \texttt{Sec}\ 0$,
- $\texttt{chi} = \texttt{Ray}\ 1$,
- $\texttt{rum} = \texttt{Wad}\ 0$,
- $\texttt{rho} = \texttt{Sec}\ 0$

$\}$

$defaultUrn :: \text{Id } \texttt{Ilk} \to \text{Address} \to \texttt{Urn}$
$defaultUrn\ id_{\texttt{ilk}}\ id_{\texttt{lad}} = \texttt{Urn}\ \{$

- $\texttt{vow} = \text{Nothing}$,
- $\texttt{cat} = \text{Nothing}$,
- $\texttt{lad} = id_{\texttt{lad}}$,
- $\texttt{ilk} = id_{\texttt{ilk}}$,
- $\texttt{art} = \texttt{Wad}\ 0$,
- $\texttt{jam} = \texttt{Wad}\ 0$

$\}$

$initialVat :: \texttt{Ray} \to \texttt{Vat}$
$initialVat\ \texttt{how}_0 = \texttt{Vat}\ \{$

- $\texttt{tau} = 0$,
- $\texttt{fix} = \texttt{Wad}\ 1$,
- $\texttt{par} = \texttt{Wad}\ 1$,
- $\texttt{how} = \texttt{how}_0$,
- $\texttt{way} = \texttt{Ray}\ 1$,
- $\texttt{joy} = \texttt{Wad}\ 0$,
- $\texttt{sin} = \texttt{Wad}\ 0$,
- $\texttt{ilks} = \varnothing$,
- $\texttt{urns} = \varnothing$,
- $\texttt{jars} =$
  $singleton\ id_{\text{DAI}}\ \texttt{Jar}\ \{$
  - $\texttt{gem} = \texttt{Gem}\ \{$

- `balanceOf` $= \varnothing$

},

- `tag` $=$ `Wad` $0$,
- `zzz` $= 0$

}

}

$initialSystem :: \texttt{Ray} \to \mathrm{System}$
$initialSystem\ \texttt{how}_0 = \mathrm{System}\ \{$

- `vat` $= initialVat\ \texttt{how}_0$,
- `era` $= 0$,
- `sender` $= id_{god}$,
- `accounts` $= mempty$

}

# 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 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 * cast (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.

$\texttt{gaze}\ id_{\texttt{urn}} = \textbf{do}$

Perform dai revaluation and rate adjustment

$\texttt{prod}$

Update price of specific debt unit

$id_{\texttt{ilk}} \leftarrow look\ (\texttt{vat}\,.\,\texttt{urn}s\,.\,ix\ id_{\texttt{urn}}\,.\,\texttt{ilk})$

$\texttt{drip}\ id_{\texttt{ilk}}$

Read parameters for risk analysis

$\texttt{era}_0 \leftarrow use\ \texttt{era}$

$\texttt{par}_0 \leftarrow use\ (\texttt{vat}\,.\,\texttt{par})$

$\texttt{urn}_0 \leftarrow look\ (\texttt{vat}\,.\,\texttt{urn}s\,.\,ix\ id_{\texttt{urn}})$

$\texttt{ilk}_0 \leftarrow look\ (\texttt{vat}\,.\,\texttt{ilk}s\,.\,ix\ (view\ \texttt{ilk}\ \texttt{urn}_0))$

$\texttt{jar}_0 \leftarrow look\ (\texttt{vat}\,.\,\texttt{jar}s\,.\,ix\ (view\ \texttt{jar}\ \texttt{ilk}_0))$

Return risk stage of CDP

$return\ (analyze\ \texttt{era}_0\ \texttt{par}_0\ \texttt{urn}_0\ \texttt{ilk}_0\ \texttt{jar}_0)$

## 4.2 Lending

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

$\texttt{open}\ id_{\texttt{urn}}\ id_{\texttt{ilk}} = \textbf{do}$

Fail if account identifier is taken

$none\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ at\ id_{\texttt{urn}})$

Create a CDP record with the sender as owner

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

$initializeTo\ (defaultUrn\ id_{\texttt{ilk}}\ id_{\texttt{lad}})$
$\quad (\texttt{vat}\ .\ \texttt{urn}s\ .\ at\ id_{\texttt{urn}})$

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

$\texttt{give}\ id_{\texttt{urn}}\ id_{\texttt{lad}} = \textbf{do}$

Fail if sender is not the CDP owner

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

Transfer ownership

$\texttt{vat}\ .\ \texttt{urn}s\ .\ ix\ id_{\texttt{urn}}\ .\ \texttt{lad} := id_{\texttt{lad}}$

$\texttt{lock}\ id_{\texttt{urn}}\ \texttt{wad}_{\texttt{gem}} = \textbf{do}$

Fail if sender is not the CDP owner

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

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

Ensure CDP exists; identify collateral type

$id_{\texttt{ilk}} \leftarrow look\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ ix\ id_{\texttt{urn}}\ .\ \texttt{ilk})$

$id_{\texttt{jar}} \leftarrow look\ (\texttt{vat}\ .\ \texttt{ilk}s\ .\ ix\ id_{\texttt{ilk}}\ .\ \texttt{jar})$

Record an increase in collateral

$increaseBy\ \texttt{wad}_{\texttt{gem}}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ ix\ id_{\texttt{urn}}\ .\ \texttt{jam})$

Take sender's tokens

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

$\texttt{pull}\ id_{\texttt{jar}}\ id_{\texttt{lad}}\ \texttt{wad}_{\texttt{gem}}$

$\texttt{free}\ id_{\texttt{urn}}\ \texttt{wad}_{\texttt{gem}} = \textbf{do}$

Fail if sender is not the CDP owner

$id_{\texttt{lad}} \leftarrow \textit{use sender}$
$\textit{owns}\ id_{\texttt{urn}}\ id_{\texttt{lad}}$

Decrease the collateral amount
$\textit{decreaseBy}\ \texttt{wad}_{\texttt{gem}}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ \textit{ix}\ id_{\texttt{urn}}\ .\ \texttt{jam})$

Roll back if undercollateralized
$\texttt{gaze}\ id_{\texttt{urn}} \ggg \texttt{aver}\ .\ (\equiv \texttt{Pride})$

Send the collateral to the CDP owner
$id_{\texttt{ilk}} \leftarrow \textit{look}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ \textit{ix}\ id_{\texttt{urn}}\ .\ \texttt{ilk})$
$id_{\texttt{jar}} \leftarrow \textit{look}\ (\texttt{vat}\ .\ \texttt{ilk}s\ .\ \textit{ix}\ id_{\texttt{ilk}}\ .\ \texttt{jar})$
$\texttt{push}\ id_{\texttt{jar}}\ id_{\texttt{lad}}\ \texttt{wad}_{\texttt{gem}}$

$\texttt{draw}\ id_{\texttt{urn}}\ \texttt{wad}_{\text{DAI}} = \mathbf{do}$

Fail if sender is not the CDP owner
$id_{\texttt{lad}} \leftarrow \textit{use sender}$
$\textit{owns}\ id_{\texttt{urn}}\ id_{\texttt{lad}}$

Update value of debt unit
$id_{\texttt{ilk}} \leftarrow \textit{look}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ \textit{ix}\ id_{\texttt{urn}}\ .\ \texttt{ilk})$
$\texttt{chi}_1 \leftarrow \texttt{drip}\ id_{\texttt{ilk}}$

Denominate draw amount in debt unit
$\mathbf{let}\ \texttt{wad}_{\texttt{chi}} = \texttt{wad}_{\text{DAI}}\ /\ \textit{cast}\ \texttt{chi}_1$

Increase debt
$\textit{increaseBy}\ \texttt{wad}_{\texttt{chi}}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ \textit{ix}\ id_{\texttt{urn}}\ .\ \texttt{art})$

Roll back unless overcollateralized
$\texttt{gaze}\ id_{\texttt{urn}} \ggg \texttt{aver}\ .\ (\equiv \texttt{Pride})$

Mint dai and send to the CDP owner
$\texttt{mint}\ id_{\text{DAI}}\ \texttt{wad}_{\text{DAI}}$
$\texttt{push}\ id_{\text{DAI}}\ id_{\texttt{lad}}\ \texttt{wad}_{\text{DAI}}$

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

Fail if sender is not the CDP owner
$id_{\texttt{lad}} \leftarrow \textit{use sender}$
$\textit{owns}\ id_{\texttt{urn}}\ id_{\texttt{lad}}$

Update value of debt unit
$id_{\texttt{ilk}} \leftarrow \textit{look}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ \textit{ix}\ id_{\texttt{urn}}\ .\ \texttt{ilk})$
$\texttt{chi}_1 \leftarrow \texttt{drip}\ id_{\texttt{ilk}}$

Roll back unless overcollateralized

$\texttt{gaze}\ id_{\texttt{urn}} \ggg \texttt{aver}\ .\ (\equiv \texttt{Pride})$

Denominate dai amount in debt unit

$\textbf{let}\ \texttt{wad}_{\texttt{chi}} = \texttt{wad}_{\text{DAI}}\ /\ \mathit{cast}\ \texttt{chi}_1$

Reduce debt

$\mathit{decreaseBy}\ \texttt{wad}_{\texttt{chi}}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ \mathit{ix}\ id_{\texttt{urn}}\ .\ \texttt{art})$

Take dai from CDP owner, or roll back

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

Destroy dai

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

$\texttt{shut}\ id_{\texttt{urn}} = \textbf{do}$

Update value of debt unit

$id_{\texttt{ilk}} \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ \mathit{ix}\ id_{\texttt{urn}}\ .\ \texttt{ilk})$

$\texttt{chi}_1 \leftarrow \texttt{drip}\ id_{\texttt{ilk}}$

Attempt to repay all the CDP's outstanding dai

$\texttt{art}_0 \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ \mathit{ix}\ id_{\texttt{urn}}\ .\ \texttt{art})$

$\texttt{wipe}\ id_{\texttt{urn}}\ (\texttt{art}_0 * \mathit{cast}\ \texttt{chi}_1)$

Reclaim all the collateral

$\mathit{jam0} \leftarrow \mathit{look}\ (\texttt{vat}\ .\ \texttt{urn}s\ .\ \mathit{ix}\ id_{\texttt{urn}}\ .\ \texttt{jam})$

$\texttt{free}\ id_{\texttt{urn}}\ \mathit{jam0}$

Nullify the CDP

$\texttt{vat}\ .\ \texttt{urn}s\ .\ \mathit{at}\ id_{\texttt{urn}} := \text{Nothing}$

## 4.3 Adjustment

$\texttt{prod} = \mathbf{do}$

$\quad \texttt{era}_0 \leftarrow \mathit{use}\ \texttt{era}$

$\quad \texttt{tau}_0 \leftarrow \mathit{use}\ (\texttt{vat} \mathbin{.} \texttt{tau})$

$\quad \texttt{fix}_0 \leftarrow \mathit{use}\ (\texttt{vat} \mathbin{.} \texttt{fix})$

$\quad \texttt{par}_0 \leftarrow \mathit{use}\ (\texttt{vat} \mathbin{.} \texttt{par})$

$\quad \texttt{how}_0 \leftarrow \mathit{use}\ (\texttt{vat} \mathbin{.} \texttt{how})$

$\quad \texttt{way}_0 \leftarrow \mathit{use}\ (\texttt{vat} \mathbin{.} \texttt{way})$

$\quad \mathbf{let}$

Time difference in seconds

$\quad\quad \mathit{age} = \texttt{era}_0 - \texttt{tau}_0$

Current target rate applied to target price

$\quad\quad \texttt{par}_1 = \texttt{par}_0 * \mathit{cast}\ (\texttt{way}_0 \uparrow\uparrow \mathit{age})$

Sensitivity parameter applied over time

$\quad\quad \mathit{wag} = \texttt{how}_0 * \mathit{fromIntegral}\ \mathit{age}$

Target rate scaled up or down

$\quad\quad \texttt{way}_1 = \mathit{inj}\ (\mathit{prj}\ \texttt{way}_0 +$

$\quad\quad\quad\quad\quad\quad \mathbf{if}\ \texttt{fix}_0 < \texttt{par}_0\ \mathbf{then}\ \mathit{wag}\ \mathbf{else} - \mathit{wag})$

$\quad \texttt{vat} \mathbin{.} \texttt{par} := \texttt{par}_1$

$\quad \texttt{vat} \mathbin{.} \texttt{way} := \texttt{way}_1$

$\quad \texttt{vat} \mathbin{.} \texttt{tau} := \texttt{era}_0$

$\quad \mathbf{where}$

Convert between multiplicative and additive form

$\quad\quad \mathit{prj}\ x = \mathbf{if}\ x \geqslant 1\ \mathbf{then}\ x - 1\ \mathbf{else}\ 1 - 1 \mathbin{/} x$

$\quad\quad \mathit{inj}\ x = \mathbf{if}\ x \geqslant 0\ \mathbf{then}\ x + 1\ \mathbf{else}\ 1 \mathbin{/} (1 - x)$

```
drip id_ilk = do
  rho₀ ← look (vat . ilks . ix id_ilk . rho)   Time stamp of previous drip
  tax₀ ← look (vat . ilks . ix id_ilk . tax)   Current stability fee
  chi₀ ← look (vat . ilks . ix id_ilk . chi)   Current value of debt unit
  rum₀ ← look (vat . ilks . ix id_ilk . rum)   Current total debt in debt unit
  joy₀ ← look (vat . joy)                      Current unprocessed stability fee revenue
  era₀ ← use era                               Current time stamp
  let
    age  = era₀ − rho₀
    chi₁ = chi₀ ∗ tax₀ ↑↑ age
    joy₁ = joy₀ + (cast (chi₁ − chi₀) :: Wad) ∗ rum₀
  vat . ilks . ix id_ilk . chi := chi₁
  vat . ilks . ix id_ilk . rho := era₀
  vat . joy := joy₁
  return chi₁
```

## 4.4 Feedback

```
mark id_jar tag₁ zzz₁ =
  auth $ do
    vat . jars . ix id_jar . tag := tag₁
    vat . jars . ix id_jar . zzz := zzz₁
```

```
tell wad_gem =
  auth $ do
    vat . fix := wad_gem
```

## 4.5 Liquidation

```
bite id_urn = do
   Fail if CDP is not in need of liquidation
    gaze id_urn ≫= aver . (≡ Panic)
   Record the sender as the liquidation initiator
```

$id_{\texttt{cat}} \leftarrow use\ sender$
$\texttt{vat . urn}s \texttt{ . } ix\ id_{\texttt{urn}} \texttt{ . cat} := \text{Just}\ id_{\texttt{cat}}$

Read current debt
$\texttt{art}_0 \leftarrow look\ (\texttt{vat . urn}s \texttt{ . } ix\ id_{\texttt{urn}} \texttt{ . art})$

Update value of debt unit
$id_{\texttt{ilk}} \leftarrow look\ (\texttt{vat . urn}s \texttt{ . } ix\ id_{\texttt{urn}} \texttt{ . ilk})$
$\texttt{chi}_1 \leftarrow \texttt{drip}\ id_{\texttt{ilk}}$

Read liquidation penalty ratio
$id_{\texttt{ilk}} \leftarrow look\ (\texttt{vat . urn}s \texttt{ . } ix\ id_{\texttt{urn}} \texttt{ . ilk})$
$\texttt{axe}_0 \leftarrow look\ (\texttt{vat . ilk}s \texttt{ . } ix\ id_{\texttt{ilk}} \texttt{ . axe})$

Apply liquidation penalty to debt
$\textbf{let}\ \texttt{art}_1 = \texttt{art}_0 * cast\ \texttt{axe}_0$

Update CDP debt
$\texttt{vat . urn}s \texttt{ . } ix\ id_{\texttt{urn}} \texttt{ . art} := \texttt{art}_1$

Record as bad debt
$increaseBy\ (\texttt{art}_1 * cast\ \texttt{chi}_1)\ (\texttt{vat . sin})$

$\texttt{grab}\ id_{\texttt{urn}} =$
$\quad \texttt{auth}\ \$\ \textbf{do}$

Fail if CDP is not marked for liquidation
$\texttt{gaze}\ id_{\texttt{urn}} \ggg\!= \texttt{aver} \texttt{ . } (\equiv \texttt{Grief})$

Record the sender as the CDP's settler
$id_{\texttt{vow}} \leftarrow use\ sender$
$\texttt{vat . urn}s \texttt{ . } ix\ id_{\texttt{urn}} \texttt{ . vow} := \text{Just}\ id_{\texttt{vow}}$

Forget the CDP's requester of liquidation
$\texttt{vat . urn}s \texttt{ . } ix\ id_{\texttt{urn}} \texttt{ . cat} := \text{Nothing}$

$\texttt{plop}\ id_{\texttt{urn}}\ \texttt{wad}_{\text{DAI}} =$
$\quad \texttt{auth}\ \$\ \textbf{do}$

Fail unless CDP is in liquidation
$\texttt{gaze}\ id_{\texttt{urn}} \ggg\!= \texttt{aver} \texttt{ . } (\equiv \texttt{Dread})$

Forget the CDP's settler
$\texttt{vat . urn}s \texttt{ . } ix\ id_{\texttt{urn}} \texttt{ . vow} := \text{Nothing}$

Return some amount of excess auction gains
$\texttt{vat . urn}s \texttt{ . } ix\ id_{\texttt{urn}} \texttt{ . jam} := \texttt{wad}_{\text{DAI}}$

$\texttt{heal}\ \texttt{wad}_{\text{DAI}} =$
$\quad \texttt{auth}\ \$\ \mathbf{do}$
$\qquad \mathit{decreaseBy}\ \texttt{wad}_{\text{DAI}}\ (\texttt{vat}\ .\ \texttt{sin})$

$\texttt{love}\ \texttt{wad}_{\text{DAI}} =$
$\quad \texttt{auth}\ \$\ \mathbf{do}$
$\qquad \mathit{decreaseBy}\ \texttt{wad}_{\text{DAI}}\ (\texttt{vat}\ .\ \texttt{joy})$

## 4.6 Governance

$\texttt{form}\ \mathit{id}_{\texttt{ilk}}\ \mathit{id}_{\texttt{jar}} =$
$\quad \texttt{auth}\ \$\ \mathbf{do}$
$\qquad \mathit{initializeTo}\ (\mathit{defaultIlk}\ \mathit{id}_{\texttt{jar}})$
$\qquad\quad (\texttt{vat}\ .\ \texttt{ilk}s\ .\ \mathit{at}\ \mathit{id}_{\texttt{ilk}})$

$\texttt{frob}\ \texttt{how}_1 =$
$\quad \texttt{auth}\ \$\ \mathbf{do}$
$\qquad \texttt{vat}\ .\ \texttt{how} := \texttt{how}_1$

$\texttt{chop}\ \mathit{id}_{\texttt{ilk}}\ \texttt{axe}_1 =$
$\quad \texttt{auth}\ \$\ \mathbf{do}$
$\qquad \texttt{vat}\ .\ \texttt{ilk}s\ .\ \mathit{ix}\ \mathit{id}_{\texttt{ilk}}\ .\ \texttt{axe} := \texttt{axe}_1$

$\texttt{cork}\ \mathit{id}_{\texttt{ilk}}\ \texttt{hat}_1 =$
$\quad \texttt{auth}\ \$\ \mathbf{do}$
$\qquad \texttt{vat}\ .\ \texttt{ilk}s\ .\ \mathit{ix}\ \mathit{id}_{\texttt{ilk}}\ .\ \texttt{hat} := \texttt{hat}_1$

$\texttt{calm}\ \mathit{id}_{\texttt{ilk}}\ \texttt{lag}_1 =$
$\quad \texttt{auth}\ \$\ \mathbf{do}$
$\qquad \texttt{vat}\ .\ \texttt{ilk}s\ .\ \mathit{ix}\ \mathit{id}_{\texttt{ilk}}\ .\ \texttt{lag} := \texttt{lag}_1$

```
cuff id_ilk mat_1 =
  auth $ do
    vat . ilks . ix id_ilk . mat := mat_1
```

```
crop id_ilk tax_1 =
  auth $ do
    drip id_ilk
    vat . ilks . ix id_ilk . tax := tax_1
```

## 4.7 Treasury

```
pull id_jar id_lad wad_gem =
  transfer id_jar wad_gem
    (InAccount id_lad)
    (InVault   id_jar)
```

```
push id_jar id_lad wad_gem =
  transfer id_jar wad_gem
    (InVault   id_jar)
    (InAccount id_lad)
```

```
mint id_jar wad_0 =
  zoom (vat . jars . ix id_jar . gem) $ do
    increaseBy wad_0 (balanceOf . ix (InVault id_jar))
```

```
burn id_jar wad_0 =
  zoom (vat . jars . ix id_jar . gem) $ do
    decreaseBy wad_0 (balanceOf . ix (InVault id_jar))
```

## 4.8 Manipulation

$\texttt{warp}\ t = \texttt{auth}\ (\mathbf{do}\ \mathit{increaseBy}\ t\ \texttt{era})$

$\texttt{mine}\ \mathit{id}_{\texttt{jar}} = \mathbf{do}$
  $\mathit{initializeTo}$
    $(\texttt{Jar}\ \{$
      - $\texttt{gem} = \texttt{Gem}\ (\mathit{singleton}\ (\text{InAccount}\ \mathit{id}_{\mathit{toy}})\ 1000000000000),$
      - $\texttt{tag} = \texttt{Wad}\ 0,$
      - $\texttt{zzz} = 0\})$

    $(\texttt{vat}\ .\ \texttt{jar}s\ .\ \mathit{at}\ \mathit{id}_{\texttt{jar}})$

$\texttt{hand}\ \mathit{dst}\ \texttt{wad}_{\texttt{gem}}\ \mathit{id}_{\texttt{jar}} = \mathbf{do}$
  $\mathit{transfer}\ \mathit{id}_{\texttt{jar}}\ \texttt{wad}_{\texttt{gem}}$
    $(\text{InAccount}\ \mathit{id}_{\mathit{toy}})$
    $(\text{InAccount}\ \mathit{dst})$

$\texttt{sire}\ \texttt{lad} = \mathbf{do}\ \mathit{prepend}\ \texttt{lad}\ \mathit{accounts}$

## 4.9 Other stuff

$\mathit{perform} :: \text{Act} \to \text{Maker}\ ()$
$\mathit{perform}\ x =$
  $\mathbf{let}\ ?\mathit{act} = x\ \mathbf{in\ case}\ x\ \mathbf{of}$
    $\texttt{Form}\ \mathit{id}\ \texttt{jar} \to \texttt{form}\ \mathit{id}\ \texttt{jar}$
    $\texttt{Mark jar tag zzz} \to \texttt{mark jar tag zzz}$
    $\texttt{Open}\ \mathit{id}\ \texttt{ilk} \to \texttt{open}\ \mathit{id}\ \texttt{ilk}$
    $\texttt{Tell wad} \to \texttt{tell wad}$
    $\texttt{Frob ray} \to \texttt{frob ray}$
    $\texttt{Prod} \to \texttt{prod}$
    $\texttt{Warp}\ t \to \texttt{warp}\ t$
    $\texttt{Give urn lad} \to \texttt{give urn lad}$
    $\texttt{Pull jar lad wad} \to \texttt{pull jar lad wad}$

```
Lock urn wad → lock urn wad
Mine id      → mine id
Hand lad wad jar → hand lad wad jar
Sire lad     → sire lad
```

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

```
transfer id_jar wad src dst =
  -- Operate in the token's balance table
  zoom (vat . jars . ix id_jar . gem . balanceOf) $ do
    -- Fail if source balance insufficient
    balance ← look (ix src)
    aver (balance ⩾ wad)
    -- Decrease source balance
    decreaseBy wad (ix src)
    -- Increase destination balance
    initializeTo 0   (at dst)
    increaseBy  wad (ix dst)
```

# 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 Act | 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' s a =
  Maker (StateT s (Except Error) a)
  deriving
    (Functor, Applicative, Monad,
     MonadError Error,
     MonadState s)
```

```
type Maker a = Maker' System a
```

```
type instance Zoomed (Maker' s) = Focusing (Except Error)
instance Zoom (Maker' s) (Maker' t) s t where
  zoom l (Maker m) = Maker (zoom l m)
```

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

## 5.3 Asserting

$$\texttt{aver}\ x = \mathit{unless}\ x\ (\mathit{throwError}\ (\mathrm{AssertError}\ ?\mathit{act}))$$

$$\begin{aligned}&\mathit{none}\ x = \mathit{preuse}\ x \ggg= \lambda\mathbf{case}\\&\quad \mathrm{Nothing} \to \mathit{return}\ ()\\&\quad \mathrm{Just}\ \_ \to \mathit{throwError}\ (\mathrm{AssertError}\ ?\mathit{act})\end{aligned}$$

$$\begin{aligned}&\mathit{look}\ f = \mathit{preuse}\ f \ggg= \lambda\mathbf{case}\\&\quad \mathrm{Nothing} \to \mathit{throwError}\ (\mathrm{AssertError}\ ?\mathit{act})\\&\quad \mathrm{Just}\ x \to \mathit{return}\ x\end{aligned}$$

We define $\mathit{owns}\ \mathit{id}_{\texttt{urn}}\ \mathit{id}_{\texttt{lad}}$ as an assertion that the given CDP is owned by the given account.

$$\begin{aligned}&\mathit{owns}\ \mathit{id}_{\texttt{urn}}\ \mathit{id}_{\texttt{lad}} = \mathbf{do}\\&\quad \mathit{id}_{\mathit{sender}} \leftarrow \mathit{use}\ \mathit{sender}\\&\quad \texttt{aver}\ (\mathit{id}_{\mathit{sender}} \equiv \mathit{id}_{\texttt{lad}})\\&\quad \mathit{return}\ \mathit{id}_{\mathit{sender}}\end{aligned}$$

## 5.4 Modifiers

$$\begin{aligned}&\texttt{auth}\ \mathit{continue} = \mathbf{do}\\&\quad s \leftarrow \mathit{use}\ \mathit{sender}\\&\quad \mathit{unless}\ (s \equiv \mathit{id}_{\mathit{god}})\\&\qquad (\mathit{throwError}\ \mathrm{AuthError})\\&\quad \mathit{continue}\end{aligned}$$

# Chapter 6

# Testing

Sketches for property stuff...

$$
\begin{array}{l}
\textbf{data}\ \text{Parameter} = \\
\quad \texttt{Fix} \mid \texttt{Par} \mid \texttt{Way} \\
\\
\textit{maintains} \\
\quad :: \text{Eq}\ a \Rightarrow \text{Lens}'\ \text{System}\ a \to \text{Maker}\ () \\
\qquad\qquad \to \text{System} \to \text{Bool} \\
\textit{maintains}\ p = \lambda m\ \texttt{sys}_0 \to \\
\quad \textbf{case}\ \textit{exec}\ \texttt{sys}_0\ m\ \textbf{of} \\
\qquad \textsf{On success, data must be compared for equality} \\
\qquad \text{Right}\ \texttt{sys}_1 \to \textit{view}\ p\ \texttt{sys}_0 \equiv \textit{view}\ p\ \texttt{sys}_1 \\
\qquad \textsf{On rollback, data is maintained by definition} \\
\qquad \text{Left}\ \_ \qquad \to \text{True} \\
\\
\textit{changesOnly} \\
\quad ::\ \ \text{Lens}'\ \text{System}\ a \to \text{Maker}\ () \\
\quad \to \text{System} \to \text{Bool} \\
\textit{changesOnly}\ p = \lambda m\ \texttt{sys}_0 \to \\
\quad \textbf{case}\ \textit{exec}\ \texttt{sys}_0\ m\ \textbf{of} \\
\qquad \textsf{On success, equalize } p \textsf{ and compare} \\
\qquad \text{Right}\ \texttt{sys}_1 \to \textit{set}\ p\ (\textit{view}\ p\ \texttt{sys}_1)\ \texttt{sys}_0 \equiv \texttt{sys}_1 \\
\qquad \textsf{On rollback, data is maintained by definition} \\
\qquad \text{Left}\ \_ \qquad \to \text{True} \\
\\
\textit{also} :: \text{Lens}'\ s\ a \to \text{Lens}'\ s\ b \to \text{Lens}'\ s\ (a, b) \\
\textit{also}\ f\ g = \textit{lens getter setter}
\end{array}
$$

```
  where
    getter x = (view f x, view g x)
    setter x (a, b) = set f a (set g b x)
```

```
keeps :: Parameter → Maker () → System → Bool
keeps Fix = maintains (vat . fix)
keeps Par = maintains (vat . par)
keeps Way = maintains (vat . way)
```

Thus:

```
foo sys₀ = all (λf → f sys₀)
  [changesOnly ((vat . par) `also`
                (vat . way))
     (perform Prod)]
```

# Appendix A

# Prelude

```haskell
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*,

Constants
$mempty, \bot, otherwise)$

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 $a \,.\, b \,.\, c$ denotes a nested accessor much like `a.b.c` in C-style languages; for more details, consult lens documentation[1].

**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*, | |
| Zoom (..), | |
| *view*, *preview*, | Reads a lens in a **do** block |
| (&~), | Lets us use a **do** block with setters ◊ *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 |

**import** Control.Lens.Zoom *as* X
**import** Control.Lens.Internal.Zoom *as* X

Where the Solidity code uses `mapping`, we use Haskell's regular tree-based map type[2].

**import** Data.Map *as* X (

| | |
|---|---|
| Map, | Type constructor for mappings |
| ∅, | Polymorphic empty mapping |
| *singleton*) | Creates a mapping with a single key–value pair |

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.

[1] Gabriel Gonzalez's 2013 article *Program imperatively using Haskell* is a good introduction.
[2] 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 with $x * y$ and $x \mathbin{/} y$ operations that do rounding instead of truncation of their intermediate results.

$$\textbf{module}\ \mathrm{Maker.Decimal}\ (\mathrm{Decimal}, \mathrm{E18}, \mathrm{E36}, \mathrm{Epsilon}\ (..))\ \textbf{where}$$

$$\textbf{import}\ \mathrm{Data.Fixed}$$

$$\begin{array}{l}\textbf{newtype}\ \mathrm{HasResolution}\ e \Rightarrow \mathrm{Decimal}\ e = \mathrm{D}\ (\mathrm{Fixed}\ e)\\ \quad \textbf{deriving}\ (\mathrm{Ord}, \mathrm{Eq}, \mathrm{Real}, \mathrm{RealFrac})\end{array}$$

We want the printed representations of these numbers to look like `"0.01"` and not `"R 0.01"`.

$$\begin{array}{l}\textbf{instance}\ \mathrm{HasResolution}\ e \Rightarrow \mathrm{Read}\ (\mathrm{Decimal}\ e)\ \textbf{where}\\ \quad \mathit{readsPrec}\ n\ s = \mathit{fmap}\ (\lambda(x, y) \rightarrow (\mathrm{D}\ x, y))\ (\mathit{readsPrec}\ n\ s)\\ \textbf{instance}\ \mathrm{HasResolution}\ e \Rightarrow \mathrm{Show}\ (\mathrm{Decimal}\ e)\ \textbf{where}\\ \quad \mathit{show}\ (\mathrm{D}\ x) = \mathit{show}\ x\end{array}$$

In the Num instance, we delegate everything except multiplication.

$$\begin{array}{l}\textbf{instance}\ \mathrm{HasResolution}\ e \Rightarrow \mathrm{Num}\ (\mathrm{Decimal}\ e)\ \textbf{where}\\ \quad x@(\mathrm{D}\ (\mathrm{MkFixed}\ a)) * \mathrm{D}\ (\mathrm{MkFixed}\ b) =\\ \qquad \mathrm{D}\ (\mathrm{MkFixed}\ (\mathit{div}\ (a * b + \mathit{div}\ (\mathit{resolution}\ x)\ 2)\\ \qquad\qquad\qquad\qquad (\mathit{resolution}\ x)))\\ \quad \mathrm{D}\ a + \mathrm{D}\ b \quad\ \ = \mathrm{D}\ (a + b)\\ \quad \mathrm{D}\ a - \mathrm{D}\ b \quad\ \ = \mathrm{D}\ (a - b)\\ \quad \mathit{negate}\ \ (\mathrm{D}\ a) = \mathrm{D}\ (\mathit{negate}\ a)\\ \quad \mathit{abs}\qquad (\mathrm{D}\ a) = \mathrm{D}\ (\mathit{abs}\ a)\end{array}$$

$\mathit{signum}\ (\mathrm{D}\ a) = \mathrm{D}\ (\mathit{signum}\ a)$
$\mathit{fromInteger}\ i = \mathrm{D}\ (\mathit{fromInteger}\ i)$

In the Fractional instance, we delegate everything except division.

$\mathbf{instance}\ \mathrm{HasResolution}\ e \Rightarrow \mathrm{Fractional}\ (\mathrm{Decimal}\ e)\ \mathbf{where}$
$\quad x@(\mathrm{D}\ (\mathrm{MkFixed}\ a))\ /\ \mathrm{D}\ (\mathrm{MkFixed}\ b) =$
$\quad\quad \mathrm{D}\ (\mathrm{MkFixed}\ (\mathit{div}\ (a * \mathit{resolution}\ x + \mathit{div}\ b\ 2)\ b))$
$\quad \mathit{recip}\ (\mathrm{D}\ a) \quad = \mathrm{D}\ (\mathit{recip}\ a)$
$\quad \mathit{fromRational}\ r = \mathrm{D}\ (\mathit{fromRational}\ r)$

We define the E18 and E36 symbols and their fixed point multipliers.

$\mathbf{data}\ \mathrm{E18}; \mathbf{data}\ \mathrm{E36}$
$\mathbf{instance}\ \mathrm{HasResolution}\ \mathrm{E18}\ \mathbf{where}$
$\quad \mathit{resolution}\ \_ = 10 \uparrow (18 :: \mathrm{Integer})$
$\mathbf{instance}\ \mathrm{HasResolution}\ \mathrm{E36}\ \mathbf{where}$
$\quad \mathit{resolution}\ \_ = 10 \uparrow (36 :: \mathrm{Integer})$

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

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