Yamfore

Whitepaper

By Big BLYMP

This page has been left intentionally blank

Contents

Contents 2
I. Introduction 4
For users 5
For Admins 7
II. Constants 9
III. Tokens 10
IV. Externals 12
V. Validators 14
PF: Price Feed 15
Datatypes 15
Constraints 15
Notes 16
PFP: Price Feed Pointer 17
Datatypes 17
Constraints 17
Gov: Governance 18
Datatypes 19
M: Main 19
Datatypes 20
Constraints 20
Notes 24
VI. Functions 25
Rationals* 25
Fees, collateral, and repayment 26
In validators 27
Other 28
VII. Examples 29
Example A 29
VIII. Transactions 30
`tx.pfp_init` 31

`tx.pfp_update` 31
`tx.pfp_close` 32
`tx.gov_init` 33
`tx.gov_update` 33
`tx.gov_close` 33
`tx.m_init` 33
`tx.m_publish` 34
`tx.m_close` 34
`tx.borrow` 35
`tx.repay` 37
`tx.exchange_cblp` 38
`tx.exchange_ada` 39
`tx.m_withdraw` 39
IX. Comments 41
Weaknesses 41
X. Appendix 43
Glossary 43
Documentation Issues 43

I. Introduction

Yamfore is a decentralized, non-custodial lending protocol built on the Cardano blockchain. It enables users to obtain crypto-backed loans using ADA, with the following features:

- **No margin calls:** Users' loans won't be liquidated due to price volatility.
- **No ongoing interest payments:** Borrowers only pay a fixed fee.
- **Indefinite loan terms:** Loans don’t expire unless the borrower chooses to repay.

CBLP is the native utility token of the protocol, used to pay loan fees and maintain system operations. Users acquire CBLP before borrowing, either through external platforms or protocol-specific mechanisms like auctions.

This document includes:

1. A plain explanation of the dApp's functionality.
2. Technical details, such as validators, constants, tokens, and transactions (tx).
3. Commentary on limitations and additional terms.

Throughout this document, we will refer to the stablecoin as USD as a convenient shorthand.

Non-technical litepaper can be found here, and other documents can be found here.

For users

The primary function of the dApp is to facilitate ADA-backed stablecoin loans. Users interact with the protocol through two main processes: borrowing and repayment.

Borrowing USD

When a user initiates a borrow transaction:

- **USD Availability**: The user receives USD, collateralized by locked ADA.
- **Collateralization**: The ADA is locked as collateral.
- **Fee Payment**: The user pays a fee in CBLP (the native utility token).
- **Token Minting**: Two tokens are minted to represent validity and authorization:
    - One token is held with the collateral.
    - The other is provided to the user.

The required fee and collateral amounts are determined by:

- **Loan Size**: The total USD borrowed.
- **Price Feed Data**: Real-time exchange rates (ADA/USD and CBLP/ADA).
- **Governance Parameters**: Protocol-defined constants and rates.

Repaying Loans

When a user repays their loan:

- **USD Repayment**: The user deposits the borrowed USD at the script address.
- **Collateral Unlocking**: The locked ADA collateral becomes available to the user.
- **Token Burning**: Validity and authorization tokens are burned.

The repayment amount is calculated based on:

- **Loan Size**: The total USD borrowed.
- **Loan Duration**: The time elapsed since borrowing.
- **Governance Parameters**: Protocol-defined rates and adjustments.

Acquiring CBLP

Before borrowing, users must acquire CBLP. This can be achieved through:

- **External Platforms**: Purchasing CBLP on a decentralized exchange (DEX).
- **Protocol Mechanisms**: Acquiring CBLP via the First Token Offering (FTO) or auction portal (out-of-scope for V2 testnet).
- **Internal Exchange**: Exchanging USD for CBLP or ADA directly through the protocol.

Deposit Staking and Rewards

All deposits made to the protocol are staked under the script credentials. Key details include:

- **Reward Accumulation**: Staked deposits earn rewards over time.
- **Reward Withdrawal**: Users can withdraw rewards to the script address.
- **Collateral Exchange**: Locked ADA can be exchanged or withdrawn as necessary.

For Admins

Administrators are integral to the functionality and sustainability of the Yamfore protocol. They oversee tasks such as token creation, governance, price feed management, interest rate maintenance, and stablecoin integration, which ensure the smooth operation and adaptability of the dApp.

CBLP Management

- **Creation**: Admins are responsible for the creation and management of CBLP, the native utility token. While the token has already been created, it may need to be reenacted for testing purposes during deployments.
- **FTO Organization**: Admins organize the First Token Offering (FTO), which operates as a single-batch, non-refundable auction.

    - Funds raised through the FTO are funneled into the auction portal (these mechanisms are outside the scope of the V2 testnet).

Price Feed Management

- **Data Requirements**: The dApp relies on near real-time exchange rate data for USD, ADA, and CBLP.
- **Reliability**: Admins ensure that price feeds remain accurate and accessible, sourcing data from external validators or platforms.
- **Upgrade Capability**: Admins can upgrade the price feed wrapper, which uses data sources likely external to the dApp, to address issues with unreliable sources and ensure the protocol's resilience.

Staking Management

- **Locked ADA**: ADA collateral from loans is staked, and Admins are responsible for delegating this collateral to appropriate pools.
- **Maximizing Returns**: Proper staking management ensures optimal returns while maintaining system integrity.

Stablecoin Integration

- **Selection**: Admins determine which stablecoins are eligible for use within the protocol, considering the growing variety available on the Cardano network.

- **Infrastructure Setup**: Both on-chain and off-chain mechanisms necessary for stablecoin interaction are initialized and maintained by Admins.

Through these critical roles, Admins ensure that Yamfore remains stable, secure, and responsive to the needs of its users and the broader ecosystem.

II. Constants

These constants are referenced at various points in the dApp:

```
tag_length = 20
short_time = 20 * 60 * 1000 // 20 minutes
max_usd_vault_amt = 50_000_000_000 // $50,000
max_vault_outputs = 10
epsilon = 10
```

III. Tokens

The following describes the token names:

```
// CIP68 style prefixes
auth_pref = #"000643b0" // cip68 "222"
validity_pref = #"000de140" // cip68 label "100"


// Already exists
cblp_label = "CBLP"


// Gov auth/validity
gov_label = "gov"
gov_auth = auth_pref + gov_label
gov_validity = validity_pref + gov_label


// Placeholder for stablecoin name
usd = "usd"


// Pfp auth/validity
pfp_label = "pfp"
pfp_auth = auth_pref + pfp_label
pfp_validity = validity_pref + pfp_label


// Main auth
admin = "admin"


// Main user auth/validity
user_label = "yamfore"
user_auth = auth_pref + user_label + <tag>
user_validity = validity_pref + user_label + <tag>
```

Each pair of user twin tokens uses an input utxo oref as a seed to create an essentially unique tag `<tag>`.

The function `mk_tag : OutputReference -> Tag` uses `blake2b_256` plutus builtin as a hash function, and truncates the result to a 20-byte (set by `constants.tag_length`) (160-bit) digest. It leaves 12 bytes of space in token names for human-friendly labels.

An explanation of why we use cip68 labels while not being cip68 compliant can be found here.

IV. Externals

The Yamfore dApp relies on several on-chain external validators to support its operations. These external components provide crucial data and functionality that, while outside the direct control of Yamfore, are essential to the protocol's effectiveness.

CBLP Integration

CBLP, the native utility token of Yamfore, is treated as an external component for specific purposes. Although directly tied to the protocol, CBLP exists on the Cardano mainnet and can appear "hardcoded" in validators. For testing scenarios, treating CBLP as a variable offers greater flexibility, enabling simulations and adjustments as needed.

Exchange Rates

The protocol requires near real-time exchange rate data for the following pairs:

- **ADA/USD**: Exchange rate between ADA and USD.
- **ADA/CBLP**: Exchange rate between ADA and CBLP.

These rates can be sourced by analyzing liquidity pools (LPs) from decentralized exchanges (DEXs), such as Minswap. Accurate and up-to-date exchange rate data is critical for ensuring the reliability of calculations involving borrowing, repayment, and collateral requirements.

Upgradable Data Layer

Given the reliance on external data sources, the dApp incorporates an upgradable layer between these sources and its internal operations. This layer is designed to:

- **Handle Unreliable Sources**: Ensure the protocol remains operational even if a specific data source becomes unavailable or inaccurate.
- **Enable Adaptability**: Allow admins to update or replace data sources as required, maintaining the dApp's integrity and reliability.

By effectively leveraging external validators and maintaining a robust interface for data integration, Yamfore ensures consistent functionality while adapting to changes within the Cardano ecosystem.

V. Validators

The on-chain component of the Yamfore dApp consists of multiple validators, each performing specific roles within the protocol. These validators may interact and rely on one another, forming a network of dependencies that is critical for the dApp's functionality.

Dependency Types

- **Hard-Dependency**: A validator is considered hard-dependent on another when its hash is explicitly "hard-coded" into the dependent validator. This creates a fixed relationship between the two.
- **Soft-Dependency**: In a soft dependency, one validator relies on another, but the dependency is flexible and can be modified if necessary.

Acyclic Dependency Graph

To ensure system integrity and prevent circular logic, the graph of hard-dependencies must remain acyclic. This means:

- If Validator A is hard-dependent on Validator B, then Validator B cannot, directly or indirectly, be hard-dependent on Validator A.
- Soft dependencies, however, allow Validator B to rely on Validator A without violating this principle.

This structure ensures that the validators operate in a clear, logical sequence, maintaining the protocol's stability and predictability.

```
mermaid
flowchart LR
    gov["
    Gov : Governance \n
    Spend, Mint
    "]
    pfp["
    Pfp : PriceFeedPointer\n
    Spend, Mint
    "]
    pf["
    Pf : PriceFeed\n
    Withdraw
    "]
    m["
    M : Main \n
    Spend, Mint, Withdraw, Publish
    "]
    gov & pfp --> m
    pf -.-> pfp & m
```

PF: Price Feed

The validator checks the presence of the necessary oracles and verifies that its own redeemer matches the exchange rate attested to by the oracle(s).

Datatypes

See the section on functions for explanations of how this is used.

```
aiken-language
type Pf2Red {
  c_num : Int,
  c_denom : Int,
  u_num : Int,
  u_denom : Int,
}
```

Constraints

Two arguments - Withdraw purpose.

Redeemer is `Pf2Red`:

1. Ascertain the true exchange rates from oracles or otherwise.
2. The numbers in the redeemer correspond to them.

Notes

We implement a few different versions.

Datum version: Loosely modelled on orcfax and uses reference inputs to discern correctness. We currently do not have an oracle system on which we can depend so we have implemented our own skeletal form for testing. This is the default implementation, with the label `pf`.

The script is parameterized by the hashes of the oracles. Reference inputs are identified by containing a recognized token.

Signature version: Relies on a trusted signature to have signed the redeemer. If there is a usable oracle on mainnet at launch, this is the suggested first implementation. This implementation has the label `pf_sig`.

Because the redeemer has been designed with the main script in mind,

it does not accommodate the additional data needed to carry out validation. Namely, a time stamp and signature.

To work around this, we supply the additional data via the redeemer of a second withdraw script. The second script does nothing.

`pf_sig` checks that:

- The pf redeemer has been signed off together with the time stamp
- The timestamp is within the short tx validity range

Emergency version: Relies on a trusted signature on the tx. The label is `pf_em`.

PFP: Price Feed Pointer

This spend/mint validator 'points' at the PF. The pointer consists of a utxo with an inline datum recording the current PF script hash. The utxo contains a validity token twinned with one held by admin.

Datatypes

Outlined below:

```
aiken-language
type PfpParams = OutputReference
type PfpDat = Credential
type Pfp3Red {
  Pfp3Update(Int, Int)
  Pfp3Close
}
type Pfp2Red {
  Pfp2Init
  Pfp2Burn
}
```

Constraints

Two arguments - Mint purpose:

1. When `red` is `Pfp2Init`
    a. `PfpParams` is spent

    b. Own mint value is
        i. (erp-auth, 1)
        ii. (erp-validity, 1)
    c. 0th output is continuing output:
        i. Payment address is own
        ii. Value is ADA and erp-validity
        iii. Datum is `PfpDat`
2. When `red` is `Pfp2Burn`
    a. Own mint has only negative quantities

Three arguments:

1. When `red` is `Pfp3Update(auth_idx, cont_idx)`
    a. Output `auth_idx` is auth output _ie_ contains erp-auth
    b. Output `cont_idx` is continuing output
        i. Payment address is own
        ii. Value is ADA and erp-validity
        iii. Datum is `PfpDat`
2. When `red` is `Pfp3Close`
    a. Own mint value has length 2. (Implies both tokens are burnt)

Gov: Governance

The `gov` validator's behaviour is identical to that of the `pfp`. Note that the `pfp` datum plays no role in its own constraints. `gov` has a different datum, stated here. We omit repeating the other parts.

Datatypes

```
aiken-language
// TODO
type GovDat {
  interest_rate: (Int, Int),
  cblp_fee: (Int, Int),
  exchange_reward: (Int, Int),
}
```

M: Main

The main validator can be sensibly invoked with any purpose: spend, mint, withdraw, publish.

There are:

- Four spend actions all intended for users:
    - Borrow,
    - Repay,
    - Exchange ADA,
    - Exchange CBLP.
- Two mint actions intended for users:
    - Mint,
    - Burn.
- Two mint actions intended for admin:
    - AdminMint,
    - AdminBurn.
- A single withdraw action:
    - Withdraw.
- A single publish action:

- Publish.

Datatypes

```
aiken-language
type MParams {
  admin_seed: OutputReference,
  gov_hash: ByteArray,
  pfp_hash: ByteArray,
  cblp_hash: ByteArray,
  usd_hash: ByteArray,
}
type MRed2 {
  Mint(OutputReference) // Tag seed
  Burn
  AdminMint
  AdminBurn
  Withdraw
  Publish
}
type MRed3 {
  Borrow
  Repay(Int) // n_vaults
  Exchange
}
type MDat {
  UsdVault
  CblpVault(Int) // Available from slot
  AdaVault
  Position(PositionP)
}
type PositionP {
  created_at : Int,
  borrow_amt : Int,
  interest_rate : (Int, Int),
}
```

Constraints

Three arguments:

1. When own redeemer is `Borrow`:
    a. Own mint value is `[(_, 1), (_, 1)]`
2. When own redeemer is `Repay(n_vaults)`:

a. Own datum is `Position({created_at, borrow_amt, interest_rate})`
b. No other inputs with own payment credential.
c. Validity range has upper bound `ub`
d. Calculate duration `duration = ub - created_at`
e. 0th output is repayment output
    i. Address is own address
    ii. Value is ada and `repay_amt` of USD
    iii. Datum is `UsdVault`
f. If `n_vaults > 1`, then then next `n_vaults - 1` outputs are identical
g. If `n_vaults < max_n_vaults`, then `amt < max_usd_vault_amt`
h. `total_repay_amt = n_vault * repay_amt`
i. Verify sufficient repay amount
j. Get own tag `tag`
k. Own mint value is `[(user_auth_<tag>, -1), (user_validity_<tag>, -1)]`

3. When own redeemer is `Exchange`:
    a. All own spends are `Exchange`, implied by own mint value is empty
    b. Defer all validation to 0th own spend
    c. Get gov params from ref inputs via gov token
    d. Get price feed data
    e. 0th output is USD vault output
        i. Address is own address
        ii. Value is ADA and `usd_out`
        iii. Datum is `UsdVault`
    f. If own datum is `CblpVault(_)` then
        i. Validity interval has lower bound `lb`
        ii. Reduce `own_inputs` to `(total, count)`
            1. All `own_inputs` have datum `CblpVault(from)`
            2. Either `from <= lb`, or admin token is spent
            3. All value is ADA and CBLP
        iii. Calculate expected leftover amount `o_expect`
        iv. If `o_expect <= epsilon`, then return true, else 1st output is leftover output
            1. Address is own address

2. Value is at least `o_expect` CBLP
3. The are no unnecessary vault inputs (proxy by average)
4. Datum is `CblpVault(0)`

g. Else own datum is `AdaVault`
   i. Reduce `own_inputs` to `(total, count)`
      1. All `own_inputs` have datum `AdaVault`
      2. All value is only ADA
   ii. Calculate expected leftover amount `o_expect`
   iii. If `o_expect <= epsilon`, then return true, else 1st output is leftover output
      1. Address is own address
      2. Value is at least `o_expect` ADA
      3. The are no unnecessary vault inputs (proxy by average)
      4. Datum is `AdaVault`

Two arguments - Mint purpose:

1. When own redeemer is `Mint(seed):
   a. Get gov params from ref inputs via gov token
   b. Get price feed data
   c. `seed` is spent
   d. Make tag `tag` from `seed`
   e. Fold over inputs
      i. Filter own payment credential
      ii. Expect only `UsdVault` datums
      iii. Aggregate `(count, usd_input_amt)`
   f. Validity range has lower bound `lb`
   g. 0th output is position
      i. Address is own address
      ii. Value is `collateral` ADA and validity token with tag `tag`
      iii. Datum is `Position({ created_at : lb, interest_rate : gov_dat.interest_rate, borrow_amt })`
      iv. Verify sufficient collateral
   h. 1st output is fee output

        i. Address is own address
        ii. Value is ADA and `fee_amt` CBLP
        iii. Datum is `CblpVault(lb)`
        iv. Verify sufficient fee
    i. If `usd_input_amt > borrow_amt` then, next output is USD vault output
        i. Address is own address
        ii. Value is ADA and `usd_output_amt` USD.
        iii. Datum is `UsdVault`
        iv. No redundant inputs `usd_output_amt <= usd_input_amt / count`
    j. Own mint value is `[(user_auth_<tag>, 1), (user_validity_<tag>, 1)]`
2. When own redeemer is `Burn`:
    a. Own mint value is of the form `[(user_auth_<tag>, -1), (user_validity_<tag>, -1)]`
3. When own redeemer is `AdminMint`
    a. `params.admin_seed` is spent
    b. Own mint value is `[(admin, 1)]`
4. When own redeemer is `AdminBurn`
    a. Own mint value is `[(admin, -1)]`
5. Otherwise, fail

Two arguments - Withdraw purpose:

1. Redeemer is `Withdraw` (else fail)
2. Own withdraw amount is `amt`
3. 0th output is ADA vault output
    a. Address is own address
    b. Value is ADA of at least `amt`
    c. Datum is `AdaVault`

Two arguments - Publish purpose:

1. Redeemer is `Publish` (else fail)
2. Input contains own admin token

Notes

1. **Some spending logic is deferred to mint.**

There are some global constraints utilized here to ensure only certain pairs of mint and actions are performed in a tx. Namely, assumptions made on which logic is executed based on the form of the mint value.

For example, spending `UsdVault` checks own mint value of the form `[(_,1), (_, 1)]`. We know this can only be the case if the validator is executed with mint purpose and redeemer `Borrow`.

Explicit coupling:

- + Spend:Borrow => Mint:Mint
- + Spend:Repay => Mint:Burn
- + Spend:Exchange => None

2. **The withdrawal purpose enforces the 0th output as an AdaVault.**

No other tx permits a 0th output as an AdaVault. This prevents a double satisfaction.

VI. Functions

Rationals*

**Note: this works like Obymare.*

Suppose `u` is a rational number representing USD to ADA conversion. It can be read as _ada per dollar_. That is, `u lovelace = 1 microdollar`, equivalently `u ada = 1 dollar`.

Suppose `c` is a rational number representing CBLP to ADA conversion. That is, `c lovelace = 1 microcbpl`, equivalently `c ada = 1 cblp`. The task of describing the constraints is made mildly more laborious in the absence of rationals.

However, it is well beyond our needs to introduce full support for rationals. Let us represent the rational `a` as a 2-tuple, `a = (a_num, a_denom)`, and assume `a_denom > 0`. Thus, for example, if `x` amount of ADA is equivalent to `y` amount of USD then `u_denom * x == u_num * y`.

Suppose we have A ADA and C CBLP.

Then the value U in USD is:

```
$$
    U = \frac{A + Cc}{u}
$$
```

Expressed in code, this conversion is:

```
 usd = (ada * c_denom + cblp * c_num) * u_denom / (u_num *
c_denom)
```

Fees, collateral, and repayment

Suppose a user does borrow from the dApp with the following quantities:

- B borrowed amount, measured in USD
- C required collateral amount, measured in ADA
- u the exchange rate USD to ADA (ie u ADA is equivalent to 1 USD)
- F required fee amount, measured in CBLP
- c the exchange rate CBLP to ADA (ie c ada is equivalent to 1 CBLP)
- f the fee rate (TODO : Be precise)
- D the duration between borrowing and repayment (TODO : what units?)
- i the (non-cumulative) interest rate at time of borrowing (TODO : what units?)
- R repay amount, measured in USD

Satisfying the minimum collateral requirement:

```
$$
    C = 2Bu
$$
```

Satisfying the minimum fee requirement:

```
$$
    F = \frac{fC}{c}
$$
```

Satisfying the minimum repayment requirement:

```
$$
    R =  (1 + i D)B
$$
```

Suppose a user does an exchange with the dApp with the following quantities:

- E the input amount, measured in USD
- r the exchange reward

- I input from vaults, in either ADA or CBLP
- O output to vaults, in either ADA or CBLP

In the ADA case:

```
$$
    O = I - ruE
$$
```

In the CBLP case:

```
$$
    O = I - ruE/c
$$
```

In validators

Here we relabel some variables:

- B `borrowed`
- C `collateral`
- F `fee`
- D `duration`

We 'flatten' the formulae and express them as an inequality, checking the sufficient.

Sufficient collateral requirement:

```
collateral * u_denom >= 2 * u_num * borrowed
```

Sufficient fee requirement:

```
fee * c_num * f_denom >=  f_num * collateral * c_denom
```

Sufficient repayment requirement:

```
repay * i_denom >= (i_denom + i_num * duration) * borrowed
```

In an exchange, the leftover output exists only if the expected leftover output amount is positive. Thus, it is easier overall to compute this value explicitly than check if an inequality is satisfied.

Expected leftover output is:

```
expect_ada = i_ada - (r_num * u_num * exchanged_usd) /
(r_denom * u_denom)

expect_cblp = i_cblp - (c_denom * r_num * u_num *
exchanged_usd) / (c_num * r_denom * u_denom)
```

Other

Calculate the number `n_vaults` of USD vault outputs and the amount `amt` of USD in each. Let `total_usd` be the total USD.

Then:

```
n_vaults = min(max_vault_outputs, total_usd/max_usd_vault_amt
+ 1)

amt = total_usd/n_vaults + 1
```

The following constraint is used in several places: The are no unnecessary vault inputs (proxy by average). By this, we mean that the value of the tokens returned must be no more than the average input.

```
change_amt <= total_amt / n_inputs
```

In the utxo selection find a set of vaults that cover the desired value, then remove the utxos of the least value until this condition is satisfied.

VII. Examples

Example A

Let:

- $B = 10_000_000_000$, equivalent to \$10,000
- $u = 7/3$, equivalent 2.3333 ADA/USD
- $c = 5629/250000$ equivalent to 0.022516 ADA/CBLP the exchange rate CBLP to ADA
- $f = 19/100$, a fee rate of 19%
- $D = 600 * 24 * 60 * 60 * 1000$ 600 day duration
- $i = 1/(25 * 365 * 24 * 60 * 1000)$ is 4% non-cumulative annual interest rate at the time of borrow

Then the collateral is:

```
$$
    C = 2 * 10_000_000_000 * 7 /3 = 46_666_666_666
$$
```

About $47k$ ADA.

The fee:

```
$$
    F = 19 * 46_666_666_6666 * 250000 / 100 * 5629 =
    393_794_042_758
$$

$$
     R = (25 * 365 + 600) * 10_000_000_000 / 25 * 365 =
     10_657_534_245
$$
```

VIII. Transactions

A transaction consists of inputs and outputs of utxos*. Each utxo has a:

- Unique reference
- An address it belongs to
- A value of ADA + native assets (possibly no native assets)
- Maybe a datum or datum hash

**Note that a utxo at a script address must have a datum or datum hash to be spendable.*

Addresses correspond to either agents (e.g., normal addresses) or script addresses. All transactions must have at least one input that belongs to a payment address, i.e., an agent's address. When there are no constraints as to who the payment address belongs to, we may write _Any_ as the address owner.

Addresses, unless otherwise stated, refer to only the payment credential while the staking credential is unrestricted. The diagrams should be indicative and informative rather than provide an exhaustive description.

In the following diagrams, boxes denote utxos. The address's owners are stated first, followed by any notable assets or ada (that is, ignoring min ada or ada value doesn't change or change is negligible) and datums where relevant. All datums are assumed inline unless stated otherwise. A download icon indicates withdrawals.

`tx.pfp_init`

Admin initializes the pointer, creating validity/auth twins. This tx is _seeded_ to ensure that the policy IDs and script address cannot be duplicated.

```
mermaid
flowchart LR
    i_admin["
    fa:fa-address-book Admin\n
    oref: seed
    "]
    s_pfp["
    fa:fa-circle PFP\n
    params: seed
    "]

    m["tx"]

    o_pfp["
    fa:fa-address-book PFP\n
    fa:fa-circle pfp-validity \n
    fa:fa-database PfpDat
    "]

    o_admin["
    fa:fa-address-book Admin\n
    fa:fa-circle pfp-auth \n
    "]
    i_admin--> m --> o_pfp & o_admin
    s_pfp -.->|Pfp2Init| m
```

`tx.pfp_update`

Admin updates the pointer datum.

```
mermaid
flowchart LR
    i_pfp["
    fa:fa-address-book PFP\n
    fa:fa-circle pfp-validity \n
    fa:fa-database PfpDat
    "]
```

```
    i_admin["
    fa:fa-address-book Admin\n
    fa:fa-circle pfp-auth \n
    "]

    m["tx"]

    o_pfp["
    fa:fa-address-book PFP\n
    fa:fa-circle pfp-validity\n
    fa:fa-database PfpDat
    "]

    o_admin["
    fa:fa-address-book Admin\n
    fa:fa-circle pfp-auth\n
    "]

    i_pfp -->|Pfp3Update| m
    i_admin --> m --> o_pfp & o_admin
```

`tx.pfp_close`

Admin closes the pointer datum, and burns auth/validity tokens.

```
mermaid
flowchart LR
    i_pfp["
    fa:fa-address-book PFP\n
    fa:fa-circle pfp-validity \n
    fa:fa-database PfpDat
    "]

    s_pfp["
    fa:fa-circle PFP\n
    "]

    i_admin["
    fa:fa-address-book Admin\n
    fa:fa-circle pfp-auth \n
    "]

    m["tx"]

    i_admin --> m
```

```
    i_pfp -->|Pfp3Close| m
    s_pfp -->|Pfp2Burn| m
```

`tx.gov_init`

This is analogous to the `tx.pfp_init`.

`tx.gov_update`

This is analogous to the `tx.pfp_update`.

`tx.gov_close`

This is analogous to the `tx.pfp_close`.

`tx.m_init`

Admin inits the main validator, minting the auth token.

```
mermaid
flowchart LR
    i_admin["
    fa:fa-address-book Admin\n
    oref: seed
    "]

    s_m["
    fa:fa-circle M\n
    params: seed ...
    "]

    m["tx"]

    o_admin["
    fa:fa-address-book Admin\n
    fa:fa-circle m-auth \n
    "]

    i_admin--> m --> o_admin
```

```
    s_m -.->|AdminMint| m
```

`tx.m_publish`

Admin delegates, re-delegates, or un-delegates the stake address. The auth token must be spent.

```
mermaid
flowchart LR
    i_admin["
    fa:fa-address-book Admin\n
    fa:fa-circle m-auth \n
    "]

    p_m["
    fa:fa-download M\n
    "]

    m["tx"]

    o_admin["
    fa:fa-address-book Admin\n
    fa:fa-circle m-auth \n
    "]

    i_admin--> m --> o_admin
    p_m -.->|Publish| m
```

`tx.m_close`

Admin delegates, re-delegates, or un-delegates the stake address. The auth token must be spent.

```
mermaid
 flowchart LR
     i_admin["
     fa:fa-address-book Admin\n
     fa:fa-circle m-auth \n
     "]

     p_m["
```

```
     fa:fa-circle M\n
     "]

     m["tx"]

     o_admin["
     fa:fa-address-book Admin\n
     "]

     i_admin--> m --> o_admin
     p_m -.->|AdminBurn| m
```

`tx.borrow`

The user borrows USD while locking up ADA as collateral. The tx must reference governance and price feed data.

This example spends from two USD vaults:

```
mermaid
flowchart LR
    i_user["
    fa:fa-address-book User\n
    fa:fa-circle ada, cblp \n
    oref: seed
    "]

    i_usd_vault_0["
    fa:fa-address-book M\n
    fa:fa-circle usd\n
    fa:fa-database UsdVault
    "]

    i_usd_vault_1["
    fa:fa-address-book M\n
    fa:fa-circle usd\n
    fa:fa-database UsdVault
    "]

    p_m["
    fa:fa-circle M\n
    "]
```

```
    r_pfp["
    fa:fa-address-book Pfp\n
    fa:fa-circle pfp-validity\n
    fa:fa-database PfCredentials
    "]

    r_gov["
    fa:fa-address-book Gov\n
    fa:fa-circle gov-validity\n
    fa:fa-database GovDat
    "]

    w_pf["
    fa:fa-download PF\n
    "]

    m["tx"]

    o_position["
    fa:fa-address-book M\n
    fa:fa-circle ada, user-validity
    fa:fa-database Position(now)
    "]

    o_fee["
    fa:fa-address-book M\n
    fa:fa-circle cblp
    fa:fa-database CblpVault
    "]

    o_usd_vault["
    fa:fa-address-book M\n
    fa:fa-circle usd
    fa:fa-database UsdVault
    "]

    o_user["
    fa:fa-address-book User\n
    fa:fa-circle usd, user-auth
    "]

    i_user --> m
    i_usd_vault_0 & i_usd_vault_1 -->|Borrow| m -->
o_position & o_fee & o_usd_vault
    m --> o_user
    p_m -.->|Mint| m
    w_pf -.->|PriceFeed| m
    r_pfp & r_gov -.-o m
```

`tx.repay`

The user repays USD unlocking their ada collateral. This example outputs two USD vaults.

```
mermaid
flowchart LR
    i_user["
    fa:fa-address-book User\n
    fa:fa-circle usd, user-auth
    "]

    i_position["
    fa:fa-address-book M\n
    fa:fa-circle ada, user-validity
    fa:fa-database Position
    "]

    p_m["
    fa:fa-circle M\n
    "]

    m["tx"]

    o_usd_vault_0["
    fa:fa-address-book M\n
    fa:fa-circle usd\n
    fa:fa-database UsdVault
    "]

    o_usd_vault_1["
    fa:fa-address-book M\n
    fa:fa-circle usd\n
    fa:fa-database UsdVault
    "]

    i_user["
    fa:fa-address-book User\n
    fa:fa-circle Ada \n
    "]

    i_user --> m
    i_position -->|Repay| m --> o_usd_vault_0 & o_usd_vault_1
    p_m -.->|Burn| m
```

`tx.exchange_cblp`

A user exchanges USD for CBLP. In this example, we have 3 script inputs, and the remaining (un-exchanged) CBLP is returned as an output.

```
mermaid
flowchart LR
    i_user["
    fa:fa-address-book User\n
    fa:fa-circle usd,
    "]

    i_cblp_vault_0["
    fa:fa-address-book M\n
    fa:fa-circle ada, cblp
    fa:fa-database CblpVault
    "]

    i_cblp_vault_1["
    fa:fa-address-book M\n
    fa:fa-circle ada, cblp
    fa:fa-database CblpVault
    "]

    i_cblp_vault_2["
    fa:fa-address-book M\n
    fa:fa-circle ada, cblp
    fa:fa-database CblpVault
    "]


    r_pfp["
    fa:fa-address-book Pfp\n
    fa:fa-circle pfp-validity\n
    fa:fa-database PfCredentials
    "]

    w_pf["
    fa:fa-download PF\n
    "]

    m["tx"]

    o_usd_vault["
    fa:fa-address-book M\n
    fa:fa-circle usd\n
```

```
    fa:fa-database UsdVault
    "]

    o_cblp_vault["
    fa:fa-address-book M\n
    fa:fa-circle ada, cblp
    fa:fa-database CblpVault
    "]

    o_user["
    fa:fa-address-book User\n
    fa:fa-circl ada, cblp \n
    "]

    i_user --> m
    i_cblp_vault_0 & i_cblp_vault_1 & i_cblp_vault_2
-->|Exchange| m --> o_usd_vault & o_cblp_vault
    m --> o_user
    w_pf -.->|PriceFeed| m
    r_pfp -.-o m
```

`tx.exchange_ada`

A user exchanges USD for ADA. This is completely analogous to `tx.exchange_cblp`, where the datums are all `AdaVault` instead of `CblpVault`. In this example, we have 3 script inputs, and the remaining (un-exchanged) CBLP is returned as an output.

`tx.m_withdraw`

```
mermaid
flowchart LR
    i_user["
    fa:fa-address-book User\n
    "]

    w_m["
    fa:fa-download M\n
    "]

    m["tx"]
```

```
    o_cblp_vault["
    fa:fa-address-book M\n
    fa:fa-circle ada\n
    fa:fa-database CblpVault
    "]

    i_user --> m
    m --> o_cblp_vault
    w_m -.->|Withdraw| m
```

IX. Comments

Weaknesses

Oracles

Oracles play a critical role in providing external data to the blockchain. However, their reliance on off-chain data introduces inherent limitations.

Oracles cannot inherently provide the same level of validity guarantees as on-chain data. This means any protocol depending on an oracle is, to some extent, vulnerable to the reliability and accuracy of external data sources.

The Role of Oracles in Yamfore

The Yamfore dApp requires an oracle to provide the stablecoin price relative to ADA. While there are on-chain proxies that could potentially fulfill this role, the protocol has chosen to use oracles for two key reasons:

- **Feasibility**: Utilizing automated market maker (AMM) liquidity pools as a price source is impractical due to the rapid changes in liquidity and prices. This volatility could cause user transactions to fail by becoming stale before reaching the blockchain nodes.
- **Longevity**: The protocol cannot rely on third-party mechanisms or platforms to operate indefinitely. Oracles offer a more adaptable solution to ensure the dApp's long-term functionality.

Updatable Oracle Mechanism

To address these challenges, Yamfore employs an updatable oracle mechanism that allows for flexibility and governance:

- **Governance Transition**: Over time and under suitable conditions, the oracle's update mechanism can be transitioned to a governance committee (e.g., multisig governance).
- **Potential Phase-Out**: If a stablecoin eventually offers its own reliable price feed, the oracle mechanism can be retired entirely.

By adopting this approach, Yamfore balances feasibility and longevity, ensuring that the protocol remains robust and adaptable while minimizing reliance on potentially volatile or transient data sources.

X. Appendix

Glossary

- Validator
    - A Plutus script which contains the logic for what is necessary to spend the funds locked at its address or mint/burn assets of a specific PolicyID. For more info see here
- Redeemer
    - Input data provided to a script. For more info see here

Documentation Issues

- Mermaidjs bug: direction of subgraphs untameable