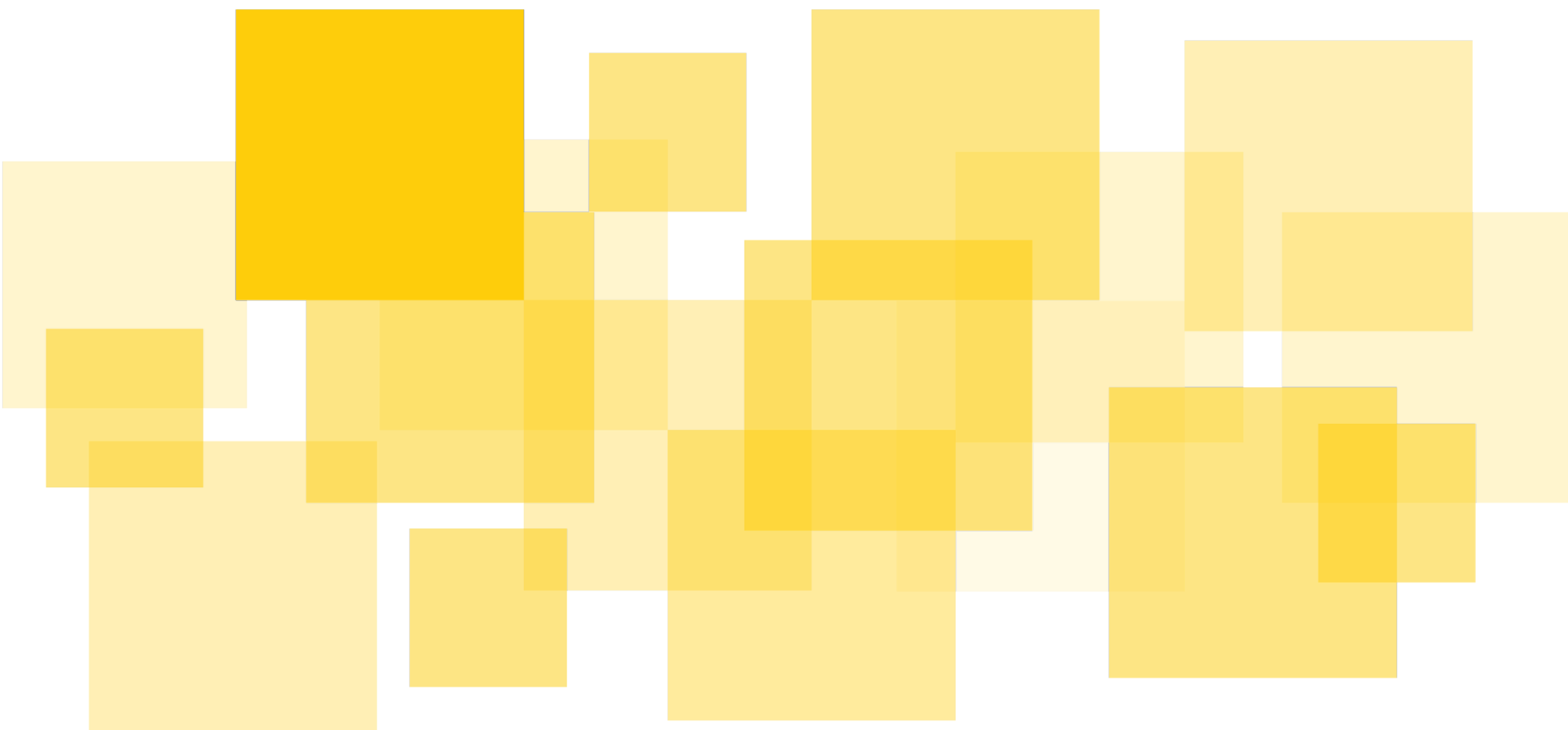
Audit Report

HydraDX Omnipool

First Delivered: 7 Sep, 2022

Followup Audit Report Delivered: 19 Oct, 2022



Prepared for HydraDX by Runtime Verification, Inc.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Summary

HydraDX team engaged Runtime Verification, Inc. in a full audit of their AMM protocol, Omnipool, on a Polkadot parachain. Omnipool protocol is a constant product market maker (CPMM). It has an innovative design of the hub asset, the LRNA (pronounces LERNA) token, which was introduced to solve the capital inefficiency problem (allowing single asset liquidity provision and breaking the boundaries of pools limited to allow asset pairs) in traditional AMMs.

Scope

The targeted code for auditing is written in Rust and developed based on the Substrate Platform of Parity. We had audited the source code for the HydraDX Omnipool protocol in two separate audits. Both reports are included in this final report.

First audit which is a formal audit of this protocol was conducted from 2022-07-25 to 2022-09-02 with the following scope:

The review encompassed 3 public code repositories with code frozen for review at commits:

- HydraDX-node: 8de6222f967cc1bfe120207a775e8f0dc8eb92a5
- HydraDX-math: f4dab244bb5d57971325f0ba6fbb8f6c1353beeb
- The registry trait of warehouse: 677a8880e2b99734741f4435d821bc1904cde69a

A **followup audit** for the major fixes to issues found in the first audit was performed from 2022-09-14 to 2022-09-28, with the following scope:

Math PR can be reviewed at
branch: audit-fixes,
frozen commit: b459299bb42455514782d741faf028bb971a6f756
Math PR:https://github.com/galacticcouncil/HydraDX-math/pull/61
Node PR can be reviewed at
branch: feat/omnipool-imbalance-update,
frozen commit: f80300c1fed514761a6601f68a2f3afe6282d6f1.
Node PR:https://github.com/galacticcouncil/HydraDX-node/pull/460
Spec PR can be reviewed at
branch: fix-imbalance-spec
commit: 35cf934999fede0c8780c6186f4bc7043cb82866
Simulations PR: N/A

The review is limited in scope to consider only provided code, though code of external libraries will be consulted whenever necessary. Non-chain and client-side portions of the codebase are *not* in the scope of this engagement.

Disclaimer **2**

Summary **3**

Formal Audit of HydraDX Omnipool Protocol **9**

Notable Findings: **46**

1. Vulnerabilities found when performing dependency linting. 46
2. Adding liquidity could be disabled using a relatively small amount of assets by manipulating the pool price of the stable coin 47
3. Rounding error could cause loss to the pool 48
4. Too much hub asset burned when removing liquidity 49
5. Hub asset can be withdrawn from the pool account via refund_refused_asset() call 50
6. Incorrect asset state update when the native currency is one of the assets being traded to Omnipool 51
7. Assets allowed to be equal when selling 52
8. Incorrect sell limit checking when buying asset_out using LRNA 53
9. Zero value for SimpleBalance does not have a consistent sign 54
10. Missing overflow checking for account balance (ORML tokens) 55
11. add_liquidity does not fully follow the math model when computing the imbalance 56

Informative findings: **57**

1. Incorrect imbalance update in pool initialization 57
2. Locked asset- unregistered asset cannot be refunded 58
3. Assumption not checked: LRNA imbalance is negative. 59
4. Updating HubAssetTradability could cause problems 60
5. Storage access in one place 61
6. Incorrect error type in the buy action when there is not enough reserve value for asset_in in the pool 62
7. Incorrect checking for sufficient balance 63
8. Duplicate computation of protocol_fee_amount when selling non-LRNA assets to the pool. 64
9. Majority of the liquidity will be owned by the pool if the withdrawal happens when the price drop significantly 65
10. Some condition checking in the specification is not implemented. 66

Follow-up Audit Report for Major Fixes **67**

Issues **98**

1. Redundant computation when calculating remove liquidity state change 98
2. Rounding error exploits when removing liquidity 99
3. Consistency: ImbalanceLRNA.negative is not a precondition in most exchange types. 100
4. Possibly large error in the imbalance when buying with LRNA 101

Assumptions

1. Assume the external library, in particular orml::MultiCurrency crate works correctly, i.e., account balances should always be updated as required.
   For example, arithmetic errors like overflow does not occur,
   $MultiCurrency::Account("Omnipool")(HubAssetId) \leq TotalIssuance(HubAssetId)$
   always true, etc.
2. $protocol_account()$, a.k.a., the balance account for $Omnipool$ will never be on the *dust removal* list or the account is in $DustRemovelWhiteList$. Thus, preconditions like
   $Balance_{asset} - \Delta_{asset} \geq ExistentialDeposit_{asset} \vee Omnipool \in DustRemovalWhitelist$
   when doing transfer or withdrawal is assumed to be always true.
3. $HubAsset$ is not in $Assets$, since it is not tradable for the current implementation.
4. $AccountId("Omnipool")$ can not be used to sign any transactions, it is not owned by any one
5. The implementation of NFT minting and other operations are not provided. We assume the positions minted as NFTs are linked by the correct position id and the initial owner of the position NFT is the corresponding liquidity provider. Furthermore, the NFT properly handles access authorization.

We do not assume full trustworthiness of governance and do **not** for example trust it with the ability to block certain users or take control of funds that the user did not intend to allow (roughly: theft). That is to say, we assume that governance would potentially sabotage the protocol if there was a clear monetary gain in doing so.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in the Disclaimer, we have used the following approaches to make our audit as thorough as possible.

First, we rigorously reasoned about the business logic of the protocol, i.e., built an abstract state machine from the code, proved key invariants upon each operation and performed rounding error analysis, validating security-critical properties such as price manipulation to ensure the absence of loopholes in the business logic and/or inconsistency between the logic, specification and the implementation.

Second, we carefully checked if the code is vulnerable to known security issues and attack vectors and the security checklist for CPMM.

Thirdly, we discussed the most catastrophic outcomes with the team and reasoned backwards from their places in the code to ensure that they are not reachable in any unintended way.

Finally, we regularly participated in meetings with the HydraDX team and offered our feedback during ongoing design discussions, and suggested development practices as well as design improvements.

This report describes the **intended** behaviour and invariants of the protocol under review and then outlines issues we have found, both in the intended behaviour and in the ways the code differs from it. We also point out lesser concerns, deviations from best practice and any other weaknesses we encounter.

Formal Audit of HydraDX Omnipool Protocol

Table of Contents

Protocol Design and Implementation
- Overall Specification
  - State Model- AMM
  - Invariants
- Operation: Pool Initializing:
  - Is the specification respected in the implementation?
  - Invariants Checking
  - Rounding Error Analysis
- Operation: Add Tokens:
  - Is the specification respected in the implementation?
  - Invariants Checking
  - Rounding Error Analysis
- Operation: Adding Liquidity- Single Asset Liquidity Provision
  - Is the specification respected in the implementation?
  - Invariants Checking
  - Rounding Error Analysis
- Operation: Removing Liquidity
  - Is the specification respected in the implementation?
  - Invariants Checking
  - Rounding Error Analysis
- Operation: Sell asset_in for asset_out, none is LRNA
  - Is the specification respected in the implementation?
  - Invariants Checking
  - Rounding Error Analysis
- Operation: Sell LRNA for asset_out
  - Is the specification respected in the implementation?
  - Invariants Checking
  - Rounding Error Analysis
- Operation: Buy asset_out using asset_in, none is LRNA
  - Is the specification respected in the implementation?
  - Invariants Checking
  - Rounding Error Analysis
- Operation: Buy asset_out using LRNA
  - Is the specification respected in the implementation?
  - Invariants Checking
  - Rounding Error Analysis
- Operation: Sacrifice position

- Operation: Set Asset Tradability
- Operation: Refund Refused Tokens
- Operation: Set Asset Weight Cap

Notable Findings:

- Vulnerabilities found when performing dependency linting.
- Adding liquidity could be disabled using a relatively small amount of assets by manipulating the pool price of the stable coin
- Rounding error could cause loss to the pool
- Too much hub asset burned when removing liquidity
- Hub asset can be withdrawn from the pool account via refund_refused_asset() call
- Incorrect asset state update when the native currency is one of the assets being traded to Omnipool
- Assets allowed to be equal when selling
- Incorrect sell limit checking when buying asset_out using LRNA
- Zero value for SimpleBalance does not have a consistent sign
- Missing overflow checking for account balance (ORML tokens)
- add_liquidity does not fully follow the math model when computing the imbalance

Informative findings:

- Incorrect imbalance update in pool initialization
- Locked asset- unregistered asset cannot be refunded
- Assumption not checked: LRNA imbalance is negative.
- Updating HubAssetTradability could cause problems
- Storage access in one place
- Incorrect error type in the buy action when there is not enough reserve value for asset_in in the pool
- Incorrect checking for sufficient balance
- Duplicate computation of protocol_fee_amount when selling non-LRNA assets to the pool.
- The majority of the liquidity will be owned by the pool if the withdrawal happens when the price drop significantly
- Some condition checking in the specification is not implemented.

Protocol Design and Implementation

Overall Specification

Design decisions

1. The position NFT does not bond to the liquidity provider (a.k.a., position NFT is transferable). It is LP's responsibility to keep the position of NFT safe.
2. A registered token will never be removed from the registry, but can be frozen to trade in the pool.
3. All assets in the pool should be registered with the asset registry.
4. The HubAssetImbalance to record the hub asset inflation.
    a. The LRNA imbalance is non-positive. It measures the amount of tokens that were added artificially in the pool, since they decrease the value of LRNA artificially, i.e. they cause some inflation.
    b. The LRNA imbalance is caused by directly swapping LRNA for one of the pool's tokens (say, T), since that adds LRNA tokens without adding a corresponding quantity of T, while at the same time removing T tokens without also removing a corresponding quantity of LRNA (i.e. the imbalance is roughly double the quantity of LRNA exchanged).
    $\Delta Imbalance = -\Delta LRNA + (\Delta T - fees) \cdot Price(T \rightarrow LRNA)$
    Note that $\Delta LRNA$ is positive, while ΔT is negative.
    c. The LRNA fees from normal exchanges are used to cover the imbalance.
    d. When adding liquidity, any imbalance causes more LRNA to be minted, so the imbalance becomes larger in absolute value. When removing liquidity, the opposite happens.

State Model- AMM

- $Balance_{token}$ represent the variables from implementation
    - $[token\ =\ HDX]\ NativeCurrency{::}\ AccountStore("Omnipool").\,free$
    - $[token\ \neq\ HDX]\ MultiCurrency{::}\ Accounts("Omnipool")(token).\,free$
- $Value_{token}\ =\ Balance^{token}_{LRNA}$, represents $Assets(token).\,hub_reserve$, meaning the amount of the hub asset (LRNA) in the $token/LRNA$ subpool.
- $Imbalance_{LRNA}$, represents $HubAssetImbalance{:}\ SimpleImbalance$

<u>Derived state definitions:</u>

- $Price_{token}\ =\ \frac{Value_{token}}{Balance_{token}}$

Invariants

1. **Swap_Invariant**: Constant product should be maintained in each subpool when fees are not considered,

$$Balance_{token} * Balance_{LRNA}^{token} = Balance_{token}' * Balance_{LRNA}^{token}'$$

2. **Non_Positive_Imbalance:** At the moment, imbalance can only be negative or 0. There should not be a way/scenario where it would become positive,
$$Imbalance_{LRNA}.negative = true$$
3. **HubAsset_Total:** The total issued hub asset (LRNA in the current case) should be equal to the LRNA asset held by the pool and all the LRNA asset that is given to liquidity providers,
$$TotalIssuance_{LRNA} = Accounts(Omnipool)(LRNA).total + \sum_{a \in Accounts}^{a \neq Omnipool} Accounts(a)(LRNA).total$$
4. **HubAsset_Accounting**: The total hub reserve of all assets (except hub asset) should be equal to the balance of hub asset (LRNA),
$$Balance_{LRNA} = \sum_{t \in Assets} Value_t$$
5. **Total_USD_Value_Accounting**: TotalTVL should always equal the sum of the tvl of each asset (except LRNA) in the pool,
$$TotalTVL = \sum_{t \in Assets} tvl_t$$
6. **Total_USD_Value_Capped**: The total approximate TVL in USD should always be bounded by the cap,
$$TotalTVL \leq TVLCap$$

Interesting properties

- If a token is overvalued, balancing the pool makes LRNA less valuable.
- If a token is undervalued, balancing the pool makes LRNA more valuable.
- $HubAsset \notin Assets$, $NativeAsset(Hdx) \in Assets$, $StableAsset \in Assets$
- Withdrawing all shares would probably leave some tokens in the pool (the only case when it would not be is when the withdrawal price is the same as the add liquidity price).
- If a token has some balance in the protocol account, then this token must have been registered in the asset registry, a.k.a. $\forall t,\ MultiCurrency{::}Accounts("Omnipool")(t) \rightarrow Asset\ Registry{::}Assets(t)$ is true.
- A set of identical withdrawals should retrieve the same amount of tokens (no swaps allowed in between, adding liquidity and withdrawing is allowed).
- Any sequence of identical liquidity adding actions should retrieve the same amount of shares (no swaps allowed in between, adding liquidity and withdrawing is allowed).
- Users can always withdraw their shares
- All entries in Assets are valid tokens.

Conventions in this document

1. The implementation section is faithful to the logic implemented in the code, including the sequence of variables computed and arrangements of variables in a computation.

2. In Rounding Analysis, we will use $\overline{Val}$ to represent the computation of the variable in the rational number domain, and $\delta_{Val} = Val - \overline{Val}$ to represent the difference between the actual value and rounded value.
3. In rounding error analysis, we focus on the key variables that change the AMM state or amounts that are leaving the pool.

Operation: Pool Initializing:

Specification:

1. Only registered assets can be added to Omnipool.
2. First two assets in the pool must be Stable Asset and Native Asset (HDX token).
3. Stable asset balance and native asset balance must be transferred to the Omnipool account manually before initializing the pool.
4. Initial liquidity of the new token being added to Omnipool must be transferred manually to the pool account prior to calling add_token.
5. Initial price of tokens is manually set by technical origin (1-hour delay) after the new token is approved by governance origin through DAP (1-day delay.)

Implementation:

Transition:

$initialize_pool(origin, stable_asset_price, native_asset_price, stable_weight_cap, native_weight_cap)$

Settings:

$Balance_{stable} = MultiCurrency::Accounts("Omnipool")(StableCoinAssetId).free$

$Balance_{HDX} = NativeCurrency::AccountStore("Omnipool").free$

$Price_{stable} = stable_asset_price$

$Price_{HDX} = native_asset_price$

$Value_{stable}' = \Delta Value_{stable} = \lfloor Price_{stable} * Balance_{stable} \rfloor$

$\Delta Value_{HDX} = \lfloor Price_{HDX} * Balance_{HDX} \rfloor$

$\Delta Balance_{LRNA} = \Delta Value_{stable} + \Delta Value_{HDX}$

$\Delta tvl_{HDX} = \lfloor \frac{\Delta Value_{HDX} * Balance_{stable}}{Value_{stable}'} \rfloor$

Pre conditions:

1. Origin is TechnicalOrigin
2. $StableCoinAssetId \notin Assets$
3. $HdxAssetId \notin Assets$
4. $Balance_{StableCoin} > 0$
5. $Balance_{HDX} > 0$
6. $Price_{stable} > 0$

7. $Price_{HDX} > 0$

Post state:

<u>Accounts:</u>

$Balance_{LRNA} \mathrel{+}= \Delta Balance_{LRNA}$

$TotalIssuance_{LRNA} \mathrel{+}= \Delta Balance_{LRNA}$

<u>Omnipool States:</u>

Assets(StableCoinAssetId)= {
hub_reserve: $\Delta Value_{stable}$,
shares: $Balance_{stable}$,
protocol_shares: $Balance_{stable}$,
tvl: $Balance_{stable}$,
cap: stable_weight_cap,
tradable: SELL | BUY | ADD_LIQUIDITY | REMOVE_LIQUIDITY,
}

Assets(NativeAssetId) = {
hub_reserve: $\Delta Value_{HDX}$,
shares: $Balance_{HDX}$,
protocol_shares: $Balance_{HDX}$,
tvl: Δtvl_{HDX}
cap: native_weight_cap,
tradable: SELL | BUY | ADD_LIQUIDITY | REMOVE_LIQUIDITY,
}

$TotalTVL' = TotalTVL + tvl_{HDX} + tvl_{stable}$

$tradable_{LRNA} = SELL$

Post condition:

$TotalTVL' <= Cap_{TVL}$

Conclusion:

Is the specification respected in the implementation?

1. Specification 1 is not guaranteed in code
2. Specification 2-4 is implemented
3. Specification 5, the initial price setting is subjective to TechnicalOrigin to act reasonably. It could be improved to be more transparent (auditable) by importing from the price feeders to get more accurate market value;

Invariants Checking

Invariants 2-6, i.e., Non_Positive_Imbalance, HubAsset_Total, HubAsset_Accouting, Total_USD_Value_Accouting, Total_USD_Value_Capped are guaranteed.
The proof is trivial.

Rounding Error Analysis

1. The amount of LRNA coin reserved for $stablecoin/LRNA$ subpool is at most 1 less, but will never be more than it should be, a.k.a., $\delta_{Value_{stable}} \in (-1, 0]$.
2. The amount of LRNA coin reserved for $HDX/LRNA$ subpool is at most 1 less, but will never be more than it should be, a.k.a., $\delta_{Value_{HDX}} \in (-1, 0]$.
3. Minted LRNA coin could be 2 less but never more than it should be, a.k.a., $\delta_{Balance_{LRNA}} \in (-2, 0]$.

Operation: Add Tokens:

Specification:

1. Token can only be added after the pool is initialized
2. Token can only be added after it is registered in asset registry
3. NextPositionId is not a key in Positions
4. The implementation should be consistent with the design specification: Add Token Spec.

Implementation:

Transition:

$add_token(\ origin,\ asset,\ initial_price,\ weight_cap,\ position_owner)$

Settings:

$Price_{asset} = initial_price$

$\Delta Value_{asset} = \lfloor Price_{asset} * Balance_{asset} \rfloor$

$\Delta tvl_{asset} = \lfloor \frac{\Delta Value_{asset} * Balance_{stable}}{Value_{stable}} \rfloor$

$$\Delta Imbalance = [Balance_{asset} \neq 0 \land Imbalance_{LRNA}.value \neq 0 \land Balance_{LRNA} \neq 0] * Decrease(\lfloor \frac{Value_{asset}' * Imbalance_{LRNA}.value}{Balance_{asset}'} * \frac{Balance_{asset}}{Balance_{LRNA}} \rfloor)$$

Pre conditions:

1. Origin is AddTokenOrigin
2. $asset \notin Assets$
3. $StableCoinAssetId \in Assets$
4. $asset \in AssetRegistry::Assets$
5. $Price_{asset} > 0$
6. $Balance_{asset} \geq MinimumPoolLiquidity$
7. $Balance_{asset} > 0$

Post state:

<u>Accounts:</u>

$Balance_{LRNA} \mathrel{+}= \Delta Value_{asset}$

$TotalIssuance_{LRNA} \mathrel{+}= \Delta Value_{asset}$

<u>Omnipool states:</u>

$TotalTVL' = TotalTVL + \Delta Value_{asset}$

$Imbalance_{LRNA} \mathrel{+}= \Delta Imbalance_{LRNA}$

Assets(asset)= {

hub_reserve: $\Delta Value_{asset}$,

shares: $Balance_{asset}$,

protocol_shares: 0,

tvl: Δtvl_{asset},
cap: weight_cap,
tradable: SELL | BUY | ADD_LIQUIDITY | REMOVE_LIQUIDITY,
}
Positions(NextPositionId) = {
asset_id: asset,
amount: $Balance_{asset}$,
shares: $Balance_{asset}$,
price: $Price_{asset}$,
}
$NextPositionId \ += \ 1$
$NFT_{Omnipool} \ += \ NextPositionId \mapsto \ position_owner$

Post condition:

$TotalTVL' \ \leq \ TVLCap$

Conclusion:

Is the specification respected in the implementation?

1. Specification 1 is true when $Assets.contains_key(StableCoinAssetId) \ \rightarrow \ initialize_pool() \ succeeded$ is true.
2. Specification 2-4 is assured

Invariants Checking

Invariants 2-6, i.e., Non_Positive_Imbalance, HubAsset_Total, HubAsset_Accouting, Total_USD_Value_Accouting, Total_USD_Value_Capped are guaranteed.
The proof is trivial.

Rounding Error Analysis

1. The minted amount of LRNA, which is the same as the amount of LRNA allocated to $asset/LRNA$ subpool, has a rounding difference $\delta_{Balance_{LRNA}} \ = \ \delta_{Value_{asset}} \ \in \ (-\ 1,\ 0]$.
2. The imbalance of the Hub Asset, LRNA has a rounding difference $\delta_{Imbalance_{LRNA}} \ \in \ [0, \frac{Imbalance_{LRNA}.value}{Balance_{LRNA}} \ + \ 1)$.

Operation: Adding Liquidity- Single Asset Liquidity Provision

Specification:

1. `add_liquidity` adds specified asset amount to the pool and in exchange gives the origin corresponding shares amount in the form of NFT at the current price.
2. Asset's tradable state must contain the ADD_LIQUIDITY flag, otherwise `NotAllowed` error is returned.
3. Asset weight cap must be respected, otherwise `AssetWeightExceeded` error is returned. Asset weight is the ratio between the new HubAsset reserve and the total reserve of HubAsset in Omnipool.
4. Adding liquidity should leave the pool price of the asset unchanged.
5. Adding liquidity should leave the token to share ratio (i.e., $\frac{Balance_{asset}}{shares_{asset}} = \frac{Balance_{asset}'}{shares_{asset}'}$) unchanged.
6. The amount of the added liquidity should be greater than 0.
7. The implementation of state changes should be consistent with the math specification in the design document, Add Liquidity Spec.
8. NextPositionId is not a key in Positions.

Implementation:

Transition:

$add_liquidity(origin,\ asset,\ amount)$

Settings:

$$who\ =\ ensure_signed(origin)$$
$$\Delta Balance_{asset}\ =\ amount$$
$$Price_{asset}\ =\frac{Value_{asset}}{Balance_{asset}}$$
$$Price_{stable}\ =\frac{Value_{stable}}{Balance_{stable}}$$
$$\Delta Value_{asset}\ = \lfloor Price_{asset}\ *\ \Delta Balance_{asset} \rfloor$$
$$\Delta shares_{asset}\ = \lfloor \frac{\Delta Balance_{asset}}{Balance_{asset}} * shares_{asset} \rfloor$$
$$weight_{asset}'\ =\ \frac{Value_{asset}'}{Balance_{LRNA}'} = \frac{Value_{asset}+\Delta Value_{asset}}{Balance_{LRNA}+\Delta Value_{asset}}$$
$$Price_{asset}'\ =\frac{\Delta Value_{asset} + Value_{asset}}{amount + Balance_{asset}}$$
$$\Delta Imbalance\ =\ [amount\ \neq\ 0\ \&\&\ Imbalance_{LRNA}.value\ \neq\ 0\ \&\&\ Balance_{LRNA}\ \neq\ 0]\ *$$
$$Decrease(\lfloor \lfloor \frac{(\Delta Value_{asset} + Value_{asset})*\ Imbalance_{LRNA}.value}{amount + Balance_{asset}} \rfloor\ *\ \frac{amount}{Balance_{LRNA}} \rfloor)$$

Pre conditions:

1. $amount \geq MinimumPoolLiquidity$
2. $amount = 0 \lor Balance_{asset}^{who} - \Delta Balance_{asset} \geq MultiCurrency::Accounts(who)(asset).frozen$
3. $asset \in Assets$
4. $StableCoinAssetId \in Assets$
5. $ADD_LIQUIDITY \in tradable_{asset}$

Post state:

<u>Accounts:</u>

$Balance_{asset} \mathrel{+}= \Delta Balance_{asset}$

$Balance_{asset}^{who} \mathrel{-}= \Delta Balance_{asset}$

$Balance_{LRNA} \mathrel{+}= \Delta Value_{asset}$

$TotalIssuance_{LRNA} \mathrel{+}= \Delta Value_{asset}$

<u>Omnipool states:</u>

$TotalTVL' = TotalTVL + tvl_{asset}' - tvl_{asset}$

$Imbalance_{LRNA} \mathrel{+}= \Delta imbalance$

Assets(asset)= {

hub_reserve: $Value_{asset} + \Delta Value_{asset}$,

shares: $shares_{asset} + \Delta shares_{asset}$,

tvl: $\lfloor \frac{Balance_{stable} * (Value_{asset} + \Delta Value_{asset})}{Value_{stable}} \rfloor$,

...

}

$NFT_Omnipool \mathrel{+}= NextPositionId \mapsto who$

$Positions \mathrel{+}= NextInstanceId \mapsto \{$

asset,

amount,

$\Delta shares_{asset}$,

$Price_{asset}'$

$\}$

$NextPositionId \mathrel{+}= 1$

Post condition:

$weight_{asset} \leq cap_{asset}$

$TotalTVL' \leq TVLCap$

Conclusion:

Is the specification respected in the implementation?

1. Specifications 1-3,7 are guaranteed.
2. Specifications 4 and 5 are guaranteed up to rounding.
3. Specification 6 is depending on $MinimumPoolLiquidity > 0$, which is not guaranteed.

4. Specification 8 is guaranteed since the position id is monotonically increasing.

Invariants Checking

Invariants 2-6, i.e., Non_Positive_Imbalance, HubAsset_Total, HubAsset_Accouting, Total_USD_Value_Accouting, Total_USD_Value_Capped are guaranteed.

Proof is trivial.

Rounding Error Analysis

1. The minted amount of LRNA, which is the same as the amount of LRNA allocated to $asset/LRNA$ subpool, has a rounding difference $\delta_{Balance_{LRNA}} = \delta_{Value_{asset}} \in (-1, 0]$.
2. The imbalance of the Hub Asset, LRNA has a rounding difference $\delta_{Imbalance_{LRNA}} \in [0, \frac{\Delta Balance_{asset}}{Balance_{asset} + \Delta Balance_{asset}} * \frac{Imbalance_{LRNA}.value}{Balance_{LRNA}} + \frac{\Delta Balance_{asset}}{Balance_{LRNA}} + 1)$.
   Note that: if $Price_{asset}$ is used to replace $\frac{\Delta Value_{asset} + Value_{asset}}{Balance_{asset} + \Delta Balance_{asset}}$, the rounding error could be $\delta_{Imbalance_{LRNA}} \in [0, \frac{\Delta Balance_{asset}}{Balance_{LRNA}} + 1)$ which would be lower than the current implementation, and the computation could be simplified.
3. The total shares of the asset in the pool and the shares given to LP in $Positions(NextPositionId)$ have the same rounding difference, $\delta_{shares_{asset}} = \delta_{shares_{asset}^{NextPositionId}} \in (-1, 0]$.

Operation: Removing Liquidity

Specification:

1. `remove_liquidity` takes shares in the form of an NFT from the origin, and returns the asset corresponding to the amount of shares. If the current price is larger than the price at investment time, the pool will also pay some amount of LRNA to the liquidity provider.
2. Asset's tradable state must contain the REMOVE_LIQUIDITY flag, otherwise a `NotAllowed` error is returned.
3. Removing liquidity should not change the pool price of the asset.
4. Removing liquidity should not change the token to share ratio, i.e., $\frac{Balance_{asset}}{shares_{asset}} = \frac{Balance_{asset}'}{shares_{asset}'}$.
5. The amount of the shares requested to remove should be greater than 0.
6. If there are no remaining shares, burn the old Position NFT. Otherwise, update the position.
7. The implementation of state changes should be consistent with the math specification in the design document, Withdraw Liquidity Spec.
8. NextPositionId is not a key in Positions
9. Impermanent loss
   a. Defined as $\frac{ValueInvestAndWithdraw}{ValueHold} - 1$
   b. Equal to $\frac{2 * \sqrt{InvestPrice(T \to LRNA) \cdot WithdrawPrice(T \to LRNA)}}{InvestPrice(T \to LRNA) + WithdrawPrice(T \to LRNA)}$
   c. It's the geometric mean over the arithmetic mean of T's price at investment time and withdrawal time, which means that it must be between 0 (exclusive, assuming reasonable prices) and 1 (inclusive).

Implementation:

Transition:

remove_liquidity(origin, position_id, amount)

Settings:

$who = ensure_signed(origin)$

$asset_id = Positions(position_id).asset_id$

$Price_{position_id} = Position(position_id).price$

$Price_{asset_id} = \frac{Value_{asset_id}}{Balance_{asset_id}}$ is the current price of the asset over the stable coin asset.

$\alpha = \frac{Price_{position_id} - Price_{asset_id}}{Price_{position_id} + Price_{asset_id}}, \beta = 1 - \alpha = \frac{2 * Price_{asset_id}}{Price_{position_id} + Price_{asset_id}}$

$\Delta b = [Price_{asset_id} < Price_{position_id}] * \lfloor \alpha * amount \rfloor$

$\Delta shares_{asset_id} = amount - \Delta b$

$$Unit^{token/share}_{asset_id} = \frac{Balance_{asset_id}}{shares_{asset_id}}$$

$$Unit^{token/share}_{position_id} = \frac{Balance_{position_id}}{shares_{posotion_id}}$$

$$\Delta Balance_{asset_id} = \lfloor \frac{Balance_{asset_id} * \Delta shares_{asset_id}}{shares_{asset_id}} \rfloor$$

$$\Delta Value_{asset_id} = \lfloor \frac{\Delta Balance_{asset_id} * Value_{asset_id}}{Balance_{asset_id}} \rfloor$$

$$\Delta imbalance = [\Delta Balance_{asset_id} \neq 0 \wedge Imbalance_{LRNA}.value \neq 0 \wedge Balance_{LRNA} \neq 0] * Increase(\lfloor \lfloor \frac{(Value_{asset_id} - \Delta Value_{asset_id}) * Imbalance_{LRNA}.value}{Balance_{asset_id} - \Delta Balance_{asset_id}} \rfloor * \frac{\Delta Balance_{asset_id}}{Balance_{LRNA}} \rfloor)$$

$$\Delta Balance^{who}_{LRNA} = [Price_{asset_id} > Price_{position_id}] * \lfloor Price_{asset_id} * (\lfloor \beta * \Delta Balance_{asset_id} \rfloor - \Delta Balance_{asset_id}) \rfloor$$

Pre conditions:

1. $NFT_{Omnipool}(position_id) = who$
2. $position_id \in Positions$
3. $Position(position_id).shares \geq amount$
4. $StableCoinAssetId \in Assets$
5. $asset_id \in Assets$
6. $REMOVE_LIQUIDITY \in tradable_{asset}$
7. $Balance_{asset_id} - \Delta Balance_{asset_id} \geq MultiCurrency::Accounts(Omnipool)(asset).frozen$
8. $Balance_{LRNA} - \Delta Value_{asset_id} \geq MultiCurrency::Accounts(Omnipool, LRNA).frozen$

Post state:

Accounts:

$Balance_{asset_id} -= \Delta Balance_{asset_id}$,

$Balance_{LRNA} -= \Delta Value_{asset_id} + \Delta Balance^{who}_{LRNA}$

$TotalIssuance_{LRNA} -= \Delta Value_{asset_id}$

$Balance^{who}_{asset_id} += \Delta Balance_{asset_id}$,

$Balance^{who}_{LRNA} += \Delta Balance^{who}_{LRNA}$

Omnipool Assets:

$Imbalance_{LRNA} += \Delta Imbalance$

$TotalTVL' = TotalTVL + tvl_{asset_id}' - tvl_{asset_id}$

$Assets(asset_id) = \{$

$hub_reserve: Value_{asset_id} - \Delta Value_{asset_id}$

$shares: shares_{asset_id} - \Delta shares_{asset_id}$

$protocol_shares: shares_{protocol} + \Delta b$

$tvl: [is_stable_asset] * (Balance_{StableCoin} - \Delta Balance_{asset_id})$

$+ [!\, is_stable_asset] * \lfloor \frac{Balance_{stable} * (Value_{asset_id} - \Delta Value_{asset_id})}{Value_{stable}} \rfloor$

...

}

$Positions'$ and $NFT_{Omnipool}'$:

$[shares_{position_id} - amount = 0]$ $Positions.remove(position_id)$ && $NFT_{Omnipool}.burn(position_id)$

$[shares_{position_id} - amount \neq 0]$ $Positions(position_id) =$

$\{ asset_id: _,$

$amount: Balance_{position_id} - \lfloor amount * Unit^{token/share}_{position_id} \rfloor,$

$shares: shares_{position_id} - amount,$

$price: _,$

$\}$

Post condition:

$$TotalTVL' \leq TVLCap$$

Conclusion:

Is the specification respected in the implementation?

1. Specifications 1,2, 6, 7, 8 are guaranteed in the implementation.
2. Specifications 3 and 4 are guaranteed up to rounding since

$$Price_{asset_id}' = \frac{Value_{asset_id}'}{Balance_{asset_id}'} = \frac{Value_{asset_id} - \frac{\Delta Balance_{asset_id} * Value_{asset_id}}{Balance_{asset_id}}}{Balance_{asset_id} - \Delta Balance_{asset_id}}$$

$$= \frac{Balance_{asset_id} - \Delta Balance_{asset_id}}{Balance_{asset_id} - \Delta Balance_{asset_id}} * \frac{Value_{asset_id}}{Balance_{asset_id}} = Price_{asset_id}$$

$$Unit^{token/share}_{asset_id}{}' = \frac{Balance_{asset_id}'}{shares_{asset_id}'} = \frac{Balance_{asset_id} - \Delta Balance_{asset_id}}{shares_{asset_id} - \Delta shares_{asset_id}}$$

Since the pool always owns some share ($shares^{protocol}_{asset_id} > 0$) and protocol share never decreases, $shares_{asset_id}' > 0$ is guaranteed. Furthermore, we have

$$Unit^{token/share}_{asset_id}{}' = \frac{Balance_{asset_id} - \frac{Balance_{asset_id} * \Delta shares_{asset_id}}{shares_{asset_id}}}{shares_{asset_id} - \Delta shares_{asset_id}} = \frac{Balance_{asset_id}}{shares_{asset_id}} = Unit^{token/share}_{asset_id}.$$

3. Specification 5 is not explicitly checked in the code.

Invariants Checking

1. Invariants 3,5, i.e., HubAsset_Total, Total_USD_Value_Accouting, are guaranteed. The proof is trivial.
2. Invariant 6, Total_USD_Value_Capped is guaranteed by post condition checking. However, withdrawing liquidity does not necessarily bring down $TotalTVL$. In the case when there is a significant drop in stable coin's pool price (where a large amount of

stable coin is sold to the pool), it could be possible that $tvl_{asset_id}' > tvl_{asset_id}$ when $Price_{stable} < \frac{Value_{asset_id} - \Delta Value_{asset_id}}{tvl_{asset_id}}$.

3. Invariant 4, HubAsset_Accounting is violated.

   $Balance_{LRNA} \mathrel{-=} \Delta Value_{asset_id} + \Delta Balance^{who}_{LRNA}$
   $TotalIssuance_{LRNA} \mathrel{-=} \Delta Value_{asset_id}$
   For the hub asset, the amount $\Delta Value_{asset_id}$ is burned and the amount $\Delta Balance^{who}_{LRNA}$ left the pool.
   Furthermore,
   $Balance_{LRNA}' = Balance_{LRNA} - \Delta Value_{asset_id} - \Delta Balance^{who}_{LRNA}$

   $$\sum_{t \in Assets} Value_t' = \sum_{t \in Assets}^{t \neq asset_id} Value_t' + Value_{asset_id} - \Delta Value_{asset_id} = \sum_{t \in Assets} Value_t' - \Delta Value_{asset_id}$$

   Thus, $Balance_{LRNA}' \neq \sum_{t \in Assets} Value_t'$, a violation of the invariant 4 is proved.

4. Invariant 2, Non_Positive_Imbalance is satisfied after this operation.

   Assume Non_Positive_Imbalance is true before calling $remove\ liquidity$ function, where $Imbalance_{LRNA}.negative = true$.

   According to the implementation,

   $Imbalance_{LRNA}' =$

   $$[{}^{*}\ \Delta imbalance > Imbalance_{LRNA}.value] * ({}^{*}\ \Delta imbalance - Imbalance_{LRNA}.value,\ false) +$$

   $$[{}^{*}\ \Delta imbalance \leq Imbalance_{LRNA}.value] * (\ Imbalance_{LRNA}.value -^{*}\ \Delta imbalance,\ true)$$

   Assume $^{*}\ \Delta imbalance > Imbalance_{LRNA}.value$ is true, we have

   $$\frac{(Value_{asset_id} - \Delta Value_{asset_id}) * Imbalance_{LRNA}.value}{Balance_{asset_id} - \Delta Balance_{asset_id}} * \frac{\Delta Balance_{asset_id}}{Balance_{LRNA}} > Imbalance_{LRNA}.value \Rightarrow$$

   $$Price_{asst_id}' * \frac{\Delta Balance_{asset_id}}{Balance_{LRNA}} > 1$$

   Since $Price_{asst_id}' * \Delta Balance_{asset_id} = Price_{asst_id} * \Delta Balance_{asset_id} = \Delta Value_{asset_id}$,

   then $\Delta Value_{asset_id} > Balance_{LRNA}$ which would reach a contradiction with the pre-condition #8.

   Thus, $Imbalance_{LRNA}.negative = true$ is maintained after the execution of remove liquidity.

Rounding Error Analysis

1. $delta_protocol_shares$: $Increase(\Delta b)$, indicates the amount of shares contributed to the pool when withdrawing at a lower price.

We have $\overline{\Delta b} - 1 < \Delta b \leq \overline{\Delta b}$ when the current price drops below the position price, there could be one share less due to rounding error and no error otherwise, i.e., $\delta_{shares^{protocol}_{asset_id}} \in (-1, 0]$.

2. $delta_shares$: $Decrease(shares_removed - \Delta b)$ is the amount of shares leaving the pool. When the current asset price drops below the position price, the computing might result in at most 1 more share than it should be. Otherwise, there will be no error, i.e., $\delta_{\Delta shares_{asset_id}} \in [0, 1)$.
3. $delta_reserve$: $Decrease(\Delta Balance_{asset_id})$ indicates the amount of the asset paid out to the eligible liquidity provider from the pool. We compute the rounding difference as follows:

$$Unit^{token/share}_{asset_id} * \Delta shares - 1 < \Delta Balance_{asset_id} \leq Unit^{token/share}_{asset_id} * \Delta shares$$

   a. When the current asset price drops, the rounding difference is $\delta_{\Delta Balance_{asset_id}} \in (-1, Unit^{token/share}_{asset_id})$.
   There could be more tokens paid out from the pool than it should be.
   b. When the price is up, the rounding error would be at most 1 less, i.e., $\delta_{\Delta Balance_{asset_id}} \in (-1, 0]$ which is favourable to the pool.

4. The amount of the hub asset reserved for the asset is reduced by $\Delta Value_{asset_id}$, where the reduced amount is the amount of LRNA being burned.

$$\Delta Balance_{asset_id} * Price_{asset_id} - 1 < \Delta Value_{asset_id} < \Delta Balance_{asset_id} * Price_{asset_id}$$

The rounding effect is calculated as
   1) When the price drops, $\delta_{\Delta Value_{asset_id}} \in (-Price_{asset_id} - 1, Unit^{token/share}_{asset_id} * Price_{asset_id})$.
   2) When price rises or remains the same, $\delta_{\Delta Value_{asset_id}} \in (-Price_{asset_id} - 1, 0]$.

In conclusion, the rounding difference of the amount of the hub asset reserved for the asset is:
   1) when price drops, $\delta_{Value_{asset_id}} \in (-Unit^{token/share}_{asset_id} * Price_{asset_id}, Price_{asset_id} + 1)$
   2) When the price goes up, $\delta_{Value_{asset_id}} \in [0, Price_{asset_id} + 1)$.

5. The change to the current imbalance is to be reduced by an amount * $\Delta imbalance$. We have
   1) when the pool price of the asset drops, we have

$$\delta_{\Delta imbalance} \in \left(-\frac{Price_{asset_id} \cdot Imbalance_{LRNA}.value}{Balance_{LRNA}} - \frac{\Delta Balance_{asset_id}}{Balance_{LRNA}} - 1, \frac{Price_{asset_id} \cdot Imbalance_{LRNA}.value}{Balance_{LRNA}} Unit^{token/share}_{asset_id} + \frac{Imbalance_{LRNA}.value}{Balance_{asset_id} - \Delta Balance_{asset_id}} \cdot \frac{\Delta Balance_{asset_id}}{Balance_{LRNA}}\right)$$

$$\delta_{Imbalance_{LRNA}} \in \left(-\frac{Price_{asset_id} \cdot Imbalance_{LRNA}.value}{Balance_{LRNA}} Unit^{token/share}_{asset_id} - \frac{Imbalance_{LRNA}.value}{Balance_{asset_id} - \Delta Balance_{asset_id}} \cdot \frac{\Delta Balance_{asset_id}}{Balance_{LRNA}},\right.$$

$$\frac{Price_{asset_id} \cdot Imbalance_{LRNA}.value}{Balance_{LRNA}} + \frac{\Delta Balance_{asset_id}}{Balance_{LRNA}} + 1)$$

2) When the pool price of the asset increases, we have:

$$\delta_{\Delta imbalance} \in (-\frac{Price_{asset_id} \cdot Imbalance_{LRNA}.value}{Balance_{LRNA}} - \frac{\Delta Balance_{asset_id}}{Balance_{LRNA}} - 1, \frac{Imbalance_{LRNA}.value}{Balance_{asset_id} - \Delta Balance_{asset_id}} \cdot \frac{\Delta Balance_{asset_id}}{Balance_{LRNA}})$$

$$\delta_{Imbalance_{LRNA}} = -\delta_{\Delta imbalance}$$

Thus the rounding difference does not always favor the pool.

6. In the case that when the current pool price of the asset increases than the invested-time pool price, some amount of hub asset will be withdrawn from the pool and deposited to the position owner. The amount is indicated by the variable $lp_hub_amount = \lfloor Price_{asset_id} * (\lfloor \beta * \Delta Balance_{asset_id} \rfloor - \Delta Balance_{asset_id}) \rfloor$.

   When price rises, $\delta_{\Delta Balance_{asset_id}} \in (-1, 0]$, the rounding difference for lp_hub_amount is $\delta_{lp_hub_amount} \in (-\beta * Price_{asset_id} - 1, 0]$ where $\beta > 1$.

   While when price drops or remains the same, there is no rounding error.

   Thus, the amount of LRNA paid out is in favour of the pool.

   However, if the difference between the current price and the position price, the position owner would receive LRNA much less than he/she should have.

7. The change to the amount of the asset, $asset_id$, on the position, in the case that the liquidity provider did a partial withdrawal, is indicated by $delta_position_reserve$: $Decrease(\lfloor shares_removed * Unit^{token/share}_{position_id} \rfloor)$. Thus, the rounding difference for the changed amount on the position is $\delta_{Balance^{NextPositionId}_{asset_id}} \in [0, -1)$.

Operation: Sell $asset_in$ for $asset_out$, none is LRNA

Specification:

1. $SELL \in tradable_{asset_in}$ and $BUY \in tradable_{asset_out}$, otherwise a `NotAllowed` error is returned.
2. Fees
   a. Protocol fee
      Taken from the LRNA amount
      Used first to cover the LRNA imbalance, then added to the native asset subpool (LRNA-HDX). $fee_{protocol} = ProtocolFee * \Delta Value_{asset_in}$
   b. Asset fee
      Taken from the $asset_out$ amount $fee_{asset} = AssetFee * \Delta Value_{asset_out}$
3. The swap will maintain the swap invariant in the subpools, $asset_in/LRNA$ and $asset_out/LRNA$, if fees are not considered.
4. The implementation should be consistent with the design specification: Swap Spec.
5. The hub asset will only be possible to be burned but never minted in swap events.

Implementation:

Transition:

sell(origin, asset_in, asset_out, amount, min_buy_amount)

Settings:

$$who = ensure_signed(origin)$$
$$\Delta Balance_{asset_in} = amount$$
$$\Delta Value_{asset_in} = \lfloor \frac{amount * Value_{asset_in}}{Balance_{asset_in} + amount} \rfloor$$
$$\Delta Value_{asset_out} = \Delta Value_{asset_in} - \lfloor ProtocolFee * \Delta Value_{asset_in} \rfloor$$
$$\Delta Balance_{asset_out_0} = \frac{Balance_{asset_out} * \Delta Value_{asset_out}}{Value_{asset_out} + \Delta Value_{asset_out}}$$
$$\Delta Balance_{asset_out} = \lfloor \Delta Balance_{asset_out_0} \rfloor - \lfloor AssetFee * \lfloor \Delta Balance_{asset_out_0} \rfloor \rfloor$$
$$Fee_{protocol} = \lfloor ProtocolFee * \Delta Value_{asset_in} \rfloor$$
$$\Delta Imbalance = min(Fee_{protocol}, Imbalance_{LRNA}.value)$$
$$\Delta Value_{HDX} = max(0, Fee_{protocol} - \Delta Imbalance)$$
$$\Delta Balance_{LRNA} = \Delta Value_{asset_out} + \Delta Value_{HDX} - \Delta Value_{asset_in}$$

Pre conditions:

1. $amount \geq Limit_{minimum_trading}$
2. $Balance^{who}_{asset_in} - amount \geq MultiCurrency: Accounts(who)(asset_in).frozen$
3. $asset_in \neq LRNA \land asset_out \neq LRNA$

4. $SELL \in tradable_{asset_in} \wedge BUY \in tradable_{asset_out}$
5. $Balance^{who}_{asset_in} - amount \geq MultiCurrency::Accounts(who)(asset_in).frozen$
6. $Balance_{asset_out} - \Delta Balance_{asset_out} \geq MultiCurrency::Accounts(Omnipool)(asset_out).frozen$
7. $MultiCurrency::Accounts(who)(asset_out).total() + \Delta Balance_{asset_out} \geq$
   $ExistentialDeposit_{asset_out} \,||\, who \in DustRemovalWhitelist$
8. $Balance_{LRNA} - (\Delta Value_{asset_in} - (\Delta Value_{asset_out} + \Delta Value_{HDX})) \geq$
   $MultiCurrency::Accounts(Omnipool)(LRNA).frozen$ if
   $\Delta Value_{asset_out} + \Delta Value_{HDX} \geq \Delta Value_{asset_in}$

Missing condition checking:

9. $asset_in \neq asset_out$

Post state:

Accounts:

$Balance^{who}_{asset_in} -= amount$

$Balance_{asset_in} += amount$

$Balance_{asset_out} -= \Delta Balance_{asset_out}$

$Balance^{who}_{asset_out} += \Delta Balance_{asset_out}$

$Balance_{LRNA} += \Delta Balance_{LRNA}$

$TotalIssuance_{LRNA} += \Delta Balance_{LRNA}$

Omnipool:

$Imbalance_{LRNA} -= \Delta Imbalance$

$Assets(HdxAssetId).hub_reserve += \Delta Value_{HDX}$

$Assets(asset_in).hub_reserve -= \Delta Value_{asset_in}$

$Assets(asset_out).hub_reserve += \Delta Value_{asset_out}$

Post condition:

$\Delta Balance_{asset_out} \geq min_buy_amount$

Conclusion:

Is the specification respected in the implementation?

1. specification s 1,2,4 is satisfied.
2. Specification 3 is guaranteed, proof same as checking invariant 1, Swap_Invariant.
3. Specification 5 is satisfied since

$$\begin{aligned}\Delta Balance_{LRNA} &= \Delta Value_{asset_out} + \Delta Value_{HDX} - \Delta Value_{asset_in} \\ &= \Delta Value_{asset_in} - Fee_{protocol} + \Delta Value_{HDX} - \Delta Value_{asset_in} \\ &= max(0, Fee_{protocol} - \Delta Imbalance) - Fee_{protocol}\end{aligned}$$

$< 0.$

Invariants Checking

1. Invariant 1, the Swap_Invariant is guaranteed up to rounding, for subpools, $asset_in/LRNA$ and $asset_out/LRNA$, if fees are not considered. Proof is sketched as follows.

Proof: let $Fee_{protocol} = \lfloor ProtocolFee * \Delta Value_{asset_in} \rfloor$,

$Fee_{Asset} = \lfloor AssetFee * \lfloor \frac{Balance_{asset_out} * \Delta Value_{asset_out}}{Value_{asset_out} + \Delta Value_{asset_out}} \rfloor$,

$\Delta Balance_{asset_in} = amount$

$\Delta Value_{asset_in} = \lfloor \frac{amount * Value_{asset_in}}{Balance_{asset_in} + amount} \rfloor$

$\Delta Value_{asset_out} = \Delta Value_{asset_in} - Fee_{protocol}$

$\Delta Balance_{asset_out} = \lfloor \frac{Balance_{asset_out} * \Delta Value_{asset_out}}{Value_{asset_out} + \Delta Value_{asset_out}} \rfloor - Fee_{Asset}$

1) For $asset_in/LRNA$ subpool, we have

$Balance_{asset_in}' = Balance_{asset_in} + \Delta Balance_{asset_in} = Balance_{asset_in} + amount$

$Balance^{asset_in}_{LRNA}{}' = Balance^{asset_in}_{LRNA} - \Delta Value_{asset_in} = Value_{asset_in} - \lfloor \frac{amount * Value_{asset_in}}{Balance_{asset_in} + amount} \rfloor$

Such that

$Balance_{asset_in}' * Balance^{asset_in}_{LRNA}{}' = (Balance_{asset_in} + amount) * (Value_{asset_in} - \lfloor \frac{amount * Value_{asset_in}}{Balance_{asset_in} + amount} \rfloor)$

If rounding is ignored,

$Balance_{asset_in}' * Balance^{asset_in}_{LRNA}{}' = Balance_{asset_in} * Value_{asset_in} = Balance_{asset_in} * Balance^{asset_in}_{LRNA}$

If rounding is considered, the constant product will increase $[0, Balance_{asset_in} + amount)$.

2) For $asset_out/LRNA$, we have

$\Delta Value_{asset_out_no_fee} = \Delta Value_{asset_in}$, $\Delta Balance_{asset_out_no_fee} = \lfloor \frac{Balance_{asset_out} * \Delta Value_{asset_out_no_fee}}{Value_{asset_out} + \Delta Value_{asset_out_no_fee}} \rfloor$

$Balance_{asset_out}' = Balance_{asset_out} - \Delta Balance_{asset_out_no_fee}$

$= Balance_{asset_out} - \lfloor \frac{Balance_{asset_out} * \Delta Value_{asset_out_no_fee}}{Value_{asset_out} + \Delta Value_{asset_out_no_fee}} \rfloor$

$= Balance_{asset_out} - \lfloor \frac{Balance_{asset_out} * \Delta Value_{asset_in}}{Value_{asset_out} + \Delta Value_{asset_in}} \rfloor$

$Balance^{asset_out}_{LRNA}{}' = Balance^{asset_out}_{LRNA} + \Delta Value_{asset_out_no_fee} = Value_{asset_out} + \Delta Value_{asset_in}$

$Balance_{asset_out}' * Balance^{asset_out}_{LRNA}{}'$

$= (Balance_{asset_out} - \lfloor \frac{Balance_{asset_out} * \Delta Value_{asset_in}}{Value_{asset_out} + \Delta Value_{asset_in}} \rfloor) * (Value_{asset_out} + \Delta Value_{asset_in})$

If rounding is ignored,

$Balance_{asset_out}' * Balance^{asset_out}_{LRNA}{}' = Balance_{asset_out} * Value_{asset_out}$.

If rounding is considered, the constant product will increase $[0, Value_{asset_out} + \Delta Value_{asset_in})$.

3) According to the design spec and the implementation, the protocol fee collected from the trade will be used to cover the impermanent loss. However, if there is any amount left, it will send to the $HDX/LRNA$ subpool, thus, this trade will also increase the constant product in the $HDX/LRNA$ subpool.

In conclusion, after the swap,

1) The Swap_Invariant for the subpools $asset_in/LRNA$ and $asset_out/LRNA$ is guaranteed if any fees and rounding are ignored.
2) If considering rounding but no fees, the constant products will increase $[0, Balance_{asset_in} + amount)$ for the $asset_in/LRNA$ subpool and $[0, Value_{asset_out} + \Delta Value_{asset_in})$ for the $asset_out/LRNA$ subpool.
3) If considered fees but no rounding, the constant product for the $asset_in/LRNA$ subpool is maintained, but the constant product for the $asset_out/LRNA$ subpool will increase $\frac{AssetFee * Balance_{asset_out} * (1 - ProtocolFee) * Value_{asset_in} * amount}{Value_{asset_in} + amount}$.
4) The constant product for the $HDX/LRNA$ will very likely increase if the protocol fee is large enough.

2. Invariants 2, 3,4,5,6 i.e., Non_Positive_Imbalance, HubAsset_Total, HubAsset_Accouting, Total_USD_Value_Accouting, Total_USD_Value_Capped are guaranteed. The proof is trivial.

Rounding Error Analysis

1. The hub asset in the $asset_in/LRNA$ subpool is indicated by
   $Value_{asset_in}' = Value_{asset_in} - \Delta Value_{asset_in}$, where $\Delta Value_{asset_in} = \lfloor \frac{amount * Value_{asset_in}}{Balance_{asset_in} + amount} \rfloor$.
   Thus, $\delta_{\Delta Value_{asset_in}} \in (-1, 0]$, and $\delta_{Value_{asset_in}} \in [0, 1)$.
2. The hub asset in the $asset_out/LRNA$ subpool is indicated by
   $Value_{asset_out}' = Value_{asset_out} - \Delta Value_{asset_out}$ where
   $\Delta Value_{asset_out} = \Delta Value_{asset_in} - \lfloor ProtocolFee * \Delta Value_{asset_in} \rfloor$. We have
   $\delta_{\Delta Value_{asset_out}} \in (ProtocolFee - 1, 1)$ and $\delta_{Value_{asset_out}} \in (-1, 1 - ProtocolFee)$.

3. The amount of $asset_out$ paid to the trader is indicated by
   $\Delta Balance_{asset_out} = \lfloor \Delta Balance_{asset_out}_0 \rfloor - \lfloor AssetFee * \lfloor \Delta Balance_{asset_out}_0 \rfloor \rfloor$, where
   $\Delta Balance_{asset_out}_0 = \frac{Balance_{asset_out} * \Delta Value_{asset_out}}{Value_{asset_out} + \Delta Value_{asset_out}}$.
   We can conclude $\Delta Balance_{asset_out}_0$ should be a bit lower than it should be before rounding. Consequently, $\Delta Balance_{asset_out}$ is less than it should be before rounding which is favourable to the pool.
4. The imbalance of the hub asset is indicated by
   $Imbalance_{LRNA}' = Imbalance_{LRNA} - \Delta Imbalance$ where

$Fee_{protocol} = \lfloor ProtocolFee * \Delta Value_{asset_in} \rfloor$,

$\Delta Imbalance = min(Fee_{protocol}, Imbalance_{LRNA}.value)$

Thus $\delta_{\Delta Imbalance} \in (-ProtocolFee - 1, 0]$ and $\delta_{Imbalance} \in [0, 1)$.

5. The amount of LRNA to be burned is

   $\Delta Balance_{LRNA} = \Delta Value_{asset_out} + \Delta Value_{HDX} - \Delta Value_{asset_in}$ where

   $\Delta Value_{HDX} = max(0, Fee_{protocol} - \Delta Imbalance)$.

   $\delta_{\Delta Value_{HDX}} \in (-1, 0]$, thus $\delta_{\Delta Balance_{LRNA}} \in (ProtocolFee - 1, 1)$.

Operation: Sell LRNA for $asset_out$

Specification:

1. $SELL \in tradable_{LRNA}$ and $BUY \in tradable_{asset_out}$, otherwise a `NotAllowed` error is returned.
2. The amount of $asset_out$ should be no less than min_buy_amount, providing price protection for traders.
3. The hub asset will only be possible to be burned but never minted in swap events.
4. The implementation should be consistent with the design specification: Swap LRNA Spec.

Implementation:

Transition:

$$sell(origin,\ asset_in,\ asset_out,\ amount,\ min_buy_amount)$$

Settings:

$$\Delta Balance_{asset_out} = \Delta Balance_{asset_out}_1 - \lfloor AssetFee * \Delta Balance_{asset_out}_1 \rfloor$$

$$\Delta Balance_{asset_out}_1 = \lfloor \frac{Balance_{asset_out} * amount}{Value_{asset_out} + amount} \rfloor$$

$$hub_imbalance = \lfloor \frac{amount * Value_{asset_out}}{Value_{asset_out} + amount} \rfloor$$

$$\Delta imbalance = hub_imbalance - \lfloor AssetFee * hub_imbalance \rfloor + amount$$

Pre conditions:

1. $amount \geq Limit_{minimum_trading}$
2. $Balance^{who}_{asset_in} - amount \geq MultiCurrency::Accounts(who)(asset_in).frozen$
3. $asset_in = LRNA$
4. $SELL \in tradable_{LRNA} \wedge BUY \in tradable_{asset_out}$
5. $aset_out \in Assets$
6. $Balance^{who}_{LRNA} - amount \geq MultiCurrency::Accounts(who)(LRNA).frozen$
7. $Balance_{asset_out} - \Delta Balance_{asset_out} \geq MultiCurrency::Accounts(Omnipool)(asset_out).frozen$
8. $MultiCurrency::Accounts(who)(asset_out).total() + \Delta Balance_{asset_out} >= ExistentialDeposit_{asset_out} \ || \ who \in DustRemovalWhitelist$

Missing condition checkings:

1. $asset_out \neq LRNA$

Post state:

<u>Accounts:</u>

$Balance^{who}_{LRNA} \ -= amount$

$Balance_{LRNA} \ += \ amount$

$Balance_{asset_out} \ -= \ \Delta Balance_{asset_out}$

$Balance^{who}_{asset_out} \ += \ \Delta Balance_{asset_out}$

Omnipool states:

$Imbalance_{LRNA} \ -= \ \Delta imbalance$

$Assets(asset_out).hub_reserv \ += \ amount$

Post condition:

1. $\Delta Balance_{asset_out} \ \geq \ min_buy_amount$

Post condition:

Is the specification respected in the implementation?

Specifications 1-4 are satisfied. There is no LRNA burned or minted, thus specification 3 also holds.

Invariants Checking

1. Invariant 1, the Swap_Invariant is guaranteed, ignoring rounding and swap fees, for the subpool $asset_out/LRNA$.

   Proof:

   $Balance_{asset_out}' \ * \ Value_{asset_out}'$

   $= \ (Balance_{asset_out} \ - \ \frac{Balance_{asset_out} * amount}{Value_{asset_out} + amount}) \ * \ (Value_{asset_out} \ + \ amount)$

   $= \ Balance_{asset_out} \ * \ Value_{asset_out}$

2. Invariants 2, 3,4,5,6 i.e., Non_Positive_Imbalance, HubAsset_Total, HubAsset_Accouting, Total_USD_Value_Accouting, Total_USD_Value_Capped are guaranteed. The proof is trivial.

Rounding Error Analysis

1. The amount of $asset_out$ paid to the trader is indicated by $\Delta Balance_{asset_out} \ = \ \Delta Balance_{asset_out}{}_1 \ - \lfloor AssetFee \ * \ \Delta Balance_{asset_out}{}_1 \rfloor$ where

   $\Delta Balance_{asset_out}{}_1 \ = \ \lfloor \frac{Balance_{asset_out} * amount}{Value_{asset_out} + amount} \rfloor.$

   Thus $\delta_{\Delta Balance_{asset_out}{}_1} \in (-\ 1, 0],$ and $\delta_{\Delta Balance_{asset_out}} \ \in (-\ (1 \ - \ AssetFee\ ),\ 1).$

2. The imbalance of the hub asset is indicated by $Imbalance_{LRNA} \; -= \; \Delta imbalance$ where $\Delta imbalance \; = \; hub_imbalance \; - \; \lfloor AssetFee \; * \; hub_imbalance \rfloor \; + \; amount$ and $hub_imbalance \; = \; \lfloor \frac{amount * Value_{asset_out}}{Value_{asset_out} + amount} \rfloor$.
   We have $\delta_{hub_imbalance} \in (-\; 1,\; 0]$, $\delta_{\Delta imbalance} \; \in (-\; (1 \; - \; AssetFee\;),\; 1)$.
   Thus $\delta_{imbalance} \; = \; -\; \delta_{\Delta imbalance} \; \in (-\; 1,\; 1 \; - \; AssetFee)$.

Operation: Buy $asset_out$ using $asset_in$, none is LRNA

Specification:

1. Given *asset_out* requested amount y, *asset_in* and *asset_out* quantity in Omnipool with amounts, T_1 and T_2, LRNA token quantities in each subpool, L_1 and L_2, calculate the amount of *asset_in* needs to be provided and swap x *asset_in* for y *asset_out* (i.e., take x from the user and give back y).
2. The hub asset is traded separately. The hub asset can only be sold to Omnipool and cannot be bought from Omnipool at the moment, i.e., LRNA can not be the *asset_out* for either buy or sell.
3. Asset's tradable states must contain SELL flag for asset_in and BUY flag for asset_out, otherwise `NotAllowed` error is returned.
4. The implementation should be consistent with the design specification: Swap Spec.
5. The hub asset will only be possible to be burned but never minted.

Implementation:

Transition:

buy(origin, asset_out, asset_in, amount, max_sell_amount)

Settings:

$$who = ensure_signed(origin)$$
$$\Delta Balance_{asset_out} = amount$$
$$Balance_{asset_out}_wo_fee = Balance_{asset_out} - \lfloor AssetFee * Balance_{asset_out} \rfloor$$
$$\Delta Value_{asset_out} = \lfloor \frac{Value_{asset_out} * amount}{Balance_{asset_out}_wo_fee - amount} \rfloor + 1$$
$$\Delta Value_{asset_in} = \lfloor \frac{\Delta Value_{asset_out}}{1 - ProtocolFee} \rfloor$$
$$\Delta Balance_{asset_in} = \lfloor \frac{Balance_{asset_in} * \Delta Value_{asset_in}}{Value_{asset_in} - \Delta Value_{asset_in}} \rfloor + 1$$
$$Fee_{protocol} = \lfloor ProtocolFee * \Delta Value_{asset_in} \rfloor$$
$$\Delta imbalance = min(Fee_{protocol}, Imbalance_{LRNA}.value)$$
$$\Delta Value_{HDX} = Fee_{protocol} - \Delta imbalance$$
$$\Delta Balance_{LRNA} = \Delta Value_{asset_out} + \Delta Value_{HDX} - \Delta Value_{asset_in}$$

Pre conditions:

1. $amount \geq Limit_{minimum_trading}$
2. $asset_in \neq LRNA \wedge asset_out \neq LRNA$
3. $SELL \in tradable_{asset_in} \wedge BUY \in tradable_{asset_out}$
4. $Balance_{asset_out} \geq amount$
5. $Balance^{who}_{asset_in} - \Delta Balance_{asset_in} \geq MultiCurrency::Accounts(who)(asset_in).frozen$

6. $MultiCurrency::Accounts(who)(asset_out).total + amount \geq ExistentialDeposit_{asset_out} \vee$
   $who \in DustRemovalWhiteList$
7. $Balance_{asset_out} - amount \geq MultiCurrency::Accounts(Omnipool)(asset_out).frozen$
8. $Balance_{LRNA} - \left|\Delta Balance_{LRNA}\right| \geq MultiCurrency::Accounts(Omnipool)(LRNA).frozen$ when $\Delta Balance_{LRNA} < 0$
9. $Value_{asset_in} \geq \Delta Value_{asset_in}$

Missing condition checking:

10. $asset_in \neq asset_out$

Post state:

<u>Accounts:</u>

$$Balance^{who}_{asset_in} -= \Delta Balance_{asset_in}$$
$$Balance_{asset_in} += \Delta Balance_{asset_in}$$
$$Balance_{asset_out} -= amount$$
$$Balance^{who}_{asset_out} += amount$$
$$Balance_{LRNA} += \Delta Balance_{LRNA}$$
$$TotalIssuance_{LRNA} += \Delta Balance_{LRNA}$$

<u>Omnipool states:</u>

$Imbalance_{LRNA} -= \Delta imbalance$

Assets[HdxAssetId].hub_reserve += $[\Delta Value_{HDX} > 0] * \Delta Value_{HDX}$

Assets(asset_in).hub_reserve -= $\Delta Value_{asset_in}$

Assets(asset_out).hub_reserve += $\Delta Value_{asset_out}$

Post condition:

1. $\Delta Balance_{asset_in} \leq max_sell_amout$

Conclusion:

Is the specification respected in the implementation?

1. specification s 1-4 are satisfied.
2. Specification 5 is satisfied since

$$\Delta Balance_{LRNA} = \Delta Value_{asset_out} + \Delta Value_{HDX} - \Delta Value_{asset_in}$$
$$= \Delta Value_{asset_out} + Fee_{protocol} - \Delta imbalance - \left\lfloor \frac{\Delta Value_{asset_out}}{1 - ProtocolFee} \right\rfloor$$

We'll ignore rounding to simplify the calculation that

$$\Delta Balance_{LRNA} = \Delta Value_{asset_out} + Fee_{protocol} - \Delta imbalance - \frac{\Delta Value_{asset_out}}{1 - ProtocolFee}$$
$$= Fee_{protocol} - \Delta imbalance - \frac{ProtocolFee * \Delta Value_{asset_out}}{1 - ProtocolFee}$$

Since $\Delta imbalance = min(Fee_{protocol}, Imbalance_{LRNA}.value)$,

1) When $Fee_{protocol} < Imbalance_{LRNA}.value$,

   $\Delta Balance_{LRNA} =- \frac{ProtocolFee * \Delta Value_{asset_out}}{1-ProtocolFee} < 0$.

2) Otherwise,

   $\Delta Balance_{LRNA} = Fee_{protocol} - Imbalance_{LRNA}.value - \frac{ProtocolFee * \Delta Value_{asset_out}}{1-ProtocolFee} < 0.$

Invariants Checking

1. Invariant 1, the Swap_Invariant is guaranteed, ignoring rounding and fees, for subpools, $asset_in/LRNA$ and $asset_out/LRNA$, if fees are not considered. The proof is sketched as follows.

   a. For the $asset_in/LRNA$ subpool, ignoring rounding and fees, we have

   $Balance_{asset_in}' * Value_{asset_in}' = (Balance_{asset_in} + \Delta Balance_{asset_in}) * (Value_{asset_in} - \Delta Value_{asset_in})$

   $= (Balance_{asset_in} + \frac{Balance_{asset_in} * \Delta Value_{asset_in}}{Value_{asset_in} - \Delta Value_{asset_in}}) * (Value_{asset_in} - \Delta Value_{asset_in})$

   $= Balance_{asset_in} * Value_{asset_in}$

   Such that the Swap_Invariant holds.

   b. For the $asset_out/LRNA$ subpool, ignoring rounding, we have

   $Balance_{asset_out}' * Value_{asset_out}' = (Balance_{asset_out} - \Delta Balance_{asset_out}) * (Value_{asset_out} + \Delta Value_{asset_out})$

   $= (Balance_{asset_out} - amount) * (Value_{asset_out} + \frac{Value_{asset_out} * amount}{Balance_{asset_out} - amount})$

   $= Balance_{asset_out} * Value_{asset_out}$

   Such that the Swap_Invariant holds.

2. Invariants 2, 3, 5, 6 i.e.,Non_Positive_Imbalance, HubAsset_Total, Total_USD_Value_Accouting, Total_USD_Value_Capped are guaranteed. The proof is trivial.

3. Invariants 4, HubAsset_Accouting is satisfied. The proof is sketched as follows.

   Proof: When $Fee_{protocol} < Imbalance_{LRNA}.value$,

   $\Delta imbalance = Fee_{protocol}$, $\Delta Value_{HDX} = 0$, thus

   $Balance_{LRNA}' = Balance_{LRNA} + \Delta Balance_{LRNA} = Balance_{LRNA} + \Delta Value_{asset_out} - \Delta Value_{asset_in}$

   $$= \sum_{t \in Assets} Value_t + \Delta Value_{asset_out} + \Delta Value_{HDX} - \Delta Value_{asset_in}$$

   $$= \sum_{t \in Assets} Value_t'$$

   When $Fee_{protocol} \geq Imbalance_{LRNA}.value$,

   $\Delta imbalance = Imbalance_{LRNA}$, $\Delta Value_{HDX} = Fee_{protocol} - Imbalance_{LRNA}$, the invariant also holds.

Rounding Error Analysis

1. The hub asset in the $asset_out/LRNA$ subpool is indicated by
   $Value_{asset_out} \mathrel{+}= \Delta Value_{asset_out}$, where $\Delta Value_{asset_out} = \lfloor \frac{Value_{asset_out} * amount}{Balance_{asset_out}_wo_fee - amount} \rfloor + 1$
   $Balance_{asset_out}_wo_fee = Balance_{asset_out} - \lfloor AssetFee * Balance_{asset_out} \rfloor$.
   We have $\delta_{Balance_{asset_out}_wo_fee} \in [0, 1)$,
   $\delta_{\Delta Value_{asset_out}} \in (- \frac{Value_{asset_out} * amount}{((1-AssetFee)*Balance_{asset_out}+1 - amount)*((1-AssetFee)*Balance_{asset_out} - amount)}, 1]$
   and $\delta_{Value_{asset_out}} = \delta_{\Delta Value_{asset_out}}$.

2. The hub asset in the $asset_in/LRNA$ subpool is indicated by $Value_{asset_in} \mathrel{-}= \Delta Value_{asset_in}$
   ,where $\Delta Value_{asset_in} = \lfloor \frac{\Delta Value_{asset_out}}{1-ProtocolFee} \rfloor$.
   $\delta_{\Delta Value_{asset_in}} \in (\delta^{lower_bound}_{\Delta Value_{asset_out}} - 1, \frac{\delta^{upper_bound}_{\Delta Value_{asset_out}}}{1-ProtocolFee}]$ where $\delta^{upper_bound}_{\Delta Value_{asset_out}} = 1$.
   Thus $\delta_{Value_{asset_in}} = - \delta_{\Delta Value_{asset_in}}$.
   Note that the rounded hub asset in the $asset_in/LRNA$ subpool after the buy operation could be much more than it should be when the $amount$ is large.
3. The amount of $asset_in$ needed to be paid by the trader is indicated by
   $\Delta Balance_{asset_in} = \lfloor \frac{Balance_{asset_in} * \Delta Value_{asset_in}}{Value_{asset_in} - \Delta Value_{asset_in}} \rfloor + 1$. Since $\Delta Value_{asset_in}$ could be much smaller than it should be, $\Delta Balance_{asset_in}$ could be even smaller which would not be favourable to the pool.
4. The imbalance of the hub asset is indicated by $Imbalance_{LRNA} \mathrel{-}= \Delta Imbalance$, where
   $Fee_{protocol} = \lfloor ProtocolFee * \Delta Value_{asset_in} \rfloor$,
   $\Delta imbalance = min(Fee_{protocol}, Imbalance_{LRNA}.value)$.
   $\delta_{Fee_{protocol}} \in (ProtocolFee * \delta^{lower_bound}_{\Delta Value_{asset_in}} - 1, ProtocolFee * \delta^{upper_bound}_{\Delta Value_{asset_in}}]$
   Thus, $\delta_{Imbalance_{LRNA}}, = -\delta_{Fee_{protocol}}$ when $Fee_{protocol} < Imbalance_{LRNA}.value$, otherwise
   $\delta_{Imbalance_{LRNA}}, = 0$.
5. The amount of LRNA to be burned is $\Delta Balance_{LRNA} =$
   $\Delta Value_{asset_out} + \Delta Value_{HDX} - \Delta Value_{asset_in}$ where $\Delta Value_{HDX} = Fee_{protocol} - \Delta imbalance$
   a. When $Fee_{protocol} < Imbalance_{LRNA}.value$,
      $\delta_{\Delta Balance_{LRNA}} = \delta_{\Delta Value_{asset_out}} - \delta_{\Delta Value_{asset_in}}$.

   b. When $Fee_{protocol} \geq Imbalance_{LRNA}.value$,
      $\delta_{\Delta Balance_{LRNA}} = \delta_{\Delta Value_{asset_out}} + \delta_{Fee_{protocol}} - \delta_{\Delta Value_{asset_in}}$.

Operation: Buy $asset_out$ using LRNA

Specification:

1. $SELL \in tradable_{LRNA}$ and $BUY \in tradable_{asset_out}$, otherwise a `NotAllowed` error is returned.
2. The hub asset is traded separately. The hub asset can only be sold to Omnipool and cannot be bought from Omnipool at the moment, i.e., LRNA can not be the *asset_out* for either buy or sell.
3. Asset's tradable states must contain SELL flag for LRNA and BUY flag for asset_out, otherwise `NotAllowed` error is returned.
4. The implementation should be consistent with the design specification: [Swap LRNA Spec]().
5. The hub asset will only be possible to be burned but never minted.

Implementation:

Transition:

buy(origin, asset_out, asset_in, amount, max_sell_amount)

Settings:

$$\Delta Balance_{asset_out} = amount$$

$$hub_denominator = Balance_{asset_out} - \lfloor AssetFee * Balance_{asset_out} \rfloor - amount$$

$$\Delta Value_{asset_out} = \lfloor \frac{Value_{asset_out} * amount}{hub_denominator} \rfloor + 1$$

$$\Delta Balance_{LRNA} = \Delta Value_{asset_out}$$

$$hub_imbalance = \lfloor \frac{\Delta Value_{asset_out} * Value_{asset_out}}{Value_{asset_out} + \Delta Value_{asset_out}} \rfloor$$

$$\Delta imbalance = hub_imbalance - \lfloor AssetFee * hub_imbalance \rfloor + \Delta Value_{asset_out}$$

Pre conditions:

1. $amount \geq Limit_{minimum_trading}$
2. $asset_in = LRNA \wedge asset_out \neq LRNA$
3. $SELL \in tradable_{LRNA} \wedge BUY \in tradable_{asset_out}$
4. $MultiCurrency::Accounts(who)(asset_out).total + amount \geq ExistentialDeposit_{asset_out} \vee$ $who \in DustRemovalWhitelist$
5. $Balance_{asset_out} - amount \geq MultiCurrency::Accounts(Omnipool)(asset_out).frozen$

Missing condition checking

6. $HubAsset \neq asset_out$

Post state:

<u>Accounts:</u>

$Balance^{who}_{LRNA}$ $-=$ $\Delta Value_{asset_out}$

$Balance_{LRNA}$ $+=$ $\Delta Value_{asset_out}$

$Balance_{asset_out}$ $-=$ $amount$

$Balance^{who}_{asset_out}$ $+=$ $amount$

Omnipool states:

$Imbalance_{LRNA}$ $-=$ $\Delta Imbalance$

Assets($asset_out$).hub_reserve += $\Delta Value_{asset_out}$

Post condition:

1. $\Delta Balance_{LRNA} = \Delta Value_{asset_out} \leq max_sell_amount$

Conclusion:

Is the specification respected in the implementation?

Specifications 1-5 are satisfied. There is no LRNA burned or minted, thus specification 5 also holds.

Invariants Checking

1. Invariant 1, the Swap_Invariant is guaranteed ignoring rounding and swap fees, for the subpool $asset_out/LRNA$.

   Proof:

   $Balance_{asset_out}' * Value_{asset_out}'$

   $= (Balance_{asset_out} - amount) * (Value_{asset_out} + \frac{Value_{asset_out} * amount}{Balance_{asset_out} - amount})$

   $= Balance_{asset_out} * Value_{asset_out}$

2. Invariants 2, 3, 4, 5, 6 i.e., Non_Positive_Imbalance, HubAsset_Total, HubAsset_Accouting, Total_USD_Value_Accouting, Total_USD_Value_Capped are guaranteed. The proof is trivial.

Rounding Error Analysis

1. The amount of hub asset reserved in the $asset_out/LRNA$ subpool after this operation is indicated by Assets($asset_out$).hub_reserve += $\Delta Value_{asset_out}$, where

   $\Delta Value_{asset_out} = \lfloor \Delta Value_{asset_out}_0 \rfloor$, $\Delta Value_{asset_out}_0 = \frac{Value_{asset_out} * amount}{hub_denominator} + 1$ and

   $hub_denominator = Balance_{asset_out} - \lfloor AssetFee * Balance_{asset_out} \rfloor - amount.$

We have $\delta_{hub_denominator} \in [0,\ 1)$, $\delta_{\Delta Value_{asset_out}0} \in (-\frac{Value_{asset_out} * amount}{(hub_denominator + 1)*hub_denominator} + 1,\ 1]$ and $\delta_{\Delta Value_{asset_out}} \in (-\frac{Value_{asset_out} * amount}{(hub_denominator + 1)*hub_denominator},\ 1]$.

Since the LRNA balance will increase by $\Delta Value_{asset_out}$, it could result in a new balance of LRNA significantly less than it should be when the requested *amount* of *asset_out* to swap out is big enough.

2. The imbalance of the hub asset is indicated by $\Delta imbalance = hub_imbalance - \lfloor AssetFee * hub_imbalance \rfloor + \Delta Value_{asset_out}$, where $hub_imbalance = \lfloor \frac{\Delta Value_{asset_out} * Value_{asset_out}}{Value_{asset_out} + \Delta Value_{asset_out}} \rfloor$.

   Calculate the exact rounding difference for *hub_imbalance* might be less meaningful here. However, it is observed that any rounding difference of $\Delta Value_{asset_out}$ would be significantly boosted in the same direction. We consider this rounding error significant.

Operation: Sacrifice position

Implementation:

Transition:

sacrifice_position(origin, position_id)

Settings:

$who \ = ensure_signed(origin)$
$asset \ = Positions(position_id).\, assets_id$

Pre conditions:

1. $position_id \ \in Positions$
2. $NFT_{Omnipool}(position_id) \ = \ who$
3. $asset \ \in \ Assets$

Post state:

$shares_{asset}^{protocol} \ += shares_{asset}^{position_id}$
$Positions.\, remove(position_id)$
$NFT_{Omnipool}.\, burn(position_id)$

Conclusion:

AMM state is not changed, such that invariants are preserved.

Operation: Set Asset Tradability

Implementation:

set_asset_tradable_state(origin, asset_id, state)

Pre conditions:

1. origin is TechnicalOrigin
2. $asset_id \in Assets$

Post state:

$[asset_id = HubAssetId]\ tradable_{LRNA} = state$

$[asset_id \neq HubAssetId]\ tradable_{asset_id} = state$

Conclusion:

AMM state is not changed, such that invariants are preserved.

Operation: Refund Refused Tokens

Specification:

1. Token which has not been accepted to the asset pool, should be refunded to the recipient.

Implementation:

refund_refused_asset(origin, asset_id, amount, recipient)

Pre conditions:

1. Origin is AddTokenOrigin
2. $asset_id \notin Assets$
3. $Balance_{asset_id} - amout \geq MultiCurrency::Accounts("Omnipool")(asset_id).frozen$
4. $MultiCurrency::Accounts(who)(asset_id).total + amount \geq ExistentialDeposit_{asset_id} || who \in DustRemovalWhitelist$

Missing condition:

$asset_id \in AssetRegistry::Assets$ – Assume only when a token is registered in the asset registry when its `ExistentialDeposit` can be provided.

$asset_id \neq HubAssetID$ - the hub asset LRNA should not leave the pool.

Post state:

$Balance_{asset_id} -= amount$

$Balance_{asset_id}^{recipient} += amount$

Post condition:-

Conclusion:

1. Refund operation might not be successful, refer to the implicit condition above.
2. AMM state is not changed, such that invariants are preserved.

Operation: Set Asset Weight Cap

Implementation:

set_asset_tradable_state(origin, asset_id, state)

Pre conditions:

1. origin is TechnicalOrigin
2. $asset_id \in Assets$

Post state:

$$Assets(asset_id).cap = cap$$

Conclusion:

AMM state is not changed, such that invariants are preserved.

Notable Findings:

1. Vulnerabilities found when performing dependency linting.

[Severity: High | Difficulty: Medium | Category: Security]

Issue:
Method: cargo audit
Adversary database scanned: RustSec Adversary Database
Referred to issue #50, issue #431

Effects: Use after free affects memory safety, that it could lead to memory leakage.

Recommendation:
HydraDX math repo: The serde_cbor crate is unmaintained. The author has archived the github repository. Alternatives proposed by the author:

- ciborium
- minicbor

HydraDX node/warehouse: 2 vulnerabilities and 1 deprecated crate. Though they seem to be related to the substrate itself, it is recommended to upgrade to the patched versions.

Status: Submitted issue #50, issue #431, under investigation.

2. Adding liquidity could be disabled using a relatively small amount of assets by manipulating the pool price of the stable coin

[Severity: High | Difficulty: Medium | Category: Security]

Issue: When adding liquidity, the tvl of an asset is vulnerable to stable coin price manipulation, since the preferred stable coin is a tradable asset in the pool. The tvl_{asset} is calculated by the current reserve hub asset for the $asset/LRNA$ subpool divides the spot price of the stable coin in Omnipool.

Adding liquidity when a significant price drop (either by observing the state or price manipulation) of the stable coin happened could let the LP add more tvl of the asset than it should be, resulting in the `TotalTVL` reaching `TVLCap` earlier than it should be.

Thus, it can be utilized by the attacker to disable the adding liquidity operation using a relatively small amount of the chosen asset.

Effects: Adding liquidity operation could be disabled by an attacker with low cost.

Recommendation: Use a more stable price to perform tvl calculation.

Status: Discussed with the HydraDX team and confirmed. An issue is submitted in #453, and fixed in PR#460.

3. Rounding error could cause loss to the pool

[Severity: High | Difficulty: Easy | Category: Functional Correctness]

Issue: The AMM state machine in Omnipool operates in the integer domain, where rounding is needed at each state change. In some cases, the rounding difference is significant and could result in a loss to the pool. A semi-complete list of rounding errors that needs to be alerted is listed as follows:

A. In this report, Section Operation: Remove Liquidity - Conclusion-Rounding error analysis- delta_reserve`. The rounding difference of $\Delta Balance$ could cause more assets to be paid out from the pool than it should be, resulting in a loss to the pool.
B. In this report, Section Operation: Remove Liquidity - Conclusion-Rounding error analysis- imbalance`, when the current price drops below the invested price, the rounding difference for the updated imbalance could be significant. It is recommended to redesign the rounding strategy such that the rounding effect is within a small and controllable range.
C. In this report, Section Operation:Operation: Buy asset_out using asset_in, none is LRNA - Conclusion-Rounding Analysis #1, the rounding effect on $\Delta Value_{asset_out}$ could be very large when the amount requested to buy is large enough. Similarly, buying an asset using LRNA, would also have a rounding error that could affect the AMM state unfavourably.

Effects: Could cause loss to the pool.

Recommendation: The rounding strategy needs to be carefully designed, and solutions to be discussed further.

Status: Issue submitted in #455, fixed in PR#61 (further audited in the *follow up audit*.).

4. Too much hub asset burned when removing liquidity

[Severity: High | Difficulty: Easy | Category: Functional Correctness]

Issue: Referring to section Removing Liquidity-Post state - Accounts:

$Balance_{LRNA} \; -= \; \Delta Value_{asset_id} + \Delta Balance^{who}_{LRNA}$

$TotalIssuance_{LRNA} \; -= \; \Delta Value_{asset_id}$

The amount $\Delta Value_{asset_id}$ is burned which results a violation of Invariant 4: LIQUIDITY_EQUVALENCE:

$$Balance_{LRNA} = \sum_{t \in Assets} Value_t$$

That after executing remove liquidity function, we have

$Balance_{LRNA}' = Balance_{LRNA} - \Delta Value_{asset_id} - \Delta Balance^{who}_{LRNA}$

$\sum_{t \in Assets} Value_t' = \sum_{t \in Assets}^{t \neq asset_id} Value_t' + Value_{asset_id} - \Delta Value_{asset_id}$

$Balance_{LRNA}' \neq \sum_{t \in Assets} Value_t'$

Effects: Describe the consequences if not fixed.

Recommendation:

$Balance_{LRNA} \; -= \; \Delta Value_{asset_id}$

$TotalIssuance_{LRNA} \; -= \; \Delta Value_{asset_id} - \Delta Balance^{who}_{LRNA}$

Status: Issue Submitted, #448, fixed in PR #445.

5. Hub asset can be withdrawn from the pool account via refund_refused_asset() call

[Severity: High | Difficulty: High | Category: Security]

Issue: Refer to section Refund Refused Asset - Implementation - Precondition, any asset satisfy the preconditions, especially $asset_id \notin Assets$ could leave the pool and send to anyone. The hub asset, LRNA also satisfy the condition. Thus, the LRNA asset could be withdrawn at any time to anyone, as long as the transaction owner has the AddTokenOrigin.

Effects: This could cause significant loss if the AddTokenOrigin is maliciously used.

Recommendation: Add a precondition to guarantee the hub asset can not be withdrawn.

Status: Issue Submitted, #447, fixed in PR #451.

6. Incorrect asset state update when the native currency is one of the assets being traded to Omnipool

[Severity: High| Difficulty: Low | Category: Functional Correctness]

Issue: According to the implementation, when the protocol fee is greater than the imbalance change, the native currency, HDX subpool's hub reserve is being added the additional amount of $protocol_fee - \Delta imbalance$. However, it is overwritten by $set_asset_state()$ function called for updating HDX's asset state afterwards.

Effects: The final hub reserve for HDX is less than it should be, causing the actual price of HDX to be less than it should be. It is considered a loss for Omnipool.

Recommendation: There are two options for correcting this issue:

- Call the function $update_hdx_subpool_hub_asset$ after set_asset_state.
- Compute the variable hub_asset_amount as a $BalanceUpdate$ type and merge it with the state changes for the asset HDX when it is one of the assets being traded.

Status: Issue Submitted, refer to issue #438, fixed in PR #445.

7. Assets allowed to be equal when selling

[Severity: High | Difficulty: Low | Category: Functional Correctness]

Issue: The sell endpoint does not check that the in and out assets are distinct. At the same time, it loads the asset data for both assets near the start of the function, and it will update the state for both at the end of the function. This means that: 1. The asset state will be first set to the "in asset state", then it will be overwritten with the "out asset state". 2. The out asset state will just register an exchange of some LRNA for some asset, which is not what happened.

Effects: The asset state no longer corresponds to reality. Besides being bad in itself, this allows users to increase asset prices unlimitedly in exchange for paying some fees.

Recommendation: Check that the assets are distinct.

Status: Issue Submitted, #436; Fixed in PR #437.

8. Incorrect sell limit checking when buying $asset_out$ using LRNA

[Severity: Medium | Difficulty: Low | Category: Functional Correctness]

Issue: In the function $buy_asset_for_hub_asset()$, line 1440 checked a post condition to ensure the amount of the LRNA coin sold to the pool does not exceed the max_sell_limit. It should check the LRNA coin needed (i.e., $*\ state_changes.asset.delta_hub_reserve$) to buy $asset_out$ rather than checking the amount of $asset_out$ that is requested.

Effects: Describe the consequences if not fixed.

Recommendation: Recommended way to fix it. If there is no straightforward solution, leave it for discussion with the team.

Status: Submitted in issue #439, fixed in PR #449.

9. Zero value for SimpleBalance does not have a consistent sign

[Severity: Medium | Difficulty: Low | Category: Functional Correctness]

Issue: SimpleImbalance::default and SimpleBalance::add set the sign of 0 to 'negative'. However, SimpleImbalance::sub sets it to 'positive'. This should not matter right now since all balances are negative, so SimpleImbalance::sub should not produce zero values. See Issue #432

Effects: If positive balances will be used, equality testing may not work properly. Also, checking whether a number is positive or negative will be more involved than it should be.

Recommendation: Make SimpleBalance::sub set the sign of 0 as 'negative'.

Status: Issue Submitted in #432. Fix in PR 443.

10. Missing overflow checking for account balance (ORML tokens)

[Severity: High | Difficulty: High | Category: Security]

Issue: It is suspected that overflow checking when minting LRNA token is not sufficient, in the do_deposit function of open-runtime-module-library-d3634ee624a28945/63b3219/tokens/src/lib.rs.

Effects: Potential overflow on hub asset balance.

Recommendation: Always use safe math when performing arithmetic operations.

Status: Issue submitted as issue #433, the team confirmed this could be an issue but out of the team's control. They are considering using another pallet.

11. add_liquidity does not fully follow the math model when computing the imbalance

[Severity: Low | Difficulty: Medium | Category: Functional Correctness]

Issue: calculate_delta_imbalance uses asset_state.reserve and asset_state.hub_reserve for its computation. add_liquidity calls recalculate_imbalance which calls calculate_delta_imbalance with new_asset_state as an argument. This means that calculate_delta_imbalance receives the updated values for asset_state.reserve and asset_state.hub_reserve (i.e., using the math notations, it receives and uses Qi+ and Ri+). However, in the math model, the imbalance delta uses the non-updated reserves (Qi and Ri).

Effects: If the rounding error is ignored, Qi/Ri should be equal to Qi+/Ri+ by design. However, the rounding effect has to be considered in the code, the implementation is more accurate compared to the design.

Recommendation: No further action is required in the code. However, it would be good to mention the difference in the design document.

Status: Issue filed in #435, the issue is solved along with other PRs.

Informative findings:

1. Incorrect imbalance update in pool initialization

[Severity: Low | Difficulty: Low | Category: Code Improvement]

Issue: The result from a recalculate_imbalance call seems to be mostly ignored (except for error checking), and the subsequent call to update_imbalance uses an incorrect imbalance delta.

Effect: Confusing code. Recalculate_imbalance does nothing useful and returns 0, while the incorrect imbalance delta argument is also 0.

Recommendation: Three options:

1) remove all imbalance code,
2) remove only the update_imbalance code and check the result from recalculate_imbalance,
3) fix the update_imbalance argument.

Status: Submitted in issue #429, solved in PR#430.

2. Locked asset- unregistered asset cannot be refunded

[Severity: Informative | Difficulty: Easy | Category: Security]

Issue: The default value of an asset's ExistentialDeposit is u128::MAX if the asset is not registered in the asset registry. Thus, the condition checking greater than or equal to the Existential Deposit would always fail. Consequently, the attempt to transfer an asset that is not registered in the asset registry would always fail if the recipient is also not in the RemovalWhiteList.

Note that if there is an assumption that only registered assets can be transferred to the pool account, then this should not be an issue (However, implementation of transfer asset to pool account is not provided with this audit).

Effects: Describe the consequences if not fixed.

Recommendation: Recommended way to fix it. If there is no straightforward solution, leave it for discussion with the team.

Status: The rating of the issue is downgraded from high security to informative. The reason is it is not possible for a user account to hold an unregistered balance less than the u128::Max value as assumed by the development team. Thus, it is not possible for any user to hold an unregistered asset in Omnipool. Thus, this issue is not possible to happen under this assumption.

However, the code implementing the balance transfer of accounts is out of this audit scope and is also not provided. We are not sure if the above assumption "a user cannot hold any balance of an unregistered asset" is a valid one. We prefer to leave the issue as it is and warn the users not to transfer an unregistered asset to the Omnipool account.

3. Assumption not checked: LRNA imbalance is negative.

[Severity: Informative | Difficulty: Low | Category: Functional Correctness]

Issue: The LRNA imbalance should always be ≤ 0 for now. Relying on this, `sell` passes the absolute value of the hub asset imbalance to `calculate_sell_state_changes` (line 951), ignoring the sign. This is fine, but the code would feel safer if it checked that the imbalance is, indeed, negative.

Effects: No bad effects known.

Recommendation: Check that the imbalance is ≤ 0.

Status: submitted Issue #434, fixed in PR #444.

4. Updating HubAssetTradability could cause problems

[Severity: Medium | Difficulty: Easy | Category: Informative]

Issue: In SetAssetTradability() call, HubAssetTradability has limited room for a change, however, there are not sufficient restrictions on the possible changes. It is a design problem rather than an implementation error.

Effects: For example, if the "AddLiquidity" is to be set for HubAsset, then the AddLiquidity interface might not work for HubAsset, since the precondition checking for $Assets.contains_key(HubAssetId)$ would not pass.
Furthermore, if $HubAsset$ is to be added to $Assets$ to satisfy the condition checking, then in the Omnipool's state mode, we have both state variables $HubAssetTradability/\ TotalTVL$ etc. and $AssetDetails$ struct in Assets to track the settings of Hub Asset, that would easily causing in-consistency problems which would further lead to incorrect calculation and economic loss.

Recommendation: If flexibility/future extensibility is considered for Hub Asset, we might recommend making Hub Asset one of the Assets in the $Assets$ map store.

Status: Submitted in issue #442, confirm fixed in PR #457.

5. Storage access in one place

[Severity: - | Difficulty: - | Category: Code Improvement]

Issue: In function `update_imbalance`, `<HubAssetImbalance<T>>::get()` is passed as an argument. It is not necessary. It is better to remove it and access this storage variable within the scope of the function, keeping storage read and write in one place.

Effects: Code readability would be improved if this change is made unless there are other reasons.

Recommendation: revise the function as follows

```
fn update_imbalance(delta_imbalance: BalanceUpdate<Balance>) -> DispatchResult {
    let current_imbalance = <HubAssetImbalance<T>>::get();
    let imbalance = match delta_imbalance {
        BalanceUpdate::Decrease(amount) => current_imbalance.sub(amount).ok_or(ArithmeticError::Overflow)?,
        BalanceUpdate::Increase(amount) => current_imbalance.add(amount).ok_or(ArithmeticError::Overflow)?,
    };
    <HubAssetImbalance<T>>::put(imbalance);

    Ok(())
}
```

Status: Issue Submitted, #441, fixed in PR #450.

6. Incorrect error type in the buy action when there is not enough reserve value for asset_in in the pool

[Severity: Low | Difficulty: Medium | Category: Informative]

Issue: Returns ArithmeticError::Overflow, but should be returning Error::NotEnoughAsset(asset_in).

https://github.com/galacticcouncil/HydraDX-node/blob/20c8ac6079b8b3d6f16a9f15e9e43353ea6b9abc/pallets/omnipool/src/lib.rs#L1052

https://github.com/galacticcouncil/HydraDX-math/blob/f4dab244bb5d57971325f0ba6fbb8f6c1353beeb/src/omnipool/math.rs#L173

Effects: Describe the consequences if not fixed.

Recommendation: Recommended way to fix it. If there is no straightforward solution, leave it for discussion with the team.

Status: Discussed with Martin, fixing this problem could cause a redesign of the whole function. Since it does not affect the correctness of the computation, it is to be left as it is now.

7. Incorrect checking for sufficient balance

[Severity: Low | Difficulty: Low | Category: Informative]

Issue: A pre condition is checked to guarantee if there is sufficient liquidity to pay out user's buy request on the Line #1068. However, this checking against the free balance of the account and the requested amount does not guarantee the withdrawal will be successful. It could be possible that there is a certain amount frozen in the account such that the withdrawable amount (free balance minus frozen balance) is less than the requested amount.

Effects: If the intention of this condition checking is to terminate the computation early when there is not sufficient liquidity, then this purpose will fail in certain situations when there is an amount frozen in the account.

Recommendation: Use the crate function *ensure_can_withdraw*() to check liquidity sufficiency

Status: Issue Submitted, #440.

8. Duplicate computation of protocol_fee_amount when selling non-LRNA assets to the pool.

[Severity: - | Difficulty: - | Category: Code Improvement]

Issue: *protocol_fee_amount* was calculated when calling *amount_without_fee* in *math:: calculate_sell_state_changes*.

Refer to the GitHub issue # (with a link)

Effects: Describe the consequences if not fixed.

Recommendation: Reuse the protocol_fee_amount computation in calculating delta_hub_reserve_out, and save the computation resources.

Status: Issue Submitted, #51, fixed in PR#54.

9. Majority of the liquidity will be owned by the pool if the withdrawal happens when the price drop significantly

[Severity: - | Difficulty: - | Category: Informative]

Issue: The protocol share of an asset is monotonically increasing in the current design. When withdrawing liquidity when the asset price dropped significantly, the majority of shares will be contributed to the pool.

Effects: Describe the consequences if not fixed.

Recommendation: Recommended way to fix it. If there is no straightforward solution, leave it for discussion with the team.

Status: Issue Submitted, in #454.

10. Some condition checking in the specification is not implemented.

[Severity: - | Difficulty: - | Category: Functional Correctness]

Issue: The specification requires some condition checking such as the shares to be withdrawn and the amount to be swapped greater than 0, it is not checked in corresponding operations.

1. Specification requires only registered assets can be added to the pool, however in operation `initialize_pool()`, it is not checked if `StableCoinAssetId` exists in AssetRegistry.

   https://github.com/galacticcouncil/HydraDX-node/blob/16cdbcf25ef2f1dac5c3569d8cd44cc028ec4076/pallets/omnipool/src/lib.rs#L364

   Note that, `HDXAssetId` is registered at genesis. Furthermore, since `Omnipool::Assets` and `AssetRegistry` are managed separately, they do not imply each other.

2. The amount to be added should be greater than 0 as the specification document requires.

   https://github.com/galacticcouncil/HydraDX-node/blob/16cdbcf25ef2f1dac5c3569d8cd44cc028ec4076/pallets/omnipool/src/lib.rs#L601

   The condition checked above does not guarantee `amount >0`

3. The amount of shares to be removed should be greater than 0.

   https://github.com/galacticcouncil/HydraDX-node/blob/16cdbcf25ef2f1dac5c3569d8cd44cc028ec4076/pallets/omnipool/src/lib.rs#L734

Effects: arguments that are not eligible should be rejected early with the correct error code.

Recommendation: add specific checks to be consistent with the specification.

Status: Issue submitted in #452, issue 10.1 is fixed in PR #458 and issue 10.3 is fixed in PR#459.

For issue 10.2, there is the integrity test implemented to assure MinimumPoolLiquidity is greater than 0, which could imply an *amount* greater than 0.

Follow-up Audit Report for Major Fixes

Table of Contents

- Protocol Design and Implementation
  - State Model (Abstracted):
    - Derived state definitions:
    - Invariants
  - Operation: Pool Initializing
    - Is the specification respected in the implementation?
    - Invariants Checking:
    - Rounding Error Analysis
  - Operation: Add Tokens:
    - Is the specification respected in the implementation?
    - Invariants Checking
    - Rounding Error Analysis
  - Operation: Adding Liquidity- Single Asset Liquidity Provision
    - Is the specification respected in the implementation?
    - Invariants Checking
    - Rounding Error Analysis
  - Operation: Removing Liquidity
    - Is the specification respected in the implementation?
    - Invariants Checking
    - Rounding Error Analysis
  - Operation: Sell asset_in for asset_out, none is LRNA
    - Is the specification respected in the implementation?
    - Invariants Checking
    - Rounding Error Analysis
  - Operation: Sell LRNA for asset_out
    - Is the specification respected in the implementation?
    - Invariants Checking
    - Rounding Error Analysis
  - Operation: Buy asset_out using asset_in, none is LRNA
    - Is the specification respected in the implementation?
    - Invariants Checking
    - Rounding Error Analysis
  - Operation: Buy asset_out using LRNA
    - Is the specification respected in the implementation?
    - Invariants Checking
    - Rounding Error Analysis
- Issues

Redundant computation when calculate remove liquidity state change
Rounding error exploit when removing liquidity
Consistency: ImbalanceLRNA.negative is not a precondition in most exchange types.
Possibly large error in the imbalance when buying with LRNA

Protocol Design and Implementation

State Model (Abstracted):

$Balance_{token}$ represent the variables from implementation

$[token\ =\ HDX]\ NativeCurrency::AccountStore("Omnipool").free$

$[token\ \neq\ HDX]\ MultiCurrency::Accounts("Omnipool")(token).free$

$TotalIssuance_{token}$ represents $MultiCurrency::TotalIssuance(token)$

$HubReserve_{Token}$ represents $Assets(token).hub_reserve$, meaning the amount of the hub asset (LRNA) in the $token/LRNA$ subpool. This notation replaced the $Value_{token}$ in the original audit report.

$Imbalance_{LRNA}$, represents $HubAssetImbalance$

Derived state definitions:

$$Price_{token} = \frac{HubReserve_{token}}{Balance_{token}}$$

Invariants

1. **Swap_Invariant**: Constant product should be maintained in each subpool when fees are not considered,

$$Balance_{token} * Balance^{token}_{LRNA} = Balance_{token}' * Balance^{token}_{LRNA}'$$

2. **Non_Positive_Imbalance:** At the moment, imbalance can only be negative or 0. There should not be a way / scenario where it would become positive,

$$Imbalance_{LRNA}.negative = true$$

3. **HubAsset_Total:** The total issued hub asset (LRNA in current case) should be equal to the LRNA asset held by the pool and all the LRNA asset that is given to liquidity providers,

$$TotalIssuance_{LRNA} = Accounts(Omnipool)(LRNA).total + \sum_{a \in Accounts}^{a \neq Omnipool} Accounts(a)(LRNA).total$$

4. **HubAsset_Accounting**: The total hub reserve of all assets (except the hub asset) should be equal to the balance of the hub asset (LRNA),

$$Balance_{LRNA} = \sum_{t \in Assets} HubReserve_{t}$$

Operation: Pool Initializing

Specification:

1. Only registered assets can be added to Omnipool.
2. First two assets in the pool must be Stable Asset and Native Asset (HDX token).
3. Stable asset balance and native asset balance must be transferred to the Omnipool account manually before initializing the pool.
4. Initial liquidity of the new token being added to Omnipool must be transferred manually to the pool account prior to calling add_token.
5. Initial price of tokens is manually set by technical origin (1 hour delay) after the new token is approved by governance origin through DAP (1 day delay.)

Implementation:

Transition:

$initialize_pool(\ origin,\ stable_asset_price,\ native_asset_price,\ stable_weight_cap,\ native_weight_cap)$

Settings:

$Balance_{stable} = MultiCurrency::\ Accounts("Omnipool")(StableCoinAssetId).free$

$Balance_{HDX} = NativeCurrency::AccountStore("Omnipool").free$

$Price_{stable} = stable_asset_price$

$Price_{HDX} = native_asset_price$

$HubReserve_{stable}' = \Delta HubReserve_{stable} = \lfloor Price_{stable} * Balance_{stable} \rfloor$

$\Delta HubReserve_{HDX} = \lfloor Price_{HDX} * Balance_{HDX} \rfloor$

$\Delta Balance_{LRNA} = \Delta HubReserve_{stable} + \Delta HubReserve_{HDX}$

$tvl = \lfloor Balance_{LRNA} * Balance_{stablecoin} / HubReserve_{stablecoin} \rfloor$

Pre conditions:

1. Origin is TechnicalOrigin
2. $StableCoinAssetId \notin Assets$
3. $HdxAssetId \notin Assets$
4. $Balance_{StableCoin} > 0$
5. $Balance_{HDX} > 0$
6. $StableCoinAssetId \in AssetRegistry$
7. $Price_{stable} > 0$
8. $Price_{HDX} > 0$

Post state:

Accounts:

$Balance_{LRNA} += \Delta Balance_{LRNA}$

$TotalIssuance_{LRNA}$ $+=$ $\Delta Balance_{LRNA}$

Omnipool States:

Assets(StableCoinAssetId)= {

hub_reserve: $\Delta HubReserve_{stable}$,

shares: $Balance_{stable}$,

protocol_shares: $Balance_{stable}$,

cap: stable_weight_cap,

tradable: SELL | BUY | ADD_LIQUIDITY | REMOVE_LIQUIDITY,

}

Assets(NativeAssetId) = {

hub_reserve: $\Delta HubReserve_{HDX}$,

shares: $Balance_{HDX}$,

protocol_shares: $Balance_{HDX}$,

cap: native_weight_cap,

tradable: SELL | BUY | ADD_LIQUIDITY | REMOVE_LIQUIDITY,

}

$tradable_{LRNA} = SELL$

Post condition:

$tvl \leq TVLCap$

Conclusion:

Is the specification respected in the implementation?

Specifications #1 - 5 are satisfied by this implementation. However, it remains a concern that it relies on the technical origin to set a reasonable price for the stable coin asset and the native asset.

Invariants Checking:

Invariants 2-4, i.e., Non_Positive_Imbalance, HubAsset_Total, HubAsset_Accouting are guaranteed.
Proof is trivial.

Rounding Error Analysis

1. Amount of LRNA coin reserved for $stablecoin/LRNA$ subpool is at most 1 less, but will never be more than it should be, a.k.a., $\delta_{HubReserve_{stable}} \in (-1, 0]$.
2. Amount of LRNA coin reserved for $HDX/LRNA$ subpool is at most 1 less, but will never be more than it should be, a.k.a., $\delta_{HubReserve_{HDX}} \in (-1, 0]$.
3. Minted LRNA coin could be 2 less but never more than it should be, a.k.a., $\delta_{Balance_{LRNA}} \in (-2, 0]$.

The above rounding differences are favourable to the pool and have negligible effects.

Operation: Add Tokens:

Specification:

1. Token can only be added after pool is initialized
2. Token can only be added after it is registered in asset registry
3. NextPositionId is not a key in Positions
4. The implementation should be consistent with the design specification: Add Token Spec.

Implementation:

Transition:

add_token(origin, asset, initial_price, weight_cap, position_owner)

Settings:

$Price_{asset} = initial_price$

$\Delta HubReserve_{asset} = \lfloor Price_{asset} * Balance_{asset} \rfloor$

$\Delta Imbalance = \lfloor \Delta HubReserve_{asset} * Imbalance_{LRNA}.value / Balance_{LRNA} \rfloor$

$tvl = \lfloor Balance_{LRNA}_post * Balance_{stablecoin} / HubReserve_{stablecoin} \rfloor$

Pre conditions:

1. Origin is AddTokenOrigin
2. $asset \notin Assets$
3. $asset \in AssetRegistry::Assets$
4. $Price_{asset} > 0$
5. $Balance_{asset} \geq MinimumPoolLiquidity$
6. $Balance_{asset} > 0$
7. $Imbalance_{LRNA}.negative$

Post state:

Accounts:

$Balance_{LRNA}' = Balance_{LRNA} + \Delta HubReserve_{asset}$

$TotalIssuance_{LRNA} += \Delta HubReserve_{asset}$

Omnipool states:

$Imbalance_{LRNA}' = (Imbalance_{LRNA}.value + \Delta Imbalance_{LRNA}, true)$

Assets(asset)= {

hub_reserve: $\Delta HubReserve_{asset}$,

shares: $Balance_{asset}$,

protocol_shares: 0,

cap: weight_cap,

tradable: SELL | BUY | ADD_LIQUIDITY | REMOVE_LIQUIDITY,

}
Positions(NextPositionId) = {
asset_id: asset,
amount: $Balance_{asset}$,
shares: $Balance_{asset}$,
price: $Price_{asset}$,
}
$NextPositionId\ += 1$
$NFT_{Omnipool}\ += NextPositionId \mapsto position_owner$

Post condition:

$Imbalance_LRNA'.negative$
$tvl\ <= TVLCap$

Conclusion:

Is the specification respected in the implementation?

Specifications 1-4 are satisfied.

Invariants Checking

Invariants 2-4, i.e., Non_Positive_Imbalance, HubAsset_Total, HubAsset_Accouting are guaranteed.
Proof is trivial.

Rounding Error Analysis

1. The minted amount of LRNA, which is the same as the amount of LRNA allocated to $asset/LRNA$ subpool, has a rounding difference $\delta_{Balance_{LRNA}} = \delta_{HubReserve_{asset}} \in (-1, 0]$.
2. The imbalance of the Hub Asset, LRNA has a rounding difference $\delta_{Imbalance_{LRNA}.value} \in (-\frac{Imbalance_{LRNA}.value}{Balance_{LRNA}} - 1, 0]$, a.k.a., considering $Imbalance_{LRNA}$ is a negative value, this rounding difference results in at most $\frac{1}{Balance_{LRNA}} + \frac{1}{Imbalance_{LRNA}.value}$ percent less than it should be. This rounding difference is negligible.

Operation: Adding Liquidity- Single Asset Liquidity Provision

Specification:

1. `add_liquidity` adds specified asset amount to pool and in exchange gives the origin corresponding shares amount in the form of NFT at current price.
2. Asset's tradable state must contain the ADD_LIQUIDITY flag, otherwise a `NotAllowed` error is returned.
3. Asset weight cap must be respected, otherwise `AssetWeightExceeded` error is returned. Asset weight is the ratio between the new HubAsset reserve and the new total reserve of HubAsset in Omnipool.
4. Adding liquidity should leave the pool price of the asset unchanged.
5. Adding liquidity should leave the token to share ratio (i.e., $\frac{Balance_{asset}}{shares_{asset}} = \frac{Balance_{asset}'}{shares_{asset}'}$) unchanged.
6. The amount of the added liquidity should be greater than 0.
7. The implementation of state changes should be consistent with the math specification in the design document, Add Liquidity Spec.
8. NextPositionId is not a key in Positions.

Implementation:

Transition:

add_liquidity(origin, asset, amount)

Settings:

$who = ensure_signed(origin)$

$\Delta Balance_{asset} = amount$

$Price_{asset} = \frac{HubReserve_{asset}}{Balance_{asset}}$

$Price_{stable} = \frac{HubReserve_{stable}}{Balance_{stable}}$

$\Delta HubReserve_{asset} = \lfloor Price_{asset} * \Delta Balance_{asset} \rfloor$

$\Delta shares_{asset} = \lfloor \frac{shares_{asset} * \Delta Balance_{asset}}{Balance_{asset}} \rfloor$

$weight_{asset} = \frac{HubReserve_{asset}'}{Balance_{LRNA}'} = \frac{HubReserve_{asset} + \Delta HubReserve_{asset}}{Balance_{LRNA} + \Delta HubReserve_{asset}}$

$Price_{asset}' = \frac{\Delta HubReserve_{asset} + HubReserve_{asset}}{amount + Balance_{asset}}$

$\Delta Imbalance = \lfloor \Delta HubReserve_{asset} * Imbalance_{LRNA}.value * Balance_{LRNA} \rfloor$

$tvl = \lfloor Balance_{LRNA}_post * Balance_{stablecoin} / HubReserve_{stablecoin} \rfloor$

Pre conditions:

1. $amount \geq MinimumPoolLiquidity$
2. $amount = 0 \lor Balance^{who}_{asset} - \Delta Balance_{asset} \geq MultiCurrency::Accounts(who)(asset).frozen$
3. $ADD_LIQUIDITY \in tradable_{asset}$
4. $Imbalance_{LRNA}.negative$

Post state:

<u>Accounts:</u>

$Balance_{asset} += \Delta Balance_{asset}$

$Balance^{who}_{asset} -= \Delta Balance_{asset}$

$Balance_{LRNA} += \Delta HubReserve_{asset}$

$TotalIssuance_{LRNA} += \Delta HubReserve_{asset}$

<u>Omnipool states:</u>

$Imbalance_{LRNA}_post = (Imbalance_{LRNA}.value + \Delta imbalance, true)$

Assets(asset)= {
hub_reserve: $HubReserve_{asset} + \Delta HubReserve_{asset}$,
shares: $shares_{asset} + \Delta shares_{asset}$,
...
}

$NFT_Omnipool += NextPositionId \mapsto who$

$Positions += NextInstanceId \mapsto \{$
asset,
amount,
$\Delta shares_{asset}$,
$Price_{asset}$'
$\}$

$NextPositionId += 1$

Post condition:

$weight_{asset} \leq cap_{asset}$

$tvl \leq TVLCap$

$Imbalance_{LRNA}.negative$

Conclusion:

Is the specification respected in the implementation?

1. Specifications 1-3,7 are guaranteed.
2. Specifications 4 and 5 are guaranteed up to rounding.
3. Specification 6 is depending on $MinimumPoolLiquidity > 0$, which is not guaranteed by integration tests.

4. Specification 8 is guaranteed since the position id is monotonically increasing.

Invariants Checking

Invariants 2-4, i.e., Non_Positive_Imbalance, HubAsset_Total, HubAsset_Accouting are guaranteed.
Proof is trivial.

Rounding Error Analysis

1. $\delta_{\Delta HubReserve_{asset}} \in (-1, 0]$
2. $\delta_{\Delta shares_{asset}} \in (-1, 0]$

These rounding differences are insignificant and are considered favorable to the pool.

Operation: Removing Liquidity

Specification:

1. `remove_liquidity` takes shares in the form of a NFT from the origin, and returns the asset corresponding to the amount of shares. If the current price is larger than the price at investment time, the pool will also pay some amount of LRNA to the liquidity provider.
2. Asset's tradable state must contain the REMOVE_LIQUIDITY flag, otherwise a `NotAllowed` error is returned.
3. Removing liquidity should not change the pool price of the asset.
4. Removing liquidity should not change the token to share ratio, i.e., $\frac{Balance_{asset}}{shares_{asset}} = \frac{Balance_{asset}'}{shares_{asset}'}$.
5. The amount of the shares requested to remove should be greater than 0.
6. If there are no remaining shares, burn the old Position NFT. Otherwise, update the position.
7. The implementation of state changes should be consistent with the math specification in the design document, Withdraw Liquidity Spec.
8. NextPositionId is not a key in Positions
9. Impermanent loss
    a. Defined as $\frac{ValueInvestAndWithdraw}{ValueHold} - 1$
    b. Equal to $\frac{2 \cdot \sqrt{InvestPrice_T \cdot WithdrawPrice_T}}{InvestPrice_T + WithdrawPrice_T}$
    c. It's the geometric mean over the arithmetic mean of T's price at investment time and withdrawal time, which means that it must be between 0 (exclusive, assuming reasonable prices) and 1 (inclusive).

Implementation:

Transition:

remove_liquidity(origin, position_id, amount)

Settings:

$who = ensure_signed(origin)$

$asset_id = Positions(position_id).asset_id$

$Price_{position_id} = Position(position_id).price$

$Price_{asset_id} = \frac{HubReserve_{asset_id}}{Balance_{asset_id}}$ is the current price of the asset over the stable coin asset.

$p_x_r = \lfloor Price_{position_id} * Balance_{asset_id} \rfloor$

$\Delta Balance_{position_id} = \lfloor \frac{amount * Balance_{position_id}}{Shares_{position_id}} \rfloor$

$\Delta b = [Price_{asset_id} < Price_{position_id}] * (\lfloor \frac{(p_x_y - HubReserve_{asset_id})*amount}{p_x_y + HubReserve_{asset_id}} \rfloor + 1)$

$$\Delta shares_{asset_id} = amount - \Delta b$$

$$\Delta Balance_{asset_id} = \lfloor \frac{Balance_{asset_id} * \Delta shares_{asset_id}}{shares_{asset_id}} \rfloor$$

$$\Delta HubReserve_{asset_id} = \lfloor \frac{\Delta Balance_{asset_id} * HubReserve_{asset_id}}{Balance_{asset_id}} \rfloor$$

$$\Delta imbalance = \lfloor \frac{\Delta HubReserve_{asset_id} * Imbalance_{LRNA}.value}{Balanace_{LRNA}} \rfloor$$

$$div1 = \lfloor HubReserve_{asset_id} * \frac{HubReserve_{asset_id} - p_x_y}{HubReserve_{asset_id} + p_x_y} \rfloor$$

$$\Delta Balance^{who}_{LRNA} = [Price_{asset_id} > Price_{position_id}] * \lfloor \frac{div1 * \Delta Shares_{asset_id}}{Shares_{asset_id}} \rfloor$$

$$\Delta Balance_{LRNA}_toburn = \Delta HubReserve_{asset_id} - \Delta Balance^{who}_{LRNA}$$

$$tvl' = \lfloor \frac{Balance_{LRNA}_post * Balance_{stablecoin}}{HubReserve_{stablecoin}} \rfloor$$

Pre conditions:

1. $amount > 0$
2. $NFT_{Omnipool}(position_id) = who$
3. $position_id \in Positions$
4. $Position(position_id).shares \geq amount$
5. $StableCoinAssetId \in Assets$
6. $asset_id \in Assets$
7. $REMOVE_LIQUIDITY \in tradable_{asset_id}$
8. $Imbalance_{LRNA}.negative$
9. $Balance^{who}_{asset_id} + \Delta Balance_{asset_id} \geq ExistentialDeposit_{asset_id} \lor who \in DustRemovalWhitelist$
10. $Balance_{asset_id} - \Delta Balance_{asset_id} \geq MultiCurrency::Accounts(Omnipool)(asset_id).frozen$
11. $Balance_{LRNA} - \Delta HubReserve_{asset_id} \geq MultiCurrency::Accounts(Omnipool, LRNA).frozen$
12. $Balance^{who}_{LRNA} + \Delta Balance^{who}_{LRNA} \geq ExistentialDeposit_{LRNA} \lor who \in DustRemovalWhitelist$

Post state:

Accounts:

$$Balance_{asset_id} -= \Delta Balance_{asset_id}$$

$$Balance^{who}_{asset_id} += \Delta Balance_{asset_id}$$

$$Balance_{LRNA}_post = Balance_{LRNA} - \Delta Balance_{LRNA}_toburn - \Delta Balance^{who}_{LRNA} = Balance_{LRNA} - \Delta HubReserve_{asset_id}$$

$$TotalIssuance_{LRNA} -= \Delta Balance_{LRNA}_toburn$$

$$Balance^{who}_{LRNA} += \Delta Balance^{who}_{LRNA}$$

Omnipool Assets:

$$Imbalance_{LRNA}' = (Imbalance_LRNA.value - \Delta imbalance, true)$$

$Assets(asset_id)' = \{$

$hub_reserve: HubReserve_{asset_id} - \Delta HubReserve_{asset_id}$

$shares: shares_{asset_id} - \Delta shares_{asset_id}$

$protocol_shares: shares_{protocol} + \Delta b$

...

}

$Positions'$ and $NFT_{Omnipool}'$:

$[shares_{position_id} - amount = 0]\ Positions.remove(position_id)\ \&\&\ NFT_{Omnipool}.burn(position_id)$

$[shares_{position_id} - amount \neq 0]\ Positions(position_id) =$

{ $asset_id$: _,

$amount: Balance_{position_id} - \Delta Balance_{position_id}$,

$shares: shares_{position_id} - amount$,

$price$: _,

}

Post condition:

$Imbalance_{LRNA}'.negative$

$tvl' \leq TVLCap$

Conclusion:

Is the specification respected in the implementation?

Yes, the specifications listed above are satisfied in the implementation, where specification 3 and 4 are guaranteed up to rounding.

Invariants Checking

Invariants #2-4 identified in the state model are satisfied after this transaction.

Rounding Error Analysis

1. $\Delta b = [Price_{asset_id} < Price_{position_id}] * (\lfloor \frac{(p_x_y - HubReserve_{asset_id})*amount}{p_x_y + HubReserve_{asset_id}} \rfloor + 1)$, indicates the amount of shares contributed to the pool when withdrawing at a lower price. We have $\delta_{\Delta b} \in (-\frac{2*amount}{(\overline{p_x_y} + HubReserve_{asset_id})^2}, 1)$.

   This rounding is diverging. However, the rounding difference that is not favorable to the pool is fairly small.
2. $\Delta shares = amount - \Delta b$ is the amount of shares leaving the pool.
   a. When the price is up, there is no rounding error.
   b. When price drop, we have $\Delta shares_{asset_id} = amount - (\overline{\Delta b} + \delta_{\Delta b})$, where $\delta_{\Delta shares_{asset_id}} = -\delta_{\Delta b}, \in (-1, \frac{2*amount}{(\overline{p_x_y} + HubReserve_{asset_id})^2})$.

3. $\Delta Balance_{asset_id}$ indicates the amount of the asset paid out to the eligible liquidity provider from the pool, $\Delta Balance_{asset_id} = \lfloor \frac{Balance_{asset_id} * \Delta shares_{asset_id}}{shares_{asset_id}} \rfloor$.

   Thus,

   1) when the price is up (a.k.a., $Price_{asset_id} \geq Price_{position_id}$), $\delta_{Balance_{asset_id}} \in (-1, 0]$

   2) when the price drops, we have

$$\delta_{Balance_{asset_id}} \in (-\frac{Balance_{asset_id}}{shares_{asset_id}} - 1, \frac{Balance_{asset_id}}{shares_{asset_id}} * \frac{2 * amount}{(\overline{p_x_y} + HubReserve_{asset_id})^2}).$$

Note that, when price drops the rounding effect is diverging, however the upper bound (which is not favorable to the pool) is bounded by a small number that is unlikely to cause significant loss to the pool.

4. $\Delta Balance^{who}_{LRNA}$ indicates the amount of hub asset paid to the liquidity provider in the case that the current asset price is greater than the position price. It is being calculated as follows:

   $p_x_r = \lfloor Price_{position_id} * Balance_{asset_id} \rfloor$

   $div1 = \lfloor HubReserve_{asset_id} * \frac{HubReserve_{asset_id} - p_x_y}{HubReserve_{asset_id} + p_x_y} \rfloor$

   $\Delta Balance^{who}_{LRNA} = [Price_{asset_id} > Price_{position_id}] * \lfloor \frac{div1 * \Delta Shares_{asset_id}}{Shares_{asset_id}} \rfloor$.

   When price is up, $\delta_{\Delta Balance^{who}_{LRNA}} \in (-1, \frac{amount}{Shares_{asset_id}} * \frac{2}{(\frac{Price_{positio_id}}{Price_{asset_id}} - \frac{1}{HubReserve_{asset_id}})^2})$. Though it is likely for the liquidity provider to receive more LRNA than should be when price is up, it is difficult for a malicious liquidity provider to execute a profitable exploit.

   Note: In case a malicious liquidity provider wants to exploit this rounding difference, he has to hold majority shares of this asset and swap a significant amount of asset before attempting to remove liquidity, such that he has to pay a significant amount of fees and outrun arbitrage bots.

   Thus, we think this rounding difference is considerably safe for the pool with a potential, difficult-to-execute risk.

5. $\Delta HubReserve_{asset_id} = \lfloor \frac{\Delta Balance_{asset_id} * HubReserve_{asset_id}}{Balance_{asset_id}} \rfloor$ indicates the changes to the hub reserve of the asset, $asset_id$. We have

   a. When price goes up, $\delta_{\Delta HubReserve_{asset_id}} \in (-\frac{HubReserve_{asset_id}}{Balance_{asset_id}} - 1, 0]$

   b. When price drops,

$$\delta_{\Delta HubReserve_{asset_id}} \in (-\frac{HubReserve_{asset_id}}{shares_{asset_id}} - \frac{HubReserve_{asset_id}}{Balance_{asset_id}} - 1, \frac{HubReserve_{asset_id}}{shares_{asset_id}} * \frac{2 * amount}{(\overline{p_x_y} + HubReserve_{asset_id})^2}).$$

   The rounding difference is diverging, however the difference could be fairly small and would have limited capacity to cause a significant pool loss.

6. $\Delta imbalance = \lfloor \frac{\Delta HubReserve_{asset_id} * Imbalance_{LRNA}.value}{Balanace_{LRNA}} \rfloor$ indicates the change to the current imbalance (always negative), a.k.a., $-(Imbalance_{LRNA}.value - \Delta Imbalance)$. Thus, we have

   a. W hen price is up, $\delta_{\Delta imbalance} \in ((-Price_{asset_id} - 1) * \frac{Imbalance_{LRNA}.value}{Balance_{LRNA}}, 0]$.

b. When price is down,

$$\delta_{\Delta imbalance} \in ((-\frac{HubReserve_{asset_id}}{Shares_{asset_id}} - Price_{asset_id} - 1) * \frac{Imbalance_{LRNA}.value}{Balance_{LRNA}}, \frac{HubReserve_{asset_id}}{shares_{asset_id}} * \frac{2 * amount}{(\overline{p_x_y} + HubReserve_{asset_id})^2})$$

The rounding difference for both cases is monotonic and is a pretty small fraction of the imbalance (close to $\frac{Price_{asset_id}}{Balance_{LRNA}}$). Thus, the rounding difference is reasonable.

Operation: Sell asset_in for asset_out, none is LRNA

Specification:

1. $SELL \in tradable_{asset_in}$ and $BUY \in tradable_{asset_out}$, otherwise a `NotAllowed` error is returned.
2. Fees
   a. Protocol fee
      Taken from the LRNA amount. Used first to cover the LRNA imbalance, then added to the native asset subpool (LRNA-HDX)
      $fee_{protocol} = ProtocolFee * \Delta HubReserve_{asset_in}$.
   b. Asset fee
      Taken from the $asset_out$ amount
      $fee_{asset} = AssetFee * \Delta HubReserve_{asset_out}$
      Returned to the $asset_out$ pool.
3. The swap will maintain the swap invariant in the subpools, asset_in/LRNA and asset_out/LRNA, if fees are not considered.
4. The implementation should be consistent with the design specification: Swap Spec.
5. The hub asset will only be burned but never minted in swap events.

Implementation:

Transition:

sell(origin, asset_in, asset_out, amount, min_buy_amount)

Settings:

$who = ensure_signed(origin)$

$\Delta Balance_{asset_in} = amount$

$\Delta HubReserve_{asset_in} = \lfloor \frac{amount * HubReserve_{asset_in}}{Balance_{asset_in} + amount} \rfloor$

$\Delta HubReserve_{asset_out} = \Delta HubReserve_{asset_in} - \lfloor ProtocolFee * \Delta HubReserve_{asset_in} \rfloor$

$\Delta Balance_{asset_out}_0 = \frac{Balance_{asset_out} * \Delta HubReserve_{asset_out}}{HubReserve_{asset_out} + \Delta HubReserve_{asset_out}}$

$\Delta Balance_{asset_out} = \lfloor (1 - AssetFee) * \lfloor \Delta Balance_{asset_out}_0 \rfloor \rfloor$

$Fee_{protocol} = \lfloor ProtocolFee * \Delta HubReserve_{asset_in} \rfloor$

$\Delta Imbalance = min(Fee_{protocol}, Imbalance_{LRNA}.value)$

$\Delta HubReserve_{HDX} = max(0, Fee_{protocol} - \Delta Imbalance)$

$\Delta Balance_{LRNA} = \Delta HubReserve_{asset_out} + \Delta HubReserve_{HDX} - \Delta HubReserve_{asset_in}$

Pre conditions:

1. $asset_in \neq asset_out$
2. $amount \geq Limit_{minimum_trading}$
3. $Balance^{who}_{asset_in} - amount \geq MultiCurrency: Accounts(who)(asset_in).frozen$
4. $asset_in \neq LRNA \wedge asset_out \neq LRNA$
5. $asset_in \in Assets \wedge asset_out \in Assets \wedge HDX \in Assets$
6. $SELL \in tradable_{asset_in} \wedge BUY \in tradable_{asset_out}$
7. $\Delta HubReserve_{asset_out} + \Delta HubReserve_{HDX} \leq \Delta HubReserve_{asset_in}$
8. $Balance_{asset_out} - \Delta Balance_{asset_out} \geq MultiCurrency::Accounts(Omnipool)(asset_out).frozen$
9. $MultiCurrency::Accounts(who)(asset_out).total() + \Delta Balance_{asset_out} \geq ExistentialDeposit_{asset_out}$
   $|| \, who \in DustRemovalWhitelist$
10. $Balance_{LRNA} - (\Delta HubReserve_{asset_in} - (\Delta HubReserve_{asset_out} + \Delta HubReserve_{HDX})) \geq$
    $MultiCurrency::Accounts(Omnipool)(LRNA).frozen$ if
    $\Delta HubReserve_{asset_out} + \Delta HubReserve_{HDX} \leq \Delta HubReserve_{asset_in}$

Post state:

Accounts:

$Balance^{who}_{asset_in} \; -= \; amount$

$Balance_{asset_in} \; += \; amount$

$Balance_{asset_out} \; -= \; \Delta Balance_{asset_out}$

$Balance^{who}_{asset_out} \; += \; \Delta Balance_{asset_out}$

$Balance_{LRNA} \; += \; \Delta Balance_{LRNA}$

$TotalIssuance_{LRNA} \; += \; \Delta Balance_{LRNA}$

Omnipool:

$Imbalance_{LRNA} \; -= \; \Delta Imbalance$

$Assets(HdxAssetId).hub_reserve \; += \; \Delta HubReserve_{HDX}$

$Assets(asset_in).hub_reserve \; -= \; \Delta HubReserve_{asset_in}$

$Assets(asset_out).hub_reserve \; += \; \Delta HubReserve_{asset_out}$

Post condition:

1. $\Delta Balance_{asset_out} \geq min_buy_amount$
2. $Imbalance'_{LRNA}.negative$

Conclusion:

Is the specification respected in the implementation?

1. Specifications 1, 2 and 5 are satisfied.
2. Specification 4 is guaranteed, proof same as checking invariant 1, Swap_Invariant.
3. Specification 6 is satisfied since

$$\begin{aligned}\Delta Balance_{LRNA} &= \Delta HubReserve_{asset_out} + \Delta HubReserve_{HDX} - \Delta HubReserve_{asset_in} \\ &= \Delta HubReserve_{asset_in} - Fee_{protocol} + \Delta HubReserve_{HDX} - \Delta HubReserve_{asset_in} \\ &= max(0,\ Fee_{protocol} - \Delta Imbalance) - Fee_{protocol} \\ &< 0.\end{aligned}$$

Invariants Checking

1. Invariant 1, the Swap_Invariant is guaranteed up to rounding, for subpools, $asset_in/LRNA$ and $asset_out/LRNA$, if fees are not considered. Proof is sketched as follows.

Proof: let $Fee_{protocol} = \lfloor ProtocolFee * \Delta HubReserve_{asset_in} \rfloor$,

$Fee_{Asset} = \Delta Balance_{asset_out_no_fee} - \Delta Balance_{asset_out}$,

$\Delta Balance_{asset_in} = amount$

$\Delta HubReserve_{asset_in} = \lfloor \frac{amount * HubReserve_{asset_in}}{Balance_{asset_in} + amount} \rfloor$

$\Delta HubReserve_{asset_out} = \Delta HubReserve_{asset_in} - Fee_{protocol}$

$\Delta Balance_{asset_out_no_fee} = \lfloor \frac{Balance_{asset_out} * \Delta HubReserve_{asset_out}}{HubReserve_{asset_out} + \Delta HubReserve_{asset_out}} \rfloor$

$\Delta Balance_{asset_out} = \lfloor (1 - AssetFee) * \Delta Balance_{asset_out_no_fee} \rfloor$

1) For $asset_in/LRNA$ subpool, we have

$Balance_{asset_in}' = Balance_{asset_in} + \Delta Balance_{asset_in} = Balance_{asset_in} + amount$

$Balance_{LRNA}^{asset_in}{}' = Balance_{LRNA}^{asset_in} - \Delta HubReserve_{asset_in} = HubReserve_{asset_in} - \lfloor \frac{amount * HubReserve_{asset_in}}{Balance_{asset_in} + amount} \rfloor$

Such that

$Balance_{asset_in}' * Balance_{LRNA}^{asset_in}{}' = (Balance_{asset_in} + amount) * (HubReserve_{asset_in} - \lfloor \frac{amount * HubReserve_{asset_in}}{Balance_{asset_in} + amount} \rfloor)$

If rounding is ignored,

$Balance_{asset_in}' * Balance_{LRNA}^{asset_in}{}' = Balance_{asset_in} * HubReserve_{asset_in} = Balance_{asset_in} * Balance_{LRNA}^{asset_in}$

If rounding is considered, the constant product will increase [0, $Balance_{asset_in} + amount$).

2) For $asset_out/LRNA$, if we ignore fees, we have

$$\begin{aligned}Balance_{asset_out}' &= Balance_{asset_out} - \Delta Balance_{asset_out_no_fee} \\ &= Balance_{asset_out} - \lfloor \frac{Balance_{asset_out} * \Delta HubReserve_{asset_out}}{HubReserve_{asset_out} + \Delta HubReserve_{asset_out}} \rfloor\end{aligned}$$

$Balance_{LRNA}^{asset_out}{}' = Balance_{LRNA}^{asset_out} + \Delta HubReserve_{asset_out}$

$Balance_{asset_out}' * Balance_{LRNA}^{asset_out}{}'$

$= (Balance_{asset_out} - \lfloor \frac{Balance_{asset_out} * \Delta HubReserve_{asset_in}}{HubReserve_{asset_out} + \Delta HubReserve_{asset_in}} \rfloor) * (HubReserve_{asset_out} + \Delta HubReserve_{asset_out})$

If rounding is ignored,

$Balance_{asset_out}' * Balance_{LRNA}^{asset_out}{}' = Balance_{asset_out} * HubReserve_{asset_out}$.

If rounding is considered, the constant product will increase [0, $HubReserve_{asset_out} + \Delta HubReserve_{asset_out}$).

3) According to the design spec and the implementation, the protocol fee collected from the trade will be used to cover the impermanent loss. However, if there is any amount left, it will send to the $HDX/LRNA$ subpool, thus, this trade will also increase the constant product in the $HDX/LRNA$ subpool.

In conclusion, after the swap,

1) The Swap_Invariant for the subpools $asset_in/LRNA$ and $asset_out/LRNA$ is guaranteed if any fees and rounding are ignored.
2) If considering rounding but no fees, the constant products will increase [0, $Balance_{asset_in} + amount$) for the $asset_in/LRNA$ subpool and [0, $HubReserve_{asset_out} + \Delta HubReserve_{asset_in}$) for the $asset_out/LRNA$ subpool.
3) If considered fees but but no rounding, the constant product for the $asset_in/LRNA$ subpool is maintained, but the constant product for the $asset_out/LRNA$ subpool will increase $\frac{AssetFee * Balance_{asset_out} * (1 - ProtocolFee) * HubReserve_{asset_in} * amount}{HubReserve_{asset_in} + amount}$.
4) The constant product for the $HDX/LRNA$ will also increase if the imbalance is 0 after the exchange.

2. Invariants 2, 3 and 4 i.e., Non_Positive_Imbalance, HubAsset_Total, HubAsset_Accouting are guaranteed. The proof is trivial.

Rounding Error Analysis

1. The hub asset in the $asset_in/LRNA$ subpool is indicated by $HubReserve_{asset_in}' = HubReserve_{asset_in} - \Delta HubReserve_{asset_in}$, where $\Delta HubReserve_{asset_in} = \lfloor \frac{amount * HubReserve_{asset_in}}{Balance_{asset_in} + amount} \rfloor$.
   Thus, $\delta_{\Delta HubReserve_{asset_in}} \in (-1, 0]$, and $\delta_{HubReserve_{asset_in}} \in [0, 1)$.
   This is rounded in favour of the pool.
2. The hub asset in the $asset_out/LRNA$ subpool is indicated by $HubReserve_{asset_out}' = HubReserve_{asset_out} - \Delta HubReserve_{asset_out}$ where $\Delta HubReserve_{asset_out} = \Delta HubReserve_{asset_in} - \lfloor ProtocolFee * \Delta HubReserve_{asset_in} \rfloor$.
   We have $\delta_{\Delta HubReserve_{asset_out}} \in (ProtocolFee - 1, 1)$ and $\delta_{HubReserve_{asset_out}} \in (-1, 1 - ProtocolFee)$.
   This is not always rounding in favour of the pool, but the difference is rather negligible.
3. The amount of $asset_out$ paid to the trader is indicated by $\Delta Balance_{asset_out} = \lfloor (1 - AssetFee) * \lfloor \Delta Balance_{asset_out}_0 \rfloor \rfloor$, where $\Delta Balance_{asset_out}_0 = \frac{Balance_{asset_out} * \Delta HubReserve_{asset_out}}{HubReserve_{asset_out} + \Delta HubReserve_{asset_out}}$
   We have

$\delta_{\Delta Balance_{asset_out}_0} \in (- \frac{\overline{\Delta Balance_{asset_out}_0} * (1 - ProtocolFee)}{HubReserve_{asset_out} + \overline{\Delta HubReserve_{asset_out}} - (1 - ProtocolFee)}, \frac{\overline{\Delta Balance_{asset_out}_0}}{HubReserve_{asset_out} + \overline{\Delta HubReserve_{asset_out}} + 1})$

$\delta_{\Delta Balance_{asset_out}} \in ((1 - AssetFee) * (\delta^{lower_bound}_{\Delta Balance_{asset_out}_0} - 1) - 1, (1 - AssetFee) * \delta^{upper_bound}_{\Delta Balance_{asset_out}_0})$

Thus, $\Delta Balance_{asset_out}_0$ can be higher than it should be before rounding and consequently, $\Delta Balance_{asset_out}$ can be higher than it should be before rounding which is not favourable to the pool.

4. The imbalance of the hub asset is indicated by
   $Imbalance_{LRNA}' = Imbalance_{LRNA} - \Delta Imbalance$ where
   $Fee_{protocol} = \lfloor ProtocolFee * \Delta HubReserve_{asset_in} \rfloor = \lfloor ProtocolFee * (\overline{\Delta HubReserve_{asset_in}} + \delta_{\Delta HubReserve_{asset_in}}) \rfloor$
   , $\Delta Imbalance = min(Fee_{protocol}, Imbalance_{LRNA}.value)$
   Thus $\delta_{\Delta Imbalance} \in (- ProtocolFee - 1, 0]$ and $\delta_{Imbalance} \in [0, 1 + ProtocolFee)$.
5. The amount of LRNA to be burned is
   $\Delta Balance_{LRNA} = \Delta HubReserve_{asset_out} + \Delta HubReserve_{HDX} - \Delta HubReserve_{asset_in}$ where
   $\Delta HubReserve_{HDX} = max(0, Fee_{protocol} - \Delta Imbalance)$.
   $\delta_{\Delta HubReserve_{HDX}} \in (- ProtocolFee - 1, 0]$, thus $\delta_{\Delta Balance_{LRNA}} \in (- ProtocolFee - 1, 1)$.

Operation: Sell LRNA for asset_out

Specification:

1. $SELL \in tradable_{LRNA}$ and $BUY \in tradable_{asset_out}$, otherwise a `NotAllowed` error is returned.
2. The amount of asset_out should be no less than min_buy_amount, providing price protection for traders.
3. The hub asset will only be possible to be burned but never minted in swap events.
4. The implementation should be consistent with the design specification: [Swap LRNA Spec](#).

Implementation:

Transition:

sell(origin, asset_in, asset_out, amount, min_buy_amount)

Settings:

$$\Delta Balance_{asset_out_1} = \lfloor \frac{Balance_{asset_out} * amount}{HubReserve_{asset_out} + amount} \rfloor$$

$$\Delta Balance_{asset_out} = \lfloor (1 - AssetFee) * \lfloor \Delta Balance_{asset_out_1} \rfloor \rfloor$$

$$Imbalance'_{LRNA} = Balance_{LRNA} + amount - \lfloor \frac{Balance_{asset_out} - \Delta Balance_{asset_out}}{Balance_{asset_out}} * \frac{HubReserve_{asset_out}}{HubReserve_{asset_out} + amount} * (Balance_{LRNA} + amount) \rfloor + \lfloor \frac{Balance_{asset_out} - \Delta Balance_{asset_out}}{Balance_{asset_out}} * \frac{HubReserve_{asset_out}}{HubReserve_{asset_out} + amount} * \frac{Balance_{LRNA} + amount}{Balance_{LRNA}} * Imbalance_{LRNA} \rfloor$$

$$\Delta imbalance = Imbalance'_{LRNA} - Imbalance_{LRNA}$$

Pre conditions:

1. $asset_in \neq asset_out$
2. $amount \geq Limit_{minimum_trading}$
3. $Balance^{who}_{LRNA} - amount \geq MultiCurrency::Accounts(who)(LRNA).frozen$
4. $asset_in = LRNA$
5. $SELL \in tradable_{LRNA} \wedge BUY \in tradable_{asset_out}$
6. $asset_out \in Assets$
7. $Imbalance_{LRNA}.negative$
8. $Balance_{asset_out} - \Delta Balance_{asset_out} \geq MultiCurrency::Accounts(Omnipool)(asset_out).frozen$
9. $MultiCurrency::Accounts(who)(asset_out).total() + \Delta Balance_{asset_out} >=$ $ExistentialDeposit_{asset_out} \,||\, who \in DustRemovalWhitelist$

Post state:

<u>Accounts:</u>

$Balance^{who}_{LRNA} \ -= \ amount$

$Balance_{LRNA} \ += \ amount$

$Balance_{asset_out} \ -= \ \Delta Balance_{asset_out}$

$Balance^{who}_{asset_out} \ += \ \Delta Balance_{asset_out}$

Omnipool states:

$Imbalance_{LRNA} \ -= \ \Delta imbalance$

$Assets(asset_out).hub_reserve \ += \ amount$

Post condition:

1. $\Delta Balance_{asset_out} \ \geq \ min_buy_amount$
2. $Imbalance_{LRNA}'.negative$

Conclusion:

Is the specification respected in the implementation?

Specifications 1, 2 and 4 are satisfied. There is no LRNA burned or minted, thus specification 3 also holds.

Invariants Checking

1. Invariant 1, the Swap_Invariant is guaranteed ignoring rounding and swap fees, for the subpool $asset_out/LRNA$.

   Proof:

   $Balance_{asset_out}' * HubReserve_{asset_out}'$

   $$= (Balance_{asset_out} - \frac{Balance_{asset_out} * amount}{HubReserve_{asset_out} + amount}) * (HubReserve_{asset_out} + amount)$$

   $$= Balance_{asset_out} * HubReserve_{asset_out}$$

2. Invariants 2, 3, and 4 i.e., Non_Positive_Imbalance, HubAsset_Total, and HubAsset_Accouting are guaranteed. The proof is trivial.

Rounding Error Analysis

1. The amount of $asset_out$ paid to the trader is indicated by $\Delta Balance_{asset_out} \ = \lfloor (1 - AssetFee) * \lfloor \Delta Balance_{asset_out}_1 \rfloor \rfloor$ where

   $\Delta Balance_{asset_out}_1 \ = \ \lfloor \frac{Balance_{asset_out} * amount}{HubReserve_{asset_out} + amount} \rfloor.$

Thus $\delta_{\Delta Balance_{asset_out} - 1} \in (-\ 1, 0]$, and $\delta_{\Delta Balance_{asset_out}} \in (-\ (2\ -\ AssetFee),\ 0)$.

2. The imbalance of the hub asset is indicated by $Imbalance'_{LRNA}$:

   $\delta_{Imbalance_{LRNA}} \in (-\ 2,\ 0]$

Operation: Buy asset_out using asset_in, none is LRNA

Specification:

1. Given y, the requested amount of $asset_out$, the quantity of $asset_in$ and $asset_out$ in the Omnipool, T_1 and T_2, and the quantity of LRNA tokens in each subpool, L_1 and L_2, "buy" calculates the amount of $asset_in$ that needs to be provided.
2. The hub asset is traded separately. The hub asset can only be sold to Omnipool and cannot be bought from Omnipool at the moment, i.e., LRNA can not be the $asset_out$ for either buy or sell.
3. Asset's tradable states must contain SELL flag for asset_in and BUY flag for asset_out, otherwise `NotAllowed` error is returned.
4. The implementation should be consistent with the design specification: Swap Spec.
5. The hub asset can only be burned. It is never minted.

Implementation:

Transition:

buy(origin, asset_out, asset_in, amount, max_sell_amount)

Settings:

$who\ =\ ensure_signed(origin)$

$\Delta Balance_{asset_out} = amount$

$Balance_{asset_out}_wo_fee = \lfloor (1 - AssetFee) * Balance_{asset_out} \rfloor$

$\Delta HubReserve_{asset_out} = \lfloor \frac{HubReserve_{asset_out} * amount}{Balance_{asset_out}_wo_fee - amount} \rfloor + 1$

$\Delta HubReserve_{asset_in} = \lfloor \frac{\Delta HubReserve_{asset_out}}{1 - ProtocolFee} \rfloor$

$\Delta Balance_{asset_in} = \lfloor \frac{Balance_{asset_in} * \Delta HubReserve_{asset_in}}{HubReserve_{asset_in} - \Delta HubReserve_{asset_in}} \rfloor + 1$

$Fee_{protocol} = \lfloor ProtocolFee\ *\ \Delta HubReserve_{asset_in} \rfloor$

$\Delta imbalance = min(Fee_{protocol}, Imbalance_{LRNA}.value)$

$\Delta HubReserve_{HDX} = Fee_{protocol} - \Delta imbalance$

$\Delta Balance_{LRNA} = \Delta HubReserve_{asset_out} + \Delta HubReserve_{HDX} - \Delta HubReserve_{asset_in}$

Pre conditions:

1. $asset_in \neq asset_out$
2. $amount\ \geq\ Limit_{minimum_trading}$
3. $asset_in\ \neq\ LRNA\ \wedge\ asset_out\ \neq\ LRNA$
4. $SELL\ \in tradable_{asset_in}\ \wedge\ BUY\ \in tradable_{asset_out}$
5. $asset_in \in Assets \wedge asset_out \in Assets \wedge HDX \in Assets$

6. $Balance_{asset_out} \geq amount$
7. $\Delta HubReserve_{asset_in} < Balance_{LRNA}$
8. $\Delta Balance_{LRNA} \leq 0$
9. $Balance^{who}_{asset_in} - \Delta Balance_{asset_in} \geq MultiCurrency::Accounts(who)(asset_in).frozen$
10. $MultiCurrency::Accounts(who)(asset_out).total + amount \geq ExistentialDeposit_{asset_out} \lor$
    $who \in DustRemovalWhiteList$
11. $Balance_{asset_out} - amount \geq MultiCurrency::Accounts(Omnipool)(asset_out).frozen$
12. $Balance_{LRNA} + \Delta Balance_{LRNA} \geq MultiCurrency::Accounts(Omnipool)(LRNA).frozen$ when $\Delta Balance_{LRNA} < 0$
13. $HubReserve_{asset_in} \geq \Delta HubReserve_{asset_in}$

Post state:

<u>Accounts:</u>

$Balance^{who}_{asset_in} -= \Delta Balance_{asset_in}$

$Balance_{asset_in} += \Delta Balance_{asset_in}$

$Balance_{asset_out} -= amount$

$Balance^{who}_{asset_out} += amount$

$Balance_{LRNA} += \Delta Balance_{LRNA}$

$TotalIssuance_{LRNA} += \Delta Balance_{LRNA}$

<u>Omnipool states:</u>

$Imbalance_{LRNA} -= \Delta imbalance$

$Assets[HdxAssetId].hub_reserve += [\Delta HubReserve_{HDX} > 0] * \Delta HubReserve_{HDX}$

$Assets(asset_in).hub_reserve -= \Delta HubReserve_{asset_in}$

$Assets(asset_out).hub_reserve += \Delta HubReserve_{asset_out}$

Post condition:

1. $\Delta Balance_{asset_in} \leq max_sell_amout$
2. $Imbalance_{LRNA}'.negative$

Conclusion:

Is the specification respected in the implementation?

1. Specifications 1-4 are satisfied.
2. The code checks that specification 5 is satisfied.

Invariants Checking

1. Invariant 1, the Swap_Invariant is guaranteed up to rounding, for subpools, $asset_in/LRNA$ and $asset_out/LRNA$, if fees are not considered. Proof is sketched as follows.
    a. For the $asset_in/LRNA$ subpool, ignoring rounding and fees, we have
    $Balance_{asset_in}' * HubReserve_{asset_in}'$
    $= (Balance_{asset_in} + \Delta Balance_{asset_in}) * (HubReserve_{asset_in} - \Delta HubReserve_{asset_in})$
    $= (Balance_{asset_in} + \frac{Balance_{asset_in} * \Delta HubReserve_{asset_in}}{HubReserve_{asset_in} - \Delta HubReserve_{asset_in}}) * (HubReserve_{asset_in} - \Delta HubReserve_{asset_in})$
    $= Balance_{asset_in} * HubReserve_{asset_in}$.
    b. For the $asset_out/LRNA$ subpool, ignoring rounding, we have
    $Balance_{asset_out}' * HubReserve_{asset_out}'$
    $= (Balance_{asset_out} - \Delta Balance_{asset_out}) * (HubReserve_{asset_out} + \Delta HubReserve_{asset_out})$
    $= (Balance_{asset_out} - amount) * (HubReserve_{asset_out} + \frac{HubReserve_{asset_out} * amount}{Balance_{asset_out} - amount})$
    $= Balance_{asset_out} * HubReserve_{asset_out}$.
2. Invariants 2 and 3, i.e., Non_Positive_Imbalance and HubAsset_Total, are guaranteed. The proof is trivial.
3. Invariant 4, HubAsset_Accouting is satisfied. The proof is sketched as follows.
    Proof: When $Fee_{protocol} < Imbalance_{LRNA}.value$,
    $\Delta imbalance = Fee_{protocol}, \Delta HubReserve_{HDX} = 0$, thus
    $Balance_{LRNA}' = Balance_{LRNA} + \Delta Balance_{LRNA} = Balance_{LRNA} + \Delta HubReserve_{asset_out} - \Delta HubReserve_{asset_in}$

    $= \sum_{t \in Assets} HubReserve_t + \Delta HubReserve_{asset_out} + \Delta HubReserve_{HDX} - \Delta HubReserve_{asset_in}$

    $= \sum_{t \in Assets} HubReserve_t'$

    When $Fee_{protocol} \geq Imbalance_{LRNA}.value$,
    $\Delta imbalance = Imbalance_{LRNA}, \Delta HubReserve_{HDX} = Fee_{protocol} - Imbalance_{LRNA}$, the invariant also holds.

Rounding Error Analysis

1. The hub asset in the $asset_out/LRNA$ subpool is indicated by $HubReserve_{asset_out} \mathrel{+}= \Delta HubReserve_{asset_out}$, where $\Delta HubReserve_{asset_out} = \lfloor \frac{HubReserve_{asset_out} * amount}{Balance_{asset_out}_wo_fee - amount} \rfloor + 1$
   $Balance_{asset_out}_wo_fee = \lfloor (1 - AssetFee) * Balance_{asset_out} \rfloor$.
   We have, $\delta_{Balance_{asset_out}_wo_fee} \in (-1, 0]$,
   $\delta_{\Delta HubReserve_{asset_out}} \in (0, \frac{\overline{\Delta HubReserve_{asset_out}}}{(Balance_{asset_out}_wo_fee - amount)} + 1)$ and
   $\delta_{HubReserve_{asset_out}} = \delta_{\Delta HubReserve_{asset_out}}$.

2. The hub asset in the $asset_in/LRNA$ subpool is indicated by
   $HubReserve_{asset_in} \mathrel{-}= \Delta HubReserve_{asset_in}$, where $\Delta HubReserve_{asset_in} = \lfloor \frac{\Delta HubReserve_{asset_out}}{1 - ProtocolFee} \rfloor$.
   $\delta_{\Delta HubReserve_{asset_in}} \in (-1, \frac{\delta^{upper_bound}_{\Delta HubReserve_{asset_out}}}{1 - ProtocolFee})$ and $\delta_{HubReserve_{asset_in}} = -\delta_{\Delta HubReserve_{asset_in}}$.
3. The amount of $asset_in$ needed to be paid by the trader is indicated by
   $\Delta Balance_{asset_in} = \lfloor \frac{Balance_{asset_in} * \Delta HubReserve_{asset_in}}{HubReserve_{asset_in} - \Delta HubReserve_{asset_in}} \rfloor + 1$. We can compute that
   $\delta_{\Delta Balance_{asset_in}} \in (-(1 + ProtocolFee), ProtocolFee * (\delta^{upper_bound}_{\Delta HubReserve_{asset_in}}))$.
   This rounding error may not be favorable to the pool.
4. The imbalance of the hub asset is indicated by $Imbalance_{LRNA} \mathrel{-}= \Delta Imbalance$, where
   $Fee_{protocol} = \lfloor ProtocolFee * \Delta HubReserve_{asset_in} \rfloor$ and
   $\Delta imbalance = min(Fee_{protocol}, Imbalance_{LRNA}.value)$.
   $\delta_{Fee_{protocol}} \in (ProtocolFee * \delta^{lower_bound}_{\Delta HubReserve_{asset_in}} - 1, ProtocolFee * \delta^{upper_bound}_{\Delta HubReserve_{asset_in}})$
   $= (-ProtocolFee - 1, \frac{ProtocolFee}{1 - ProtocolFee} * \delta^{upper_bound}_{\Delta HubReserve_{asset_out}})$
   Thus:
   When $Fee_{protocol} < Imbalance_{LRNA}.value$ and $\overline{Fee_{protocol}} < Imbalance_{LRNA}.value$ we have
   $\delta_{Imbalance_{LRNA}'} = -\delta_{Fee_{protocol}}$.
   When $Fee_{protocol} > Imbalance_{LRNA}.value$ and $\overline{Fee_{protocol}} > Imbalance_{LRNA}.value$ we have
   $\delta_{Imbalance_{LRNA}'} = 0$.
   Otherwise, $\delta_{Imbalance_{LRNA}'}$ is somewhere between the two.
5. The amount of LRNA to be burned is
   $\Delta Balance_{LRNA} = \Delta HubReserve_{asset_out} + \Delta HubReserve_{HDX} - \Delta HubReserve_{asset_in}$.
   We have $\Delta HubReserve_{HDX} = Fee_{protocol} - \Delta imbalance$, which means that
   $\delta_{\Delta Balance_{LRNA}} \approx \delta_{Fee_{protocol}}$. This gives $\delta_{\Delta Balance_{LRNA}} = \delta_{\Delta Balance_{LRNA}} + \delta_{Fee_{protocol}} - \delta_{\Delta HubReserve_{asset_in}}$.

Operation: Buy asset_out using LRNA

Specification:

1. $SELL \in tradable_{LRNA}$ and $BUY \in tradable_{asset_out}$, otherwise a `NotAllowed` error is returned.
2. The hub asset is traded separately. The hub asset can only be sold to Omnipool and cannot be bought from Omnipool at the moment, i.e., LRNA can not be the *asset_out* for either buy or sell.
3. The implementation should be consistent with the design specification: Swap LRNA Spec.
4. The hub asset will only be burned. It will never be minted.

Implementation:

Transition:

buy(origin, asset_out, asset_in, amount, max_sell_amount)

Settings:

$$\Delta Balance_{asset_out} = amount$$

$$hub_denominator = \lfloor (1 - AssetFee) * Balance_{asset_out} \rfloor - amount$$

$$\Delta Balance_{LRNA} = \lfloor \frac{HubReserve_{asset_out} * amount}{hub_denominator} \rfloor + 1$$

$$Imbalance'_{LRNA} = Balance_{LRNA} + \Delta Balance_{LRNA}$$
$$- \lfloor \frac{Balance_{asset_out} - \Delta Balance_{asset_out}}{Balance_{asset_out}} * \frac{HubReserve_{asset_out}}{HubReserve_{asset_out} + \Delta Balance_{LRNA}} * (Balance_{LRNA} + \Delta Balance_{LRNA}) \rfloor$$
$$+ \lfloor \frac{Balance_{asset_out} - \Delta Balance_{asset_out}}{Balance_{asset_out}} * \frac{HubReserve_{asset_out}}{HubReserve_{asset_out} + \Delta Balance_{LRNA}} * \frac{Balance_{LRNA} + \Delta Balance_{LRNA}}{Balance_{LRNA}} * Imbalance_{LRNA} \rfloor$$

$$\Delta imbalance = Imbalance'_{LRNA} - Imbalance_{LRNA}$$

Pre conditions:

1. $asset_in \neq asset_out$
2. $amount \geq Limit_{minimum_trading}$
3. $asset_in = LRNA \wedge asset_out \neq LRNA$
4. $asset_out \in Assets$
5. $SELL \in tradable_{LRNA} \wedge BUY \in tradable_{asset_out}$
6. $MultiCurrency::Accounts(who)(asset_out).total + amount \geq ExistentialDeposit_{asset_out} \vee$
   $who \in DustRemovalWhitelist$
7. $Balance_{asset_out} - amount \geq MultiCurrency::Accounts(Omnipool)(asset_out).frozen$

Post state:

<u>Accounts:</u>

$Balance^{who}_{LRNA} -= \Delta HubReserve_{asset_out}$

$Balance_{LRNA} += \Delta HubReserve_{asset_out}$

$Balance_{asset_out} -= amount$

$Balance^{who}_{asset_out} += amount$

<u>Omnipool states:</u>

$Imbalance_{LRNA} -= \Delta Imbalance$

$Assets(asset_out).hub_reserve += \Delta HubReserve_{aseet_out}$

Post condition:

1. $\Delta Balance_{LRNA} = \Delta HubReserve_{asset_out} \leq max_sell_amount$
2. $Imbalance'_{LRNA}.negative$

Conclusion:

Is the specification respected in the implementation?

Specifications 1-3 are satisfied. There is no LRNA burned or minted, thus specification 4 also holds.

Invariants Checking

1. Invariant 1, the Swap_Invariant is guaranteed ignoring rounding and swap fees, for the subpool $asset_out/LRNA$.

   Proof:

   $Balance_{asset_out}' * HubReserve_{asset_out}'$

   $= (Balance_{asset_out} - amount) * (HubReserve_{asset_out} + \frac{HubReserve_{asset_out} * amount}{Balance_{asset_out} - amount})$

   $= Balance_{asset_out} * HubReserve_{asset_out}$

2. Invariants 2, 3 and 4, 5, 6 i.e., Non_Positive_Imbalance, HubAsset_Total and HubAsset_Accouting are guaranteed. Proof is trivial.

Rounding Error Analysis

1. The amount of hub asset reserved in the $asset_out/LRNA$ subpool after this operation is indicated by $Assets(asset_out).hub_reserve += \Delta Balance_{LRNA}$, where

   $\Delta Balance_{LRNA} = \lfloor \frac{HubReserve_{asset_out} * amount}{hub_denominator} \rfloor + 1$ and

   $hub_denominator = \lfloor (1 - AssetFee) * Balance_{asset_out} \rfloor - amount.$

We have $\delta_{hub_denominator} \in (-1, 0]$ and $\delta_{\Delta Balance_{LRNA}} \in (0, \frac{\overline{\Delta Balance_{LRNA}}}{(\overline{hub_denominator}-1)} + 1)$.

When the *amount* of *asset_out* that is being swapped is close to the limit, i.e., close to $(1 - AssetFee) * Balance_{asset_out}$, this error could be very large.

2. The imbalance of the hub asset is indicated by

$$Imbalance'_{LRNA} = Balance_{LRNA} + \Delta Balance_{LRNA} - \lfloor \frac{Balance'_{asset_out}}{Balance_{asset_out}} * \frac{HubReserve_{asset_out}}{HubReserve'_{asset_out}} * Balance'_{LRNA} \rfloor + \lfloor \frac{Balance'_{asset_out}}{Balance_{asset_out}} * \frac{HubReserve_{asset_out}}{HubReserve'_{asset_out}} * \frac{Balance'_{LRNA}}{Balance_{LRNA}} * Imbalance_{LRNA} \rfloor.$$

Calculating the exact rounding difference for *hub_imbalance* might be less meaningful here. Its minimum is 0, while its maximum is

$$max(\delta_{\Delta Balance_{LRNA}}) * (1 + \frac{Balance'_{asset_out} * HubReserve_{asset_out} * (Balance_{LRNA} - HubReserve'_{asset_out}) * (1 - \frac{Imbalance_{LRNA}}{q})}{Balance_{asset_out} * HubReserve'_{asset_out} * (HubReserve_{asset_out} + max(\delta_{\Delta Balance_{LRNA}}))})$$

Note that $\frac{HubReserve'_{asset_out}}{Balance'_{asset_out}}$ is the final price, while $\frac{HubReserve_{asset_out}}{Balance_{asset_out}}$ is the initial price for *asset_out*. When selling LRNA, the price increases, so $\frac{Balance'_{asset_out} * HubReserve_{asset_out}}{Balance_{asset_out} * HubReserve'_{asset_out}}$ is less than 1. This means that the error is lower than $max(\delta_{\Delta Balance_{LRNA}}) * (1 + \frac{(Balance_{LRNA} - HubReserve'_{asset_out}) * (1 - \frac{Imbalance_{LRNA}}{q})}{(HubReserve_{asset_out} + max(\delta_{\Delta Balance_{LRNA}}))})$

Usually $Imbalance_{LRNA} << q$. This means that the above can be approximated by

$max(\delta_{\Delta Balance_{LRNA}}) * (1 + \frac{Balance_{LRNA} - HubReserve_{asset_out}}{HubReserve_{asset_out} + max(\delta_{\Delta Balance_{LRNA}})})$.

Also, $Balance_{LRNA}$ is probably a few times larger than $HubReserve_{asset_out}$, but, if the current subpool is fairly small compared to the other ones (e.g. when a new token is added), this difference can be significant, and $\frac{Balance_{LRNA} - HubReserve_{asset_out}}{HubReserve_{asset_out} + max(\delta_{\Delta Balance_{LRNA}})}$ can be approximated by $\frac{Balance_{LRNA}}{HubReserve_{asset_out}} - 1$. This means that the maximum above is approximated by

$$max(\delta_{\Delta Balance_{LRNA}}) * \frac{Balance_{LRNA}}{HubReserve_{asset_out}} = (\frac{HubReserve_{asset_out} * amount}{\overline{hub_denominator} * (\overline{hub_denominator} - 1)} + 1) * \frac{Balance_{LRNA}}{HubReserve_{asset_out}}$$

$$= \frac{\overline{\Delta Balance_{LRNA}}}{(\overline{hub_denominator} - 1)} + \frac{Balance_{LRNA}}{HubReserve_{asset_out}}.$$

This is usually dominated by the $\frac{Balance_{LRNA}}{HubReserve_{asset_out}}$ term, but can be much larger when $\overline{hub_denominator}$ is small (say, 2).

Issues

1. Redundant computation when calculate remove liquidity state change

[Severity: Informative | Difficulty: - | Category: Code Improvement]

Issue: The following two code snippets calculate the current pool price *current_price* of the asset and the changes to the hub reserve *delta_hub_reserve_hp* of this asset after the liquidity removal operation.

```
let current_price = asset_state.price()?;

let delta_hub_reserve_hp = delta_reserve_hp
  .checked_mul(current_hub_reserve_hp)
  .and_then(|v| v.checked_div(current_reserve_hp))?;
```

The implementation of the *delta_hub_reserve_hp* could reuse the computation result of *current_price*. However, it seems the rust type U256 does not provide rounded multiplication to a rational number.

Effects: Describe the consequences if not fixed.

Recommendation: Recommended way to fix it. If there is no straightforward solution, leave it for discussion with the team.

Status: No fix, leave as it is. Comments from @Martin: The reason is that we usually want to multiply first and then divide. If we use an already calculated price, the division is done first and then it is multiplied..

2. Rounding error exploit when removing liquidity

[Severity: Low | Difficulty: High | Category: Security]

Issue: The detailed reasoning is referred to the Section *Rounding Error Analysis #4* of the operation of *Remove Liquidity*.

The implementation of hub_transferred based on two rounded intermediate variables p_x_r and div1 would result in a rounding error that can be exploited, though the condition for the attack to be profitable is hard to reach.

The conditions to a profitable attack require:

1. The amount of shares to be removed should be close to the total shares of the asset;
2. The attacker (who is the liquidity provider) manipulates the price of the asset such that it raises significantly (for example, 100 times of the price in the position). The attacker would have to provide a large fund and pay a large amount of fees to fulfill this manipulation while outrunning the arbitrator.

Effects: This rounding could result in the trader being paid slightly more than he/she should be and with a low possibility and high cost of exploitation based on price manipulation.

Recommendation: It is recommended to always round towards one direction and favourable to the pool. A recommended fix is to calculate the intermediate results as a fixed point value, and round the hub_transferred at the last step only. For example, in rust alike pseudo-code:

```
let hub_transferred = if current_price > position_price {
      let price_sum = current_price.checked_add(&position_price)?;
      let price_sub = current_price.checked_sub(&position_price)?;
      let div1 = price_sub.checked_div(&price_sum);

     let div2 =
to_balance!(current_hub_reserve_hp.checked_mul(delta_shares_hp)?.checked_di
v(current_shares_hp)?).ok()?;
     div1.checked_mul_int(div2);
}...
```

In such a way, the rounding error would be bound in $(-\ div1,\ 0]$, and $div1$ would always be less than 1, favorable to the pool.
Please be aware that the above recommended implementation is not tested.

Status: Issue submitted as #64. The proposed fix, different from the proposed fix in PR #70 rounded up the value of p_x_y that solves the problem of diverging rounding results, such that δ_{div1} and $\delta_{\Delta Balance^{who}_{LRNA}}$ both have an upbound less than 0. The threat of the exploit by price manipulation is removed consequently.

3. Consistency: $Imbalance_{LRNA}.negative$ is not a precondition in most exchange types.

[Severity: Informative | Difficulty: - | Category: -]

Issue: All endpoints involving liquidity check $Imbalance_{LRNA}.negative$ both as a precondition and as a postcondition. However, when doing exchanges it is checked only as a postcondition, the only exception being when selling LRNA.

Effects: No user-visible effects, it's just that the $Imbalance_{LRNA}.negative$ invariant is checked in a non-consistent way.

Recommendation: Check the invariant in a consistent way, unless there is a reason to do otherwise. As an example, the invariant could always be checked as a precondition.

Status: Issue Submitted (link), fixed in PR #70 .

4. Possibly large error in the imbalance when buying with LRNA

[Severity: Low | Difficulty: Easy | Category: Functional Correctness]

Issue: The lower bound for the imbalance computation error is negative, and its absolute value is roughly $\frac{\overline{\Delta Balance_{LRNA}}}{(hub_denominator - 1)} + \frac{Balance_{LRNA}}{HubReserve_{asset_out}}$. In most cases, the first term is almost 0, and the second one is small. However, in principle, $\overline{hub_denominator}$ can be very small (say, it can be 2), so the entire error can be very large.

Effects: The imbalance will be smaller (larger in absolute value), which, over time, will increase the value of LRNA, which is probably not an issue. However, at the same time, it will decrease profits from HDX fees.

Status: Issue Submitted (link), fixed in PR#70 to round up the fees collected such that the result would be favourable to the pool.

Appendix: Issue Severity/Difficulty Classification

Our issues ranking system is based on two axes, severity and difficulty. Severity covers "how bad would it be if someone exploited this", and is ranked Informative, Low, Medium, and High. The difficulty is "how hard is it for someone to exploit this", and is ranked Low, Medium, and High.

This document is guidance for security ratings and is constantly changing. Runtime Verification maintains full discretion about the classification of issues. Furthermore, the lead auditor reserves the right to change severity or difficulty ratings as needed for each situation.

Severity Ranking

Severity refers to how bad it is if this issue is exploited. This means that the effects of the exploit affect the severity, but who can do the exploit does not.
If a given attack seems to fit multiple criteria here, use the most severe classification.

High Severity

- Permanent deadlock of some or all protocol operations.
- Loss of any non-trivial amount of user or protocol funds.
- Core protocol properties do not hold.
- Arbitrary minting of tokens by untrusted users.
- DOS attacks make the system (or any vital part of the system) unusable.

Medium Severity

- Sensible or desirable properties over the protocol do not hold, but no known attack vectors due to this ("looks risky" feeling).
- Non-responsive or non-functional systems are possible, but recovery of user funds can still be guaranteed.
- Temporary loss of user funds, guaranteed to be recoverable via an external algorithmic mechanism like a treasury.
- Loss of small amounts of user funds (eg. bits of gas fees) that serve no protocol purpose.
- Griefing attacks which make the system less pleasant to interact with, are potentially used to promote a competitor.
- System security relies on assumptions about externalities like "valid user input" or "working monitoring server".
- Deployments are not verifiable, so phishing attacks may be possible.

Low Severity

- Slow processing of user transactions can lead to changed parameters at transaction execution time.
- Function reverts on some inputs that it could safely handle.
- Users receive fewer funds than expected in a purely mathematical model, but the bounds on this error are very small.

- Users are not protected from obviously bad choices (eg. trading into an asset with zero value).
- System accumulates dust (eg. due to rounding errors) that is unrecoverable.

Informative Severity

- Not following best coding practices. Examples include:
    - Missing input validation or state sanity checks,
    - Code duplication,
    - Bad code architecture,
    - Unmatched interfaces or bad use of external interfaces,
    - Use of outdated or known problematic toolchains (eg. bad compiler version),
    - Domain specific code smells (eg. not recycling storage slots on EVM).
- Gas optimizations.
- Non-intuitive or overly complicated behaviours (which may lead to users and/or auditors misunderstanding the code).
- Lack of documentation, or incorrect/inconsistent documentation.
- Known undesired behaviours when the security model or assumptions do not hold.

Difficulty Ranking

Difficulty refers to how hard it is to actually accomplish the exploit. The things that increase difficulty are how expensive the attack is, who can perform the attack, and how much control you need to accomplish the attack. Note that when analyzing the expense difficulty of an attack, you must take into account flash loans.
If an attack fits multiple categories here, because of factors X which makes it severity S1 and Y which makes severity S2, then you need to decide:

- Are both X and Y necessary to make the attack happen, then use the higher difficulty.
- If only one of X and Y is necessary, then use the lower difficulty.

High Difficulty

- Only trusted authorized users can perform the attack (eg. core devs).
- Performing the attack costs significantly more than how much you benefit (eg. it costs 10x to do the attack vs what is actually won).
- Performing the attack requires coordinating multiple transactions across different blocks and can be stopped if detected early enough.
- Performing the attack requires control of the network, to delay or censor given messages.
- Performing the attack requires convincing users to participate (eg. bribe the users).

Medium Difficulty

- Semi-authorized (or whitelisted) users can perform the attack (eg. "special" nodes, like validators, or staking operators).
- Performing the attack costs close to how much you benefit (eg. 0.5x - 2x).
- Performing the attack requires coordinating multiple transactions across different blocks, but cannot be stopped if detected early enough.

Low Difficulty

- Anyone who can create an account on the network can perform the attack.
- Performing the attack costs much less than how much you benefit (eg. < 0.5x).
- Performing the attack can happen within a single block or transaction (or transaction group).
- Performing the attack only requires access to a modest amount of capital and a flash-loan system.