SingularityNet Unbonded Staking Pool Technical Specification

May 13, 2022

Contents

1 Introduction

`Plutarch` is an eDSL in Haskell for writing on-chain scripts on Cardano. The intention is to write `SingularityNet`'s unbonded stake pool code in `Plutarch` for optimised script size & execution units, thus reducing transaction fees compared to `PlutusTx`.

Production off chain code will be written in `Cardano Transaction Library (CTL)`, an API to balance and submit transactions using browser-integrated wallets and `Ogmios`.

Definition 1.1 (`NFT State UTXO`)**.** A common design pattern in Plutus contract involves minting an NFT with a (unique) currency symbol which parametrises the validator (staking pool) in question. This is just a UTXO with the unit value for this currency symbol (and a fixed token name), for us, the datum injectively corresponds to an associated list (a `Map`). We'll refer to this UTXO as the **NFT State UTXO**.

Definition 1.2 (`Assoc. List UTXO`)**.** This spec. adapts an on-chain associated list. The UTXOs used for elements of the list will be called **Assoc. List UTXOs**. Its `CurrencySymbol` is given by `assocListCs` in Definition 2.3 with `TokenName = TokenName hashedPkh`, for `hashedPkh` in Definition 2.2. Its datum is `EntryDatum` in Definition 3.2.

We may sometimes interchange the word "Token" and "UTXO" although this can sometimes be confusing, so we will try to stick to "UTXO".

Aim: to write a unbonded staking pool for `SingularityNet`'s `AGIX` token.

2 Minting Policies

2.1 NFT State Minting Policy

This subsection is identical to that of bonded staking except for the hardcoded `TokenName`.

This is a standard (genuine) NFT minting policy parametrised over `TxOutRef` and optionally `TokenName` with the former used for uniqueness of the NFT `CurrencySymbol`. The `TokenName` can be globally hardcoded to "`UnbondedStaking`" for example.

On-chain minting conditions:

- The parameterised UTXO (`TxOutRef`) is part of the transaction inputs
- `txInfoMint` contains exactly one `AssetClass` of *unit* value. The `ownCurrencySymbol` with the hardcoded `TokenName` matches the aforementioned `AssetClass`
- The minting policy can optionally ensure the initial datum is `StateDatum Nothing True` (see Definition 3.1) although we cannot verify the validator address as this would cause mutual recursion

Definition 2.1 (`nftCs`)**.** this minting policy creates a genuine NFT (by virtue of the UTXO parameter) with a `CurrencySymbol` that cannot be reproduced. We will call this the **nftCs**.

2.2 Associated List Minting Policy

This subsection is identical to that of bonded staking.

The minting policy is parameterised by `nftCs` from Definition 2.1. The uniqueness of the `nftCs` means the currency symbol of this minting policy is unique, constant and injectively associated to the NFT State UTXO. On-chain minting conditions

- Check `txInfoSignatories` is a singleton list, the individual `PaymentPubKeyHash`'s underlying `BuiltinByteString` should be `blake2b_256` hashed to create a `TokenName` after rewrapping, see Definition 2.2. This of course implies the transaction should be signed by only this user

- `txInfoMint` contains exactly one `AssetClass` of ± 1 value (burning is allowed). The `ownCurrencySymbol` with `TokenName` above matches the aforementioned `AssetClass`
- The relevant inductive condition within Section 7 is met

This minting policy will mint value for UTXOs that form the associated list. Minting means adding to the associated list, whilst burning means removing from the associated list.

Note that this token can only be minted if the NFT State UTXO is initially at the validator address, so by induction, we have a unique and identifiable associated list for each staking pool.

Definition 2.2 (`hashedPkh`)**.** the pattern of taking a `PaymentPubKeyHash` and `blake2b_256` hashing the underlying `BuiltinByteString` to create another `BuiltinByteString` will be a common pattern. This will be used for creating `TokenNames` as above but also as `keys` for each `Entry`, see `key` in Definition 3.4. We will refer to the output `BuiltinByteString` as **hashedPkh**.

Definition 2.3 (`assocListCs`)**.** this minting policy has a `CurrencySymbol` that we define as **assocListCs**. This `CurrencySymbol` is unqiuely associated to the original NFT State UTXO and provides a `CurrencySymbol` for each associated list element (with different `TokenName` per user).

3 Datums

This section is different to that of bonded staking.

```
data UnbondedStakingDatum
   = StateDatum (Maybe BuiltinByteString) Bool
   | EntryDatum Entry
   | AssetDatum
```

Definition 3.1 (`StateDatum`)**.** This represents the staking pool/associated list and is datum for the NFT State UTXO. `Nothing` says the list is empty, `Just` the key (corresponding to `TokenName`) to the head of the associated list. We do not keep track of size here (compared to bonded) because the unbonded pool has no limit.

Recall the `AssetClass` for the NFT State comes from Subsection 2.1 with hardcoded `TokenName`. However, the `TokenName` inside `Maybe` is a hashed `PaymentPubKeyHash`, see Definition 2.2. The `Bool` represents the state of the pool, `True` means the pool is open, otherwise closed. This should be initiated to `True`.

Definition 3.2 (`EntryDatum`)**.** a wrapper over the `Entry` type found in Definition 3.4.

Definition 3.3 (`AssetDatum`)**.** is the datum for staked asset UTXOs at the script address, acting as dummy datum.

```
data Entry = Entry
  { key :: BuiltinByteString
  , deposited :: Natural
  , newDeposit :: Natural
  , rewards :: NatRatio
  , totalRewards :: Natural
  , totalDeposited :: Natural
  , open :: Bool
  , next :: Maybe BuiltinByteString
  }
```

Definition 3.4 (`Entry`)**.**

is the entry of the associated map in question. The UTXO with this datum should have `CurrencySymbol`, `assocListCs` from Definition 2.3 and `TokenName = TokenName key`.

Definition 3.5 (`key`)**.** is given by the `blake2b_256` hash of the `PaymentPubKeyHash` of the minting/burning user in question, see Definition 2.2.

Definition 3.6 (`deposited`)**.** how much a user has deposited so far for staking, we do not have a `staked` field like the bonded case, because of unlimited pool size.

Definition 3.7 (`newDeposit`)**.** tells the admin whether or not the user deposited in the recent cycle. 0 means they have not deposited in the recent cycle, `nonZero` means they have deposited `nonZero` in the recent cycle. This is needed for the admin to calculate rewards from the previous cycle if the user did not withdraw. This prevents confusion because the user could deposit in the new cycle but this should not count towards rewards in the previous cycle (from previous deposits) during the `rewards` update.

Definition 3.8 (`rewards`)**.** These are rewards accrued so far (across different cycles). This is updated by the admin during the admin deposit phase. This should be initiated to zero for new user deposits and unchanged for further deposits. This can be thought of as a lower bound for rewards at the start of a new cycle, basically all rewards accrued so far from *previous* cycles. Once the bonding period restarts for the upcoming cycle, the rewards will be greater or equal to this field. This is required as we do not have timestamped deposits. You can think of it like Markov chain (without the probablistic element) where we only care about the current state. See Subsection 6.1 for an example.

Definition 3.9 (`totalRewards`)**.** how much the admin deposited during the admin cycle for upcoming rewards in this withdrawal cycle.

Definition 3.10 (`totalDeposited`)**.** how much users have deposited (including their rewards so far) for this withdrawal/deposit cycle, allowing the user to proportionally work out their rewards.

Definition 3.11 (`open`)**.** the state of the pool, `True` means the pool is open and otherwise closed. Only the admin can change the state of the pool. This is initiated to `True` by induction.

Definition 3.12 (`next`)**.** a pointer to the `key` of the next UTXO in the associated list. `Nothing` indicates we are at the tail.

4 Redeemers

4.1 Minting Redeemers

This subsection is identical to that of bonded staking.

```
data MintingAction
   = Stake
   | Withdraw
```

Definition 4.1 (`Stake`)**.** redeemer used for staking (minting +1) and creating a UTXO for the associated list.

Definition 4.2 (`Withdraw`)**.** redeemer used for withdrawing (burning +1) and removing a UTXO from the associated list.

These redeemers should be used with the associated list minting policy in Subsection 2.2.

Since minting checks are forwarded to the validator, these redeemers could be deemed unnecessary, we can keep them for now to show intent.

4.2 Validator Redeemers

This subsection is different to that of bonded staking.

```
data UnbondedStakingAction
   = AdminAct Natural Natural -- (totalRewards totalDeposited) for updating UTXOs
   -- and depositing rewards
   | StakeAct Natural PaymentPubKeyHash
   | WithdrawAct PaymentPubKeyHash
   | CloseAct
```

Definition 4.3 (`AdminAct`)**.** redeemer for the **Admin** to deposit staking tokens to the validator and update rewards for each users Assoc. List UTXO. The first parameter signifies how much rewards will be added for the upcoming cycle. The second parameter signifies how much has been deposited (including rewards) in total. These respectively fill out `totalRewards` and `totalDeposited` in `EntryDatum`.

Definition 4.4 (`StakeAct`)**.** redeemer for staking tokens with `PaymentPubKeyHash` of `Natural` amount.

Definition 4.5 (`WithdrawAct`)**.** redeemer for withdrawing *all* staked tokens and rewards for a given user's `PaymentPubKeyHash`. This can be called when the pool is both open or closed.

Definition 4.6 (`CloseAct`)**.** redeemer for the admin to close the stake pool (for now, the users should then withdraw the tokens themselves). This is callable only during the admin phase to prevent contention.

5 Initialisation

This section is different to that of bonded staking.

5.1 Minting NFT State

Off-chain logic is required by the administrator/operator to initially mint an NFT with the following datum (see Definition 3.1):

```
StateDatum Nothing True
```

defining the on-chain associated list of validator/stake pool. The NFT `TokenName` can be hardcoded to "`UnbondedStaking`" or anything else, provided it's fixed for the codebase. The Boolean is default to `True` to show the pool is open.

On-chain `Maps` can theoretically increase the transaction size to no end. This technical spec will implement an adapted version of the on-chain associated list.

5.2 Validator Parameters

This subsection is different to that of bonded staking.

The currency symbol of the NFT then parametrises the validator as follows:

```
data UnbondedPoolParams = UnbondedPoolParams
  { upp'start :: POSIXTime -- absolute time
  , upp'userLength :: POSIXTime -- a time delta
  , upp'adminLength  :: POSIXTime -- a time delta
  , upp'bondingLength :: POSIXTime -- a time delta
  , upp'interestLength :: POSIXTime -- a time delta
  , upp'increments :: Natural
  , upp'interest :: NatRatio -- interest per increment
  , upp'minStake :: Natural
  , upp'maxStake :: Natural
  , upp'admin :: PaymentPubKeyHash
  , upp'unbondedAssetClass :: AssetClass
  , upp'nftCs :: CurrencySymbol -- this uniquely parameterises the validator
  , upp'assocListCs :: CurrencySymbol -- CurrencySymbol for on-chain associated list UTXOs
 }
```

The parameters are configurable by the administrator/operator at the start and fixed for the duration of the staking period.

These parameters are visualised in Equations 1, 2.

Definition 5.1 (`upp'start`)**.** the *absolute* `POSIXTime` the staking pool starts (the first cycle), i.e. when staking deposits can be taken from users for the first cycle.

Definition 5.2 (`upp'userLength`)**.** the *timedelta* for how long users can deposit (for the upcoming cycle) Should be thought of a positive number that can be added to some other starting point (as opposed to a fixed `POSIXTimeRange`). We could also use `Natural` or `Integer` for any subsequent "timedelta". Withdrawing during this phase will not give rewards unless they were already previously earned. In particular, a user can make a new deposit in this phase and withdraw at the same time, and their recent deposit should not accrue rewards.

Definition 5.3 (`upp'adminLength`)**.** the *timedelta* for the admin to update UTXOs on chain. In particular, the admin should update `rewards`, reset Assoc. List UTXO's `newDeposit` to zero, update the global `totalRewards` and `totalDeposited`.

Definition 5.4 (`upp'bondingLength`)**.** the *timedelta* for how long bonding can occur for a given cycle. The user can withdraw during this phase (unlike bonded) with rewards determined by the approximate (discretised) time they withdraw.

Definition 5.5 (`upp'interestLength`)**.** the *timedelta* for the discretised intervals for earning rewards. Because we cannot have a notion of continuous absolute time on chain, we need to discretise into intervals. We require `upp'interestLength` $\leq$ `upp'bondingLength` and `upp'bondingLength` is divisible by `upp'interestLength`. Rewards are then determined and "rounded to the nearest discretised interval".

Definition 5.6 (`upp'increments`)**.** $\frac{\texttt{upp'bondingLength}}{\texttt{upp'interestLength}}$ which should be exactly divisible. This is for on-chain convenience. If the user decides to withdraw during `upp'bondingLength`, the updated rewards can be calculated as `rewards`+(`deposited`+`rewards`)$*$(1+`upp'interest`)^k where $1 \leq k \leq$ `upp'increments`. Note that by this stage, `newDeposit` should be initially zero or reset to zero by the admin.

Definition 5.7 (`upp'interest`)**.** a positive (non-zero) ratio fixed decimal. This is fixed rate for one time increment in annual percentage yield.

Definition 5.8 (`upp'minStake`)**.** minimum amount required to stake by a wallet.

Definition 5.9 (`upp'maxStake`)**.** maximum amount possible to stake by a wallet.

Definition 5.10 (`upp'admin`)**.** the `PaymentPubKeyHash` of the administrator.

Definition 5.11 (`upp'unbondedAssetClass`)**.** the asset class of the token being staked, i.e. `AGIX`.

Definition 5.12 (`upp'nftCs`)**.** currency symbol of the NFT to identify the NFT state UTXO of the pool, see `StateDatum` in Definition 3.1.

Definition 5.13 (`upp'assocListCs`)**.** currency symbol of the associated list UTXOs, see `EntryDatum` in Definition 3.2.

5.3 Initiate Staking Pool

The following step should be taken to initiate the staking pool

- The minted NFT from Subsection 5.1 should be sent to the validator address determined by Subsection 5.2. This of course requires determining `upp'assocListCs` which is possible after obtaining `upp'nftCs`.

6 Unbonded Staking Schema

This section is different to that of bonded staking.

For reference, here is an example of just over one cycle/iteration

$$\texttt{upp'start} \xrightarrow[\texttt{upp'userLength}]{\overbrace{\textit{Deposit/Withdraw } 0}^{\texttt{No rewards}}} B \xrightarrow[\texttt{upp'adminLength}]{\overbrace{\textit{Admin Updates } 0}^{\texttt{Admin Deposits 0}}} C \xrightarrow[\texttt{upp'bondingLength}]{\overbrace{\textit{Bonding/Withdraw } 0}^{\texttt{Dynamic rewards}}} D \xrightarrow[\texttt{upp'userLength}]{\overbrace{\textit{Deposit/Withdraw } 1}^{\texttt{Previous rewards}}} E \xrightarrow[\texttt{upp'adminLength}]{\overbrace{\textit{Admin Updates } 1}^{\texttt{Admin Deposits 1}}} F \ldots \tag{1}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\texttt{cycle/iteration 0}}$$

Here's an explanation of the different windows:

- Users deposit during the first `upp'userLength`, these would count towards `newDeposit` so no rewards are available yet. The only way to get rewards here is to have timestamped UTXOs which would increase complexity. A user could also withdraw their deposited amount here (again with no rewards).
- Admin updates UTXOs during first `upp'adminLength`. Usually, the admin should increment `rewards` for the previous `upp'bondLength` (**none for the first cycle**), as well as `totalRewards` and `totalDeposited`. The latter two are determined off chain for the **upcoming** bonding cycle. `totalDeposited` should include rewards accrued so far so we get can proportionally get how much they own in total. The admin should also reset all `newDeposit`s to zero. This is needed because by the next `upp'userLength`, all new deposits need to be distinguishable from old deposits.
- Bonding begins and the user can withdraw whenever, the longer they wait, the more they are rewarded. Rewards are rounded to the nearest discretised interval.
- For the next `upp'userLength` cycle, the user reward is initially zero on the `EntryDatum`, but they can withdraw all the rewards from the previous bonding period. This is verfiable on chain using time checks. These rewards would be based on their `deposited` subtract any `newDeposit` from that round. Of course, they can make further deposits to `newDeposit` (and `deposited`) in this phase. `newDeposit` is required so they cannot get unearned rewards by depositing and withdrawing in this (same) phase.
- The cycle repeats with admin deposits, and UTXO updates as before. This `upp'adminLength` should update `rewards` for the first bonding period, so we can forget about the first cycle. This removes the requirement for timestamped deposits as they are adjusted after every cycle. This however means rewards only accrue during `upp'bondingLength` and *not* the other windows

A diagramatic explanation for `upp'bondingLength`:

$$\underbrace{x_o \xrightarrow[\texttt{upp'interestLength}]{} x_1 \xrightarrow[\texttt{upp'interestLength}]{} x_2 \xrightarrow[\texttt{upp'interestLength}]{} x_3 \xrightarrow[\texttt{upp'interestLength}]{} x_4 \to \dots}_{\texttt{upp'bondingLength}} \tag{2}$$

The discretised interval the user falls into with their contract (which can be verified on chain) will determine their rewards. This implies rewards are calculated approximately as we do not have a notion of verifiable (continuous) absolute time on chain. When a bonding stage is reached, the `rewards` field should be updated by the admin just prior (during the reccent admin stage), meaning (dynamic) rewards in this bonding cycle can be calculated from `deposited` and `rewards`.

If a user tries to withdraw during `upp'userLength`, they will have accrued rewards for previous `upp'bondingLength` periods and new deposits should not contribute. Of course, their first `upp'userLength` will have no rewards initially. This is reflected in the first `upp'adminLength` where rewards must be zero for everyone.

6.1 Example

Here is an example for one user:

1. `upp'userLength` 0
   - User mints an Assoc. List UTXO depositing 5 tokens. Make sure `open` is the same as neighbouring `open`s and `True`. `totalRewards` and `totalDeposited` can either be zero or taken from neighbours (we'll choose zero), these values are meaningless until the admin cleans them up to for global consistency. Ensure `rewards` is initiated to zero. Ensure `newDeposit` = `deposited` = 5
   - The user decides to make another deposit in the same period of +3. Now `newDeposit` = `deposited` = 8 with everything else unchanged. Note: withdrawals withdraw *everything* so we can ignore incremental withdrawals.

- If the user decides to withdraw, the validator should check they can withdraw `deposited` = 8 plus any rewards. Rewards would be calculated from interest acquired by `deposited - newDeposit + rewards` = 8 - 8 + 0 = 0. In other words, their recent deposits should have zero rewards. Also, they have zero rewards in the first place, so can only withdraw their deposit (8). This formula is a special case of the formula in `upp'userLength` 1 (see `withdrawRewards`) where rewards are always zero.

2. `upp'adminLength` 0

   - Note: admin does not update `rewards` as we are yet to complete a bonding cycle (`upp'bondingLength`). This is a special case for the **first** admin windows, subsequent admin windows will update rewards for the previous bonding window, but because there was no prior bonding, we have no rewards. This also happens to coincide with the `updatedRewards` formula in `upp'adminLength` 1, where `updatedRewards` $= 0 + (8 - 8 + 0) = 0$
   - After "updating rewards to zero", admin resets all Assoc. List UTXOs `newDeposit` to zero
   - After "updating rewards to zero", admin updates `totalDeposited` by summing all deposits and rewards off chain (using Assoc. List UTXOs)
   - After "updating rewards to zero", admin picks rewards for the upcoming cycle and updates all `totalRewards` accordingly.

3. `upp'bondingLength` 0

   - Bonding begins for this cycle, the user can withdraw whenever. Should they decide to withdraw, they will get the following rewards:
     `withdrawRewards` = `rewards` + $\frac{f}{(1+\texttt{upp'interest})^k}$
     where $1 \leq k \leq$ `upp'increments` for the appropriate k which should be on-chain verified. The user can submit their k off chain via the appropriate `POSIXTimeRange` and the validator will check this range has the correct difference (delta). At this point, `rewards` is still zero but we get interest from our deposit as intended. Furthermore, the user will withdraw their `deposited` of course. This is *not* an update on the datum (`EntryDatum`) but an actual withdrawal (burning the token). f is a user factor defined as $f := \frac{\texttt{deposited} - \cancel{\texttt{newDeposit}} + \texttt{rewards}}{\texttt{totalDeposited}} * \texttt{totalRewards}$. Note the inverse relation between k and time passed by, as we reach the end of the bonding period, k tends to 1 so we get almost the proportioned rewards. Also note that `newDeposit` **is zero here from the previous Admin cycle**
   - If they decide to not withdraw, they will reach the next deposit/withdrawal cycle.

4. `upp'userLength` 1

   - With `newDeposit`'s reset to zero. The user decides to make a further deposit +2 (they do not have to of course). So `newDeposit` = 2 and `deposited` = 10 (incremented).
   - If the user decides to withdraw here, the user will get (along with `deposited`)
     `withdrawRewards` = `rewards` + f, no k because they come from the just-completed bonding cycle. As before, this is *not* an update on the datum but an actual withdrawal (burning the token). We subtract `newDeposit` so they cannot get unearned rewards from depositing and withdrawing in the same cycle.
     f is a user factor defined as $f := \frac{\texttt{deposited} - \texttt{newDeposit} + \texttt{rewards}}{\texttt{totalDeposited}} * \texttt{totalRewards}$
   - We emphasise: no rewards accrue during `upp'userLength`s

5. `upp'adminLength` 1

   - Finally, if the user did not withdraw in the recent deposit/withdrawal cycle (with previously accrued rewards), the admin updates rewards of this user to
     `updatedRewards` = `rewards` + f with $f := \frac{\texttt{deposited} - \texttt{newDeposit} + \texttt{rewards}}{\texttt{totalDeposited}} * \texttt{totalRewards}$. This is an increment on the original `rewards` from recent bonding cycle. This update should be reflected in the datum
   - After updating rewards, admin resets all Assoc. List UTXOs `newDeposit` to zero

- After updating rewards, admin updates `totalDeposited` by summing all deposits (including rewards) off chain (using Assoc. List UTXOs) for the upcoming cycle
- After updating rewards, admin picks rewards for the upcoming cycle and updates `totalRewards` accordingly.

To prevent contention, the admin should only be able to close a pool during `upp'adminLength`. The admin should update all `open` fields to `False` (including the NFT State UTXO). At this point, the admin should update rewards so users can withdraw `deposited` + `rewards` (not needing to worry about the calculation). This is the same `rewards` calculation used in any other `upp'adminLength`. This seems trustless but the validator could verify this too. For now, we leave it for the user to withdraw their tokens (and not the admin to distribute but we can change this). Should the admin decide to close, `totalRewards` and `totalDeposited` fields do not matter, `rewards` should be updated so users can simply withdraw (`deposited` + `rewards`) in a straightforward manner.

6.2 User Stake

A user stake requires the Assoc. List UTXO from Subsection 2.2 to be minted (initial deposit) *or* the Assoc. List UTXO to be spent (further deposits).

6.2.1 On Chain

Overlapping on-chain conditions for any type of deposit (initial or subsequent) with `StakeAct` redeemer (see Definition 4.4), if any of these conditions do not hold, validation should fail,

- Signed by `PaymentPubKeyHash` from the redeemer. The list of signers should be a singleton list
- Transaction must occur after `upp'start` and in any `upp'userLength` period (abusing terminology slightly).
- Check the requested stake amount is positive (since `Natural` includes zero)
- Check the correct amount of the unbonded asset class (see Definition 5.11) is deposited to the validator address with `AssetDatum` as datum (see Definition 3.3)
- The user cannot withdraw any (staked) asset UTXOs
- Check the `newDeposit` (Definition 3.7) field has been initiated (first deposit of the cycle) or incremented (redeposits during the *same* cycle)
  1. In particular, initial new deposits for a given cycle are $0 \rightarrow$ `amount`
  2. Redeposits during the *same* cycle will have `updatedNewDeposit` $\rightarrow$ `prevNewDeposit + amount` for a given UTXO

For the field `deposited` (not `newDeposit`), further conditions required for an **Initial deposit (insertion):**

- Check the inductive conditions from Subsection 7.1, this means there may be different combinations of UTXOs minted, spent (and sent back to the validator) depending on the insertion type. At a high level, check a UTXO with `CurrencySymbol`, `upp'assocListCs` and `TokenName = TokenName hashedPkh` is minted, where `hashedPkh` is defined in 2.2. Check the corresponding `EntryDatum` datum from Definition 3.2 has been initiated accordingly, namely with *correct amount deposited, zero rewards etc.* This UTXO should be sent to the validator and inserted into the correct position of the on-chain associated list. If the NFT State UTXO is spent (as part of induction), it must be sent back to the validator also, potentially altered by head insertion
- Related to the previous, `txInfoMint` contains exactly one `AssetClass` of 1 value. The `CurrencySymbol` should be checked with 2.3 and `TokenName` with the (hashed) signer

- Check the requested stake amount is between the minimum and maximum allowed amount (inclusive of bounds say), see Definitions 5.8, 5.9. This enables users to stake multiple times (see **Further deposits** below)
- The `open` must be initiated as `True` (with neighbouring inductive UTXO(s) also `True`)

For the field `deposited` (not `newDeposit`), further conditions required for a **Further deposits (update):**

- Since induction was already used for the initial deposit, we just need to spend Assoc. List UTXO of the user looking to make further deposits.
- We can verify the correct UTXO is being spent by comparing `CurrencySymbol`, `upp'assocListCs` and `TokenName = TokenName hashedPkh`, where `hashedPkh` is defined in 2.2.
- The corresponding `EntryDatum` datum from Definition 3.2 has been *incremented* accordingly e.g. the `deposited` field. We should check the incremented amount is between the minimum and maximum allowed amount (inclusive of bounds say), see Definitions 5.8, 5.9. Its `key` can also be verified
- This Assoc. List UTXO should be sent to the validator otherwise unchanged.
- The `open` must be `True` when redepositing and stay `True`

Note we are checking increments for both `newDeposit` and `deposited` separately, the former is needed by the admin in Subsection 6.3. Although since `newDeposit` isn't type `Maybe`, we can just treat them the same.

6.2.2 Off Chain

Using the `stakeAct` redeemer:

- Signed by `PaymentPubKeyHash` from the redeemer and only this user
- Calculate the relevant `POSIXTimeRange` for the transaction to fall inside the `upp'userLength` for user staking. You may need to get the current `POSIXTime` off chain
- The user needs to query UTXOs at the validator address to find their position in the associated list. If they are not in the list, this is an **initial deposit** so the associated list minting policy from subsection 2.2 must be called with relevant inductive conditions. These essentially mirror the validator's conditions. Otherwise, if they are already in the list, this is a **further deposit**. For example, the `newDeposit` (Definition 3.7) field is initiated correctly (first deposit of the cycle) or incremented (redeposits in the same cycle) in the output (and sent to the validator) etc.
- Deposit the correct amount of the unbonded asset class (see Definition 5.11) to the validator address with `AssetDatum` as datum (see Definition 3.3)

For **Initial deposits**:

- Call the minting policy as above and send the relevant minted Assoc. List UTXO to validator address. The `EntryDatum` must be initiated as specified by the validator/inductive conditions (`deposited` = `amount`, `rewards` = 0 etc.)

For **Further deposits**:

- Spend the Assoc. List UTXO and increment the the `deposited` (and `newDeposited`) field (in `EntryDatum`) by the amount deposited and nothing else changed. See the induction conditions for more details

6.3 Admin Deposit (Bonding Period)

The admin should update Assoc. List UTXOs and deposit rewards for users. Use `adminAct newTotalRewards newTotalDeposited`. The admin is required to update the said UTXOs *during* the designated admin window (`upp'adminLength`) to:

1. Update the datum's `rewards`, `updatedRewards` = `rewards`$+f$ with $f := \frac{\texttt{deposited}-\texttt{newDeposit}+\texttt{rewards}}{\texttt{totalDeposited}} * \texttt{totalRewards}$. These are just rewards for the last complete bonding period (which would exclude any new deposits). Note this is zero for the **first** admin period since `rewards` = 0 and `deposited` = `newDeposit`.
2. *After* the above, update Assoc. List datum's `newDeposit` to zero
3. *After* `rewards` are updated, set the datum's `totalRewards` (Definition 3.9) to something *new* (admin's choice)
4. *After* `rewards` are updated, set the datum's `totalDeposited` (Definition 3.10) to how much has be deposited (including rewards) in total so far (this can be checked off chain).

Recall, we are (admin) depositing for the *upcoming* bonding cycle and updating rewards for the *most recently passed* bonding cycle. These increments mean we don't need an entire timestamp history of deposits, instead we have a Markov-like structure (without probability).

6.3.1 On chain

If any of these conditions do not hold, validation should fail,

- Signed by `upp'admin`, contained in the Subsection 5.2
- Transaction must occur after `upp'start` *and* in any `upp'adminLength` period (abusing terminology), which can easily be calculated on chain
- Multiple Assoc. List UTXOs (*not* NFT State Token) can be spent and updated. The `Entry(Datum)` type of *inputs* should be:

```
Entry
  { key = _
  , deposited = amount
  , newDeposit = newDeposit
  , rewards = rewards
  , totalRewards = _
  , totalDeposited = _
  , open = True -- must be open
  , next = _
  }
```

- These should be updated to:

```
Entry
  { key = _ -- unchanged
  , deposited = amount -- unchanged
  , newDeposit = 0 -- reset to zero
  , rewards = updatedRewards
  , totalRewards = newTotalRewards
  , totalDeposited = newTotalDeposited
  , open = True -- unchanged
  , next = _ -- unchanged
  }
```

- Here, `updatedRewards` (as recently defined) should be deposited to the validator for each user (with datum `AssetDatum` and rounding up to be safe). We should be very careful about rounding when admin depositing as user withdrawals in Subsection 6.4 will also round (down) and we do not want insufficient funds at the validator. Recall that `updatedRewards` are calculated before reseting `rewards` to zero (as we need to know how much was deposited recently).
- Since this is not entirely trustless, we could ensure `rewards` $\leq$ `updatedRewards` on chain, it should be impossible to reduce any already accrued rewards. We can of course verify `updatedRewards` using the redeemer parameters
- `totalRewards` and `totalDeposited` are updated via the admin act parameters.
- All updated Assoc. List UTXOs must be sent back to the validator, otherwise unchanged
- The admin cannot withdraw (staked) asset UTXOs

6.3.2 Off chain

Using the `adminAct newTotalRewards newTotalDeposited` redeemer:

- The admin should sign the transaction
- Calculate the relevant `POSIXTimeRange` for the transaction to fall inside the `upp'adminLength`. You may need to get the current `POSIXTime` off chain
- The admin should find all UTXOs at the validator with `assocListCs` as its `CurrencySymbol` to create the above transaction, updating the datums in the same way that the validator requires
- Batching of UTXOs can be done to save on the total number of transactions. Saving already dealt with UTXOs locally could help with not adding rewards twice
- The UTXOs carrying the staked asset with datum `AssetDatum` may be separated into unit (valued) UTXOs or at least a UTXO per user to help with contention issues during withdrawal. The first option will cost more due to minimum Ada requirements

6.4 User Withdraw

6.4.1 On chain

A user must withdraw *all* their staked tokens and rewards in one transaction with `WithdrawAct` redeemer (see Definition 4.5), if any of these conditions do not hold, validation should fail

- Signed by `PaymentPubKeyHash` from the redeemer. The list of signers should be a singleton list
- Transaction must occur after `upp'start` *and* in any `upp'userLength` *or* `upp'bondingLength` period (abusing terminology slightly), which can easily be calculated on chain. We can relax this condition if the pool is **closed** (`False`)
- Check the inductive conditions from Subsections 7.2, 7.3, this means there may be different combinations of UTXOs burned, spent (and sent back to the validator) depending on the removal type. At a high level, check a UTXO with `CurrencySymbol`, `upp'assocListCs` and `TokenName = TokenName hashedPkh` is burned, where `hashedPkh` is defined in 2.2.

  Definition 3.2 provides the correct withdrawal rewards amount during `upp'bondingLength`:

  1. **Closed pool:** `rewards`. Recall, the admin sorts out the rewards in the closing admin cycle
  2. **Open pool:** `withdrawRewards` $=$ `rewards` $+ \frac{f}{(1+\texttt{upp'interest})^k}$

     where $1 \leq k \leq$ `upp'increments` with $f := \frac{\texttt{deposited} - \cancel{\texttt{newDeposit}} + \texttt{rewards}}{\texttt{totalDeposited}} * \texttt{totalRewards}$. k can be found off chain via the correct `POSIXTimeRange` and its length (delta) should be verified on chain. Note that `newDeposit` was recently set to zero, furthermore, there is an inverse relation betweek k and time passed. As we tend towards the end of the bonding period, $k = 1$ and we almost get all rewards. But closer to the start, $k \sim$ `upp'increments` and we get less.

The withdrawal during `upp'userLength` is:

1. **Closed pool:** `rewards`. Recall, the admin sorts out the rewards in the closing admin cycle. Same as before.
2. **Open pool:** `withdrawRewards` = `rewards`$+f$ with $f := \frac{\texttt{deposited}-\texttt{newDeposit}+\texttt{rewards}}{\texttt{totalDeposited}} * \texttt{totalRewards}$. These are rewards for the last completed bonding cycle that should not include any recent deposits whilst open.

If the NFT State UTXO is spent (as part of **open withdrawal** induction), it must be sent back to the validator also, potentially altered by head withdrawal. The on-chain associated should be correctly updated as part of the inductive conditions.

To make withdrawals flexible, we can bound the allowed withdrawal amount to be `deposited` $\leq$ `withdrawalAmount` $\leq$ `deposited`$+$`(updated)Rewards` (per user) to prevent insufficient fund issues stopping a deposit withdrawal atleast.

- Related to the previous, `txInfoMint` contains exactly one `AssetClass` of -1 value. The `CurrencySymbol` should be checked with 2.3 and `TokenName` with the (hashed) signer
- Check the correct amount from of the unbonded asset class (see Definition 5.11) is sent to the signer's address (from the validator). The validator can round rewards down when validating withdrawals. The datum of these UTXOs can be checked as 3.3 in the input and output.

We should emphasise that care is needed on rounding behaviour when withdrawing as the requested withdrawal amount off chain must match what the validator expects.

6.4.2 Off chain

With the `WithdrawAct` redeemer:

- Signed by `PaymentPubKeyHash` from the redeemer and only this user
- Calculate the relevant `POSIXTimeRange` for the transaction to fall inside the `upp'userLength` or `upp'bondingLength`. Withdrawing during bonding will require a `POSIXTimeRange` of "length" `upp'interestLength`. You may need to get the current `POSIXTime` off chain
- The user needs to query UTXOs at the validator address to find their position in the associated list. If they are not in the list, cancel the transaction, otherwise, burn their Assoc. List Token; requiring the associated list minting policy from subsection 2.2 to be called with relevant inductive conditions.
- Withdraw the correct amount of the unbonded asset class (see Definition 5.11) to the user address with `AssetDatum` as datum (see Definition 3.3), sending any change back to the validator.

6.5 Admin Close

With the `CloseAct` redeemer (see Definition 4.6),

6.5.1 On Chain

- Signed by `upp'admin`, contained in the Subsection 5.2
- Transaction must occur after `upp'start` and during `upp'adminLength`
- Check all `open` fields are set to `False` on NFT State and Assoc. List UTXOs.
- Check `rewards` has been updated using the formula for updating in a normal `upp'adminLength`. Namely, rewards for the most recently completed bonding cycle. *Other fields are irrelevant now so we can leave them unchanged.*

6.5.2 Off Chain

- Signed by `upp'admin`
- Create a transaction with `POSIXTimeRange` during a `upp'adminLength`
- Spend all relevant UTXOs and change all `open` fields are set to `False`, sending back the validator.
- For now, we leave it for individual users to withdraw their tokens, as they will want to burn their Assoc. List UTXOs for min Ada too.
- Work out rewards for users and update their `rewards` field. Leave Assoc. List UTXOs at the validator that users will need to withdraw/burn to get their min Ada.
- Withdraw the NFT State UTXO to the admin address as this is not needed for **closed** withdrawing - see Subsection 7.3
- Calculate how many tokens are not part of rewards off chain and withdraw leftovers to admin. This requires admin trust and could potentially be made trustful with a folding datum and proof of how much is leftover (ignore this for simplicity)

6.6 User Query

Write a simple contract for a user to query their `EntryDatum` on the on-chain associated list. This is purely off chain and does not require validation logic.

6.7 Deposited Query

Write a simple contract for a user to query the total amount deposited at the validator. This should focus on the Assoc. List UTXO datums only (`deposited` field) as people could waste their tokens by sending to the validator. This is purely off chain and does not require validation logic.

7 Induction Conditions

This section is different to that of bonded staking.

These conditions are needed on the minting policy and validator. The scripts should prove that its `CurrencySymbol` can only be minted to insert (deposit stake) or remove (withdraw stake) from the associated list. These assume the NFT State UTXO has already been deposited at the validator address.

For **Staking**, we always mint 1 token. Notice that by construction below, we can only mint this token once for an individual user due to inequality conditions. This does not prevent multiple deposits by a user, as they would ignore the minting policy for non-initial deposits. For **Withdrawing**, we always burn 1 token.

7.1 Staking

It is important to realise that multiple deposits by a given user for staking does not involve minting a new token. Therefore, all the reward amounts below are initialised to zero for minting/depositing.

7.1.1 Head Stake

A head stake is when the NFT State UTXO is already at the validator address and we want to either initiate or alter the head element

1. **Minting logic:** Use the `Stake` minting redeemer, 4.1. In this case, check the NFT State UTXO is part of the inputs (with no other inputs of `CurrencySymbol`, `assocListCs`, see Definition 2.3). Therefore, checks are automatically forwarded to the validator.
2. **Implied Validator logic:** Use the `StakeAct` validator redeemer, 4.4 with a singleton signature for the transaction. This signature provides the key, `hashedPkh` and `TokenName` of course. There are two scenarios:

- For the initial head stake, check the NFT State UTXO is part of the inputs with datum `StateDatum Nothing True` (fail with `False`). Also check the NFT State UTXO has output datum `StateDatum (Just hashedPkh) True` with the relevant `TokenName = TokenName hashedPkh` from 2.2 and everything else unchanged. We should check the associated list UTXO has been deposited to the validator, with `assocListCs` as `CurrencySymbol` and datum

  ```
  Entry
    { key = hashedPkh
    , deposited = amount
    , newDeposit = Just amount -- new deposit to help with rewards calculation
    , rewards = 0 -- initiate to zero
    , totalRewards = 0 -- admin can update for upcoming cycle
    , totalDeposited = 0 -- admin can update for upcoming cycle
    , open = True -- by induction of an open pool
    , next = Nothing
    }
  ```

  The validator should check the (non staking) UTXOs are all unit value in inputs and outputs with correct `CurrencySymbol`s, where outputs will contain the minted UTXO in question. Note that the validator is essentially doing minting checks.

  The `newDeposit` should be initiated by the amount deposited. Incrementing occurs when the user deposits multiple times in *one* cycle, but this won't happen here since we are minting.

  The `amount` of staking UTXOs should be deposited to the validator of course.

  Note: the initial head stake is equivalent to an initial tail stake so do not define the latter.

- For altering the head, check the NFT State UTXO is part of the inputs with datum `StateDatum (Just currentHead) True`. Check the proposed hashed `BuiltinByteString`, `proposedHead` is **less** than `currentHead`. Also check the NFT State UTXO has output datum `StateDatum (Just proposedHead) True` with the relevant `TokenName = TokenName proposedHead` from 2.2 and everything else unchanged. We should check the associated list UTXO has been deposited to the validator, with `assocListCs` as `CurrencySymbol` and datum

  ```
  Entry
    { key = proposedHead
    , deposited = amount
    , newDeposit = Just amount -- new deposit to help with rewards calculation
    , rewards = 0 -- initiate to zero
    , totalRewards = 0 -- admin can update for upcoming cycle
    , totalDeposited = 0 -- admin can update for upcoming cycle
    , open = True -- by induction of an open pool
    , next = Just currentHead
    }
  ```

  The validator should check the (non staking) UTXOs are all unit value in inputs and outputs with correct `CurrencySymbol`s, where outputs will contain the minted UTXO in question.

  The `newDeposit` should be initiated by the amount deposited. Incrementing occurs when the user deposits multiple times in *one* cycle, but this won't happen here since we are minting.

  The `amount` of staking tokens should again be deposited to the validator of course.

7.1.2 Inbetween Stake

Given head insertions, there can now be a chain of Assoc. List UTXOs at the validator. An inbetween stake is where we mint (deposit) a token between two Assoc. List UTXOs (note, the NFT State UTXO must not be included) but the validator will be invoked by both UTXOs by induction.

1. **Minting logic:** Use the `Stake` minting redeemer, 4.1. Check there are two Assoc. List UTXOs are part of the input. Both UTXOs will invoke the validator so adjacency checks can be forwarded. To be stringent, we make sure the NFT State UTXO is not part of the inputs.

2. **Implied Validator logic:** Use the `StakeAct` validator redeemer, 4.4 with a singleton signature for the transaction. This signature provides the proposed key, `middleKey` and `TokenName` of course. Check there are two *adjacent* Assoc. List state UTXOs as part of the input. Adjacency can be checked by verifying the existence of the following datum structure (along with their `CurrencySymbol` & `TokenNames`):

```
Entry
  { key = firstKey
  , deposited = amount
  , newDeposit = firstNewDeposit
  , rewards = firstRewards
  , totalRewards = firstTotalRewards
  , totalDeposited = firstTotalDeposited
  , open = True -- by induction of an open pool
  , next = Just secondKey -- crucial
  }
```

```
Entry
  { key = secondKey
  , deposited = _
  , newDeposit = _
  , rewards = _
  , totalRewards = _ -- may be different to firstTotalRewards in a deposit cycle
  -- across different Entries and should be fixed during upcoming admin cycle
  , totalDeposited = _ -- may be different to firstTotalDeposited in a deposit
  -- cycle across different Entries and should be during upcoming admin cycle
  , open = True -- should be the same
  , next = _
  }
```

Check that: `firstKey` < `middleKey` < `secondKey`. The `TokenName` of the newly minted UTXO should be `TokenName middleKey` of unit value.

Check that: `open` is invariant. `totalRewards` and `totalDeposited` need not be invariant as the admin will fix this for the upcoming cycle.

We should check the associated list UTXO has been deposited to the validator, with `assocListCs` as `CurrencySymbol` and datum

```
Entry
  { key = middleKey
  , deposited = amount'
  , newDeposit = Just amount'
  , rewards = 0
  , totalRewards = 0
  , totalDeposited = 0
  , open = True
  , next = Just secondKey -- this middle UTXO now points to the latter UTXO
  }
```

The previous first entry should have its datum changed to

```
Entry
  { key = firstKey -- unchanged
  , deposited = amount -- unchanged
  , newDeposit = firstNewDeposit -- unchanged
  , rewards = firstRewards -- unchanged
  , totalRewards = firstRewards -- unchanged
  , totalDeposited = firstTotalDeposited -- unchanged
```

```
  , open = True -- unchanged
  , next = Just middleKey -- the first UTXO now points to the middle UTXO
  }
```

The second entry should be unchanged in datum and value.

The validator should check the (non staking) UTXOs are all unit value in inputs and outputs with correct `CurrencySymbol`s, where outputs will contain the minted UTXO in question.

The `newDeposit` should be initiated by the amount deposited. Incrementing occurs when the user deposits multiple times in *one* cycle, but this won't happen here since we are minting.

The `amount'` of staking tokens should be deposited to the validator of course.

7.1.3 Tail Stake

We complete the induction by validating tail insertions:

1. **Minting logic:** Use the `Stake` minting redeemer, 4.1. Check the Assoc. List UTXO (exactly one) is part of the inputs without the NFT State UTXO. Checks are then forwarded to the validator.

2. **Implied Validator logic:** Use the `StakeAct` validator redeemer, 4.4 with a singleton signature for the transaction. This signature provides the proposed key, `tailKey` and `TokenName` of course. Check there is exactly one Assoc. List state UTXOs as part of the input (without NFT State UTXO). Verifying it is indeed the tail can be checked by observing the following datum structure (along with the usual `CurrencySymbol` checks):

```
Entry
  { key = tailKey
  , deposited = amount
  , newDeposit = tailNewDeposit
  , rewards = tailRewards
  , totalRewards = tailTotalRewards
  , totalDeposited = tailTotalDeposited
  , open = True -- by induction of an open pool
  , next = Nothing -- Nothing means it is the tail
  }
```

Check that: `tailKey` $<$ `proposedTailKey`. The `TokenName` of the newly minted UTXO should be `TokenName proposedTailKey` of unit value.

We should check the associated list UTXO has been deposited to the validator, with `assocListCs` as `CurrencySymbol` and datum (this part of the minting check)

```
Entry
  { key = proposedTailKey
  , deposited = amount'
  , newDeposit = Just amount'
  , rewards = 0
  , totalRewards = 0
  , totalDeposited = 0
  , open = True -- by induction of an open pool
  , next = Nothing -- it is the new tail
  }
```

The previous tail should have its datum changed to

```
Entry
  { key = tailKey -- unchanged
  , deposited = amount -- unchanged
  , newDeposit = tailNewDeposit -- unchanged
  , rewards = tailRewards -- unchanged
```

```
, totalRewards = tailTotalRewards -- unchanged
, totalDeposited = tailTotalDeposited -- unchanged
, open = True -- unchanged
, next = Just proposedTailKey -- the previous tail now points to the new tail
}
```

The validator should check the (non staking) UTXOs are all unit value in inputs and outputs with correct `CurrencySymbol`s, where outputs will contain the minted UTXO in question.

The `newDeposit` should be initiated by the amount deposited. Incrementing occurs when the user deposits multiple times in *one* cycle, but this won't happen here since we are minting.

The `amount'` of staking tokens should be deposited to the validator of course.

7.2 Withdrawing (Open Pool)

For withdrawing, the minting policy should check the relevant UTXOs are part of the inputs. Checks on -1 valued being minted with the correct `TokenName = TokenName hashedPkh` can be forwarded to the validator as previously.

7.2.1 Head Withdraw

The head withdraw is when the NFT State UTXO and head Assoc. List UTXOS are already at the validator address and being validated.

1. **Minting logic:** Use the `Withdraw` minting redeemer, 4.2. In this case, check the NFT State UTXO and exactly one Assoc. List UTXO is part of the inputs. Therefore, checks (like checking it is indeed the head) are automatically forwarded to the validator by staking/deposit induction.

2. **Implied Validator logic:** Use the `WithdrawAct` validator redeemer, 4.5 with a singleton signature for the transaction. This signature provides the key, `currentHead` and `TokenName` of course. There are two scenarios:

   - When the list only consists of the head element, check the NFT State UTXO is part of the inputs with datum `StateDatum (Just currentHead) True` (see Subsection 7.3 for closed withdrawals). Also check the NFT State UTXO has output datum `StateDatum Nothing True` (unchanged) and everything else unchanged. We should check the associated list UTXO is part of the input and being burnt, with `assocListCs` as `CurrencySymbol` and datum

     ```
     Entry
       { key = currentHead
       , deposited = amount
       , newDeposit = _
       , rewards = rewards
       , totalRewards = totalRewards
       , totalDeposited = totalDeposited
       , open = True -- open withdrawal
       , next = Nothing -- the list only contains one element
       }
     ```

     We should check the associated list UTXO has been burnt (-1)

     The validator should check the (non staking) UTXOs are all unit value in inputs and outputs with correct `CurrencySymbol`s. Note that the validator is essentially doing minting checks.

     - If **open** during `upp'userLength`, withdraw `amount` plus $\texttt{withdrawRewards} = \texttt{rewards} + f$ with $f := \frac{\texttt{amount} - \texttt{newDeposit} + \texttt{rewards}}{\texttt{totalDeposited}} * \texttt{totalRewards}$ (rewards for the last bonding cycle).
     - If **open** during `upp'bondingLength`, withdraw `amount` plus $\texttt{withdrawRewards} = \texttt{rewards} + \frac{f}{(1+\texttt{upp'interest})^k}$ for $1 \leq k \leq \texttt{upp'increments}$ with $f := \frac{\texttt{amount} - \cancel{\texttt{newDeposit}} + \texttt{rewards}}{\texttt{totalDeposited}} * \texttt{totalRewards}$ (rewards for current incomplete bonding window).

This all assumes the admin doesn't automatically distribute rewards upon closure. This is all verifiable on chain.

- When the list contains more than one element, check the NFT State and head Assoc. List UTXOs are part of the inputs with datum `StateDatum (Just currentHead)`. Check the proposed hashed `BuiltinByteString`, equals the `currentHead`. Also check the NFT State UTXO has output datum `StateDatum (Just newHead)` and everything else unchanged. `newHead` can be obtained by looking at the head UTXO with datum (this should be verified by checking its `TokenName` of course)

```
Entry
  { key = currentHead
  , deposited = amount
  , newDeposit = _
  , rewards = rewards
  , totalRewards = totalRewards
  , totalDeposited = totalDeposited
  , open = True -- open withdrawal
  , next = Just newHead -- the list contains more than one element
  }
```

We should check the associated list UTXO has been burnt (-1)

The validator should check the (non staking) UTXOs are all unit value in inputs and outputs with correct `CurrencySymbol`s.

The same amount of staking UTXOs should be withdraw to the address that hashes to `currentHead` from the validator address (see previous bullet point).

7.2.2 Other Withdraw

Other withdrawals is where we burn (-1) a token. Two list UTXOs (the NFT State UTXO must not be included). It is intended for the **latter** (ordered) UTXO to be withdrawn. The latter can be the tail or any UTXO after the head.

1. **Minting logic:** Use the `Withdraw` minting redeemer, 4.2. Check there are two Assoc. List UTXOs are part of the input. Both UTXOs will invoke the validator so adjacency checks can be forwarded. To be stringent, we make sure the NFT State UTXO is not part of the inputs.

2. **Implied Validator logic:** Use the `WithdrawAct` validator redeemer, 4.5 with a singleton signature for the transaction. This signature provides the proposed key, `withdrawKey` and `TokenName` of course. Check there are two *adjacent* Assoc. List state UTXOs as part of the input. Adjacency can be checked by verifying the existence of the following datum structure (along with their `CurrencySymbol` & `TokenName`s):

```
Entry
  { key = firstKey
  , deposited = amount
  , newDeposit = firstNewDeposit
  , rewards = rewards
  , totalRewards = firstTotalRewards
  , totalDeposited = firstTotalDeposited
  , open = True -- open withdrawal
  , next = Just secondKey
  }

Entry
  { key = secondKey
  , deposited = amount'
  , newDeposit = _
  , rewards = rewards'
```

```
, totalRewards = _
, totalDeposited = _
, open = True -- open withdrawal
, next = secondNext
}
```

Check that: `withdrawKey` == `secondKey`. The `TokenName` of the burnt UTXO should be `TokenName withdrawKey` of -1 value.

The first UTXO should have its datum changed to

```
Entry
  { key = firstKey -- unchanged
  , deposited = amount -- unchanged
  , newDeposit = firstNewDeposit -- unchanged
  , rewards = rewards -- unchanged
  , totalRewards = firstTotalRewards -- unchanged
  , totalDeposited = firstTotalDeposited -- unchanged
  , open = bool -- unchanged
  , next = secondNext -- the first UTXO now points to the UTXO after the second
  -- UTXO (if any) - it is Nothing if withdrawing the tail.
```

The validator should check the (non staking) UTXOs are all unit value in inputs and outputs with correct `CurrencySymbol`s.

- If **open** during `upp'userLength`, withdraw `amount'` plus `withdrawRewards'` = `rewards'` + f with $f := \frac{\texttt{amount'} - \texttt{newDeposit'} + \texttt{rewards'}}{\texttt{totalDeposited}} * \texttt{totalRewards}$ (rewards for the last bonding cycle).
- If **open** during `upp'bondingLength`, withdraw `amount'` plus `withdrawRewards'` = `rewards'` + $\frac{f}{(1+\texttt{upp'interest})^k}$ for $1 \leq k \leq$ `upp'increments` with $f := \frac{\texttt{amount'} - \cancel{\texttt{newDeposit'}} + \texttt{rewards'}}{\texttt{totalDeposited}} * \texttt{totalRewards}$ (rewards for current incomplete bonding window).

This all assumes the admin doesn't automatically distribute rewards upon closure.This is all verifiable on chain.

7.3 Withdrawing (Closed Pool)

For withdrawing, the minting policy should check the relevant UTXO (just the single Assoc. List UTXO of the user). Checks on -1 valued being minted with the correct `TokenName = TokenName hashedPkh` can be forwarded to the validator as previously.

1. **Minting logic:** Use the `Withdraw` minting redeemer, 4.2. In this case, exactly one Assoc. List UTXO is part of the inputs (no NFT State UTXO). Therefore, checks are automatically forwarded to the validator by staking/deposit induction. We can **ignore** the requirement for the NFT State UTXO if the pool is closed as the admin will have withdrawn this. We do not need to update the ordering of the Assoc List UTXO because there will be no further deposits.
2. **Implied Validator logic:** Use the `WithdrawAct` validator redeemer, 4.5 with a singleton signature for the transaction. This signature provides the key and `TokenName` of course. We just need to confirm we are burning this token and withdrawing `deposited` + `rewards` (`rewards` was already updated during the admin closure cycle)