SingularityNet Bonded Staking Pool Technical Specification

April 19, 2022

Contents

1 Introduction

`Plutarch` is an eDSL in Haskell for writing on-chain scripts on Cardano. The intention is to write `SingularityNet`'s bonded 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 bonded staking pool for `SingularityNet`'s `AGIX` token.

2 Minting Policies

2.1 NFT State Minting Policy

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 "`BondedStaking`" 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` (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

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

```
data BondedStakingDatum
   = StateDatum (Maybe BuiltinByteString, Natural)
   | EntryDatum Entry
   | AssetDatum
```

Definition 3.1 (`StateDatum`)**.** This represents the staking pool/associated list and is datum for the NFT State UTXO. **The first tuple element:** `Nothing` says the list is empty, `Just` the key (corresponding to `TokenName`) to the head of the associated list. **The second tuple element:** represents the size remaining in the same way as `sizeLeft` in Definition 3.6, this is initiated to `bpp'size` (Definition 5.7) and updated during bonding by the admin.

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.

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
  , sizeLeft :: Natural
  , newDeposit :: Maybe Natural
  , deposited :: Natural
  , staked :: Natural
  , rewards :: NatRatio
  , 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 (`sizeLeft`)**.** The size remaining in the staking pool. This should be set to `neighbouringSizeLeft` when a user deposits (or redeposits), where `neighbouringSizeLeft` can be determined by adjacent UTXOs used for insertion/updating (of course they must match between adjacent UTXOs or something is wrong). This is of course inaccurate after a deposit, but the admin is expected to update this field for NFT State and all Assoc. List UTXOs during bonding. This provides a localised notion of how much size is remaining, although this can be overflown during depositing phase since we do not have a

notion of global size (this would introduce contention issues). Hence, this field is used with `newDeposit` off chain to determine the effective staked amount, `staked`. There is trust for the admin to provide the correct effective staked amount, if the user dislikes the amount, they can simply withdraw their `deposited` amount.

Definition 3.7 (`newDeposit`)**.** tells the admin whether or not the user deposit is new (initial or subsequent deposit). We need this because of potential pool overflow. `Nothing` means they have not deposited in the recent cycle, `Just newDeposit` means they have deposited `newDeposit` in the recent cycle.

Definition 3.8 (`deposited`)**.** The deposited amount by a user (this can be through multiple deposits during the same or different cycles)

Definition 3.9 (`staked`)**.** The *effective* staked amount by a user, this should be initiated to zero for new user deposits and unchanged for further deposits. In most scenarios, this is equal to `deposited`, however, they will differ if a user decides to write their own off-chain code and overflow the stakepool capacity. This field is configured by the admin during bonding period.

Definition 3.10 (`rewards`)**.** Similarly, these are the *effective* rewards accrued so far determined by the `staked` amount (and itself), *not* `deposited`. Again, this should be initiated to zero for new user deposits and unchanged for further deposits. This field is updated by the admin during bonding period.

The withdrawal amount is simply `deposited` + `rewards`, rounded down.

4 Redeemers

4.1 Minting Redeemers

```
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

```
data BondedStakingAction
  = AdminAct Natural -- 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 parameter is the new `sizeLeft` for Assoc. List UTXOs and NFT State UTXO (second tuple field).

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`.

Definition 4.6 (`CloseAct`)**.** redeemer for the admin to close the stake pool after the **last** withdrawal period has ended, withdrawing all possible leftover tokens.

5 Initialisation

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, size)
```

defining the on-chain associated list of validator/stake pool. The NFT `TokenName` can be hardcoded to "`BondedStaking`" or anything else, provided it's fixed for the codebase. The `size` can also be globally hardcoded, and should be the same as `bpp'size` in Subsection 5.2.

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

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

```
data BondedPoolParams = BondedPoolParams
  { bpp'iterations :: Natural
  , bpp'start :: POSIXTime -- absolute time
  , bpp'end :: POSIXTime -- absolute time
  , bpp'userLength :: POSIXTime -- a time delta
  , bpp'bondingLength :: POSIXTime -- a time delta
  , bpp'interest :: NatRatio
  , bpp'size :: Natural
  , bpp'minStake :: Natural
  , bpp'maxStake :: Natural
  , bpp'admin :: PaymentPubKeyHash
  , bpp'bondedAssetClass :: AssetClass
  , bpp'nftCs :: CurrencySymbol -- this uniquely parameterises the validator
  , bpp'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 and 2.

Definition 5.1 (`bpp'iterations`)**.** the number of cycles (timesteps) in the Annual Yield Percentage calculation e.g. for six months with timesteps of one month, `bpp'iterations` = 6. Here `bpp'duration` = one month.

Definition 5.2 (`bpp'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 5.3 (`bpp'end`)**.** the *absolute* `POSIXTime` the staking pool ends, after which the admin can close. This can be precalculated for convenience as `bpp'end` = `bpp'start` + `bpp'iterations` $*$ (`bpp'userLength` + `bpp'bondingLength`) + `bpp'userLength`. Note the extra `bpp'userLength` for the final withdrawal period.

Definition 5.4 (`bpp'userLength`)**.** the *timedelta* for how long users can deposit (for the upcoming cycle) or withdraw (at the end of a 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". Note that these periods overlap as in Equations 1,2

Definition 5.5 (`bpp'bondingLength`)**.** the *timedelta* for how long bonding can occur for a given cycle.

Definition 5.6 (`bpp'interest`)**.** a positive (non-zero) ratio fixed decimal. This is fixed rate for one cycle in annual percentage yield.

Definition 5.7 (`bpp'size`)**.** the size of the pool. Note: this is **not** used on chain to find the remaining pool size as this would cause contention issues. In particular, a user deposit would either require folding over all Assoc. List UTXOs to determine the total amount already staked, or, the NFT State UTXO could hold the total amount already staked - both scenarios would cause contention issues. This is taken to be the same as `size` inside the initial `StateDatum` in Subsection 5.1.

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

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

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

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

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

Definition 5.13 (`bpp'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 `bpp'assocListCs` which is possible after obtaining `bpp'nftCs`.

6 Bonded Staking Schema

For reference, here is an example of two cycles/durations

$$\texttt{bpp'start} \xrightarrow[\texttt{bpp'userLength}]{\mathit{Deposit}\ 0} B \overbrace{\xrightarrow[\texttt{bpp'bondingLength}]{\mathit{Bonding}\ 0}}^{\texttt{Admin Deposits 0}} C \xrightarrow[\texttt{bpp'userLength}]{\mathit{WithdrawOrDeposit}\ 1} D \overbrace{\xrightarrow[\texttt{bpp'bondingLength}]{\mathit{Bonding}\ 1}}^{\texttt{Admin Deposits 1}} E \xrightarrow[\texttt{bpp'userLength}]{\mathit{WithdrawOrDeposit}\ 1} \dots \tag{1}$$

Eventually, we will hit the end, as below:

$$\dots \overbrace{\xrightarrow[\texttt{bpp'bondingLength}]{\mathit{Bonding}\ n-1}}^{\texttt{Admin Deposits n-1}} X \xrightarrow[\texttt{bpp'userLength}]{\mathit{WithdrawOrDeposit}\ n} Y \overbrace{\xrightarrow[\texttt{bpp'bondingLength}]{\mathit{Bonding}\ n}}^{\texttt{Admin Deposits n}} Z \xrightarrow[\texttt{bpp'userLength}]{\mathit{Withdraw}\ n+1} \texttt{bpp'end} \tag{2}$$

In the above, we have `bpp'iterations` $= n+1$ cycles/iterations, with an extra withdrawal period, after which the admin can close the staking pool and withdraw any leftovers.

The admin deposits any rewards and updates Assoc. List UTXOs during the bonding period (this correct amount can be verified on chain). The start of the next cycle allows users to deposit further or withdraw *all* of their stake and rewards. If the admin chooses not to deposit rewards, the user can simply take their original stake out - wasting bonding time and fees.

6.1 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).

6.1.1 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 during `bpp'start` and `bpp'end` *and* in any `bpp'userLength` period (abusing terminology slightly) apart from the final withdrawal period, which can easily be calculated on chain
- Check the requested stake amount is positive (since `Natural` includes zero)
- Check the correct amount of the bonded 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 `Nothing` → `Just amount`
  2. Redeposits during the *same* cycle will have `Just prevNewDeposit` → `Just (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`, `bpp'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 staked, 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)
- By creating the Assoc. List UTXO datum (to be inserted) using the induction conditions, check the amount deposited is less or equal than `sizeLeft` (Definition 3.6) of `EntryDatum`.

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`, `bpp'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. We should also check the incremented amount is less or equal than `sizeLeft` (Definition 3.6) of `EntryDatum`. Its `key` can also be verified

- This Assoc. List UTXO should be sent to the validator otherwise unchanged.

Note that by checking `sizeLeft`, we only have a local check that the pool size has not been exceeded in one deposit cycle. However, the bonding period from Subsection 6.2 aims to fix this (per cycle).

Also note we are checking increments for both `newDeposit` and `deposited` separately, the former is needed by the admin in Subsection 6.2.

6.1.2 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 `bpp'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.
- The contract should calculate the the total `deposited` amount across all Assoc. List UTXOs on the on-chain associated list and determine whether their deposit will exceed `bpp'size`. The contract should not go ahead if it exceeds. Note that we cannot check on-chain whether this is exceeded globally (only locally via `sizeLeft`) This means a user could write their own contract to deposit (provided local `sizeLeft` is not exceeded), but the bonding period will allow the admin to fix any overflow
- Deposit the correct amount of the bonded 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`, `staked` = 0 etc.)

For **Further deposits**:

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

6.2 Admin Deposit (Bonding Period)

The admin should update NFT State & Assoc. List UTXOs and deposit rewards for users. The admin is required to update the said UTXOs *during* bonding (and before the next withdrawal/deposit period) to:

1. The NFT State UTXO's second tuple field for size remaining
2. Similarly, to update the datum's `sizeLeft` (Definition 3.6)
3. To update Assoc. List datum's `newDeposit` (Definition 3.7) to `Nothing`
4. To update the datum's `staked` (Definition 3.9)
5. To update the datum's `rewards` (Definition 3.10)

6.2.1 On chain

The admin should use the `AdminAct newSizeLeft` redeemer (see Definition 4.3), where `newSizeLeft` is determined off chain. If any of these conditions do not hold, validation should fail,

- Signed by `bpp'admin`, contained in the Subsection 5.2
- Transaction must occur during `bpp'start` and `bpp'end` *and* in any `bpp'bondingLength` period (abusing terminology), which can easily be calculated on chain
- The NFT State UTXO and multiple Assoc. List UTXOs can be spent and updated. There are three types of UTXO datum *inputs* to account for

  1. ```
     StateDatum (_, oldSizeLeft) -- oldSizeLeft must equal all Assoc. List
     -- UTXO's sizeLeft by induction
     ```
  2. ```
     Entry
       { key = _
       , sizeLeft = oldSizeLeft -- must equal all other UTXOs by induction
       , newDeposit = Nothing -- user didn't deposit during recent cycle
       , deposited = _
       , staked = _
       , rewards = rewards
       , next = _
       }
     ```
  3. ```
     Entry
       { key = _
       , sizeLeft = oldSizeLeft -- must equal all other UTXOs by induction
       , newDeposit = Just newDeposit -- user did deposit during recent cycle
       -- at least once
       , deposited = _
       , staked = staked
       , rewards = rewards
       , next = _
       }
     ```

- These should be respectively updated to:

  1. ```
     StateDatum (_, newSizeLeft)
     ```
  2. ```
     Entry
       { key = _ -- unchanged
       , sizeLeft = newSizeLeft
       , newDeposit = Nothing
       , deposited = _ -- unchanged
       , staked = _ -- unchanged
       , rewards = newRewards
       , next = _ -- unchanged
       }
     ```
  3. ```
     Entry
       { key = _ -- unchanged
       , sizeLeft = newSizeLeft
       , newDeposit = Nothing -- changed to Nothing
       , deposited = _ -- unchanged
       , staked = newStaked -- updated
       , rewards = newRewards
       , next = _ -- unchanged
       }
     ```

- Here, `newRewards` = `bpp'interest` $*$ (`updatedStaked` + `rewards`) 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 6.3 will also round (down) and we do not want insufficient funds at the validator.
- In the case of new deposits in the recent cycle (`Just newDeposit` $\rightarrow$ `Nothing`), we cannot check **on chain** what `newStaked` should be without introducing a folding proof to know the total `deposited` across all Assoc. List UTXOs, this is calculated off chain by the admin. Since we are already entrusting the admin to add rewards, we can also trust the admin to update the effective staked amount. If the user is not happy with this amount, they can simply withdraw their `deposited` amount and any previously accrued `rewards`.
- However, we can ensure `staked` $\leq$ `updatedStaked` and `rewards` $\leq$ `updatedRewards` on chain, it should be impossible to reduce any already accrued rewards and effective staked amount
- Similarly, the `newSizeLeft` is determined off chain and cannot be checked on chain. It would not make sense for the admin to sabotage this value
- All updated Assoc. List UTXOs must be sent back to the validator, otherwise unchanged
- The NFT State UTXO must be sent back to the validator if spent, otherwise unchanged
- The admin cannot withdraw (staked) asset UTXOs

6.2.2 Off chain

Using the `AdminAct newSizeLeft` redeemer:

- The admin should sign the transaction
- Calculate the relevant `POSIXTimeRange` for the transaction to fall inside the `bpp'bondingLength`. 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
- The subtlety on updating the `staked` field (and therefore `rewards`) can be done as follows:
  1. Calculate the total amount deposited across all Assoc. List UTXOs off chain, defined as `totalDeposited`. Do not count using the amount of underlying staked assets at the validator address as people can send (wasting) UTXOs to the validator, the Assoc. List UTXOs are more reliable
  2. If `totalDeposited` $\leq$ `bpp'size`, then we can set `newStaked` equal to `deposited` (for `Just newDeposit` UTXOs only, ignore the `Nothing` ones for simplicity) for each user (as output to the validator). In otherwords, all *newly* deposited users will have the same effective stake as amount they deposited
  3. Otherwise, `totalDeposited` $>$ `bpp'size`. We need to filter out `newDeposit` = `Just newDeposit` Assoc. List UTXOs as they must be the culprit UTXOs that have caused global pool overflow (although not locally). Furthermore, their `staked` value has not been updated (it cannot during user deposit). Determine the amount of overflow, `overflow` = `totalDeposited` $-$ `bpp'size`. Furthermore, for each **overflowing** user, `totalNewDeposit` = $\sum_{\texttt{users}}$ `newDeposit` ($>$ `overflow`) provides the total deposited during the *recent* cycle. `extraStake` = `totalNewDeposit`$-$ `overflow` provides the amount of extra effective stake available before hitting the `bpp'size` threshold. This should be *proportionally* distributed amongst *recent* (problematic) depositers (careful with rounding). Distribution means incrementing their `staked` fields accordingly.
  4. Update `rewards` with up-to-date `staked` value for *all* users, `newRewards` = `bpp'interest` $*$ (`updatedStaked` + `rewards`)
  5. Send $\sum_{\texttt{all}}$ `newRewards` of the staked asset to the validator address

After all `staked` fields have been updated and rewards added, we can determine `newSizeLeft` = `bpp'size` $-\sum_{\texttt{all}}$ `staked` and update all UTXO's `sizeLeft` field (including NFT State) with this value accordingly.

- 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. We may also need to save more information locally if dealing with overflow
- 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

6.3 User Withdraw

6.3.1 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 during `bpp'start` and `bpp'end` *and* in any `bpp'userLength` period (abusing terminology slightly) apart from the first deposit period, which can easily be calculated on chain
- Check the inductive conditions from Subsection 7.2, 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`, `bpp'assocListCs` and `TokenName = TokenName hashedPkh` is burned, where `hashedPkh` is defined in 2.2. The corresponding `EntryDatum` datum from Definition 3.2 provides the correct withdrawal amount `deposited` + `rewards`. If the NFT State UTXO is spent (as part of 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.
- 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 `deposited`+`rewards`, of the bonded 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.

6.3.2 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 `bpp'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, 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 bonded 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.

6.4 Admin Close

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

6.4.1 On Chain

- Signed by `bpp'admin`, contained in the Subsection 5.2
- Transaction must occur after `bpp'end`

6.4.2 Off Chain

- Signed by `bpp'admin`
- Create a transaction with `POSIXTimeRange` after `bpp'end`
- Any leftover UTXOs can be withdrawn to the admin by spending all UTXOs at the validator address

6.5 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.

6.6 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.

6.7 Staked Query

Write a simple contract for a user to query the total amount effectively staked at the validator. This should focus on the `staked` field of Assoc. List UTXO datums. This is purely off chain and does not require validation logic. It makes more sense to call this outside of the bonding period as the admin could be updating `staked` during that time but anytime is probably okay.

7 Induction Conditions

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`. Also check the NFT State UTXO has output datum `StateDatum (Just hashedPkh)` 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
       , sizeLeft = nftStateUtxoSize
       , newDeposit = Just amount -- new deposit to signify potential pool overflow
       , deposited = amount
       , staked = 0 -- effective staked amount to be updated by admin
       , rewards = 0 -- to be updated by admin, a mint means no rewards initially
       , 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 `nftStateUtxoSize` can be taken from the NFT State UTXO datum (second tuple element). This is of course inaccurate with depositing but this will be updated during bonding by the admin.

     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)`. Check the proposed hashed `BuiltinByteString`, `proposedHead` is **less** than `currentHead`. Also check the NFT State UTXO has output datum `StateDatum (Just proposedHead)` 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
       , sizeLeft = nftStateUtxoSize
       , newDeposit = Just amount -- new deposit to signify potential pool overflow
       , deposited = amount
       , staked = 0 -- effective staked amount to be updated by admin
       , rewards = 0 -- to be updated by admin, a mint means no rewards initially
       , next = Just currentHead -- the current head has been displaced by the
       }
     ```

     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 `nftStateUtxoSize` can be taken from the NFT State UTXO datum (second tuple element). This is of course inaccurate with depositing but this will be updated during bonding by the admin.

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
  , sizeLeft = firstSizeLeft
  , newDeposit = firstNewDeposit
  , deposited = amount
  , staked = firstStaked
  , rewards = firstRewards
  , next = Just secondKey -- crucial
  }

Entry
  { key = secondKey
  , sizeLeft = secondSizeLeft -- should be equal to firstSizeLeft by induction
  , newDeposit = _
  , deposited = _
  , staked = _
  , rewards = _
  , next = _
  }
```

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

   Check that: `firstSizeLeft` $==$ `secondSizeLeft`.

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

```
Entry
  { key = middleKey
  , sizeLeft = firstSizeLeft -- == secondSizeLeft
  , newDeposit = Just amount'
  , deposited = amount'
  , staked = 0
  , rewards = 0
  , 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
  , sizeLeft = firstSizeLeft -- unchanged
  , newDeposit = firstNewDeposit -- unchanged
  , deposited = amount -- unchanged
  , staked = firstStaked  -- unchanged
  , rewards = firstRewards -- 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.

Once again, the deposit makes the `sizeLeft` field incorrect, but it is up to the admin to correct this during bonding.

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
  , sizeLeft = tailSizeLeft
  , newDeposit = tailNewDeposit
  , deposited = amount
  , staked = tailStaked
  , rewards = tailRewards
  , 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
  , sizeLeft = tailSizeLeft -- equal by induction
  , newDeposit = Just amount'
  , deposited = amount'
  , staked = 0
  , rewards = 0
  , next = Nothing -- it is the new tail
  }
```

The previous tail should have its datum changed to

```
Entry
  { key = tailKey -- unchanged
  , sizeLeft = tailSizeLeft -- unchanged
  , newDeposit = tailNewDeposit -- unchanged
  , deposited = amount -- unchanged
  , staked = tailStaked -- unchanged
  , rewards = tailRewards -- 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.

Once again, the deposit makes the `sizeLeft` field incorrect, but it is up to the admin to correct this during bonding.

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

7.2 Withdrawing

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)`. Also check the NFT State UTXO has output datum `StateDatum Nothing` 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
       , sizeLeft = _ -- we could check this is equal to nftStateUtxoSize as they
       -- should equal by induction
       , newDeposit = _
       , deposited = amount
       , staked = _
       , rewards = rewards
       , 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.

We could optionally check `sizeLeft` in the Assoc. List UTXO equals that of the NFT State UTXO size left although they must equal by induction.

The `amount` + `rewards` of staking UTXOs should be withdraw to the address that hashes to `currentHead` from the validator address.

- 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
    , sizeLeft = _ -- we could check this is equal to nftStateUtxoSize but they
    -- should equal by induction
    , newDeposit = _
    , deposited = amount
    , staked = _
    , rewards = rewards
    , 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 `amount` + `rewards` of staking UTXOs should be withdraw to the address that hashes to `currentHead` from the validator address.

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
     , sizeLeft = firstSizeLeft -- == secondSizeLeft
     , newDeposit = firstNewDeposit
     , deposited = amount
     , staked = firstStaked
     , rewards = rewards
     , next = Just secondKey
     }
   ```

```
Entry
  { key = secondKey
  , sizeLeft = secondSizeLeft -- == firstSizeLeft
  , newDeposit = _
  , deposited = amount'
  , staked = _
  , rewards = rewards'
  , next = secondNext
  }
```

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

We could optionally check `firstSizeLeft` == `secondSizeLeft` although this must be true by induction.

The first UTXO should have its datum changed to

```
Entry
  { key = firstKey -- unchanged
  , sizeLeft = firstSizeLeft -- unchanged
  , newDeposit = firstNewDeposit -- unchanged
  , deposited = amount -- unchanged
  , staked = firstStaked -- unchanged
  , rewards = rewards -- 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.

The `amount' + rewards'` of staking UTXOs should be withdraw to the address that hashes to `withdrawKey` from the validator address.