Tinyman v2: Analysis of Invariants

List of Protocol Properties

In addition to the standard CPMM invariants, we have the following properties (variables below refer to pool state variables):

1. `1 <= total_fee_share <= 100`

2. `3 <= protocol_fee_ratio <= 10`

3. `asset_i_cumulative_price` is non-decreasing

4. `issued_pool_tokens == 0` iff `asset_1_reserves == 0` iff `asset_2_reserves == 0`

5. `issued_pool_tokens <= sqrt(asset_1_reserve * asset_2_reserve)`

6. if `issued_pool_tokens > 0`
   then `circulating_pool_tokens == issued_pool_tokens - LOCKED_POOL_TOKENS`
   else `circulating_pool_tokens == 0`

7. Outside of a flash loan or a flash swap,
   `get_balance(asset_i_id) >= asset_i_reserves + asset_i_protocol_fees`

8. if `issued_pool_tokens == 0`
   then `get_balance(pool_token_asset_id) == POOL_TOKEN_TOTAL_SUPPLY`
   else `get_balance(pool_token_asset_id) >= POOL_TOKEN_TOTAL_SUPPLY - issued_pool_tokens + LOCKED_POOL_TOKENS`

9. The minimum algo balance of the approval app contract must be maintained at all times.

Analysis of Invariants

We list below informal arguments for why Properties 1-9 above are invariants of the protocol. The statements are necessarily imprecise and do not constitute formal proofs of correctness, but they describe strategies that can be formalized to build these proofs.

Invariants 1, 2 and 3:

Invariants 1, 2 and 3 have trivial arguments and are thus omitted.

Invariant 4:

`issued_pool_tokens == 0`
  iff `asset_1_reserves == 0`
  iff `asset_2_reserves == 0`

Initially, right after bootstrapping the pool, `issued_pool_tokens`, `asset_1_reserves` and `asset_2_reserves` are all 0.

All non-AMM ops do not change the reserves or the issued tokens, so the property is preserved.

AMM ops:

1. add_initial_liquidity
    - (Note: the property doesn't need to hold initially)
    - The reserves are set to user-specified input amounts, asserted to be non-zero.
    - `issued_pool_tokens` is set to a value that is asserted to be greater than 1000.
    - All are non-zero, and thus the property holds.

2. add_liquidity
    - Initially, the property is assumed to hold.
    - Since `issued_pool_tokens` is asserted to be non-zero, the reserves are also non-zero.
    - Reserves are updated as either:
      `asset_i_reserves += asset_i_amount`
      Or
      `asset_i_reserves += asset_i_amount - protocol_fee_amount`
        - Since the fee is a less-than-one fraction of the input amount, the reserves remain positive.
    - `issued_pool_tokens` is updated as:
      `issued_pool_tokens = new_issued_pool_tokens - fee_as_pool_tokens`
        - `new_issued_pool_tokens > issued_pool_tokens` (since `new_issued_pool_tokens - issued_pool_tokens = pool_tokens_out` is asserted to be non-zero)

        - `fee_as_pool_tokens < pool_tokens_out` (`pool_tokens_out` is asserted to be non-zero after deducting the fee)
        - Therefore, the updated `issued_pool_tokens` strictly increases and continues to be non-zero.

3. remove_liquidity
    - Suppose the property holds initially (assuming we have the assertion that `issued_pool_tokens` is non-zero, the reserves are also non-zero initially).
    - if `(removed_pool_token_amount + LOCKED_POOL_TOKENS) == issued_pool_tokens`, then all become zero and we are done.
        - Note: Can only remove two assets in this case
    - So suppose not all liquidity is to be removed,
        - `issued_pool_tokens` must be greater than the removed tokens, so the new `issued_pool_tokens` is non-zero
        - The output amounts are less than the reserves (calculated as a fraction of the reserves that is less than one and are also rounded down), and so the new reserves are non-zero.
        - If removing liquidity in two assets, we are done.
        - Otherwise, we are removing liquidity in one asset:
            - By the post-condition of **calculate_fixed_input_swap**, the output reserve would still be non-zero.
            - The input reserve would trivially stay non-zero.
            - Therefore, the property still holds.

4. Swap
    - Suppose the property holds initially.
        - If any of the reserves is zero, the swap fails, so we may assume that the reserves are both non-zero (which implies, using this property, that `issued_pool_tokens` is non-zero as well).
    - The input supply increases by the swap amount and the fee, so it remains positive.
    - For the output supply,
        - In the “fixed input” swap, **calculate_fixed_input_swap** guarantees that the `output_amount < output_supply`, so the new `output_supply` is still non-zero.
        - In the “fixed output” swap, if the requested min is equal to the `output_supply`, the swap fails (div by zero -- see the pre-condition of **calculate_fixed_output_swap**). Therefore, `output_amount < output_supply`, and the new `output_supply` is still non-zero.
    - Finally, `issued_pool_tokens` is not touched and thus remains positive.

5. Flash_loan (the state is untouched, and so the invariant vacuously holds)

6. verify_flash_loan

- Suppose the property holds initially.
- Note that the reserves cannot be 0 since the logic requires that at least one of the output amounts is positive and that it's bounded by its asset reserve. Therefore, the reserves and `issued_pool_tokens` are all non-zero.
- After verification, the new reserves have either increased or stayed the same (non-zero), while `issued_pool_tokens` remains unchanged (non-zero). Therefore, the property still holds.

7. flash_swap (the state is untouched, and so the invariant vacuously holds)

8. Verify_flash_swap
    - Suppose the property holds initially.
    - Note that the reserves cannot be 0 since the logic requires that at least one of the output amounts is positive and that it's bounded by its asset reserve. Therefore, the reserves and `issued_pool_tokens` are all non-zero.
    - If any of the reserves becomes zero, `new_k` will be 0 while `old_k` > 0; thus, **check_invariant** fails. Therefore, a successful flash swap implies that the new reserves are both positive, while `issued_pool_tokens` remains unchanged (non-zero). Therefore, the property still holds.

Non-counterexamples

Scenario 1:

This scenario, which I initially thought was possible, would break the invariant:

1. Suppose both reserves initially are zero, and thus `issued_pool_tokens = 0`.
2. The pool receives a donation of X units of asset A.
3. Someone requests a flash loan of X units of asset A and then pays back X + Fee (X needs to be large enough for Fee to be positive) → **This fails because, in the flash_loan block, the logic checks that X is bounded above by the reserve, which is not true here**.
4. Now asset A reserve is positive, while the other reserve and `issued_pool_tokens` are both 0.

Scenario 2:

Another scenario that I thought was possible and that would essentially enable an attacker to claim extra amounts of pool assets.

Suppose we have a pool with `asset_i_reserve == X > 0`, but with an underlying balance `asset_i_balance == X + e`, where `e > 0` (so the asset has some non-zero extra amount). The attacker issues a flash swap group with the following:

1. A request for `asset_i` with amount `e`. So the `asset_i` balance after the transfer is just `X`.
2. A verify txn, which would pass all the checks: The balances are the same (there was no payment from the user for the swap), and so the reserves remain the same, all the fees are zero, **but the pool**

invariant breaks → This check fails because `e` is deducted from the asset reserve, and now the new k is lower than the old k.

Note 1: the check that the fees are non-zero, which we talked about before, would cause this scenario to fail earlier.

Note 2: this also fails in flash loans since the flash_loan logic checks that an actual repayment was made (regardless if it's for an extra amount).

Invariant 5:

`issued_pool_tokens <= sqrt(asset_1_reserve * asset_2_reserve)`

Initially, right after bootstrapping the pool, `issued_pool_tokens`, `asset_1_reserves` and `asset_2_reserves` are all 0.

All non-AMM ops do not change the reserves or the issued tokens, so the property is preserved.

AMM ops:

1. add_initial_liquidity
    - (Note: the property doesn't need to hold initially)
    - The reserves are set to user-specified input amounts (which are asserted to be non-zero).
    - `issued_pool_tokens` is set as the floor of the square root of the product of the reserves.
    - So the property holds.

2. add_liquidity
    - Initially, the property is assumed to hold.
    - The reserves `asset_i_reserves` (R_i) are updated as follows, with (i,j) in {(1,2), (2,1)}:

      R'_i = R_i + `asset_i_amount`

      R'_j = R_j + `asset_j_amount` - `protocol_fee_amount`
    - So, old_k = R_1 R_2, and new_k = R'_1 R'_2.
    - Let the old `issued_pool_tokens` be T, and the new `issued_pool_tokens` be T'. T' is set as follows (F is `fee_as_pool_tokens`):

      T' = sqrt(new_k) / (sqrt(old_k) / T) - F

      = (T / sqrt(old_k)) sqrt(new_k) - F

      <= (T / sqrt(old_k)) sqrt(new_k) since F >= 0

      <= sqrt(new_k) since T / sqrt(old_k) <= 1 by assumption
    - Therefore, the property holds.

3. remove_liquidity
    - Suppose the property holds initially.

- If `(removed_pool_token_amount + LOCKED_POOL_TOKENS) == issued_pool_tokens`, then all relevant variables become 0, and we are done.
- So suppose not all liquidity is to be removed.
    - Let X = `removed_pool_token_amount` and T = `issued_pool_tokens`
    - New reserves R'_i are calculated as follows:
      R'_i = R_i - (X / T) R_i = (1 - X / T) R_i
    - New issued pool tokens T' = T - X
    - Now, given that T <= sqrt(R_1 R_2), it can be shown that T' <= sqrt(R'_1 R'_2) through a sequence of standard algebraic manipulations.
    - Note that due to rounding, the product R'_1 R'_2 may be slightly larger than its real value, but that's in favor of the inequality.
    - If we are removing liquidity in two assets, we are done.
    - Otherwise, we are removing liquidity in one asset:
        - Due to the swap fee, the product of the new reserves R'_1 R'_2 increases further in value, while the new value of the issued pool tokens T' is unchanged, and thus the property still holds.

4. Swap
    - Suppose the property holds initially.
    - `issued_pool_tokens` does not change, and thus T' = T (the new issued is the same as the old issued).
    - The logic of the swap succeeds only if sqrt(old_k) <= sqrt(new_k), and given that T <= sqrt(old_k), we conclude that T' <= sqrt(new_k), as desired.

5. Flash_loan (the state is untouched, and so the invariant vacuously holds)

6. verify_flash_loan
    - Suppose the property holds initially.
    - After verification, the new reserves have either increased or stayed the same, while `issued_pool_tokens` remains unchanged. Therefore, the property still holds.

7. flash_swap (the state is untouched, and so the invariant vacuously holds)

8. Verify_flash_swap
    - Suppose the property holds initially.
    - Note that the reserves cannot be 0 since the logic requires that at least one of the output amounts is positive and that it's bounded by its asset reserve. Therefore, the reserves and issued_pool_tokens are all non-zero.
    - If any of the reserves become zero, new_k will be 0 while old_k > 0; thus, check_invariant fails. Therefore, a successful flash swap implies that the new reserves are both positive, while `issued_pool_tokens` remains unchanged (non-zero). Therefore, the property still holds.

Invariant 6:

if `issued_pool_tokens > 0`
    then `circulating_pool_tokens == issued_pool_tokens - LOCKED_POOL_TOKENS`
    else `circulating_pool_tokens == 0`

`circulating_pool_tokens` is a “ghost variable,” a variable that does not exist in the implementation of the contract but can be added -- conceptually -- to track an amount in the protocol and then referred to in this analysis.

If added to the state of a pool, `circulating_pool_tokens` would be set/updated in the contract as follows:

1. Bootstrap: `circulating_pool_tokens = 0`
2. Non-AMM ops wouldn’t need to update `circulating_pool_tokens`
    - Note: claim_extra does not change the circulating amount since “extra,” by definition, is circulating tokens that happen to be in the pool.
3. swap, flash_loan, verify_flash_loan, flash_swap, and verify_flash_swap do not change `circulating_pool_tokens`.
4. add_initial_liquidity
    - The tokens that go out to the pooler are the circulating tokens, so we set `circulating_pool_tokens = pool_token_out`
5. add_liquidity
    - The additional tokens that go out to the pooler are added to the currently circulating tokens: `circulating_pool_tokens += pool_token_out - fee_as_pool_tokens`
6. remove_liquidity
    - Remove the amount to be burned from the circulating amount: `circulating_pool_tokens -= removed_pool_token_amount`

Therefore, for a bootstrapped pool, only AMM ops that add or remove liquidity may affect the property.

1. add_initial_liquidity
    - Suppose the property initially holds (so `circulating_pool_tokens == 0`).
    - `issued_pool_tokens` is set to a positive value greater than `LOCKED_POOL_TOKENS`.
    - `circulating_pool_tokens` is set to `pool_token_out`, which is equal to the desired amount.
    - Therefore the property still holds

2. add_liquidity
    - Suppose the property initially holds (and so `circulating_pool_tokens == issued_pool_tokens - LOCKED_POOL_TOKENS`).
    - We have the following updates:

        C’ = C + `pool_token_out - fee_as_pool_tokens`

T’ = T + `pool_token_out - fee_as_pool_tokens`

- Therefore, the property still holds.

3. remove_liquidity
   - Suppose the property initially holds (so `circulating_pool_tokens == issued_pool_tokens - LOCKED_POOL_TOKENS`).
   - If all liquidity is to be removed, i.e. `removed_pool_token_amount == circulating_pool_tokens`, then we have the following updates:
     `circulating_pool_tokens = 0`
     `issued_pool_tokens = 0`
     And so the property still holds.
   - Otherwise, if not all liquidity is to be removed (i.e., `removed_pool_token_amount < circulating_pool_tokens`), `removed_pool_token_amount` is deducted from both the circulating and the issued amounts, so the property still holds.

Invariant 7:

Outside of a flash loan or a flash swap,
`get_balance(asset_i_id) >= asset_i_reserves + asset_i_protocol_fees`

Initially, when the pool is bootstrapped, the reserves and protocol fees are all set to 0, and so the property holds.

Note: Donations do not invalidate the property.

Below, we may refer to the two assets of a pool as A_i and A_j, with balances, reserves, and protocol fees B_i, R_i, F_i, and B_j, R_j, F_j, respectively. The post-state variables are primed, e.g., B_i is the original pre-state balance of A_i while B’_i is the post-state balance of the asset.

The following ops may affect this property:

1. claim_fees
   - Suppose the property holds initially.
   - The fee is deducted from the asset balance, and `asset_i_protocol_fees` is set to 0. Therefore, the property still holds.

2. claim_extra
   - Suppose the property holds initially.
   - The balance is reduced by the excess amount beyond the reserve and the protocol fee, and thus the new balance is exactly `asset_i_reserves + asset_i_protocol_fees`. Therefore, the property still holds.

3. add_initial_liquidity
    - Suppose the property holds initially.
    - The balance is increased by `asset_i_amount`, and `asset_i_reserves` is set to exactly `asset_i_amount`. Simple algebraic manipulations can be used to show that the property still holds.

4. add_liquidity
    - Suppose the property holds initially.
    - The balances are increased by `asset_1_amount` and `asset_2_amount`.
    - The reserves `asset_i_reserves` (R_i) and fees `asset_i_protocol_fees` (F_i) are updated as follows, with (i,j) in {(1,2), (2,1)}:

        R’_i = R_i + `asset_i_amount`
        F’_i = F_i
        R’_j = R_j + `asset_i_amount` - `protocol_fee_amount`
        F’_j = F_j + `protocol_fee_amount`
    - Therefore, in both cases, the property still holds.

5. remove_liquidity
    - Suppose the property holds initially.
    - If removing liquidity (fully or partially) in two assets, the balances and reserves decrease by `asset_i_amount`, while the protocol fees remain unchanged, and thus the property still holds.
    - Otherwise, we are removing liquidity in one asset, A_i. We have the following updates:

        B’_i = B_i - (`asset_i_amount + swap_output_amount`)
        R’_i = R_i - `asset_i_amount - swap_output_amount`
        F’_i = F_i
        B’_j= B_j
        R’_j = R_j - `asset_j_amount + (swap_amount + poolers_fee_amount)`
        F’_j = F_j + `protocol_fee_amount`

        The property holds trivially for A_i. For A_j, since `swap_amount == asset_j_amount - (poolers_fee_amount + protocol_fee_amount)`, the original reserve is reduced by `protocol_fee_amount`, and thus the property also holds.

6. swap
    - Suppose the property holds initially.
    - For a “fixed-input” swap, we have the following updates:

        B’_i = B_i + `input_amount`
        R’_i = R_i + `swap_amount + poolers_fee_amount`
        F’_i = F_i + `protocol_fee_amount`
        B’_j= B_j - `output_amount`
        R’_j = R_j - `output_amount`

F'_j = F_j

The property holds trivially for the output asset. For the input asset, since `input_amount == swap_amount + poolers_fee_amount + protocol_fee_amount`, the property holds for the input asset as well.

- For a "fixed-output" swap, we have the following updates:

    B'_i = B_i + `swap_amount + total_fee_amount`
    R'_i = R_i + `swap_amount + poolers_fee_amount`
    F'_i = F_i + `protocol_fee_amount`
    B'_j= B_j - `output_amount`
    R'_j = R_j - `output_amount`
    F'_j = F_j

  The property holds trivially for the output asset. For the input asset, since `total_fee_amount == poolers_fee_amount + protocol_fee_amount`, the property holds for the input asset as well.

7. verify_flash_loan
    - Suppose the property holds initially (before a flash_loan request).
    - For a loan on asset A_i, we have the following updates:

        B'_i = B_i - `asset_i_output_amount + asset_i_input_amount`
        R'_i = R_i + `asset_i_poolers_fee_amount`
        F'_i = F_i + `asset_i_protocol_fee_amount`

      Since `asset_i_input_amount >= asset_i_output_amount + asset_i_total_fee_amount` , the net balance B'_i >= B_i + `asset_i_total_fee_amount` , and thus the property holds.

8. verify_flash_swap
    - Suppose the property holds initially.
    - For a flash swap on assets A_1 and A_2 that is paid back in both assets, we have the following updates:

        B'_i = B_i - `asset_i_output_amount` + `asset_i_input_amount`
        R'_i = R_i - `asset_i_output_amount` + `asset_i_input_amount` - `asset_i_protocol_fee_amount`
        F'_i = F_i + `asset_i_protocol_fee_amount`

      Therefore, the property holds for both assets. Note that borrowing one asset and/or repayment in one asset is a special case of the above.

Invariant 8:

if `issued_pool_tokens == 0`
    then `get_balance(pool_token_asset_id) == POOL_TOKEN_TOTAL_SUPPLY`

else `get_balance(pool_token_asset_id) >= POOL_TOKEN_TOTAL_SUPPLY - issued_pool_tokens + LOCKED_POOL_TOKENS`

In particular, if `issued_pool_tokens > 0`, then `get_balance(pool_token_asset_id) >= LOCKED_POOL_TOKENS`

Initially, when the pool is bootstrapped, `issued_pool_tokens == 0`, and the balance is `POOL_TOKEN_TOTAL_SUPPLY`. Therefore, the property holds.

Note: We assume donations of pool tokens are not possible when `issued_pool_tokens == 0`. When `issued_pool_tokens > 0`, donations do not invalidate the property.

Let T and T' be the initial and final issued pool tokens and B and B' be the initial and final pool balances. We denote `POOL_TOKEN_TOTAL_SUPPLY` by P and `LOCKED_POOL_TOKENS` by L.

The following ops may affect this property:

1. claim_extra
    - Suppose the property holds initially.
    - The operation succeeds when `issued_pool_tokens > 0` only, so we may assume that's the case.
    - The balance is reduced by an amount equal to the number of circulating tokens that happen to be in the pool, and so the balance decreases by (B is the pool's balance):
      B - `LOCKED_POOL_TOKENS` - `POOL_TOKEN_TOTAL_SUPPLY` + `issued_pool_tokens`
    - Therefore, the property still holds.

2. add_initial_liquidity
    - Suppose the property holds initially, and thus the pool balance is exactly `POOL_TOKEN_TOTAL_SUPPLY`.
    - `issued_pool_tokens` is set to a positive value (greater than `LOCKED_POOL_TOKENS`)
    - The balance is decreased by `issued_pool_tokens - LOCKED_POOL_TOKENS`. Therefore, we get B >= `POOL_TOKEN_TOTAL_SUPPLY`, which is true, and so the property still holds.

3. add_liquidity
    - Suppose the property holds initially, and thus we have for the pool balance B:
      B >= `POOL_TOKEN_TOTAL_SUPPLY - issued_pool_tokens + LOCKED_POOL_TOKENS`
    - We have the following updates:
      B' = B - (`pool_tokens_out` - `fee_as_pool_tokens`)
      T' = T + `pool_tokens_out` - `fee_as_pool_tokens`
    - T' stays positive (it either increases or remains the same), and thus we need to have the following:

      $$B' >= P - T' + L$$

⇒ B - (`pool_tokens_out` - `fee_as_pool_tokens`) >=
P - (T + `pool_tokens_out` - `fee_as_pool_tokens`) + L
⇒ B >= P - T + L
which holds by assumption.

- Therefore, the property still holds.

4. remove_liquidity
    - Suppose the property holds initially, and thus we have for the pool balance B:
      B >= P - T + L
    - If removing all liquidity, we have `removed_pool_token_amount` == T - L,
        - So we have the following updates:
          B' = B + `removed_pool_token_amount` == B + T - L
          T' = 0
        - So we need to have B' == P, or B + T - L == P, or B == P - T + L, which holds by assumption
    - If removing liquidity partially (in one or two assets),
        - we have the following updates
          B' = B + `removed_pool_token_amount`
          T' = T - `removed_pool_token_amount`
        - So we need to have B' >= P - T' + L, or B >= P - T + L (by substitution), which holds by assumption
    - Therefore the property still holds

Invariant 9:

Minimum algo balance of the approval app contract that must be maintained at all times.

The min balance of the approval contract account increases temporarily when the account submits the cost-budget-increasing txn by 100000 microalgos, so the account needs to maintain this additional amount whenever that's needed. We assume the approval contract account is initially created with having this amount at least in its balance.

claim_extra leaves this amount in the balance whenever algo donations are claimed.

The min balance of the approval account permanently increases only when a pool opts in. The pool sends the required amount to the approval contract during opting in.

There is no other way to send algos out of the contract account or increase the min balance requirement. Therefore, the property holds.