Standardized Yield - A token standard for yield generating mechanisms

Vu Nguyen[*]
vu@pendle.finance

Long Vuong[*]
longvh@pendle.finance

October 14, 2022

Abstract

DeFi has been growing rapidly with increasingly innovative protocols. Even with the variety of DeFi protocols, the core utility that they provide largely remains the same: users generate yield by staking or providing liquidity to the protocol. Despite this, most protocols build their yield generating mechanisms differently, necessitating a manual integration every time a protocol builds on top of another protocol's yield generating mechanism. In this paper, we introduce a model to generalise all yield generating mechanisms in DeFi. Using this model, we are proposing Standardized Yield (SY), a new token standard to standardize the interaction with all yield generating mechanisms in DeFi.

[*]All authors contributed equally to this work

1 Overview

We will first introduce Generic Yield Generating Pool (GYGP), a model to describe most yield generating mechanisms in DeFi, in section 2. Then, we will propose SY, a token standard for any yield generating mechanism that conforms to the GYGP model. If yield generating mechanisms of every protocol are converted into SY, it will enable unprecedented levels of composability in DeFi.

In section 4, we will talk about Simple GYGP, a subset of GYGP that is very common in DeFi.

In the last section, we will discuss the potential use cases for SY and their implications for DeFi

2 Generic Yield Generating Pool

In this section, we are introducing "Generic Yield Generating Pool", a model for a family of yield generating mechanisms, that should cover most of DeFi.

2.1 Definitions

pool In each yield generating mechanism, there is a central pool that contains value contributed by users.

asset is a unit to measure the value of the pool. At time t, the pool has a total value of $A(t)$ **assets**

share is a unit that represents ownership of the pool. At time t, there are $S(t)$ **shares** in total.

reward tokens over time, the pool earns $n_{rewards}$ types of reward tokens ($n_{rewards} \geq 0$). At time t, $R_i(t)$ is the amount of **reward token** i that has accumulated for the pool since $t = 0$

exchangeRate At time t, the exchange rate $E(t)$ is simply how many **assets** each **share** is worth

$$E(t) = A(t)/S(t)$$

user At time t, each user u has $s_u(t)$ **shares** in the pool, which is worth $a_u(t) = s_u(t) * E(t)$ **assets**. At time t, user u will have accrued a total of $r_{u_i}(t)$ **reward token** i. The following always hold true:

$$S(t) = \sum s_u(t)$$
$$A(t) = \sum a_u(t)$$
$$R_i(t) = \sum r_{u_i}(t)$$

可以推出

$$s_u(t) = \frac{a_u(t)}{E(t)}$$

在SYS的Lemma 6中会用到
SYS白皮书的SYIndex就相当于SY白皮书的exchangeRate

At $t1$, a **asset** is equivalent to $s_u = \frac{a}{SY\ Index(t1)}$ **SY tokens**

2.2 State changes

Let's denote $t*$ as the timestamp of the previous state change before the current state change. Over time, there can only be 3 types of events that will change the state of the pool:

1. A user u deposits d_a **assets** worth into the pool at time t (d_a could be negative, which means a withdraw from the pool. $d_a + a_u(t*) \geq 0$). $d_s = d_a/E(t*)$ new shares will be created and given to user u (or removed and burned from user u when d_a is negative)

用户存款

$$\begin{aligned} a_u(t) &= a_u(t*) + d_a \\ s_u(t) &= s_u(t*) + d_s = s_u(t*) + d_a/E(t*) \\ A(t) &= A(t*) + d_a \\ S(t) &= S(t*) + d_s = S(t*) + d_a/E(t*) \\ E(t) &= A(t)/S(t) = E(t*) \end{aligned}$$

For the very first deposit at time t then $S(t*) = 0$, $E(t*) = E_0$ which is a constant called **initial exchange rate**

2. The pool earns d_a (or loses $-d_a$ if d_a is negative) **assets** at time t. The **exchange rate** simply increases (or decreases if d_a is negative) due to the additional **assets**:

rebasing tokens才有

$$\begin{aligned} A(t) &= A(t*) + d_a \\ S(t) &= S(t*) \\ E(t) &= A(t)/S(t) = E(t*) + d_a/S(t*) \end{aligned}$$

For every user:

$$\begin{aligned} s_u(t) &= s_u(t*) \\ a_u(t) &= a_u(t*) + d_a \frac{s_u(t*)}{S(t*)} \end{aligned}$$

3. The pool earns $d_{r_i}(t)(d_{r_i}(t) > 0)$ **reward token** i at time t

和当前池子的底层资产不一致的token

$$R_i(t) = R_i(t*) + d_{r_i}(t)$$

Every user will receive a certain amount of **reward token** i

$$\begin{aligned} r_{u_i}(t) &= r_{u_i}(t*) + d_{r_{iu}}(t) \\ \sum d_{r_{iu}}(t) &= d_{r_i}(t) \end{aligned}$$

2.3 Examples

Yield generating mechanism	USDC lending in Compound	stake LOOKS in Looksrare	stake 3crv in Convex
asset	USDC	LOOKS	3crv pool's liquidity D
shares	cUSDC	shares (in contract)	3crv LP token
reward tokens	COMP	WETH	CRV + CVX
exchange rate	Increases with USDC lending yield	increases with LOOKS rewards	increases due to swap fees

3 SY: A token standard for GYGP

In this section, we are proposing SY, a token standard for any yield generating mechanism that conforms to the GYGP model. On top of the definitions in 2.1.1, we need to define 2 more concepts:

Input tokens are tokens that can be converted into **assets** to enter the pool. Each SY can accept several possible **input tokens** $token_{in_i}$

Output tokens are tokens that can be redeemed from **assets** when exiting the pool. Each SY can have several possible **output tokens** $token_{out_i}$

3.1 The SY Interface

Based on the definitions so far, we can introduce the interfaces for SY:

On top of the ERC20 interface, SY has the following functions:

```
event Deposit(
    address indexed caller,
    address indexed receiver,
    address indexed tokenIn,
    uint256 amountDeposited,
    uint256 amountSYOut
);

event Redeem(
    address indexed caller,
    address indexed receiver,
    address indexed tokenOut,
    uint256 amountSYToRedeem,
    uint256 amountTokenOut
);

event ClaimRewardss(address indexed user, address[] rewardTokens, uint256[] rewardAmounts);

function deposit(
    address receiver,
    address tokenIn,
    uint256 amountTokenToDeposit,
    uint256 minSharesOut
) external returns (uint256 amountSharesOut);

function redeem(
    address receiver,
    uint256 amountSharesToRedeem,
    address tokenOut,
    uint256 minTokenOut
) external returns (uint256 amountTokenOut);

function redeemAfterTransfer(
    address receiver,
    address tokenOut,
    uint256 minTokenOut
) external returns (uint256 amountTokenOut);

function exchangeRate() external view returns (uint256 res);

function claimRewards(address user) external returns (uint256[] memory rewardAmounts);

function accruedRewards(address user) external view returns (uint256[] memory rewardAmounts);

function getRewardTokens() external view returns (address[] memory);
```

```
    function yieldToken() external view returns (address);

    function getTokensIn() external view returns (address[] memory res);

    function getTokensOut() external view returns (address[] memory res);

    function isValidTokenIn(address token) external view returns (bool);

    function isValidTokenOut(address token) external view returns (bool);

    function previewDeposit(address tokenIn, uint256 amountTokenToDeposit)
        external
        view
        returns (uint256 amountSharesOut);

    function previewRedeem(address tokenOut, uint256 amountSharesToRedeem)
        external
        view
        returns (uint256 amountTokenOut);

    enum AssetType {
        TOKEN,
        LIQUIDITY
    }

    function assetInfo()
        external
        view
        returns (
            AssetType assetType,
            address assetAddress,
            uint8 assetDecimals
        );
```

3.2 Functions in the SY interface

3.2.1 *deposit*

```
    function deposit(
        address receiver,
        address tokenIn,
        uint256 amountTokenToDeposit,
        uint256 minSharesOut
    ) external returns (uint256 amountSharesOut);
```

This function will deposit *amountTokenToDeposit* of **input token** i (*tokenIn*) to mint new SY shares.

This function will convert the $amountTokenToDeposit$ of **input token** i into d_a worth of **asset** and deposit this amount into the pool for the *receiver*, who will receive *amountSharesOut* of SY tokens (**shares**), according to state change type 1 as defined in section 2.2

This function should revert if *amountSharesOut* $<$ *minSharesOut*

3.2.2 $redeemAfterTransfer$

```
function redeemAfterTransfer(
    address receiver,
    address tokenOut,
    uint256 minTokenOut
) external returns (uint256 amountTokenOut);
```

This function will use the floating amount d_s of SY tokens (**shares**) in the SY contract to redeem to **base token** i (*tokenOut*). This pattern is similar to UniswapV2 which allows for more gas efficient ways to interact with the contract.

This function will redeem the d_s **shares**, which is equivalent to $d_a = d_s * E(t)$ **assets**, from the pool, according to state change type 1 as defined in section 2.2. The d_a **assets** is converted into exactly $amountTokenOut$ of **base token** i (*tokenOut*).

This function should revert if $amountTokenOut < minTokenOut$

3.2.3 *redeem*

```
function redeem(
    address receiver,
    uint256 amountSharesToRedeem,
    address tokenOut,
    uint256 minTokenOut
) external returns (uint256 amountTokenOut);
```

This function is similar to $redeemAfterTransfer$, but will explicitly deposit *amountSharesToRedeem* into the SY contract for the redemption.

3.2.4 *claimRewards*

```
function claimRewards(address user) external returns (uint256[] memory rewardAmounts);
```

Let's say t_{last} is the last time this function was called for user u ($user$). At time t, for each **reward token** i, this function will send the unclaimed reward tokens to user u, which is equal to:

$$r_i^{out} = r_{u_i}(t) - r_{u_i}(t_{last})$$

rewardAmounts is simply the array of r_i^{out}

3.2.5 *exchangeRate*

```
function exchangeRate() external view returns (uint256 res);
```

At time t, this function returns $E(t) * 1e18$

3.2.6 *accruedRewards*

```
function accruedRewards(address user) external view returns (uint256[] memory rewardAmounts);
```

At time t, this function returns an array of accrued rewards $r_{u_i}(t) - r_{u_i}(t_{last})$ for all the reward tokens.

3.2.7 *yieldToken*

```
function yieldToken() external view returns (address);
```

This read-only function returns the underlying yield token address.

3.2.8 *getTokensIn*

```
function getTokensIn() external view returns (address[] memory res);
```

This read-only function returns the list of all **input tokens**

3.2.9 *getTokensOut*

```
function getTokensOut() external view returns (address[] memory res);
```

This read-only function returns the list of all **output tokens**

3.2.10 *isValidTokenIn*

```
function isValidTokenIn(address token) external view returns (bool);
```

This read-only function returns whether *token* is a compatible **input token**

3.2.11 *isValidTokenOut*

```
function isValidTokenOut(address token) external view returns (bool);
```

This read-only function returns whether *token* is a compatible **output token**

3.2.12 *getRewardTokens*

```
function getRewardTokens() external view returns (address[] memory);
```

This read-only function returns the list of all **reward tokens**

3.2.13 *previewDeposit*

```
function previewDeposit(address tokenIn, uint256 amountTokenToDeposit)
    external
    view
    returns (uint256 amountSharesOut);
```

This read-only function returns the amount of shares that a user would have received if they deposit *amountTokenToDeposit* of *tokenIn*.

3.2.14 *previewRedeem*

```
function previewRedeem(address tokenOut, uint256 amountSharesToRedeem)
    external
    view
    returns (uint256 amountTokenOut);
```

This read-only function returns the amount of *tokenOut* that a user would have received if they redeem *amountSharesToRedeem* of *tokenOut*.

3.2.15 *assetInfo*

```
function assetInfo()
    external
    view
    returns (
        AssetType assetType,
        address assetAddress,
        uint8 assetDecimals
    );
```

This read-only function contains information to interpret what the asset is. *assetDecimals* is the decimals to format asset balances.
For *assetType* and *assetAddress*:

- If asset is an ERC20 token, $assetType = 0$, $assetAddress$ is the asset token address
- If asset is liquidity of an AMM (like sqrt(k) in UniswapV2 forks), $assetType = 1$, $assetAddress$ is the address of the AMM pool

4 Simple GYGP

In this section, we will introduce Simple GYGP, a GYGP subset that describes most GYGPs in DeFi.

4.1 Definition

In the general mechanics of a GYGP, we have defined exactly how a change in **asset** balance of the pool will be distributed among the users (in state change number 2 in Section 2.2). However, an increase in **reward tokens** of the pool might be distributed in many different ways (in state change number 3 in Section 2.2).

Simple GYGPs are GYGPs where an influx of rewards are immediately distributed equally among the users, based on their current **shares**

More concretely, for state change number 3: when there is an earning of $d_{r_i}(t)$ $(d_{r_i}(t) > 0)$ **reward token** i at time t:

$$R_i(t) = R_i(t*) + d_{r_i}(t)$$

3. The pool earns $d_{r_i}(t)(d_{r_i}(t) > 0)$ **reward token** i at time t

$$R_i(t) = R_i(t*) + d_{r_i}(t)$$

Every user will receive a certain amount of **reward token** i

$$r_{u_i}(t) = r_{u_i}(t*) + d_{r_{i_u}}(t)$$

$$\sum d_{r_{i_u}}(t) = d_{r_i}(t)$$

For every user, they get equally distributed based on their **shares**:

$$r_{u_i}(t) = r_{u_i}(t*) + d_{r_i}(t) \times \frac{s_u(t)}{S(t)}$$

Let's define the **SY reward index** $rewardIndex_i(T)$ for reward token i at time T:

$$rewardIndex_i(T) = \sum_0^T \frac{d_{r_i}(t)}{S(t)}$$

We can see that for a user u who has the same amount of shares s_u from t_1 to t_2:

$$r_{u_i}(t_2) = r_{u_i}(t_1) + s_u \times (rewardIndex_i(t_2) - rewardIndex_i(t_1)) \qquad (1)$$

This means that to implement a Simple GYGP, we can just store the global $rewardIndex_i(t)$ and each user's entitled rewards could be updated every time their amount of shares changes, according to the formula above.

可以推出

不需要对每个用户更新r的份额时都做除法，更节省gas

4.2 Examples of Simple GYGP

- Sushi's Onsen rewards (in SUSHI) for stakers
- Compound's COMP rewards for suppliers
- Looksrare's WETH rewards for LOOKS stakers
- Traderjoe's USDC rewards for sJOE stakers

$$\begin{aligned} R_i(t_2) &= \sum r_{u_i}(t_2) \\ &= \sum r_{u_i}(t_1) + \sum s_u(t_1) \times (rewardIndex_i(t_2) - rewardIndex_i(t_1)) \\ &= R_i(t_1) + S(t_1) \times (rewardIndex_i(t_2) - rewardIndex_i(t_1)) \end{aligned}$$

4.3 Simple SY

进而推出

The SY corresponding to Simple GYGP is simply known as Simple SY. Simple SY's implementation follows exactly how reward tokens are distributed according to Simple GYGP.

$$\begin{aligned} d_{r_i} &= R_i(t_2) - R_i(t_1) \\ &= S(t_1) \times (rewardIndex_i(t_2) - rewardIndex_i(t_1)) \end{aligned}$$

这个式子在SYS的Lemma6的证明中会用到

5 Use cases for SY

5.1 Money markets

If a money market takes in yield generating tokens in the form of SY as collateral (like LP tokens, or liquid staking tokens), it will be much easier to integrate the claiming of reward tokens as well as accounting for the additional returns from the SY.

If the money market's lending position follows SY itself, it will make it much easier for other protocols to build on top of the lending positions.

5.2 Vaults

Vaults usually have to deal with various types of yield generating mechanisms. If all these mechanisms follow the same SY standard, it will be much cleaner and easier to write the strategies.

5.3 Dapp aggregators

Any yield generating mechanisms that follow SY standard could be integrated in dapp aggregators like Zapperfi and Zerion instantly, since the interfaces for querying, displaying and interacting with them are exactly the same.

5.4 Wallets

Wallets will be able to show user positions and APYs easily in any SY that they participate in.