

# Super Composable Yield

Pendle

April 20, 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 Super Composable Yield (SCY), a new token standard to standardize the interaction with all yield generating mechanisms in DeFi.

# 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 SCY, a token standard for any yield generating mechanisms that conform to the GYGP model. If the yield generating mechanisms of every protocol are converted into SCY, 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 SCY 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.

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

**shares** 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 **accounting assets** each **shares** is worth

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

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

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

## 2.2 State changes

Over time, there could only be 3 types of events to change the state of the pool:

1. A user  $u$  deposits  $d_a$  **accounting assets** worth into the pool at time  $t + 1$  ( $d_a$  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)

$$\begin{aligned} a_u(t + 1) &= a_u(t) + d_a \\ s_u(t + 1) &= s_u(t) + d_s = s_u(t) + d_a/E(t) \\ A(t + 1) &= A(t) + d_a \\ S(t + 1) &= S(t) + d_s = S(t) + d_a/E(t) \\ E(t + 1) &= A(t + 1)/S(t + 1) = E(t) \end{aligned}$$

For the very first deposit at time  $t + 1$  when  $S(t) = 0$ ,  $E(t) = E_0$  which is a constant called **initial exchange rate**

2. The pool earns  $d_a$  (or loses  $-d_a$  if  $d_a$  is negative) **accounting assets** at time  $t + 1$ . The **exchange rate** simply increases due to the additional **accounting assets**:

$$\begin{aligned} A(t + 1) &= A(t) + d_a \\ S(t + 1) &= S(t) \\ E(t + 1) &= A(t + 1)/S(t + 1) = E(t) + d_a/S(t) \end{aligned}$$

For every user:

$$\begin{aligned} s_u(t + 1) &= s_u(t) \\ a_u(t + 1) &= a_u(t) + d_a \frac{s_u(t)}{S(t)} \end{aligned}$$

3. The pool earns  $d_{r_i}$  ( $d_{r_i} > 0$ ) **reward token  $i$**  at time  $t + 1$

$$R_i(t + 1) = R_i(t) + d_{r_i}$$

For every user, they get distributed a certain amount of **reward token  $i$**

$$\begin{aligned} r_{u_i}(t + 1) &= r_{u_i}(t) + d_{r_{i_u}} \\ \sum d_{r_{i_u}} &= d_{r_i} \end{aligned}$$

## 2.3 Examples

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

### 3 SCY: A token standard for GYGP

In this section, we are proposing SCY, a token standard for any yield generating mechanisms that conform to the GYGP model. On top of the definitions in 2.1.1, we need to define another concept:

#### 3.1 Base tokens

**base tokens:** tokens that can be converted into **accounting assets** to enter the pool. When exiting the pool, **accounting asset** can be converted back into **base tokens** again. Each SCY could accept several possible **base tokens**

At time  $t$ , we can convert  $b_i$  **base token**  $i$  into  $b_i \times c_i^{in}(t)$  **accounting asset**. We can also convert  $d_a$  **accounting asset** into  $d_a / c_i^{out}(t)$  **base token**  $i$ . It always holds that  $c_i^{in}(t) \leq c_i^{out}(t)$

#### 3.2 The SCY Interface

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

On top of the ERC20 interface, an SCY has the following functions:

```
function mintNoPull(
    address receiver ,
    address baseTokenIn ,
    uint256 minAmountScyOut
) external returns (uint256 amountScyOut);

function redeemNoPull(
    address receiver ,
    address baseTokenOut ,
    uint256 minAmountBaseOut
) external returns (uint256 amountBaseOut);

function mint(
    address receiver ,
    address baseTokenIn ,
    uint256 amountBaseToPull ,
    uint256 minAmountScyOut
) external returns (uint256 amountScyOut);

function redeem(
    address receiver ,
    address baseTokenOut ,
    uint256 amountScyToPull ,
    uint256 minAmountBaseOut
) external returns (uint256 amountBaseOut);

function updateGlobalRewards() external;
function updateUserRewards(address user) external;
function redeemReward(address user) external returns (uint256[] memory outAmounts);

function scyIndexCurrent() external returns (uint256);
function scyIndexStored() external view returns (uint256);

function getBaseTokens() external view returns (address[] memory);
function isValidBaseToken(address token) external view returns (bool);
function getRewardTokens() external view returns (address[] memory);

// Metadata
function assetDecimals() external view returns (uint8);
function assetId() external view returns (bytes32);
```

### 3.3 Functions in the SCY interface

#### 3.3.1 *mintNoPull*

```
function mintNoPull(  
    address recipient ,  
    address baseTokenIn ,  
    uint256 minAmountScyOut  
) external returns (uint256 amountScyOut);
```

This function expects that there are already  $b_i$  **base token  $i$**  (*baseTokenIn*) in the SCY contract. This pattern is similar to UniswapV2 (and UniswapV3) pools, which allow for maximum composability by minimising token transfers. For example, a router contract could swap from some other token to **base asset  $i$**  which is sent directly to the SCY contract before *mint* is called.

This function will convert the  $b_i$  of **base asset  $i$**  into  $d_a$  worth of **accounting asset** and deposit this amount into the pool for the *recipient*, who will receive *amountScyOut* of SCY tokens (**shares**), according to state change type 1 as defined in section 2.2

This function should revert if  $amountScyOut < minAmountScyOut$

#### 3.3.2 *redeemNoPull*

```
function redeem(  
    address recipient ,  
    address baseTokenOut ,  
    uint256 minAmountBaseOut  
) external returns (uint256 amountBaseOut);
```

This function expects that there are already  $d_s$  SCY tokens (**shares**) in the SCY contract. Similar to *mint*, this pattern allows for better composability.

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

This function should revert if  $amountBaseOut < minAmountBaseOut$

#### 3.3.3 *mint and redeem*

These are the pull versions of *mintNoPull* and *redeemNoPull*

#### 3.3.4 *updateGlobalRewards*

```
function updateGlobalRewards() external;
```

At time  $t$ , this function updates the global accounting for rewards, basically updating  $R_i(t)$  (for  $i = 1..n_{rewards}$ ), after all the state change type 3 that has happened, as defined in Section 2.2.

#### 3.3.5 *updateUserRewards*

```
function updateUserRewards(address user) external;
```

At time  $t$ , this function updates the accounting for rewards for user  $u$  (*user*), basically updating  $r_{u_i}(t)$  (for  $i = 1..n_{rewards}$ ), after all the state change type 3 that has happened, as defined in Section 2.2.

### 3.3.6 *redeemRewards*

```
function redeemReward(address user) external returns (uint256[] memory outAmounts);
```

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:

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

*outAmounts* is simply the array of  $r_i^{out}$

### 3.3.7 *scyIndexCurrent*

```
function scyIndexCurrent() external returns (uint256);
```

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

### 3.3.8 *scyIndexStored*

```
function scyIndexStored() external view returns (uint256);
```

This read-only function returns the last saved value of  $E(t)$ .

### 3.3.9 *getBaseTokens*

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

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

### 3.3.10 *isValidBaseToken*

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

This read-only function returns whether *token* is a **base token**

### 3.3.11 *getRewardTokens*

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

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

### 3.3.12 *assetDecimals*

```
function assetDecimals() external view returns (uint8);
```

This read-only function returns the decimals for formatting the raw balances of **accounting asset** into user friendly balances

### 3.3.13 *assetDecimals*

```
function assetId() external view returns (bytes32);
```

This read-only function returns a string to identify the **accounting asset** being used in the SCY.

The convention is:

- If the **accounting asset** is equivalent to a token, *assetId* is simply the token address converted into bytes32
- If the **accounting asset** is equivalent to the liquidity of an AMM pool, *assetId* = "liquidity:" + bytes32(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 **accounting 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  $d_{r_i}$  ( $d_{r_i} > 0$ ) **reward token**  $i$  at time  $t + 1$ :

$$R_i(t + 1) = R_i(t) + d_{r_i}$$

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

$$r_{u_i}(t + 1) = r_{u_i}(t) + d_{r_i} \frac{s_u(t)}{S(t)}$$

### 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

### 4.3 Simple SCY

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

## 5 Use cases for SCY

### 5.1 Money markets

If a money market takes in yield generating tokens in the form of SCY 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 SCY.

If the money market's lending position follows SCY 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 SCY standard, it will be much cleaner and easier to write the strategies.

### **5.3 Dapp aggregators**

Any yield generating mechanisms that follow SCY 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 SCY that they participate in.