Beanstalk: Husk Proposal

aki & friends

January 6, 2022

1 Introduction

Beanstalk provides an innovative model for an algorithmic credit-backed stablecoin, Bean, whereby new Beans enter the supply via a queue of Pods. The success of the protocol depends on Bean holders being willing to Sow their Beans, which means burning them in exchange for new Pods minted at the end of the queue. Currently, the Pod queue exceeds the Bean supply at a ratio of about 11.5:1, representing an estimated waiting time of about 3 years between Sowing and Harvesting.

Just as wETH is a wrapper atop ETH to make it ERC-20 compliant, we propose creating an ERC-20 token, Husk, which abstracts away the non-fungible nature of Pods. As long as we can price Pods properly, any Plot, which is a contiguous range of Pods, can effectively be represented by some number of fungible tokens that hold the same value. Husks allow for the creation of a liquidity pool in a decentralized constant product automated market maker (AMM) like Uniswap, providing immediate liquidity for Beanstalk creditors.

Creating a liquid market for Pods is likely to support the health of the protocol by decreasing the lock-up costs borne by individual users and making the market dynamics more efficient. This proposal may work well in tandem with a NFT pod marketplace centered around asks and bids that allows for custom pricing models and estimates at the expense of consistent liquidity.

2 Pricing Pods

2.1 Pricing Function

We propose a function to determine the price of an unripened Pod (P_D) at a specific position (n) in the Field queue of length D with an arbitrary Weather (w), given the price of a Bean in terms of some other asset Y ($\bar{P}_{\emptyset:Y}$):

$$P_D(n) = \frac{\bar{P}_{\emptyset:Y}}{(1 + \frac{w}{100})^{\frac{n}{D}}} \tag{1}$$

2.2 Derivation

Since a Harvestable Pod can be redeemed for a Bean at any time, they are equivalent in price. As such, a Pod that has just become Harvestable (i.e. position 0 in the queue) has a price of $\bar{\emptyset}1$:

$$P_D(0) = \bar{P}_{\bar{\emptyset}:Y} \tag{2}$$

The price of a Pod at the end of the queue is given as a function of the weather (w), since at any time where there is Soil ($S > 0$), we can lend Beans and expect to receive $1 + \frac{w}{100}$ Pods per Sown Bean. Thus, the price of a Pod at the end of the queue with length D is given by:

$$P_D(D) = \frac{\bar{P}_{\bar{\emptyset}:Y}}{1 + \frac{w}{100}} \tag{3}$$

Let us assume an exponential relationship between the position of a Pod in the Field and its price (refer to 2.3 for justification). We can use Equations 2 and 3 to solve a generic equation $P_D(n) = ab^n$ to determine $a = \bar{P}_{\bar{\emptyset}:Y}$ and $b = (\frac{1}{1+\frac{w}{100}})^{\frac{1}{D}}$. A simple substitution results in Equation 1.

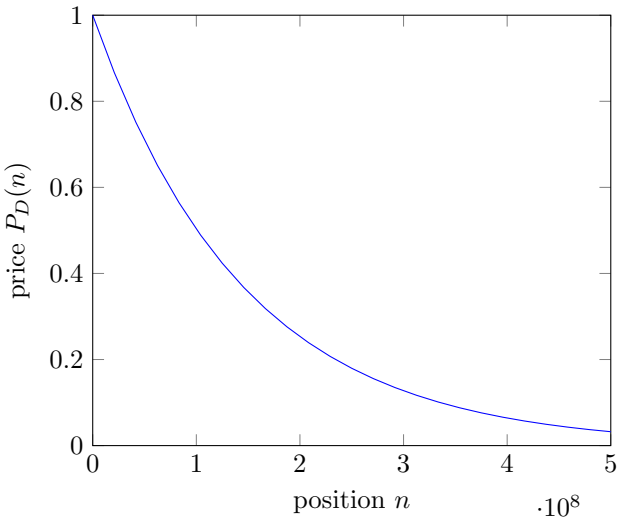
2.3 Justification

Conceptually, a Pod is a non-callable zero-coupon bond where the discount rate on the maturity value is $1/w$. A zero-coupon bond with some face value P_{FV}, interest rate r, and t years to maturity can be priced as follows:

$$P = \frac{P_{FV}}{(1+r)^t}$$

Indeed, this equation bears the same inverse exponential relationship and is of the same form as our derived price function. Unlike the typical zero-coupon bond, Pods are not time-bound and do not expire, so the exponent in the denominator must be replaced by some measure of the Pod's maturity.

We can additionally verify that the price function outlined in Equation 1 makes intuitive sense by plotting the relationship. For a Pod line of length 500M and Weather set at 3000%:



3 Husk Liquidity Pool

As an ERC-20 representation of Pods, Husks allow us to utilize AMMs such as Uniswap to manage and provide liquidity, rather than necessitating an NFT marketplace to facilitate transactions. The introduction of a Husk liquidity pool may also correct inefficiencies in the Pod pricing function. As such, we propose the creation of a $\not\Diamond$:HUSK liquidity pool and seeding it with initial liquidity.

The pricing function outlined in Equation 1 includes the price of a Bean in terms of some other asset Y and determines the price of a specific Pod in terms of Y as well. The creation of a $\not\Diamond$:HUSK liquidity pool provides the price of a Bean in Husks determined by the market, eliminating the need to make an implicit assumption like 1 Husk = $\not\Diamond 1$ while determining how many Husks to mint and burn during wrapping and unwrapping Plots.

Like other Beanstalk price oracle readings, we propose the use of a time-weighted average price (TWAP) of the $\not\Diamond$:HUSK liquidity pool to provide some amount of manipulation resistance:

$$P_D(n) = \frac{\bar{P}_{\not\Diamond:HUSK}}{(1+\frac{w}{100})^{\frac{n}{D}}} \tag{4}$$

4 Husk Implementation

4.1 Overview

The objective of introducing Husks to the Beanstalk ecosystem is to create a fungible representation of Pods. The difference in prices of distinct Pods is han-

dled while wrapping and unwrapping Plots; a more valuable Plot is exchanged with a larger number of Husks.

Using the price function defined in Equation 4, we define the current total Husk supply (H) to be:

$$H = \int_0^D P_D(n)\, dn$$

The max total Husk supply (H_{max}) occurs in the case that every existing Pod in the queue becomes Harvestable: $H_{max} = D * \bar{P}_{\not{0}:HUSK}$. The lower bound for the Husk supply is 0, in the cases where there are no wrapped Pods or every Pod has been Harvested. These bounds are strictly adhered to at any point in time as Husks are burned when unwrapping and redeeming Pods.

4.2 Wrapping

Wrapping a Plot locks it up in the Beanstalk Diamond storage and adds a number of Husks to the caller's balance. This amount of Husks is computed by applying the Pod pricing function to the range of Pods contained in the Plot.

```
function wrapPlot(uint256 id, uint256 start, uint256 end) {
    transferPlot(msg.sender, address(this), id, start, end);
    uint256 amount = getPlotPrice(id, start, end);
    husk().mint(msg.sender, amount);
}
```

4.3 Unwrapping

Unwrapping a Plot releases it to the user who called the method, burning a number of Husks from the user's balance determined by applying the Pod pricing function to the range of Pods contained in the Plot. If the user does not have the requisite amount of Husks, the transaction is cancelled. Any user can unwrap any locked Plot as long as they have enough Husks.

```
function unwrapPlot(uint256 id, uint256 start, uint256 end) {
    uint256 amount = getPlotPrice(id, start, end);
    husk().burn(amount);
    transferPlot(address(this), msg.sender, id, start, end);
}
```

5 Glossary

The following variables are used in this proposal:

D - The number of unripened Pods in the queue.

$\overline{P}_{\not\Diamond}$ - Beanstalk oracle price of $\not\Diamond 1$.

$\overline{P}_{\not\Diamond:HUSK}$ - TWAP price of the $\not\Diamond$:HUSK liquidity pool.

$\overline{P}_{\not\Diamond:Y}$ - The price of $\not\Diamond 1$ in terms of some asset Y.

$P_D(n)$ - Price of a Pod at position n in the Field queue, where index 1 signifies the next Pod to become Harvestable.

w - The percentage of additional Beans Harvested from 1 Sown $\not\Diamond$.