Financial Virtual Machine

Alexander Angel Waylon Jepsen Colin Roberts Estelle Sterrett

March 1, 2023

This is a living document. This print's version is: v0.1.0

Abstract

This document descibes the design implications of building a Finite State Machine (FSM) with its own opcodes on top of the Ethereum Virtual Machine (EVM). We examine this design pattern in the context of Decentralized Finance (DeFi) in what we call the Financial Virtual Machine (FVM). The FVM is designed to be a general purpose FSM that can be used to interact with variety of financial primitives. Also, it provides an interface over atomicity and the ability to reason about program correctness. In particular, FVM enables users to take multiple actions on their portfolios within an *single* commit to Ethereum. The individual transactions in FVM can also accrue transient debt, but no lasting debt will be written to the blockchain due to the validation of state after processing. For example, rebalancing an arbitrary amount of positions can be done atomically which drastically simplifies both the design and removes the need for signing multiple transactions. The FVM is a critical component of Primitive's Portfolio protocol, which builds on top of Constant Function Market Maker (CFMM) Liquidity Provider (LP) positions.

1 Introduction

Portfolio is a modular system that provides efficient decentralized financial infrastructure for portfolio management. It consists of two main components, a virtual machine known as Financial Virtual Machine (FVM) designed with internal accounting and execution logic, coupled with agnostic support for Constant Function Market Maker (CFMM) trading functions. By using the two components together, Liquidity Providers (LPs) gain capabilities of allocating to a variety of portfolios, all while allowing atomicity for CFMM transactions. It is also possible to use the accounting to serve as a means to rebalance portfolios and route swaps across pools; both can be done in aggregate.

Section 2 covers the FVM's functionality, implementation, and examples. We can apply some of these techniques in order to scale orderflow and provide some new features to the Decentralized Exchange (DEX) space. Then in Appendix A we will also discuss further theory on Automated Market Makers (AMMs), their payoffs, and their associated liquidity distributions in order to motivate future work.

2 Financial Virtual Machine

We examine constructing a Virtual Machine (VM) on top of the Ethereum Virtual Machine (EVM) Instruction Set Architecture (ISA) (set of opcodes) with specific interest in Decentralized Finance (DeFi) applications. Programming languages such as Solidity are an interface for the EVMs ISA that ultimately are compiled into the machine bytecode. Since the EVM is a stack machine, bytecode is executed iteratively by the EVM when a block is validated. The block is then sent as a commit to Ethereum so that the resultant world-state is updated from the transactions in the block. Note that transactions are not committed to Ethereum as a single transaction, but rather a set of transactions that are executed atomically given their approval of Ethereum's state transition function.

The atomicity of applications on the EVM can be handled in many different ways. For example, layer-2 protocols such as Arbitrum, Optimism, ZK-Sync, and Aztec are built on top of the EVM and rollup transactions into a single atomic commit to Ethereum. We show another example of how to extend atomicity by constructing a VM on top of the EVM that has its own state transition function. To execute Ethereum opcodes atomically, we must constrain our computation to reside in a single EVM commitment or transaction. These machines can use Ethereum's world-state as read-only-memory and can only write to the world-state when the machine halts upon valid execution that Ethereum can accept. That is, Ethereum's memory can be rewritten only when both the VM halts and the VM is in a valid state. This section will introduce the concept of VMs on top of Ethereum and describe our implementation of the FVM.

2.1 Background on State Machines

To discuss the topic more formally, let us introduce a few notions on state machines. First, we are interested in building on top of the EVM which is a quasi-Turing complete machine [11]. Note that Ethereum's gas limit imposes execution to halt once a certain amount of computation is performed, hence quasi-Turing complete. Our implementation will be a Finite State Machine (FSM) that bootstraps execution from EVM.

Formally, a FSM is a defined as $\{\mathcal{S}, \mathcal{I}, \mathcal{O}, \nu, \omega, \text{and } \mathcal{F}\}$ such that

- $\mathcal{S}$ is a non-empty finite set of ***states***.
- $\mathcal{I}$ is a non-empty finite set of ***inputs***.

- $\mathcal{O}$ is a non-empty finite set of ***outputs***.
- ν is the ***state transition function*** $\nu : \mathcal{S} \times \mathcal{I} \to \mathcal{S}$.
- ω is the ***output function*** $\omega : \mathcal{S} \times \mathcal{I} \to \mathcal{O}$.
- $\mathcal{F}$ is a set of ***final states***.

FSMs are a simple model for computation. Turing machines are more powerful than FSMs, but they are also more complex. A CPU has a finite amount of inputs, outputs, and states and thus can be modeled as a FSM. However, a computer as a whole has infinite internal interactions that can be modeled as an unbounded machine or Turing machine. Furthermore FSM's can provide a contraint system that allows for only certain state transitions to occur. This makes it is easier to reason about correctness.

2.2 Theory

There are many common financial transactions that can be modeled as a FSM. Furthermore, Ethereum provides the most primitive environment that natively enables transfer of value via token. This lets us build further abstractions of financial transactions as a FSM on top of Ethereum. Our canonical example will be that of a DEX where users can provide and remove liquidity or swap one basket of tokens for another.

Innovations in DeFi have shown that there are some unique implications of transaction atomicity when making commits to the EVM. One classic example is flash loans which are used for arbitrage. Without verifiable transaction atomicity [6], atomic multi-leg arbitrage requires users to maintain a large amount of funds. Atomicity is an essential concept for Portfolio because it allows traders to wrap multiple financial transactions into one atomic commit to Ethereum. Portfolio leverages the FVM to provide internal accounting across all liquidity pools. Succinctly, we will sequentially execute many FVM opcodes for every Ethereum commit and define our own rules for valid state transitions.

First, let us define the FVM as a FSM such that:

- $\mathcal{S}$ is a collection of all CFMM pool states and a single user portfolio.
- $\mathcal{I}$ is an (infinite) collection of all strings of FVM opcodes.
- $\mathcal{O}$ is a single user's portfolio value.
- ω is the output function $\omega : \mathcal{S} \to \mathcal{O}$.
- ν is the state transition $\mathcal{S} \times \mathcal{I} \to \mathcal{S}$.
- $\mathcal{F}$ is the (finite) set of final valid states given by the constraint of ν.

Most importantly, the ν for the FVM requires that the portfolio value is invariant after the aggregated execution of all FVM opcodes specified by the inputs i:

$$\nu(s,i) = s_f \text{ iff } \omega(s_f) = \omega(s). \tag{1}$$

The state s of pools and a single user's portfolio is loaded from the EVM's world state.

To build an instruction set that satisfies the above constraints of a FSM with Solidity, the state of the FVM must be transient. Transient storage is a design put forward in EIP-1153 [2] that enables a in-transaction temporary memory. The EIP preposes two additional opcodes for reading and writing to temporary non persistant storage. This design concept exposes an interface of execution atomicity that can be leveraged in a number of ways. As an EIP, transient storage was preposed in 2018, but it has not been implemented in any forks. Our implementation achieves similar functionality to EIP-1153 by implementing a function to loop through jumps.

The pool and user portfolio states s are fetched as read-only from outside our FSM from the EVM. Immediately, we can determine that the set of states $\mathcal{S}$ and outputs $\mathcal{O}$ are both singletons containing just the user and pool data s and the portfolio value $\omega(s)$ respectively. Once this has happened, the FVM is given inputs that can modify the state and hold the mutated state in transient memory and not as a commit to Ethereum. In essence, the state transition function and output function are independent of the choice of state $s \in \mathcal{S}$. Once the string of opcodes $i \in \mathcal{I}$ are exhausted by the FVM, the output mapping ν is applied to the final mutated state s_f. Using our condition in Equation (1) we yield a binary output

$$\begin{cases} \text{Valid}, & \nu(s,i) \models \omega(s_f) = \omega(s) \\ \text{Invalid}, & \nu(s,i) \not\models \text{otherwise}. \end{cases} \tag{2}$$

The requirement for validity s_f defines the set $\mathcal{F}$. In effect, the validity of $\omega(s,i)$ requires that the user cannot change their portfolio's value by engaging with the Portfolio protocol. Given that $\omega(s,i)$ is valid, the FVM can attempt to commit the mutated state s' to the EVM as a single atomic transaction where Ethereum's state transition will also be verified. These invariants ensure that Portoflio does not allow a user to accrue irrecoverable debt. However, it is important to mention that users may alter the portfolio values of other users, but their states are not loaded into the machine for execution.

Also, FVM does hold instances of temporary mutated states s', but these states are transient. This construction results in a fully functional transient FSM on top of the EVM. The properties of the state transition ν ensure the correctness of computation as long as the single system invariant is satisfied. Note that FVM is not Turing complete.

2.3 Implementation

One can think of the system invariant given by Equation (2) as a way to ensure the solvency of the protocol. This machine consists of basic opcodes that are to be ran with Portfolio pool objects. The purpose of these instructions is to provide a way to execute multiple financial transactions, any one of which may require transient debt for a user. By doing so, we get the benefits of atomicity and transactional efficiency.

The FVM abstracts the token accounting for native network currencies (e.g. Ether), ERC20 tokens, and liquidity positions all into the Portfolio contracts. Along with the accounting system, the FVM implements the data structures for the CFMM's curves, token pairs, state of pools, and liquidity positions. Finally, FVM handles the execution of orders through a single order "process" method and a multi-order "jump process" method. These processes expect to receive an a string of FVM instructions $i \in \mathcal{I}$, which, in our case, is a single byte representation of an operation for the machine to process. The individual instructions that FVM can string together are given in Table 1.

Instruction	Bytecode
UNKNOWN	0x00
ALLOCATE	0x01
UNSET02	0x02
DEALLOCATE	0x03
CLAIM	0x04
SWAP	0x05
UNSET06	0x06
UNSET07	0x07
UNSET08	0x08
UNSET09	0x09
CREATE_POOL	0x0B
CREATE_PAIR	0x0C
UNSET0D	0X0D
INSTRUCTION_JUMP	0xAA

Table 1: The different instructions for the FVM.

The instructions can be summarized as follows:

UNKNOWN: This is the default opcode. It is used to represent an unknown opcode, and is used to initialize the FVM's state.

ALLOCATE: This instruction is used to add liquidity to a pool. It maintains invariant pricing for each pool that is interacted with.

DEALLOCATE: This instruction is used to remove liquidity from a pool. It maintains invariant pricing for each pool that is interacted with.

CLAIM: Collects all the fees generated from a positive invariant for (i.e., only if overreplicating, see Appendix A.2.1).

SWAP: This instruction is used to swap between the tokens of a pool. It maintains the invariant of the trading curve.

CREATE_POOL: This instruction is used to create a new pool. Initially, pools are not deployed with any capital, but are deployed with parameters for the CFMM as well as an initial price.

CREATE_PAIR: This instruction is used to initialize a new pair of assets for which pools can be created.

INSTRUCTION_JUMP: This instructions is used to jump to a different instruction in the FVM's state via FVM's pointer.

UNSETXX: This instruction is not in use at the moment but can be implemented later.

The state for any user consists of any of their current positions, their token balances, and the current state of the pools they interact with. For a quick example, let the user u have with public address $\mathrm{Address}_u$ and a position in a pool $p \in \mathrm{Pool}$ which we call LPT_p, a balance of x Token X, and a balance of y of Token y, the state loaded from the EVM is

$$s = \begin{pmatrix} \mathrm{Address}_u \\ \mathrm{LPT}_p \\ x \\ y \\ \mathrm{Pools} \end{pmatrix}. \tag{3}$$

Note that x and y are the balances for the user in Portfolio that were added outside of FVM.

Pools themselves have their own state (discussed in Appendix A.1) which consist of reserves $\mathbf{R}$, trading function φ, and any parameters the trading function may need (e.g., strike, expiry). These parameters must be stored in the on-chain state of the contract, so they can be loaded prior to performing actions from inputs to FVM. FVM has four data structures to hold these three key groups of information: a token pair (Pair), a parameter set (Curve), state of a liquidity pool (Pool), and a liquidity position (Position). There are two operations to instantiate these pieces of data: `CREATE_PAIR`, `CREATE_POOL`. Positions are mapped using the owner of the position and the pool ID which is created after executing `ALLOCATE`. Pairs and pools use a nonce-based method to increment their ID to reference their data. Pairs have an ID number up to three bytes in size, for example the first pair created has an ID of `0x000001`, the second has `0x000002`, etc. For pools, it's the same, but up to eight bytes.

We can then allow the user to provide their own inputs $i \in \mathcal{I}$ to the FVM that can modify the state s. For example, if the user wants to deallocate liquidity from pool p, swap on pool q to get the correct ratio in order to allocate liquidity into new pool r we can put

$$i = \begin{pmatrix} \texttt{DEALLOCATE}: [\mathrm{LPT}_p] \\ \texttt{SWAP}: [x_0, q \in \mathrm{Pools}] \\ \texttt{CREATE\_POOL}: [\varphi, \mathbf{P}, r \in \mathrm{Pools}] \\ \texttt{ALLOCATE}: [x_1, y_1, r] \end{pmatrix} \tag{4}$$

Notice that in i, the user may take on transient debt before all opcodes have been executed.

Recall that after applying ν to the state s and inputs i, must not change in order to confirm validity. If the portfolio value does change, the instructions will revert and signal an invalid commit to Ethereum. Do remember that a user's portfolio value may change, but not from their own interaction with pools (except for the accounted loss or gain from fees paid to LPs).

Using this low level machine, the logic of the CFMM is implemented in the Portfolio smart contract, but executed by FVM's methods. Portfolio implements the logic for creating pools, adding or removing liquidity from them, and swapping between the tokens of the pool. Since each transaction maintains invariant portfolio value via CFMM invariants and verification that the user does not take on lasting debt. Of course, the accounting to handle fees which do represent an accounted decrease of a swappers portfolio value that add the relevant LP's portfolios. Once that is assured, the transaction is committed to Ethereum's on-chain state.

All transaction data should begin with the four byte function selector for the multiprocess instruction opcode. This is different from the standard way to interact with Ethereum smart contracts, which usually use the first four bytes of the keccak256 hash of the target function's signature [11]. To summarize, no functions other than the multiprocess function are being called on Portfolio protocol, only data in the form of FVM opcodes is sent directly to the smart contract.

The ID of the pool is the combination of both those ID nonces, e.g. `0x000002 01 00000001`, where pairNonce/pairId is 3 bytes, 1 byte for isMutablePool, 4 bytes for poolNonce. Creating pools does not require the user to pay for the initial liquidity, but does require a price to be set.

2.3.1 Processing and Encoding

To process a transaction, we need a way of inputting bytecode to the FVM and storing state. The Portfolio contract will receive a message from the user and will trigger the contract's custom multiprocess function. The message will be a sequence of bytes $b_0 b_1 \dots b_n$.

Inside of custom multiprocess is a call to the `jumpProcess` method on the FVM. This call will check the first byte for `0xAA` (`INSTRUCTION_JUMP` opcode) and, if not there, then it will execute the relevant existing opcode. The message is passed on to the FVM and the FVM executes the bytecode using a pointer, e.g., we reference a byte in the message by $b_{\texttt{pointer}}$. Messages are processed serially and the pointer is incremented. An occurence of the `JUMP-INSTRUCTION` instruction can be used to jump to the pointer location of the next FVM instruction. The message is processed and decoded into a sequence of instructions to be comitted to Ethereum. If the message begins with an `JUMP_INSTRUCTION`, the FVM will jump to the instruction at the pointer. From there, we call the `process` method on the FVM to execute the bytecode.

We can use the FVM contract as an interface in order to create instruction messages. For each of the FVM instructions, there is a corresponding encode and decode function. The encode functions take in the parameters for the instruction and return a byte array while the decode functions invert this process and revert on invalid byte strings. To make messages more compact after encoding each instruction, we use a reverse run-length encoding scheme.

2.4 Orderflow with FVM

Given the way that FVM abstracts accounting, we can use this interface to implement a flash loan pattern in which collateralization is always ensured and transactions revert when a condition is not met. This provides us a way in which we can improve orderflow without having to worry about the collateralization of the protocol. Also, this gives users a way to perform a sequence of transactions without having to sign for each individual one.

There are future directions to consider with FVM. For instance, pools over the same pairs can be aggregated in the transient state of FVM. This would allow for simpler order routing and better execution prices on swaps as well as improved portfolio replication. Specifically, with bounded payoffs we can combine liquidity distributions and generate an aggregate trading curve that represents both pools simultatenously and integrate further accounting into FVM.

By doing this, we can route across both pools and unify liquidity. In effect, FVM can be used to quote a single price per pair and abstract away the notion of individual pool swapping. This makes the protocol more scalable and allows us to have a simpler swap interface. These routing features are not yet implemented, but are in development.

References

[1] Adams, H., Zinsmeister, N., Salem, M., Keefer, R., and Robinson, D. Uniswap V3 Core.

[2] Alexey Akhunov, and Moody Salem. EIP-1153: Transient storage opcodes. https://eips.ethereum.org/EIPS/eip-1153, 2018.

[3] Angeris, G., Agrawal, A., Evans, A., Chitra, T., and Boyd, S. Constant Function Market Makers: Multi-Asset Trades via Convex Optimization. 31.

[4] Angeris, G., Evans, A., and Chitra, T. Replicating Market Makers, Mar. 2021.

[5] Angeris, G., Evans, A., and Chitra, T. Replicating Monotonic Payoffs Without Oracles, Nov. 2021.

[6] Chandler, B., Stiles, P., and Blinken, J. DeFi Flash Loans: What 'Atomicity' Makes Possible - Why Does This Innovation Not Already Exist in Traditional Finance? *SSRN Electronic Journal* (2022).

[7] Milionis, J., Moallemi, C. C., Roughgarden, T., and Zhang, A. L. Automated Market Making and Loss-Versus-Rebalancing. 21.

[8] Natolski, J., and Werner, R. Mathematical analysis of different approaches for replicating portfolios. *European Actuarial Journal 4*, 2 (Dec. 2014), 411–435.

[9] Sterrett, E., Angel, A., Czernik, M., and experience. Primitive whitepaper. *PrimitiveXYZ* (2021).

[10] Sterrett, E., Jepsen, W., and Kim, E. Replicating Portfolios: Constructing Permissionless Derivatives, June 2022.

[11] Wood, D. G. Ethereum.

A Automated Market Making

In this section, we will discuss automated market making via CFMMs in decentralized networks, i.e., DeFi. AMMs are a solution in DeFi that addresses orderflow, price discovery, and exchange of assets autonomously. We will specifically focus on CFMMs which source their liquidity from LPs. LPs in CFMMs are positions that replicate some portfolio outlook and they can earn fees on swaps on the CFMM. Using CFMMs to achieve portfolios is a beneficial tool in DeFi since it replaces the need of manually rebalancing a portfolio. Instead of having to manually replicate a portfolio via rebalancing, one can simply provide liquidity to a CFMM pool and earn the payoff of the portfolio autonomously. Primitive's Portfolio is designed to facilitate this type of liquidity provisionr.

One way to describe portfolio allocation is by the studying the entire space of portfolios that any particular collection of AMM LP positions could provide. The collection of such LP portfolios are in correspondence with Replicating Market Maker (RMM) which were described by [5]. We are not limited to static RMM pools, but can also consider dynamic RMM pools that can be updated over time or other conditions. One such example that we have implemented is an RMM that replicates the payoff of a Black-Scholes European covered call option. We will refer to this RMM position as RMM-CC. Note that RMMs such as RMM-CC are able to achieve the payoff is via arbitrage against the corresponding CFMM pool.

Lastly, we can think of RMMs with bounded portfolio values nicely in terms of their liquidity distributions. By liquidity distributions we mean the amount of open order density that is available at each price point. This is partially motivated by the notion of concentrated in UniswapV3 [1]. Using this framing, we can combine liquidity distributions simply and view time-varying RMM positions as a dynamic way to allocate concentrated liquidity.

A.1 CFMM Background

At the moment, most DEX fall under the classification of CFMMs. The CFMM acts as a direct counterparty to every trade submitted on supported trading pairs. This, of course, means CFMMs must maintain reserves of each supported asset deposited by LPs, as well as provide a price for every submitted trade.

Suppose we have a collection of n tokens that can be exchanged for another. The ***reserves*** of n assets $\boldsymbol{R} \in \mathbb{R}^n_+$ is called an ***n-asset liquidity pool*** where for every $i \in \{1,\dots,n\}$, the quantity R_i represents the quantity of asset i in the pool. Reserves change when trades are executed.

Let $\boldsymbol{\Delta}, \boldsymbol{\Delta'} \in \mathbb{R}^n_+$ be the ***tendered basket*** and ***received basket***, respectively. We refer to the tuple, $(\boldsymbol{\Delta}, \boldsymbol{\Delta'}) \in \mathbb{R}^n_+ \times \mathbb{R}^n_+$, as a *proposed trade*. Specifically, Δ_i and Λ_i denote the amount of asset i tendered. Fees are typically applied to the tendered basket and they are a means of incentivizing users to provide liquidity. We define the *fee* as a parameter $\gamma \in (0,1]$.

Definition A.1. A *Constant Function Market Maker (CFMM)* is an n-asset pool $\mathbb{R}^n_+$, and a *trading function* φ

$$\varphi\colon \mathbb{R}^n_+ \to \mathbb{R}. \tag{5}$$

Given any $\mathbf{R}$, the value $\varphi(\mathbf{R}) = k$ is called the *invariant*.

CFMMs have two independent actors. Actors who tender a basket of assets $\boldsymbol{\Delta}$ in order to receive a nonzero basket $\boldsymbol{\Delta'}$ are *swappers*. Actors who provide those assets that can be swapped are called the *LPs*. Let us first discuss swappers.

Definition A.2. Let $(\boldsymbol{\Delta}, \boldsymbol{\Delta'}) \neq 0$, then this trade is a *valid swap* if

$$\varphi(\boldsymbol{R} + \gamma\boldsymbol{\Delta} - \boldsymbol{\Delta'}) = \varphi(\boldsymbol{R}). \tag{6}$$

Graphically, Definition A.2 dictates that valid swaps move reserves along the *invariant set* where $\varphi = k$. An illustration of these curves is given in Figure 2 for RMM-CC at various times. Note that when $\gamma = 1$, the swapper is simply paying $\boldsymbol{\Delta}$ in order to receive $\boldsymbol{\Delta'}$. In the case that $\gamma < 1$ (which is typical), the trade is accepted based on the discounted tendered basket, $\gamma\boldsymbol{\Delta}$, but the reserves are still increased by the full $\boldsymbol{\Delta}$. This remainder $(1-\gamma)\boldsymbol{\Delta}$ serves to increase the value the LP's share of the pool which we call a Liquidity Provider Token (LPT).

Since the trading function gives a method for calculating exchange quantities, it also in turn, provides a means to computing marginal prices of each token in a CFMM. That is, the condition for a valid swap Equation (6) allows us to determine both the relative marginal prices of each asset and the portfolio value of an LP. For us, we will follow [3] and let our n-th asset be the numeraire.

Definition A.3. Given a CFMM the *price vector* is

$$\boldsymbol{P} := \nabla\varphi, \tag{7}$$

the *reported price of asset i* is

$$p_i := \frac{P_i}{P_n}, \tag{8}$$

and the *value of the reserves* is

$$V(\boldsymbol{R}) := \frac{1}{P_n}\boldsymbol{P}^\top\boldsymbol{R}. \tag{9}$$

It should be noted that while this price only depends on the CFMM's reserves, it will generally track the wider market prices of each asset pairs by virtue of arbitrage. In general, for a given trading function φ, the reserves $\boldsymbol{R}$ can be determined from $\boldsymbol{P}$. For the two token case, we will write the price $S = p_1$ and denote reserves of Token X and Token Y as a function price by $R_x(S)$ and $R_y(S)$.

Further, we see that the portfolio value for a LP can be written in terms of the price as well, i.e., by the composition $V(\boldsymbol{R}(\boldsymbol{P}))$. We may, with abuse of notation, just put $V(\boldsymbol{P})$ or $V(S)$ in the case of two tokens. Note, that while the relationship of trading functions to portfolio value is simple, the inverse is much less direct and requires the use of convex optimization. This is exactly the idea of RMMs studied in [4].

Next, let us discuss LPs in more depth. LPs tender assets to a pool in exchange they receive a quantity of LPTs representing their share of pool ownership. The LPT can always be exchanged for reserve assets if an LP wishes to exit their position. Upon exiting, an LP receives a basket $\boldsymbol{\Delta'}$ for their LPT. Their share is defined by their initial tendered basket $\boldsymbol{\Delta}$ and the accumulated swap fees. We refer to the actions an LP performs as ***liquidity changes***.

Swappers and LPs have a symbiotic relationship so long as $\gamma \neq 1$. For any swap $(\boldsymbol{\Delta}, \boldsymbol{\Delta'})$ there will be $(1-\gamma)\boldsymbol{\Delta}$ placed into the pool which increases the amount of the underlying tokens that an LPT is worth. Also, in CFMMs, the reported price is updated through arbitrage (informed swaps). This mechanism finances the portfolio for certain RMM positions. For more information, see Appendix A and [5, 9].

A LP executes a trade of the form $(\boldsymbol{\Delta}, 0)$ or $(0, \boldsymbol{\Delta'})$ that does not modify the pool's price. Specifically, LPs must provide or remove liquidity along directions that preserve the gradient of the trading function.

Definition A.4. A *valid liquidity change* is a trade $(\boldsymbol{\Delta}, 0)$ or $(0, \boldsymbol{\Delta'})$ such that

$$p(\boldsymbol{R}) = p(\boldsymbol{R}+\boldsymbol{\Delta}) = p(\boldsymbol{R}-\boldsymbol{\Delta'}) \tag{10}$$

Since the LP maintains a constant share of the pool, the LPT is always worth some amount of reserve assets. Therefore, we can use Equation (9) to define the value of the LPT as a function of the reserves corresponding to their amount of LPTs.

Example A.1. As a brief example, a limit order can be built as a collection of 2-token CFMMs with a trading function called a *constant sum trading function*. We define a generic constant sum trading function by

$$\varphi_{\text{cs}}(\mathbf{R}) = P_x R_x + P_y R_y = \boldsymbol{P}^T\boldsymbol{R}, \tag{11}$$

such that $P_x, P_y \in \mathbb{R}_+$. The constant sum trading function dictates that valid swaps only execute at a single reported price which is akin to a limit order. The constants P_1 and P_2 define the price S_0 by

$$S_0 := \frac{\partial_1\varphi_{\text{cs}}}{\partial_2\varphi_{\text{cs}}} = \frac{P_1}{P_2} \tag{12}$$

where ∂_i is the partial derivative with respect to the i^{th} reserve.

Assuming competitive arbitrage, valid liquidity changes will add only Token X or Token Y to the pool depending on some external market price S. If $S > S_0$, then the LP will add Token X to the pool and if $S < S_0$ they will add Token Y. The case where $S = S_0$ would not be long lasting, as active trading will move the price S.

A.2 Replicating Portfolios and RMMs

Replicating portfolios have been a subject of study in quantitative finance for some time [8]. Due to the autonomous nature of DEXs, portfolio replication can be done natively with CFMMs. RMMs introduced the ability to map from some target portfolio value function to an associated trading function. This mapping is achieved via convex optimization and the solution maps portfolio value functions to trading functions that replicate this portfolio to their LPs [5]. The RMM result yields a robust framework to constructing replicating portfolios on-chain.

Formally, given any concave, non-negative, non-decreasing portfolio value function V, there exists a CFMM trading function φ_V. The problem for φ_V is to solve the following optimization problem:

$$\varphi_V(\boldsymbol{R}) = \inf_c \left(c^\top\boldsymbol{R} - V(\boldsymbol{R})\right). \tag{13}$$

The above provides a realization of a map $V \mapsto \varphi_V$. To simplify notation, we will denote this mapping by $\text{rmm}(V) = \varphi_V$ and note that this association is Fenchel conjugacy. The inverse mapping to the rmm was given earlier by Definition A.3, i.e.,

$$V = \nabla\varphi^\top. \tag{14}$$

A.2.1 RMM-CC

One example of a portfolio value function is the Black-Scholes value function for a European call option between a pair of assets. If we denote the Black-Scholes value function by V_{CC}, then the associated trading function is given by

$$\varphi_{V_{\mathrm{CC}}}(R_x, R_y) = R_y - K\Phi(\Phi^{-1}(1 - R_x) - \sigma\sqrt{\tau}) = k \quad (15)$$

where Φ is the cumulative distribution function (CDF) of the standard normal distribution, σ is the volatility of the underlying asset, τ is the time to maturity, and K is the strike price of the option. We refer to the above trading function as ***RMM-CC***. Note that RMM-CC was originally defined in [4]. The invariant given by the value k can be interpreted as the pool's replication status. If $k = 0$, then the pool is perfectly replicating the covered-call payoff. The case $k < 0$ and $k > 0$ refer to under and over replication of the covered-call payoff, respectively.

Looking at Equation (15) we can note that $R_x \in [0, 1]$ and $R_y \in [0, K]$ when $k = 0$. In order to allow for an unbounded amount of liquidity to enter the pool, we must normaliza the total X reserves to be in the range $[0, 1]$. That is, we must normalize the pool reserves at each deposit/remove event.

To perform normalization we must know how much of each asset each LPT contains. For an existing pool this is simply looking at the pre-existing normalized quantities. At the start of a pool's creation, we must specify the marginal price of the curve. This, in turn, sets the quantity of each asset an LPT should represent, call these quantities x_0 and y_0.

If there are N shares already in the pool and we seek to add/remove enough such that there would be N' shares left, we must renormalize reserves again:

$$\frac{R_x}{N} = \frac{R'_x}{N'} = R_{x_0} \quad (16)$$

where R_{x_0} is the normalized quantity of X that each LPT should represent. These normalized quantities are used for all calculations in place of the raw reserve quantities. We can visualize fractional shares of the RMM-CC LPT by Figure 1. For the rest of the derivations for simplicity, however, we will act as though there is only 1 LPT in the pool. Next we turn to the implied marginal pricing function.

It is important to remark that the RMM-CC trading function changes over time. In the implementation, the time update is applied only when swaps occur. This time variance of the trading function is necessary for the portfolio value to change just as the option would when it nears expiry. We can see this graphically in Figure 2.

Using Definition A.3 we can get the price of the RMM-CC pool as

$$S(R_x) = Ke^{\Phi^{-1}(1-R_x)\sigma\sqrt{\tau} - \frac{\sigma^2\tau}{2}} \quad (17)$$

Note that R_y is not needed to compute the price of the pool as R_y can be implicitly computed from R_x and the invariant. Also, while this price is supported on R^+, prices near the extremes of the scale are practically unachieavable due to the nature of the dependence on the standard normal distribution. In a sense, there is drastically reduced trade liquidity near the extremes, so while in theory it is possible to trade there, it is not practical especially in the case of discrete prices.

Once again, we can use Definition A.3 to see that the portfolio value associated to $\varphi_{V_{\mathrm{CC}}}$ is that of a Black–Scholes covered call. Upon rearranging and isolating for $R_x(S)$ and in turn $R_y(S)$, we obtain the portfolio value

$$V(S) = S(1 - \Phi(d_1)) + K\Phi(d_2) + k \quad (18)$$

where

$$d_1 = \frac{\log(S/K) + (\sigma^2/2)\tau}{\sigma\sqrt{\tau}} \quad (19)$$

$$d_2 = d_1 - \sigma\sqrt{\tau}. \quad (20)$$

The difference between V and the Black–Scholes covered call value is exactly invariant term k. Hence why we refer to perfect portfolio replication as $k = 0$. However, the portfolio of a covered call is not self-financing (requires external funding to pay the equivalent of a call premium over time). This means without an additional income structure, we will see a decrease in k over time by exactly the value of the accumulated call premium by expiry.

One means for closing this replication error would be to implement a swap fee on the trading function. As time til expiry decreases, the pool will report decreasing prices as well. Hence, the pool will incentivize arbitrage even if reference markets prices are static. Primitive did extensive work using Python simulations to show that a properly optimized swap fee was sufficient in probabilistically reducing this error [10]. These results are based on a minimum expected trade volume and lead to a wide range of potential fees based on this assumption. We do, however, see an outperformance in some cases where observed trade volume exceeds expectation.

Being an onchain implementation of a covered call with minimal trust dependencies, we can use the LPT as a robust building block for other replicating portfolios. As shown in [10], we can use our LPT to achieve everything from long options, to binaries, to more complicated structured products such as liquidation-free lending. These action are possible with other CFMM implementations as well, however, due to the clean and bounded nature of a covered call portfolio, RMM-CC serves as a simpler building block for now.

A.3 Liquidity

There is a large body of research around the connection between CFMMs and the portfolio value of their LPTs, but there is less written on the connection with the order books or liquidity distributions. This fact explains the emphasis on using CFMMs to achieving a specific portfolio value. However, we can, at least in the case of bounded payoffs, think of CFMMs as a method to dynamicaly allocate concentrated liquidity. UniswapV3 is a good example of this, where the liquidity is concentrated over discrete price ticks.

Let us define a notion of liquidity based on trade depth that allows us to allocate to CFMMs based on our desired distribution. This functionality allows one to passively imple-

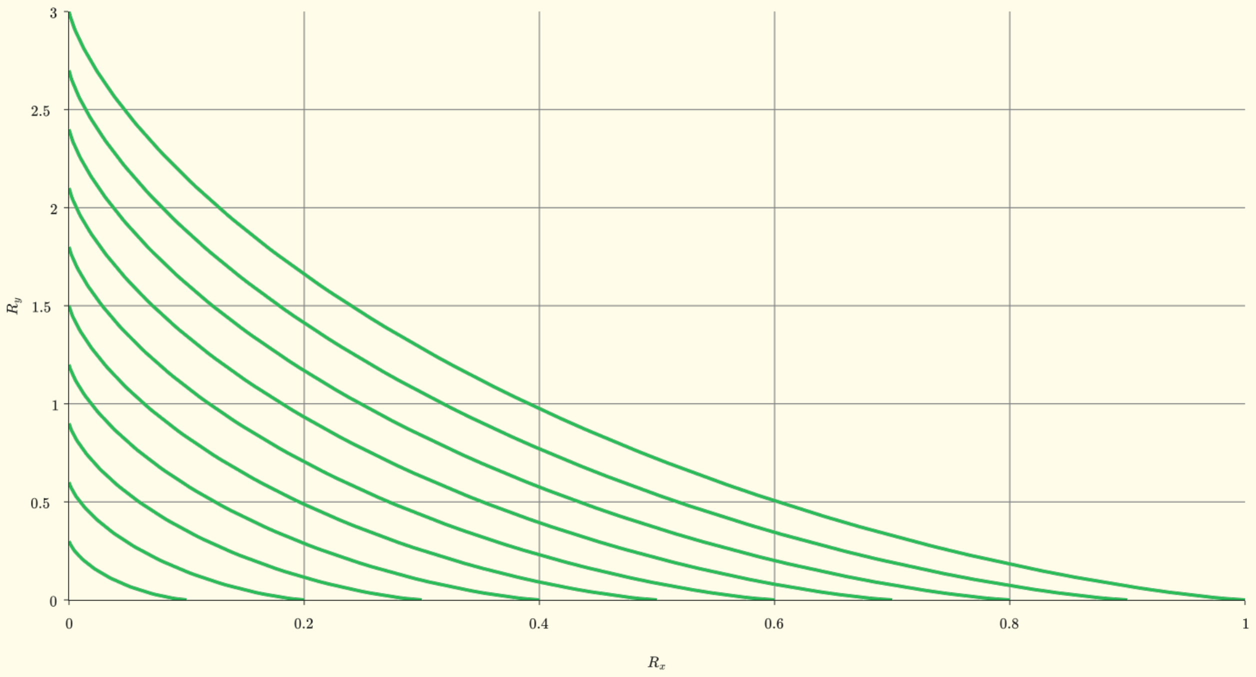

Figure 1: Fractional liquidity positions represented as curves.

ment an expectation on price action and earn fee income accordingly, with minimal need for manual re-balance events requiring action. For now, we will only consider the two token case, but similar same logic applies to $n > 2$ tokens. It will also be possible to convert from CFMM to liquidity distributions.

First, we can define ***depth*** as the swap quantity required to move the price of the pool up dp. Intuitively, we can think of depth in this case as a density of open orders in an continuous limit order book. We will assume that we are tendering an amount Δ_y and receiving Δ_x in order to change the price. Note that we can always determine a $\Delta_x(S)$ due to the convexity of φ. Assuming this is a sufficiently smooth trading function, we can define our depth as

$$l = \frac{d\Delta_x}{dS} \tag{21}$$

This definition is a measure of trade size required to move the price upwards by dS We can recover the R_x reserve by integrating:

$$R_x(S) = R_{x_0} - \Delta_x = R_{x_0} - \int_{S_0}^{S} l(p)dp \tag{22}$$

where $S_0 = S(R_{x_0})$. We can now apply [7, Lemma 1], which states that $V'(S) = R_x(S)$. Thus to recover a portfolio value function, we need to integrate one more time:

$$V(S) = \int_{S_0}^{S} \left(R_{x_0} - \int_{S_0}^{w} l(p)dp \right) dw \tag{23}$$

Given [5], we can recover the trading function via $\varphi = \mathrm{rmm}(V)$.

Note this construction works in reverse too. That is, given a trading function one can determine the associated liquidity distribution. Its worth mentioning, however, that this definition of liquidity has differing implications depending on the type of curve or distribution being represented. For example, a trading function with an finite total depth L given by

$$L = \int_{\mathbb{R}_+} l(S)dS \tag{24}$$

will have a bounded portfolio value function. Hence, trading functions with finite depth cannot support unbounded portfolio value functions. Whereas vice versa also applies: if the total depth is infinite, the portfolio value function will be unbounded. An example of bounded and unbounded payoffs are RMM-CC and the geometric mean market maker, respectively.

Example A.2. Note that the portfolio value for perfect replication of RMM-CC is bounded above by strike price K times the number of LPTs (i.e., the number of covered calls being held). To compute l, we need trade size in terms of price. Using Equation (17), we can determine $\Delta_x(S)$ as the inverse of this function given some initial reserve quantity R_{x_0}. Doing this yields:

$$\Delta_x = R_{x_0} + \Phi\left(\frac{\log\frac{S}{K} + \frac{1}{2}\sigma^2\tau}{\sigma\sqrt{\tau}}\right) - 1 \tag{25}$$

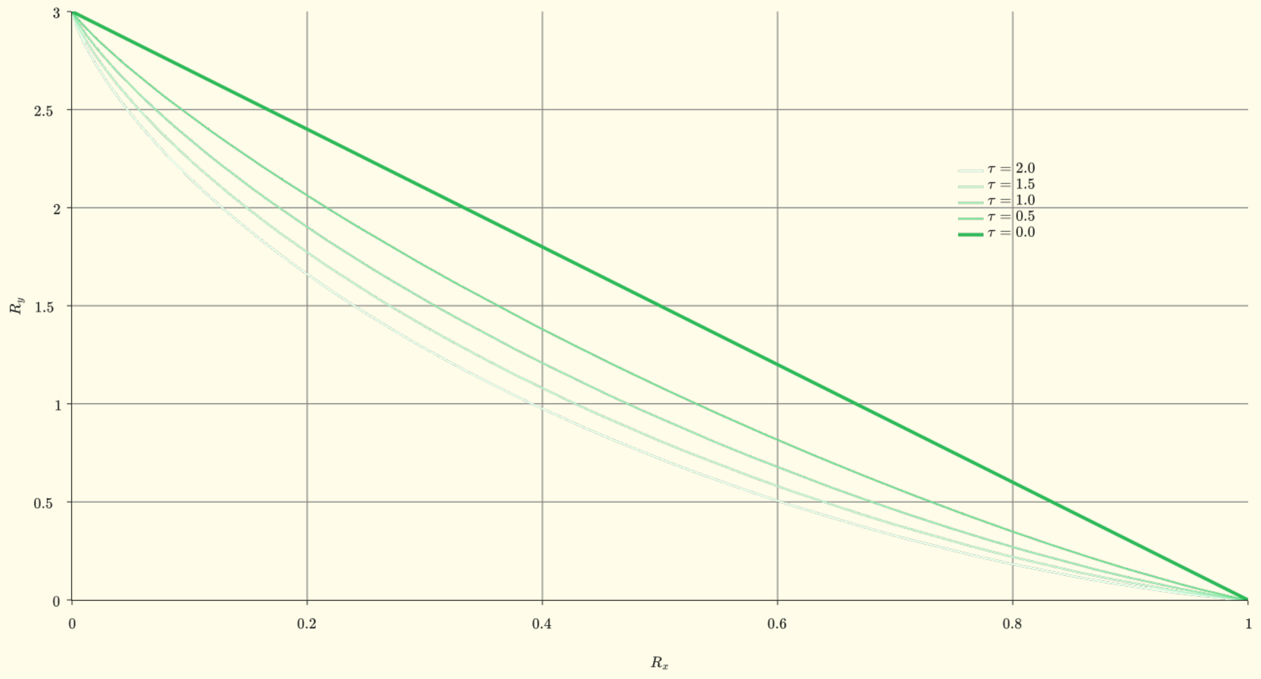

Figure 2: Time changing trading curve for RMM-CC with $K = 3$ and $\sigma = 1/2$.

To get the liquidity distribution we differentiate to get

$$l(S,\tau) = \frac{1}{S\sigma\sqrt{\tau}}\phi\left(\frac{\log\frac{S}{K} + \frac{1}{2}\sigma^2\tau}{\sigma\sqrt{\tau}}\right) \tag{26}$$

This can be seen in Figure 3 for various different times til expiry. Also, the liquidity distribution happens to be the absolute value of the covered call's payoff Greek Gamma. Note that the total depth L is finite and, moreover, $L = 1$. In the limit as $\tau \to \infty$, we have that $l(S,\tau) \to \delta(S-K)$ where δ is the Dirac delta distribution. That is, $l(S,\tau)$ is a Delta sequence.

Let us examine this further. Suppose we have L limit orders for a token X at a price K. Intuitively, we can model this as providing liquidity at a single price point, i.e., a constant sum trading function. The corresponding distrubition is then given by $L\delta(S-S_0)$. Applying D^{-1} will yield the payoff of a limit order of size L (or covered call at expiry).

We get D^{-1} by solving the following boundary value problem

$$\begin{cases} DV = \mu & S \in \mathbb{R}_+ \\ V(0) = 0 & \frac{d}{dS}V(0) = 0. \end{cases} \tag{27}$$

For illustration, we will break our specific problem with $\mu = M\delta(S-S_0)$ into two boundary value problems that we solve consecutively.

Solving the first problem we will recover a function we denote by Δ which you can think of as the Greek Delta of the portfolio value we have

$$\begin{cases} -\frac{d\Delta}{dS} = L\delta(S-K) \\ \Delta(0) = 0 \end{cases} \tag{28}$$

Note, Δ in this example is not meant to collide with the notation of the tendered basket! This problem is solved by integrating to get

$$\Delta(S) = -\int_{\mathbb{R}_+} L\delta(S-S_0)dS = \begin{cases} C & S < S_0 \\ C - L & S \geq S_0 \end{cases} \tag{29}$$

and then using our boundary condition $\Delta(0) = 1$ to get $C = L$ and thus

$$\Delta(S) = \begin{cases} L & S < S_0 \\ 0 & S \geq S_0 \end{cases} \tag{30}$$

Next, we integrate another boundary value problem

$$\frac{dV}{dS} = \Delta(S) \tag{31}$$

$$V(0) = 0. \tag{32}$$

Using the process above, we would get

$$V(S) = \begin{cases} LS & S \leq S_0 \\ LS_0 & S > S_0 \end{cases} \tag{33}$$

Below in Figure 4 we plot the portfolio value of a limit order of size L at price K.

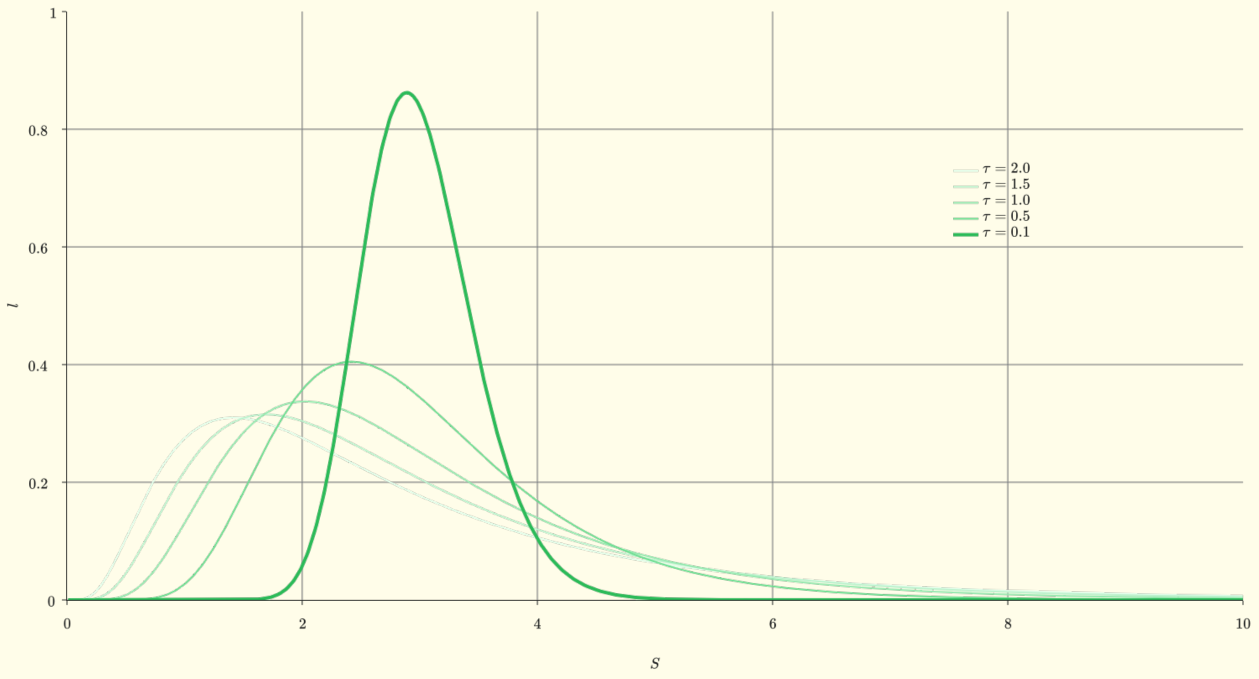

Figure 3: Time varying liquidity distribution for RMM-CC with $K = 3$ and $\sigma = 1/2$.

Furthermore, we can quickly see that

$$M = \int_{\mathbb{R}_+} M\delta(S - S_0)dS \tag{34}$$

Notice that for non-singular distributions (the Dirac delta being singular), there will be no open orders at any S, but only over a range of S values (this is a property of continuous measures).

By following similar work as in [5], we will find that the corresponding trading curve to V is given by

$$\varphi_V(S) = R_x + \frac{1}{K}R_y = L. \tag{35}$$

Visually, we can see this constant sum market as the $\tau = 0$ case in Figure 2.

Example A.2 shows us that there exists a "commutative diagram" for compactly supported trading functions, bounded sublinear portfolio value functions, and finite mass liquidity distributions in the case of two tokens. This diagram is given by Figure 5.

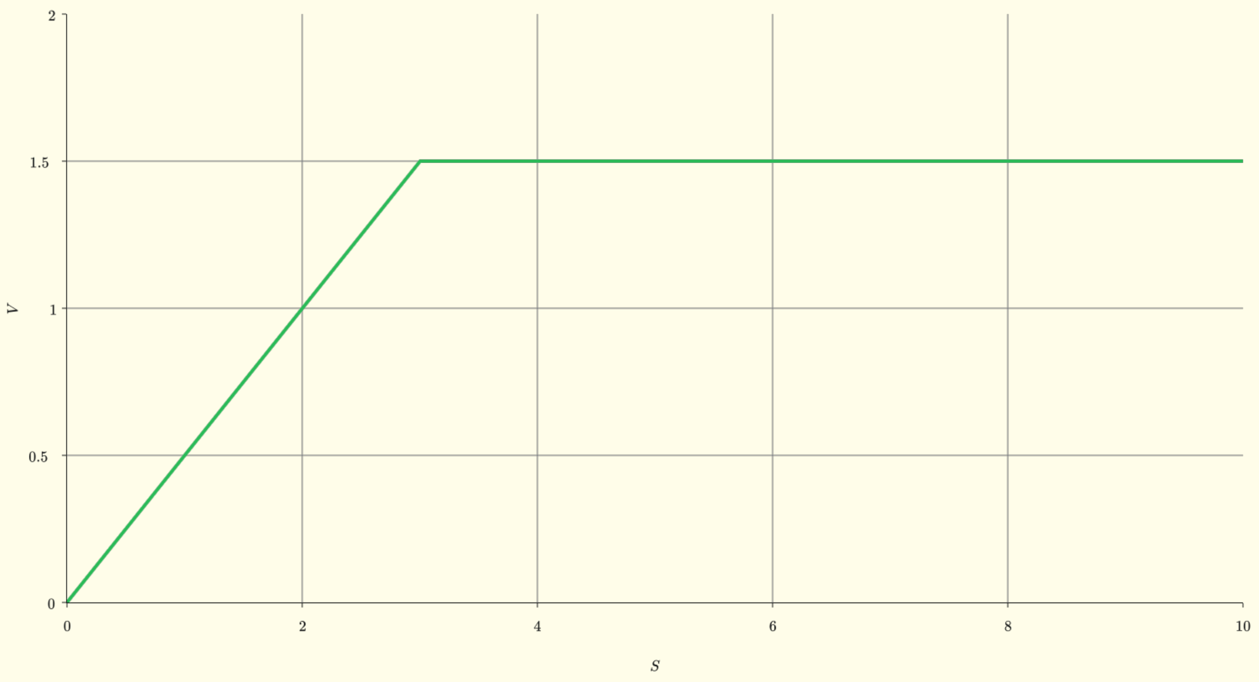

Figure 4: The payoff of a limit order of size $L = 0.5$ at price $K = 3$.

$$\begin{tikzcd}
 & \text{Liquidity Distribution} \arrow[dl, bend right, "D^{-1}"'] \arrow[dr, dashed] & \\
\text{Payoffs} \arrow[ur, "D"] \arrow[rr, bend right, "\mathrm{rmm}"'] & & \text{Trading Functions} \arrow[ll, "V(\boldsymbol{R})"] \arrow[ul, dashed, bend right]
\end{tikzcd}$$

Figure 5: The diagram showing the relationships between liquidity distributions, trading functions, and portfolio value functions. The dotted arrows imply maps via composition.

Glossary

layer-2 Layer two protocols leverage the security of a layer one network to offer more transaction throughput at a lower cost.

RMM-CC Primitive's implementation of a CFMM that replicates the payoff of a covered call, built on the FVM.

Acronyms

AMM Automated Market Maker

CFMM Constant Function Market Maker

DeFi Decentralized Finance

DEX Decentralized Exchange

EVM Ethereum Virtual Machine

FSM Finite State Machine

FVM Financial Virtual Machine

ISA Instruction Set Architecture

LP Liquidity Provider

LPT Liquidity Provider Token

RMM Replicating Market Maker

VM Virtual Machine