"Trading Mechanisms in an Adversarial Economy" by Daniel Engel,
Ph.D., Brown University, May, 2023.

Decentralized finance (or DeFi) has become a booming area of applied distributed computing. Over the course of the last several years, many mechanisms and distributed protocols have emerged as the backend for new financial applications in the DeFi setting.

One of the most popular mechanisms, an automated market maker (AMM), is an automaton that has custody of several pools of assets, and is capable of trading those assets with clients at rates set by a mathematical formula. Unlike traditional "order book" traders, AMMs trade directly with clients, and do not need to match up (and wait for) compatible buyers and sellers. Today, AMMs such as Uniswap, Bancor, and others have become one of the most popular ways to trade electronic assets on a single blockchain.

Additionally, for assets managed across distinct blockhains (e.g. trading ETH for BTC), many cross-chain protocols have been proposed to synchronize multiple blockchain to ensure reasonable trade outcomes for parties involved. These have become the main vehicles for mutually distrusting parties to seamlessly exchange assets between blockchains.

The goal of this thesis is to develop a mathematical theory that can serve as a guide to understand to what extent AMMs and cross-chain protocols can be used to solve various trade problems arising in the decentralized financial setting.

This thesis introduces: (1) An axiomatic characterization of desirable properties any AMM should satisfy. (2) The first definition of what it means to compose AMMs in an appropriate way, illustrating both the expressive power and limitations of networks of AMMs. (3) A discussion of the tradeoffs an AMM design imposes on both the traders and AMM investors, or liquidity providers. (4) Several novel cross-chain protocols for trading financial options. (5) A consensus hierarchy for the cross-chain synchronization setting.

Trading Mechanisms in an Adversarial Economy

by

Daniel Engel

B.S., Georgia Institute of Technology, 2017

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2023

This dissertation by Daniel Engel is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date: _____

Maurice Herlihy, Director

Recommended to the Graduate Council

Date: _____

Amy Greenwald, Reader
Brown University

Date: _____

Seny Kamara, Reader
Brown University

Approved to the Graduate Council

Date: _____

Thomas A. Lewis
Dean of the Graduate School

Acknowledgements

First of all, I would like to thank my advisor, Maurice Herlihy. Maurice has been supportive and helped me develop into a researcher. His advising style in combination with his ability to identify research problems with interesting mathematical structure has made it a pleasure to work with him.

I am also thankful to my other committee members Amy Greenwald and Seny Kamara for serving on my dissertation committee. Learning mechanism design from Amy and cryptography from Seny helped to shape the perspective of this thesis. I also owe many thanks to my collaborators Yingjie Xue and Sucharita Jayanti. The debates sparked in our research meetings provided the necessary fuel to keep projects alive and interesting.

The computer science department's staff has also played a huge part in helping make this thesis come together. I want to thank Lori Agresti, Lauren Clarke, Genie DeGouveia, Jane Martin, Jane McIlmail, Jesse Polhemus, and Dawn Reed for their work that allowed me to spend most of my time just focusing on research. I'd also like to thank the other grad students who helped me initially build a social circle in the department through our GCB nights, namely Nediyana Daskalova, Vikram Saraph, Ghous Amjad, Archita Agarwal, Marylin George, and Michael Markovitch.

I wouldn't have gotten very far in my academic journey if it wasn't for my family. In particular I'd like to thank my sister Jessica and her husband Gio for their patience, while I attempted to explain my work. Additionally, my mom, who very early on helped foster my love for puzzles. My growth as a person I owe to my friends. I am grateful for Nathaniel and Dylan, some of the most emotionally supportive friends I've ever had. Many thanks to Marina and Matteo, for their source of joy and intensity on and off the dance floor. To Beckett, for their companionship, empathy, and their positive influence on all of my relationships with others.

Additionally, I'm thankful for all of the miscellaneous others I interacted with while raving and skating around New England. These activities and the shared sense of community they have provided, have helped me feel grounded and restored after many instances of feeling burnt out.

I couldn't have done this without all of these folks.

Contents

1 Introduction **1**

1.1 DeFi Model . . . 1

1.2 Overview . . . 2

1.2.1 AMMs vs Traditional order books . . . 2

1.2.2 Combining AMMs . . . 3

1.2.3 Tradeoffs between AMMs . . . 3

1.2.4 Characteristics of 2D AMMs . . . 4

1.2.5 Cross-Chain Protocols . . . 4

1.2.6 Cross-Chain Limitations . . . 5

1.3 Related Work . . . 6

2 An AMM Model **9**

2.1 Mathematical Background . . . 9

2.2 Motivation . . . 11

2.3 Traders and Arbitrageurs . . . 12

2.3.1 Informal 2D Example . . . 12

2.3.2 Trader Requirements . . . 13

2.3.3 Provider Requirements . . . 14

2.3.4 AMM Definition . . . 15

2.4 AMMs as Manifolds . . . 18

2.4.1 Topological Manifolds . . . 18

2.4.2 Differentiable AMMs . . . 19

3 AMM composition **22**

3.1 Operations . . . 22

3.1.1 Projection . . . 22

3.1.2 Virtualizing Assets . . . 23

3.2 Sequential Composition . . . 26

3.2.1 One-to-One Composition . . . 26
3.2.2 Many-to-One Composition . . . 28
3.2.3 Many-to-Many Composition . . . 30
3.3 Parallel Composition . . . 31

4 Comparing AMM Designs **35**
4.1 AMM Measures . . . 35
4.1.1 AMM Capitalization . . . 35
4.1.2 Divergence Loss . . . 40
4.1.3 Linear Slippage . . . 44
4.1.4 Angular Slippage . . . 45
4.1.5 Load . . . 46
4.2 AMM Measures under Composition . . . 47
4.2.1 Sequential Composition . . . 47
4.2.2 Parallel Composition . . . 50

5 Characteristics of 2D AMMs **54**
5.1 The AMM-Capital Correspondence Theorem . . . 54
5.2 Special AMM Families . . . 58
5.2.1 Beta Space . . . 58
5.2.2 Symmetric AMMs . . . 60
5.2.3 Other Miscellaneous AMMs . . . 62
5.2.4 Example: Skewed constant-product . . . 63
5.2.5 Example: Log-Product . . . 63
5.3 Dynamic AMMs . . . 64
5.3.1 Valuation Change . . . 64
5.3.2 Distribution Change . . . 65
5.3.3 Expected Load . . . 65
5.3.4 Expected Divergence Loss . . . 66
5.3.5 Example: Constant-Product . . . 68
5.3.6 Optimal AMM . . . 69

6 Cross-Chain Protocols **71**
6.1 Motivation and Synchronization Primitives . . . 72
6.1.1 Hashlocks . . . 74
6.1.2 Path Signatures . . . 75
6.1.3 Cross-Chain Proofs . . . 75

6.2 A Simple Swap Protocol . . . 77
6.3 Layered Protocols . . . 78
6.3.1 Two-Party Transferable Swap . . . 80
6.3.2 Transfer Leader Position . . . 82
6.3.3 Transfer Follower Position . . . 87
6.3.4 Handling Multiple Candidates . . . 87
6.3.5 Security Properties . . . 89
6.3.6 Leader Transfer Properties . . . 90
6.3.7 Follower Transfer Properties . . . 91
6.3.8 Leader Transfer (Multiple Buyers) Properties . . . 92

7 Cross-Chain Limits **94**
7.1 A Formal Cross-chain Model . . . 94
7.1.1 Weak and Strong Consensus . . . 94
7.2 Hashlock Limitations . . . 96
7.3 Path-signature Lower Bound . . . 102
7.4 A Cross-Chain Consensus Hierarchy . . . 103
7.4.1 Two Parties . . . 103
7.4.2 Three or More Parties . . . 104

A Mathematica Code **111**

B Contract Code **113**

C Protocol Security Proofs **127**
C.0.1 Misc Proofs . . . 127
C.0.2 No UNDERWATER . . . 130

List of Figures

4.1 Balanced Capitalization of AMMs $(x, 1/x)$ (symmetric, blue) and $(x, 1/x^2)$ (asymmetric, orange) at stable states. 36

4.2 Angular vs linear slippage 45

5.1 $(\alpha, \beta) \in (1,2) \times (1,2)$ that correspond to the AMMs in $\tilde{H}^{-1}(Beta(\Delta^1; (1,2) \times (1,2)))$ 59

5.2 $\beta(\alpha) = 3 - \alpha$ line corresponding to the Balancer (monomial) family of AMMs in $\tilde{H}^{-1}(Beta(\Delta^1; (1,2) \times (1,2)))$ 60

5.3 The $\beta(\alpha) = \alpha$ line corresponding to all the symmetric AMMs in $\tilde{H}^{-1}(B(\Delta^1; (1,2) \times (1,2)))$ 62

5.4 The point of intersection of the $\beta(\alpha) = \alpha$ line and the $\beta(\alpha) = 3 - \alpha$ line corresponds to the constant-product AMM $xy = 1$. Namely it is contained in the space of symmetric AMMs and the beta space. . 62

5.5 Overview of some subspaces of 2D AMMs and their relationships. 63

5.6 Expected load for $A := (x, 1/x)$ at $1/2$ as a function of Beta distribution parameters. (Mathematica source is shown in Appendix A.) 66

6.1 In the figure, $x.Function$ means party x calls $Function$ in the contract. E.g. $Alice.MutSwapAB$ means Alice creates the swap contract AB and escrows her assets. The blue arrow depicts contract AB and the green arrow depicts contract BA and the orange one depicts contract CA. The text above each arrow depicts the state of the contract. For example, $swap = h_{A_1}, receiver = Alice, mutating = false$ means $swap\_hashlock = h_{A_1}$, and Alice can claim the asset in the contract if she provides pre-image of h_{A_1}. Here, *replace* is short for *replace_hashlock*. 84

6.2 A protocol execution demonstrating why Bob needs two more *replaceLeader()* rounds than Carol. If not, then he lacks full Δ to call *replaceLeader()* on BA after Carol calls it on AB. Here we assume Alice has reported consistent signatures in the *Mutate Lock Phase* but has reported them a Δ apart. Colored blocks represent time periods when functions can be called. The *Leader* suffix is excluded from the function calls for simplicity. 86

7.1 n-cast implementation 98

7.2 (k, n)-cast implementation 99

7.3 K_3: Smallest graph for which (k, n)-broadcast fails. In this case $n = 6$, $1 \leq k < 6$. 99

7.4 Partition of graphs in SC_n that can be solved with (k, n)-broadcast where $1 \leq k < n$ and $n = m(m-1)$.102

7.5 Illustration of construction for Lemma 7.4.1, for $k = 1$ alternating path. 105

C.1 State transition until Carol replaces Alice on AB contract 131

Chapter 1

Introduction

In *distributed systems*, the actor of interest is the process or node. Processes communicate via messages over a shared network to solve some kind of coordination problem such as agreement. A set of rules for each process to follow to solve such a problem, is called a *protocol*. Occasionally nodes or the network can *fail*. The primary node failure modes considered are simple crash failures and byzantine failures (where nodes start sending arbitrary messages).

In classical *microeconomics*, the fundamental unit is the economic agent, with needs and desires. As external market forces change, agents interact with each other via the marketplace to meet their respective preferences. Ultimately, the goal of the economic mechanism is to allocate goods and services to meet the demands of the economic agents.

Decentralized finance (DeFi) is an emergent research area where both the economic and distributed computing perspectives can be useful. The marketplace itself is now an autonomous distributed system. Economic agents now have preferences over the the final states of the distributed system. Agents participate in distributed protocols to send and receive goods in the marketplace. The goal of this thesis is to develop a mathematical theory to understand the capabilities and limitations of various DeFi primitives that have emerged as the foundation for many blockchain applications.

1.1 DeFi Model

DeFi is typically used in reference to financial services built on top of blockchain infrastructure. This thesis is framed more abstractly. The ideal service that a blockchain provides is a trusted state machine, though for our purposes the blockchain data structure itself and associated consensus mechanism are irrelevant. Namely, the questions raised here have very little to do with the implementation details of blockchain technology itself [1]. Instead, they are really about the scientific and engineering problems of safely transferring value among autonomous distrusting parties. This problem will remain of enduring importance to society, independently of whether particular blockchain technologies

[1] In the parlance of blockchain technology, these are often called *layer two* protocols because they are built on top of the main blockchain consensus layer (*layer one*).

bloom or fade, whether certain asset bubbles expand or pop, or whether regulatory agencies do or do not intervene to protect gullible investors.

To make this generality clear, we will mostly refer to blockchains as nodes. We assume these nodes are themselves myopic and only have a view of their own local state. On the other hand, agents or parties have a global view of the state of all nodes, which we assume is publicly visible. Nodes can only update their state in response to external calls made by agents. We will interchangeably refer to these calls as *messages* or *transactions* whenever convenient. All agents are assumed to be able to at any moment, send messages to a node and not be censored or prohibited to do so by other agents.

Importantly, we assume agents are self-interested and act in accordance with a personal utility function as is standard in game theory and mechanism design [54]. Thus, coordination tasks involving multiple self-interested agents only really make sense when their incentives are aligned with a shared outcome. Otherwise, there is no incentive to participate in achieving a collective decision.

1.2 Overview

1.2.1 AMMs vs Traditional order books

A traditional limit order book exchange maintains a list of buy and sell orders for particular assets. The order book is typically managed by a so called *market maker*. Periodically, the market maker clears entries from the order book by matching buy and sell orders. Generally, a market maker is compensated a small commission for providing this service. Order books are associated with traditional finance markets and are the standards for publicly traded assets like stocks, bonds, options, etc [1].

Order books are synchronous by nature. A trader submits a buy or sell, and must wait some amount of time until the market maker completes the trade. This might be undesirable for a trader wanting to make an effectively instantaneous trade at the current market exchange rate. Additionally, the problem of the market maker finding the optimal matching between buy and sell orders is often a computationally intractable problem. Hence, traders can suffer as a result of not getting an optimal trade at the current market rate.

An *automated market maker* (AMM) is an alternative trading mechanism first proposed in the context of DeFi. It is a deterministic automaton that maintains custody of several pools of assets, and is capable of trading those assets with traders at rates set by a mathematical formula. Unlike traditional order books, AMMs trade directly with traders, and do not need to match up (and wait for) compatible buyers and sellers. AMMs are appealing to traders because their trades effectively execute instantaneously and at the publicly viewable rate that the AMM specifies [2] This is in contrast to alternative mechanisms such as sealed-bid auctions, where the price of the auctioned asset is computed once all parties have committed their payments to the auctioneer.

[2]In practice, this is only partly true due to an attack called *frontrunning* [30]. This occurs when one party (say Alice) is able to execute her transaction before another party (say Bob), thereby forcing Bob to have trade at a slightly different exchange rate than he anticipated [20].

AMMs are not managed by an interactive market maker as in order books. Instead they are managed by a single trusted state machine (a node). Typically their initial asset pools are initialized by investors. After initialization, traders are free to execute an arbitrary number of trades without waiting for a market maker to accept or match the trade. Most AMMs used in practice prevent traders from entirely depleting asset reserves by appropriately adjusting the marginal exchange rate between the assets being traded.

Whenever a trader makes a trade, they are charged a small transaction fee which is then claimed by the investors at some future time. AMM investors are often called *liquidity providers*.

Today, there are many AMMs implemented across various blockchains. Typically trading on multiple AMMs happens on the same blockchain, however, with the assistance of *cross-chain protocols* it is possible to trade with multiple AMMs across distinct blockchains [62]. AMMs such as Uniswap [6], Bancor [42], and others have become some of the most popular ways to trade electronic assets.

In Chapter 2, we give an axiomatic treatment of AMMs. We define them in such a way to capture most existing AMM designs but also allow flexibility for future improvement.

1.2.2 Combining AMMs

In traditional financial markets, agents link together multiple financial transactions. For example, one could take out a bank loan which is then used to buy a stock. It is conceivable that both bank loans and stock shares are assets both managed by an AMM. It is then natural to try to understand exactly the effect routing multiple assets through a sequence of AMMs has on the final assets received. This type of behavior we define as *sequential composition* and is described in detail in Section 3.2.

Similarly, when faced with multiple alternatives for trading the same underlying assets, a trader might split one asset into multiple investments. Suppose a trader is faced with identical vendors who start with the same supply of goods at the same price per unit good. As the trader buys goods from the first vendor, it is expected that the first vendor will increase the price of its goods. The trader has the incentive to then start trading with the second vendor since it will now be offering the lower price. Similarly for AMMs, a trader might split some assets across AMMs that trade identical goods in order to get the best deal in the market. We refer to this as *parallel composition* and discuss its properties in Section 3.3.

1.2.3 Tradeoffs between AMMs

A central concept when studying trader behavior in traditional markets is *arbitrage*. If some vendor in a marketplace is selling oranges for 1 coin and say some other vendor is buying oranges for 3 coins, it is customary for a profiteering trader to buy the cheaper oranges and resell them at the higher price. The price discrepancy incentives traders to drain the supply of the cheaper vendor and fill the reserves of the more expensive vendor. The cheaper vendor will start charging more coins per orange and the expensive vendor will charge less. Classical economics suggests that these price adjusments will converge to some agreed upon *market price* or *market valuation* for the good.

In the contexts of AMMs, the loss each trader incurs as a result of price adjustments from a prior trade is known as *slippage*. Similarly, if the market price abruptly changes due to external factors or some traders acquire insider trading knowledge, the AMM current price prediction might become inaccurate. As a result, *arbitrageurs* will seize the trade opportunity, thereby exposing the AMM liquidity providers to a *divergence loss*.

In Chapter 4, we precisely define these measures and show how they can be used to compare AMM designs. Additionally, in Section 4.2 we show the relationship between our composition operators and how they relate to various AMM metrics.

1.2.4 Characteristics of 2D AMMs

The value of an AMMs holdings is known as its *capitalization*, a concept we introduce in Chapter 4. While initially used as a measure to understand effects on liquidity providers under market fluctuations, the capitalization has a much more fundamental relationship with an AMM. Namely, there is a precise sense in which a 2D AMM and its corresponding capitalization function are interchangeable representations of the same underlying mathematical object (Section 5.1).

Section 5.2 gives an overview of several interesting subclasses of 2D AMMs that are frequently used in practice. In doing so, the utility of the main result of Section 5.1 becomes clear: to understand restricted subspaces of AMMs, it is often easier to consider subspaces of their corresponding capitalization functions instead.

Finally, in Section 5.3 we describe multiple strategies for dynamically adjusting an AMM in response to changing market conditions. In particular we discuss a sense in which it makes sense to talk about an *optimal AMM* with respect to some distributional assumptions about the future behavior of the market.

1.2.5 Cross-Chain Protocols

The goal of Chapter 6 is to give an overview of cross-chain problems and present protocols for solving them. Section 6.1, gives an overview of the synchronization tools that are used to build typical cross-chain protocols.

Cross-chain protocols have been proposed to build traditional financial services such as: asset exchanges, loan providers, auctions, etc. Importantly the underlying assets in these problems can conceivably span multiple distinct nodes in a network.

One of the simplest type of cross-chain problems is called an *atomic swap* (asset exchange). For example, say two parties (Alice and Bob) want to trade two currencies: *guilders* for *florins* [3], but both currencies are managed on the distinct nodes A, B respectively. Because A and B cannot directly communicate with each other, they must rely on some sort of external interaction from the two trading parties to ensure the trade successfully completes. The issue is, that both A and B know at least one party is honest, but don't know whether it is Alice or Bob. A much more general version of this problem can be described by a directed graph, where each vertex is a party, and each edge is a distinct node managing a single asset. Protocols for solving these *directed graph tasks* are reviewed in Section 6.2.

[3] Florins and guilders are obsolete currencies, used here fantastically for the purpose of concreteness.

Consider a slightly related, but different problem. Suppose again, that Alice and Bob want to trade *guilders* and *florins* managed on nodes A, B respectively. Say they lock up their funds, or *escrow* them, at some point before the trade completes. Additionally, say that Alice has the ability choose whether all assets are traded (*commit*) or stay with their respective owners (*abort*). If Alice crashes (she never shows up to trade), then Bob has been tricked into locking up his funds until the trade aborts.

The distributed computing perspective would tell us these crashes are inescapable, and Bob has to contend with the potential risk that Alice might propose trades with him, only to abort at the last minute. However, the financial perspective tells us that Alice in fact has *optionality* to complete a trade with Bob. *Options* themselves are derivative assets that can be traded. Typically then, in this case, Bob would charge Alice a premium for the opportunity cost he loses for having his assets locked, thereby being unable to spend them elsewhere.

Section 6.3 embraces the fact that this scenario presents a legitimate options contract between Alice and Bob. Several protocols are then presented to allow both Alice and Bob to sell their *positions* in this options contract to other interested parties. Importantly, they are now trading their financial position as opposed to the underlying assets.

1.2.6 Cross-Chain Limitations

Cross-chain protocols rely on a handful of synchronization primitives to coordinate state changes across distinct nodes. These primitives serve as a way to coordinate behavior between transacting parties with veiled intentions. For example, an honest party could be transacting with a coalition of devious parties all of whom reveal their true intentions as the steps of a cross-chain protocol unfold. However, before a party executes any calls on a node, the node has no way of distinguishing that party as being honest or malicious.

Any reasonable cross-chain protocol should ensure that any honest parties don't end up worse after following the steps of the protocol. Additionally, if all parties do indeed intend to work towards a collective outcome, then any good cross-chain protocol better guarantee they can do so. We will see in Chapter 7, that these mild requirements lead to fundamental limitations on what classes of coordination problems can be solved with a given synchronization object.

Section 7.1 introduces a synchronous model of computation for studying the limitations of cross-chain coordination problems. In Section 7.2, we classify exactly which types of simple trading problems can be solved with a common sychronization tool called a *hashlock*. Section 7.3 gives a lower bound result for implementing a *reliable broadcast* from a primitive called *path signatures*. Finally, Section 7.4 introduces a *consensus hierarchy* for classifying coordination tasks based on their complexity. Additionally, it introduces a framework for comparing the synchronization power of cross-chain primitives. This is analogous to the well-known consensus hierarchy from multiprocessor synchronization.

1.3 Related Work

Angeris and Chitra [5] introduce a *constant function market maker* model and consider conditions that ensure that agents who interact with AMMs correctly report asset prices. Our work, based on a similar but not identical AMM model, focuses on properties such as defining AMM composition, AMM topology, and the role of stable points.

Alternative measures have been proposed to those we discuss in Chapter 4. In particular, a measure known as LVR (loss-versus-holding) or "lever" has more recently been proposed [47]. One way LVR can be understood is as an opportunity cost liquidity providers incur for locking their assets in an AMM, thereby limiting their optionality.

Uniswap [6, 63] is a family of constant-product AMMs that originally traded between ERC-20 tokens [32] and ether cryptocurrency. Trading between ERC-20 assets requires sequential composition of the kind analyzed in Section 3.2. Uniswap v2 [3] added direct trading between selected pairs of ERC-20 tokens, and Uniswap v3 [3] allows liquidity providers to restrict the range of prices in which their asset participate, giving rise to a form of parallel composition of the kind analyzed in Section 3.3. There are many are more examples of AMMs: Pourpouneh *et al.* [51] contains a survey.

Before there were AMMs for decentralized finance, there were AMMs for *event prediction markets*, where parties trade securities that pay a premium if and only if some event occurs within a specified time. A community of researchers has focused on prediction-market AMMs [2, 16, 15, 36, 37]. Despite superficial similarities, event prediction AMMs and security AMMs differ from DeFi AMMs in important ways: pricing models are different because prediction outcome spaces are discrete rather than continuous, prediction securities have finite lifetimes, and composition of AMMs is not a concern.

The optimal arbitrage problem considered in Chapter 2 is not new. A consumer choosing an optimal bundle of goods for a fixed set of prices is the same as an arbitrageur choosing an optimal point on an AMM with respect to a market valuation. This is known as the *expenditure minimization problem* [46]. While AMMs and consumer indifference surfaces are mathematically similar, they are different in application. In particular, traders interact with AMMs via composition, an issue that does not arise in the consumer model.

Cross-chain options have their origin in cross-chain atomic swap protocols. A cross-chain atomic swap enables two parties to exchange assets across different blockchains. An atomic swap is implemented via hashed timelock contracts(HTLC)[48]. There are a variety of protocols proposed [60, 7, 34, 31] and implemented [13, 22]. Herlihy *et al.* proposed protocols for atomic multi-party swaps [40] and more general atomic cross-chain deals [41].

Several researchers [35, 38, 44, 64] have noted that most two-party swap protocols effectively act as poorly-designed options [43], because one party has the power to decide whether to go through with the agreed-upon swap without compensation for its counterparty.

A number of proposals [35, 38, 44, 64, 24, 50, 59] address the problem of optionality in cross-chain atomic swaps by introducing some form of premium payment, where a party that chooses not to complete the swap pays a premium to the counterparty. Robinson [52] proposes to reduce the influence of optionality by splitting each swap into a sequence of very small swaps. Han *et al.* [35] quantified optionality unfairness in atomic swap using the Cox-Ross-Rubinstein

option pricing model [18], treating the atomic swap as an American-style option. The Black-Scholes (BS) Model [11] can be used to estimate the value of European-style options.

Liu [44] proposed an alternative approach where option providers are are paid up-front for providing optionality, as in the conventional options market. In this protocol, Alice explicitly purchases an option from Bob by paying him an nonrefundable premium. Tefagh *et al.* [55] proposed a similar protocol which enables Alice to deposit her principal later than Bob. None of these works have considered how to close an option owner's position by transferring that option to a third party.

There are protocols that allow blockchains to communicate (cross-chain proofs), however they either rely on external third parties [33] or their applicability requires the introduction of centralized services, modifications to existing software, and doesn't guarantee reliable message delivery

The use of HTLCs for two-party cross-chain swaps is generally attributed to Nolan [48]. HTLCs have adapted to several uses [10, 12, 21, 49]. Herlihy [40] extended HTLCs to support multi-party swaps on directed graphs.

Herlihy *et al.* [41] introduce the notion of *cross-chain deals*. They focus on how conventional notions of atomicity are inadequate for an adversarial environment, and give protocols using both HTLCs and a central coordinating blockchain. Zakhary *et al.* [61] propose a cross-chain swap protocol for proof-of-work blockchains using a *witness blockchain* as a central coordinator.

The BAR (byzantine, altruistic and rational) model [4, 17] supports cooperative services spanning autonomous administrative domains that are resilient to Byzantine and rational manipulations. BAR-tolerant systems assume a bounded number of Byzantine faults, and as such do not fit our adversarial model, where any number of parties may be Byzantine, rationally or not.

In finance, *optionality* [43] is the notion that there is value in acquiring the right, but not the obligation, to invest in something later. Atomic swap based on HTLCs exposes such optionality to both parties. However, multiple researchers [35, 38, 44] have observed that both parties are exposed to sore loser attacks where the counterparty reneges at critical points in the protocol. Robinson [52] proposes to reduce vulnerability to sore loser attacks by splitting each swap into a sequence of very small swaps, an approach that works only for fungible, divisible tokens.

Xue and Herlihy [59] show how to incorporate premiums into multi-party swaps, auctions, and brokered sales. Prior work was focused exclusively on two-party swaps, and proposed *asymmetric* protocols, meaning that only one party pays a premium to the other, protecting only that side of the swap from a sore loser attack. These protocols include Han *et al.* [35], Eizinger *et al.* [24], Liu [44], the Komodo platform [50], Eizinger et al. [24], and the Arwen protocols [38].

Xu *et al.* [58] analyze the success rate of cross-chain swaps using HTLCs. Liu [44] proposed an atomic swap protocol that protects both parties from sore loser attacks, structured so that Alice explicitly purchases an option from Bob, and her premium is never refunded. There is no obvious way to extend this protocol to applications other than two-party swaps. Tefagh *et al.* [55] propose a similar protocol based on an options model.

The contents of this thesis are drawn from various sources. Chapters 2-4 are based on joint work with Maurice Herlihy [25, 26],while Chapter 5 is based on ongoing work with Maurice Herlihy. Section 6.3 is based on joint work

with Yingjie Xue and Maurice Herlihy [29, 27]. Finally, Chapter 7 is derived from ongoing work with Sucharita Jayanti and Maurice Herlihy. As a result, some of the results are simply stated but the proofs are omitted. The proofs are to be included in the corresponding conference paper [28].

Chapter 2

An AMM Model

2.1 Mathematical Background

Given a set X, a relation on X is a subset of $X \times X$.

A *topological space* is a set X with a collection of subsets U of X such that

- $\emptyset, X \in U$
- U is closed under finite intersections
- U is closed under arbitrary unions

U is called a *topology*. Formally, the space is denoted (X, U) though U will often be omitted. A set is called *open* if it is contained in U. A set S is called *closed* if S^c is contained in U. The *closure* of a subset $S \subseteq X$ is the intersection of all closed sets containing S. The *interior* of a subset $S \subseteq X$ is the union of all open sets contained in S. We use the notation $int(S)$ to refer to the interior of S. The *exterior* of a subset $S \subseteq X$ is the complement of its closure. The *boundary* of a subset $S \subseteq X$ is the set of points in the closure of S that are not contained in the interior of S. We use ∂S to refer to the boundary of S. Given two topological spaces (X, U) and (Y, V) and a map $f : X \to Y$, we say f is *continuous* if $f^{-1}(\tilde{V}) \in U$ for all $\tilde{V} \in V$. Given two topological spaces (X, U) and (Y, V) and a map $f : X \to Y$, we say f is a *homeomorphism* if

- f is a bijection
- $f : X \to Y$ and $f^{-1} : Y \to X$ are both continuous

Unless otherwise mentioned, a bolded $\boldsymbol{x}$ denotes a vector. If $\boldsymbol{x} \in \mathbb{R}^n$ we will use x_i to denote the i-th component of $\boldsymbol{x}$. We will use $\mathbb{R}^n_{\geq 0} = \{\boldsymbol{x} \in \mathbb{R}^n : x_i \geq 0\}$. Similarly, we will use $\mathbb{R}^n_{++} = \{\boldsymbol{x} \in \mathbb{R}^n : x_i > 0\}$ and $\mathbb{R}^n_{--} = \{\boldsymbol{x} \in \mathbb{R}^n : x_i < 0\}$. Given $x, y \in \mathbb{R}^n$, we will use the notation $x << y$ to mean $x_i < y_i$ for all $i \in \{1, ..., n\}$.

A *norm* on a real vector space V is a function $|| \cdot || : V \to \mathbb{R}$ such that for all $\boldsymbol{v}, \boldsymbol{u} \in V$

- $||\boldsymbol{v}|| \geq 0$

- $||\alpha v|| = |\alpha|||\boldsymbol{v}||$ for all $\alpha \in \mathbb{R}$
- $||\boldsymbol{u} + \boldsymbol{v}|| \leq ||\boldsymbol{u}|| + ||\boldsymbol{v}||$

We will make use of multiple different norms. Let $C(\mathbb{R})$ be the set of all real-valued functions $f : \mathbb{R} \to \mathbb{R}$. We will use $C^k(\mathbb{R})$ for $k \geq 0$, to be the space of all real-valued functions $f : \mathbb{R} \to \mathbb{R}$ that are k-times differentiable. When k is zero, $C^k(\mathbb{R})$ will refer to the space of all continuous functions. We will also use C^k to refer to $C^k(\mathbb{R})$. We say f is *integrable* if

$$|\int_{\mathbb{R}} f(x)dx| < \infty$$

Given $f \in C(\mathbb{R})$, let the L^1 norm be defined as

$$||f||_1 := \int_{\mathbb{R}} |f(x)|dx$$

and the L^2 norm be defined as

$$||f||_2 := \sqrt{\int_{\mathbb{R}} f(x)^2 dx}$$

Analogously, given $\boldsymbol{x} \in \mathbb{R}^n$, the ℓ^1 norm is defined by

$$||\boldsymbol{x}||_1 = \sum_{i=1}^{n} |x_i|$$

and the ℓ^2 norm is given by

$$||\boldsymbol{x}||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

Given a function $C : \mathbb{R}^{n+1} \to \mathbb{R}$ and some $f : \mathbb{R}^n \to \mathbb{R}$, such that $C(\boldsymbol{x}, f(\boldsymbol{x})) = 0$ for all $\boldsymbol{x} \in \mathbb{R}^n$, then will call C an *implicit curve* and f an *explicit curve*. If we have a differentiable implicit curve C, we will use the notation $\nabla C(\boldsymbol{x}) = (\frac{\partial C}{\partial x_1}(\boldsymbol{x}), ..., \frac{\partial C}{\partial x_{n+1}}(\boldsymbol{x}))$ at each point $\boldsymbol{x} \in \mathbb{R}^{n+1}$.

We will use $\Delta^n = \{\boldsymbol{x} \in \mathbb{R}^n_{\geq 0} : x_i \geq 0, ||\boldsymbol{x}||_1 = 1\}$. Given a real vector space V and a subset $X \subseteq V$, we say V is convex if for all $\boldsymbol{x}, \boldsymbol{y} \in X$, $\alpha\boldsymbol{x} + (1-\alpha)\boldsymbol{y} \in X$ for all $\alpha \in [0, 1]$. We say X is *strictly convex* if the above only holds for $\alpha \in (0, 1)$.

A function $f : \mathbb{R}^n \to \mathbb{R}$ is *convex* if for all $\alpha \in [0, 1]$, and any $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$

$$f(\alpha\boldsymbol{x} + (1-\alpha)\boldsymbol{y}) \leq \alpha f(\boldsymbol{x}) + (1-\alpha)f(\boldsymbol{y})$$

and is strictly convex if for $\alpha \in (0, 1)$

$$f(\alpha\boldsymbol{x} + (1-\alpha)\boldsymbol{y}) < \alpha f(\boldsymbol{x}) + (1-\alpha)f(\boldsymbol{y})$$

The *epigraph* of a function $f : \mathbb{R}^n \to \mathbb{R}$ is given by $epi(f) = \{(\boldsymbol{x}, y) \in \mathbb{R}^{n+1} : y \geq f(\boldsymbol{x})\}$. The *upper set* of a function $f : \mathbb{R}^n \to \mathbb{R}$ for $\alpha \in \mathbb{R}$ is given by $upper(f; a) = \{\boldsymbol{x} \in \mathbb{R}^n : f(\boldsymbol{x}) \geq \alpha\}$. We will use $upper(f) = upper(f; 0)$. A function $f : \mathbb{R}^n \to \mathbb{R}$ is *quasiconcave* if $upper(f; \alpha)$ is convex for all $\alpha \in \mathbb{R}$. A function $f : \mathbb{R}^n \to \mathbb{R}$ is *strictly quasiconcave* if $upper(f; \alpha)$ is strictly convex for all $\alpha \in \mathbb{R}$.

2.2 Motivation

In this chapter, we develop a general axiomatic framework leading to a definition of an AMM. This mathematical framework captures all key properties current AMMs being used satisfy, but is also general enough to include future developments. AMMs are generally, state machines that store reserves of assets and readily make trades at publicly listed prices.

Here is an example of a *constant-product* AMM, inspired by a simplified version of Uniswap (version 1). In practice, an AMM is implemented as a contract on a blockchain such as Ethereum [57]. At any time, the contract owns x units of asset X, and y units of asset Y. The contract state (x, y) can change, but is subject to the invariant that the product xy is constant. The AMM's states lie along the hyperbolic curve $xy = c$, for some constant c. If a client transfers dx units of X to the AMM, the AMM will transfer dy units of Y back to the client, where $(x + dx) \cdot (y - dy) = x \cdot y$. The client profits if the value of dy units of Y exceeds the value of dx units of X in the current market (or at another AMM). At state (x, y), the *price* of a unit of Y (in units of X) is just the negative of the curve's derivative, the slope of the tangent at that point. Buying units of Y moves the point on the curve, causing *price slippage*, where buying each successive unit of Y makes the next unit more expensive. While the curve's first derivative defines price, its second derivative measures price slippage.

Different parties may have different opinions on the relative values of X and Y. Any such opinion is expressed as a *valuation* $\boldsymbol{v} \in int(\Delta^n)$. Given a valuation (v_X, v_Y) for two assets X and Y, and a constant-product AMM in state (x, y), the value of the AMM's holdings is the dot product $v_X x + v_Y y$. The market values of this AMM's X and Y holdings tend to be equal: $v_X x = v_Y y$, because when they are not, any client can make a risk-free *arbitrage* profit by re-balancing x and y.

We want to generalize the notion of an AMM in a way that encompasses (most) useful behaviors while excluding (most) pathological behaviors, all while leaving enough freedom to accommodate specific circumstances. There are many choices. What other pricing formulas, besides constant-product, make sense? What happens if an AMM manages more than two kinds of assets? Can AMMs be composed, and what kinds of composition make sense? To answer these questions, we need to identify which common-sense properties we want AMMs to display, how those informal properties translate into mathematical properties, and how those properties fare under different notions of composition.

Here is a first step at constructing a more general notion of an AMM. An *n-dimensional AMM* allows trades among n asset types $X_1, \ldots, X_n$. The states of an AMM $\mathcal{A}$ are points in $\mathbb{R}^n$, constrained to lie on a smooth manifold. (We abuse notation by using $\mathcal{A}$ to refer both to the AMM automaton and its manifold.) A *trade* moves the AMM from state $\boldsymbol{x} = (x_1, \ldots, x_n)$ to $\boldsymbol{x}' = (x'_1, \ldots, x'_n)$. The trader pays $x'_i - x_i$ units of X_i where a negative amount is interpreted as a transfer in the other direction, from the AMM to the trader. Payments for a trade $\boldsymbol{x} \mapsto \boldsymbol{x}'$ are summarized by the vector $\boldsymbol{x} - \boldsymbol{x}'$: each positive component is a payments from the AMM to the trader, and each negative component is a payment from the trader to the AMM.

Why are states of $\mathcal{A}$ restricted to a smooth manifold? The existence of a first derivative ensure that the asset

prices are well-defined, the existence of the second derivative ensures that price slippage varies differentiably, and so on.

A valuation expresses an opinion on the relative values of the asset types. A trader who moves an AMM from $\boldsymbol{x}$ to $\boldsymbol{x}'$ makes a profit under valuation $\boldsymbol{v}$ if $\boldsymbol{v} \cdot (\boldsymbol{x} - \boldsymbol{x}')$ is positive, and otherwise incurs a loss. A *stable state* for an AMM $\mathcal{A}$ and valuation $\boldsymbol{v}$ is a point $\boldsymbol{x} \in \mathcal{A}$ that minimizes the dot-product $\boldsymbol{v} \cdot \boldsymbol{x}$. A trader can make an arbitrage profit by moving the AMM from any point to a stable state, and no trader can make a profit by moving the AMM out of a stable state.

Of course, this model is idealized in several ways. Asset pools are not continuous variables: they assume discrete values. Computation is not infinite-precision: round-off errors and numerical instability are concerns.

2.3 Traders and Arbitrageurs

The goal then initially is to compactly describe a set of properties that most existing AMMs share. We first try to identify the core properties informally that describe AMMs used in practice by considering constant product AMMs. Then we describe a minimal set of conditions necessary for both liquidity providers and traders, the two types of agents that interact with AMMs. It is worth emphasizing that the set of conditions we require are actually weaker than the ones used for AMMs in practice. We will make these distinctions later in Section 2.4.

2.3.1 Informal 2D Example

In the following discussion we apply the *no-arbitrage* condition which is typical for example when deriving results in asset pricing theory [19]. The no-arbitrage condition simply assumes profit opportunities between multiple markets are exploited by traders (*arbitrageurs*) until the opportunities no longer exist. AMMs for two assets, assets X_1 and X_2 are governed by the function $x_2 = f(x_1)$. If the state of the constant product AMM is initially $(x_1^{'}, x_2^{'})$ and moves to (x_1, x_2) where $x_1 > x_1^{'}$, then the trader pays $\delta = x_1 - x_1^{'}$ of asset X_1 to the AMM, for a profit in asset X_2 of

$$x_2 - x_2^{'} = f(x_1) - f(x_1 + \delta) \approx -\delta f^{'}(x_1) = \delta r > 0$$

where $r = -f^{'}(x_1)$ is instantaneous exchange rate between asset X_1 and asset X_2. This suggests that we need $f^{'}(x_1) < 0$. For the exchange rate to be defined we require $f \in C^1$. For constant product AMMs, $x_2 = f(x_1) = \frac{c}{x_1}$ which clearly satisfies this condition.

If an agent has a choice between an external market and an AMM, we want to ensure they cannot continue to pump profit from it using services like flash loans [9]. Suppose we have a two asset AMM at state (x_1, x_2) with $f^{'}(x_1) = -r$ and say an agent has access to an external market, again with the same fixed rate of exchange r. Ignoring trading fees, which are negligible, if an agent borrows $\delta \geq 0$ of X_1, then using the AMM followed by the linear market, he can make a profit of

$$\frac{f(x_1) - f(x_1 + \delta)}{r} - \delta \tag{1}$$

So we require

$$f(x_1) - f(x_1 + \delta) \leq r\delta$$

In a similar way, if an agent borrows ϵ of X_2, then if he first uses the AMM he will lose

$$\epsilon = f(x_1 + \delta) - f(x_1)$$

of asset X_2 where $\delta \leq 0$. Since he will gain $-\delta r$ of X_1, for no-arbitrage we require

$$-\delta r \leq f(x_1 + \delta) - f(x_1)$$

or

$$f(x_1) - f(x_1 + \delta) \leq r\delta \tag{2}$$

equivalently. Since this holds for any $\delta \in \mathbb{R}$ by (1),(2), we see that

$$f(x_1 + \delta) \geq f(x_1) - r\delta = f(x_1) + rf^{'}(x_1)$$

Geometrically, the linear approximation to f lies below f. This is in equivalent to f being *convex* and in fact is the case for AMMs used in practice. In terms of derivatives this also means that $f^{''}(x) \geq 0$ for each $x \in \mathbb{R}_+$. So we need $f \in C^2$ and f to be convex. Note that in fact the constant-product AMM satisfies the slightly stronger conditions, namely $f''(x) > 0$ and f is *strictly convex*.

Another desirable property of the constant-product gadget is it's ability to match any external exchange rate. Namely, for $f(x_1) = \frac{c}{x_1}$ we have $f^{'}(x_1) = \frac{-c}{x_1^2} = \frac{-x_2}{x_1}$. By moving around the reserve values of x_1 and x_2 the constant-product AMM is able to achieve any possible valid exchange rate. We refer to an AMM that satisfies this property as being *expressive*. To rule out linear functions that do not satisfy this criteria, we should strengthen our requirements on f and demand that it is *strictly convex*. Additionally, the constant-product example also has the property that all derivatives are valid exchange rates, we call this property *stability*.

This provides some intuition for the types of properties AMMs need to satisfy. However, as we will be described in the following section, we choose a related way to define AMMs that make them much more convenient for analysis. Namely, it allows us to leverage tools from convex analysis and differential geometry to understand properties of different AMMs.

2.3.2 Trader Requirements

Popular AMMs like Uniswap [6], Bancor [42],Curve [23] all use implicit curves that can be written explicitly as strictly convex functions. As described in Section 2.3.1, strict convexity is in fact an essential property.

The natural generalization of this to $n \geq 2$ assets, is the n-dimensional AMM which we might choose to describe as some $f : \mathbb{R}_+^n \to \mathbb{R}$. While this representation is valid, we can define many of the desirable properties more generally.

We want to extend the idea of *expressiveness* to more general sets. We do this in terms of valuations $\mathbf{v} \in int(\Delta^n)$.

Definition 2.3.1. *Given a non-empty set $S \subseteq \mathbb{R}^n_{++}$ and valuation $\mathbf{v} \in int(\Delta^n)$, we say $\mathbf{x} \in S$ is a* stable state *of S if $\mathbf{v} \cdot \mathbf{x} \leq \mathbf{v} \cdot \mathbf{x}'$ for any $\mathbf{x}' \in S$.*

We are interested in non-empty sets $S \subseteq \mathbb{R}^n_{++}$ that satisfy the following properties:

For every market behavior, there is a state corresponding the prevailing market price.

Property 2.3.0.1. *Expressive: Every $\mathbf{v} \in int(\Delta^n)$ has a stable state in S.*

Every state is the stable state for some market price.

Property 2.3.0.2. *Stability: Every $\mathbf{x} \in \partial S$ is the stable state for some $\mathbf{v} \in int(\Delta^n)$.*

2.3.3 Provider Requirements

Now we consider the standpoint of the liquidity provider, an agent that provides the reserves necessary for traders to actually trade. Providers choose to "cash-out" when they believe they will make profit based on what they originally deposited.

In many ways, providers are mathematically the same as consumers in classical economics. The following exposition closely follows the standard rational consumer framework seen in classical microeconomics [46].

We make use of many basic mathematical results from classical economics [46] during this discussion.

A *preference relation* is a relation that arises in economic contexts to model the preference a consumer has between different bundles of goods [46]. We model a liquidity providers preference with a preference relation $\leq_p$ over $\mathbb{R}^n_{++}$. For our purposes, $\leq_p$ is a relation on $\mathbb{R}^n_{++}$.

We now review a few relevant types of properties of preference relations for completeness.

Definition 2.3.2. *Given a preference relation $(\leq_p, \mathbb{R}^n_{++})$, we say $\leq$ is complete if for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n_{++}$, $\mathbf{x} \leq_p \mathbf{y}$ or $\mathbf{x} \geq_p \mathbf{y}$.*

Completeness ensures all asset bundles can be compared by a provider, effectively the provider is decisive in its preferences.

Definition 2.3.3. *Given a preference relation $(\leq_p, \mathbb{R}^n_{++})$, we say $\leq_p$ is transitive if for all $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}^n_{++}$, if $\mathbf{x} \leq_p \mathbf{y}$ and $\mathbf{y} \leq_p \mathbf{z}$, then $\mathbf{x} \leq_p \mathbf{z}$.*

Transitivity captures the assumed rational behavior of the provider.

Definition 2.3.4. *Given a preference relation $(\leq_p, \mathbb{R}^n_{++})$, we say $\leq_p$ is rational if it is complete and transitive.*

Rationality is a property we will assume for $\leq_p$ going forward.

Definition 2.3.5. *Given a preference relation* $(\leq_p, \mathbb{R}^n_{++})$*, we say* $\leq_p$ *is* strictly monotonic *if for* $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n_{++}, \mathbf{x} \neq \mathbf{y}$*, if* $\mathbf{y} \geq \mathbf{x}$ *then* $\mathbf{y} >_p \mathbf{x}$*.*

If a profit-maximizing provider can get more of all asset types then that is preferable.

Definition 2.3.6. *Given a preference relation* $(\leq_p, \mathbb{R}^n_{++})$*, we say* $\leq_p$ *is* strictly convex *if for any* $\mathbf{x} \in S$ *and* $\mathbf{y}, \mathbf{z} \in S$ *such that* $\mathbf{y} \neq \mathbf{z}$ *where* $\mathbf{y} \geq \mathbf{x}$*,* $\mathbf{z} \geq \mathbf{x}$*, then* $\alpha\mathbf{y} + (1-\alpha)\mathbf{z} > \mathbf{x}$ *for any* $\alpha \in (0,1)$*.*

The previous definition of a preference in our context, describes a liquidity provider who would strictly prefer to have a diversified portfolio rather than not. The reason for this is that it allows liquidity providers to take on less risk if some of assets they are investing in rapidly fluctuate in value. This risk-management strategy is typically known as *diversification*.

Definition 2.3.7. *Given a preference relation* $(\leq, \mathbb{R}^n_{++})$*, we say* $\leq$ *is* continuous *if for any sequence of pairs* $\{(x_n, y_n)\}_{n=1}^{\infty} \subset S$ *such that* $x_n \leq y_n$ *for each* n*, then* $\lim_{n\to\infty} x_n \leq \lim_{n\to\infty} y_n$*.*

The previous, rather technical result, ensures provider preferences vary continuously.

We make the following assumptions about $\leq_p$:

Axiom 2.3.1. $\leq_p$ *is rational*

Axiom 2.3.2. $\leq_p$ *is strictly monotonic*

Axiom 2.3.3. $\leq_p$ *is strictly convex*

Axiom 2.3.4. $\leq_p$ *is continuous*

Definition 2.3.8. *We say a preference relation* $(\leq_p, \mathbb{R}^n_{++})$ *is a provider preference if it satisfies Axioms 2.3.1-2.3.4.*

2.3.4 AMM Definition

We now define an AMM as some object consistent with Axioms 2.3.1-2.3.4. To start we assume we are given some provider preference $\leq_p$.

Definition 2.3.9. *Given a set non-empty set* $S \subset \mathbb{R}^n_{++}$ *we say it is an AMM of dimension* n *if there exists some provider preference* $(\leq_p, \mathbb{R}^n_{++})$ *and some* $\mathbf{x} \in \mathbb{R}^n_{++}$ *with* $S = S_{\mathbf{x}} = \{\mathbf{y} \in \mathbb{R}^n_{++} : \mathbf{y} \geq_p \mathbf{x}\}$ *so that* $S = S_{\mathbf{x}}$*. In this case we call* $S_{\mathbf{x}}$ *the AMM induced from* $\leq_p$ *with base point* $\mathbf{x}$*.*

When referring to an AMM, we will initially use the tuple $(\leq_p, \mathbf{x})$.

The set $S_{\mathbf{x}}$ can be thought of as the set of future acceptable states to a liquidity provider if the initial reserves are in state $\mathbf{x}$. To ensure consistency with our trader requirements, we need to check this AMM definition satisfies Properties 2.3.0.1-2.3.0.2. Along the way we will show some properties of AMMs that will be useful later.

The next result gives us a convenient way of specifying AMMs rather than using preference relations. The proofs of Lemmas 2.3.5-2.3.7 are omitted since they mirror results for consumer preferences [46].

Lemma 2.3.5. *Given a preference relation $(\leq, \mathbb{R}^n_{++})$ that is*

- $\leq$ *rational*
- $\leq$ *continuous*

Then there is a continuous function $C : \mathbb{R}^n_{++} \to \mathbb{R}$ that represents $\leq$.

Lemma 2.3.6. *If the preference relation $\leq$ from Lemma 3 is strictly monotonic, then C is a strictly increasing function. That is if $\mathbf{x} \neq \mathbf{y}, \mathbf{y} \geq \mathbf{x}$ then $C(\mathbf{x}) < C(\mathbf{y})$.*

Lemma 2.3.7. *If the preference relation $\leq$ from Lemma 3 is strictly convex, then C is strictly quasiconcave.*

Lemmas 2.3.5-2.3.7 tell us that we can alternatively just think of an AMM as the upper set of some continuous, strictly increasing, strictly quasiconcave function C. Namely we can write $S_{\mathbf{x}} = \{\mathbf{y} \in \mathbb{R}^n_{++} : C(\mathbf{y}) \geq C(\mathbf{x})\}$. We will also use the tuple $(C, \mathbf{x})$ when referring to an AMM, and call this the *implicit* representation.

Alternatively, the next result tells us that if we find such a continuous C, it induces a provider preference.

Lemma 2.3.8. *Given some $C : \mathbb{R}^n_{++} \to \mathbb{R}$ that is*

- *Continuous*
- *Increasing*
- *Strictly quasiconcave*

then there exists a provider preference $(\leq_p, \mathbb{R}^n_{++})$ where for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n_{++}$, $\mathbf{x} \leq_p \mathbf{y}$ iff $C(\mathbf{x}) \leq C(\mathbf{y})$.

Lemmas 2.3.5-2.3.8 allow us to freely change our representation from a provider preference to a continuous function. Modeling investors using a preference relation provides a straightforward representation from an axiomatic point of view. However, it is much easier in practice for computations to work with alternative representations that are equivalent to the original preference relation. Because of this, we choose to work with the implicit and explicit curves that represents the same underlying preference relation. Additionally, these are also the ways AMMs are described in practice.

Lemma 2.3.9. *Given an n dimensional AMM $S_{\mathbf{x}}$ induced from $\leq_p$, if $\mathbf{y} =_p \mathbf{x}$ then $S_{\mathbf{x}} = S_{\mathbf{y}}$.*

Proof. By symmetry we only need to show one direction. Let $\mathbf{z} \in S_{\mathbf{x}}$ so $\mathbf{y} =_p \mathbf{x} \leq_p \mathbf{z}$ meaning $\mathbf{z} \in S_{\mathbf{y}}$. □

The above result says that the AMM definition doesn't depend on the base point $\mathbf{x}$, any equivalent point will do.

Lemma 2.3.10. *An n dimensional AMM $S_{\mathbf{x}}$ is closed.*

Proof. Note that $A = \{\alpha \in \mathbb{R} : \alpha \geq C(\mathbf{x})\}$ is a closed set. We can write $S_{\mathbf{x}} = \{\mathbf{y} \in \mathbb{R}^n_{++} : C(\mathbf{y}) \geq C(\mathbf{x})\} = C^{-1}(A)$. The inverse image of a closed set under a continuous function is closed so we have our result. □

Lemma 2.3.11. *Given an n dimensional AMM $S_{\mathbf{x}}$ induced from $\leq_p$, $\partial S_{\mathbf{x}} = \{\mathbf{y} \in \mathbb{R}^n_{++} : \mathbf{y} =_p \mathbf{x}\}$.*

Proof. Suppose that $\mathbf{y} =_p \mathbf{x}$ so $C(\mathbf{y}) = C(\mathbf{x})$. From the previous lemma we know $S_\mathbf{x} = S_\mathbf{y}$. Choose some $\epsilon > 0$ and consider $B_\mathbf{y}(\epsilon)$. Choose $\mathbf{y}_l << \mathbf{y}$ and $\mathbf{y}_r >> \mathbf{y}$ such that $\mathbf{y}_l, \mathbf{y}_r \in B_\mathbf{y}(\epsilon)$. We can do this by scaling $\mathbf{y}$ appropriately. Since C is increasing $C(\mathbf{y}_l) < C(\mathbf{y}) < C(\mathbf{y}_r)$ meaning $\mathbf{y}_l \in S_\mathbf{x}^c$ and $\mathbf{y}_r \in S_\mathbf{x}$, namely $\mathbf{y} \in \partial S_\mathbf{x}$. In the other direction say $\mathbf{y} \in \partial S_\mathbf{x}$ and suppose $C(\mathbf{y}) > C(\mathbf{x})$. Since $S_\mathbf{x}$ is closed we know $\partial S_\mathbf{x} \subset S_\mathbf{x}$. Because $\mathbf{y} \in \partial S_\mathbf{x}$, we can construct an increasing sequence $\{\mathbf{y}_n\}_{n=1}^\infty \subseteq S_\mathbf{x}^c$ where for each n we have $C(\mathbf{y}_n) < C(\mathbf{y})$ and $\mathbf{y}_n \to \mathbf{y}$. Since $\{\mathbf{y}_n\}_{n=1}^\infty$ increasing and C is an increasing function we know $\{C(\mathbf{y}_n)\}_{n=1}^\infty$ is an increasing sequence with upper bound $C(\mathbf{x})$. Thus $\lim_{n\to\infty} C(\mathbf{y}_n) \leq C(\mathbf{x})$. But continuity tells us $\lim_{n\to\infty} C(\mathbf{y}_n) = C(\mathbf{y})$, meaning $C(\mathbf{y}) \leq C(\mathbf{x})$, a contradiction. □

We now verify Property 2.3.0.1.

Lemma 2.3.12. *If $S \subseteq \mathbb{R}^n_{++}$ is non-empty and closed, then every $\mathbf{v} \in int(\Delta^n)$ has a stable state in S.*

Proof. Since $S \neq \emptyset$, there is some $\mathbf{x} \in S$. Let $S' = \{\mathbf{x}' \in \mathbb{R}^n_+ : \mathbf{v} \cdot \mathbf{x}' \leq \mathbf{v} \cdot \mathbf{x}\}$, a compact set. Since S is closed, $\tilde{S} = S \cap S'$ is compact. A stable state is then the solution the following optimization problem:

$$\min_{\mathbf{x} \in \tilde{S}} \mathbf{v} \cdot \mathbf{x}$$

We are guaranteed the minimum exists since $\mathbf{v} \cdot \mathbf{x}$ is a continuous function on the compact set $\tilde{S}$. □

Theorem 2.3.13. *AMMs satisfy Property 2.3.0.1.*

Proof. Consider some n dimensional AMM $S_\mathbf{x}$. $S_\mathbf{x}$ is closed by Lemma 2.3.10 so by Lemma 2.3.12, $S_\mathbf{x}$ satisfies Property 2.3.0.1. □

Lemma 2.3.14. *If $S \subseteq \mathbb{R}^n_{++}$ is a non-empty,strictly convex set, then for any $\mathbf{v} \in int(\Delta^n)$, the stable state for $\mathbf{v}$ is unique.*

Proof. Fix $\mathbf{v} \in int(\Delta^n)$ and let $\mathbf{x}, \mathbf{x}' \in S$ where $\mathbf{x} \neq \mathbf{x}'$ but $w = \mathbf{v} \cdot \mathbf{x} = \mathbf{v} \cdot \mathbf{x}'$. Choose some $\alpha \in (0,1)$ and let $\tilde{\mathbf{x}} = \alpha\mathbf{x} + (1-\alpha)\mathbf{x}' \in int(S)$ by strict convexity. Since $int(S)$ is open we can find some $\epsilon > 0$ such that $B_{\tilde{\mathbf{x}}}(\epsilon) \subset int(S)$. Now choose $\mathbf{x}^* \in B_{\tilde{\mathbf{x}}}(\epsilon)$ where $\mathbf{x}^* < \tilde{\mathbf{x}}$ (in each coordinate). Then we have $\mathbf{v} \cdot \mathbf{x}^* < \mathbf{v} \cdot \tilde{\mathbf{x}} = \alpha\mathbf{v} \cdot \mathbf{x} + (1-\alpha)\mathbf{v} \cdot \mathbf{x}' = \alpha w + (1-\alpha)w = w$, a contradiction. □

Thus we can rightfully talk about *the* stable state for a valuation $\mathbf{v}$, since the stable state is unique.

Lemma 2.3.15. *If $S \subseteq \mathbb{R}^n_{++}$ is a non-empty, closed, strictly convex set, then for any $\mathbf{v} \in int(\Delta^n)$, the stable state for $\mathbf{v}$ is in ∂S.*

Proof. Fix $\mathbf{v} \in \text{int}(\Delta^n)$ and let $\mathbf{x}^*$ be its stable state. Suppose that $\mathbf{x}^* \notin \partial S$ so $\mathbf{x} \in \text{int}(S)$. Similar to Lemma 2.3.14, we can find $\epsilon > 0$ and an open ball $B_{\mathbf{x}^*}(\epsilon) \subset \text{int}(S)$. Choosing $\mathbf{x} \in B_{\mathbf{x}^*}(\epsilon)$ with $\mathbf{x} << \mathbf{x}^*$ we get $\mathbf{v} \cdot \mathbf{x} < \mathbf{v} \cdot \mathbf{x}^*$, a contradiction. □

The following two results give a construction for associating a valuation with each point of an AMM's boundary.

Lemma 2.3.16. *Given an n dimensional AMM $S_\mathbf{x}$, for every $\mathbf{y} \in \partial S_\mathbf{x}$, there is some $\mathbf{w} \in -\mathbb{R}^n_{++}$ such that $\mathbf{w} \cdot \mathbf{y} > \mathbf{w} \cdot \mathbf{z}$ for all $\mathbf{z} \in S_\mathbf{x}$, $\mathbf{z} \neq \mathbf{y}$.*

Proof. Let $\mathbf{y} \in \partial S_{\mathbf{x}}$. Since $S_{\mathbf{x}}$ is strictly convex, by the supporting hyperplane theorem [14], there is some $\mathbf{w} \in \mathbb{R}^n, \mathbf{w} \neq 0$ such that $\mathbf{w} \cdot \mathbf{y} > \mathbf{w} \cdot \mathbf{z}$ for all $\mathbf{z} \in S_{\mathbf{x}}$, $\mathbf{z} \neq \mathbf{y}$. We now just need to argue that $\mathbf{w} \in \mathbb{R}^n_{--}$. Say $\mathbf{w} \in \mathbb{R}^n_{++}$. Choose any $\epsilon > 0$ and let $\tilde{\mathbf{y}} = \mathbf{y} + \epsilon \mathbf{w}$ so that $\mathbf{w} \cdot \tilde{\mathbf{y}} = \mathbf{w} \cdot \mathbf{y} + \epsilon ||\mathbf{w}||^2 > \mathbf{w} \cdot \mathbf{y}$. By monotonicity $\tilde{\mathbf{y}} \in S_{\mathbf{x}}$ which gives us a contradiction. Now consider the case when $\mathbf{w} \in \mathbb{R}^n / (\mathbb{R}^n_{--} \cup \mathbb{R}^n_{++})$, namely $\mathbf{w}$ cannot have all strictly positive or all strictly negative entries. We now construct an orthogonal vector $\tilde{\mathbf{w}}$ to $\mathbf{w}$. For all of the non-negative entries of $\mathbf{w}$ in $\tilde{\mathbf{w}}$, replace them with the sum of the absolute value of the negative values. For all of the negative entries of $\mathbf{w}$ in $\tilde{\mathbf{w}}$, replace them with the sum of all of the non-negative entries of $\mathbf{w}$. This guarantees $\tilde{\mathbf{w}} >> 0$ and $\tilde{\mathbf{w}} \cdot \mathbf{w} = 0$. Pick some $\epsilon > 0$ and let $\tilde{\mathbf{y}} = \mathbf{y} + \epsilon \tilde{\mathbf{w}}$ so $\tilde{\mathbf{w}} \cdot \tilde{\mathbf{y}} = \tilde{\mathbf{w}} \cdot \mathbf{y} + \epsilon \tilde{\mathbf{w}} \cdot \mathbf{w} = \tilde{\mathbf{w}} \cdot \mathbf{y}$ and yet $\tilde{\mathbf{y}} \in S_{\mathbf{x}}$ by monotonicity. So we get a contradiction. Thus $\mathbf{w} \in \mathbb{R}^n_{++}$. □

Theorem 2.3.17. *AMMs satisfy Property 2.3.0.2.*

Proof. Let $S_{\mathbf{x}}$ be an n dimensional AMM and let $\mathbf{y} \in \partial S_{\mathbf{x}}$. Choose $\mathbf{w} \in -\mathbb{R}^n_{++}$ as described in Lemma 2.3.16 for $\mathbf{y}$. So we have $\mathbf{w} \cdot \mathbf{y} > \mathbf{w} \cdot \mathbf{z}$ for $\mathbf{z} \in S_{\mathbf{x}}$, $\mathbf{z} \neq \mathbf{y}$. If we negate $\mathbf{w}$ and rescale the result so that the elements sum to 1 then we get some $\mathbf{v} \in int(\Delta^n)$. Thus we have $\mathbf{v} \cdot \mathbf{y} < \mathbf{v} \cdot \mathbf{z}$ for all $\mathbf{z} \in S_{\mathbf{x}}$ where $\mathbf{z} \neq \mathbf{y}$. Thus $\mathbf{y}$ is a stable state for $\mathbf{v}$. □

2.4 AMMs as Manifolds

In this section we look at some important results regarding the topology of AMMs. We also look at some alternative ways of talking about AMMs that can be useful in practice. We show that AMMs are indeed topological manifolds and show how in some cases they can be made into differentiable manifolds.

2.4.1 Topological Manifolds

Lemma 2.4.1. *Given n dimensional AMM S, ∂S can be written uniquely as the image of a strictly convex, continuous function in $n-1$ coordinates.*

Proof. Consider some $\mathbf{y} \in \partial S$. Without loss of generality fix the first $n-1$ coordinates of $\mathbf{y}$. Now let $f(y_1, ..., y_{n-1}) = \{z \in \mathbb{R}_{++} : (y_1, ..., y_{n-1}, z) \in \partial S\}$. Note $y_n \in f(y_1, ..., y_{n-1})$ so $|f(y_1, ..., y_{n-1})| \neq \emptyset$. Suppose there are two distinct elements $z, z^{'} \in f(y_1, ..., y_{n-1})$ where $z < z^{'}$. Since $(y_1, ..., y_{n-1}, z^{'}) \in \partial S$ we know by Theorem 2.3.17, it is a stable state for some $\mathbf{v} \in int(\Delta^n)$. But we have $\mathbf{v} \cdot (y_1, ..., y_{n-1}, z) < \mathbf{v} \cdot (y_1, ..., y_{n-1}, z^{'})$ by construction, contradicting the fact that $(y_1, ..., y_{n-1})$ is a stable state for $\mathbf{v}$. Hence $f(y_1, ..., y_{n-1})$ is a function. Because $ep(f) = S$, a strictly convex set, this is equivalent to f being a strictly convex function. Convex functions on Euclidean spaces are continuous so f is continuous. □

In the following discussion we will use $\cong$ to denote homeomorphic.

Lemma 2.4.2. *For any n dimensional AMM S, ∂S is homeomorphic to $int(\Delta^n)$.*

Proof. From Lemma 2.4.1, we can write $\partial S_{\mathbf{x}} = im(f)$ for some continuous function $f : \mathbb{R}^{n-1}_{++} \to \mathbb{R}$. Now define $\phi : \mathbb{R}^{n-1}_{++} \to \mathbb{R}^n_{++}$ by $\phi(\mathbf{x}) = (\mathbf{x}, f(\mathbf{x}))$. Note that $\text{im}(\phi) = \partial S$ so ϕ is certainly surjective onto ∂S. If $\phi(\mathbf{x}) = \phi(\mathbf{y})$,

then $(x_1, ..., x_{n-1}) = (y_1, ..., y_{n-1})$ so $f(x_1, ..., x_{n-1}) = f(y_1, ..., y_{n-1})$ and $\mathbf{x} = \mathbf{y}$, meaning ϕ is injective. ϕ is a continuous function since its coordinate functions are continuous. Additionally, for $\mathbf{y} = (y_1, ..., y_{n-1}, y_n) \in \partial S_{\mathbf{x}}$, $\phi^{-1}(\mathbf{y}) = (y_1, ..., y_{n-1})$, a projection, which is a continuous function. Thus ϕ is a homeomorphism and $\partial S_{\mathbf{x}} \cong \mathbb{R}^{n-1}_{++}$. It is also the case that $int(\Delta^n) \cong B^{n-1}$, the open unit ball in $\mathbb{R}^{n-1}$. Since $\mathbb{R}^{n-1}_{++}$ and B^{n-1} are both open convex sets in $\mathbb{R}^{n-1}$, we have $\mathbb{R}^{n-1}_{++} \cong B^{n-1}$. Transitively then we get $\partial S \cong int(\Delta^n)$. □

Lemma 2.4.3. *The boundary of all n dimensional AMMs are homeomorphic. Namely they are all n dimensional manifolds.*

Proof. Let S, S' be two n dimensional AMMs. Applying Lemma 2.4.2 twice we get $S \cong int(\Delta^n)$ and $S' \cong int(\Delta^n)$. Using transitivity of homeomorphisms we then get $S \cong S'$. Additionally, because $int(\Delta^n)$ is a topological manifold of dimension $n-1$, so are AMMs of dimension n. □

Thus we see if the dimension of an AMM is n, the dimension of the underlying topological manifold is really $n-1$. This ensures our notion of dimension for an AMM is well-defined. Intuitively, $n-1$ is the number of degrees of freedom for the AMM.

For any n-dimensional AMM, define the map $\nu : int(\Delta^n) \to \partial S$ that sends a valuation to its stable state. In general, ν is not a homeomorphism. However, we get the following result:

Lemma 2.4.4. *For an n-dimensional AMM S, the stable state map $\nu : int(\Delta^n) \to \partial S$ is a continuous bijection.*

Proof. By Theorem 2.3.17, ν is surjective and by uniqueness of stable states (Lemma 2.3.14), ν is injective. For continuity consider some $\{\mathbf{v}_n\}_{n=1}^{\infty} \subset int(\Delta^n)$ where $\lim_{n\to\infty} \mathbf{v}_n = \mathbf{v}$. Let $\tilde{\mathbf{x}} = \lim_{n\to\infty} \nu(\mathbf{v}_n)$ and $\mathbf{x}^* = \nu(\mathbf{v})$. Suppose $\tilde{\mathbf{x}} \neq \mathbf{x}^*$. Note that $\mathbf{v} \cdot \mathbf{x}^* < \mathbf{v} \cdot \tilde{\mathbf{x}}$ by definition of stable state. Letting $\overline{\mathbf{x}} = \frac{\mathbf{x}^* + \tilde{\mathbf{x}}}{2}$ where by strict convexity we know $\overline{\mathbf{x}} \in int(S)$. We also have that $\mathbf{v} \cdot \mathbf{x}^* < \mathbf{v} \cdot \overline{\mathbf{x}} < \mathbf{v} \cdot \tilde{\mathbf{x}}$. Notice now that $\mathbf{v}_n \cdot \overline{\mathbf{x}} > \mathbf{v}_n \cdot \nu(\mathbf{v}_n)$ by definition, so taking limits we get $\mathbf{v} \cdot \overline{\mathbf{x}} > \mathbf{v} \cdot \tilde{\mathbf{x}}$, a contradiction. Thus $\lim_{n\to\infty} \nu(\mathbf{v}_n) = \nu(\mathbf{v})$. □

Our current definition of an AMM is not sufficient to guarantee that ν^{-1} is also continuous. Consider the function $C : \mathbb{R}^2_{++} \to \mathbb{R}$:

$$C(x, y) = \begin{cases} xy - 10 & (x, y) \in (0, 1] \times \mathbb{R}_{++} \\ xy - 9x - 1 & (x, y) \in [1, \infty) \times \mathbb{R}_{++} \end{cases}$$

Note that C satisfies the properties of Lemma 2.3.8, for the AMM given by $S_{(1,9)} = \{(x, y) \in \mathbb{R}^2_{++} : C(x, y) \geq C(1, 9) = 0\}$. However, the point $(1, 9)$ presents a difficulty because ν^{-1} changes abruptly in a neighborhood of $(1, 9)$. This illustrates the following: Even if the underlying provider preference varies continuously, this does not guarantee that the exchange rate on the AMM boundary varies continuously.

2.4.2 Differentiable AMMs

As we saw in the previous example, we can not guarantee a continuously varying market valuation for our definition of AMMs. Going forward, it is then more useful to consider a slightly more restricted classes of AMMs (*differentiable*

AMMs) that are consistent with this kind of common-sense property. All AMMs used in practice satisfy this.

Definition 2.4.1. *Given an n dimensional AMM* $S_{\mathbf{x}} = \{\mathbf{y} \in \mathbb{R}^n_{++} : C(\mathbf{y}) \geq C(\mathbf{x})\}$*, we say* $S_{\mathbf{x}}$ *is a* C^k *n dimensional AMM if* C *is* C^k *for some* $k \geq 0$*.*

If C is C^1 then we say $S_{\mathbf{x}}$ is a *differentiable AMM.*

Lemma 2.4.5. *For an n-dimensional differentiable AMM S, the stable state map* $\nu : int(\Delta^n) \to \partial S$ *is a homeomorphism.*

Proof. From Lemma 2.4.4, we know ν is both a bijection and is continuous, so it is enough to just show ν^{-1} is continuous. For any $\mathbf{v}$ with stable state $\mathbf{x}$, the first-order conditions tell us that $\mathbf{v} = \lambda \nabla C(\mathbf{x})$ for some $\lambda \in \mathbb{R}, \lambda \neq 0$. Since C is increasing we know $\lambda > 0$. Thus $\mathbf{v}$ as a function of $\mathbf{x}$ can be written $\mathbf{v}(\mathbf{x}) = \lambda(\mathbf{x})\nabla C(\mathbf{x})$. We know $\nabla C(\mathbf{x})$ is continuous since C is C^1 so it is enough to check $\lambda(\mathbf{x})$ is continuous. The normalization condition on $\mathbf{v}(\mathbf{x})$ and uniqueness of $\lambda(\mathbf{x})$ tells us that $\lambda(\mathbf{x}) = \frac{1}{\sum_{i=1}^n \frac{\partial C(\mathbf{x})}{\partial x_i}}$, a continuous function since each $\frac{\partial C(\mathbf{x})}{\partial x_i}$ is continuous. Thus ν^{-1} is continuous. □

Theorem 2.4.6. *Given any two n-dimensional differentiable AMMs* S, S' *there is a stable state preserving homemorphism* $\mu : \partial S \to \partial S'$*. Namely,* μ *is a correspondence between stable states for each valuation* $\mathbf{v} \in int(\Delta^n)$*.*

Proof. By Lemma 2.4.5, we can find $\nu : int(\Delta^n) \to \partial S$ and $\nu' : int(\Delta^n) \to \partial S'$, both homeomorphisms. Define $\mu = \nu' \circ \nu^{-1}$. Since homeomorphisms are closed under composition, μ is a homeomorphism. Let $\mathbf{v} \in int(\Delta^n)$ with stable states $\mathbf{x} \in \partial S, \mathbf{x}' \in \partial S'$ so $\nu(\mathbf{v}) = \mathbf{x}$ and $\nu'(\mathbf{v}) = \mathbf{x}'$. Thus $\mu(\mathbf{x}) = \nu'(\nu^{-1}(\mathbf{x})) = \nu'(\mathbf{v}) = \mathbf{x}'$, namely μ preserves stable states. □

Lemma 2.4.7. *Every n-dimensional differentiable AMM can be expressed as the epigraph of a strictly convex, differentiable function.*

Proof. Let $S_{\mathbf{x}}$ be an n-dimensional AMM with associated function $C : \mathbb{R}^n_{++} \to \mathbb{R}$. By the first-order condition we used in Lemma 18, namely $\mathbf{v}(\mathbf{x}) = \lambda(\mathbf{x})\nabla C(\mathbf{x})$ for $\lambda(\mathbf{x}) > 0$, we know that $\nabla C(\mathbf{x}) \neq 0$ for each $\mathbf{x} \in \partial S_{\mathbf{x}}$. Using the implicit function theorem, we know there is some C^1 function $f : \mathbb{R}^n_{++} \to \mathbb{R}$ such that $(\mathbf{y}, f(\mathbf{y})) \in \partial S_{\mathbf{x}}$ for each $\mathbf{y} \in \mathbb{R}^n_{++}$. Lemma 2.4.1 tells us that this function must be unique, so f is also strictly convex with $epi(f) = S_{\mathbf{x}}$. □

We call the function from Lemma 2.4.7, the *explicit* representation for the AMM.

Lemma 2.4.8. *Given an n dimensional AMM* $(C, \mathbf{x})$ *and its explicit representation* f*, then* $c_i = \lim_{y_i \to \infty} f(y_1, ..., y_i, ..., y_{n-1})$ *exists for each* $i \in \{1, ..., n-1\}$*.*

Proof. Choose some $i \in \{1, ..., n-1\}$. Assume WLOG that $C(\mathbf{x}) = 0$. If not, then we can just redefine a new $C'(\mathbf{y}) = C(\mathbf{y}) - C(\mathbf{x})$ so $C'(\mathbf{x}) = 0$. For any $\mathbf{y} \in \mathbb{R}^{n-1}_{++}$ we have $C(\mathbf{y}, f(\mathbf{y})) = 0$. Applying the chain rule yields

$$\frac{\partial f}{\partial y_i} = \frac{-\frac{\partial C}{\partial x_i}}{\frac{\partial C}{\partial x_n}}$$

By the analysis from Lemma 2.4.5, we know $\nabla C(\mathbf{x}) > 0$ meaning $\frac{\partial f}{\partial y_i} < 0$ so f is a strictly decreasing function in each variable. If we let $g(y) = f(y_1, ..., y_{i-1}, y, y_{i+1}, ..., y_{n-1})$, then we have $g(y) \geq 0$ for each $y \in \mathbb{R}^+$. For any increasing sequence $\{y^{(j)}\}_{j=1}^{\infty}$, the sequence $\{g(y^{(j)})\}_{j=1}^{\infty}$ is bounded below and strictly decreasing so $c_i = \lim_{y\to\infty} g(y) \geq 0$ exists. □

Definition 2.4.2. *Given an explicit representation f for an n dimensional AMM, we say f is in standard form if $\lim_{y_i\to\infty} f(y_1, ..., y_i, ..., y_{n-1}) = 0$ for each i.*

By Lemma 2.4.8, every AMM can be put in standard form by an appropriate translation by $\mathbf{c} = (c_1, ..., c_{n-1})$.

Chapter 3

AMM composition

In this section, we provide a natural algebraic structure on the space of all AMMs. Namely, we define multiple methods of composition between AMMs that reflect how traders interact with them in practice. Importantly, the space of AMMs is closed under all of the defined operations. Additionally for each operation, we explicitly show the relationship between stable states in the original AMM(s), and the AMM resulting from applying the operation.

When solving for the stable state of a valuation $\boldsymbol{v}$ for some AMM A we are solving for:

$$\arg\min_{\boldsymbol{x}\in A} \boldsymbol{v}\cdot\boldsymbol{x}$$

Note that this solution is invariant under scaling of $\boldsymbol{v}$. It is then useful to introduce the following notation: If we are given some $\boldsymbol{v}\in\mathbb{R}^n_{++}$, we can associate it with the normalized version $\tilde{\boldsymbol{v}} = \frac{\boldsymbol{v}}{||\boldsymbol{v}||_1}$ where $\tilde{\boldsymbol{v}}\in int(\Delta^n)$. We will freely go back and forth between the valuation $\tilde{\boldsymbol{v}}$ and its unormalized version $\boldsymbol{v}$. Notationally we will distinguish the normalized version as $\tilde{\boldsymbol{v}}$ from the original $\mathbf{v}$ if convenient.

3.1 Operations

It is useful to be able to reduce an AMM's dimension, perhaps by ignoring some assets, or by creating "baskets" of distinct assets that can be treated as a unit. Here we introduce two tools for reducing dimensionality: *projection*, and *asset virtualization*.

3.1.1 Projection

An AMM may provide the ability to trade across a variety of asset types, but traders may choose to restrict their attention to a subset, ignoring the rest. Perhaps the ignored assets are too volatile, or not volatile enough, or there are regulatory barriers to owning them.

Mathematically, the *projection* operator acts on an AMM by fixing some state coordinates to constant values and letting the rest vary. We will show that projecting an AMM in this way yields another AMM of lower dimension. Informally, traders are free to ignore uninteresting assets.

Definition 3.1.1. *Let $\boldsymbol{x} = (x_1, \ldots, x_n)$, $\boldsymbol{y} = (y_1, \ldots, y_m)$, and $\boldsymbol{a} = (a_1, \ldots, a_n)$ be a constant. The projection of A onto $\boldsymbol{a}$ is given by $A_{\boldsymbol{a}}(\boldsymbol{y}) = A(\boldsymbol{a}, \boldsymbol{y}) = 0$*

Lemma 3.1.1. *Given an $(n+m)$-dimensional AMM $A(\boldsymbol{x}, \boldsymbol{y}) = 0$ and $\boldsymbol{a} \in \mathbb{R}^n_{>0}$, the projection $A_{\boldsymbol{a}}(\boldsymbol{y})$ is an m-dimensional AMM.*

Proof. It is enough to check that $A_{\boldsymbol{a}}$ is differentiable, strictly increasing, and $upper(A)$ is strictly convex. Because $A(\boldsymbol{x}, \boldsymbol{y})$ is differentiable, so is $A(\boldsymbol{a}, \boldsymbol{y}) = A_{\boldsymbol{a}}(\boldsymbol{y})$. To show that $A_{\boldsymbol{a}}$ is strictly increasing, let $\boldsymbol{x}' >> \boldsymbol{x}$.

$$A_{\boldsymbol{a}}(\boldsymbol{x}) = A(\boldsymbol{a}, \boldsymbol{x}) \ < A(\boldsymbol{a}, \boldsymbol{x}') = A_{\boldsymbol{a}}(\boldsymbol{x}').$$

To show that $upper(A_{\boldsymbol{a}})$ is strictly convex, pick distinct $\boldsymbol{x}$ and $\boldsymbol{x}'$ in $upper(A_{\boldsymbol{a}})$. Namely $A(\boldsymbol{a}, \boldsymbol{x}) = A_{\boldsymbol{a}}(\boldsymbol{x}) \geq 0$ and $A(\boldsymbol{a}, \boldsymbol{x}^{'}) = A_{\boldsymbol{a}}(\boldsymbol{x}') \geq 0$. For $t \in (0, 1)$

$$A_{\boldsymbol{a}}(t\boldsymbol{x} + (1-t)\boldsymbol{x}') = A(t\boldsymbol{a} + (1-t)\boldsymbol{a}, t\boldsymbol{x} + (1-t)\boldsymbol{x}') = A(t(\boldsymbol{a}, \boldsymbol{x}) + (1-t)(\boldsymbol{a}, \boldsymbol{x}^{'})) > 0$$

by the strict convexity of $upper(A)$. □

Lemma 3.1.2. *For index set I, let $\boldsymbol{v} = (v_i | i \in I)$ be a valuation for a sequence of asset types $X = (X_i | i \in I)$. For $J \subset I$, $\boldsymbol{v}' = (\frac{v_j}{1 - \sum_{k \notin J} v_k} | j \in J)$ is a valuation for $X' = (X_j | j \in J) \subset X$.*

We say that X' *inherits* $\boldsymbol{v}'$ from valuation $\boldsymbol{v}$ of X.

The next lemma states that stable states persist under projection.

Lemma 3.1.3. *Let $A(\boldsymbol{x}, \boldsymbol{y}) = 0$ be an $(n+m)$-dimensional AMM, $\boldsymbol{v} \in int(\Delta^{n+m})$ a valuation on $(\boldsymbol{x}, \boldsymbol{y})$. If $(\boldsymbol{a}, \boldsymbol{b})$ is the stable state for $\boldsymbol{v}$ in $A(\boldsymbol{x}, \boldsymbol{y})$ then $\boldsymbol{b}$ is the stable state for the inherited valuation $\boldsymbol{v}^{'}$.*

Proof. Suppose the stable state for $\boldsymbol{v}^{'}$ is $\boldsymbol{b}' \neq \boldsymbol{b}$, namely $\boldsymbol{v}^{'} \cdot \boldsymbol{b}^{'} < \boldsymbol{v}^{'} \cdot \boldsymbol{b}$. Scaling both sides by $1 - \sum_{i=1}^{n} v_i$ yields $(v_{n+1}, \ldots, v_{n+m}) \cdot \boldsymbol{b}' < (v_{n+1}, \ldots, v_{n+m}) \cdot \boldsymbol{b}$. Adding $(v_1, \ldots, v_n) \cdot \boldsymbol{a}$ to both sides yields $v \cdot (\boldsymbol{a}, \boldsymbol{b}') < v \cdot (\boldsymbol{a}, \boldsymbol{b})$, contradicting the assumption that $(\boldsymbol{a}, \boldsymbol{b})$ is the stable state for $\boldsymbol{v}$. □

3.1.2 Virtualizing Assets

It is sometime convenient to create a "virtual asset" from a linear combination of assets. Here we show that replacing a set of assets traded by an AMM with a single virtual asset is also an AMM. This construction works for any linear combination, although the most sensible combination reflects the relative market value of the assets.

Here is a simple example of asset virtualization. Consider an AMM that trades across three asset types, X, Y, Z, defined by the constant-product formula $A(x, y, z) := xyz - 8 = 0$, initialized in state $(2, 2, 2)$. A trader believes that 2 units of Y are always worth 1 unit of Z, and that it makes sense to link them in that ratio by creating a virtual asset W worth $2/3$ units of Y and $1/3$ unit of Z, and to trade in a single denomination of W instead of individual denominations of Y and Z.

Formally, the trader defines W in terms of the valuation $\boldsymbol{v} = (2/3, 1/3)$ on Y, Z. The virtualized AMM $A|\boldsymbol{v}$ is defined by

$$(A|\boldsymbol{v})(x, w) = A(x, \frac{2w}{3}, \frac{w}{3} + 1) = x\frac{2w}{3}(\frac{w}{3} + 1) - 8 = 0,$$

with initial state $(2, 3)$. (The "+1" in the Z coordinate appears because 2 Y and 2 Z units are not evenly divisible into W units.)

Let $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, \ldots, y_m)$. Let $A(\boldsymbol{x}, \boldsymbol{y}) = A(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0$ be an $(n+m)$-dimensional AMM with initial state $(\boldsymbol{a}, \boldsymbol{b}) = (a_1, \ldots, a_n, b_1, \ldots, b_m)$. Let us create a virtual asset Z from $y_1, \ldots, y_m$, using the valuation $\boldsymbol{v} = (v_1, \ldots, v_m)$.

Let $c \in \mathbb{R}_{>0}$ be the largest value such that $\boldsymbol{b} - c\boldsymbol{v} \geq \boldsymbol{0}$. The value c is the number of Z assets in $\boldsymbol{b}$, and $\boldsymbol{r} = \boldsymbol{b} - c\boldsymbol{v}$ is the vector of residues if $\boldsymbol{b}$ is not evenly divisible into Z units. The virtualized AMM is given by

$$(A|\boldsymbol{v})(\boldsymbol{x}, z) = A(\boldsymbol{x}, v_1 c + r_1, \ldots, v_m c + r_m) = 0,$$

with initial state $(a_1, \ldots, a_n, c)$.

The next lemma says that in any AMM state, it is always possible to virtualize any set of assets.

Lemma 3.1.4. *Let A be an $(m + n)$-dimensional AMM in state $(\boldsymbol{a}, \boldsymbol{b})$, where $\boldsymbol{a} \in \mathbb{R}^m_{++}$, $\boldsymbol{b} \in \mathbb{R}^n_{++}$, and valuation $\boldsymbol{v} \in int(\Delta^n)$. We claim that for any $\boldsymbol{a}' \in \mathbb{R}^m_{++}$, there is a unique $t \geq 0$ such that $(\boldsymbol{a}', \boldsymbol{b} + t\boldsymbol{v})$ is a state of A.*

Proof. We seek $t \in \mathbb{R}$ such that $A(\boldsymbol{a}', \boldsymbol{b} + t\boldsymbol{v}) = 0$. There are several cases. If $A(\boldsymbol{a}', \boldsymbol{b}) = 0$, then $t = 0$ and we are done. Suppose $A(\boldsymbol{a}', \boldsymbol{b}) < 0$. If $A(\boldsymbol{a}', \boldsymbol{b} + \boldsymbol{v}) = 0$, then $t = 1$ and we are done. If $A(\boldsymbol{a}', \boldsymbol{b} + \boldsymbol{v}) < 0$, pick a vector $\boldsymbol{c} = (c_1, \ldots, c_n) \in \mathbb{R}^n_{>0}$ such that $A(\boldsymbol{a}', \boldsymbol{b} + \boldsymbol{c}) = 0$. Let $\epsilon > 0$, $s_i = \frac{c_i + \epsilon}{v_i}, 0 \leq i \leq n.$ and $s = \max_{0 \leq i \leq n} s_i$. It follows that $s\boldsymbol{v} > \boldsymbol{c} + \epsilon\boldsymbol{1}$, and $\boldsymbol{b} + s\boldsymbol{v} > \boldsymbol{b} + \boldsymbol{c} + \epsilon\boldsymbol{1}$. Since $A_{\boldsymbol{a}'}$ is strictly increasing, $A(\boldsymbol{a}', \boldsymbol{b} + s\boldsymbol{v}) > A_{\boldsymbol{a}'}(\boldsymbol{b} + \boldsymbol{c}) = 0$. Define $\alpha(t) : [0, 1] \to \mathbb{R}$ by $\alpha(t) = A_{\boldsymbol{a}'}(\boldsymbol{b} + \boldsymbol{v} + t(s-1)\boldsymbol{v})$. Because α is continuous, the intermediate value theorem guarantees a unique $t^* \in (0, 1)$ such that $\alpha(t^*) = 0$. Taking $t = (1 + t^*(s - 1))$ establishes the claim. If $A(\boldsymbol{a}', \boldsymbol{b} + \boldsymbol{v}) > 0$, let $s_i = \frac{c_i - \epsilon}{v_i}$ and $s = \min_{0 \leq i \leq n} s_i$, and the claim follows from a symmetric argument.

Suppose $A(\boldsymbol{a}', \boldsymbol{b}) > 0$. Pick a vector $\boldsymbol{c} = (c_1, \ldots, c_n) \in \mathbb{R}^n_{>0}$ such that $A(\boldsymbol{a}', \boldsymbol{b} - \boldsymbol{c}) = 0$. Let $\epsilon > 0$, $s_i = \frac{c_i + \epsilon}{v_i}, 0 \leq i \leq n.$ and $s = \max_{0 \leq i \leq n} s_i$. It follows that $s\boldsymbol{v} > \boldsymbol{c} + \epsilon\boldsymbol{1}$, and $\boldsymbol{b} - s\boldsymbol{v} < \boldsymbol{b} - \boldsymbol{c} - \epsilon\boldsymbol{1}$, so $A(\boldsymbol{a}', \boldsymbol{b} - s\boldsymbol{v}) < A(\boldsymbol{a}', (\boldsymbol{b} - \boldsymbol{c}) = 0$. Let $\alpha(t) : [0, 1] \to \mathbb{R}$ be $\alpha(t) = A(\boldsymbol{a}', (\boldsymbol{b}+)$. As before, the intermediate value theorem guarantees a unique $t^* \in (0, 1)$ such that $\alpha(t^*) = 0$. Taking $t = t^* s$ establishes the claim. □

Theorem 3.1.5. *Given an $(n + m)$-dimensional AMM $A(\boldsymbol{x}, \boldsymbol{y}) = 0$, and a valuation $\boldsymbol{v} \in int(\Delta^m)$, the virtualized $(A|\boldsymbol{v})(\boldsymbol{x}, z)$ is an $(n + 1)$-dimensional AMM.*

Proof. It is enough to check that $A|\boldsymbol{v}$ is differentiable, strictly increasing, and *upper*$(A|\boldsymbol{v})$ strictly convex. $(A|\boldsymbol{v})$ is differentiable because A is differentiable.

To show that $A|\boldsymbol{v}$ is strictly increasing, let $\boldsymbol{x}' \geq \boldsymbol{x}$ and $z' \gneqq z$.

$$\begin{aligned}(A|\boldsymbol{v})(x_1,\ldots,x_n,z) &= A(x_1,\ldots,x_n,v_1z+r_1,\ldots,v_mz+r_m)\\ &< A(x_1',\ldots,x_n',v_1z'+r_1,\ldots,v_mz'+r_m)\\ &= (A|\boldsymbol{v})(x_1',\ldots,x_n',z').\end{aligned}$$

To show that $\mathit{upper}(A|\boldsymbol{v})$ is strictly convex, pick distinct $(\boldsymbol{x},z)$ and $(\boldsymbol{x}',z')$ on the manifold: $(A|\boldsymbol{v})(\boldsymbol{x},z) = (A|\boldsymbol{v})(\boldsymbol{x}',z') = 0$. For $t \in (0,1)$,

$$\begin{aligned}(A|\boldsymbol{v})(t\boldsymbol{x}+(1-t)\boldsymbol{x}', tz+(1-t)z') &= A(t\boldsymbol{x}+(1-t)\boldsymbol{x}', \boldsymbol{v}(tz+(1-t)z')+\boldsymbol{r})\\ &= A(t\boldsymbol{x}+(1-t)\boldsymbol{x}', t(\boldsymbol{v}z+r)+(1-t)(\boldsymbol{v}z'+r))\\ &= A(t(\boldsymbol{x},\boldsymbol{v}z+r)+(1-t)(\boldsymbol{x}',\boldsymbol{v}z'+r)) > 0\end{aligned}$$

by the strict convexity of $\mathit{upper}(A)$. □

Stability for virtualization

Say we have some $(n+m)$-dimensional AMM $A(\boldsymbol{x},\boldsymbol{y}) = 0$ and a valuation $\boldsymbol{v} \in \mathrm{int}(\Delta^m)$. If this AMM begins in state $(\mathbf{a},\boldsymbol{b})$, we defined the the virtualized AMM to have the corresponding state so that $(A|\boldsymbol{v})(\mathbf{a},c) = A(\mathbf{a},c\boldsymbol{v}+\mathbf{r}) = A(\mathbf{a},\boldsymbol{b}) = 0$. We call this the virtualized AMM $(A|\boldsymbol{v})$ with respect to state $(\mathbf{a},\boldsymbol{b})$. Writing this out more generally for any state in the virtualized AMM we have $(A|\boldsymbol{v})(\boldsymbol{x},t) = A(\boldsymbol{x},\boldsymbol{b}+\boldsymbol{v}(t-c)) = 0$. Notice that this definition depends on $\boldsymbol{b}$ and $\boldsymbol{v}$. Also, c is a constant determined once we fix $\boldsymbol{b}$ and $\boldsymbol{v}$.

Since $(A|v)$ is an $(n+1)$-dimensional AMM, once we choose $\mathbf{a} \in \mathbb{R}^n_{++}$, we have $t = f(\mathbf{a})$. So the virtualized AMM can be described by the set of $(\boldsymbol{x}, f(\boldsymbol{x}))$ where $A(\boldsymbol{x},\boldsymbol{b}+\boldsymbol{v}(f(\boldsymbol{x})-c)) = 0$.

We have the following stability relationship between an AMM and its virtualization:

Lemma 3.1.6. *Suppose $(\boldsymbol{a}^*,\boldsymbol{b}^*)$ is the stable state on the AMM $A(\boldsymbol{x},\boldsymbol{y})$ for the valuation $(\boldsymbol{v},\mathbf{w})$. Then $(\boldsymbol{a}^*, f(\boldsymbol{a}^*))$ is the stable state with respect to the valuation $(\boldsymbol{v}, ||\mathbf{w}||_2^2)$ on the virtualized AMM $(A|\mathbf{w})(\boldsymbol{x},t)$ with respect to state $(\boldsymbol{a}^*,\boldsymbol{b}^*)$.*

Proof. Suppose that $(\boldsymbol{a}^*, f(\boldsymbol{a}^*))$ is not a stable state for $(\boldsymbol{v}, ||\mathbf{w}||_2^2)$. Namely, there is some different point $(\boldsymbol{a}, f(\boldsymbol{a})) \in A|\mathbf{w}$ where $\boldsymbol{v}\cdot\boldsymbol{a}+||\mathbf{w}||_2^2 f(\boldsymbol{a}) < \boldsymbol{v}\cdot\boldsymbol{a}^*+||\mathbf{w}||_2^2 f(\boldsymbol{a}^*)$. Now define $\boldsymbol{b} = \boldsymbol{b}^*+\mathbf{w}(f(\boldsymbol{a})-f(\boldsymbol{a}^*))$, which by the virtualization construction we have $(\boldsymbol{a},\boldsymbol{b}) \in A$. Then note we have

$$\begin{aligned}&\boldsymbol{v}\cdot\boldsymbol{a}+\mathbf{w}\cdot\boldsymbol{b} = \boldsymbol{v}\cdot\boldsymbol{a}+\mathbf{w}\cdot\boldsymbol{b}^*+\mathbf{w}\cdot\mathbf{w}(f(\boldsymbol{a})-f(\boldsymbol{a}^*))\\ &= \boldsymbol{v}\cdot\boldsymbol{a}+||\mathbf{w}||_2^2 f(\boldsymbol{a})-||\mathbf{w}||_2^2 f(\boldsymbol{a}^*)+\mathbf{w}\cdot\boldsymbol{b}^*\\ &< \boldsymbol{v}\cdot\boldsymbol{a}^*+||\mathbf{w}||_2^2 f(\boldsymbol{a}^*)-||\mathbf{w}||_2^2 f(\boldsymbol{a}^*)+\mathbf{w}\cdot\boldsymbol{b}^* \qquad \text{(By assumption)}\\ &= \boldsymbol{v}\cdot\boldsymbol{a}^*+\mathbf{w}\cdot\boldsymbol{b}^*\end{aligned}$$

But this is a contradiction since $(\boldsymbol{a}^*,\boldsymbol{b}^*)$ is the stable state for $(\boldsymbol{v},\mathbf{w})$, so we have the result. □

3.2 Sequential Composition

AMMs are intended to be composed. A Uniswap AMM typically converts between an ERC-20 token and ether cryptocurrency. To convert, say, apple tokens to banana tokens, one would first convert apples to ether, then ether to bananas. Bancor uses a proprietary BNT token for the same purpose. Some form of composition seems to be essential to making AMMs useful, but we will see that while there are many ways in which AMMs might be composed, not all of them make sense. The most basic property one would demand is closure under composition: the result of composing two AMMs should itself be an AMM.

Being closed under composition should not be taken for granted. For example, consider two Uniswap-like constant-product AMMs: $A = (x, c_A/x)$, initialized in state $(a, 1/a)$, and $B = (y, c_B/y)$, initialized in state $(b, 1/b)$. If we were to define their composition as $A \otimes B = (x, h(x))$, where

$$h(x) = \frac{1}{b + \frac{1}{a} - \frac{1}{x}} = \frac{ax}{x - a + abx}. \tag{3.1}$$

The set of constant-product AMMs is thus not closed under composition. Thus we need an alternative notion of composition in this case.

3.2.1 One-to-One Composition

We first consider the result of composing 2-dimensional AMMs, that is, AMMs that trade between two asset types.

Let $A = (x, f(x))$, initialized to $(a, f(a))$, and $B = (y, g(y))$, initialized to $(b, g(b))$. A trades between asset types X and Y, and B between Y and Z. Their composition, initialized to $(a, g(b))$, trades between X and Z.

Operationally, composition is defined as follows.

- Move A from state $(a, f(a)$ to state $(x, f(x))$, yielding profit-loss vector $(a - x, f(a) - f(x))$.
- Add $f(a) - f(x)$ to the Y balance of B, yielding new state $(b + f(a) - f(x), g(b + f(a) - f(x))$.

This trade takes the composition from $(a, g(b))$ to $(x, g(b + f(a) - f(x)))$. Let $h(x) = g(b + f(a) - f(x))$. The composition $A \otimes B$ is given in the form $(x, h(x))$. Because f, g are differentiable:

Lemma 3.2.1. *$(A \otimes B)(x, y)$ is differentiable.*

Lemma 3.2.2. *$(A \otimes B)(x, y)$ is strictly increasing.*

Proof. We show that if $(x', y') \geq (x, y)$, where at least one coordinate is strictly greater, then $(A \otimes B)(x', y') > (A \otimes B)(x, y)$. Recall that f and g are strictly decreasing by hypothesis. There are two cases. First, suppose $x' > x$

and $y' \geq y$.

$$x' > x$$
$$f(x') < f(x)$$
$$b + f(a) - f(x') > b + f(a) - f(x)$$
$$g(b + f(a) - f(x')) < g(b + f(a) - f(x))$$
$$y - g(b + f(a) - f(x')) > y - g(b + f(a) - f(x))$$
$$y' - g(b + f(a) - f(x')) > y - g(b + f(a) - f(x))$$
$$(A \otimes B)(x, y') > (A \otimes B)(x, y)$$

The second case, where $x' \geq x$ and $y' > y$ is similar. □

Lemma 3.2.3. *The upper contour set upper$(A \otimes B)$ is strictly convex.*

Proof. Since *upper*$(A \otimes B) = epi(h)$, by Lemma 2.4.7, it is enough to check the strict convexity of h. Pick two different x and x'. Recall that for $t \in (0,1), f(tx + (1-t)x') < tf(x) + (1-t)f(x')$, and similarly for $g(y)$ by hypothesis. For $t \in (0,1)$,

$$f((1-t)x + tx') < (1-t)f(x) + tf(x')$$
$$b + f(a) - f((1-t)x + tx') > (1-t)(b + f(a) - f(x)) + t(b + f(a) - f(x'))$$
$$g(b + f(a) - f((1-t)x + tx')) < g((1-t)(b + f(a) - f(x)) + t(b + f(a) - f(x')))$$
$$h((1-t)x + tx) < (1-t)h(x) + th(x')$$

which gives us the result. □

The next lemma tells us how stability relates to composition:

Lemma 3.2.4. *Say (v, w, v') is some valuation. If $(a^*, f(a^*))$ is the stable state on the AMM $(x, f(x))$ for the valuation (v, w) and $(b^*, g(b^*))$ is the stable state on the AMM $(y, g(y))$ for the valuation (w, v'), then $(a^*, h(a^*))$ is the stable state for the valuation (v, v').*

Proof. Assume for contradiction that $(a^*, h^*(a))$ is not a stable state for (v, v'). So there is some $x \neq a$ where $(x, h(x))$ is the stable state. Note that by assumption for $x \neq a$ and $y \neq b$ we have

$$va + wf(a) < vx + wf(x) \tag{1}$$
$$wb + v'g(b) < wy + v'g(y) \tag{2}$$

Additionally by assumption we have

$$\begin{aligned}
&vx + v'h(x) < va + v'h(a) = va + v'g(b + f(a) - f(a) = va + v'g(b) \\
&= va + v'g(b) = va + wf(a) - wf(a) + wb - wb + v'g(b) \\
&< vx + wf(x) - wf(a) + wy + v'g(y) - wb && \text{(By (1),(2))} \\
&= vx + w(f(x) - f(a)) + w(y - b) + v'g(y) \\
&= vx + w(f(x) - f(a)) + w(f(a) - f(x)) + v'g(b + f(a) - f(x)) \\
&&& \text{(Taking } y = b + f(a) - f(x) \neq b \text{, possible since } f \text{ strictly decreasing)} \\
&= vx + v'h(x)
\end{aligned}$$

a contradiction.

□

The converse is not true, as the following example shows:

Consider two constant product AMMs A $(x, \frac{1}{x})$ and B $(y, \frac{1}{y})$ both initially in state $(1, 1)$. So the composed AMM is given by $A \otimes B$ $(x, \frac{x}{2x-1})$ with state $(1, 1)$. Now if $(v, w, v') = (\frac{1}{4}, \frac{1}{2}, \frac{1}{4})$, $(1, 1)$ is the stable state of $A \otimes B$ with respect to (v, w, v'). However, the stable state for (v, w) on A is $(\sqrt{2}, \frac{1}{\sqrt{2}})$ and (w, v') on B is $(\frac{1}{\sqrt{2}}, \sqrt{2})$.

3.2.2 Many-to-One Composition

AMM A trades asset types $X_1, \ldots, X_m, Z$, with initial state $(\boldsymbol{a}, f(\boldsymbol{a}))$, where $\boldsymbol{a} = (a_1, \ldots, a_m)$, and $f(\boldsymbol{a})$ is the explicit function defining the Z coordinate in terms of the others.

AMM B trades asset types $Z, Y_1, \ldots, Y_n$, with initial state $(c, \boldsymbol{b}, g(c, \boldsymbol{b}))$, where $\boldsymbol{b} = (b_1, \ldots, b_{n-1})$, and $g(c, \boldsymbol{b})$ is the implicit function defining the Y_n coordinate in terms of the others.

The Z asset flows between A and B but is not directly accessible to traders. The composition $A \otimes B$ trades asset types $X_1, \ldots, X_m, Y_1, \ldots, Y_n$, with initial state $(\boldsymbol{a}, \boldsymbol{b}, h(\boldsymbol{a}, \boldsymbol{b}))$, for h to be defined.

Operationally, the composition works as follows. The trader changes each a_i to x_i, $0 \leq i \leq m$, and each b_i to y_i, $0 \leq i \leq n-1$. Let $\boldsymbol{x} = (x_1, \ldots, x_m)$ and $\boldsymbol{y} = (y_1, \ldots, y_{n-1})$. The new state of A is $(\boldsymbol{x}, f(\boldsymbol{x}))$. The amount of Z that flows from A to B is $f(\boldsymbol{a}) - f(\boldsymbol{x})$. The new state of B is $(c + f(\boldsymbol{a}) - f(\boldsymbol{x}), \boldsymbol{y}, g(c + f(\boldsymbol{a}) - f(\boldsymbol{x})))$. The new state of $A \otimes B$ is $(\boldsymbol{x}, \boldsymbol{y}, g(c + f(\boldsymbol{a}) - f(\boldsymbol{x}), \boldsymbol{y}))$

Define $h(\boldsymbol{x}, \boldsymbol{y}) = g(c + f(\boldsymbol{a}) - f(\boldsymbol{x}), \boldsymbol{y})$. $(A \otimes B)(\boldsymbol{x}, \boldsymbol{y}, z) = z - h(\boldsymbol{x}, \boldsymbol{y})$.

Lemma 3.2.5. *$(A \otimes B)(\boldsymbol{x}, \boldsymbol{y}, z)$ is differentiable.*

Proof. Immediate because f, g are differentiable by hypothesis. □

Lemma 3.2.6. *$(A \otimes B)(\boldsymbol{x}, \boldsymbol{y}, z)$ is strictly increasing.*

Proof. We show that if $(\boldsymbol{x}', \boldsymbol{y}', z') \gneqq (\boldsymbol{x}, \boldsymbol{y}, z)$, then $(A \otimes B)(\boldsymbol{x}', \boldsymbol{y}', z') > (A \otimes B)(\boldsymbol{x}, \boldsymbol{y}, z)$. There are two cases. First, suppose $(\boldsymbol{x}', \boldsymbol{y}') \gneqq (\boldsymbol{x}, \boldsymbol{y})$ and $z' \geq z$.

$$(\boldsymbol{x}', \boldsymbol{y}') \gneqq (\boldsymbol{x}, \boldsymbol{y})$$
$$f(\boldsymbol{x}', \boldsymbol{y}') < f(\boldsymbol{x}, \boldsymbol{y})$$
$$\boldsymbol{b} + f(\boldsymbol{a}) - f(\boldsymbol{x}', \boldsymbol{y}') > \boldsymbol{b} + f(\boldsymbol{a}) - f(\boldsymbol{x}, \boldsymbol{y})$$
$$g(\boldsymbol{b} + f(\boldsymbol{a}) - f(\boldsymbol{x}', \boldsymbol{y}')) < g(\boldsymbol{b} + f(\boldsymbol{a}) - f(\boldsymbol{x}, \boldsymbol{y}))$$
$$z - g(\boldsymbol{b} + f(\boldsymbol{a}) - f(\boldsymbol{x}', \boldsymbol{y}')) > z - g(\boldsymbol{b} + f(\boldsymbol{a}) - f(\boldsymbol{x}, \boldsymbol{y}))$$
$$z' - g(\boldsymbol{b} + f(\boldsymbol{a}) - f(\boldsymbol{x}', \boldsymbol{y}')) > z - g(\boldsymbol{b} + f(\boldsymbol{a}) - f(\boldsymbol{x}, \boldsymbol{y}))$$
$$(A \otimes B)(\boldsymbol{x}', \boldsymbol{y}', z') > (A \otimes B)(\boldsymbol{x}, \boldsymbol{y}, z)$$

The second case, where $\boldsymbol{x}' \geq \boldsymbol{x}$ and $z' > z$ is similar. □

Lemma 3.2.7. *upper*$(A \otimes B)$ *is strictly convex.*

Proof. Again it is enough to show h is strictly convex. Pick two different points $(\boldsymbol{x}, \boldsymbol{y})$ and $(\boldsymbol{x}', \boldsymbol{y}')$. Recall that f and g are strictly convex by hypothesis. For $t \in (0, 1)$,

$$f((1-t)(\boldsymbol{x}, \boldsymbol{y}) + t(\boldsymbol{x}', \boldsymbol{y}')) < (1-t)f(\boldsymbol{x}, \boldsymbol{y}) + tf(\boldsymbol{x}', \boldsymbol{y}')$$
$$b + f(a) - f((1-t)(\boldsymbol{x}, \boldsymbol{y}) + t(\boldsymbol{x}', \boldsymbol{y}')) > (1-t)(b + f(a) - f(\boldsymbol{x}, \boldsymbol{y})) + t(b + f(a) - f(\boldsymbol{x}', \boldsymbol{y}'))$$
$$g(b + f(a) - f((1-t)(\boldsymbol{x}, \boldsymbol{y}) + t(\boldsymbol{x}', \boldsymbol{y}'))) < g((1-t)(b + f(a) - f(\boldsymbol{x}, \boldsymbol{y})) + t(b + f(a) - f(\boldsymbol{x}', \boldsymbol{y}')))$$
$$h((1-t)(\boldsymbol{x}, \boldsymbol{y}) + t(\boldsymbol{x}', \boldsymbol{y}')) < (1-t)h(\boldsymbol{x}, \boldsymbol{y}) + th(\boldsymbol{x}', \boldsymbol{y}'))$$

□

The next result generalizes the result show for the one-to-one case.

Theorem 3.2.8. *Say* $(\boldsymbol{v}, w, \boldsymbol{v}')$ *is some valuation,* $\boldsymbol{v} \in \mathbb{R}^n$*,* $\boldsymbol{v}' \in \mathbb{R}^m$*. If* $(\boldsymbol{a}^*, f(\boldsymbol{a}^*))$ *is the stable state on the AMM* $(\boldsymbol{x}, f(\boldsymbol{x}))$ *for the valuation* $(\boldsymbol{v}, w)$ *and* $(c, \boldsymbol{b}^*, g(c, \boldsymbol{b}^*))$ *is the stable state on the AMM* $(z, \boldsymbol{y}, g(z, \boldsymbol{y}))$ *for the valuation* $(w, \boldsymbol{v}')$*, then* $(\boldsymbol{a}^*, \boldsymbol{b}^*, h(\boldsymbol{a}^*, \boldsymbol{b}^*))$ *is the stable state for the valuation* $(\boldsymbol{v}, \boldsymbol{v}')$*.*

Proof. Assume for contradiction that $(\boldsymbol{a}^*, \boldsymbol{b}^*, h(\boldsymbol{a}^*, \boldsymbol{b}^*))$ is not a stable state for $(\boldsymbol{v}, \boldsymbol{v}')$. So there is some $(\boldsymbol{x}, \boldsymbol{y}) \neq (\boldsymbol{a}^*, \boldsymbol{b}^*)$ where $(\boldsymbol{x}, \boldsymbol{y}, h(\boldsymbol{x}, \boldsymbol{y}))$ is the stable state. We write $\boldsymbol{v}' = (\boldsymbol{v}'_{m-1}, v'_m)$ to separate out the first $m-1$ components from the m-th component. Note that by assumption for $\boldsymbol{x} \neq \boldsymbol{a}^*$ and $(z, \boldsymbol{y}) \neq (c, \boldsymbol{b}^*)$ we have

$$\boldsymbol{v} \cdot \boldsymbol{a}^* + wf(\boldsymbol{a}^*) < \boldsymbol{v} \cdot \boldsymbol{x} + wf(\boldsymbol{x}) \tag{1}$$
$$wc + \boldsymbol{v}'_{m-1} \cdot \mathbf{b} + v'_m g(c, \mathbf{b}) < wz + \boldsymbol{v}'_{m-1} \cdot \mathbf{y} + v'_m g(z, \mathbf{y}) \tag{2}$$

Additionally by assumption we have

$$\begin{aligned}
&\boldsymbol{v}\cdot\boldsymbol{x}+\boldsymbol{v}'_{n-1}\cdot\boldsymbol{y}+v'_m h(\boldsymbol{x},\boldsymbol{y}) < \boldsymbol{v}\cdot\boldsymbol{a}^*+\boldsymbol{v}'_{m-1}\cdot\boldsymbol{b}^*+v'_m h(\boldsymbol{a}^*,\boldsymbol{b}^*)\\
&=\boldsymbol{v}\cdot\boldsymbol{a}^*+wf(\boldsymbol{a}^*)-wf(\boldsymbol{a}^*)+wc-wc+\boldsymbol{v}'_{m-1}\cdot\boldsymbol{b}^*+v'_m g(c,\boldsymbol{b}^*)\\
&<\boldsymbol{v}\cdot\boldsymbol{x}+wf(\boldsymbol{x})-wf(\boldsymbol{a}^*)+wz+\boldsymbol{v}'_{m-1}\cdot\boldsymbol{y}+v'_m g(z,\boldsymbol{y})-wc && \text{(By (1),(2))}\\
&=\boldsymbol{v}\cdot\boldsymbol{x}+\boldsymbol{v}'_{m-1}\cdot\boldsymbol{y}+v'_m g(z,\boldsymbol{y}) && (z=c+f(\boldsymbol{a}^*)-f(\boldsymbol{x}))\\
&=\boldsymbol{v}\cdot\boldsymbol{x}+\boldsymbol{v}'_{m-1}\cdot\boldsymbol{y}+v'_m g(c+f(\boldsymbol{a}^*)-f(\boldsymbol{x}),\boldsymbol{y})\\
&=\boldsymbol{v}\cdot\boldsymbol{x}+\boldsymbol{v}'_{m-1}\cdot\boldsymbol{y}+v'_m h(\boldsymbol{x},\boldsymbol{y})
\end{aligned}$$

a contradiction.

□

3.2.3 Many-to-Many Composition

What does it mean to compose two AMMs that share multiple assets? We will see that this definition requires some care.

Here is the most obvious definition. Let W, X, Y, Z be asset types. Consider two 3-dimensional AMMs, A defined by $wxy = 1$ and B defined by $xyz = 8$. where A is in state $(1, 1, 1)$ and B in state $(2, 2, 2)$. We want to compose them into a 2-dimensional AMM $(A \oplus B)$ between W and Z treating X and Y as "hidden" assets.

A trader adds $dx > 0$ units of X to $(A \oplus B)$:

1. pick any $dy, dz \leq 0$ satisfying $(1 + dx)(1 + dy)(1 + dz) = 1$, yielding a profit-loss vector on Y, Z of (dy, dz).
2. Subtract this profit-loss vector from the first two components of B, then solve for $dw \leq 0$ so that $(2 - dy)(2 - dz)(2 + dw) = 8$.

More generally, let A be initialized in (a, b, c), B in (b', c', d). and $A \oplus B$ in (a, d). Then (w, z) is in $A \oplus B$ if there exist $dx, dy \leq 0$ (the assets transferred) such that $(w, b + dx, c + dy)$ is in A and $(b' - dx, c' - dy, z)$ is in B.

Here is why this naïve definition of composition is flawed. As before, $A(w, x, y) = wxy - 1 = 0$ starts in state $(1, 1, 1)$, $B(x, y, z) = xyz - 8 = 0$ starts in $(2, 2, 2)$. Let $dw = 3$, $dx = -\frac{1}{2}$, $dy = -\frac{1}{2}$, so dz satisfies:

$$(2 - dx)(2 - dy)(2 + dz) = \frac{25}{4}(2 + dz) = 2$$

implying $dz = -42/25$. If instead $dw = 3$, $dx = -\frac{1}{4}$, $dy = -\frac{2}{3}$, then dz satisfies:

$$(2 - dx)(2 - dy)(2 + dz) = 6(2 + dz) = 2$$

meaning $dz = -5/3$. So both $(4, \frac{8}{25})$ and $(4, \frac{1}{3})$ are valid states in the naïve composition, violating the requirement that each coordinate is a function of the others.

This shows that AMMs joined by more than one hidden asset type provide too many degrees of freedom to compose easily. In a practical sense, however, if two AMMs agree on a valuation for the hidden assets, then it makes

sense to transfer them in proportion to their agreed-upon relative values. To define composition for AMMs with multiple hidden assets, we create a single virtual asset from a convex combination of the hidden assets (Section 3.1.2). Although any convex combination would work, it makes sense to use the current market valuation, if one exists. By reducing the number of hidden assets to one, virtualization reduces many-to-many composition to many-to-one composition, analyzed in Section 3.2.2.

We state a final stability result now under this most general type of sequential composition.

Theorem 3.2.9. *Say $(\boldsymbol{v}, \mathbf{w}, \boldsymbol{v}')$ is some valuation, $\boldsymbol{v} \in \mathbb{R}^n_{++}, \mathbf{w} \in \mathbb{R}^k_{++}, \boldsymbol{v}' \in \mathbb{R}^m_{++}$. Suppose that $(\boldsymbol{a}^*, \boldsymbol{b}^*)$ is the stable state for $(\boldsymbol{v}, \mathbf{w})$ on the AMM $A(\boldsymbol{x}, \boldsymbol{y})$ and $(\boldsymbol{c}^*, \mathbf{d}^*, e^*)$ is the stable state for $(\mathbf{w}, \boldsymbol{v}')$ on the AMM $B(\boldsymbol{z}, \mathbf{r}, q)$. If we write $A|\mathbf{w}$ as $(\boldsymbol{x}, f(\boldsymbol{x}))$ and $B|\mathbf{w}$ as $(\boldsymbol{z}, \mathbf{r}, g(\boldsymbol{z}, \mathbf{r}))$, then $(\boldsymbol{a}^*, \mathbf{d}^*, h(\boldsymbol{a}^*, \mathbf{d}^*))$, where $e^* = h(\boldsymbol{a}^*, \mathbf{d}^*)$ is the stable state on $(A|\mathbf{w}) \oplus (B|\mathbf{w})$ with respect to $(\boldsymbol{v}, \boldsymbol{v}')$.*

Proof. Applying Lemma 3.1.6 twice, we get $(\boldsymbol{a}^*, f(\boldsymbol{a}^*))$ stable wrt $(\boldsymbol{v}, ||\mathbf{w}||_2^2)$ and $(t^*, \mathbf{d}^*, g(t^*, \mathbf{d}^*))$ is stable wrt $(||\mathbf{w}||_2^2, \boldsymbol{v})$ where $e^* = g(t^*, \mathbf{d}^*)$. Now applying Theorem 3.2.8 we get $(\boldsymbol{a}^*, \mathbf{d}^*, h(\boldsymbol{a}^*, \mathbf{d}^*))$ is stable on $(A|\mathbf{w}) \oplus (B|\mathbf{w})$ with respect to $(\boldsymbol{v}, \boldsymbol{v}')$.

□

This result, while seemingly rather technical is further evidence that our notion of sequential composition is a reasonable one. Namely, it says stability isn't affected by the internal assets between the AMMs being composed.

3.3 Parallel Composition

Parallel composition arises when a trader is faced with multiple AMMs, but wants to treat them them as if they were a single AMM. In sequential composition, the composed AMMs exchange "hidden" assets . In parallel composition, the composed AMMs compete for overlapping assets.

Suppose Alice wants to trade asset X for asset Y. Bob and Carol both offer AMMs to convert from X to Y. Bob's AMM is $B(x, y) = x^2y = 3/4$ in state $(1, 3/4)$, while Carol's AMM is $C(x, y) = xy = 1$ in state $(1, 1)$. Alice would like to compose the two AMMs and treat them as one AMM. Bob provides a better initial rate of exchange for small trades, but Carol provides less slippage for large trades. One can check that if Alice converts 1 unit of X, she gets more Y assets from Bob than from Carol, while if she converts 3 units, she gets more from Carol.

A rational Alice will split her assets between B and C to maximize her return. Suppose $A(x, y)$ is in state $(a, f(a))$ and $B(x, y)$ in $(b, g(b))$. Alice splits her d assets, transferring td to Bob's AMM and $(1 - t)d$ to Carol's, returning

$$f(a) - f(a + tx) + g(b) - g(b + (1 - t)x)$$

units of Y. Let $h(x) = f(a+tx)+g(b+(1-t)x)$ Define the *parallel composition* of B and C with respect to $v = (t, 1-t)$ to be $(B||C)(x, y) = y - h(x) = 0$. We will also use the notation $h_t(x)$, where $h_t(x) = f(a + tx) + g(b + (1 - t)x)$. Note that this is defined for each $t \in [0, 1]$.

Lemma 3.3.1. *$(B||C)(x, y)$ is a 2-dimensional AMM.*

Proof. $(B||C)(x,y)$ is differentiable because f and g are differentiable. To check that $(B||C)(x,y)$ is strictly increasing, let $x' \geq x$ and $y' \geq y$ where at least one inequality is strict. For the first case, suppose $x' > x$ and $y' \geq y$.

$$f(a+tx') < f(a+tx)$$
$$g(b+(1-t)x') < g(b+(1-t)x)$$
$$f(a+tx') + g(b+(1-t)x') < f(a+tx) + g(b+(1-t)x)$$
$$h(x') < h(x)$$
$$y - h(x') > y - h(x)$$
$$y' - h(x') > y - h(x)$$

The case where $x' \geq x$ and $y' > y$ is similar.

To check that $\mathit{upper}((B||C)(x,y))$ is strictly convex, we can verify h is strictly convex. Pick distinct x, x'. For $s \in (0,1)$,

$$sf(a+tx) + (1-s)f(a+tx') > f(s(a+tx) + (1-s)(a+tx'))$$
$$sg(b+(1-t)x) + (1-s)g(b+(1-t)x') > g(s(b+(1-t)x) + (1-s)(b+(1-t)x'))$$
$$s(f(a+tx) + g(b+(1-t)x)) + (1-s)(f(a+tx') + g(b+(1-t)x')) > f(s(a+tx) + (1-s)(a+tx')) + g(s(b+(1-t)x) + (1-s)(b+(1-t)x'))$$
$$sh(x) + (1-s)h(x') > h(sx + (1-s)x')$$

which is what we want. □

Parallel composition is well-defined for any valuation, but what valuation should a rational Alice pick? Differentiating with respect to t yields

$$-xf'(a+tx) + xg'(b+(1-t)x) = 0 \tag{3.2}$$
$$xf'(a+tx) = xg'(b+(1-t)x) \tag{3.3}$$
$$f'(a+tx) = g'(b+(1-t)x). \tag{3.4}$$

Alice maximizes her return when she splits her assets so that Bob and Carol end up offering the same rate. Informally, if Bob had ended up providing a better rate, then Alice should have given him a larger share. If there is no $t \in (0,1)$ that satisfies Equation 3.2, Alice should give all her assets to the AMM with the better rate.

How should parallel composition be defined for AMMs with multiple asset types? Suppose Alice has some combination $\boldsymbol{d}$ of assets in $X_1, \ldots, X_p$ that she wants to convert into some combination of assets in $Y_1, \ldots, Y_q$. Alice has a choice of two alternative AMMs: $B(x_1, \ldots, x_p, y_1, \ldots, y_q)$ and $C(x_1, \ldots, x_p, y_1, \ldots, y_q)$. One sensible way to define parallel composition is through asset virtualization. Alice's input asset vector $\boldsymbol{d}$ induces a valuation:

$$(\frac{d_1}{\sum_{i=0}^{p} d_i}, \ldots, \frac{d_p}{\sum_{i=0}^{p} d_i}).$$

which can be used to define a virtual asset X from $X_1, \ldots, X_p$. Alice chooses a valuation $\boldsymbol{v}$ for her $Y_1, \ldots, Y_q$ outputs (say the market valuation) which can be used to define a virtual asset Y from $Y_1, \ldots, Y_q$. After asset virtualization, the alternative AMMs have the form: $\tilde{B}(x, y)$ and $\tilde{C}(X, Y)$, and the definition of parallel composition proceeds as before.

The next result is a stability result tells us how a trader should behave if confronted with two AMMs being composed in parallel, when they are both at stable states.

Lemma 3.3.2. *Let (v, v') be some valuation, and let A be the AMM $(x, f(x))$ and B be the AMM $(y, g(y))$. Say that $(a, f(a))$ and $(b, g(b))$ are both stable states wrt (v, v'). Then $(0, h_t(0))$ is the stable state on $A||B$ with respect to (v, v') for all $t \in \mathbb{R}$.*

Proof. By assumption we have

$$\begin{aligned} va + v'f(a) &< v(a + tx) + v'f(a + tx) && (\forall t \in \mathbb{R}) \\ \implies v'f(a) &< tvx + v'f(a + tx) && (1) \end{aligned}$$

and

$$\begin{aligned} vb + v'g(b) &< v(b + (1 - t)x) + v'g(b + (1 - t)x) && (\forall t \in \mathbb{R}) \\ \implies v'g(b) &< (1 - t)vx + v'g(b + (1 - t)x) \end{aligned}$$

So we get

$$\begin{aligned} v0 + v'h_t(0) &= v'h_t(0) = v'f(a) + v'g(b) \\ &< tvx + v'f(a + tx) + (1 - t)vx + v'g(b + (1 - t)x) && \text{(By (1),(2))} \\ &= vx + v'(f(a + tx) + g(b + (1 - t)x)) = vx + v'h(x) \end{aligned}$$

so $(0, h_t(0))$ is a stable state for (v, v'). □

If $(\boldsymbol{x}, f(\boldsymbol{x}))$ and $(\boldsymbol{y}, g(\boldsymbol{y}))$ are two n-dimensional AMMs in states $(\boldsymbol{a}, f(\boldsymbol{a}))$ and $(\boldsymbol{b}, g(\boldsymbol{b}))$, we can do parallel composition as follows. Let $\mathbf{t} \in [0, 1]^{n-1}$ and define $h_{\mathbf{t}}(\boldsymbol{x}) = f(\boldsymbol{a} + \mathbf{t} * \boldsymbol{x}) + g(\boldsymbol{a} + \mathbf{t} * \boldsymbol{x})$, where $*$ is component-wise multiplication.

Lemma 3.3.3. *Let $(\boldsymbol{v}, v')$ be some valuation, and let A be the AMM $(\boldsymbol{x}, f(\boldsymbol{x}))$ and B be the AMM $(\boldsymbol{y}, g(\boldsymbol{y}))$. Say that $(\boldsymbol{a}, f(\boldsymbol{a}))$ and $(\boldsymbol{b}, g(\boldsymbol{a}))$ are both stable states wrt $(\boldsymbol{v}, v')$. Then $(\mathbf{0}, h_{\mathbf{t}}(\mathbf{0}))$ is the stable state on $A||B$ with respect to $(\boldsymbol{v}, v')$ for all $\mathbf{t} \in \mathbb{R}^{n-1}$.*

Proof. By assumption we have

$$\begin{aligned} \boldsymbol{v} \cdot \boldsymbol{a} + v'f(\boldsymbol{a}) &< \boldsymbol{v} \cdot (\boldsymbol{a} + \mathbf{t} * \boldsymbol{x}) + v'f(\boldsymbol{a} + \mathbf{t} * \boldsymbol{x}) && (\forall t \in \mathbb{R}) \\ \implies v'f(\boldsymbol{a}) &< (\mathbf{t} * \boldsymbol{x}) \cdot \boldsymbol{v} + v'f(\boldsymbol{a} + \mathbf{t} * \boldsymbol{x}) && (1) \end{aligned}$$

and

$$\boldsymbol{v} \cdot \boldsymbol{b} + v' f(\boldsymbol{b}) < \boldsymbol{v} \cdot (\boldsymbol{b} + \mathbf{t} * \boldsymbol{y}) + v' g(\boldsymbol{b} + \mathbf{t} * \boldsymbol{y}) \qquad (\forall t \in \mathbb{R})$$

$$\implies v' f(\boldsymbol{b}) < (\mathbf{t} * \boldsymbol{y}) \cdot \boldsymbol{v} + v' g(\boldsymbol{b} + \mathbf{t} * \boldsymbol{y}) \qquad (2)$$

So we get

$$\begin{aligned}
&\boldsymbol{v} \cdot \mathbf{0} + v' h_t(\mathbf{0}) = v' h_t(\mathbf{0}) = v' f(\boldsymbol{a}) + v' g(\boldsymbol{b}) \\
&< (\mathbf{t} * \boldsymbol{x}) \cdot \boldsymbol{v} + v' f(\boldsymbol{a} + \mathbf{t} * \boldsymbol{x}) + (\mathbf{1} - \mathbf{t}) * \boldsymbol{x} \cdot \boldsymbol{v} + v' g(\boldsymbol{b} + (\mathbf{1} - \mathbf{t}) * \boldsymbol{x}) \qquad \text{(By (1),(2))} \\
&= \boldsymbol{v} \cdot \boldsymbol{x} + v'(f(\boldsymbol{a} + \mathbf{t} \cdot \boldsymbol{x}) + g(\boldsymbol{b} + (\mathbf{1} - \mathbf{t}) \cdot \boldsymbol{x})) = \boldsymbol{v} \cdot \boldsymbol{x} + v' h(\boldsymbol{x})
\end{aligned}$$

so $(0, h_t(0))$ is a stable state for $(\boldsymbol{v}, v')$. □

Most generally, if we have two AMMs $A(\boldsymbol{x}, \mathbf{z})$ and $B(\boldsymbol{y}, \mathbf{z}')$, and some valuation $\mathbf{w}$, we can write $A|\mathbf{w}$ as $(\boldsymbol{x}, f(\boldsymbol{x}))$ and $B|\mathbf{w}$ as $(\boldsymbol{y}, g(\boldsymbol{y}))$. We then have parallel composition as defined as before.

Theorem 3.3.4. *Let $(\boldsymbol{v}, \boldsymbol{v}')$ be some valuation, and let $A(\boldsymbol{x}, \mathbf{z})$ and $B(\boldsymbol{y}, \mathbf{z}')$ be two AMMs. Say $(\boldsymbol{a}^*, \boldsymbol{b}^*)$ is the stable state for A and $(\mathbf{c}^*, \mathbf{d}^*)$ is the stable state for B, both with respect to $(\boldsymbol{v}, \boldsymbol{v}')$.*

Proof. Lemma 3.1.6 tells us $(\boldsymbol{a}^*, f(\boldsymbol{a}^*))$ and $(\boldsymbol{c}^*, g(\boldsymbol{c}^*))$ are stable on $A|\boldsymbol{v}'$ and $B|\boldsymbol{v}'$, both with respect to $(\boldsymbol{v}, ||\boldsymbol{v}'||_2^2)$. The previous lemma then gives us $(\mathbf{0}, h_{\mathbf{t}}(\mathbf{0}))$ is the stable state for $(A|\boldsymbol{v}')||(B|\boldsymbol{v}')$ with respect to the valuation $(\boldsymbol{v}, ||\boldsymbol{v}'||_2^2)$. □

Chapter 4

Comparing AMM Designs

In Section 3 we focused on how traders interact with AMMs. On the other side of each trade, *liquidity providers* (or *providers*) fund the AMMs, lending them reserves to be traded, intending to profit from fees charged for trades.

The interaction between traders and providers is a kind of alternating game: providers fund the AMMs and set fee rates, a trader executes a trade across multiple AMMs as a single atomic transaction, then providers can change AMM funding levels, re-invest or withdraw fees, and so on back and forth. Often we can model providing liquidity or charging fees as adjusting the constant in the AMM's level-set representation. The trader executes a trade where $A(\boldsymbol{x}) = c$, and after that trade providers change the AMM to $A(\boldsymbol{x}) = c'$, where the change from c to c' reflects fees charged or liquidity added or removed.

Throughout Section 4, we will restrict our attention to 2D AMMs. Additionally, when necessary we will assume that the underlying AMM is C^k for a sufficiently large $k \geq 0$. The level of differentiability will be explicitly mentioned in each appropriate context.

In Section 4.1, we propose several relevant measures that quantify the cost for traders and providers. We then show that these costs often cannot be fully eliminated. Hence, they serve more as a way to quantify tradeoffs between different AMM designs. Finally, we show in Section 4.2 how these various measures change under our previously defined notions of composition.

4.1 AMM Measures

In the following sections (in reference to an arbitrary AMM), we will use $\Phi(v) = (\phi(v), f(\phi(v)))$ to denote the stable state map. Similary, the inverse map will be denoted by $\Psi(x) = (\psi(x), 1 - \psi(x))$ where $\psi(x) = v$.

4.1.1 AMM Capitalization

Let $(v, 1 - v)$ be a valuation with stable state $(x, f(x))$. What is a useful way to define the *capitalization* (total value) of an AMM's holdings? It may be appealing to pick one asset to act as *numéraire*, computing the AMM's

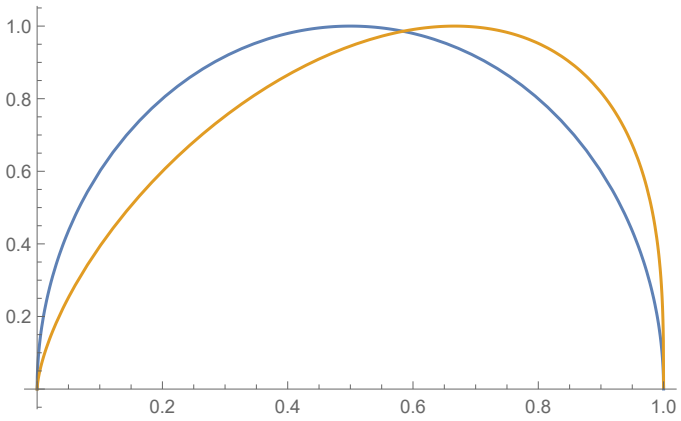

Figure 4.1: Balanced Capitalization of AMMs $(x, 1/x)$ (symmetric, blue) and $(x, 1/x^2)$ (asymmetric, orange) at stable states.

capitalization at point $(x, f(x))$ in terms of that asset alone:

$$\text{numcap}_X(v; A) := \phi(v) + \left(\frac{1-v}{v}\right) f(\phi(v)).$$

For example, the numéraire capitalization of the constant-product AMM at the stable state for valuaton v is $2\sqrt{\frac{1}{v} - 1}$.

Unfortunately, this notion of capitalization can lead to counter-intuitive results if the numéraire asset becomes volatile. As the value of the numéraire X tends toward zero ($v \to 0$), arbitrageurs will replace the more valuable Y with the less valuable X. As the AMM fills up with increasingly worthless units of X, its numéraire capitalization grows without bound, so an AMM whose holdings have become worthless has infinite numéraire capitalization.

A more robust approach is to choose a formula that balances the asset classes in proportion to their relative valuations. Define the (balanced) *capitalization* at $(x, f(x))$ to be the sum of the two asset pools weighted by their relative values. Let $\boldsymbol{v} = (v, 1-v)$ and $\boldsymbol{x} = (x, f(x))$.

$$\text{cap}(x, v; A) := vx + (1-v)f(x) = \boldsymbol{v} \cdot \boldsymbol{x}. \tag{4.1}$$

If $\boldsymbol{v} = (v, 1-v)$ is the current market valuation, then A will usually be in the corresponding stable state $\Phi(v) = (\phi(v), f(\phi(v)))$, yielding

$$\text{cap}(v; A) := \text{cap}(\phi(v), v; A) = \boldsymbol{v} \cdot \Phi(v). \tag{4.2}$$

For example, AMMs $A := (x, 1/x)$ and $B := (x, 1/x^2)$ have capitalization at their stable states:

$$\text{cap}(v; (x, 1/x)) = 2\sqrt{v(1-v)},$$
$$\text{cap}(v; (x, 1/x^2)) = \frac{3v\sqrt[3]{\frac{1}{v} - 1}}{2^{2/3}}.$$

See Figure 5.5. Both have minimum capitalization 0 at $v = 0$ and $v = 1$, where one asset is worthless, and maximal capitalizations at respective valuations $v = 1/2$ and $v = 2/3$.

One delicate matter is the domain of our definition for the capitalization function. Namely, when $\text{cap}(v; A)$ was originally defined, it was implicit to the discussion that $v \in (0, 1)$. If we want to extend our definition of $\text{cap}(v; A)$ to

be defined on $[0,1]$, then how should $\text{cap}(v;A)$ be defined on the endpoints $\{0,1\}$? At minimum, to ensure at least continuity, we must must impose the following boundary conditions

- $\text{cap}(0;A) = \lim_{v\to 0} \text{cap}(v;A)$
- $\text{cap}(1;A) = \lim_{v\to 1} \text{cap}(v;A))$

That is, we want an extension of c so that it is defined on $[0,1]$ and the extension is continuous.

It's sufficient then to verify that these limits do exist and subsequently compute their values. For example

$$\begin{aligned}
&\lim_{v\to 0} \text{cap}(v;A) = v\phi(v) + (1-v)f(\phi(v)) \\
&= \lim_{v\to 0} v\phi(v) + \lim_{v\to 0}(1-v)f(\phi(v)) \\
&= \lim_{v\to 0} v\phi(v) + f(\phi(0)) \\
&= \lim_{v\to 0} v\phi(v)
\end{aligned}$$

Unfortunately it's not immediately clear what the value of this limit is for an arbitrary AMM. One idea is to try and simplify this limit just using tricks from standard calculus. Since $\lim_{v\to 0} \frac{\phi(v)}{\frac{1}{v}} = \frac{\infty}{\infty}$, we could instead compute the limit $\lim_{v\to 0} \frac{\phi'(v)}{\frac{-1}{v^2}}$. This doesn't really get us anywhere since we still get an indeterminate form of $\frac{\infty}{\infty}$ as we take higher order derivatives. Since the above approach doesn't work for computing this limit, we have to come up with a different method to get the desired result.

When the AMM A is understood, we will sometimes write $c(v)$ instead of $\text{cap}(v;A)$.

Lemma 4.1.1. *If $A := (x, f(x))$ is an AMM with differentiable capitalization function $c(v)$, then*

- $c'(v) = v\phi(v) + (1-v)f(\phi(v))$
- $\lim_{v\to 0} c'(v) = \infty$
- $\lim_{v\to 1} c'(v) = -\infty$

Proof. Computing the first-derivative for $v \in (0,1)$ we get

$$\begin{aligned}
&c'(v) = \phi(v) + v\phi'(v) + (1-v)f'(\phi(v))\phi'(v) - f(\phi(v)) \\
&\phi(v) + v\phi'(v) + (1-v)\frac{-v}{1-v}\phi'(v) - f(\phi(v)) \\
&= \phi(v) - f(\phi(v))
\end{aligned}$$

Thus we get

$$\begin{aligned}
&\lim_{v\to 0} c'(v) \\
&= \lim_{v\to 0} \phi(v) - \lim_{v\to 0} f(\phi(v)) \\
&= \infty - f(\lim_{v\to 0} \phi(v)) && \text{(By continuity of } f\text{)} \\
&= \infty - 0 = \infty
\end{aligned}$$

Additionally, we have

$$\begin{aligned}
&\lim_{v\to 1} c'(v) \\
&= \lim_{v\to 1} \phi(v) - \lim_{v\to 1} f(\phi(v)) \\
&= 0 - f(\lim_{v\to 1} \phi(v)) && \text{(By continuity of } f\text{)} \\
&= 0 - \infty = -\infty
\end{aligned}$$

□

Lemma 4.1.2. *If $A := (x, f(x))$ is an AMM with C^2 capitalization function $c(v)$, then c is strictly concave on $(0, 1)$.*

Proof. Recall from Lemma 4.1.1 that

$$c'(v) = \phi(v) - f(\phi(v))$$

Taking another derivative gives

$$\begin{aligned}
c''(v) &= \phi'(v) - f'(\phi(v))\phi'(v) \\
&= \phi'(v)(1 - \frac{-v}{1-v}) \\
&= \phi'(v)(\frac{1-v+v}{1-v}) = \frac{\phi'(v)}{1-v}
\end{aligned}$$

Now, taking a derivative wrt to v of both sides of

$$f'(\phi(v)) = \frac{-v}{1-v}$$

yields

$$\phi'(v) = \frac{-1}{(1-v)^2 f''(\phi(v))}$$

Recall that f is strictly convex, so $f''(\phi(v)) > 0$ for all $v \in (0, 1)$.

We can then write the second derivative of the capitalization as

$$c''(v) = \frac{-1}{(1-v)^3 f''(\phi(v))} < 0$$

Thus c is strictly concave. □

The following result shows existence and characterization for the value where $\text{cap}(A; v)$ is maximized.

Theorem 4.1.3. *Given an AMM $A := (x, f(x))$ with C^2 capitalization function $c(v)$:*

- *There is a unique point $v^* \in (0, 1)$ where $c(v)$ is maximized.*
- *$c'(v) > 0$ for $v \in (0, v^*)$, $c'(v) < 0$ for $v \in (v^*, 1)$, and $c'(v^*) = 0$ when $v = v^*$.*

- *The point v^* satisfies $\phi(v^*) = f(\phi(v^*))$.*

Proof. From Lemma 4.1.1, $\lim_{v\to 0} c'(v) = \infty$ and $\lim_{v\to 1} c'(v) = -\infty$. Thus we can choose v_1 close enough to 0 so that $c'(v_1) > 0$ and v_2 close enough to 1 so that $c'(v_2) < 0$. Because $c \in C^2(0,1)$, c' is continuous, by the intermediate value theorem we know there is some $v^* \in (0,1)$ where $c'(v^*) = 0$. Because c is strictly concave, via Lemma 4.1.2, this is a global maximum. Now suppose that $c'(\tilde{v}) \leq 0$ for some $\tilde{v} \in (0, v^*)$. Since c strictly concave we know $c''(v) < 0$ for all $v \in (0,1)$. This gives

$$c'(v^*) - c'(\tilde{v}) = \int_{\tilde{v}}^{v^*} c''(v)dv < 0$$

so

$$c'(\tilde{v}) > c'(v^*) = 0$$

giving us a contradiction. A symmetric argument shows that is must be the case that $c'(v) < 0$ for $v \in (v^*, 1)$.

Finally, by the first-order condition

$$c'(v^*) = 0$$

and from the computation in Lemma 4.1.1, we immediately get

$$\phi(v^*) = f(\phi(v^*))$$

□

Lemma 4.1.4. *For any 2D AMM A,*$\lim_{v\to 0} c(v), \lim_{v\to 1} c(v)$ *both exist.*

Proof. From Theorem 4.1.3, since $c'(v) < 0$ on $(v^*, 1)$, c is decreasing on this interval. We additionally know that $c(v) \geq 0$ for all $v \in (0,1)$. So c is a decreasing bounded function on $(v^*, 1)$ meaning $\lim_{v\to 0} c(v)$ exists. Existence of the other limit follows from a similar argument. □

Lemma 4.1.5. *For any 2D AMM A,* $\lim_{v\to 0} c(v) = 0$, $\lim_{v\to 1} c(v) = 0$.

Proof. We just show the first limit since the other proceeds in a similar way. Let $\alpha = \lim_{v\to 0} c(v)$ which exists by Lemma 4.1.4. We want to show $\alpha = 0$. We already know that $\alpha \geq 0$ because $c(v) \geq 0$ for all $v \in (0,1)$. Fix some $v \in (0, v^*)$ and fix some $\epsilon > 0$. By definition of the limit we know there is some δ_1 such that if $w < \delta_1$ then $|\alpha - c(w)| < \frac{\epsilon}{2}$. Since $v \in (0, v^*)$ we know from Theorem 4.1.3 that $c'(v) > 0$. Now choose $\delta = \min\{\delta_1, \frac{\epsilon}{2c'(v)}\}$. Define $g(w) = c'(v)(w - v) + c(v)$ for $w \in [0,1]$. That is, the tangent line to c at $(v, c(v))$. Since c is strictly concave we know $g(w) \geq c(w)$ for all $w \in (0,1)$. We can rewrite $g(w)$ as

$$\begin{aligned} g(w) &= c'(v)(w - v) + c(v) \\ &= wc'(v) - v\phi(v) + vf(\phi(v)) + v\phi(v) + f(\phi(v)) - vf(\phi(v)) \\ &= wc'(v) + f(\phi(v)) \end{aligned}$$

If we now choose $w < \delta$ we get

$$\begin{aligned}
|\alpha| &\leq |\alpha - c(w)| + c(w) \\
&< \frac{\epsilon}{2} + g(w) \\
&\leq \frac{\epsilon}{2} + wc'(v) + f(\phi(v)) \\
&\leq \frac{\epsilon}{2} + \frac{\epsilon}{2c'(v)}c'(v) + f(\phi(v)) \\
&= \epsilon + f(\phi(v))
\end{aligned}$$

Our choice of ϵ was arbitrary so $\alpha \leq f(\phi(v))$. But our choice of $v \in (0, v^*)$ was also arbitrary. So taking limits of both sides gives

$$\alpha \leq \lim_{v \to 0} f(\phi(v)) = f(\phi(0)) = 0$$

So $\alpha = 0$, which is what we wanted. □

The previous results give a clear description of capitalization functions in general. Namely, Theorem 4.1.3 guarantees existence/uniqueness of a maximizing point v^* for every $cap(v; A)$ function. Additionally, we have shown that to extend $cap(v; A)$ to the entire interval $[0, 1]$, we should define $cap(0; A) = cap(1; A) = 0$.

4.1.2 Divergence Loss

We now introduce the main measure associated with the cost for providers.

Consider the following simple game. A liquidity provider funds an AMM A, leaving it in the stable state $\boldsymbol{x}$ for the current market valuation $\boldsymbol{v}$. Suppose the market valuation changes from $\boldsymbol{v}$ to $\boldsymbol{v}'$, and a trader submits an arbitrage trade that would take A to the stable state $\boldsymbol{x}'$ for the new valuation. The provider has a choice: (1) immediately withdraw its liquidity from A instead of accepting the trade, or (2) accept the trade. It is not hard to see that the provider should always choose to withdraw. The shift to the new valuation changes the value of the AMM's capitalization function. If $\boldsymbol{x}$ is the stable state for $\boldsymbol{v}$, then it cannot be stable for $\boldsymbol{v}'$, so the difference between the capitalizations $\boldsymbol{v}' \cdot \boldsymbol{x} - \boldsymbol{v}' \cdot \boldsymbol{x}'$ must be positive, so the arbitrage trader would profit at the provider's expense. (In practice, a provider would take into account the value of current and future fees in making this decision.)

Define the *divergence loss* for an AMM $A = (x, f(x))$ to be

$$\begin{aligned}
\text{divloss}(v, v'; A) &:= \boldsymbol{v}' \cdot \Phi(v) - \boldsymbol{v}' \cdot \Phi(v') \\
&= v'\phi(v) + (1 - v')f(\phi(v)) - (v'\phi(v') + (1 - v')f(\phi(v')))
\end{aligned}$$

where $\Phi(v, 1 - v) = (\phi(v), f(\phi(v)))$. Sometimes it is useful to express divergence loss in the trade domain, as a function of liquidity pool size instead of valuation:

$$\begin{aligned}
\text{divloss}^*(x, x'; A) &:= \text{divloss}(\psi(x), \psi(x'); A) \\
&= \Psi(x') \cdot \boldsymbol{x} - \Psi(x') \cdot \boldsymbol{x}' \\
&= \psi(x')x + (1 - \psi(x'))f(x) - (\psi(x')x' + (1 - \psi(x'))f(x').
\end{aligned}$$

Informally, divergence loss measures the difference in value between funds invested in an AMM and funds left in a wallet. Recall that by definition $\boldsymbol{v}'$ minimizes $\Phi(v') \cdot \boldsymbol{v}'$, so divergence loss is always positive when $\boldsymbol{v} \neq \boldsymbol{v}'$.

Is it possible to bound divergence loss by bounding trade size? More precisely, can an AMM guarantee that any trade that adds δ or fewer units of X incurs a divergence loss less than some ϵ, for positive constants δ, ϵ?

Unfortunately, no. There is a strong sense in which divergence loss can be shifted, but never eliminated. For example, for the constant-product AMM $A := (x, 1/x)$, the divergence loss for a trade of size δ is

$$\text{divloss}^*(x, x+\delta; A) = \frac{\delta^2}{2\delta x^2 + x^3 + \delta^2 x + x} \tag{4.3}$$

Holding δ constant and letting x approach 0, the divergence loss for even constant-sized trades grows without bound. This property holds for all AMMs.

Theorem 4.1.6. *No AMM can bound divergence loss even for bounded-size trades.*

Proof. For AMM $A := (x, f(x))$,

$$\begin{aligned}&\text{divloss}^*(x, x+\delta; A)\\ &= \psi(x+\delta)x + (1-\psi(x+\delta))f(x) - (\psi(x+\delta)(x+\delta) + (1-\psi(x+\delta))f(x+\delta)).\end{aligned}$$

Note that $\lim_{x\to 0}(1-\psi(x+\delta)) > 0$, and $\lim_{x\to 0} f(x) = \infty$. All other terms have finite limits, so $\lim_{x\to 0} \text{divloss}^*(x, x+\delta; A) = \infty$. □

What is a provider's worst-case exposure to divergence loss? Consider an AMM $A := (x, f(x))$ in state $(a, f(a))$. As the X asset becomes increasingly worthless, the valuation v' approaches $(0, 1)$ as its stable state approaches $(\infty, 0)$. In that case

$$\lim_{v'\to 0} \text{divloss}(v, v'; A) = (v', 1-v') \cdot (a, f(a)) - (v', 1-v') \cdot (\phi(v'), f(\phi(v'))) = f(a).$$

Symmetrically, if the Y asset becomes worthless,

$$\lim_{v'\to 1} \text{divloss}(v, v'; A) = (v', 1-v') \cdot (a, f(a)) - (v', 1-v') \cdot (\phi(v'), f(\phi(v'))) = a.$$

Consider now how divergence loss is affected when a trader makes an arbitrary number of successive trades. Define a *X-partition* to be a sequence of values $\{x_i\}_{i=1}^{\infty}$ in $\mathbb{R}_{>0}$ such that $x_1 = a$, $x_i < x_{i+1}$, and $\lim_{n\to\infty} x_n = \infty$.

Given a partition $P_X = \{x_i\}_{i=1}^{n}$, define the *total divergence loss* with respect to that partition as

$$\text{divloss}(P_X; A) = \sum_{i=1}^{\infty} \text{divloss}^*(x_i, x_{i+1}; A)$$

Writing this out explicitly gives

$$\begin{aligned}
&\sum_{i=1}^{\infty} \text{divloss}^*(x_i, x_{i+1}; A) \\
&= \sum_{i=1}^{\infty} \boldsymbol{v}_{i+1} \cdot (\Phi(\boldsymbol{v}_i) - \Phi(\boldsymbol{v}_{i+1)}) \\
&= \sum_{i=1}^{\infty} \boldsymbol{v}_{i+1} \cdot ((\phi(v_i), f(\phi(v_i))) - (\phi(v_{i+1}), f(\phi(v_{i+1})))) \\
&= \sum_{i=1}^{\infty} (v_{i+1}, 1 - v_{i+1}) \cdot (\phi(v_i) - \phi(v_{i+1}), f(\phi(v_i)) - f(\phi(v_{i+1}))) \\
&= \sum_{i=1}^{\infty} v_{i+1}(\phi(v_i) - \phi(v_{i+1})) - v_{i+1}(f(\phi(v_i)) - f(\phi(v_{i+1}))) + \sum_{i=1}^{\infty} f(\phi(v_i)) - f(\phi(v_{i+1})) \\
&= \sum_{i=1}^{\infty} v_{i+1}(\phi(v_i) - \phi(v_{i+1})) - v_{i+1}(f(\phi(v_i)) - f(\phi(v_{i+1}))) + f(\phi(v_1)) \\
&= \sum_{i=1}^{\infty} v_{i+1}[\phi(v_i) - \phi(v_{i+1}) + f(\phi(v_{i+1})) - f(\phi(v_i))] + f(\phi(v_1))
\end{aligned}$$

Note that each term $\phi(v_i) - \phi(v_{i+1}) + f(\phi(v_{i+1})) - f(\phi(v_i)) \leq 0$. We also know that $v_1 \geq v_{i+1}$ for each i. This gives us the upper bound

$$\sum_{i=1}^{\infty} v_{i+1}[\phi(v_i) - \phi(v_{i+1}) + f(\phi(v_{i+1})) - f(\phi(v_i))] + f(\phi(v_1)) \leq f(\phi(v_1)) = f(x)$$

and the lower bound

$$\begin{aligned}
&\sum_{i=1}^{\infty} v_{i+1}[\phi(v_i) - \phi(v_{i+1}) + f(\phi(v_{i+1})) - f(\phi(v_i))] + f(\phi(v_1)) \\
&\geq v_1 \sum_{i=1}^{\infty} [\phi(v_i) - \phi(v_{i+1}) + f(\phi(v_{i+1})) - f(\phi(v_i))] + f(\phi(v_1)) \\
&= v_1[\phi(v_1) - f(\phi(v_1))] + f(\phi(v_1)) \\
&= v_1\phi(v_1) + (1 - v_1)f(\phi(v_1)) \\
&= \text{cap}(v; A)
\end{aligned}$$

Simply put

$$\text{cap}(v; A) \leq \text{divloss}(P_X; A) \leq f(a)$$

Let v^* be the point where $\text{cap}(v; A)$ is maximized. If we let $x^* = \phi(v^*)$, then we know that $f(x^*) = x^*$. We also know that $\text{cap}(v; A) = v^*x^* + (1 - v^*)f(x^*) = f(x^*)$. But if $x = x^*$ then this means

$$f(x^*) \leq \text{divloss}(P_X; A) \leq f(a)$$

which is entirely independent of the chosen partition P_X, so

$$f(x^*) \leq \text{divloss}(A) \leq f(a)$$

The previous equation provides an explicit picture of to what degree we can remove divergence loss. No matter how you choose to drain asset type Y by depositing asset type X, in the end, the provider can lose no more than asset $f(a)$, which we should expect. However, a provider could minimize its worst-case divergence loss by ensuring $f(a) = f(x^*)$, or namely $x^* = a$.

We get a similar result when trading along the Y-axis. Define a *Y-partition* to be a sequence of elements $\{y_i\}_{i=1}^{\infty}$ in $\mathbb{R}_{>0}$ such that $y_1 = f(a)$, $y_i \leq y_{i+1}$, and $\lim_{n\to\infty} y_n = \infty$.

Let $P_Y = \{y_i\}_{i=1}^{\infty}$ be a Y-partition, $g = f^{-1}(x)$. and $y_i = f(x_i)$ so $x_i = f^{-1}(y_i) = g(y_i)$. Thus $\Phi(\boldsymbol{v}_i) = (x_i, f(x_i)) = (g(y_i), f \circ g(y_i))$ where $f \circ g(y_i) = f \circ f^{-1}(y_i) = y_i$. That is, $\Phi(\boldsymbol{v}_i) = (g(y_i), y_i)$. The total cost with respect to this partition is

$$\text{divloss}(P_Y; A) = \sum_{i=1}^{\infty} \text{divloss}^*(y_i, y_{i+1}; A)$$

For symmetry, define $v_i' = 1 - v_i$.

$$\begin{aligned}
&\sum_{i=1}^{\infty} \text{divloss}^*(y_i, y_{i+1}; A) \\
&\qquad = \sum_{i=1}^{\infty} \boldsymbol{v}_{i+1} \cdot (\Phi(\boldsymbol{v}_i) - \Phi(\boldsymbol{v}_{i+1})) \\
&\qquad = \sum_{i=1}^{\infty} \boldsymbol{v}_{i+1} \cdot (g(y_i) - g(y_{i+1}), y_i - y_{i+1}) \\
&\qquad = \sum_{i=1}^{\infty} (1 - v_{i+1}')(g(y_i) - g(y_{i+1}) + v_{i+1}'(y_i - y_{i+1}) \\
&\qquad = \sum_{i=1}^{\infty} v_{i+1}'[y_i - y_{i+1} + g(y_{i+1}) - g(y_i)] + \sum_{i=1}^{\infty} (g(y_i) - g(y_{i+1}) \\
&\qquad = \sum_{i=1}^{\infty} v_{i+1}'[y_i - y_{i+1} + g(y_{i+1}) - g(y_i)] + a
\end{aligned}$$

Note that $g(y_i) - g(y_{i+1})) + y_{i+1} - y_i \leq 0$ and $v_1' \geq v_{i+1}'$ for each i. We now get the upper bound

$$\sum_{i=1}^{\infty} v_{i+1}'[y_i - y_{i+1} + g(y_{i+1}) - g(y_i)] + a \leq a$$

and the lower bound

$$\begin{aligned}
&\sum_{i=1}^{\infty} v_{i+1}'[y_i - y_{i+1} + g(y_{i+1}) - g(y_i)] + a \\
&\geq v_1' \sum_{i=1}^{\infty} y_i - y_{i+1} + g(y_{i+1}) - g(y_i) + a \\
&= v_1'(y_1 - g(y_1)) + a \\
&= (1 - v_1)(f(a) - a) + a \\
&= (1 - v_1) f(a) + v_1 a = \text{cap}(v; A)
\end{aligned}$$

So again we get the bounds

$$\text{cap}(v; A) \leq \text{divloss}(P_Y; A) \leq a$$

leading to

$$x^* \leq \text{divloss}(A) \leq a$$

Similar to the X-axis case this inequality is tight if $y_1 = f(x^*)$ and $\text{divloss}(P_Y; A) = a$. This immediately leads to a conservation result for divergence loss. Namely, if an AMM is in state $(x^*, f(x^*))$, then

$$\text{divloss}(P_X; A) + \text{divloss}(P_Y; A) = x^* + f(x^*) = 2x^* = 2\,\text{cap}(v^*; A)$$

where the partitions P_X, P_Y are arbitrary. This provides another simple interpretation for $\boldsymbol{v}^*$. The valuation $\boldsymbol{v}^*$ corresponds to the AMM state where half of the wealth may be lost to X trades and half can be lost to Y trades.

The provider's worst-case exposure to divergence loss in state $(a, f(a))$ is then $\max(a, f(a))$. The *minimum* worst-case exposure occurs when $a = f(a)$. Recall from Section 4.1.1 that this fixed-point is exactly the state that maximizes the AMM's capitalization. Note that this provides a practical strategy for providers: when initializing an AMM $(x, f(x))$, the initial state should always be that for which $x = f(x)$ in order to minimize worst-case divergence loss.

Divergence loss is sometimes called *impermanent loss*, because the loss vanishes if the assets return to their original valuation. The inevitability of impermanent loss does not mean that an AMM's capitalization cannot increase, only that there is always an opportunity cost to the provider for not cashing in earlier.

4.1.3 Linear Slippage

Linear slippage measures how increasing the size of a trade diminishes that trader's rate of return. Let $A := (x, f(x))$ be an AMM in stable state $(a, f(a))$ for valuation v. Suppose a trader sends $\delta > 0$ units of X to A, taking A from $(a, f(a))$ to $(a + \delta, f(a + \delta))$, the stable state for valuation $v' < v$. If the rate of exchange were linear, the trader would receive $-\delta f'(a)$ units of Y in return for δ units of X. In fact, the trader receives only $f(a) - f(a + \delta)$ units, for a difference of $-\delta f'(a) - f(a) + f(a + \delta)$.

The *linear slippage*(with respect to X) is the value of this difference:

$$\begin{aligned}
\text{linslip}_X(v, v'; A) &:= (1 - v')\left(-\delta f'(x) + f(x + \delta) - f(x)\right) \\
&= (1 - v')\left(\delta \frac{v}{1 - v} + f(x + \delta) - f(x)\right) \\
&= \left(\frac{1 - v'}{1 - v}\right)\left(v\phi(v') - v\phi(v) + (1 - v)f(\phi(v')) - (1 - v)f(\phi(v))\right) \\
&= \left(\frac{1 - v'}{1 - v}\right)\left(\boldsymbol{v} \cdot \Phi(v') - \boldsymbol{v} \cdot \Phi(v)\right)
\end{aligned}$$

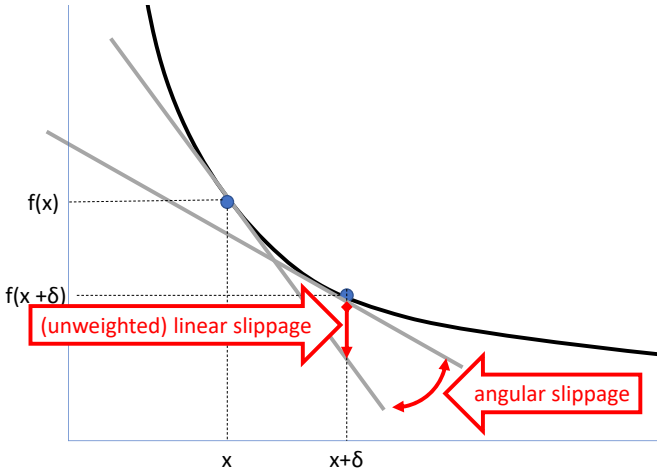

Figure 4.2: Angular vs linear slippage

or just

$$\text{linslip}_X(v, v'; A) = \left(\frac{1-v'}{1-v}\right)(\boldsymbol{v} \cdot \Phi(v') - \boldsymbol{v} \cdot \Phi(v)) \tag{4.4}$$

Note that the linear slippage can be related to the divergence loss via:

$$\text{linslip}_X(v, v'; A) = \left(\frac{1-v'}{1-v}\right)\text{divloss}(v', v; A).$$

In the trade domain, $\text{linslip}^*_X(x, x'; A) = \text{linslip}_X(\psi(x), \psi(x'); A)$. Linear slippage with respect to Y is defined symmetrically.

For example, the linear slippage for the constant-product AMM $A := (x, 1/x)$ for a trade of size δ is

$$\text{linslip}^*_X(x, x+\delta; A) = -\frac{\delta^2(\delta + x)}{x^2\left(\delta^2 + x^2 + 2\delta x + 1\right)} \tag{4.5}$$

Just as for divergence loss, holding δ constant and letting x approach 0, the linear slippage across constant-sized trades grows without bound.

Theorem 4.1.7. *No AMM can bound linear slippage for bounded-size trades.*

Proof. As in the proof of Theorem 4.1.6, the claim follows because $\lim_{x->0} \text{linslip}^*_X(x, x+\delta; A) = \infty$. □

4.1.4 Angular Slippage

Angular slippage measures how the size of a trade affects the exchange rate between the two assets. This measure focuses on how a trade affects the traders who come after. Recall that the (instantaneous) exchange rate in state $(x, f(x))$ is the slope of the tangent $f'(x)$. Let $\theta(x)$ denote the angle of that tangent with the X-axis. (We could equally well use the tangent's angle with the Y-axis.) A convenient way to measure the change in price is to measure the change in angle. Consider a trade that carries A from valuation v with stable state $(x, f(x))$, to valuation v' with stable state $(x', f(x'))$. Define the *angular slippage* of that trade to be the difference in tangent angles at x and x' (expressed in the valuation domain):

$$\text{angslip}(v, v'; A) = \theta(\phi(v')) - \theta(\phi(v)).$$

In the trade domain, $\text{angslip}^*(x, x'; A) := \text{angslip}(\phi(x), \phi(x'); A)$. Angular slippage is *additive*: for distinct valuations $v < v' < v''$, $\text{angslip}(v, v''; A) = \text{angslip}(v, v'; A) + \text{angslip}(v', v''; A)$. Note that linear slippage is not additive.

Angular slippage and linear slippage are different ways of measuring the same underlying phenomenon: their relation is illustrated in Figure 4.2.

Here is how to compute angular slippage. By definition,$\tan\theta(x) = -\frac{1}{f'(x)}$. Let $x' > x$, $\boldsymbol{v} = (v, 1-v)$, and $\boldsymbol{v}' = (v', 1-v')$. So we get

$$\begin{aligned}
&\text{angslip}(v, v'; A) \\
&= \theta(x') - \theta(x) \\
&= \arctan\left(\frac{1}{-f'(x')}\right) - \arctan\left(\frac{1}{-f'(x)}\right) \\
&= \arctan\left(\frac{1-v'}{v'}\right) - \arctan\left(\frac{1-v}{v}\right) \\
&= \arctan\left(\frac{(\frac{1-v'}{v'}) - (\frac{1-v}{v})}{1 + (\frac{1-v'}{v'})(\frac{1-v}{v})}\right) \\
&= \arctan\left(\frac{v - v'}{\boldsymbol{v} \cdot \boldsymbol{v}'}\right)
\end{aligned}$$

from which it follows that

$$\text{angslip}(v, v'; A) = \theta(x') - \theta(x) = \arctan\left(\frac{v - v'}{\boldsymbol{v} \cdot \boldsymbol{v}'}\right) \tag{4.6}$$

The next lemma says that the overall angular slippage, $\text{angslip}(0, \infty)$, is a constant independent of the AMM.

Theorem 4.1.8. *For every AMM A,* $\text{angslip}(0, \infty; A) = \pi/2$.

Proof. Consider an AMM $A := (x, f(x))$. As $\lim_{x\to 0} f'(x) = -\frac{v}{1-v} = -\infty$, implying $v = 1$. As $\lim_{x\to\infty} f'(x) = -\frac{v}{1-v} = 0$ implying $v = 0$. Because $\tan \text{angslip}(0, \infty; A) = \frac{1-0}{(1,0)\cdot(0,1)} = \infty$, $\text{angslip}(0, \infty; A) = \frac{\pi}{2}$. □

The additive property means that no AMM can eliminate angular slippage over every finite interval. Lowering angular slippage in one interval requires increasing it elsewhere.

Corollary 4.1.8.1. *For any AMM A, and any level of slippage s, $0 < s < \pi/2$, there is an interval $(x_0, x_1) \subset \mathbb{R}_{>0}$ such that* $\text{angslip}^*(x_0, x_1; A) > s$.

For example, the Curve [23] AMM advertises itself as having lower slippage than its competitors. Theorem 4.1.8 helps us understand this claim: compared to a constant-product AMM, Curve does have lower slippage than a constant-product AMM for stable coins when they trade at near-parity, but it must have higher slippage when the exchange rate wanders out of that interval.

4.1.5 Load

Divergence loss is a cost to providers, and linear slippage is a cost to traders. Controlling one without controlling the other is pointless because AMMs function only if both providers and traders consider their costs acceptable. The

following measure attempts to balance provider-facing and trader-facing costs. The *load* (with respect to X) across an interval is the product of that interval's divergence loss and linear slippage:

$$\text{load}_X(v, v'; A) := \text{divloss}(v, v'; A)\,\text{linslip}_X(v, v'; A) \tag{4.7}$$

Load can also be expressed in the trade domain:

$$\text{load}^*_X(x, x'; A) := \text{divloss}^*(x, x'; A)\,\text{linslip}^*_X(x, x'; A)$$

4.2 AMM Measures under Composition

4.2.1 Sequential Composition

Recall that the *sequential composition* of two AMMs is constructed by using the output of one AMM as the input to the other. For example, if A trades between florins and guilders, and B trades between guilders and francs, then their sequential composition $A \oplus B$ trades between florins and francs. A trader might deposit florins in A, receiving guilders, then deposit those guilders in B, receiving francs. In this section, we investigate how linear slippage and divergence loss interact with sequential composition.

Consider two AMMs $A = (x, f(x)), B = (y, g(y))$, where A trades between X and Y, and B between Y and Z. If A is in state $(a, f(a))$ and B in state $(b, g(b))$ then their sequential composition is $A \oplus B := (x, h(x))$, where $h(x) = g(b + f(a) - f(x))$ (Section 3.2.1).

Let $\boldsymbol{v} = (v_1, v_2, v_3)$ be the market valuation linking X, Y, Z, inducing pairwise valuations

$$v_{12} = \frac{v_1}{v_1 + v_2},\ v_{23} = \frac{v_2}{v_2 + v_3},\ v_{13} = \frac{v_1}{v_1 + v_3},$$

along with their vector forms $\boldsymbol{v}_{12} = (v_{12}, 1 - v_{12}),\ \boldsymbol{v}_{23} = (v_{23}, 1 - v_{23}),\ \boldsymbol{v}_{13} = (v_{13}, 1 - v_{13})$. Let $\boldsymbol{v}' \neq \boldsymbol{v}$ be a three-way valuation inducing analogous pair-wise valuations. Let $\phi_A, \phi_B, \phi_{AB} : (0, 1) \to \mathbb{R}_{>0}$ be the stable state maps for $A, B, A \oplus B$ respectively, and $\Phi_A, \Phi_B, \Phi_{AB} : (0, 1) \to \mathbb{R}^2_{>0}$ their vector forms.

Our composition rules apply when A and B start in their respective stable states[1] for a market valuation $\boldsymbol{v}$: $(a, f(a))$ is the stable state for v_{12}, $(a+\delta, f(a+\delta))$ for v'_{12}. $(b, g(b))$ for v_{23}, and $(b+f(a)-f(a+\delta), g(b+f(a)-f(a+\delta))$ for v'_{23}. We now analyze the changes in divergence loss and linear slippage when the market valuation changes from $\boldsymbol{v}$ to $\boldsymbol{v}'$.

[1] If A and B do not start in stable states for the current market valuation, then an arbitrage trader will eventually put them there.

Divergence Loss

Initially, the combined capitalization of A and B is $v_1 a + v_2(f(a)+b) + v_3 g(b)$. A trader sends δ units to A, changing the combined capitalization by

$$\begin{aligned}\delta v_1' + v_2'(f(a+\delta) - f(a)) &= v_1'(\phi_A(v_{12}') - \phi_A(v_{12})) + v_2'(f(\phi_A(v_{12}') - f(\phi_A(v_{12})))\\ &= (v_1' + v_2')(\boldsymbol{v}_{12}' \cdot \Phi_A(v_{12}') - \boldsymbol{v}_{12}' \cdot \Phi_A(v_{12}))\\ &= -(v_1' + v_2') \operatorname{divloss}(v_{12}, v_{12}'; A)\end{aligned}$$

thus

$$\delta v_1' + v_2'(f(a+\delta) - f(a)) = -(v_1' + v_2') \operatorname{divloss}(v_{12}, v_{12}'; A) \tag{4.8}$$

Next the trader sends the assets returned from the first trade to B, changing the combined capitalization by

$$\begin{aligned}&v_2'(f(a) - f(a+\delta)) + v_3'(g(b + f(a) - f(a+\delta)) - g(b))\\ &\qquad = v_2'(\phi_B(v_{23}') - \phi_B(v_{23})) + v_3'(g(\phi_B(v_{23}')) - g(\phi_B(v_{23}))\\ &\qquad = (v_2' + v_3')(\boldsymbol{v}_{23}' \cdot \Phi_B(v_{23}') - \boldsymbol{v}_{23}' \cdot \Phi_B(v_{23}))\\ &\qquad = -(v_2' + v_3') \operatorname{divloss}(v_{23}, v_{23}'; B)\end{aligned}$$

which yields

$$v_2'(f(a) - f(a+\delta)) + v_3'(g(b + f(a) - f(a+\delta)) - g(b)) = -(v_2' + v_3') \operatorname{divloss}(v_{23}, v_{23}'; B) \tag{4.9}$$

Finally, treating both trades as a single transaction changes the combined capitalization by:

$$\begin{aligned}\delta v_1' + v_3'(h(a+\delta) - h(a)) &= v_1'(\phi_{AB}(v_{13}') - \phi_{AB}(v_{13})) + v_3'(h(\phi_{AB}(v_{13}')) - h(\phi_{AB}(v_{13}))\\ &= (v_1' + v_3')((\boldsymbol{v}_{13}' \cdot \Phi_{AB}(v_{13}') - \boldsymbol{v}_{13}' \cdot \Phi_{AB}(v_{13}))\\ &= -(v_1' + v_3') \operatorname{divloss}(v_{13}, v_{13}'; A \oplus B)\end{aligned}$$

giving

$$\delta v_1' + v_3'(h(a+\delta) - h(a)) = -(v_1' + v_3') \operatorname{divloss}(v_{13}, v_{13}'; A \oplus B) \tag{4.10}$$

Combining Equations 4.8-4.10 yields

$$\operatorname{divloss}(v_{13}, v_{13}'; A \oplus B) = \left(\frac{1 - v_3'}{1 - v_2'}\right) \operatorname{divloss}(v_{12}, v_{12}'; A) + \left(\frac{1 - v_1'}{1 - v_2'}\right) \operatorname{divloss}(v_{23}, v_{23}'; B). \tag{4.11}$$

The effect of sequential composition on divergence loss is linear but not additive: the divergence loss of the composition is a weighted sum of the divergence losses of the components.

Linear Slippage

With respect to $\boldsymbol{v}$, a trader who sends δ units of X to A incurs the following slippage

$$\begin{aligned} v_2'(-\delta f'(a) + f(a+\delta) - f(a)) &= v_2' \left(\frac{v_1}{v_2}(\phi_A(v_{12}') - \phi_A(v_{12})) + (f(\phi_A(v_{12}')) - f(\phi_A(v_{12})) \right) \\ &= \frac{v_2'}{v_2}(v_1 + v_2)(\boldsymbol{v}_{12} \cdot \Phi_A(v_{12}') - \boldsymbol{v}_{12} \cdot \Phi_A(v_{12})) \\ &= (v_1' + v_2') \operatorname{linslip}_X(v_{12}, v_{12}'; A). \end{aligned}$$

so

$$v_2'(-\delta f'(a) + f(a+\delta) - f(a)) = (v_1' + v_2') \operatorname{linslip}_X(v_{12}, v_{12}'; A). \tag{4.12}$$

Next the trader sends the assets returned from the first trade to B, incurring the following slippage

$$\begin{aligned} v_3' \left(-(f(a) - f(a+\delta))g'(b) + g(b + f(a) - f(a+\delta)) - g(b)\right) &= v_3' \left((f(a) - f(a+\delta))\frac{v_2}{v_3} + g(b + f(a) - f(a+\delta)) - g(b) \right) \\ &= \frac{v_3'}{v_3}(v_2 + v_3) \left(\boldsymbol{v}_{23} \cdot \Phi_B(v_{23}') - \boldsymbol{v}_{23} \cdot \Phi_B(v_{23})\right) \\ &= (v_2' + v_3') \operatorname{linslip}_X(v_{23}, v_{23}'; B). \end{aligned}$$

giving

$$v_3' \left(-(f(a) - f(a+\delta))g'(b) + g(b + f(a) - f(a+\delta)) - g(b)\right) = (v_2' + v_3') \operatorname{linslip}_X(v_{23}, v_{23}'; B). \tag{4.13}$$

Finally, treating both trades as a single transaction yields slippage:

$$\begin{aligned} v_3'(-\delta h'(a) + h(a+\delta) - h(a)) &= v_3' \left(\frac{v_1}{v_3}(\phi_{AB}(v_{13}') - \phi_{AB}(v_{13})) + (h(\phi_{AB}(v_{13}')) - h(\phi_{AB}(v_{13})) \right) \\ &= \frac{v_3'}{v_3}(v_1 + v_3)(\boldsymbol{v}_{13} \cdot \Phi_{AB}(v_{13}') - \boldsymbol{v}_{13} \cdot \Phi_{AB}(v_{13})) \\ &= (v_1' + v_3') \operatorname{linslip}_X(v_{13}, v_{13}', A \oplus B) \end{aligned}$$

giving as a result

$$v_3'(-\delta h'(x) + h(x+\delta) - h(x))(v_1' + v_3') \operatorname{linslip}_X(v_{13}, v_{13}', A \oplus B) \tag{4.14}$$

Combining Equations 4.12-4.14 yields

$$\operatorname{linslip}_X(v_{13}, v_{13}'; A \oplus B) = \left(\frac{1 - v_3'}{1 - v_2'} \right) \operatorname{linslip}_X(v_{12}, v_{12}'; A) + \left(\frac{1 - v_1'}{1 - v_2'} \right) \operatorname{linslip}_X(v_{23}, v_{23}'; B) \tag{4.15}$$

Angular Slippage

A trader sends δ to A, where $f(a+\delta)$ is the stable state for v_{12}', $g(b + f(a) - f(a+\delta))$ the stable state for v_{23}'. By construction,

$$\begin{aligned} h'(a) = -g'(b)f'(a) = -\frac{v_{23}}{1 - v_{23}} \frac{v_{12}}{1 - v_{12}} = -\frac{v_2}{v_3}\frac{v_1}{v_2} = -\frac{v_1}{v_3} \\ h'(a+\delta) = -g'(b + f(a) - f(a+\delta))f'(a+\delta) = -\frac{v_{23}'}{1 - v_{23}'} \frac{v_{12}'}{1 - v_{12}'} = -\frac{v_2'}{v_3'}\frac{v_1'}{v_2'} = -\frac{v_1'}{v_3'} \end{aligned}$$

Define $\theta_A(x), \theta_B(y), \theta_B(y)$ to be the respective angles of $f'(x)$, $g'(y)$, and $h'(x)$ with their X-axes. We can express the tangents of the composite AMM's angles in terms of the tangents of the component AMMs' angles.

$$\tan\theta_{AB}(x) = -\frac{1}{h'(x)} = -\frac{1}{-g'(b+f(a)-f(x))f'(x)}$$
$$\tan\theta_{AB}(a) = \frac{v_3}{v_2}\frac{v_2}{v_1} = \frac{v_3}{v_1}, \qquad \tan\theta_{AB}(a+\delta) = \frac{v'_3}{v'_2}\frac{v'_2}{v'_1} = \frac{v'_3}{v'_1}.$$

The component AMMs A and B determine the valuations v_1, v_2, v'_1, v'_2, which induce the remaining valuations $v_3, v'_3, v_{12}, v_{23}, v_{13}, v'_{12}, v'_{23}, v'_{13}$. Since

$$\begin{aligned}\text{angslip}(v_{13}, v_{13'}, A \oplus B) &= \arctan\left(\frac{v'_3}{v'_1}\right) - \arctan\left(\frac{v_3}{v_1}\right) \\ &= \arctan\left(\frac{v_1 v'_3 - v'_1 v_3}{v_1 v'_1 + v_3 v'_3}\right) \\ &= \arctan\left(\frac{v_{13} - v'_{13}}{\boldsymbol{v}_{13} \cdot \boldsymbol{v}_{13}}\right)\end{aligned}$$

it is then immediate that

$$\text{angslip}(v_{13}, v_{13'}, A \oplus B) = \arctan\left(\frac{v_{13} - v'_{13}}{\boldsymbol{v}_{13} \cdot \boldsymbol{v}'_{13}}\right) \tag{4.16}$$

It follows that the angular slippage of the sequential composition of two AMMs can be computed from the component AMMs' valuations.

Load

Combining Equation 4.11 and Equation 4.15 yields

$$\begin{aligned}\text{load}_X(v_{13}, v'_{13}; A \oplus B) = &\left(\frac{1-v'_3}{1-v'_2}\right)^2 \text{load}_X(v_{12}, v'_{12}; A) + \left(\frac{1-v'_1}{1-v'_2}\right)^2 \text{load}_X(v_{23}, v'_{23}; B) \\ &+ \left(\frac{(1-v'_3)(1-v'_1)}{(1-v'_2)^2}\right) \text{divloss}(v_{12}, v'_{12}; A)\, \text{linslip}_X(v_{23}, v'_{23}; B) \\ &+ \left(\frac{(1-v'_3)(1-v'_1)}{(1-v'_2)^2}\right) \text{divloss}(v_{23}, v'_{23}; B)\, \text{linslip}_X(v_{12}, v'_{12}; A)\end{aligned} \tag{4.17}$$

It follows that the load of a sequential composition is a weighted sum of the loads of the components, plus additional (strictly positive) cross-terms.

4.2.2 Parallel Composition

Recall that parallel composition (Section 3.3) arises when a trader is presented with two AMMs $A := (x, f(x))$ and $B := (y, g(y))$, both trading assets X and Y, and seeks to treat them as a single combined AMM $A||B$. Let $\phi_A, \phi_B : (0,1) \to \mathbb{R}_{>0}$ be the stable state maps for A, B respectively, with $\Phi_A, \phi_B : (0,1) \to \mathbb{R}^2_{>0}$ their vector forms, and $\psi_A, \psi_B : \mathbb{R}_{>0} \to (0,1)$ their inverses.

As shown in Section 3.3 a trader who sends δ units of X to the combined AMM maximizes return by splitting those units between A and B, sending $t\delta$ to A and $(1-t)\delta$ to B, for $0 \le t \le 1$, where $f'(x + t\delta) = g'(y + (1-t)\delta)$.

We assume traders are rational, and always split trades in this way. Because the derivatives are equal, $x + t\delta$ and $y + (1-t)\delta$ are stable states of A and B respectively for the same valuation v', so $x + t\delta = \phi_A(v')$, and $y + (1-t)\delta =$

$\phi_B(v')$. If A is in state $(a, f(a))$ and B in $(b, g(b))$, then $A||B := (x, h(x))$, where $h(x) := (f(a+tx) + g(b+(1-t)x)$. Our composition rules apply when both A, B are in their stable states for valuation $\boldsymbol{v} = (v, 1-v)$. We analyze the change in divergence loss and linear slippage when the common valuation changes from $\boldsymbol{v}$ to $\boldsymbol{v'} = (v', 1-v')$. The new valuation $\boldsymbol{v'}$ may be the new market valuation, or it may be the best the trader can reach with a fixed budget of δ.

Divergence Loss

If a trader sends δ units of X to $A||B$, the combined capitalization changes by

$$\begin{aligned}
&v'\delta + (1-v')(f(a+t\delta) - f(a) + g(b+(1-t)\delta) - g(b)) \\
&= v'(a + t\delta - a + b + (1-t)\delta - b) \\
&\qquad + (1-v')(f(\phi_A(v')) - f(\phi_A(v)) + g(\phi_B(v')) - g(\phi_B(v))) \\
&= v'(\phi_A(v') - \phi_A(v) + \phi_B(v') - \phi_B(v)) \\
&\qquad + (1-v')(f(\phi_A(v')) - f(\phi_A(v)) + g(\phi_B(v')) - g(\phi_B(v))) \\
&= \boldsymbol{v'} \cdot \phi_A(v') - \boldsymbol{v'} \cdot \phi_A(v) + \boldsymbol{v'} \cdot \phi_B(v') - \boldsymbol{v'} \cdot \phi_B(v) \\
&= -\operatorname{divloss}(v, v'; A) - \operatorname{divloss}(v, v'; B)
\end{aligned}$$

so

$$\operatorname{divloss}(v, v'; A||B) = \operatorname{divloss}(v, v'; A) + \operatorname{divloss}(v, v'; B). \tag{4.18}$$

It follows that divergence loss under parallel composition is additive.

Linear Slippage

Note first that

$$\begin{aligned}
h'(x) &= tf'(a+tx) + (1-t)g'(b+(1-t)x) \\
&= t\frac{-v}{(1-v)} + (1-t)\frac{-v}{(1-v)} \\
&= \frac{-v}{(1-v)} \\
\delta &= t\delta + (1-t)\delta \\
\delta &= \phi_A(v') - \phi_A(v) + \phi_B(v') - \phi_B(v)
\end{aligned}$$

When a trader sends δ units of X to $A||B$, the resulting slippage is given by

$$\begin{aligned}
&(1-v')(-\delta h'(x)+h(x+\delta)-h(x)) \\
&=(1-v')\left(\delta\frac{v}{1-v}+h(x+\delta)-h(x)\right) \\
&=\left(\frac{1-v'}{1-v}\right)(\delta v+(1-v)(h(x+\delta)-h(x))) \\
&=\left(\frac{1-v'}{1-v}\right)(\delta v+(1-v)(f(a+t(x+\delta))+g(b+(1-t)(x+\delta)) \\
&\qquad -(f(a+tx)+g(b+(1-t)x)))) \\
&=\left(\frac{1-v'}{1-v}\right)(\delta v+(1-v)(f(a+t(x+\delta))-f(a+tx) \\
&\qquad +g(b+(1-t)(x+\delta))-g(b+(1-t)x))) \\
&=\left(\frac{1-v'}{1-v}\right)(\delta v+(1-v)(f(\phi_A(v'))-f(\phi_A(v))+g(\phi_B(v'))-g(\phi_B(v)))) \\
&=\left(\frac{1-v'}{1-v}\right)(v(\phi_A(v')-\phi_A(v)+\phi_B(v')-\phi_B(v)) \\
&\qquad +(1-v)(f(\phi_A(v'))-f(\phi_A(v))+g(\phi_B(v'))-g(\phi_B(v)))) \\
&=\left(\frac{1-v'}{1-v}\right)(v(\phi_A(v')-\phi_A(v))+(1-v)(f(\phi_A(v'))-f(\phi_A(v))) \\
&\qquad +v(\phi_B(v')-\phi_B(v))+(1-v)(g(\phi_B(v'))-g(\phi_B(v)))) \\
&=\left(\frac{1-v'}{1-v}\right)(\boldsymbol{v}\cdot\Phi_A(v')-\boldsymbol{v}\cdot\Phi_A(v))+(\boldsymbol{v}\cdot\Phi_B(v')-\boldsymbol{v}\cdot\Phi_B(v)) \\
&=\mathrm{linslip}_X(v,v';A)+\mathrm{linslip}_X(v,v';B).
\end{aligned}$$

showing

$$\mathrm{linslip}_X(v,v';A||B)=\mathrm{linslip}_X(v,v';A)+\mathrm{linslip}_X(v,v';B) \tag{4.19}$$

Linear slippage is thus additive under parallel composition.

Linear slippage is also linear under scalar multiplication. Any AMM $A:=(x,f(x))$ can be *scaled* by a constant $\alpha>0$ yielding a distinct AMM $\alpha A:=(\alpha x,\alpha f(x))$. Let $(x,f(x))$ be the stable state for valuation $\boldsymbol{v}$, and $(x',f'(x'))$ the stable state for $\boldsymbol{v}'$.

$$\mathrm{linslip}_X(v,v';\alpha A)=\alpha\,\mathrm{linslip}_X(v,v';A). \tag{4.20}$$

which follows from

$$\begin{aligned}
\mathrm{linslip}_X(v,v';\alpha A)&=\boldsymbol{v}'\cdot(\alpha x,\alpha f(x))-\boldsymbol{v}'\cdot(\alpha x',\alpha f(x')) \\
&=\alpha(\boldsymbol{v}'\cdot\Phi(v)-\boldsymbol{v}'\cdot\Phi(v') \\
&=\alpha\,\mathrm{linslip}_X(v,v';A).
\end{aligned}$$

Angular Slippage

Because both A and B go from stable states for v to stable states for v',

$$f'(a) = g'(a) = h'(a) = \frac{-v}{1-v}, \quad f'(a+t\delta) = g'(a+(1-t)\delta) = h'(a+\delta) = \frac{-v'}{1-v'}$$

It follows that

$$\text{angslip}(v, v'; A||B) = \text{angslip}(v, v'; A) = \text{angslip}(v, v'; B). \tag{4.21}$$

Load

Combining Equation 4.18 and Equation 4.19 yields

$$\begin{aligned} &\text{load}_X(v, v'; A||B) \\ &= \text{load}_X(v, v'; A) + \text{load}_X(v, v'; B) \\ &+ \text{divloss}(v, v'; A)\,\text{linslip}_X(v, v'; B) + \text{divloss}(v, v'; B)\,\text{linslip}_X(v, v'; A) \end{aligned} \tag{4.22}$$

It follows that the parallel composition's load is the sum of the components' loads, plus additional (strictly positive) cross-terms.

Chapter 5

Characteristics of 2D AMMs

In Section 5.1, we show that the notion of capitalization introduced in Section 4.1.1 has an important *dual* relationship with the underlying AMM. This correspondence can be used to freely move back and forth between an AMM and its associated capitalization function when convenient.

During Section 5.2, we classify many of these designs and prove general properties of these classes. The classes discussed capture AMMs used in practice.

5.1 The AMM-Capital Correspondence Theorem

Suppose we start with some capitalization function $\text{cap}(v; A)$ for some 2D AMM A. We can define

$$p(v) = \frac{\text{cap}(v; A)}{||\text{cap}(v; A)||_1}$$

Clearly then in this case $p(v) \geq 0$ and

$$\begin{aligned}
&\int_0^1 p(v)dv \\
&= \frac{1}{||\text{cap}(v; A)||_1} \int_0^1 \text{cap}(v; A)dv \\
&= \frac{1}{||\text{cap}(A)||_1} * ||\text{cap}(A)||_1 = 1
\end{aligned}$$

That is, a given capitalization function there is a natural association to a probability distribution on $[0, 1]$.

Alternatively, if we are first given some strictly concave distribution $p(v)$ on $[0, 1]$, how do we find the corresponding capitalization function? The easiest way is to just to choose $p(v)$ itself. Note then that $v^* \in [0, 1]$ in this case is just the point where $p(v)$ is maximized on $[0, 1]$.

This suggests that maybe there is some way of understanding 2D AMMs by instead understanding strictly concave distributions on $[0, 1]$. We will now show that this correspondence can indeed be made.

Let A_2 be the space of all 2D AMMs. That is $A \in A_2$ means $A := (x, f(x))$ where f satisfies the AMM axioms introduced in Section 2. Additionally, let $P(\Delta^1)$ be the space of all strictly concave probability distributions on $\Delta^1 = [0, 1]$ that are twice-differentiable. Formally, say $p \in P(\Delta^1)$ if:

- $p \in C^2[0,1]$
- p strictly concave
- $p'(0) = \infty$, $p'(1) = -\infty$, and $||p||_1 = 1$

Now define the map $H : A_2 \to P(\Delta^1)$ as

$$H(A) = H(v; A) = \frac{\operatorname{cap}(v; A)}{||\operatorname{cap}(v; A)||_1}$$

Two AMMs that lead to the same distribution are structurally the same since they should just be rescalings of each others capitalization function. Formally we can define the following relation $\sim$ on A_2. That is $A \sim B$ iff $\operatorname{cap}(v; A) = \gamma \operatorname{cap}(v; B)$ for some $\gamma > 0$ and for all $v \in [0,1]$. We need to verify this is an equivalence relation.

Lemma 5.1.1. *The relation $\sim$ on A_2 is an equivalence relation.*

Proof. Choosing $\gamma = 1$ verifies $A \sim A$. If $A \sim B$ then choosing $\frac{1}{\gamma}$ gives $B \sim A$. Finally, if $A \sim B$, $B \sim C$ for γ_1, γ_2, then $\gamma = \gamma_1\gamma_2$ verifies $A \sim C$. So $\sim$ is actually an equivalence relation. □

Now define $\tilde{A_2} = A_2/\sim$, namely the quotient space of A_2 under $\sim$. Let us write $[A] \in \tilde{A_2}$ where A is a representative element of A_2. Define the map $\tilde{H} : \tilde{A_2} \to P(\Delta^1)$ for $[A] \in \tilde{A_2}$ as

$$\tilde{H}([A]) = H(A) = H(v; A)$$

$\tilde{H}$ is indeed the map that will give us the correspondence desired. The following results now show that $\tilde{H}$ is a bijection.

First we verify $\tilde{H}$ is well-defined.

Lemma 5.1.2. *The map $\tilde{H} : \tilde{A_2} \to P(\Delta^1)$ is well-defined.*

Proof. Suppose $[A] = [B]$. So $\operatorname{cap}(v; A) = \gamma \operatorname{cap}(v; B)$ for some $\gamma > 0$. Then we have

$$\begin{aligned}
\tilde{H}([A]) = H(A) &= \frac{\operatorname{cap}(v; A)}{||\operatorname{cap}(v; A)||_1} \\
&= \frac{\gamma \operatorname{cap}(v; B)}{||\gamma \operatorname{cap}(v; B)||_1} \\
&= \frac{\gamma \operatorname{cap}(v; B)}{\gamma||\operatorname{cap}(v; B)||_1} \\
&= \frac{\operatorname{cap}(v; B)}{||\operatorname{cap}(v; B)||_1} \\
&= H(B) = \tilde{H}([B])
\end{aligned}$$

□

Additionally, we now show that $\tilde{H}$ is injective.

Lemma 5.1.3. *The map $\tilde{H} : \tilde{A_2} \to P(\Delta^1)$ is injective.*

Proof. Suppose $\tilde{H}([A]) = \tilde{H}([B])$. Then

$$\frac{\text{cap}(v;A)}{||\,\text{cap}(v;A)||_1} = \frac{\text{cap}(v;B)}{||\,\text{cap}(v;B)||_1}$$

or equivalently

$$\text{cap}(v;A) = \frac{||\,\text{cap}(v;A)||_1}{||\,\text{cap}(v;B)||_1}\,\text{cap}(v;B)$$

Since $||cap(v;A)||_1 > 0$ for any AMM A, setting $\gamma = \frac{||\,\text{cap}(v;A)||_1}{||\,\text{cap}(v;B)||_1}$ tells us $A \sim B$ or $[A] = [B]$, so $\tilde{H}$ is injective. □

The previous Lemma 5.1.3 ensures that $im\tilde{H} \subset P(\Delta^1)$. Namely, modulo scaling, the space of all 2D AMMs is effectively contained within the space of strictly concave probability distributions on $[0,1]$.

It remains then to verify if $\tilde{H}$ is surjective. This is the next question we answer.

Theorem 5.1.4. *Given a capitalization function $c \in C^2[0,1]$, there exists an AMM with that capitalization function.*

Proof. Suppose we are given some $c \in P(\Delta)^1$. Define $\phi(v) = c(v) + (1-v)c'(v)$. Note that

$$\phi(0) = c(0) + c'(0) = 0 + \infty = \infty$$
$$\phi(1) = c(1) + 0 = 0$$

We get

$$\phi'(v) = c'(v) - c'(v) + (1-v)c''(v)$$
$$= (1-v)c''(v) < 0$$

since c strictly concave. So $\phi(v)$ is strictly decreasing. Thus we know it has a differentiable inverse. If we write $x = \phi(v)$ then we can write $v = \psi(x)$ where $\psi = \phi^{-1}$. Since $\phi(0) = \infty, \phi(1) = 0$, by continuity we know $\psi(\infty) = 0, \psi(0) = 1$. Now define f as $f(x) = c(\psi(x)) - \psi(x)c'(\psi(x))$. Now

$$\lim_{x\to 0} f(x) = c(\psi(0)) - \psi(0)c'(\psi(0))$$
$$= c(1) - c'(1) = 0 - (-\infty) = \infty$$
$$\lim_{x\to\infty} f(x) = c(\psi(\infty)) - \psi(\infty)c'(\psi(\infty))$$
$$= c(0) = 0$$

As a function of v, $f(\phi(v)) = c(v) - vc'(v)$. Taking derivatives gives

$$f'(\phi(v))\phi'(v) = c'(v) - vc''(v) - c'(v) = -vc''(v)$$
$$\implies f'(\phi(v)) = \frac{-vc''(v)}{\phi'(v)} < 0$$

since c strictly concave, $\phi'(v) < 0$. If we write f as a function of v, $f(\phi(v)) = c(v) - vc'(v)$, then we get

$$\begin{aligned}
&v\phi(v) + (1-v)f(\phi(v)) \\
&v[c(v) + (1-v)c'(v)] + (1-v)[c(v) - vc'(v)] \\
&= vc(v) + vc'(v) - v^2c'(v) + c(v) - vc(v) - vc'(v) + v^2c'(v) \\
&= c(v)
\end{aligned}$$

Taking derivatives twice gives

$$\begin{aligned}
c''(v) &= \phi'(v) - f'(\phi(v))\phi'(v) \\
&\phi'(v)(1 - f'(\phi(v)))
\end{aligned}$$

Combining this with an earlier result gives

$$\begin{aligned}
f'(\phi(v)) &= \frac{-vc''(v)}{\phi'(v)} \\
&= \frac{-v\phi'(v)(1 - f'(\phi(v)))}{\phi'(v)} \\
&= -v(1 - f'(\phi(v))) \\
&\implies f'(\phi(v)) = \frac{-v}{1-v}
\end{aligned}$$

Taking one more derivative we get

$$\begin{aligned}
f''(\phi(v))\phi'(v) &= \frac{-1}{(1-v)^2} \\
\implies f''(\phi(v)) &= \frac{-1}{\phi'(v)(1-v)^2} > 0
\end{aligned}$$

so f is strictly convex. Thus f is a 2D AMM with capitalization function $c(v)$. □

This leads to the main result of the section:

Theorem 5.1.5. *The set of all 2D normalized AMMs is in a bijection with the set $P(\Delta)^1$.*

Proof. Lemma 5.1.3 told us that $\tilde{H}$ is injective. Similarly, the previous lemma (Theorem 5.1.4) verified that $\tilde{H}$ was surjective. So $\tilde{H}$ is a bijection between $\tilde{A}_2$ and $P(\Delta^1)$. □

The value of result can be interpreted in multiple ways. For one, it effectively reduces the study of 2D AMMs to something that is perhaps more familiar, a restricted family of probability distributions over an interval.

Additionally, it has the following practical benefit: An AMM designer could design a (normalized) capitalization function c according to their needs, say, based on how they anticipate the market will behave in the future. Theorem 5.1.5 guarantees that they can always find a corresponding AMM $A = \tilde{H}^{-1}(c)$ with that capitalization function!

5.2 Special AMM Families

We now provide an overview of many interesting families of AMMs and illustrate some of their unique properties. Theorem 5.1 will prove to be a valuable tool for switching between an AMM and its corresponding capitalization function throughout the following sections.

5.2.1 Beta Space

One particularly common probability distribution defined on $[0,1]$ is the *beta distribution* [56]. The beta distribution for shape parameters $\alpha, \beta > 0$ is given by the probability density function

$$f(x;\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$$

where

$$B(\alpha,\beta) = \int_0^1 x^{\alpha-1}(1-x)^{\beta-1}dx$$

We will write $Beta(\alpha,\beta)$ to refer to the distribution $f(x;\alpha,\beta)$. Note that the parameters α,β would give an interesting parametrization for capitalization functions if we could ensure that the corresponding $f(x;\alpha,\beta)$ was strictly concave. This is in fact not the case for all $\alpha,\beta > 0$.

We now make a brief digression to understand exactly for which α,β parameters this is indeed the case. Namely, we want to understand for which (α,β) we have $Beta(\alpha,\beta) \in P(\Delta^1)$. Towards this aim, let $Beta(\Delta^1) = \{Beta(\alpha,\beta) : (\alpha,\beta) \in \mathbb{R}^2_{>0}, Beta(\alpha,\beta) \in P(\Delta^1)\}$, which we will simply call *beta space*. It's convenient to denote subsets of $Beta(\Delta^1)$ by $Beta(\Delta^1;S)$ where $S \subseteq \mathbb{R}^2_{>0}$ denotes the subset of allowable parameters (α,β). For example when $S = \mathbb{R}^2_{>0}$ then $Beta(\Delta^1;\mathbb{R}_{>0}\times\mathbb{R}_{>0}) = Beta(\Delta^1)$. In this way we can interpret $Beta(\Delta^1;\cdot)$ as a map from sets of parameters into the space of all Beta distributions in $Beta(\Delta^1)$.

Now, any $f(x) = f(x;\alpha,\beta) \in Beta(\Delta^1)$ must satisfy

- $f(0) = f(1) = 0$
- $f'(0) = \infty, f'(1) = -\infty$
- $f''(x) < 0$ for $x \in (0,1)$

Lemma 5.2.1. *The beta space $Beta(\Delta^1)$ is given by $Beta(\Delta^1,(1,2)\times(1,2))$.*

Proof. Recall f is of the form $f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$. To ensure the first condition $f(0) = 0^{\alpha-1} = 0$ we need $\alpha > 1$. Similarly for $f(1) = 0$ we need $\beta > 1$.

Let $g(x) = lnf(x)$ so $g' = \frac{f'}{f}$. Since f is positive for $x \in (0,1)$ we can find the critical point of f' by finding it for g'. We have $g'(x) = \frac{\alpha-1}{x} + \frac{\beta-1}{1-x} = 0$ when $x = \frac{\alpha-1}{\alpha+\beta-2}$.

Note now that

$$f'(x) = f(x)g'(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}(\frac{1-\alpha}{x} + \frac{\beta-1}{1-x})$$
$$\propto x^{\alpha-2}(1-x)^{\beta-1} + x^{\alpha-1}(1-x)^{\beta-2}$$

Now we check when $\lim_{x\to 0} f'(x) = \infty$. Clearly at least $\alpha - 2, \beta - 1 < 0$ or $\alpha - 1, \beta - 2 < 0$. Since we require $\beta, \alpha > 1$, it must be the case that either $\alpha, \beta < 2$. Say $\beta < 2$ and $\alpha \geq 2$. Then $\lim_{x\to 0} f'(x) = 0$. So if $\beta < 2$ we must also have $\alpha < 2$. Similarly if $\alpha < 2$ and $\beta \geq 2$, then $\lim_{x\to 1} f'(x) = 1$. So it's necessary that $(\alpha, \beta) \in (1, 2) \times (1, 2)$.

Finally we need to verify that for $(\alpha, \beta) \in (1, 2) \times (1, 2)$, $f''(x; \alpha, \beta) < 0$ for all $x \in (0, 1)$. Note that

$$g'' = \frac{ff'' - (f')^2}{f^2}$$

which we can rewrite as

$$f'' = \frac{g''f^2 + (f')^2}{f}$$

Since $f > 0$ it is then enough to verify that $g'' < -(\frac{f'}{f})^2 = -(g')^2$. Recall that

$$g'(x) = \frac{\alpha - 1}{x} + \frac{1 - \beta}{1 - x}$$

and

$$g''(x) = \frac{1 - \alpha}{x^2} + \frac{1 - \beta}{(1 - x)^2}$$

which gives

$$\begin{aligned} & g''(x) + (g'(x))^2 \\ &= \frac{1 - \alpha}{x^2} + \frac{1 - \beta}{(1 - x)^2} + \frac{(\alpha - 1)^2}{x^2} + \frac{2(\alpha - 1)(1 - \beta)}{x(1 - x)} + \frac{(1 - \beta)^2}{(1 - x)^2} \\ &= \frac{(\alpha - 1)(\alpha - 2)}{x^2} + \frac{(\beta - 1)(\beta - 2)}{(1 - x)^2} + \frac{2(\alpha - 1)(1 - \beta)}{x(1 - x)} \end{aligned}$$

Each of the above terms is negative for $(\alpha, \beta) \in (1, 2) \times (1, 2)$ so we're done. Thus $Beta(\Delta^1) = Beta(\Delta^1; (1, 2) \times (1, 2)) \subseteq P(\Delta^1)$. □

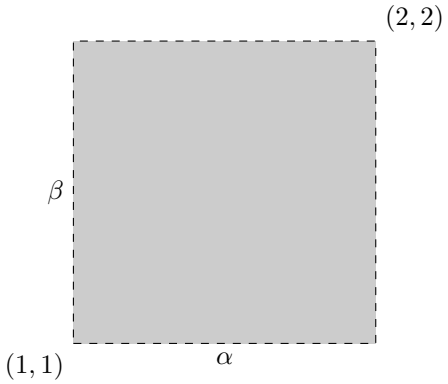

Figure 5.1: $(\alpha, \beta) \in (1, 2) \times (1, 2)$ that correspond to the AMMs in $\tilde{H}^{-1}(Beta(\Delta^1; (1, 2) \times (1, 2)))$

The beta space $\tilde{H}^{-1}(Beta(\Delta^1; (1, 2) \times (1, 2)))$ contains many of the AMMs we are well acquainted with. Consider an AMM with the constraint function $C(x, y) = x^a y^b = 1$ where $a, b > 0$. As a function of x we can write this as

$$y = f(x) = \frac{1}{x^{\frac{a}{b}}}$$

Balancer specifies AMMs of this form where $a+b=1, a>0, b>0$ [45]. Notice that $\frac{a}{b}=\frac{1}{b}-1=\frac{1-b}{b}$ which takes on all the values in $(0,\infty)$ for $b\in(0,1)$. In either case, these examples are captured by the following family of AMMs which we call the *monomial family*. That is $f(x)=\frac{1}{x^\gamma}$ where $\gamma\in(0,\infty)$. This gives

$$-\frac{v}{1-v}=f'(x)=\frac{-\gamma}{x^{\gamma+1}}$$

which solving for x yields

$$x=\phi(v)=(\gamma\frac{1-v}{v})^{\frac{1}{\gamma+1}}$$

This results in the capitalization function

$$\begin{aligned}c(v)&=v\phi(v)+(1-v)f(\phi(v))\\&=v^{\frac{\gamma}{\gamma+1}}(1-v)^{\frac{1}{\gamma+1}}(\gamma^{\frac{1}{\gamma+1}}+\frac{1}{\gamma^{\frac{\gamma}{\gamma+1}}})\\&\propto v^{\frac{\gamma}{\gamma+1}}(1-v)^{\frac{1}{\gamma+1}}\end{aligned}$$

Setting $\alpha-1=\frac{\gamma}{\gamma+1}$ and $\beta-1=\frac{1}{\gamma+1}$ and solving gives

$$\begin{aligned}\frac{\alpha-1}{2-\alpha}&=\frac{2-\beta}{\beta-1}\\\implies \beta&=3-\alpha\end{aligned}$$

This gives a line through $(1,2)\times(1,2)$ showing that the monomial family effectively occupies this subspace of the beta space.

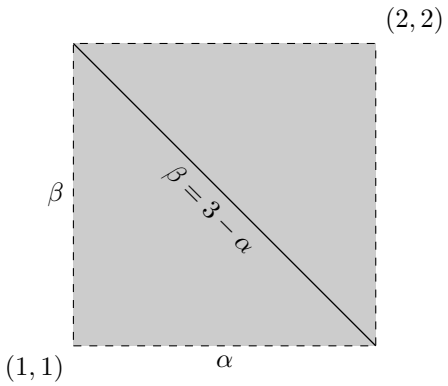

Figure 5.2: $\beta(\alpha)=3-\alpha$ line corresponding to the Balancer (monomial) family of AMMs in $\tilde{H}^{-1}(Beta(\Delta^1;(1,2)\times(1,2)))$

5.2.2 Symmetric AMMs

Recall, from Figure 5.5, that the valuation that maximizes an AMM's capitalization is not necessarily $\frac{1}{2}$.

There is however, a family of AMMs where this is indeed the case.

Definition 5.2.1. *We say a 2D AMM $A := (x, f(x))$ is symmetric if $f = f^{-1}$.*

Both the Uniswap and Curve definitions satisfy this property [6].

Lemma 5.2.2. *The stable state map for any symmetric AMM satisfies $f(\phi(v)) = \phi(1 - v)$*

Proof. Let $g = f^{-1} = f$.

$$f'(y) = g'(y) = \frac{1}{f'(f^{-1}(y))} = \frac{1}{f'(f(y))} = \frac{1}{f'(x)} = -\frac{(1-v)}{v} = \frac{-(1-v)}{1-(1-v)}$$

Thus $y = \phi(1 - v)$. □

Theorem 5.2.3. *Any symmetric AMM has maximum capitalization at $\boldsymbol{v} = (\frac{1}{2}, \frac{1}{2})$.*

Proof. From the proof of Lemma 4.1.2 we know $\phi'(v) < 0$ for $v \in (0, 1)$, so $\phi(v)$ is strictly decreasing. Applying Theorem 4.1.3 and Lemma 5.2.2 tells us that capitalization is maximized when $\phi(v) = x = y = \phi(1 - v)$. Because ϕ is strictly decreasing, $\phi(v) = \phi(1 - v)$ can only occur if $v = 1 - v$ or $v = \frac{1}{2}$. □

Is the symmetry in the AMM function f reflected in some kind of symmetry in the capitalization function? The following theorem gives us such a characterization.

Theorem 5.2.4. *An AMM $A := (x, f(x))$ is symmetric iff $c(v) = c(1 - v)$.*

Proof. First assume A is symmetric so $f(\phi(v)) = \phi(1 - v)$. So we have

$$c(v) = v\phi(v) + (1 - v)\phi(1 - v)$$

But then

$$c(1 - v) = (1 - v)\phi(1 - v) + v\phi(v) = c(v)$$

In the other direction, suppose $c(v) = c(1 - v)$. The construction from Theorem 5.1 defines

$$\phi(v) = c(v) + (1 - v)c'(v)$$

so

$$\begin{aligned}\phi(1 - v) &= c(1 - v) - vc'(1 - v) \\ &= c(v) - vc'(v) = f(\phi(v))\end{aligned}$$

meaning f is symmetric. □

If we have an AMM $A := (x, f(x))$ we write c_f or c_A to denote its capitalization function.

The linear structure of the subspace of symmetric AMMs suggests a more general result.

Lemma 5.2.5. *Symmetric AMMs are closed under convex combinations.*

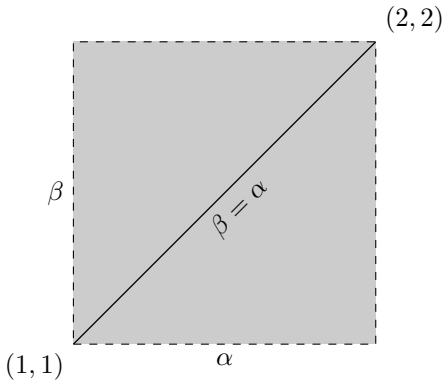

Figure 5.3: The $\beta(\alpha) = \alpha$ line corresponding to all the symmetric AMMs in $\tilde{H}^{-1}(B(\Delta^1; (1,2) \times (1,2)))$

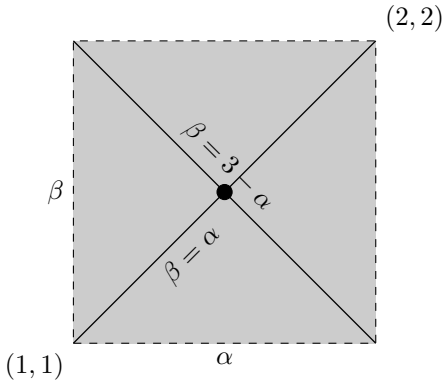

Figure 5.4: The point of intersection of the $\beta(\alpha) = \alpha$ line and the $\beta(\alpha) = 3 - \alpha$ line corresponds to the constant-product AMM $xy = 1$. Namely it is contained in the space of symmetric AMMs and the beta space.

Proof. Let c_A, c_B be the capitalization functions for two symmetric AMMs A, B. Let $\alpha_1, \alpha_2 \geq 0$ where $\alpha_1 + \alpha_2 = 1$. Define $c_{AB}(v) = \alpha_1 c_A(v) + \alpha_2 c_B(v)$. We have

$$\begin{aligned} c_{AB}(1-v) &= \alpha_1 c_A(1-v) + \alpha_2 c_B(1-v) \\ &= \alpha_1 c_A(v) + \alpha_2 c_B(v) \\ &= c_{AB}(v) \end{aligned}$$

so $c_{AB}(v)$ is symmetric. □

5.2.3 Other Miscellaneous AMMs

A function $C : \mathbb{R}^n_{>0} \to \mathbb{R}$ is *homogeneous* if for all $t > 0$ and any $\boldsymbol{x} \in \mathbb{R}^n_{>0}$, $C(t\boldsymbol{x}) = t^k C(\boldsymbol{x})$ for some integer k. Uniswap's and Balancer's constraint functions are examples of homogeneous functions. For constant product AMMs where $C(\boldsymbol{x}) = \prod_{i=1}^n x_i$, for any $t > 0$, $C(t\boldsymbol{x}) = t^n C(\boldsymbol{x})$. Similarly Balancer's constraint is $C(\boldsymbol{x}) = \prod_{i=1}^n x_i^{w_i}$ for some valuation $\boldsymbol{w} = (w_1, \ldots, w_n)$, so $C(t\boldsymbol{x}) = tC(\boldsymbol{x})$. Homogeneous AMM functions allow the relationships between fee parameters and pool assets to be easily computed.

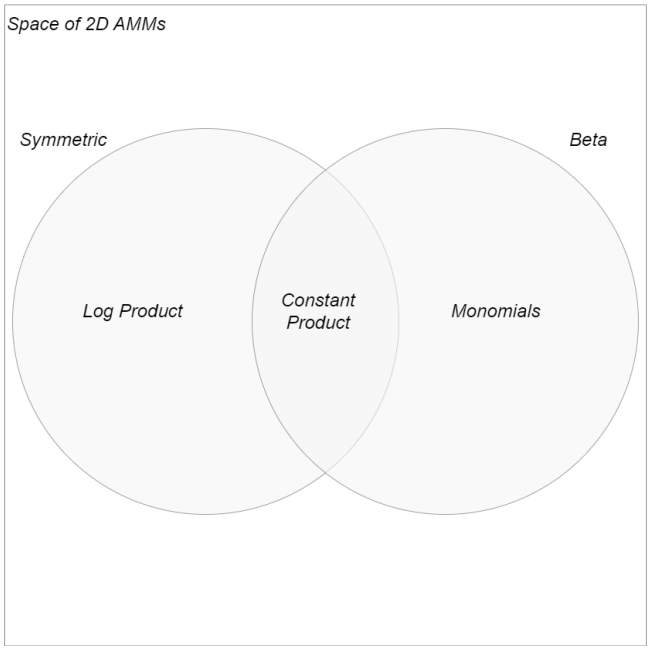

Figure 5.5: Overview of some subspaces of 2D AMMs and their relationships.

Even though most AMMs used in practice tend to be homogenous and often symmetric, there are examples of AMMs that fit within our framework that satisfy neither or only some of these properties.

5.2.4 Example: Skewed constant-product

Here is an example of an AMM which is neither homogeneous nor symmetric.

Consider the constraint $C(x, y) = x^2y - x = 1$. Explicitly we can write this as $f(x) = \frac{1}{x} + \frac{1}{x^2}$. This trivially satisfies our AMM axioms. If C were homogenous there would be some integer k such that $C(2) = 2^k C(1)$. But note that

$$2^k C(1) = -2^k$$

and

$$C(2) = 5$$

Clearly there is no integer k such that $-2^k = 5$ so C is not homogeneous.

5.2.5 Example: Log-Product

Here is an example of an AMM which is not homogenous but is symmetric.

Consider the constraint $C(x, y) = \log(x + 1)\log(y + 1) = 1$. We can write this as $f(x) = e^{\frac{1}{log(x+1)}} - 1$. So

$f'(x) = \frac{-e^{\frac{1}{log(x)}}}{xlog^2(x)} < 0$ and $f''(x) = e^{\frac{1}{log(x)}}(\frac{1}{x^2log^4(x)} + \frac{2}{x^2log^3(x)} + \frac{1}{x^2log^2(x)})$. Since

$$\frac{1}{x^2log^4(x)} + \frac{2}{x^2log^3(x)} + \frac{1}{x^2log^2(x)}$$
$$= \frac{1 + 2log(x) + log^2(x)}{x^2log^4(x)}$$

and

$$1 + 2log(x) + log^2(x)$$

is minimized at $x = \frac{1}{\sqrt{e}}$ with value $\frac{3}{4} > 0$, we know $f''(x) > 0$. Finally when $x \to \infty$ we have $log(y + 1) = 0$ so $y = 0$. By symmetry $x = 0$ when $y \to \infty$. So $f(x)$ is indeed an AMM. It is also trivially symmetric. Again, if C were homogeneous there would be some integer k where $C(2) = 2^k C(1)$. But we have

$$2^k C(1) = 2^k(2log2 - 1)$$

and

$$C(2) = 2log3 - 1$$

and the equation

$$2^k = \frac{2log3 - 1}{2log2 - 1}$$

has no integer solution. Thus, C is not homogeneous.

5.3 Dynamic AMMs

So far we have proposed several ways to quantify the costs associated with AMMs. Now we turn our attention to strategies for adapting to cost changes. Here we summarize two broad strategies motivated by our proposed cost measures. We focus on adjustments that might be executed automatically, without demanding additional liquidity from providers.

5.3.1 Valuation Change

Suppose an AMM learns, perhaps from a trusted Oracle service, that its assets' market valuation has moved away from the AMM's current stable state, leaving the providers exposed to substantial divergence loss. Specifically, suppose $A := (x, f(x))$ has valuation v_1 with stable state (a_1, b_1), when it learns that the market valuation has changed to v_2 with stable state (a_2, b_2).

An arbitrage trader would move A from (a_1, b_1) to (a_2, b_2), pocketing a profit. Informally, A can eliminate that divergence loss by "pretending" to conduct that arbitrage trade itself, leaving the state the same, but moving the curve. We call this strategy *pseudo-arbitrage*.

A changes its function using linear changes of variable in x and y. Suppose $a_1 > a_2$ and $b_2 > b_1$. First, replace x with $x - (a_1 - a_2)$, shifting the curve along the X-axis. Next, replace y with $y - (b_2 - b_1)$, shifting the curve along the y-axis. The transformed AMM is now $A' := (x, f(x - (a_1 - a_2)) - (b_2 - b_1))$. The current state (a_1, b_1) still lies on the shifted curve. but now with slope $\frac{v_2}{v_2 - 1}$, matching the new valuation.

The advantage of this change is that A's providers are not longer exposed to divergence loss from the new market valuation. The disadvantage is that A now has more units of X than it needs, but not enough units of Y to cover all possible trades. The AMM must refuse trades that would lower its Y holdings below zero, and there are $(a_1 - a_2)$ units of X inaccessible to future AMM traders. The liquidity providers might withdraw this excess, they might "top up" with more units of Y to rebalance the pools, or they might leave the extra balance to cover future pseudo-arbitrage changes. The AMM A's ability to conduct trades only while the valuation stays within a certain range is similar to Uniswap v3's "concentrated liquidity" option.

Previously, we have defined AMMs to satisfy boundary conditions $f(0) = \infty$ and $f(\infty) = 0$. Pseudo-arbitrage produces AMMs that violate these boundary conditions, although they continue to satisfy the other AMM axioms.

5.3.2 Distribution Change

Instead, we will focus on a strategy that ensures that when we modify the original AMM, it continues to be an AMM.

Suppose an AMM learns, perhaps from its recent trading history, some probability distribution over future market valuations. The AMM may replace its current function with one that improves some expected cost measure, say, reducing expected load or increasing expected capitalization. Replacing AMM $A := (x, f(x))$, in the stable state for the market valuation, with another $\widetilde{A} := (x, \widetilde{f}(x))$, must follow certain common-sense rules.

First, any such replacement should not change the AMM's reserves: if the AMM is in state $(a, f(a))$, then the updated AMM is in state $(a, \widetilde{f}(a))$ where $f(a) = \widetilde{f}(a)$. Adding or removing liquidity requires the active participation of the AMM's providers, which can certainly happen, but not as part of the kind of automatic strategy considered here.

Second, any such replacement should not change the AMM's current exchange rate: if the AMM is in state $(a, f(a))$, then the updated AMM is in state $(a, \widetilde{f}(a))$ where $f'(a) = \widetilde{f}'(a)$. To do otherwise invites further divergence loss. If $(a, f(a))$ is the stable state for the current valuation, and $f'(a) \neq \widetilde{f}'(a)$, then $(a, \widetilde{f}(a))$ is not stable, and a trader can make an arbitrage profit (imposing divergence loss) by moving the AMM's state back to the stable state.

For example, an AMM A's *expected capitalization* under distribution p is

$$E_p[A] := \int_0^1 p(v) \boldsymbol{v} \cdot \Phi(v) dv,$$

where $\Phi(v) = (\phi(v), f(\phi(v)))$.

5.3.3 Expected Load

We have seen that cost measures such as divergence loss, linear slippage, angular slippage, and load cannot be bounded in the worst case. Nevertheless, these costs can be shifted. Not all AMM states are equally likely. For

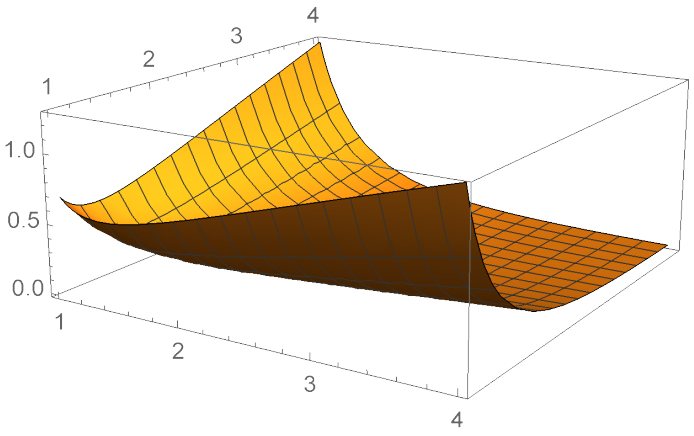

Figure 5.6: Expected load for $A := (x, 1/x)$ at $1/2$ as a function of Beta distribution parameters. (Mathematica source is shown in Appendix A.)

example, one would expect stablecoins to trade at near parity [23].

Suppose we are given a probability density for future valuations. This distribution might be given *a priori*, or it may be learned from historical data. Can we compare the behavior of alternative AMMs given such a distribution?

Let $p(v')$ be the distribution over possible future valuations. The expected load when trading X for Y starting in the stable state for valuation $\boldsymbol{v}$ is

$$\int_0^v P(v'|v' < v)\,\mathrm{load}_X(v, v'; A)dv'$$

Weighting this expectation with the probability $P(v > v')$ that the trade will go in that direction yields

$$P(v > v')\int_0^v P(v'|v' < v)\,\mathrm{load}_X(v, v'; A)dv' = \int_0^v p(v')\,\mathrm{load}_X(v, v'; A)dv'.$$

Define the *expected load* of AMM A at valuation v to be:

$$E_p[\mathrm{load}(v; A)] := \int_0^v p(v')\,\mathrm{load}_X(v, v'; A)dv' + \int_v^1 p(v')\,\mathrm{load}_Y(v, v'; A)dv'.$$

Of course, one can compute the expected value of any the measures proposed here, not just load.

Figure 5.6 shows the expected load for $A := (1, 1/x)$, starting at valuation $v = 1/2$, where the expectation is taken over the $Beta(\alpha_1, \alpha_2)$ distributions where parameters α_1, α_2 range independently from 1 to 4. Inspecting the figure shows that symmetric distributions $\beta(\alpha, \alpha)$, which are increasingly concentrated around $1/2$ as α grows, yield decreasing loads as the next valuation becomes increasingly likely to be close to the current one. By contrast, asymmetric distributions, which favor unbalanced valuations, yield higher loads because the next valuation is likely to be farther from the current one.

5.3.4 Expected Divergence Loss

If you're a liquidity provider one cost measure you care about is divergence loss. Namely how much do you lose if a trader reacts more quickly to you in response to a market change. If the current state is $(\phi(v), f(\phi(v))$ then it is not unreasonable to consider a probability distribution conditional on the current state $p(v'|v)$.

The liquidity provider wants to minimize

$$E_{p(\cdot|v)}[\text{divloss}(v, v'; A)] := \int_0^1 p(v'|v)\,\text{divloss}(v, v'; A)dv'$$

Because $\text{divloss}(v, v'; A) \geq 0$ we know

$$\int_0^1 p(v'|v)\,\text{divloss}(v, v'; A)dv' \geq 0 \tag{5.1}$$

By definition

$$\begin{aligned}
0 &\leq \int_0^1 p(v'|v)\,\text{divloss}(v, v'; A)dv' \\
&\leq \int_0^1 p(v'|v)\boldsymbol{v}' \cdot [\Phi(v) - \Phi(v')]dv' \\
&\leq \int_0^1 p(v'|v)\boldsymbol{v}' \cdot \Phi(v)dv' - \int_0^1 p(v'|v)\boldsymbol{v}' \cdot \Phi(v')dv' \\
\int_0^1 p(v'|v)\boldsymbol{v}' \cdot \Phi(v')dv' &\leq \int_0^1 p(v'|v)\boldsymbol{v}' \cdot \Phi(v)dv'
\end{aligned}$$

Let $\mu = E_{p(\cdot|v)}[v']$, the mean of the distribution $p(v')$. Note that the RHS is just

$$\begin{aligned}
&\int_0^1 p(v'|v)\boldsymbol{v}' \cdot \Phi(v)dv' \\
&= \int_0^1 p(v'|v)[v'\phi(v) + (1 - v')f(\phi(v))]dv' \\
&= \phi(v)\int_0^1 p(v'|v)v'dv' + f(\phi(v))\int_0^1 p(v'|v)(1 - v')dv' \\
&= \phi(v)\mu + f(\phi(v))(1 - \mu)
\end{aligned}$$

Additionally, the LHS is simply the expected capitalization with respect to $p(v'|v)$. This tells us that to minimize expected divergence loss, we need to maximize expected capital.

An optimal AMM A' for p at v, if one exists, would achieve the upper bound where expected capital is maximized:

$$E_p[A'] = \int_0^1 p(v'|v)\boldsymbol{v}' \cdot \Phi(v')dv' = \mu\phi(v) + f(\phi(v))(1 - \mu) \tag{5.2}$$

Even if the optimal isn't achieved, we at the very least have the distribution-dependent upper bound

$$E_p[A'] \leq \mu \cdot \Phi(v)$$

Additionally, by the Cauchy-Schwarz inequality we have

$$\begin{aligned}
|\mu \cdot \Phi(v)| &\leq ||\mu||_2 * ||\Phi(v)||_2 \\
&\leq ||\Phi(v)||_2 \qquad \text{(Since } ||\mu||_2 \leq 1 \text{ on } [0, 1]\text{)}
\end{aligned}$$

leading to the distribution-independent upper bound

$$E_p[A'] \leq ||\Phi(v)||_2$$

5.3.5 Example: Constant-Product

Let's start with arguably the simplest example, the constant-product. Recall, it is symmetric with

$$\phi(v) = \sqrt{\frac{1-v}{v}}$$

so

$$\begin{aligned}\text{cap}(v; A) &= v\phi(v) + (1-v)\phi(1-v) \\ &= 2\sqrt{v(1-v)}\end{aligned}$$

Assume that $v = \frac{1}{2}$ and $p(v'|v)$ is the uniform distribution, so $\mu = v = \frac{1}{2}$.

The expected capital is

$$\begin{aligned}&\int_0^1 \text{cap}(v; A)dv \\ &= 2\int_0^1 \sqrt{v(1-v)}dv\end{aligned}$$

Now, consider an ellipse with center $(\frac{1}{2}, 0)$ with minor radius of $\frac{1}{2}$ and major radius of 1. If $y(v)$ is the upper half of the ellipse, the implicit ellipse equation is:

$$\frac{(v-\frac{1}{2})^2}{(\frac{1}{2})^2} + \frac{y(v)^2}{1^2} = 1$$

Solving for $y(v)$ gives

$$\begin{aligned}y(v) &= \sqrt{1 - 4(v - \frac{1}{2})^2} \\ &= \sqrt{1 - 4v^2 + 4v - 1} \\ &= 2\sqrt{v(1-v)}\end{aligned}$$

This is exactly $\text{cap}(v; A)$ for the constant-product AMM. The area of the ellipse is simply $\pi * (\frac{1}{2}) = \frac{\pi}{2}$. The expected capital is half of this area, namely $\frac{\pi}{4}$. Thus

$$\begin{aligned}&\int_0^1 cap(v; A)dv \\ &= \int_0^1 2\sqrt{v(1-v)} = \frac{\pi}{4}\end{aligned}$$

With no extra constraints other than maximizing expecting capital, we get

$$\begin{aligned}&\mu \cdot \Phi(v) = 1 \\ &> \frac{\pi}{4} = \int_0^1 cap(v; A)dv\end{aligned}$$

so the constant-product doesn't achieve the optimal value.

5.3.6 Optimal AMM

The previous section demonstrated that the constant-product doesn't achieve the upper bound for expected capital. In this section we formally define the notion of an optimal AMM using expected capital. Additionally, we show that while the optimum cannot ever be achieved it can be approximated with AMMs. We demonstrate this with a family of capitalization functions that converge to the optimal to the uniform distribution when $v = \frac{1}{2}$.

Recall that it is a necessary condition that for any AMM A, that $\text{cap}(v; A)$ is a strictly concave function. This is what allowed us to show the uniqueness of the optimal point $v = v^*$.

Let $C_2(v^*, \alpha)$ to be the set of twice-differentiable functions on $[0, 1]$ with a maximum value α occurring at some $v^* \in [0, 1]$. Note that we intentionally are not restricting this set to only include strictly concave functions, it is a much larger set. Given some fixed v and a distribution $p(v'|v)$, we then define the *optimal capitalization* with respect to v, p as

$$C(v, \alpha, p) = \sup_{c \in C_2(v,\alpha)} \int_0^1 c(v')p(v'|v)dv'$$

An optimal $\tilde{c} \in C_2(v^*, \alpha)$ (if it exists), would then need to satisfy

$$\int_0^1 \tilde{c}(v')p(v'|v)dv' = C(v, p)$$

As we will see, $\tilde{c}$ will exist. However, it won't ever be a capitalization function. Instead, the best we can do is a find a sequence of capitalization functions converging to the optimal value.

If A is an AMM with capitalization function c, then we know from Section 5.1, that $c = \lambda q$ for $q \in P(\Delta^1)$. Since this has to hold everywhere, we know it must hold at $v = v^*$ so $\lambda = \frac{c(v^*)}{q(v^*)}$. So we can just write $c(v') = \frac{c(v^*)}{q(v^*)}q(v')$ where $q \in P(\Delta^1)$.

Notice that $\frac{q(v')}{q(v^*)} < 1$ whenever $v' \neq v^*$ so $\frac{c(v^*)q(v')}{q(v^*)} < c(v^*)$. Thus we get

$$\begin{aligned}
&E_p[A] \\
&= \int_0^1 c(v')p(v'|v)dv' \\
&= \int_0^1 \frac{c(v^*)q(v)}{q(v^*)}p(v'|v)dv' \\
&< \int_0^1 c(v^*)p(v'|v)dv' && (\tfrac{c(v^*)q(v')}{q(v^*)} < c(v^*) \text{ for all } v \neq v^*) \\
&= c(v^*)||p||_1 \\
&= \alpha
\end{aligned}$$

Setting $\tilde{c}(v) = \alpha$ to be a constant function, then clearly gives $\tilde{c}(v) \in C_2(v^*, \alpha)$. Additionally, we have

$$\alpha = \int_0^1 \tilde{c}(v')p(v'|v^*)dv' = C(v, \alpha, p)$$

Since $\tilde{c}$ is a constant function, it clearly cannot be a capitalization function because it isn't strictly concave. Thus an optimal cannot be achieved if we restrict ourselves to the space of valid capitalization functions.

We now give an example to show that it's possible to construct a sequence of capitalization functions $\{f_n(v')\}_{n\in\mathbb{N}}$, all achieving their maximum at $v' = v^*$ with value $\alpha = f_n(v^*)$, such

$$\int_0^1 f_n(v')p(v'|v^*)dv' \to C(v^*, \alpha, p) = \alpha$$

for p the uniform distribution $U(0, 1)$.

Dividing all sides by α, it is enough to just show we can construct a sequence of capitalization functions $f_n(v')$ with $f_n(v^*) = 1$ where

$$\int_0^1 f_n(v')p(v'|v^*)dv' \to 1 \tag{5.3}$$

We now construct such a sequence for the uniform distribution where $v^* = \frac{1}{2}$.

Consider the parametrized family of functions

$$f_n(v) = 2^{\frac{2}{n}} \sqrt[n]{v(1-v)}$$

restricted to $v \in [0, 1]$, where $n \in \mathbb{R}_{++}$.

Notice a few things. When $n = 2$ this is just $\text{cap}(v; A)$ where A is the constant-product function given by $xy = 1$. We also have

$$\begin{aligned} &f_n(0) = 0 \\ &f_n(\frac{1}{2}) = 1 \\ &f_n'(\frac{1}{2}) = 0 \\ &f_n''(v) < 0 \qquad \text{(all } v \in [0, 1]\text{)} \end{aligned}$$

Also $f_n(v) = f_n(1 - v)$, namely it is symmetric.

We also have

$$\begin{aligned} &\int_0^1 f_n(v)dv \\ &= \frac{\Gamma(1 + \frac{1}{n})^2}{\Gamma(2 + \frac{2}{n})} \end{aligned}$$

where Γ is the well-known gamma function. It is known that the gamma function is continuous so

$$\begin{aligned} &\lim_{n\to\infty} \int_0^1 f_n(v)dv \\ &= \lim_{n\to\infty} \frac{\Gamma(1 + \frac{1}{n})^2}{\Gamma(2 + \frac{2}{n})} \\ &= \frac{\Gamma(1)^2}{\Gamma(2)} = 1 \end{aligned}$$

Thus $||f_n||_1 \to 1$.

Chapter 6

Cross-Chain Protocols

Many aspects of decentralized finance can be understood as an extension of classical distributed computing. In this chapter, we illustrate this evolution through two interrelated notions: failure and fault-tolerance.

In classical distributed computing, a failure to complete a multi-party protocol is typically attributed to hardware malfunctions. A fault-tolerant protocol is one that responds to such failures by rolling the system back to an earlier consistent state. In the presence of Byzantine failures, a failure may be the result of an attack, and a fault-tolerant protocol is one that ensures that attackers will be punished and victims compensated.

In modern decentralized finance however, failure to complete a protocol can be considered a legitimate option, not a transgression. For example, in Section 6.3, we will see a situation in which the expected honest behavior allows a party choose to either engage or renege from following a protocol.

Consider the classical problem of *atomic commitment*: how can we install updates at multiple databases or ledgers in such a way that guarantees that if all goes well, all updates are installed, but if something goes wrong, all updates are discarded. The classical challenge is, of course, tolerating failures: databases can crash or communication can be lost or delayed.

This is one of the oldest problems in distributed computing, and not surprisingly, it is central to key problems in DeFi. First, we explore the technical solutions (protocols), where DeFi has tended to borrow, whether consciously or not, from prior solutions in distributed computing. Later, in Chapter 7 we explore underlying conceptual frameworks, where DeFi extends the notion of fault-tolerance well beyond the classical models of distributed computing.

In Section 6.1, we review the well-known *two-phase commit protocol* [8], a classical technique for making atomic updates to independently-failing databases in a distributed system. In Section 6.2, we consider the *cross-chain atomic swap* problem, where mutually-suspicious parties exchange assets atomically across distinct nodes. The simplest atomic swap protocols are based on *hashed timelocked contracts* [40, 48] (HTLCs). The HTLC primitive is described in Section 6.1.1. Technically, HTLC protocols closely resemble classical two-phase commit. The principal difference between the two protocols is in their underlying conceptual frameworks. In two-phase commit, a failure is typically an operational malfunction at a node or a network, while in atomic swap, a failure could also be a malicious

action chosen by an adversarial party.

This distinction becomes more pronounced in Section 6.3. In both two-phase commit and atomic cross-chain swap protocols, fault tolerance means that if one party falls silent in the middle, the other parties are eventually made whole: database replicas are eventually restored, and escrowed assets are eventually refunded. For distributed computing's two-phase commit, the story ends there, but for DeFi's atomic swap protocol, the story has just begun. In finance, the ability to abandon or to complete an in-progress swap is called an *option*, and options themselves have value. Any party who abandons an atomic swap should compensate the other parties by paying a small fee called a *premium*. Treating failures as compensated options is alien to classical distributed computing models, where all parties implicitly are on the same team, but it opens up a range of new research challenges for distributed computing. Incorporating premiums into atomic swaps turns out to be a challenging technical problem [59], effectively requiring nesting one atomic commitment protocol within another.

Section 6.3 takes the notion of optionality to the next level. What if one party could sell such an option to another? Alice, who has paid for the option to complete or cancel a swap, should be able to transfer that option to Bob for a fee. Alice would relinquish her power over the swap's outcome, and Bob would assume all of Alice's power, including the power to complete or cancel the swap, and the right to be compensated if another party cancels the swap. This problem is also technically challenging, as it requires embedding yet another atomic commitment mechanism within other nested atomic commitment mechanisms.

Originally, abandoning an atomic commitment protocol was considered a simple operational failure, and the meaning of fault-tolerance was simply to restore integrity and availability. When the parties become autonomous and potentially adversarial, however, failures can become deliberate choices, and the meaning of fault-tolerance must be extended to provide financial compensation to any victims of other parties' choices. Once failures become options (in the financial sense), then those options themselves become assets to be traded.

In Sections 6.1, we give an overview of the basic synchronization tools that are typically used in cross-chain protocols. Section 6.2 provides a survey of how these primitives have been using in trading protocols where mutually distrusting parties want coordinate transfers on multiple distinct nodes. Finally, Section 6.3 presents novel cross-chain protocols that are bootstrapped on top of swap protocols in order to be able to trade cross-chain options.

6.1 Motivation and Synchronization Primitives

Imagine we have a distributed database with a number of replicas. These replicas might be identical, or they may hold different portions of the database (so-called shards). For simplicity, assume Alice's node holds one replica, and Bob's node holds another. A node may *crash* (cease operation), and later *recover* (resume operation). Node memory is divided into *volatile* memory lost on a crash, and *stable* memory that survives crashes.

A *transaction* is a sequence of steps that modifies both replicas. As a transaction executes, Alice and Bob accumulate a list of tentative changes. If the transaction *commits*, those changes take effect, and if the transaction *aborts*, they are discarded.

The *two-phase commit protocol* [8] is a classical technique for ensuring *atomicity*: if a transaction makes tentative changes at both Alice's node and Bob's node, then the transaction either commits at both nodes or aborts at both.

Here is the simplest form of this protocol. One party, say Carol (a trusted third-party), is chosen as the *coordinator*.

1. *Prepare phase*

   - Carol, the coordinator, instructs Alice and Bob to record their tentative changes in stable storage, so they will not be lost in a crash.
   - If Alice is able to write her changes to stable storage, she sends Carol a *yes* vote. At this point, some or all of the database becomes inaccessible pending the outcome of the transaction. If for any reason, Alice cannot save her changes, she sends Carol a *no* vote. Bob does the same.

2. *Commit phase*

   - If Carol receives two *yes* votes, she instructs Alice and Bob to apply their tentative changes, committing the transaction. If Carol receives a *no* vote, or if either Alice or Bob fails to respond in time, she instructs them to discard their tentative changes, aborting the transaction. Before Carol sends her decision to Alice and Bob, she records her decision in stable memory, in case she herself crashes.
   - Alice follows Carol's instructions. If Alice crashes after preparing but before Carol decides, Alice must learn the transaction's outcome from Bob or Carol before resuming use of her database.

This description is vastly simplified, and omits many practical considerations, but it serves as a baseline for the more complex DeFi commitment protocols considered in later sections. The key pattern is that commitment requires that each party agrees to lock up a set of tentative changes, thereby freezing something of value (here, the database) until the outcome of the protocol becomes known.

For our purposes, a *node* is a tamper-proof distributed ledger (or database) that tracks ownership of *assets*. An asset can either be owned by a a *party* or a *contract*. An asset can be a cryptocurrency, a token, an electronic deed to property, and so on. There are multiple nodes managing different kinds of assets between contracts and parties. We focus here on applications where mutually-untrusting parties trade assets across multiple distinct nodes. Because we treat nodes simply as ledgers, our results do not depend on specific blockchain technology (such as proof-of-stake vs proof-of-work). We assume only that nodes/ledgers are highly available, tamper-proof, publicly readable, and are capable of running arbitrary computations.

The distinction between a party and a contract is as follows. A contract is a node-resident program initialized and called by the parties. A party is a non-deterministic agent with economic incentives to make particular particular calls to contracts. A party can create a contract on a node, or call a function exported by an existing contract. Contract code and state are public, so a party calling a contract function knows what code will be executed. Contract code is deterministic because contract execution is replicated, and all executions must agree.

A contract can read or write ledger entries on the node where it resides, but it cannot actively access data from the outside world, including calling contracts on other nodes. Although there are primitives that allow nodes to

communicate (cross-chain proofs: Section 6.1.3), they have weaknesses that make them difficult to use in practice: for example, incompatibility problems, lack of decentralization, and non-deterministic guarantees of message delivery [53]. Therefore, we assume different nodes do not communicate.

A contract on node A can learn of a change to a node B only if some party explicitly informs A of B's change, along with some kind of "proof" that the information about B's state is correct. Contract code is passive, public, deterministic, and trusted, while parties are active, autonomous, and potentially dishonest.

At any given time each party has a given financial *position*. This is characterized by their current and future financial obligations/allowances. Financial positions are managed by rules on contracts that can reside across multiple distinct nodes.

Our execution model is *synchronous*: there is a known upper bound Δ on the propagation time for one party's change to the nodes state, plus the time to be noticed by the other parties. Specifically, valid transactions/calls sent a node will be applied and visible to participants within a known, bounded time Δ.

A party *sending* a contract call doesn't guarantee instantaneous *execution* on the corresponding contract. This is a side-effect of the synchronous network model. If a party sends a call at some time t to a node, in the worst case it arrives to the node and gets executed at time $t + \Delta$. This latency is important when designing protocols. Thus we reserve the usage of *sending* to mean when the contract call leaves the party and use *called*/*created*/*executed* to mean when the call actually arrives at the contract and performs a state update.

As noted, we assume nodes are always available, tamper-proof, and that they correctly execute their contracts. Although parties may display Byzantine behavior, contracts can limit their behavior by rejecting unexpected contract calls.

The synchronous model naturally leads to a round structure. A *protocol* is a sequence of rules every party should follow in each round based on the state of all nodes. Namely, we assume protocols prescribe a finite set of possible allowable behaviors for parties participating in a protocol. Parties that follow the protocol rules are called *compliant* (honest), and those who do not are called *malicious* (adversarial,deviating,etc).

We make standard cryptographic assumptions. Each party has a public key and a private key, and any party's public key is known to all. Messages are signed so they cannot be forged, and they include single-use labels ("nonces") so they cannot be replayed. Parties have access to a collision-resistant hash function $H(\cdot)$.

6.1.1 Hashlocks

A *hashed timelocked contract* (HTLC) or simply a *hashlock* for short, is a pair (h, t) where $h = H(s)$ for some secret s and t, a *timeout*, is some positive multiple of Δ. Implicit to the use of an HTLC are two parties, a *sender* and *receiver* of the associated funds. When a party, the sender, creates a HTLC they temporarily surrender their rights to manage the asset to the contract. This process is called *escrowing* their asset. Afterwards, the *hashlock* contract has the following behavior:

- If a party reveals the secret s before the timeout t (checked by comparing $h = H(s)$), then the contract transfers

the asset's ownership to the receiver

- Otherwise, if s isn't revealed by time t, then the sender gets their asset refunded

The time t is measured relative the HTLC creation time. For example, if a HTLC $(h, k\Delta)$ is created at time T, then it expires at time $T + k\Delta$.

The hashlock is the foundation for many simple protocols that allow trading assets between parties cross multiple nodes. Examples of hashlock-based swaps are given in Section 6.2.

The hashlock primitive relies on two core properties of the hash function $H(\cdot)$. Namely, that is impractical for a party who only knows $h = H(s)$ to be able to recover s, and additionally, that it is unlikely that there is some $s' \neq s$ such that $h = H(s')$. Even with such an idealized functionality, there are very realistic trade scenarios where hashlocks fail, and are not sufficient to solve a given trade problem. Section 7.2 precisely characterizes the class of atomic-swap problems where hashlocks fail.

6.1.2 Path Signatures

We assume each party has public and private key that they can use for creating digitally signed messages. Digital signatures provide a way for a party to vouch or authenticate the legitimacy of a message. We assume digital signatures cannot be forged. We will use the notation $sig(p, m)$ to be the digital signature of a message m by a party p. A party is simply identified and associated with its public key. Keeping its private key hidden is necessary to prevent forgery of its digital signature.

A *path-signature* is a nested series of *digital signatures*. Suppose there are a set of parties $\mathbb{P}$. For the purpose of designing protocols, a *path-signature* is a tuple (h, p, σ) where $h = H(s)$ for some secret s (similar to hashlocks) generated by a party p_0, $p = (p_0, ..., p_k)$ is a sequence of k distinct parties in $\mathbb{P}$ (represented via their public keys), and σ is the result of the following operations:

$$\sigma = sig(p_k, ...sig(p_1, h)...)$$

Namely, σ is the result of parties p_1 through p_k successively digitally signing the hash of party p_0's secret.

Given a path-signature (h, p, σ), it can be verified with a secret s as follows:

- Commit if $h = H(s)$ and σ is a valid digital signature for all parties $(p_1, ..., p_k)$, Abort otherwise

Path-signatures are an essential building block for cross-chain protocols. In Chapter 7, we show they are strictly more powerful than hashlocks in their computational power.

6.1.3 Cross-Chain Proofs

Cross-chain proofs are a primitive that allows for one node to convince another node that it has successfully changed state. Of course, nodes are unable to directly communicate so it is the responsible of parties to route the generated proofs between nodes.

Fix a set of parties $\mathbb{P}$. The cross-chain proof primitive contains two important pieces of state: A vector of *votes* $\boldsymbol{v}$, has $v_i \in \{0, 1, \perp\}$ and $\boldsymbol{v}$ has an entry for each party in $\mathbb{P}$. Additionally, there is an expiration time T after which no more votes will be collected.

It is initialized as follows: Each v_i is initialized to $\perp$. T is set sufficiently large into the future for every party to vote. Assume the primitive has access to the current time t.

A cross-chain proof exposes the following interface to parties:

- $Vote(p, v)$:, if party p calls and this is p's first such call, set $v_i = v$ where is restricted to $v \in \{0, 1\}$
- $Prove()$: if $t < T$ return $\perp$, otherwise if all $v_i = 1$, return 1, else return 0

For our purposes, the implementation of the $Prove()$ function is not of much concern. Namely, we introduce this primitive as a useful tool for building protocols. Additionally, it is also useful as a comparison benchmark for other synchronization primitives.

In particular, we will show in Chapter 7, that path-signatures and cross-chain proofs have the same computational power since they can build a universal primitive called an *atomic broadcast*. However, the main drawback of path-signatures when compared to cross-chain proofs is their efficiency. Namely, Section 7.3 shows an *atomic broadcast* lower bound when implemented using path-signatures.

Access to a cross-chain proof leads to the cross-chain analog to a *two-phase commit* protocol [41]. It assumes there are several distinct nodes with contracts $C_1, ..., C_n$ managing party asset rights. Additionally, there is some coordinator node $\tilde{C}$ responsible for generating cross-chain proofs.

The protocol roughly works as follows:

1. *Prepare phase*
    - All parties *escrow* their respective assets on each relevant contract $C_1, ..., C_n$.
2. *Commit phase*
    - If all escrowed contracts look acceptable to a party P, then they call $Vote(P, 1)$ on $\tilde{C}$, otherwise they call $Vote(P, 0)$
    - Once $t \geq T$, all parties call $Prove()$ to generate a proof to send to all contracts $C_1, ..., C_n$. If a contract receives a proof of 1, then it *Commits* changes, otherwise it *Aborts*

Note the similarities to the *two-phase commit* protocol from the beginning of the chapter. Additionally, this protocol has two key properties we will see repeated throughout this chapter and Chapter 7 for other protocols:

- When all parties are compliant, all contracts *Commit*.
- As long as at least one party is compliant, all contracts $C_1, ..., C_n$ will agree on either *Abort* or *Commit*

6.2 A Simple Swap Protocol

Consider the following scenario. Alice has invested in the guilder cryptocurrency, while Bob has invested in the florin cryptocurrency. Alice and Bob would both like to diversify: Alice wants to trade some of her guilders for florins, and Bob wants the opposite trade. Such an exchange would be almost trivial if both currencies resided on the same node, but florins reside on node F, and guilders on node G. Naturally, Alice and Bob do not trust one another, we are presented with a more difficult version of the earlier Section 6.1 atomic commitment problem: is there a safe way to guarantee that either both transfers happen, or neither happens, *given untrusting participants*.

The two-phase commit protocol is a good start, but it assumes that all parties are acting in good faith. Each node reports honestly whether it was able to prepare, and the coordinator does not lie about the votes it received. Nevertheless, we can build an atomic cross-chain swap by "hardening" the classical two-phase commit protocol.

The notion of *escrow* plays the role of stable storage: an escrow contract is given custody of Alice's coins, along with a hashlock h and a timeout. If s is presented to the contract before the timeout, then Alice's coins are transferred to Bob, and if not, those coins are refunded to Alice. Bob creates a symmetric escrow contract, only with Alice's hashlock and a different timeout.

Here is the hardened two-phase commit protocol.

1. *Prepare phase*

    - Alice transfers her guilders to her escrow contract with timeout 2Δ.
    - When Bob verifies that Alice's coins have been escrowed, he transfers his florins to his escrow contract with timeout Δ.

2. *Commit phase*

    - When Alice verifies that Bob has put his florins in escrow, she sends her secret to the escrow contract on node F, unlocking and collecting Bob's florins. Alice has now recorded her secret on node F.
    - As soon as Alice's secret appears on the node F, Bob forwards s to the escrow contract on node G, unlocking and collecting Alice's guilders.

Placing coins in escrow is the analog of writing updates to stable storage and then voting to commit: each party gives up the ability to back out. For two-phase commit, it does not matter which party writes first to stable storage. For the atomic swap, however, Alice must escrow first, and Bob second, because Alice controls the secret, and she could steal Bob's coins if he escrowed first. The choice of timeouts is critical: if Bob's timeout were 2Δ instead of Δ, then Alice could wait until the timeout was about to expire to claim Bob's florins, leaving Bob without enough time to claim Alice's guilders. Atomic swap is less forgiving than two-phase commit: if Bob falls asleep and fails to claim Alice's guilders before 2Δ timeout, then Bob loses the coins on both nodes.

The timeouts are set sufficiently large, so that as long as a party follows the protocol (they are compliant), they are guaranteed if they lose their asset, they will also gain the other parties asset. For example, if Alice loses asset

a she must have revealed her secret (since Bob doesn't know s). A compliant Alice would only have done so by revealing it to node B, thereby receiving asset b. Thus regardless if Alice loses asset a, she is guaranteed to get asset b as long as she is compliant. Similarly, say compliant Bob loses asset b. In the worst-case, Alice's escrowed HTLC expires in one round. Thus a compliant Bob will have enough time to reveal s to node A and claim Alice's asset a.

Additionally, if both parties are compliant, both assets get swapped. After the first two rounds, both HTLCs have been created on both nodes A, B by compliant Alice and Bob respectively. After the third and fourth rounds, both Alice and Bob have revealed s on nodes B, A respectively, completing the transfer.

This swap problem can be defined much more generally. Multiple parties trading multiple assets across distinct nodes can be represented as a directed graph, where vertices represent parties and edges represent asset transfers on nodes [40]. For the simple-two party swap protocol illustrated above, hashlocks were enough to ensure Alice and Bob could trade their assets. However, we will show in Section 7.2, that once the directed graph representing the trade becomes sufficiently large (in terms of number of edges), it is in fact impossible to use hashlocks to conduct a trade fairly for all parties.

6.3 Layered Protocols

While technically correct, swap protocols (Section 6.2) are flawed: once both assets are escrowed, Alice can take her time deciding whether to trigger the swap. Cryptocurrencies are notoriously volatile, so if the value of Bob's asset (guilders) goes up relative to Alice's asset (florins) before the timeout expires, she can choose to complete the swap at the last minute. If the value goes down, she is free to walk away without penalty. Bob may be reluctant to accept such a deal.

This protocol is unfair to Bob because only Alice has *optionality*: at the end, he cannot back out of the deal, but she can. If she does back out, he gets his guilders back, but only after a possibly long delay while the market is moving against him. Bob thus incurs the opportunity cost of not being able to use his coins while they are escrowed. By contrast, in conventional finance, this deal would be structured as an *option contract*, where Alice pays Bob a fee, called a *premium*, to compensate him if she walks away without completing the deal. This serves as a form of insurance for Alice, if she is anticipating that the price of purchasing guilders will increase.

In the world of *decentralized finance* (Defi), florins and guilders are managed on distinct nodes, Alice and Bob are autonomous agents, and they do not trust each other. Moreover, they have no recourse to third-party arbiters or to courts of law.

Cross-chain atomic swap options require carefully-defined distributed protocols [44, 59]. The protocol proposed by Xue and Herlihy [59] is structured like an iterated two-phase commit protocol: in the first phase the parties escrow premiums. If all goes well, in the next phase they escrow coins. If all goes well in the final phase they complete the swap. Any party who drops out of the protocol ends up paying a premium to the other, and in the end, Alice has the optionality, but Bob has been compensated for his risk.

As long as Alice has optionality, that optionality has value. Alice should be able to sell her position to a third

party Carol. (Bob may also want to sell his position to a third party David). For starters, we focus on Alice's position transfer since Bob's transfer will be similar. There are many reasons Alice and Carol might agree to such a deal. Alice may want to liquidate her position because she needs cash. Perhaps Alice and Carol have different opinions on the future values of florins versus guilders, or they have different levels of risk tolerance.

In conventional finance, such a transfer is simple: Bob grants Alice an options contract, Alice signs over that contract to Carol, and all agreements are enforced by civil law. In the lawless world of decentralized finance, by contrast, transferable options require a carefully-crafted distributed protocol.

The distributed protocols presented in this section are a building block towards a more ambitious goal. One can imagine more complex cross-chain deals where parties acquire various rights and obligations (options, futures, derivatives, and so on). It should be possible for a party who holds unrealized rights or obligations to sell those rights or obligations to another party, atomically transferring its position without being hindered by any third party. Here, we show how to make certain cross-chain options, *transferable*.

The following discussion is organized as follows: We provide an overview of our proposed transferable two-party swap protocols in Section 6.3.1 and detailed protocols are described in the following sections. We prove security properties of our proposed protocols in Section 6.3.5. Detailed pseudocode is also provided in Appendix B for the protocols.

Informally, our proposed protocols can address option transfer in the following scenarios:

1. Alice (the option owner, i.e the one who has a right to buy an asset) transfers her position, that is, her right to exercise, to another party (Carol).

2. Bob (the option provider, i.e. the one who provides the owner the right) transfers his position, namely his obligation, to another party (David).

3. Alice and Bob concurrently transfer their positions to Carol and David, respectively.

Take the option position transfer between Alice and Carol for example. The proposed transfer protocol ensures that if Alice and Carol are honest, then Alice relinquishes her rights, Carol assumes Alice's rights, Carol appropriately compensates Alice, and Bob cannot veto the transfer. If Carol cheats, then Alice's rights remain with Alice.

A *transferable atomic swap option* (or "transferable swap") protocol roughly must satisfy the following properties. (These properties are defined more precisely in Section 6.3.5.)

- *Liveness*: If Alice and Carol both follow the protocol, then (1) Carol acquires the right to buy Bob's coins at the same price and deadline, (2) Alice loses that right, and she is paid by Carol, (3) Bob cannot veto the transfer.

- *Safety*: As long as one of Alice or Carol follows the protocol, (1) the protocol completes before the option expires, and (2) when the protocol completes, exactly one of Alice or Carol has the right to trigger the atomic swap with Bob, (3) Bob's position in the option does not change if he conforms.

Here is a high-level summary of our protocol. The contract linking Alice and Bob is a delayed swap: Alice escrows her asset, then Bob escrows his. These escrow contracts are controlled by a *hashlock* h. Until each contract's timeout expires, that contract will complete its side of the swap when Alice produces a *secret* A_1 such that $h = H(A_1)$, where $H(\cdot)$ is a predefined cryptographic hash function. Alice has optionality because she alone knows the secret.

Note that the option we discuss here is different from that found in traditional finance, where the law enforces all required transactions. The option we describe here requires Alice to escrow her asset before acquiring optionality. If she doesn't commit her assets to be locked during the protocol, she could always launch a *double spend* attack to spend those assets elsewhere but meanwhile claim Bob's assets by exercising the option.

Here is how Carol can buy Alice's position. First, there must be enough time to complete this transfer before Alice's option expires. Roughly speaking, Carol generates a secret, then Alice and Carol swap the roles of Carol's and Alice's secrets: following the transfer, Alice's secret will no longer trigger the swap with Bob, but Carol's secret will.

There are two contracts (discussed in detail below). Contract AB controls Alice's possible payment to Bob, while BA controls Bob's payment to Alice. The optionality transfer itself is structured like a cross-chain swap, exchanging roles instead of assets. In the first phase, Alice marks contracts AB and BA as *mutating*, temporarily preventing any transfer to or from Bob. This step prevents Alice or Carol from creating a chaotic situation by triggering a partial asset swap with Bob while the optionality transfer is in progress. In the second phase, Carol replaces the (hash of) Alice's secret and address in both AB and BA with the (hash of) Carol's secret and address. Just as for regular swaps, the optionality transfer protocol will time out and revert if Alice or Carol fails to take a step in time.

There is still a danger that Alice and Carol might cheat Bob by making inconsistent changes to contracts AB and BA. Because AB and BA cannot coordinate directly, the protocol has a built-in delay to give Bob an opportunity to *contest* a malformed transfer and to revert its changes. If the transfer is well-formed, Bob can expedite the protocol by actively approving of the transfer, perhaps in return for an extra premium. If Bob remains silent, the protocol will proceed after the delay expires. Bob contests the transfer by providing proof that Alice signed inconsistent changes to AB and BA.

To keep the presentation uncluttered, we omit some functionality that would be expected in a full protocol, but that is not essential for optionality transfer. For example, there would be additional steps where parties deposit premiums to compensate one another if one party leaves the other's assets temporarily trapped in escrow, where Alice pays Bob a premium to encourage prompt transfer approval, and where Alice posts a bond to be slashed if she is caught sending inconsistent information to AB and BA.

6.3.1 Two-Party Transferable Swap

We will now describe several protocols for transferring option positions. First we will describe a protocol that allows Alice to transfer her position as the option owner in a swap with Bob, to a third-party Carol. Next, we will give a protocol that allows Bob to transfer his position as the option provider in a swap with Alice, to a third-party David. Finally, we show that both of these protocols can be run concurrently and can also support multiple concurrent

buyers. Solidity code for these contracts appears in Appendix B.

Recall, that a party that follows the protocol is *conforming*, and a party that does not is *adversarial*. A party's *principal* is its asset escrowed at a contract during the protocol execution. We use *transfer* and *replace* interchangeably. First we describe the initial setup where Alice and Bob first lock their principals using a hashlock. We will call the party who owns the pre-image to the hashlock (Alice,the initial owner of the option) a *leader* and the counterparty (Bob) a *follower*. This terminology is similar to prior two-party, HTLC-based cross-chain swaps.

Setup

1. Agreement. Alice and Bob agree on a time *startLeader* to start the protocol, usually shortly after this agreement. Alice and Bob agree on the amounts $Asset_A$, $Asset_B$ forming their principals, as well as the number of rounds dT[1], where after $T = startLeader + dT \cdot \Delta$ when Alice's optionality on the BA contract expires. Alice generates a secret A_1 and $H(A_1)$ which is used as *swap_hashlock* to redeem both principals as in a typical two-party swap.

2. Escrow. After the agreement is achieved,

   (a) Within Δ, Alice creates the AB contract, escrowing $Asset_A$, setting $H(A_1)$ as the *swap_hashlock* and sets $T + \Delta$ as the timeout for the AB contract to expire.

   (b) Once Bob sees $Asset_A$ is escrowed with the correct timeout, before Δ elapses, Bob creates the BA contract, escrowing $Asset_B$, setting $H(A_1)$ as the *swap_hashlock* and sets T as the timeout for BA contract to expire.

A swap option between Alice and Bob, where Alice has optionality, is determined by Alice and Bob's addresses, i.e. *sender* and *receiver* in contracts, the *swap_hashlock* used for redemption, and the timeout T for refunds. To make the swap option transferable to Carol, we will need to replace Alice's address with Carol's as *sender* in the AB contract and as *receiver* in the BA contract, and replace the *swap_hashlock* with a new one generated by Carol. In short, we must atomically replace the fields $(AB.sender, BA.receiver, AB.swap\_hashlock, BA.swap\_hashlock)$.

The contracts export the following functions of interest.

- Asset transfer related: the receiver calls *claim*() to withdraw the asset in the contract. This function requires that the pre-image of the *swap_hashlock* is sent to *claim*() before the contract expires. The sender calls $refund()$ to withdraw the asset after the contract expires.

- Option transfer related: *mutateLockLeader*() temporarily freezes the assets and stores a *replace_hashlock* used to replace Alice's role (the replacement between Carol and Alice is also enforced by a hashlock mechanism) and the new *swap_hashlock* if Carol successfully replaces Alice. The call to *replace*() completes the replacement for Carol.

[1]The requirement that $dT \geq 4$ is inherited from the standard two-party swap protocol [40]

For example, consider the AB contract. The *mutateLockLeader*() function freezes the assets, tentatively records the new *sender* as *candidate_sender*, the new *swap_hashlock* for redemption, and records a short-term *replace_hashlock* to be used by *replace*() to finalize the replacement. Later, when *replace*() is called with the secret matching the *replace_hashlock*, these tentative changes become permanent. The *replaceLeader*() function has a timeout. If that timeout expires, the internal *revertLeader*() function unfreezes the assets, discards the tentative changes, and restores the contract's previous state.

The challenge is how to ensure the transfer is atomic on both contracts. If the transfer is not atomic, Bob will be cheated if his principal can be claimed but he cannot claim Alice's principal. To protect Bob, the contract allows Bob to inspect the changes and ensure atomicity by relaying changes on one contract to another.

6.3.2 Transfer Leader Position

Our protocol consists of 3 phases, each of which we describe in turn:

1. *Mutate Lock Phase.* Alice locks the assets on both contracts and tentatively transfers her position to Carol.
2. *Consistency Phase.* Bob makes sure the tentative changes on both contracts are consistent.
3. *Replace/Revert Phase.* Carol replaces Alice's role or she gives up and Alice gets her option back.

I: Mutate Lock Phase Assume Alice and Carol agree on $Asset_C$ for transferring her role to Carol at time *startLeader*, which is the start time the following executions are based on[2]. Carol has secrets C_1, C_2 and generates hashlocks $H(C_1), H(C_2)$. Let $C_{msg} = [H(C_1), H(C_2)]$ be the message Carol sends to specify that she would like to use for [*replace_hashlock*, *swap_hashlock*], respectively.

In this phase, Carol prepares tentative payment $Asset_C$ to Alice to replace Alice's role. Alice locks the AB and BA contracts so that the currently escrowed assets cannot be claimed from the original swap.

1. By $startLeader + \Delta$, Carol creates the CA contract, escrowing $Asset_C$, setting the *swap_hashlock* as $H(C_1)$ for Alice to redeem $Asset_C$, and the timeout to be $startLeader + 9\Delta$. She also sends $C_{msg} = [H(C_1), H(C_2)]$ to Alice.
2. If the previous step succeeds, before Δ elapses, Alice concurrently calls *mutateLockLeader*() on both AB and BA contracts, tentatively setting [*replace_hashlock* $= H(C_1)$, *swap_hashlock* $= H(C_2)$] to both contracts, and setting *candidate_sender* = Carol.*address* on AB and *candidate_receiver* = Carol.*address* on BA contract. If Alice is conforming, *mutateLockLeader*() should be called by $startLeader + 2\Delta$ on both AB and BA.

II: Consistency Phase It is possible that in the worst case, in the *Mutate Lock Phase*, Alice could report inconsistent tentative changes to AB, BA by either mutating one contract but not the other, or by reporting different

[2]The protocol requires that $startLeader \leq T - 9\Delta$. Otherwise, there may not be sufficient time to complete the replacement.

hashlocks to each contract. Because the two contracts cannot communicate, they have no way of knowing what is happening on the other contract. One such example of an attack is shown in Fig. 6.2.

Because of these attacks, in the *Consistency Phase*, Bob is given a period to ensure both AB and BA have the same changes. If one of the contracts is not changed, then Bob forwards the change (by forwarding Alice's signature on the contract she signed) to the other contract. If both contracts are tentatively changed but the changed are different, then Bob can call *contestLeader*() to prove that Alice has lied and reported inconsistent changes signed by Alice.

1. If Bob sees that Alice only calls *mutateLockLeader*() on $AB(BA)$, then within Δ, Bob should call *mutateLockLeader*() on $BA(AB)$ contract and set $[replace\_hashlock = H(C_1), swap\_hashlock = H(C_2)]$ by forwarding Alice's signature.

2. If Bob sees Alice has called *mutateLockLeader*() on both AB and BA , but the changes are different, i.e. either one or more hashlocks are not the same, or the new candidate sender does not correspond to the new candidate receiver, then before a Δ elapses, Bob should call *contestLeader*() on both contracts, forwarding Alice's signature on one contract to the other. In this case, the tentative change will be reverted.

3. If Bob sees Alice has called *claim*() with her secret A_1 on BA to redeem $Asset_B$, and she also successfully called *mutateLockLeader*() on AB, then Bob should call *contestLeader*() on AB before a Δ elapses to revert the tentative change. Bob then calls *claim*() using A_1 on AB to redeem $Asset_A$.

Importantly, Bob is able to contest when either Alice reports inconsistent mutation signatures between contracts or if she preemptively reveals her swap secret. Bob's ability to call *contestLeader*() ensures that he's able to maintain his position in the original trade if Alice deviates from the protocol by reporting inconsistent changes.

III: Replace/Revert Phase After the *Consistency Phase* has ended, in the *Replace/Revert Phase*, the swap either finalizes to one between Carol and Bob or reverts back to the original swap between Alice and Bob. Once Carol sees consistent changes with no successful *contestLeader*() calls by Bob, she calls *replaceLeader*() to make herself the new leader of the swap. Figure 6.1 shows an execution of the protocol where all parties are conforming. If Bob sees Carol only call *replaceLeader*() on one contract, he uses her revealed secret C_1 to finalize the replacement on the other contract.

1. When Carol sees the tentative changes are the same on both AB and BA contract, and Bob is not able to contest anymore on both contracts [3], then before Δ elapses, she calls *replaceLeader*() on both contracts with her secret C_1.

2. If Bob sees Carol only called *replaceLeader*() on one contract $BA(AB)$, then within Δ he calls *replaceLeader*() on $AB(BA)$.

[3]If Bob mutates a contract first, there is no contest window on that contract (He is implicitly approving of the tentative changes). If Alice mutates a contract first, Bob is given 2Δ to dispute that mutation.

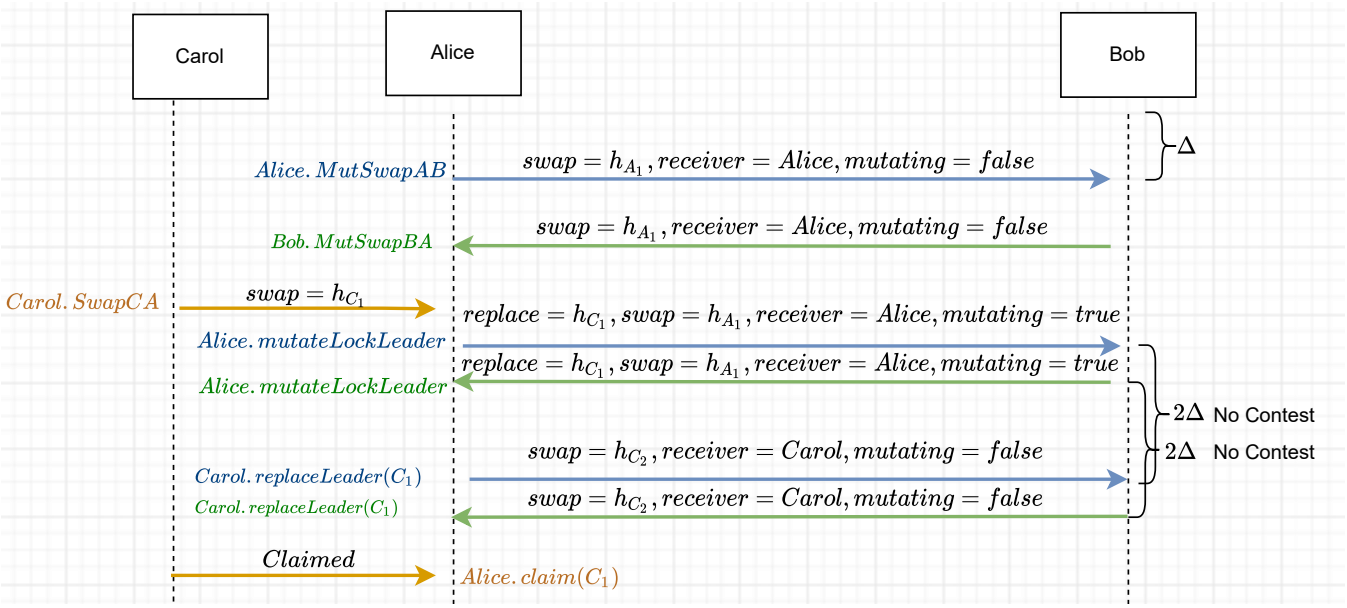

Figure 6.1: In the figure, $x.Function$ means party x calls $Function$ in the contract. E.g. $Alice.MutSwapAB$ means Alice creates the swap contract AB and escrows her assets. The blue arrow depicts contract AB and the green arrow depicts contract BA and the orange one depicts contract CA. The text above each arrow depicts the state of the contract. For example, $swap = h_{A_1}, receiver = Alice, mutating = false$ means $swap\_hashlock = h_{A_1}$, and Alice can claim the asset in the contract if she provides pre-image of h_{A_1}. Here, $replace$ is short for $replace\_hashlock$.

3. If Carol gives up the replacement and does not call $replaceLeader()$ when she can, i.e. $(now - AB(BA).mutation.start\_time > 6 * \Delta)$, then the tentative changes can be reverted by $revertLeader()$ causing the assets to be unfrozen.

4. If Alice sees C_1 which is passed by $replaceLeader()$, then within Δ she calls $claim()$ to claim $Asset_C$ on the CA contract.

The *Replace/Revert Phase* marks the point of the protocol where Alice's position is actually transferred to Carol. Before this phase, Alice has only set up a tentative transfer to Carol. If Carol changes her mind and gives up the replacement, the tentative swap between Carol and Bob can revert back to the original one between Alice and Bob by calling $revertLeader()$. Instead of calling $revertLeader()$ directly, Alice and Bob can alternatively call $mutateLockLeader()$,$refund()$,or $claim()$ at this point in the protocol since these will all automatically call $revertLeader()$. Alice would do this if after an unsuccessful attempt to transfer her swap, she wants to attempt another transfer, reclaim her escrowed funds, or exercise the swap.

Timeouts Timeouts are critical to guarantee the correctness of our protocol. They were omitted in the earlier protocol descriptions for simplicity. Here, we provide the timeouts we set in each step.

The BA contract expires at time T, the last time for the leader to send her secret to redeem the principal. The

AB contract expires at time $T + \Delta$.

Denote the time when *mutateLockLeader*() is called on $AB(BA)$ contract as $AB(BA).mutation.start\_time$. After the mutation starts on any contract, Bob is given time 2Δ to contest Alice's mutation:

$AB(BA).contest.timeout = AB(BA).mutation.start\_time + 2\Delta$.

When Bob is able to contest, and Bob has not called *mutateLockLeader*() himself, Carol cannot call *replaceLeader*(). After the contest period elapses, Carol has 2Δ to call *replaceLeader*(). Here we have $AB(BA).replace.timeout\_Carol = AB(BA).mutation.start\_time + 4\Delta$.

If Carol deviates and only calls *replaceLeader*() on one contract, we allow Bob to call *replaceLeader*() on the another contract. We give Bob 2Δ more than Carol to call *replaceLeader*(). That is, $AB(BA).replace.timeout\_Bob = AB(BA).mutation.start\_time + 6\Delta$.

We see that it takes 6Δ from the start of a *mutateLockLeader*() call to finalize a replacement in the worst case. Thus, $AB(BA).mutation.start\_time + 6\Delta \leq T - \Delta$, which means $AB(BA).mutation.start\_time \leq T - 7\Delta$.

The last person that is able to call *mutateLockLeader*() is Bob. He is given one more Δ than Alice on each contract to call this function in case Alice decides to just call it on one contract. The deadline for Alice to call *mutateLockLeader*() should be $T - 7\Delta$. The deadline for Bob to call *mutateLockLeader*() should be $T - 6\Delta$.

Consider the case when Alice is adversarial by not calling *mutateLockLeader*() on BA but does call it on AB by $T - 7\Delta$. If Bob is compliant, then he calls *mutateLockLeader*() by $T - 6\Delta$ on the BA contract. Because we require $AB.mutation.start\_time \leq T - 7\Delta$, Bob is guaranteed 6Δ to a do a full replacement in the worst case. Conforming Carol's safety is also guaranteed since she can give up the replacement if she finds there is not sufficient time to call *replaceLeader*() then *claim*(). In that case, due to the timeout of the original swap, adversarial Alice loses her opportunity to sell her position.

For CA edge, after Carol escrows $Asset_C$, in the worst case it takes 7Δ to complete the replacement, (Δ to start the mutation and 6Δ to complete the replacement). After the replacement, it takes one Δ for Alice to redeem. Thus, the timeout for CA edge is $startLeader + 9\Delta$.

The reason why we have 2Δ for Bob to contest, and 2Δ more than Carol to call *replaceLeader*() is that the start time of mutation can be staggered by Δ on two contracts, which will be described in detail in proof.

An Optimization

Because of the *contestLeader*() time window, in the worst case, Carol has to wait 2Δ after a *mutateLockLeader*() call to decide whether to release her secret or not. We can speed up the process by adding an *approveLeader*() method. Instead of waiting for 2Δ even though Bob does not contest at all, once Bob sees Alice has made consistent mutations on both edges, Bob can call *approveLeader*() to approve the pending transfer. In other words, by calling *approveLeader*(), Bob gives up the right to contest meaning Carol will not have to wait out the contest window in order to call *replaceLeader*(). In that case, if Bob cooperates, the transfer would be finalized sooner. If Bob doesn't cooperate, the transfer would be finalized later but not blockable if Alice and Carol are conforming.

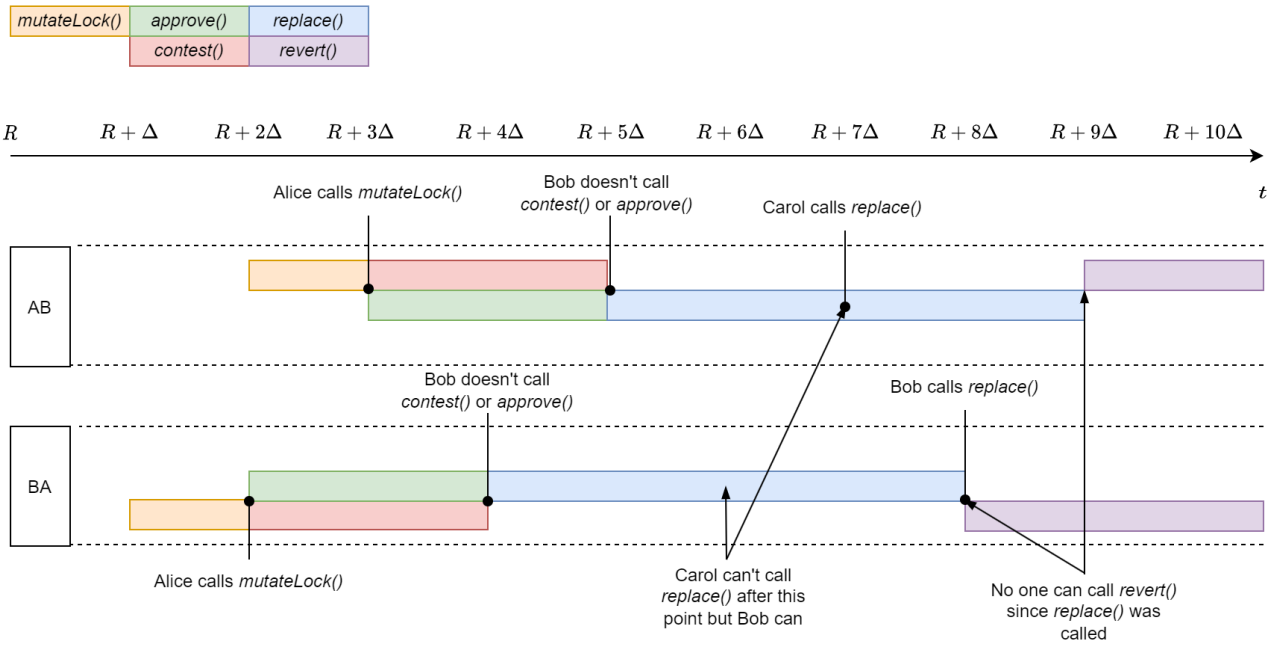

Figure 6.2: A protocol execution demonstrating why Bob needs two more *replaceLeader()* rounds than Carol. If not, then he lacks full Δ to call *replaceLeader()* on BA after Carol calls it on AB. Here we assume Alice has reported consistent signatures in the *Mutate Lock Phase* but has reported them a Δ apart. Colored blocks represent time periods when functions can be called. The *Leader* suffix is excluded from the function calls for simplicity.

This can be accomplished by adding to the $AB(BA)$ contracts an *approved* flag indicating whether Bob has given up his ability to call *contestLeader*().

Phases II-III are the only ones affected.

The following instruction are added to the *Consistency Phase*:

1. Once Bob sees Alice has called *mutateLockLeader*() on both AB and BA with the same change, then before a Δ elapses, Bob should call *approveLeader*() on both AB and BA.

and to the *Replace/Revert Phase*:

1. If Carol sees Bob approved the tentative consistent changes on both AB and BA, then before a Δ elapses, she calls *replaceLeader*() on both AB and BA.

If in addition to conforming to the base protocol in Section 6.3.2, Bob also executes the approve step shown previously, then we call Bob *altruistic*. This just indicates that Bob is willing to speed up the termination of the protocol even if it doesn't necessarily change his financial position.

6.3.3 Transfer Follower Position

We will now describe a protocol that allows Bob to transfer his position in a swap with Alice, to a third-party David. The main difference with this protocol and the previous one, is that Bob cannot limit Alice's optionality. Namely, he cannot unilaterally lock the asset in the BA contract since this would limit the optionality Alice purchased. Effectively Alice must always be able to claim the funds on BA with her secret until the swap itself times out. This weakening of the constraint that we had in the previous protocol, allows for a much simpler protocol. It consists of 2 phases:

1. *Mutate Phase*
2. *Replace/Revert Phase*

I: Mutate Phase

1. Bob and David agree on $Asset_D$ at time *startFollower*, similar to the replacing leader protocol. David creates DB contract, escrowing $Asset_D$, setting the *swap_hashlock* as $H(D_1)$ for Bob to redeem $Asset_D$ and the timeout be $startFollower + 5\Delta$.
2. If the previous step succeeds, and Alice has not revealed A_1 before Δ elapses, Bob concurrently calls *mutateLockFollower*() on AB and *mutateLockFreeFollower*() on BA contracts, tentatively setting [*replace_hashlock* $= H(D_1)$] to both contracts, and setting *candidate_sender* = David.*address* on BA and *candidate_receiver* = David.*address* on AB. We use *mutateLockFreeFollower*() because on the AB contract, this mutation does not lock the asset on AB. This is necessary to ensure Alice does not temporarily lose her optionality.

II: Replace/Revert Lock Phase

1. If David sees Bob successfully call *mutateLockFollower*(), *mutateLockFreeFollower*() on both AB and BA respectively, then within Δ he concurrently executes *replaceFollower*() with D_1 on AB and BA to replace Bob.
2. If Bob sees David call *replaceFollower*() on either AB or BA, within Δ he should call *claim*() on DB to claim $Asset_D$.

From this point on, we will use Protocol 6.3.2 to refer the protocol in Section 6.3.2, Protocol 6.3.2 to the refer to the protocol with extra steps from Section 6.3.2, Protocol 6.3.3 to refer to the protocol in Section 6.3.3, and Protocol 6.3.4 to refer to the protocol in Section 6.3.4.

6.3.4 Handling Multiple Candidates

When Alice decides to tentatively transfer her swap option to Carol, it could be the case that Carol is intending to just grief Alice. In the worst case, Alice will have her option locked for 6Δ by Carol that simply doesn't participate in the *Replace/Revert* Phase of Protocol 6.3.2.

To combat this, Alice might want to initiate several concurrent transfers with multiple buyers at once.

A straightforward solution to this is, after Alice gets her position back from a failed transfer with $Carol_i$, Alice can call *mutateLockLeader*() again to indicate she wants to transfer her option to a new option buyer $Carol_{i+1}$. In the worst case, in this protocol, Alice would spend 6Δ with her option locked for each potential buyer who doesn't comply.

One question then is, can we have Alice potentially transfer her position to multiple buyers at the same time, or with less waiting time than 6Δ in between each new buyer? Although we cannot let Alice transfer her option to multiple buyers simultaneously, we can create enough overlap between each of the potential to reduce Alice's waiting time for each potential buyer. The protocol description below outlines how Alice is able to reduce her waiting time for each potential buyer to 4Δ rather than 6Δ.

The main idea of the protocol is to run a concurrent version of Protocol 6.3.2 for each potential buyer. Alice assigns each potential buyer a ticket (*sequence number*). A *shared counter* is synchronized between the AB and BA contracts. The shared counter assigns an ordering to each of the potential buyers to execute their respective versions of Protocol 6.3.2. In this way, it serves as a first-come first-serve mechanism for Alice to sell her swap option.

We will use $Carol_i$ to denote the buyer who is assigned the i-th sequence number by Alice. On each contract AB and BA, each potential buyer $Carol_i$, will have their own associated state structure $state_i$, similar to Protocol 6.3.2. This keeps track of the state relevant to the 3 different phases from Protocol 6.3.2.

The protocol is defined as follows:

- Initially *counter* on AB and BA contracts is initialized to 0.
- For each potential buyer, Alice assigns a unique sequence number *seq* (starting from 0 and growing by 1 for each assignment) to them, which represents that party's position in the queue for a tentative replacement. If Alice assigns the same sequence number to different parties and sends both of them to the *AB/BA* contracts, the conflict is resolved via the *Consistency Phase* from the original Protocol 6.3.2. It is the same as when Alice deviates by sending inconsistent mutation transactions to both contracts, in which case Bob contests and the tentative transfer is reverted.
- Alice sends mutation transactions as in Protocol 6.3.2. The main difference is that the mutation transactions now include a sequence number *seq* for the candidate replacement. The *mutateLockLeader*(), *contestLeader*() functions all take this sequence number as parameters as part of the mutation. If *counter* == *seq*, the transaction is accepted. Otherwise, the transaction is rejected.

  If Alice is conforming, *counter* should be synchronized on *AB/BA* within Δ. If a tentative transfer is in progress, a new mutation transaction with *seq* == *counter* + 1 will be accepted only if 4Δ has elapsed after the first mutation. A mutate transaction with *seq* == *counter* can serve as mutation transaction for a contest of the current tentative transfer. After the tentative change is reverted, i.e. *revertLeader()* is called, the *counter* is incremented by 1. This design can be optimized by accepting the mutation transactions with larger sequence numbers and store them in a queue for future use. These can then take effect immediately after 4Δ has

elapsed after the current tentative mutation happens and *revertLeader()* has been called. The reason why 4Δ is sufficient interval between the execution of base protocols for two potential buyers is that, after 4Δ, there is only 2Δ left for the mutation to be reverted if the previous potential buyer gives up. The later arriving buyer can use this window to finish its consistency phase and after it ends, its replace phase can start without waiting.

- A mutation transaction can take effect (The corresponding candidate can call *replaceLeader()*) only after the previous mutation transaction has expired, meaning it is reverted and Alice regains the position. Then after that the new candidate can execute Protocol 6.3.2 to replace Alice.

The protocol for transferring the follower position to multiple candidates is similar.

6.3.5 Security Properties

Our leader and follower transfer protocols have many shared properties. Those properties are outlined those properties here.

Recall that assets escrowed in the contracts are called *principal*. The principals involved in the swap option are always Alice's principal and Bob's principal. Here, the original *option provider* is Bob (follower) and the *option owner* is Alice (leader). In the leader transfer protocol and the follower transfer protocol, what is transferred is a position in the swap option. If Alice transfers her position it is to Carol, and Bob to David. We say one party owns a position in an option if they are *option owner* (Alice/Carol): one can release a secret, receiving Bob's principal by relinquishing Alice's, or let the option expire and Alice's principal is refunded to them, or *option provider* (Bob/David): one provides the option owner the right, but not obligation, for the exchange of Alice's principal with Bob's.

From now on, we call a leader/follower who wants to transfer their position as position seller, and the one who wants to replace them as position buyer.

- *Liveness*: In a transfer of positions, if all parties are conforming, then leader/follower transfers their position to a buyer and the position seller gets proper payment from the buyer.

- *Transfer independence*: A compliant position seller (leader/follower) transferring their position to another compliant position buyer, can successfully transfer their position without the cooperation of a third counterparty (follower/leader).

- *Non-blocking transfer with adversarial counterparty*: If a compliant position seller (leader/follower) is transferring their position to another compliant position buyer, a third counterparty (follower/leader) cannot interfere with the transfer.

- *Transfer atomicity*: If a compliant position seller (leader/follower) loses their position to the buyer, then they receive the expected principal from the position buyer.

- *No UNDERWATER for a conforming party (Safety)*: *UNDERWATER* means a party loses their outgoing principal without getting their incoming principal. No *UNDERWATER* guarantees a conforming party's safety since it will never end up with losing their principal without getting principals from others.

Theorem 6.3.1. *Protocol 6.3.2 and 6.3.3 satisfy No UNDERWATER for Alice, Bob, Carol and David:*

- *If Alice is conforming, then if she loses her principal, she either gets Bob's or Carol's principal, or both.*
- *If Bob is conforming, then if he loses his principal, he gets Alice's principal or David's, or both.*
- *If Carol is conforming, then if she loses her principal, she gets Alice's principal or Bob's principal, or both.*
- *If David is conforming, then if he loses his principal, he gets Alice's principal or Bob's principal, or both.*

Proof. See Appendix C.0.2. □

6.3.6 Leader Transfer Properties

We first demonstrate how our leader transfer protocol satisfies the general transfer properties from Section 6.3.5.

Because the detailed proofs of the following results are long and similar to those that we will see for the follower transfer protocol (Section 6.3.7), we omit them here. See Appendix C for further details.

Theorem 6.3.2. *Protocol 6.3.2 satisfies liveness: If Alice, Bob, and Carol are all conforming, then Alice gets Carol's principal, Carol gets Alice's position, and Bob maintains his position.*

Theorem 6.3.3. *Protocol 6.3.2 satisfies transfer independence: Alice can transfer her position to Carol without Bob's participation.*

Theorem 6.3.4. *Protocol 6.3.2 satisfies non-blocking transfer: Alice can transfer her position to Carol even if Bob is adversarial.*

Theorem 6.3.5. *Protocol 6.3.2 satisfies transfer atomicity: If Alice loses her position in the swap, then she can claim Carol's principal.*

In addition to the properties from Section 6.3.5, it is desirable for a leader transfer protocol to have one additional property.

Given two protocols P, P' that satisfy *non-blocking transfer with adversarial counterparty (Bob)*, Bob is called *altruistic* in P' with respect to P if Bob conforming in P' terminates faster than Bob conforming in P in executions where all parties are conforming. Importantly, it is not necessary that Bob should be incentivized to choose to follow P' over P.

When Alice and Carol are conforming, even though adversarial Bob cannot block the transfer, Bob can delay the transfer for a few rounds. The following next property we introduce is a slightly stronger version of *non-blocking* property that is ideal for a leader transfer protocol.

1. *Timely transfer with altruistic Bob*: A protocol P' satisfies this property if there exists a protocol P satisfying *non-blocking transfer with adversarial counterparty (Bob)* where Bob is *altruistic* in P' with respect to P.

Theorem 6.3.6. *Protocol 6.3.2 satisfies timely transfer with altruistic Bob.*

Proof. Since Protocol 6.3.2 is *non-blocking* by Theorem 6.3.4, it is enough to show that Bob is altruistic in Protocol 6.3.2 with respect to Protocol 6.3.2. By assumption all parties are conforming. In both protocols, Carol will have created $MutSwapCA$ by $startLeader + \Delta$ and Alice will have called $mutateLockLeader()$ on AB and BA by $startLeader + 2\Delta$. In Protocol 6.3.2 Bob now calls $approveLeader()$ on $MutSwapAB$ and $MutSwapBA$ by $startLeader + 3\Delta$. Thus the Replace/Revert Phase for Protocol 6.3.2 begins because Bob skips the contest phase. In Protocol 6.3.2, since Alice is compliant, her signatures on $MutSwapAB$ and $MutSwapBA$ are consistent. Thus Bob won't call $contest()$ on $MutSwapAB$ or $MutSwapBA$ during this 2Δ Consistency Phase. Carol waits 2Δ after $startLeader + 2\Delta$. In this case, the Replace/Revert Phase begins at $startLeader + 4\Delta$. Thus Bob is altruistic in Protocol 6.3.2 with respect to Protocol 6.3.2. □

6.3.7 Follower Transfer Properties

We now demonstrate how our follower transfer protocol also satisfies the general transfer properties from Section 6.3.5. Again, the proofs are included in the Appendix C.

Theorem 6.3.7. *Protocol 6.3.3 satisfies liveness: If Alice, Bob, and David are all conforming, and Alice doesn't reveal A_1, then Bob gets David's principal, David gets Bob's position, and Alice maintains her position.*

Theorem 6.3.8. *Protocol 6.3.3 satisfies transfer independence: Bob can transfer his position to David without Alice's participation.*

Theorem 6.3.9. *Protocol 6.3.3 satisfies non-blocking transfer: Bob can transfer his position to David even if Alice is adversarial.*

Note that Alice is not adversarial by choosing to reveal her secret A_1 since this just means she is exercising her option. This is consistent with how she should behave in the original swap protocol.

Theorem 6.3.10. *Protocol 6.3.3 satisfies transfer atomicity: If Bob loses his swap position, then he can claim David's principal.*

On top of the base properties described in Section 6.3.5, any protocol transferring Bob's position should also be *Optionality preserving*: If Alice is compliant, she never loses her ability to exercise the original swap option.

Namely, it is unfair for Alice to temporarily lose her right to use her option without her consent.

Theorem 6.3.11. *Protocol 6.3.3 is optionality preserving.*

Proof. It's enough to show that Alice can always claim the principal on BA, if she reveals her secret A_1 by timeout T. The added functions from Protocol 6.3.3, namely $mutateLockFreeFollower()$,$replaceFollower()$, and $revertFollower()$ only modify the state of $follower\_mutation$. However, any call to $claim()$ on BA doesn't depend on the state of $follower\_mutation$. Namely, any call to $claim()$ is independent of the state of the follower transfer protocol. So its enough that Alice reveal A_1 before T. □

There is nothing preventing both Protocol 6.3.2 and Protocol 6.3.3 from being run concurrently. That is, if Alice wants to transfer her position to Carol, and Bob wants to transfer his position to David, they can both do so simultaneously. This follows immediately from the fact that the parts of the contracts managing the state for each individual protocol are entirely disjoint and cannot affect each other.

6.3.8 Leader Transfer (Multiple Buyers) Properties

Besides the general properties mentioned above, the protocol that handles multiple candidate buyers (Protocol 6.3.4) satisfies the following unique properties.

- *First-come first-serve (FCFS)*: If Alice is conforming (the buyer who tentatively pays to Alice earlier gets a smaller sequence number), then the earlier arriving buyer has priority to replace Alice's position.
- *Starvation freedom*: If Alice is conforming and a conforming buyer, say $Carol_j$, gets a sequence number j, then $Carol_j$ can replace Alice's position if she is conforming and all $Carol_i$ where $i < j$ gives up the replacement.

Theorem 6.3.12. *If Alice is conforming, the counter on both contracts are synchronized with inconsistency for at most* Δ.

Proof. The counter is initialized as 0 and it incremented only after *revertLeader()* is enabled. The *revertLeader()* is enabled only when 6Δ elapses after the start time of *mutateLockLeader()* being called. Since the start time of *mutateLock()* on both contracts are staggered by at most Δ, when the counter is incremented on one contract, it can be incremented on another contract within Δ. □

Theorem 6.3.13. *If Alice is conforming, Protocol 6.3.4 satisfies FCFS.*

Proof. If Alice is conforming, she issues *mutateLockLeader()* transactions with ascending sequence numbers to different potential buyers, in an arrive-earlier-smaller-sequence manner. For every potential buyer, a base transfer protocol (Protocol 6.3.2) is run as normal. Without loss of generality, at time t, suppose on AB contract the counter is i and on BA the counter is $j = i - 1$. On AB, $Carol_i$ has priority. By Theorem 6.3.12, we know the counter on BA will become i within Δ and then $Carol_i$ has priority. The reason why the counter is incremented from j to i is that $Carol_j$ gives up the replacement. Then we see on both contracts $Carol_i$ has priority. It is obvious $Carol_k$ where $k \geq i + 1$ cannot start their replacement unless $Carol_i$ gives up, i.e. the replacement phase ends and the mutation is reverted, since the counter is i now. Therefore the protocol satisfies FCFS. □

Theorem 6.3.14. *If Alice is conforming, Protocol 6.3.4 is starvation free.*

Proof. Similar to Proof 6.3.8, if Alice is conforming and a buyer $Carol_j$, gets a sequence number j, then the base protocol for $Carol_j$ can start underlying others' base protocols. If all $Carol_i$ where $i < j$ gives up the replacement, then the counter is incremented to j and then $Carol_j$ can call *replaceLeader()* to replace Alice. □

Chapter 7

Cross-Chain Limits

The previous chapter (Chapter 6) presented various cross-chain trading problems that can be solved with an assortment of synchronization primitives. This chapter answers fundamental questions about the synchronous cross-chain model. At first, we introduce the *atomic broadcast* abstraction, a useful construct that unifies the functionality of all of the commonly used synchronization tools in cross-chain protocols.

In Section 7.2, we provide a characterization result for the class of swaps that can be solved with hashlocks. Section 7.3 shows a lower bound for an atomic broadcast implemented using path-signatures. Finally, Section 7.4 gives a series of fundamental impossibility results for the synchronous cross-chain model of computation. Importantly, these results lead to a *cross-chain consensus hierarchy* for the cross-chain context, mirroring in some ways the well-known consensus hierarchy from multiprocessor synchronization [39].

7.1 A Formal Cross-chain Model

7.1.1 Weak and Strong Consensus

For a party P, $\mathcal{C}$ is a subset of contracts of size n, and some message $m \in \mathcal{M}$, a protocol $\mathcal{P}$ solves n-broadcast(P, $\mathcal{C}$,m) if it satisfies the following:

- *Agreement*: If at least one party is compliant in $\mathcal{P}$, all contracts decide on either receiving or ignoring the message m.
- *Fairness*: If P is compliant, then all contracts in $\mathcal{C}$ decide to receive the message m.
- *Termination*: $\mathcal{P}$ terminates after a finite number of rounds.

A protocol $\mathcal{P}$ solves *cross-chain consensus* (n-CCC) if it satisfies the following:

- *Agreement*: If at least one party is compliant in $\mathcal{P}$, all contracts decide on one output value.

- *Validity*: The decision value for a contract must have been a registered input to some contract.
- *Liveness*: If all parties are compliant in $\mathcal{P}$ and all contracts start with the same inputs, they decide that input.
- *Termination*: $\mathcal{P}$ terminates after a finite number of rounds.

Note that validity is strictly stronger than liveness, making liveness redundant to state in the above definition. We do so, because liveness does not always follow if we weaken the validity condition.

There are two alternative version of n-CCC we can define by adjusting the validity condition. We say a protocol satisfies *strong validity* if: The decision value for a contract C must have been a registered input value to contract C. Similarly, we say a protocol satisfies *weak validity* if: The decision value for a contract C must be some value in $\{0, 1\}$. We define *strong* n-CCC as n-CCC with *strong validity* and *weak* n-CCC as n-CCC with *weak validity* respectively.

We now show that for any number of parties $\mathbb{P}$ and contracts $\mathbb{C}$ with $n = |\mathbb{C}|$, that n-CCC is equivalent to n-cast$(\cdot,\mathbb{C},\cdot)$. In the following two protocols we assume each contract has a vector consisting of inputs for each party. We refer to this vector as a *registered vector* and the underlying set of values as the *registration set*. Initially all entries in the vector for all contracts are initialized to $\perp$. We assume that each entry of a registration vector is write-once.

Protocol 1. *n-CCC from n-cast*

1. In round 1, all parties P with value $v_P \in \{0, 1\}$ call n-cast$(P,\mathbb{C},v_P)$. No broadcast call from a party P is interpreted as broadcasting input $v_P = 0$.
2. Each contract decides the minimum of the values from its registration set.

In the next protocol, assume party P is the one trying to call n-cast with message m.

Protocol 2. *n-cast from n-CCC*

1. In round 1, P sends m to all contracts individually.
2. In round 2, all other parties send m to any contract that didn't receive m in round 1.
3. If a contract received m from P in round 1, it writes m to all entries in its registration vector.
4. In round 3, run a n-CCC protocol with registration vectors from the first two rounds.

Theorem 7.1.1. *n-CCC is equivalent to n-cast for any number of parties.*

Proof. Let $\mathcal{P}$ be Protocol 1. Termination is trivially satisfied since n-cast satisfies termination and $\mathcal{P}$ runs in a finite number of rounds. By n-cast agreement, each contract receives the same registration vector. Because the decision rule is deterministic, all contracts reach agreement. If all contracts decide 0 then it must have been the case that some party called n-cast with 0. Similarly if all contracts decide 1 then it must be the case that all parties called

n-cast with 1. In either case, the decision value was contained in the registration set of all contracts so $\mathcal{P}$ satisfies validity.

Let $\tilde{\mathcal{P}}$ be Protocol 2. Again, termination of $\tilde{\mathcal{P}}$ follows from n-CCC termination and the fact that $\tilde{\mathcal{P}}$ runs in a finite number of rounds. Now suppose that P sends m to all contracts in round 1 resulting in all contracts having the all m registration vector at the end of round 2. By validity for n-CCC, in round 3, $\tilde{\mathcal{P}}$ must decide m if P remains compliant. Thus $\tilde{\mathcal{P}}$ satisfies fairness. Agreement follows immediately from agreement for n-CCC. □

7.2 Hashlock Limitations

Complex swaps can be captured as a directed graph (digraph) where each node is a party, and each arc is a chain labeled with the asset to be traded. As noted earlier, *hashlocks* are a simple escrow mechanism originally proposed for simple swaps. It is natural to ask which swap digraphs can be solved with hashlocks alone, and which require additional cryptographic gadgets such as signatures. The answer will depend on the graph-theoretic properties of the swap digraph.

Given a set X, 2^X denotes its power set, and X^r denotes its *r-fold* Cartesian product $\times_{i=1}^{r-1} X$. Given a simplex σ, $\sigma|_{\mathcal{C}}$ refers to the face of σ with vertices labeled only from the set $\mathcal{C} \subseteq \mathbb{C}$. Given two simplices σ, τ, we say they are *X-indistinguishable*, written $\sigma \sim_X \tau$, if $\sigma, \tau \in X$ and $\sigma \cap \tau \neq \emptyset$. Two simplices σ_1, σ_n are called *X-path-connected*, denoted $\sigma_1 \leftrightsquigarrow_X \sigma_n$ if there is a sequence of simplices $(\sigma_1, \sigma_2, ..., \sigma_n)$ where for $1 \leq i \leq n-1$, $\sigma_i \sim_X \sigma_{i+1}$. Clearly $\leftrightsquigarrow_X$ is an equivalence relation. The equivalence classes are called *path components*.

Directed Graph Tasks A large class of tasks can be constructed from directed graphs. A *directed graph* is some $G = (V, E)$ where $V = \mathbb{P}$ and $E \subset \mathbb{P} \times \mathbb{P}$. Additionally, we will use the notation $V(G) = V, E(G) = E$. There is at most one edge between each pair of parties, no self-edges, and we associate a unique contract with each $e \in E$, which we refer to as $C(e)$. For convenience, we will refer to e and $C(e)$ interchangeably. For $e \in E$, use $sender(e)$ to mean $v \in V$ such that $e = (v, \cdot)$ and $receiver(e)$ to mean $v \in V$ such that $e = (\cdot, v)$. For, $v \in V$, let $in(v) = \{e \in E : e = (\cdot, v)\}$, $out(v) = \{e \in E : e = (v, \cdot)\}$, and $local(v) = in(v) \cup out(v)$. Given a graph $G = (V, E)$ and some $v \in V$, we will use G_v to be the directed graph $G_v = (V \backslash \{v\}, E')$ where $E' = \{e \in E : e \notin local(v)\}$.

Given a directed graph $G = (V, E)$, a *directed graph task* is defined as follows. The input complex $\mathcal{I}$ is a single simplex. For each $e \in E$, the input simplex $\mathcal{I}$ reflects the initial owner of some asset being managed by $C(e)$, that is $\mathcal{I} = \{(C(e), sender(e)) : e \in E\}$. The output complex is defined as $\mathcal{O} = \{(C(e), v) : e \in E, v \in \{sender(e), receiver(e)\}\}$.

The utility for party P, $u_P : \mathcal{O} \to \mathbb{R}$ is constrained as follows:

$$u_P(\sigma) \in \begin{cases} \{0\} & \sigma|_{local(P)} = \mathcal{I}|_{local(P)} \\ (0, \infty) & \sigma|_{out(P)} = \mathcal{I}|_{out(P)}, \sigma|_{in(P)} \neq \mathcal{I}|_{in(P)} \\ (-\infty, 0) & \sigma|_{out(P)} \backslash \mathcal{I}|_{out(P)} \neq \emptyset, \sigma|_{in(P)} \cap \mathcal{I}|_{in(P)} \neq \emptyset \\ (0, \infty) & \sigma|_{out(P)} \backslash \mathcal{I}|_{out(P)} \neq \emptyset, \sigma|_{in(P)} \cap \mathcal{I}|_{in(P)} = \emptyset \end{cases}$$

Lemma 7.2.1. *For tasks in SC_k, $\cap_{P \in \mathbb{P}} \Delta_P^+(\mathcal{I}) = \{(C(e), receiver(e)) : e \in E\}$*

Proof. Let $\sigma \in \cap_{P \in \mathbb{P}} \Delta_P^+(\mathcal{I})$ and fix some edge $e \in E$. Suppose $(C(e), sender(e)) \in \sigma$. Let $P = receiver(e)$ and suppose for some $e' \in out(P)$ that $(C(e'), receiver(e')) \in \sigma$. But $(C(e), sender(e)) \in \sigma$ this means $u_P(\sigma) < 0$ so $\sigma \notin \Delta_P^+(\mathcal{I})$ so $\sigma \notin \cap_{P \in \mathbb{P}} \Delta_P^+(\mathcal{I})$. Hence for all $e' \in out(P)$ we must have $(C(e'), sender(e')) \in \sigma$. Let Q be some other party $Q \neq P$. By strong connectivity there is some path $p = (e_1, ..., e_r)$ from P to Q. Repeatedly applying the above argument for each edge e_i in p implies that for all $e' \in out(Q)$, $(C(e'), sender(e')) \in \sigma$. Because Q was arbitrary, this means $\sigma = \mathcal{I}$. But $\mathcal{I} \notin \Delta_P^+(\mathcal{I})$ for any P, giving a contradiction. □

Collectively then $(\mathcal{I}, \mathcal{O}, \{u_P\}_{P \in \mathbb{P}})$ define a *directed graph task*. Let DG_k denote the collection of all directed graph tasks on k contracts. Additionally, let SC_k be the directed graph tasks on k contracts for which the underlying directed graph is *strongly connected*. A graph $G = (V, E)$ is called *complete* if there is exactly one edge between each pair of distinct parties. We use K_n to denote the *complete graph task* on n parties. Clearly, $K_n \subsetneq SC_{n(n-1)} \subsetneq DG_{n(n-1)}$.

Broadcast Protocols *Broadcast protocols* take place in two phases, each of which consists of synchronous rounds. The first phase is called an *escrow phase*, where parties surrender their rights to manage the ownership of an asset to a contract. The second phase is called a *reveal phase*, where parties make calls to broadcast primitives to reveal private values to contracts. Because we are interested in lower bound and impossibility results, we will assume there is a hard synchronization point between these two phases.

The *escrow phase* works as follows. Each contract C has an *escrow bit* (initially 0), written C_{bit} indicating whether any broadcast values can affect the state of the contract. Each contract C has a pre-specified subset of parties, written C_{escrow}, who can toggle the escrow bit to 1. In this phase, parties can only make *toggle* calls of the form $toggle(P, C)$ where P is the sending party and C is the receiving contract. If $P \in C_{escrow}$ then the escrow bit is set to 1, otherwise nothing happens. For directed graph tasks, for an edge $e \in E$, we will assume $C(e)_{escrow} = \{sender(e)\}$, reflecting the transfer of ownership implied by the graph structure. An escrow phase then consists of several rounds of the following:

1. All parties take snapshots of all contracts' states.

2. Parties make a finite number of *toggle* calls to contracts.

```
primitive n−cast {
  bool calledInRound = false;
  public function send(v,Q) {
    if  (|Q| == n && !calledInRound) {
      for(P in Q){
        send(v,P)
      }
      calledInRound = true;
    }
  }

    private function reset(){
        calledInRound = false;
    }
}
```

Figure 7.1: n-cast implementation

The *reveal* phase works as follows. Each party $P \in \mathbb{P}$ starts with a set of private values $V_P = \{v_P^1, ..., v_P^\ell\}$ which initially partition a set V, that is $V = \cup_{P\in\mathbb{P}} V_P$ and V_P, V_Q are disjoint for $P \neq Q$. It is not necessary that $V_P \neq \emptyset$ for all P, but we assume it is true for at least one $P \in \mathbb{P}$. The set of parties for which $V_P \neq \emptyset$ are called *leaders*, and we write $\mathcal{L} \subseteq \mathbb{P}$ to denote the set of leaders. Each V_P can grow in response to a party taking a global snapshot at the beginning of a round. Compliant parties broadcast their entire V_P to all contracts.

A n-broadcast is a call of the form $bcast_n(v, \mathcal{Q})$ that atomically reveals $v \in V$ to all contracts in $\mathcal{Q} \subseteq \mathbb{C}$ if $|\mathcal{Q}| = n$. The implementation is defined in Figure 7.1.

A (k, n)-broadcast with $1 \leq k < n$ is a synchronization primitive with the implementation given in Figure 7.2.

In Figure 7.1 and Figure 7.2, all functions are assumed atomic. Parties are only allowed to call public methods, private methods are called instantaneously at the beginning of each round, before any party calls.

Contracts $C \in \mathbb{C}$ store a *value set* $V_C \subseteq V$ that can grow via broadcast calls. The state of each contract is given by a *value history*. Formally, a *value history* is a sequence $\vec{h} = (p_1, ..., p_r) \in (2^V)^r$ where if $1 \leq i \leq r-1$, then $p_i \subseteq p_{i+1}$. We adopt the convention that a superscript denotes number of rounds for a value history.

Let $\vec{h}^r_\emptyset = (p_1, ..., p_r)$ where all $p_i = \emptyset$ and $\vec{h}^r_\mathcal{Q} = (p_0, ..., p_r)$ where each $V_P \subseteq p_i$ for all $P \in Q \subseteq \mathbb{P}$. Also, let $\sigma^r_\emptyset$ be the simplex with all vertices in state $\vec{h}^r_\emptyset$ and σ^r_Q be the simplex with all vertices in the state $\vec{h}^r_\mathcal{Q}$.

We now explore when broadcast protocols can solve strongly connected tasks.

It turns out that the K_3 task is the smallest directed graph task for which hashlocks fail. This was perhaps the motivation for the path signature protocol needed in [40].

We now identify the much larger class of problems for which (k, n)-broadcast fails and show K_3 is a special case. If we have a strongly connected graph $G = (V, E)$ where $|V| \geq 3$ and G_v is cyclic for all $v \in V$, then we say G is *strongly cyclic*. Let $C_{1,n}$ be the set of all strongly cyclic graphs on n contracts.

```
primitive (k,n)−cast {
  k−cast cast0;
  n−cast cast1;
  int castNumber = k;
  bool calledInRound = false;
  public function send(v,Q) {
    if (|Q| == castNumber && !calledInRound) {
      if(castNumber == k){
        cast0.send(v,Q)
        castNumber = n;
      }else{
        cast1.send(v,Q)
      }
      calledInRound = true;
    }
  }

    private function reset(){
        calledInRound = false;
    }

}
```

Figure 7.2: (k, n)-cast implementation

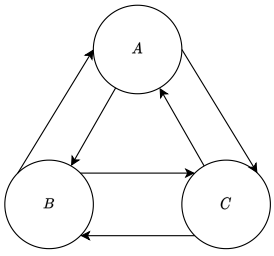

Figure 7.3: K_3: Smallest graph for which (k, n)-broadcast fails. In this case $n = 6$, $1 \leq k < 6$.

Lemma 7.2.2. *For tasks in SC_k, $\cap_{P\in\mathbb{P}}\Delta_P^+(\mathcal{I}) = \{(C(e), receiver(e)) : e \in E\}$*

Proof. Let $\sigma \in \cap_{P\in\mathbb{P}}\Delta_P^+(\mathcal{I})$ and fix some edge $e \in E$. Suppose $(C(e), sender(e)) \in \sigma$. Let $P = receiver(e)$ and suppose for some $e' \in out(P)$ that $(C(e'), receiver(e')) \in \sigma$. But $(C(e), sender(e)) \in \sigma$ this means $u_P(\sigma) < 0$ so $\sigma \notin \Delta_P^+(\mathcal{I})$ so $\sigma \notin \cap_{P\in\mathbb{P}}\Delta_P^+(\mathcal{I})$. Hence for all $e' \in out(P)$ we must have $(C(e'), sender(e')) \in \sigma$. Let Q be some other party $Q \neq P$. By strong connectivity there is some path $p = (e_1, ..., e_r)$ from P to Q. Repeatedly applying the above argument for each edge e_i in p implies that for all $e' \in out(Q)$, $(C(e'), sender(e')) \in \sigma$. Because Q was arbitrary, this means $\sigma = \mathcal{I}$. But $\mathcal{I} \notin \Delta_P^+(\mathcal{I})$ for any P, giving a contradiction. □

Lemma 7.2.3. *Given any cycle in a graph $G \in SC_n$, for a $(n-1, n)$-broadcast protocol, if all parties in the cycle are compliant, then all incoming edges e to any party in the cycle must decide $receiver(e)$.*

Proof. Let $\mathcal{C}$ be a cycle in G where we label the parties $(P_1, ..., P_m)$, ordered how their respective edges appear in $\mathcal{C}$. Let $\mathcal{Q}$ be the set of leaders outside of C but in G.

Suppose $\mathcal{Q} = \emptyset$. Consider some $\sigma \in \cap_{P_i\in\mathcal{C}}\mathcal{P}_{P_i}(\mathcal{I})$. Since $\mathcal{Q} = \emptyset$, each contract in σ has the same state as it would in the all compliant execution. Namely, $\sigma \in \cap_{P\in\mathbb{P}}\mathcal{P}_P(\mathcal{I})$, so by definition of solvability, all contracts in σ must decide $receiver(e)$ by Lemma 7.2.2. In particular, all incoming edges to parties in $\mathcal{C}$ must have their contracts decide $receiever(e)$ in σ.

Now say $\mathcal{Q} \neq \emptyset$. Let $Q \in \mathcal{Q}$ where by assumption $Q \in \mathcal{L}$ so $V_P \neq \emptyset$. Suppose the task is solved in $r \geq 2$ rounds. We will now show that the all compliant simplex $\sigma^r_{\mathcal{Q}\cup\mathcal{C}} \in \cap_{P\in Q\cup\mathcal{C}}\mathcal{P}_P(\mathcal{I})$ is path-connected to $\sigma^r_{(\mathcal{Q}\cup\mathcal{C})\setminus Q} \in \cap_{P\in(\mathcal{Q}\cup\mathcal{C})\setminus Q}\mathcal{P}_P(\mathcal{I})$ while all parties in C remain compliant.

Say wlog that each edge in e has contract state $h^r_{\mathcal{Q}\cup\mathcal{C}} = (V_1, ..., V_1)$ where each $V_P \subseteq V_1$ for all $P \in \mathcal{Q}\cup\mathcal{C}$. In round 1, say for concreteness that Q reveals V_Q to all contracts but some incoming edge $e = (P_1, P_2)$. Thus after r rounds, e has contract state $(V_2, V_1, ..., V_1)$. Note that (P_2, P_3) decides on P_3 since $h^r_{\mathcal{Q}\cup\mathcal{C}}$ is its contract state in the all compliant execution. If e doesn't decide on P_2 with contract state $(V_2, V_1, ..., V_1)$, then this is a contradiction since P_2 is assumed compliant. Hence e must decide $receiver(e) = P_2$ for contract state $(V_2, V_1, ..., V_1)$.

This is indistinguishable for e to the execution where Q reveals V_Q in round 2 on all edges incoming to parties in $\mathcal{C}$ which gives $(V_2, V_1, ..., V_1)$ for all their respective contract states. Suppose that for one such $e' \neq e$ where $receiver(e') \neq P_2$, $C(e')$ does not decide $receiver(e')$. Now let $\tilde{P} = receiver(e')$. Since $\tilde{P}$ is a vertex in the cycle, construct the sequence of vertices from $\tilde{P}$ to P_2 of the form $(v_1, ..., v_s)$ where $v_1 = \tilde{P}, v_s = P_2$ and each two adjacent vertices form an edge in $\mathcal{C}$. Note that $e = (v_{s-1}, v_s)$ and we know that $C(e)$ decides P_2. If $v_{s-1} = \tilde{P}$ then $C(e')$ must decide $\tilde{P}$ since $\tilde{P}$ is compliant. Otherwise, (v_{s-2}, v_{s-1}) must decide v_{s-1} since v_{s-1} is assumed compliant. Repeatedly applying this remark resolves to the previous case, thus $C(e')$ must decide $receiver(e') = \tilde{P}$.

In general this two step procedure can be iterated r times by party Q, ensuring that any incoming edge e to a party in $\mathcal{C}$ must decide on $receiver(e)$ for any contract state of the form $(V_2, ..., V_2, V_1, ..., V_1)$ where V_2 is repeated l times. In particular, V_2 can be repeated r times, resulting in the global state $\sigma^r_{(\mathcal{Q}\cup\mathcal{C})\setminus Q}$.

But because Q was arbitrary, we can simply redefine $\mathcal{Q}$ to exclude Q and repeat the same argument until $\mathcal{Q} = \emptyset$ which was the original case considered. □

Lemma 7.2.4. *For any broadcast protocol that solves a task in SC_k, $k \geq 2$, in the all compliant execution all contracts must have their escrow bits successfully toggled.*

Proof. Suppose not. Let $e \in E$ in the underlying graph for which $C(e)$ didn't have its escrow bit toggled in the escrow phase. Let r be the fewest number of rounds for which the broadcast protocol solves the task. Run the all compliant execution for r rounds and let $\tau \in \cap_{P \in \mathbb{P}} \mathcal{P}_P^r(\sigma)$. By assumption then $\delta((C(e), \cdot)) = (((C(e)), receiver(e))$. However, since the escrow bit on $C(e)$ was never toggled, after r rounds $C(e)$ will have the value history $\vec{h}_\emptyset^r$ independent of all V_P. But $(C(e), \vec{h}_\emptyset^r) \in \sigma_\emptyset^r$ so by definition of solvability $\delta((C(e), \vec{h}_\emptyset^r)) = (C(e), sender(e))$, a contradiction. □

Theorem 7.2.5. *There is no (k, n)-broadcast protocol that solves any $G \in C_{1,n}$ for $1 \leq k < n$.*

Proof. It is enough to consider $k = n - 1$. Let r be the earliest round in the escrow phase for which some edge e has its escrow bit toggled and let $v = sender(e)$. By Lemma 7.2.4 we know such a round exists.

Since G is strongly cyclic, we know G_v contains some cycle with vertices V. Let $\mathcal{C}_v$ be the smallest subgraph of G_v containing V and edges between vertices in V, namely $V(\mathcal{C}_v) = V$ and $E(\mathcal{C}_v) = \{e \in E(G_v) : e = (v, w), v, w \in V(\mathcal{C}_v)\}$.

Let $\tilde{\mathcal{C}}_v$ be the subgraph of G_v defined by $V(\tilde{\mathcal{C}}_v) = V(G_v) \backslash V(C_v)$ and $E(\tilde{\mathcal{C}}_v) = \{e \in E(G_v) : e = (v, w), v, w \in V(\tilde{C}_v)\}$.

We will now show that if $p = (v_1, ..., v_s)$ is any path in G with $v_1 = v$ and $v_s \in V(C_v)$, then the parties in $V(C_v)$ are sufficient to decide (v_1, v_2) if they run for enough rounds. By Lemma 7.2.3, since (v_{s-1}, v_s) is an incoming to the cycle $\mathcal{C}$, we know that after some number of rounds l that contract $C((v_{s-1}, v_s))$ will decide v_s if all parties in $V(\mathcal{C}_v)$ are compliant in the reveal phase. If $v_{s-1} = v$ then we have the result. Otherwise, consider some compliant v_i in the path for which (v_i, v_{i+1}) has the desired property, where $2 \leq i \leq s - 1$. Because v_i is compliant and (v_i, v_{i+1}) decided v_{i+1}, to ensure our safety condition for v_i, it must be the case that all of its incoming edges decide v_i. In particular (v_{i-1}, v_i) must decide v_i.

Consider the following execution in which v is compliant: In round r in the escrow phase, v escrows the outgoing edge (v_1, v_2). All other parties deviate and do not escrow at all during the escrow phase. During the reveal phase, all other parties simulate their all compliant execution on (v_1, v_2). By the property just shown of the path p, we know that (v_1, v_2) decides v_2. Additionally, since all other edges $e \neq (v_1, v_2)$ never had their escrow bits toggled, they all decide $sender(e)$. In particular any e of the form $e = (\cdot, v)$ must decide $sender(e) \neq v$, violating safety for v since it was assumed compliant.

Thus if can show that we can always construct a path $p = (v_1, ..., v_s)$ as described from v to $\mathcal{C}_v$, then we are done. Pick a $w \in V(\mathcal{C}_v)$. By strong connectivity there is such a path $p = (v_1, ..., v_s)$ where $v_1 = v$ and $v_s = w$ so we are done. □

Corollary 7.2.5.1. *There is no (k, n)-broadcast protocol that solves K_n for $1 \leq k < n, n \geq 3$.*

Proof. Noticing that $K_n \in C_{1,m}$ where $m = n(n-1)$ immediately yields the result. □

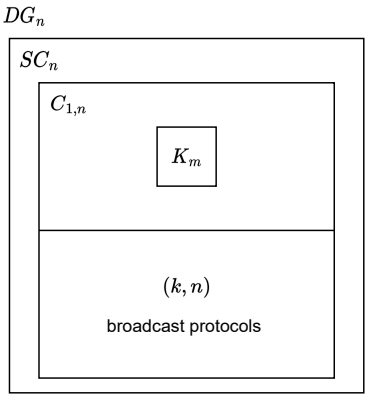

Figure 7.4: Partition of graphs in SC_n that can be solved with (k,n)-broadcast where $1 \leq k < n$ and $n = m(m-1)$.

Corollary 7.2.5.2. *There is no $(k,6)$-broadcast protocol that solves K_3 when $1 \leq k < n$.*

Proof. This is a special case of Corollary 7.2.5.1. □

Corollary 7.2.5.3. *No hashlock protocol solves the K_3 task.*

Proof. Hashlocks are defined as $(1,6)$-broadcast so by Corollary 7.2.5.2 the result follows immediately. □

The results concerning SC_n are summarized in Figure 7.4.

7.3 Path-signature Lower Bound

This section shows that using path signatures to implement k-broadcast with n parties requires at least n synchronous rounds, even in the best case.

Assume $\mathbb{P} = \{A, P_1, ..., P_{n-1}\}$ is the set of parties where A is a distinguished party with private input value v. A *path signature* is a string of the form $s = p_1 p_1 ... p_k v$ where each $p_i \in \mathbb{P} \backslash \{A\}$ and for no two $i \neq j$ does $p_i = p_j$. Given a path signature $s = p_1 p_1 ... p_k v$, denote the length by $|s| = k + 1$. We consider v as a path signature of length 1 and $\emptyset$ as a path signature of length 0. For a path signature s, let $Set(s)$ denote the set of underlying parties in the path signature, where if v is in the path signature, then $A \in Set(s)$. For $k \in \mathbb{N}_0$, and any $Q \subseteq \mathbb{P}$ let $\mathcal{PS}_k(Q) = \{s : s \text{ is a path signature, } |s| = k, \text{ and } s \text{ only contains characters in } Q\}$. Also, for $Q \subseteq \mathbb{P}$, let $\mathcal{PS}(Q) = \cup_{k \in \mathbb{N}_0} \mathcal{PS}_k(Q)$.

If s, s' are two path signatures where $|s| \leq |s'|$ and we can write $s' = p_1 ... p_k s$, then we call s' an *extension* of s, and say s' *extends* s.

Path signatures are implemented with *digital signatures*. Any time a compliant party P sees a path signature $s \in \mathcal{PS}_k(\mathbb{P})$ on a contract, if $P \notin Set(s)$, then they create the path signature $s' = Ps$ and send it to all contracts.

Each contract maintains a state consisting of all path signatures it has seen. Formally, contracts $C \in \mathbb{C}$ store a *signature set* $S_C \in 2^{\mathcal{PS}(\mathbb{P})}$. Their state consists of a *signature history* which is a sequence $\vec{h} = (p_1, ..., p_r) \in (2^{\mathcal{PS}(\mathbb{P})})^r$ where if $1 \leq i \leq r-1$, then $p_i \subseteq p_{i+1}$.

For non-empty $Q \subseteq \mathbb{P}$, let σ_Q^r be the simplex resulting from all parties in Q being compliant for r rounds. Let σ_m^r for $m \geq 2$ be a shorthand for σ_Q^r where $Q = \{A, P_1, .., P_{m-1}\}$. Let σ_0 be the simplex where every vertex outputs 0 and σ_1 where every vertex outputs 1.

In the combinatorial language, the k-broadcast task is given by $\mathcal{I} = \sigma = \sigma_0$, $\Delta_P(\sigma) = \sigma_0 \cup \sigma_1$, and $\Delta_P^+(\sigma) = \sigma_1$ for all $P \in \mathbb{P}$. It follows that $\delta(\cup_{P\in\mathbb{P}}\mathcal{P}_P(\sigma)) \in \sigma_0 \cup \sigma_1$. Additionally for $r \geq 1$, $\sigma_\emptyset^r \in \cup_{P\in\mathbb{P}}\mathcal{P}_P^r(\sigma)$ since all parties $P_1, ..., P_{n-1}$ could simply remain compliant while A doesn't reveal v. Similarly, $\sigma_{\mathbb{P}}^r \in \cup_{P\in\mathbb{P}}\mathcal{P}_P^r(\sigma)$ by definition.

Theorem 7.3.1. *For any protocol $\mathcal{P}$ that solves the k-broadcast task in r rounds, $\sigma_{\mathbb{P}}^r \not\rightsquigarrow_X \sigma_\emptyset^r$ where X is the 1-compliant complex.*

Proof. Suppose not. So $\sigma_{\mathbb{P}}^r \leftrightsquigarrow_X \sigma_\emptyset^r$ where X is the 1-compliant complex. Then there is a sequence of simplices $(\tau_1, ..., \tau_s)$ where each τ_i is in the 1-compliant complex, for $1 \leq i \leq s-1$ $\tau_i \sim \tau_{i+1}$, and $\tau_1 = \sigma_{\mathbb{P}}^r, \tau_s = \sigma_\emptyset^r$. By definition of the task, $\delta(\tau_1) = \delta(\sigma_{\mathbb{P}}^r) = \sigma_1$ and $\delta(\tau_s) = \delta(\sigma_\emptyset^r) = \sigma_0$.

We will now show that for $1 \leq i \leq s-1$, if $\delta(\tau_i) = \sigma_1$ then $\delta(\tau_{i+1}) = \sigma_1$. Since $\tau_i \sim_X \tau_{i+1}$, there is some contract C in the intersection $\tau_i \cap \tau_{i+1}$ for which $\tau_i, \tau i + 1$ are X-indistinguishable. Since $(C, \cdot) \in \tau_i$ by assumption $\delta((C, \cdot)) = 1$. Because τ_{i+1} is 1-compliant we know $\delta(\tau_{i+1}) = \sigma_0$ or $\delta(\tau_{i+1}) = \sigma_1$. If $\delta(\tau_{i+1}) = \sigma_1$ we are done, so assume otherwise. Clearly this is a contradiction since $\delta((C, \cdot)) = 1$ and $(C, \cdot) \in \tau_{i+1}$.

Applying this $s-1$ times implies $\delta(\sigma_\emptyset^r) = \delta(\tau_s) = \sigma_1$ but this violates the requirement that $\delta(\sigma_\emptyset^r) = \sigma_0$.

□

Theorem 7.3.2. *For n parties and 2 contracts, $\sigma_m^k \leftrightsquigarrow_X \sigma_{m-1}^k$, where $2 \leq m \leq n$,$1 \leq k < n$, and X is the 1-compliant complex.*

Lemma 7.3.3. *For n parties and 2 contracts, $\sigma_2^k \leftrightsquigarrow_X \sigma_\emptyset^k$ where and $1 \leq k < n$, and X is the 1-compliant complex.*

Theorem 7.3.4. *For n parties, there is no path signature protocol that solves 2-cast in fewer than n rounds.*

Proof. Suppose there was such a protocol $\mathcal{P}$. Consider running $\mathcal{P}$ for $1 \leq k < n$ rounds. Apply Theorem 7.3.2, $n-2$ times to get $\sigma_i^k \leftrightsquigarrow_X \sigma_{i+1}^k$ where $2 \leq i \leq n$ with X is the 1-compliant complex. By transitivity of $\leftrightsquigarrow_X$ this then implies that $\sigma_{\mathbb{P}}^k = \sigma_n^k \leftrightsquigarrow_X \sigma_2^k$. By Lemma 7.3.3 we get $\sigma_2^k \leftrightsquigarrow_X \sigma_\emptyset^k$. Again, applying transitivity of $\leftrightsquigarrow_X$ gives $\sigma_{\mathbb{P}}^k \leftrightsquigarrow_X \sigma_\emptyset^k$. But this violates Theorem 7.3.1. □

Corollary 7.3.4.1. *For n parties, there is no path signature protocol that solves k-cast on k contracts in fewer than n rounds.*

7.4 A Cross-Chain Consensus Hierarchy

7.4.1 Two Parties

Here we give an explicit combinatorial construction to show that 1-cast isn't enough to solve weak 2-CCC for two parties.

Let σ_x^k be the simplex where all contracts have state x and all parties have been compliant for $k \geq 0$ rounds.

An *alternating path* is a sequence of simplices $\tau = (\sigma_0, ..., \sigma_{n+1})$ with $n \geq 1$ satisfying the following properties:

- Each successive pair of simplices overlap on exactly one contract whose identity alternates for each pair.
- $\sigma_0 = \sigma_x^k$ and $\sigma_{n+1} = \sigma_y^k$ for some $k \geq 0$, $x \neq y$
- For $1 \leq i \leq n$, exactly one party is compliant in σ_i and each contract has a different state.
- For $1 \leq i \leq n-1$, if one party is compliant in σ_i, then the other party is compliant in σ_{i+1},

Informally, an alternating path, path connects two all-compliant simplices with distinct inputs.

Lemma 7.4.1. *Suppose that $\sigma_x^k \leftrightsquigarrow \sigma_y^k$ for some $k \geq 0$ via an alternating path. Then we can always construct an alternating path so that $\sigma_x^{k+1} \leftrightsquigarrow \sigma_y^{k+1}$.*

Proof. First, we show the existence of such an alternating path for $k = 0$. Let σ be the simplex where A has input x on both C_1, C_2 and B has input x, y on C_1, C_2 respectively. Note that $\sigma \sim \sigma_x^0$ to C_1 and $\sigma \sim \sigma_y^0$ to C_2. Additionally, AB are compliant in σ_x^0, σ_y^0 and A is the only compliant party in σ. Hence $(\sigma_x^0, \sigma, \sigma_y^0)$ is an alternating path.

Now assume the conclusion is true for some arbitrary $k \geq 0$. Let $\tau = (\sigma_0, ..., \sigma_{n+1})$ be the alternating path from round k. We will construct a new alternating path $\tau' = (\sigma'_0, ..., .\sigma'_{n+3})$ for round $k+1$. Assume WLOG that party A was compliant in σ_1 and σ_0, σ_1 overlap on contract C_1.

Let $\sigma'_0 = \sigma_x^{k+1}$ and $\sigma'_{n+3} = \sigma_y^{k+1}$. For $1 \leq i \leq n+2$, where i is odd, for σ'_i, run compliant B starting from σ_{i-1}. For $1 \leq i \leq n+2$, where i is even, for σ'_i, run compliant A starting from σ_{i-1}.

Additionally, when $1 \leq i \leq n+2$, where i is odd, have A simulate its σ'_{i+1} calls on C_1 and σ'_{i-1} calls on C_2. Similarly, when $1 \leq i \leq n+2$, where i is even, have B simulate its σ'_{i+1} calls on C_2 and σ'_{i-1} calls on C_1.

By construction, for $0 \leq i \leq n+2$, $\sigma_i \sim \sigma_{i+1}$ to C_2 when i is even and C_1 when i is odd. A, B are both compliant in σ'_0, σ'_{n+3}. Additionally, for $1 \leq i \leq n+2$, for odd i, B is the only compliant party, and for even i, A is the only compliant party. Thus, τ' is an alternating path connecting $\sigma_x^{k+1} \leftrightsquigarrow \sigma_y^{k+1}$.

□

Theorem 7.4.2. *Two parties cannot solve weak 2-CCC using 1-cast.*

Proof. Consider σ_1^k and σ_0^k after some $k \geq 0$ rounds. By Lemma 7.4.1, $\sigma_1^k \leftrightsquigarrow \sigma_0^k$. Recall that σ_1^k, σ_0^k decide $1, 0$ respectively by liveness. But this violates agreement because σ_1^k, σ_0^k are path-connected. □

Though 1-cast cannot solve consensus for two parties, once 2-cast becomes available it becomes universal in the sense that it can solve k-CCC for $k \geq 2$.

7.4.2 Three or More Parties

For three partes, the situation is more interesting. In general $(k-1)$-cast cannot solve k-CCC for $k \geq 2$.

Theorem 7.4.3. *Three parties cannot solve k-CCC using $(k-1)$-cast.*

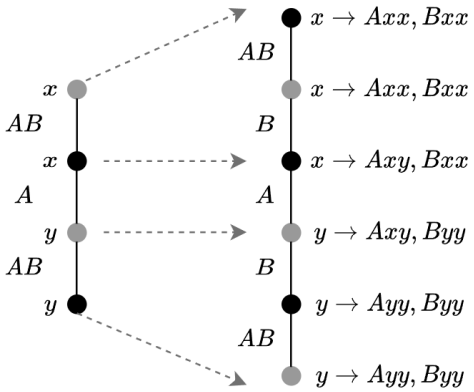

Figure 7.5: Illustration of construction for Lemma 7.4.1, for $k = 1$ alternating path.

Bibliography

[1] Order Book Official Definition.

[2] Jacob Abernethy, Yiling Chen, and Jennifer Wortman Vaughan. An optimization-based framework for automated market-making. In *Proceedings of the 12th ACM Conference on Electronic Commerce*, EC '11, page 297–306, New York, NY, USA, 2011. Association for Computing Machinery.

[3] Hayden Adams, Noah Zinsmeister, and Dan Robinson. Uniswap v2 core. `https://uniswap.org/whitepaper.pdf`, March 2020. As of 8 February 2021.

[4] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proceedings of the twentieth ACM symposium on operating systems principles*, SOSP '05, pages 45–58, New York, NY, USA, 2005. ACM.

[5] Guillermo Angeris and Tarun Chitra. Improved price oracles: Constant function market makers (june 1, 2020), 2020.

[6] Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An analysis of uniswap markets, 2019.

[7] AtomicDEX. First atomicdex stress test successfully completed. `https://atomicdex.io/first-atomicdex-stress-test-successfully-completed/`, November, 2019.

[8] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[9] Binance Academy. What are flash loans in DeFi?, 2020.

[10] bitcoinwiki. Atomic cross-chain trading.

[11] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. In *World Scientific Reference on Contingent Claims Analysis in Corporate Finance: Volume 1: Foundations of CCA and Equity Valuation*, pages 3–21. World Scientific, 2019.

[12] Sean Bowe and Daira Hopwood. Hashed time-locked contract transactions.

[13] Sean Bowe and Daira Hopwood. Hashed time-locked contract transactions. `https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki`, 2017. As of 9 January 2018.

[14] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[15] Yiling Chen and David M. Pennock. A utility framework for bounded-loss market makers. In *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*, UAI'07, page 49–56, Arlington, Virginia, USA, 2007. AUAI Press.

[16] Yiling Chen and Jennifer Wortman Vaughan. A new understanding of prediction markets via no-regret learning. In *Proceedings of the 11th ACM Conference on Electronic Commerce*, EC '10, page 189–198, New York, NY, USA, 2010. Association for Computing Machinery.

[17] A. Clement, H. Li, J. Napper, J. P. Martin Martin, L. Alvisi, and M. Dahlin. BAR primer. In *Proceedings of the international conference on dependable systems and networks (DSN), DCC symposium*, 2008. Place: Anchorage, AK.

[18] John C Cox, Stephen A Ross, and Mark Rubinstein. Option pricing: A simplified approach. *Journal of financial Economics*, 7(3):229–263, 1979.

[19] Jaksa Cvitanic and Fernando Zapatero. *Introduction to the economics and Mathematics of Financial Markets*. MIT Press, 2004.

[20] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, X. Zhao, I. Bentov, Lorenz Breidenbach, and A. Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *ArXiv*, abs/1904.05234, 2019.

[21] DeCred. Decred cross-chain atomic swapping.

[22] Decred. `https://github.com/decred/atomicswap`, 2018. As of 18 February 2021.

[23] Michael Egorov. Stableswap - efficient mechanism for stablecoin liquidity. `https://www.curve.fi/stableswap-paper.pdf`, November 2019. As of 8 Februaary 2021.

[24] Thomas Eizinger, Lloyd Fournier, and Phillip Hoenisch. The state of atomic swaps. `http://diyhpl.us/wiki/transcripts/scalingbitcoin/tokyo-2018/atomic-swaps/`, 2018.

[25] Daniel Engel and Maurice Herlihy. Composing networks of automated market makers. In *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, AFT '21, page 15–28, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Daniel Engel and Maurice Herlihy. Presentation and publication: Loss and slippage in networks of automated market makers, 2021.

[27] Daniel Engel, Maurice Herlihy, and Yingjie Xue. Failure is (literally) an Option: Atomic Commitment vs Optionality in Decentralized Finance. In Colette Johnen, Elad Michael Schiller, and Stefan Schmid, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 66–77, Cham, 2021. Springer International Publishing.

[28] Daniel Engel, Sucharita Jayanti, and Herlihy Maurice. A consensus hierarchy for cross-chain synchronization. 2023.

[29] Daniel Engel and Yingjie Xue. Transferable cross-chain options, 2022.

[30] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. Sok: Transparent dishonesty: Front-running attacks on blockchain. In Andrea Bracciali, Jeremy Clark, Federico Pintore, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 170–189, Cham, 2020. Springer International Publishing.

[31] Altcoin.io Exchange. The first ethereum ¡¿bitcoin atomic swap. `https://blog.altcoin.io/the-first-ethereum-bitcoin-atomic-swap-79befb8373a8`, October 2017. Altcoin.io Exchange.

[32] Ethereum Foundation. Erc20 token standard. `https://theethereum.wiki/w/index.php/ERC20_Token_Standard`, 2019. As of 6 April 2019.

[33] Christopher Goes. The interblockchain communication protocol: An overview. *arXiv preprint arXiv:2006.15918*, 2020.

[34] Trey Griffith. Sparkswap: Trade across blockchains without custody risk. `https://medium.com/sparkswap/sparkswap-trade-across-blockchans-without-custody-risk-a6bfe08013e8`, July, 2018.

[35] Runchao Han, Haoyu Lin, and Jiangshan Yu. On the optionality and fairness of Atomic Swaps. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 62–75, Zurich Switzerland, October 2019. ACM.

[36] Robin Hanson. Combinatorial Information Market Design. *Information Systems Frontiers*, 5(1):107–119, January 2003.

[37] Robin Hanson. Logarithmic market scoring rules for modular combinatorial information aggregation. *Journal of Prediction Markets*, 1(1):3–15, 2007.

[38] Ethan Heilman, Sebastien Lipmann, and Sharon Goldberg. The arwen trading protocols, January 2019.

[39] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, jan 1991.

[40] Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, PODC '18, pages 245–254, New York, NY, USA, 2018. ACM. Number of pages: 10 Place: Egham, United Kingdom tex.acmid: 3212736.

[41] Maurice Herlihy, Barbara Liskov, and Liuba Shrira. Cross-chain Deals and Adversarial Commerce. *Proceedings of the VLDB Endowment*, 13(2):100–113, October 2019. arXiv: 1905.09743.

[42] Eyal Hertzog, Guy Benartzi, and Galia Benartzi. Bancor protocol, 2017.

[43] Desmond J. Higham. *An introduction to financial option valuation: mathematics, stochastics and computation*. Cambridge Univ. Press, Cambridge, 4. printing edition, 2009.

[44] James A. Liu. Atomic Swaptions: Cryptocurrency Derivatives. *arXiv:1807.08644 [cs, q-fin]*, March 2020. arXiv: 1807.08644.

[45] Fernando Martinelli and Nikolai Mushegian. Balancer: A non-custodial portfolio man- ager, liquidity provider, and price sensor. https://balancer.finance/whitepaper/, 2109. As of 2 February 2021.

[46] Andreu Mas-Collell, Michael Whinston, and Jerry R. Green. *Microeconomic Theory*. Oxford University Press, 1995.

[47] Jason Milionis, Ciamac C. Moallemi, Tim Roughgarden, and Anthony Lee Zhang. Quantifying loss in automated market makers. In *Proceedings of the 2022 ACM CCS Workshop on Decentralized Finance and Security*, DeFi'22, page 71–74, New York, NY, USA, 2022. Association for Computing Machinery.

[48] T. Nolan. Atomic swaps using cut and choose, February 2016.

[49] The Komodo Organization. The BarterDEX whitepaper: A decentralized, open-source cryptocurrency exchange, powered by atomic-swap technology.

[50] Komodo Platform. Advanced blockchain technology, focused on freedom. `https://docs.komodoplatform.com/basic-docs/start-here/core-technology-discussions/introduction.html#note-on-changes-since-whitepaper-creation-cr-2019`, July,2019.

[51] Mohsen Pourpouneh, Kurt Nielsen, and Omri Ross. Automated Market Makers. IFRO Working Paper 2020/08, University of Copenhagen, Department of Food and Resource Economics, July 2020.

[52] Dan Robinson. Htlcs considered harmful. `http://diyhpl.us/wiki/transcripts/stanford-blockchain-conference/2019/htlcs-considered-harmful/`, 2019.

[53] Peter Robinson. Survey of crosschain communications protocols. *Computer Networks*, 200:108488, 2021.

[54] Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2008.

[55] Mojtaba Tefagh, Fateme Bagheri, Amirhossein Khajehpour, and Melika Abdi. Capital-free futures arbitrage, October, 2020.

[56] Wikipedia. Beta distribution — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Beta%20distribution&oldid=1145261204, 2023. [Online; accessed 23-March-2023].

[57] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[58] Jiahua Xu, Damien Ackerer, and Alevtina Dubovitskaya. A Game-Theoretic Analysis of Cross-Chain Atomic Swaps with HTLCs. *arXiv:2011.11325 [cs]*, April 2021. arXiv: 2011.11325.

[59] Yingjie Xue and Maurice Herlihy. Hedging Against Sore Loser Attacks in Cross-Chain Transactions. In *ACM Symposium on Principles of Distributed Computing*, 2021.

[60] Jake Yocom-Piatt. On-chain atomic swaps. https://blog.decred.org/2017/09/20/On-Chain-Atomic-Swaps/, September, 2017. Decred Blog.

[61] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. Atomic commitment across blockchains. *CoRR*, abs/1905.02847, 2019. arXiv: 1905.02847 tex.bibsource: dblp computer science bibliography, https://dblp.org tex.biburl: https://dblp.org/rec/bib/journals/corr/abs-1905-02847 tex.timestamp: Mon, 27 May 2019 13:15:00 +0200.

[62] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J. Knottenbelt. SoK: Communication Across Distributed Ledgers. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, volume 12675, pages 3–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2021. Series Title: Lecture Notes in Computer Science.

[63] Yi Zhang, Xiaohong Chen, and Daejun Park. Formal specification of constant product (x . y = k) market maker model and implementation. https://github.com/runtimeverification/verified-smart-contracts/blob/uniswap/uniswap/x-y-k.pdf, 2018.

[64] ZmnSCPxj. An argument for single-asset lightning network. https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-December/001752.html, 2018. As of 10 January 2021.

Appendix A

Mathematica Code

The following Mathematica code was used to generate Figure 5.6.

```
g[x_] := InverseFunction[f][x];
\[Phi][v_] := InverseFunction[f'][-v/(1-v)];
divloss[v1_,v2_] :=
   v2 \[Phi][v1] + (1-v2) f[\[Phi][v1]] - (v2 \[Phi][v2] + (1-v2) f[\[Phi][v2]]);

linslipx[v1_,v2_] :=
   ((1-v2)/(1-v1)) (v1 \[Phi][v2] - v1 \[Phi][v1]
 + (1-v1) f[\[Phi][v2]] - (1-v1) f[\[Phi][v1]]);

linslipy[v1_,v2_] :=
   (v2/v1) ((1-v1) f[\[Phi][v2]] - (1-v1) f[\[Phi][v1]]
 + v1 \[Phi][v2] - v1 \[Phi][v1]);

loadx[v1_,v2_] := divload[v1,v2] linslipx[v1,v2];
loady[v1_,v2_] := divloss[v1,v2] linslipy[v1,v2];

exploadx[v_,\[Alpha]1_,\[Alpha]2_] :=
   PDF[BetaDistribution[\[Alpha]1, \[Alpha]2]][v] loadx[1/2,v];
explody[v_,\[Alpha]1_,\[Alpha]2_] :=
   PDF[BetaDistribution[\[Alpha]1, \[Alpha]2]][v] loady[1/2,v];
expload[\[Alpha]1_, \[Alpha]2_] :=
 NIntegrate[exploadx[v, \[Alpha]1, \[Alpha]2], {v, 0, 1/2}] +
  NIntegrate[explody[v, \[Alpha]1, \[Alpha]2], {v, 1/2, 1}]

expslipx[v_,\[Alpha]1_,\[Alpha]2_] :=
   PDF[BetaDistribution[\[Alpha]1, \[Alpha]2]][v] linslipx[1/2,v];
expslipy[v_,\[Alpha]1_,\[Alpha]2_] :=
   PDF[BetaDistribution[\[Alpha]1, \[Alpha]2]][v] linslipy[1/2,v];
```

```
expslip[\[Alpha]1_, \[Alpha]2_] :=
 NIntegrate[expslipx[v, \[Alpha]1, \[Alpha]2], {v, 0, 1/2}] +
  NIntegrate[expslipy[v, \[Alpha]1, \[Alpha]2], {v, 1/2, 1}]

exploadx[v_,\[Alpha]1_,\[Alpha]2_] :=
    PDF[BetaDistribution[\[Alpha]1, \[Alpha]2]][v] divloss[1/2,v];
exploady[v_,\[Alpha]1_,\[Alpha]2_] :=
    PDF[BetaDistribution[\[Alpha]1, \[Alpha]2]][v] divloss[1/2,v];
exploa[\[Alpha]1_, \[Alpha]2_] :=
 NIntegrate[exploadx[v, \[Alpha]1, \[Alpha]2], {v, 0, 1/2}] +
  NIntegrate[exploady[v, \[Alpha]1, \[Alpha]2], {v, 1/2, 1}]

Plot3D[exploa[\[Alpha]1, \[Alpha]2], {\[Alpha]1, 1, 4}, {\[Alpha]2,
  1, 4}, PlotRange -> All, MeshFunctions -> {#3 &}]
```

Appendix B

Contract Code

The following contract code is used in Protocols 6.3.2-6.3.3.

```
1
2  //Assume the following:
3
4  // now is the time the transaction is included in a block
5  // A clear() function on the Mutation struct that resets all fields to default values
6  // A hash function H
7  // || denotes concatenation of inputs to a hash function
8  // A sig (digital signature) object with the following functions
9  //  valid(address) -> bool, returns true if valid signature by address
10 //  msg() -> [hash], returns an array of hashlocks if the signature signed such a message, null otherwise
11
12 contract MutSwapAB{
13
14    struct FollowerMutation{
15        //Signature by follower to allow candidate to take its position
16        Sig voucher;
17        //Candidate party to replace follower
18        address candidate_receiver;
19        //Hashlock used to replace follower
20        uint replace_hash_lock;
21        //Time mutation begins
22        uint start_time;
23        //Flag for freezing asset when a tentative replacement is happening
24        bool mutating;
25        //Controls whether asset is locked or not during mutation
26        bool can_lock_asset;
27    }
28
29    struct LeaderMutation{
```

```
        //Signature by follower to allow candidate to take its position
        Sig voucher;
        //Candidate party to replace follower
        address candidate_sender;
        //Party who called mutateLockLeader
        address mutator;
        //Hashlock used to replace follower
        uint replace_hash_lock;
        //Hashlock for exercising option
        uint swap_hash_lock;
        //Time mutation begins
        uint start_time;
        //Used for optimistic execution of protocol
        bool approved;
        //Flag for freezing asset when a tentative replacement is happening
        bool mutating;
        //Controls whether asset is locked or not during mutation
        bool can_lock_asset;
    }


    //State information for base swap protocol
    Asset asset; //Reference to preferred token contract
    address sender; //Current sender of escrowed funds
    address receiver; //Current receiver of escrowed funds
    address leader; //Leader of the original swap protocol
    address follower; //Follower of the original swap protocol
    uint swap_hash_lock; //Hashlock of the swap protocol
    uint T_AB; //Timeout for locked asset
    uint delta = 10 minutes; //Assumed worst case transaction inclusion time (Is arbitrary)

    //State info associated with mutable leader position
    LeaderMutation leader_mutation;

    //State info associated with mutable follower position
    FollowerMutation follower_mutation;

    //Controls whether follower/leader positions can be changed
    bool mutable;

    function MutSwapAB(Asset _asset,uint start,uint dT,address _sender,address _receiver,uint _swap_hash_lock,address
         _leader,address _follower,bool _mutable){
        require(msg.sender == _sender); //Sender can only escrow their own funds
        require(leader != follower); //Leader and follower should be distinct
        require(leader == _sender);
```

```
74        require(follower == _receiver);
75
76        this.mutable = _mutable;
77
78        //Inital mutation states
79        leader_mutation.mutating = false;
80        leader_mutation.approved = false;
81        follower_mutation.mutating = false;
82
83        sender = _sender;
84        receiver = _receiver;
85        swap_hash_lock = _swap_hash_lock; //Hashlock for the initial swap
86        leader = _leader;
87        follower = _follower;
88        asset = _asset;
89        asset.send(address(this));
90        T_AB = start + (dT+1)*delta; //Sender is Alice
91
92        if(mutable){
93            //For AB contract, asset is actively locked during leader and follower transfer
94            leader_mutation.can_lock_asset = true;
95            follower_mutation.can_lock_asset = true;
96        }
97    }
98
99    function claim(string secret){
100
101       //If a previous mutation lock was never completed, revert to original swap
102       if(mutable){
103           if(leader_mutation.mutating){
104               revertLeader();
105           }
106           if(follower_mutation.mutating){
107               revertFollower();
108           }
109       }
110
111       require(msg.sender == receiver.id); //Only receiver can call claim
112       require(now <= T_AB); //Must be before timeout
113       require(H(secret) == swap_hash_lock); //Claim conditional on revealing secret
114       asset.send(receiver);
115   }
116
117   function refund(){
118
```

```
        //If a previous mutation lock was never completed, revert to original swap
        if(mutable){
            if(leader_mutation.mutating){
                revertLeader();
            }
            if(follower_mutation.mutating){
                revertFollower();
            }
        }

        require(msg.sender == sender); //Only sender can call refund
        require(now > T_AB); //After lock has timed out
        asset.send(sender);
    }

    function mutateLockLeader(Sig sig,address _candidate_sender, uint _replace_hash_lock,uint _swap_hash_lock){
        require(mutable);

        //If a previous mutation lock is stale, then call revert to allow for a new mutation lock to be made
        if(leader_mutation.mutating){
            revertLeader();
        }

        require(!leader_mutation.mutating); //Only one mutate_lock at a time
        require(msg.sender == sender || msg.sender == receiver); //Only sender or receiver can mutate
        if(msg.sender == leader){ //Alice has less time to call mutateLock
            require(T_AB >= now + 8*delta);
        }else if(msg.sender == follower){
            //Bob is given more time to call mutateLock in response to Alice
            require(T_AB >= now + 7*delta);
        }

        require(sig.valid(leader)); //Requres Alice's sig of Carol's hashlocks
        require(sig.msg() == [_replace_hash_lock,_swap_hash_lock]); //The msg has to just be the candidate's hashlocks
        leader_mutating.mutating = true;
        leader_mutating.voucher = sig; //Proof that candidate was approved
        leader_mutating.candidate_sender = _candidate;
        leader_mutating.mutator = msg.sender;
        leader_mutating.replace_hash_lock = _replace_hash_lock;
        leader_mutating.swap_hash_lock = _swap_hash_lock;
        leader_mutating.start_time = now; //Used for flexible timeouts in the transfer
    }

    function mutateLockFollower(Sig sig,address _candidate_receiver, uint _replace_hash_lock){
        require(mutable);
```

```
164
165         //If a previous mutation lock is stale, then call revert to allow for a new mutation lock to be made
166         if(follower_mutation.mutating){
167             revertFollower();
168         }
169
170         require(msg.sender == follower); //Only Bob can call
171         require(!follower_mutation.mutating);
172         require(follower_mutation.lock_asset); //Only can call mutateLockFollower on AB contract
173         require(sig.valid(follower)); //Bob's valid signature
174         require(sig.msg() == [_replace_hash_lock]);
175         require(T_AB >= now + 3*delta); //Need enough time for David to call claim
176
177         follower_mutation.mutating = true;
178         follower_mutation.voucher = sig; //Proof that candidate was approved by previous owner of the position
179         follower_mutation.candidate_receiver = _candidate_receiver;
180         follower_mutation.replace_hash_lock = _replace_hash_lock;
181         follower_mutation.start_time = now; //Used for flexible timeouts in the transfer
182     }
183
184      function replaceLeader(string secret){
185         require(mutable);
186         require(leader_mutation.mutating);
187         bool candidate_round = false;
188         bool follower_round = false;
189
190         //If Bob mutated, can skip contesting phases since he is implicitly approving of the signature he called
                mutateLock with
191         if(leader_mutation.mutator == follower || (leader_mutation.mutator == leader && leader_mutation.approved)){ //
                Can immediately call replace once Bob has mutated
192             //Carol can replace once Bob has called mutate
193             candidate_round = (msg.sender == leader_mutation.candidate_sender) && (4*delta >= now - leader_mutation.
                    start_time > 0);
194             //Bob can call replace
195             follower_round = (msg.sender == follower) && (6*delta >= now - leader_mutation.start_time > 0);
196
197         }
198         else{ //Contesting (pessimistic case)
199             //Carol can replace in round right after last chance at contesting
200             candidate_round = (msg.sender == leader_mutation.candidate_sender) && (4*delta >= now - leader_mutation.
                    start_time > 2*delta);
201             //Bob can replace in the 3 rounds right after last chance at mutating round
202             follower_round = (msg.sender == follower) && (6*delta >= now - leader_mutation.start_time>0);
203         }
204
```

```
205
206         require(candidate_round || follower_round);
207         require(H(secret) == leader_mutation.replace_hash_lock);
208
209         sender = leader_mutation.candidate_sender; //Carol takes refund optionality from Alice
210         swap_hash_lock = mutation.swap_hash_lock;
211
212         leader_mutation.approved = false;
213         leader_mutation.mutating = false;
214         leader_mutation.mutation.clear();
215     }
216
217     //If no contesting has occurred, transfer can be complete
218     function replaceFollower(string secret){
219         require(mutable);
220         require(follower_mutation.mutating);
221         require(2*delta >= now - follower_mutation.start_time > 0); //2 rounds given for replacement
222
223         require(H(secret) == follower_mutation.replace_hash_lock);
224
225         receiever = follower_mutatation.candidate_receiver; //David takes refund optionality from Bob
226
227         follower_mutatation.clear();
228         follower_mutation.mutating = false;
229     }
230
231      //Contest if transferring party tries to transfer to two different parties simultaneously
232     //Note this method only does something if it can be proved that Alice lied
233     //This method can't do anything if Alice is honest
234     function contestLeader(Sig sig,string secret){
235         require(mutable);
236         require(msg.sender == follower); //Only allow Bob to call this
237         require(leader_mutation.mutating); //Can only contest a mutation if one has occured
238         require(!leader_mutation.approved); //Cannot contest after approval
239         require(leader_mutation.mutator == leader); //Can only contest a mutation when Alice called it, not Bob
240         require(2*delta >= now - leader_mutation.start_time > 0); //Can contest only in the 2 rounds after mutation
241         //Can contest if Alice creates two inconsistent signatures or tries to reveal preimage of hashlock too early
242         require(sig.valid(leader) || H(secret) == swap_hash_lock); //Make sure Alice actually signed the sig
243         if(sig != leader_mutation.voucher || H(secret) == swap_hash_lock){ //Checks if Alice reported inconsistent sigs
244             leader_mutation.clear();
245             leader_mutation.mutating = false;
246         }
247
248     }
249
```

```
    //Issue is really in the sequential consistent case
    //Bob can call approve in any next two rounds after a mutation
    function approveLeader(){
       require(mutable);
       require(msg.sender == follower); //Only Bob can approve
       require(leader_mutation.mutating); //Can only approve after a mutation has begun
       require(leader_mutation.mutator == leader); //Can only call approve when Alice was the one who called mutate
       require(2*delta >= now - leader_mutation.start_time > 0); //Can approve only in the 2 rounds after mutation
       leader_mutation.approved = true;
    }


    //If replacement doesn't happen quickly enough, can revert back to the old swap
    //Isn't called directly in order to save an extra round of the protocol
    function revertLeader(){
        require(mutable);
        require(leader_mutation.mutating);
        require(now - leader_mutation.start_time > 6*delta);

        leader_mutation.approved = false;
        leader_mutation.mutating = false;
        leader_mutation.clear();
    }

    //If replacement doesn't happen quickly enough, can revert back to the old swap
    //Isn't called directly in order to save an extra round of the protocol
    function revertFollower(){
        require(mutable);
        require(follower_mutation.mutating);
        require(now - follower_mutation.start_time > 2*delta);

        follower_mutation.mutating = false;
        follower_mutation.clear();
    }

}
```

```

//Assume the following:

// now is the time the transaction is included in a block
// A clear() function on the Mutation struct that resets all fields to default values
// A hash function H
// || denotes concatenation of inputs to a hash function
// A sig (digital signature) object with the following functions
```

```
9  //  valid(address) -> bool, returns true if valid signature by address
10 //  msg() -> [hash], returns an array of hashlocks if the signature signed such a message, null otherwise
11
12 contract MutSwapBA{
13
14     struct FollowerMutation{
15         //Signature by follower to allow candidate to take its position
16         Sig voucher;
17         //Candidate party to replace follower
18         address candidate_sender;
19         //Hashlock used to replace follower
20         uint replace_hash_lock;
21         //Time mutation begins
22         uint start_time;
23         //Flag for freezing asset when a tentative replacement is happening
24         bool mutating;
25         //Controls whether asset is locked or not during mutation
26         bool can_lock_asset;
27     }
28
29     struct LeaderMutation{
30         //Signature by follower to allow candidate to take its position
31         Sig voucher;
32         //Candidate party to replace follower
33         address candidate_receiver;
34         //Party who called mutateLockLeader
35         address mutator;
36         //Hashlock used to replace follower
37         uint replace_hash_lock;
38         //Hashlock for exercising option
39         uint swap_hash_lock;
40         //Time mutation begins
41         uint start_time;
42         //Used for optimistic execution of protocol
43         bool approved;
44         //Flag for freezing asset when a tentative replacement is happening
45         bool mutating;
46         //Controls whether asset is locked or not during mutation
47         bool can_lock_asset;
48     }
49
50
51     //State information for base swap protocol
52     Asset asset; //Reference to preferred token contract
53     address sender; //Current sender of escrowed funds
```

```
54    address receiver; //Current receiver of escrowed funds
55    address leader; //Leader of the original swap protocol
56    address follower; //Follower of the original swap protocol
57    uint swap_hash_lock; //Hashlock of the swap protocol
58    uint T_BA; //Timeout for locked asset
59    uint delta = 10 minutes; //Assumed worst case transaction inclusion time
60
61    //State info associated with mutable leader position
62    LeaderMutation leader_mutation;
63
64    //State info associated with mutable follower position
65    FollowerMutation follower_mutation;
66
67    //Controls whether follower/leader positions can be changed
68    bool mutable;
69
70    function MutSwapBA(Asset _asset,uint start,uint dT,address _sender,address _receiver,uint _swap_hash_lock,address
          _leader,address _follower,bool _mutable){
71        require(msg.sender == _sender); //Sender can only escrow their own funds
72        require(leader != follower); //Leader and follower should be distinct
73        require(follower == _sender);
74        require(leader == _receiver);
75
76        this.mutable = _mutable;
77
78        //Inital mutation states
79        leader_mutation.mutating = false;
80        leader_mutation.approved = false;
81        follower_mutation.mutating = false;
82
83        sender = _sender;
84        receiver = _receiver;
85        swap_hash_lock = _swap_hash_lock; //Hashlock for the initial swap
86        leader = _leader;
87        follower = _follower;
88        asset = _asset;
89        asset.send(address(this));
90        T_BA = start + dT*delta;
91
92        if(mutable){
93            leader_mutation.lock_asset = true;
94            follower_mutatation.lock_asset = false; //Prevents Bob from denying Alice her optionality by locking her
                  asset
95        }
96    }
```

```
97
98  function claim(string secret){
99
100     //If a previous mutation lock was never completed, revert to original swap
101     if(mutable){
102         if(leader_mutation.mutating){
103             revertLeader();
104         }
105     }
106
107     require(msg.sender == receiver.id); //Only receiver can call claim
108     require(now <= T_BA); //Must be before timeout
109     require(H(secret) == swap_hash_lock); //Claim conditional on revealing secret
110     asset.send(receiver);
111 }
112
113 function refund(){
114
115     //If a previous mutation lock was never completed, revert to original swap
116     if(mutable){
117         if(leader_mutation.mutating){
118             revertLeader();
119         }
120         if(follower_mutation.mutating){
121             revertFollower();
122         }
123     }
124
125     require(msg.sender == sender); //Only sender can call refund
126     require(now > T_BA); //After lock has timed out
127     asset.send(sender);
128 }
129
130 function mutateLockLeader(Sig sig,address _candidate_receiver, uint _replace_hash_lock,uint _swap_hash_lock){
131     require(mutable);
132
133     //If a previous mutation lock is stale, then call revert to allow for a new mutation lock to be made
134     if(leader_mutation.mutating){
135         revertLeader();
136     }
137
138     require(!leader_mutation.mutating); //Only one mutate_lock at a time
139     require(msg.sender == sender || msg.sender == receiver); //Only sender or receiver can mutate
140     if(msg.sender == leader){ //Alice has less time to call mutateLock
141         require(T_BA >= now + 7*delta);
```

```
142         }else if(msg.sender == follower){
143             //Bob is given more time to call mutateLock in response to Alice
144             require(T_BA >= now + 6*delta);
145         }
146 
147         require(sig.valid(leader)); //Requres Alice's sig of Carol's hashlocks
148         require(sig.msg() == [_replace_hash_lock,_swap_hash_lock]); //The msg has to just be the candidate's hashlocks
149         leader_mutating.mutating = true;
150         leader_mutating.voucher = sig; //Proof that candidate was approved
151         leader_mutating.candidate = _candidate_receiver;
152         leader_mutating.mutator = msg.sender;
153         leader_mutating.replace_hash_lock = _replace_hash_lock;
154         leader_mutating.swap_hash_lock = _swap_hash_lock;
155         leader_mutating.start_time = now; //Used for flexible timeouts in the transfer
156     }
157 
158 
159     //Provides similar functionality to mutateLockLeader but doesn't lock the escrowed resource from being locked
160     function mutateLockFreeFollower(Sig sig,address _candidate_sender,uint _replace_hash_lock){
161         require(mutable);
162 
163         require(msg.sender == follower); //Only Bob can call attest
164         require(!follower_mutation.mutating);
165         require(!follower_mutation.lock_asset);
166         require(sig.valid(follower)); //Bob's valid signature
167         require(sig.msg() == [_replace_hash_lock]);
168         require(T_BA >= now + 2*delta); //Need enough time for David to call claim
169 
170         follower_mutation.mutating = true;
171         follower_mutation.voucher = sig; //Proof that candidate was approved
172         follower_mutation.candidate_receiver = _candidate_sender;
173         follower_mutation.replace_hash_lock = _replace_hash_lock;
174         follower_mutation.start_time = now; //Used for flexible timeouts in the transfer
175     }
176 
177      function replaceLeader(string secret){
178         require(mutable);
179         require(leader_mutation.mutating);
180         bool candidate_round = false;
181         bool follower_round = false;
182 
183         //If Bob mutated, can skip contesting phases since he is implicitly approving of the signature he called
                mutateLock with
184         if(leader_mutation.mutator == follower || (leader_mutation.mutator == leader && leader_mutation.approved)){ //
                Can immediately call replace once Bob has mutated
```

```
185         //Carol can replace once Bob has called mutate
186         candidate_round = (msg.sender == leader_mutation.candidate_receiver) && (4*delta >= now - leader_mutation.
                start_time > 0);
187         //Bob can call replace
188         follower_round = (msg.sender == follower) && (6*delta >= now - leader_mutation.start_time > 0);
189
190     }
191     else{ //Contesting (pessimistic case)
192         //Carol can replace in round right after last chance at contesting
193         candidate_round = (msg.sender == leader_mutation.candidate_receiver) && (4*delta >= now - leader_mutation.
                start_time > 2*delta);
194         //Bob can replace in the 3 rounds right after last chance at mutating round
195         follower_round = (msg.sender == follower) && (6*delta >= now - leader_mutation.start_time > 0);
196     }
197
198
199     require(candidate_round || follower_round);
200     require(H(secret) == leader_mutation.replace_hash_lock);
201
202     receiever = leader_mutation.candidate_receiver; //Carol takes receiving optionality from Alice
203     swap_hash_lock = mutation.swap_hash_lock;
204
205     leader_mutation.approved = false;
206     leader_mutation.mutating = false;
207     leader_mutation.mutation.clear();
208 }
209
210 //If no contesting has occurred, transfer can be complete
211 function replaceFollower(string secret){
212     require(mutable);
213     require(follower_mutation.mutating);
214     require(2*delta >= now - follower_mutation.start_time > 0); //2 rounds given for replacement
215
216     require(H(secret) == follower_mutation.replace_hash_lock);
217
218     sender = follower_mutatation.candidate_sender; //David takes refund optionality from Bob
219
220     follower_mutatation.clear();
221     follower_mutation.mutating = false;
222 }
223
224  //Contest if transferring party tries to transfer to two different parties simultaneously
225 //Note this method only does something if it can be proved that Alice lied
226 //This method can't do anything if Alice is honest
227 function contestLeader(Sig sig,string secret){
```

```
228         require(mutable);
229         require(msg.sender == follower); //Only allow Bob to call this
230         require(leader_mutation.mutating); //Can only contest a mutation if one has occured
231         require(!leader_mutation.approved); //Cannot contest after approval
232         require(leader_mutation.mutator == leader); //Can only contest a mutation when Alice called it, not Bob
233         require(2*delta >= now - leader_mutation.start_time > 0); //Can contest only in the 2 rounds after mutation
234         //Can contest if Alice creates two inconsistent signatures or tries to reveal preimage of hashlock too early
235         require(sig.valid(leader) || H(secret) == swap_hash_lock);
236         if(sig != leader_mutation.voucher || H(secret) == swap_hash_lock){ //Checks if Alice reported inconsistent sigs
237             leader_mutation.clear();
238             leader_mutation.mutating = false;
239         }
240
241     }
242
243     //Issue is really in the sequential consistent case
244     //Bob can call approve in any next two rounds after a mutation
245     function approveLeader(){
246         require(mutable);
247        require(msg.sender == follower); //Only Bob can approve
248        require(leader_mutation.mutating); //Can only approve after a mutation has begun
249        require(leader_mutation.mutator == leader); //Can only call approve when Alice was the one who called mutate
250        require(2*delta >= now - leader_mutation.start_time > 0); //Can approve only in the 2 rounds after mutation
251        leader_mutation.approved = true;
252     }
253
254
255     //If replacement doesn't happen quickly enough, can revert back to the old swap
256     //Isn't called directly in order to save an extra round of the protocol
257     function revertLeader(){
258         require(mutable);
259         require(leader_mutation.mutating);
260         require(now - leader_mutation.start_time > 6*delta);
261
262         leader_mutation.approved = false;
263         leader_mutation.mutating = false;
264         leader_mutation.clear();
265     }
266
267     //If replacement doesn't happen quickly enough, can revert back to the old swap
268     //Isn't called directly in order to save an extra round of the protocol
269     function revertFollower(){
270         require(mutable);
271         require(follower_mutation.mutating);
272         require(now - follower_mutation.start_time > 2*delta);
```

```
273
274         follower_mutation.mutating = false;
275         follower_mutation.clear();
276     }
277
278 }
```

Appendix C

Protocol Security Proofs

C.0.1 Misc Proofs

First, we provide proofs for the properties of the leader transfer protocol.

Lemma C.0.1. *If Alice and Carol are both conforming in the Mutate Lock Phase, then both AB and BA are mutate locked with $(H(C_1), H(C_2))$ by $startLeader + 2\Delta$.*

Proof. If Carol is conforming, she escrows her principal to Alice by time $R + \Delta$. Alice, after observing the creation of CA, will then call *mutateLockLeader()* within a Δ on both AB and BA, passing $(H(C_1), H(C_2))$ as the transfer hashlock and the new swap hashlock. For any contract, if it is not mutate locked, then the *mutateLockLeader()* sent by Alice can be included in the blockchain by $startLeader + 2\Delta$. If the *mutateLockLeader()* transaction sent by Alice can not be included in the blockchain due to someone else calling it first, it must be that Bob called *mutateLockLeader()* on the contract with $(H(C_1), H(C_2))$ before her. In any case, within $startLeader + 2\Delta$, both contracts are mutate locked by $(H(C_1), H(C_2))$. □

If there is a successful call to *mutateLockLeader()* on AB with $(H(C_1), H(C_2))$, let the first time that this transaction is called be t_{AB}. Similarly, let t_{BA} be the first time *mutateLockLeader()* is called on BA with $(H(C_1), H(C_2))$.

Theorem 6.3.2. *Protocol 6.3.2 satisfies liveness: If Alice, Bob, and Carol are all conforming, then Alice gets Carol's principal, Carol gets Alice's position, and Bob maintains his position.*

Proof. If Alice starts the transfer procedure, and all parties are conforming, then Carol owns the option and paid her principal to Alice. The argument is as follows.

By Lemma C.0.1, we know *mutateLockLeader()* has been called by $R + 2\Delta$ on AB and BA with $(H(C_1), H(C_2))$. After $\max\{t_{AB}, t_{BA}\} + 2\Delta \leq startLeader + 4\Delta$, Carol will know that Bob cannot successfully call *contest()* on either AB or BA. Carol then releases her secret C_1 to AB and BA by $\max\{t_{AB}, t_{BA}\} + 3\Delta \leq startLeader + 5\Delta$. Then AB and BA are transferred to a new state where the swap hashlock is $H(C_2)$, the sender of AB contract becomes Carol, and the receiver of BA contract becomes Carol. At this point, these contracts can be more appropriately

renamed as as CB and BC respectively. That is to say, Carol has set up a swap with Bob using her swap hashlock $H(C_2)$. Carol now owns the option. Because the timeout on CA is $startLeader + 10\Delta$, Alice can use C_1 to redeem Carol's principal on CA by revealing C_1 by $startLeader + 6\Delta$. . □

Theorem 6.3.3. *Protocol 6.3.2 satisfies transfer independence: Alice can transfer her position to Carol without Bob's participation.*

Proof. If Alice and Carol are both conforming, then by Lemma C.0.1, we know *mutateLock()* will be called with $(H(C_1), H(C_2))$ on both AB and BA by $startLeader + 2\Delta$. In the worst case, after $\max\{t_{AB}, t_{BA}\} + 2\Delta$, Carol should release C_1 to replace Alice on both AB and BA. This results in the replacement of the old swap hashlock $H(A_1)$ with $H(C_2)$. Bob does not have to call any functions to help Carol replace Alice's role. Thus, conforming Alice and Carol can trade without Bob's cooperation. □

We have proven that conforming Alice and Carol can trade the option without Bob cooperating. Now we look at this property from another perspective. Can Bob actively prevent Alice selling her option to Carol? The following result guarantees Bob cannot block Alice and Carol.

Theorem 6.3.4. *Protocol 6.3.2 satisfies non-blocking transfer: Alice can transfer her position to Carol even if Bob is adversarial.*

Proof. The only way for Bob to block the transfer of an option is to contest. To contest, Bob needs to show a different mutation signed by Alice or by showing Alice's secret to the old hashlock. As long as Alice is conforming to the protocol and does not collude with Bob, which is the case in this context, an adversarial Bob cannot forge her signature and successfully call *contest()*. □

Theorem 6.3.5. *Protocol 6.3.2 satisfies transfer atomicity: If Alice loses her position in the swap, then she can claim Carol's principal.*

Proof. Conforming Carol only loses her principal if she releases her secret C_1. She only releases her secret C_1 after she sees both AB and BA are both mutate locked by $(H(C_1), H(C_2))$ and at least one of the conditions are met: (1)Bob has approved both AB and BA; (2) both of his contest windows on AB and BA have ended. Bob can no longer contest this result either due to timeout of his contesting window or Bob has approved this result. Then when Carol sends her secret C_1 to both AB and BA, she replaces Alice's role and changes the swap hashlock to $H(C_2)$ thus she owns the option.

Conforming Alice loses her option only if Carol releases her secret C_1 on some contract. We know by Lemma C.0.1 that in the worst case *mutateLockLeader()* will be called on AB and BA by $startLeader + 2\Delta$. Thus, the latest time any party can call *replace()* is. $startLeader + 8\Delta$. So if Alice loses her option there must have been a *replace()* call by $startLeader + 8\Delta$. Since CA has timeout $startLeader + 10\Delta$, Alice is guaranteed enough time to claim Carol's principal. □

The remaining proofs are for the properties of the follower transfer protocol.

Lemma C.0.2. *In Protocol 6.3.3, if Bob, David are both compliant, and Alice does nothing, then Bob gets David's principal, and David gets Bob position.*

Proof. By assumption, we know the *Mutate Phase* ends $startFollower + 2\Delta$. At this point, $candiate\_sender = David.address$ on BA and $candidate\_receiver = David.address$ on AB. By $startFollower + 3\Delta$ we know $David$ will reveal D_1 to AB and BA. At this point $sender = David$ on BA and $receiver = David$ on AB. Thus David has successfully taken Bob's position. When Bob forwards D_1 to the DB, it will arrive no later than $startFollower + 4\Delta < startFollower + 5\Delta$. So Bob gets David's principal. □

Theorem 6.3.7. *Protocol 6.3.3 satisfies liveness: If Alice, Bob, and David are all conforming, and Alice doesn't reveal A_1, then Bob gets David's principal, David gets Bob's position, and Alice maintains her position.*

Proof. Because Alice is compliant and doesn't reveal A_1, she does nothing according to Protocol 6.3.3. Since Bob and David are both compliant, the result follows from Lemma C.0.2. □

Theorem 6.3.8. *Protocol 6.3.3 satisfies transfer independence: Bob can transfer his position to David without Alice's participation.*

Proof. Assume Alice does nothing and Bob, David are compliant. By Lemma C.0.2, the result follows immediately. □

Theorem 6.3.9. *Protocol 6.3.3 satisfies non-blocking transfer: Bob can transfer his position to David even if Alice is adversarial.*

Proof. Alice's only ability during Protocol 6.3.3, is to choose to reveal her secret A_1. Since Bob and David are compliant, David creates DB by $startFollower + \Delta$. In the worst case, Bob calls $mutateLockFreeFollower()$ on BA at the last available opportunity, at $T - 2\Delta$. Since Bob is compliant, he also calls $mutateLockFollower()$ on AB by $T - 2\Delta$. Since David is compliant, he calls $replace()$ on AB, BA by $T - \Delta$, thereby revealing D_1 to Bob. Both replace calls are then completed by $startFollower + 3\Delta$. Bob then has enough time to reveal D_1 on DB since its timeout is $startFollower + 5\Delta$. If Alice reveals A_1 by T on BA, then David will have Bob's position so he on AB and BA. Thus he can reveal A_1 on AB since it has timeout $T + \Delta$. □

Theorem 6.3.10. *Protocol 6.3.3 satisfies transfer atomicity: If Bob loses his swap position, then he can claim David's principal.*

Proof. Suppose a conforming Bob loses his swap position. By Protocol 6.3.3, we know $mutateLockFollower()$ and $mutateLockFreeFollower()$ are called on AB and BA respectively by $startFollower + 2\Delta$ by Bob. By assumption Bob loses his swap position. Thus David must have revealed D_1 on AB or BA before the $replaceFollower()$ timeout. Since each $replaceFollower()$ timeout is 2Δ, the latest David could have revealed D_1 is $startFollower + 4\Delta$. Conforming Bob sends D_1 to DB which arrives by $startFollower + 5\Delta$. Since the timeout on DB is $startFollower + 5\Delta$, this ensures Bob gets David's principal. □

C.0.2 No UNDERWATER

The goal of this section is to prove Theorem 6.3.1.

If the reselling is started by Alice calling *mutateLockLeader()* on AB and/or BA contract, the state of the corresponding contract will be changed. We first define states of contracts. These are referenced from Appendix B. A contract's state is defined by the following elements:

- *sender*: *sender* can call *refund()* if the receiver does not redeem it before swap hashlock times out.
- *receiver*: *receiver* can call *claim()* and pass the preimage of swap hashlock to get the asset escrowed in the contract.
- *mutating*: *mutating* is a bool where *mutating=true* means the asset is temporarily not redeemable due to some active mutation in progress. In other words, the right to claim the asset is locked. On the other hand, *mutating=false* means the asset can be claimed if the preimage to the hashlock is sent to the contract.
- *swap_hash_lock*: The swap hashlock is the hashlock used to claim assets. If the preimage of the hashlock is sent to *claim()*, then the receiver can claim the asset.
- *replace_hash_lock*: If *replace_hash_lock != nil*, that means Alice would like to transfer her option. *replace_hash_lock* stores the hashlock for the transfer.
- *candidate_sender/candidate_receiver*: denotes the new sender or receiver when transfer is finalized, i.e. Carol replaces Alice's role.

We take AB as an example to illustrate the states. The states on BA are symmetric which can be inferred from the context.

- $Ready2Claim(H(A_1), AB)$ denotes state $(sender = Alice, receiver = Bob, mutating = false, swap\_hash\_lock = H(A_1))$, meaning if A_1 is sent before the contract expires, Bob can claim Alice's principal. We will use $Ready2Claim(H(A_1))$ for simplicity when which contract we are referring to is obvious from the context.
- $MutateLockContestable(Carol, H(C_1), H(C_2), AB)$ denotes the state $(sender = Alice, receiver = Bob, mutating = true, replace\_hash\_lock = H(C_1), swap\_hash\_lock = H(C_2), candidate_receiver = Carol)$, which means Alice tentatively locks the asset and transfers her role to Carol. In this state Bob is able to call *contest()*. We will use $MutateLockContestable(H(C_1), H(C_2))$ for simplicity when other parts we are referring to are obvious from the context. $MutateLockContestable$ is just a preparation for Carol to replace Alice's role. The *replaceLeader()* cannot be called until the state defined below.
- $MutateLockNonContestable(Carol, H(C_1), H(C_2), AB)$. In this state Bob cannot call *contest()* because his contest window has passed or he approved the mutation. At this state, Carol can release her secret C_1 to replace Alice. The state when Carol can replace Alice is denoted as

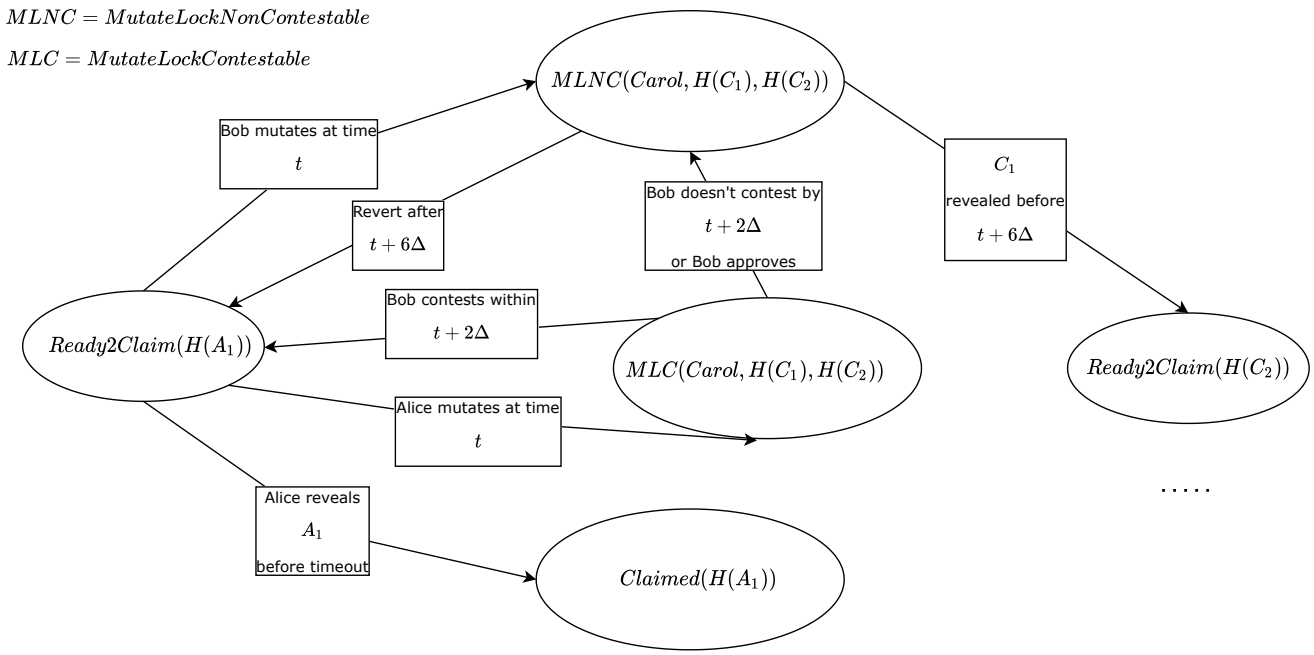

Figure C.1: State transition until Carol replaces Alice on AB contract

$MutateLockNonContestable(Carol, H(C_1), H(C_2), AB)$. $MutateLockNonContestable(H(C_1), H(C_2))$ is the short name.

- $Ready2Claim(H(C_2), CB)$. After $MutateLockNonContestable(Carol, H(C_1), H(C_2), AB)$ is reached, Carol can call *replace()* to replace Alice's role by sending C_1. After the replacement, the state becomes $(sender = Carol, receiver = Bob, mutation\_lock = false, swap\_hash\_lock = H(C_2))$, denoted as $Ready2Claim(H(C_2), CB)$. $Ready2Claim(H(C_2))$ is the short name.
- $Claimed(H(S), AB)$ means the asset on AB contract is claimed by a secret S which is the preimage to $H(S)$ and Bob gets the asset. In this context, S can be A_1 or C_2. $Claimed(H(S))$ is the short name.

The state transition is depicted in Fig C.1.

There are two stages of changes of state on the AB and BA contracts.

1. First stage. Alice and Bob aim to change the state of both contracts from $Ready2Claim(H(A_1))$ to a mutation state $MutateLockNonContestable(H(C_1), H(C_2))$, during which state $MutateLockContestable(H(C_1), H(C_2))$ may be reached temporally, corresponding to Mutate Lock Phase and Consistency Phase.
2. Second stage. Starting from $MutateLockNonContestable(H(C_1), H(C_2))$, Bob and Carol are involved in an atomic change from $MutateLockNonContestable(H(C_1), H(C_2), Carol)$ to $Ready2Claim(H(C_2))$ by Carol

releasing a secret C_1. It is similar to a standard atomic swap where Carol is the leader and Bob is the follower, corresponding to the Replace/Revert Phase.

Theorem C.0.3. *Conforming Alice will never end up in UNDERWATER.*

Proof. If Carol never creates CA or creates CA with different conditions than what Carol and Alice agreed upon, then Alice does nothing and so she doesn't lose her option.

Otherwise, consider the case when Carol creates CA with the conditions Alice and Carol had agreed upon.

If Alice is conforming, by Lemma C.0.1 then both AB and BA are mutate locked with $(H(C_1), H(C_2))$ by $startLeader + 2\Delta$. This means AB and BA both have reached the state $MutateLockContestable(H(C_1), H(C_2))$ by $startLeader + 2\Delta$. Since Alice is conforming, by $startLeader + 4\Delta$, the state will transition to $MutateLockNonContestable(H(C_1), H(C_2))$ since Bob cannot forge Alice's signature and contest.

After $MutateLockNonContestable(H(C_1), H(C_2))$, if Carol releases C_1, Alice observes it in the Replace/Revert phase, and can get Carol's principal as shown in Theorem 6.3.5. If Carol does not release C_1, then eventually, after $t_{AB} + 6\Delta$ and $t_{BA} + 6\Delta$ respectively, the states on both contracts can be reverted to $Ready2Claim(H(A_1))$ which means Alice still owns the option and it is unlocked. If Alice finally owns the option, then she does not end up in *UNDERWATER* by the guarantee of atomic swap. If Alice loses her option(maybe only lose a role on one contract), she gets Carol's principal, with/without losing her principal escrowed in her old option, which is acceptable for her. □

Theorem C.0.4. *Conforming Carol will never end up in UNDERWATER.*

Proof. After Carol escrows her principal to Alice on CA and sends her hashlocks $H(C_1), H(C_2)$ to Alice, Carol does not do anything in the first stage but observe. She only joins in the protocol after she observes that both contracts are in $MutateLockNonContestable(H(C_1), H(C_2))$ state. Otherwise, she is silent and she will not end up in *UNDERWATER* since her principal will be refunded eventually. After she observes $MutateLockNonContestable(H(C_1), H(C_2))$ state on both contracts, she releases C_1, moving $MutateLockNonContestable(H(C_1), H(C_2))$ state to $Ready2Claim(H(C_2))$. Then she owns the option provided by Bob and takes over Alice's option. In that case, she may lose her principal to Alice due to the release of C_1. Say, in the worst, she loses her principal to Alice.

Then, if Carol decides to exercise the option, she gets Bob's principal. That means, her principal is exchanged with Bob's. If she does not exercise the option and let it expire, she can get Alice's principal. That means, her principal is exchanged with Alice's. In any case, Carol never ends up in *UNDERWATER*. □

To prove that conforming Bob does not end up in *UNDERWATER*, we need to to prove the consistency of states on two contracts. Specifically,

- In the first stage, if Alice calls *mutateLockLeader()* in any contract, either both contracts eventually gets $MutateLockNonContestable(H(C_1), H(C_2)$ or reverted back to $Ready2Claim(H(A_1))$.

- In the second stage, if Carol releases C_1, both contracts are eventually at state $Ready2Claim(H(C_2))$. If Carol does not releases C_1, both contracts are eventually are in state $Ready2Claim(H(A_1))$.

Suppose, without loss of generality, that AB contract is the first to update its state from $Ready2Claim(H(A_1))$ to a new state by Alice calling *mutateLockLeader()* at some time t, assuming Bob is conforming.

Theorem C.0.5. *If BA is not claimed by $t+\Delta$, then BA will be mutate locked by $t+\Delta$.*

Proof. Since Bob is conforming, when Alice calls *mutateLockLeader()* on the AB contract, Bob can send *mutateLockLeader()* transaction on the BA within Δ. Recall that the start time that *mutateLockLeader()* is called is denoted as t_{AB} and t_{BA} respectively. Then in BA *mutateLockLeader()* is called on BA by $t_{AB}+\Delta$.

If Bob's *mutateLockLeader()* is included on BA, then $t_{BA} \leq t_{AB}+\Delta$. If Bob's *mutateLockLeader()* on BA is not included, then it must be Alice was able to call it before he was. In either case, $t_{BA} \leq t_{AB}+\Delta$. □

The case that by $t+\Delta$, the other contract's state changes to $Claimed(H(A_1))$ will be analyzed in Theorem C.0.8.

Theorem C.0.6. *Suppose two contracts both eventually reach $MutateLockNonContestable(H(C_1), H(C_2))$. Then, either both contracts are eventually $Ready2Claim(H(A_1))$ or $Ready2Claim(H(C_2))$.*

Proof. Since we know the start time of mutation between two contracts does not stagger beyond Δ. Denote the start time on two contracts as t_{first} and t_{second} respectively where $t_{first} \leq t_{second}$. Since Bob is conforming, $t_{second} - t_{first} \leq \Delta$. We denote the timeout for Carol to release C_1 on AB and BA contracts as t_1,t_2 respectively, and the timeout for Bob to release C_1 as t_3,t_4. Without loss of generality, assume $t_1 = t_{first}+4\Delta$ and $t_2 = t_{second}+4\Delta$. $t_3 = t_{first}+6\Delta$ and $t_4 = t_{second}+6\Delta$. We see that the latest timeout for Carol to release C_1 is t_2. The earliest timeout for Bob to send C_1 satisfies $t_3 \geq t_2+\Delta$ since we have $t_3 = t_{first}+6\Delta \geq t_{second}+5\Delta = t_2+\Delta$. That means, if Carol releases C_1, then it is eventually sent to both contracts and the state is $Ready2Claim(H(C_2))$. If Carol does not release C_1, then both contracts are reverted to $Ready2Claim(H(A_1))$ after timeout t_3 and t_4, respectively. □

The atomic changes between AB and BA are a bit more complicated, since starting from $Ready2Claim(H(A_1))$, there are multiple possible state changes available on both contracts: $Claimed(H(A_1))$, $MutateLockContestable(H(C_1), H(C_2))$, and $MutateLockNonContestable(H(C_1), H(C_2))$. The states $Claimed(H(A_1))$ and $MutateLockNonContestable(H(C_1), H(C_2))$ are not contestable. Assuming Bob is conforming and relay whatever he sees from one contract to another contract.

Lemma C.0.7. *If a contract is transferred to $MutateLockContestable(H(C_1), H(C_2))$ at t, then within $t+2\Delta$, it learns the other contract's update and then agree on either reverting back if there is a conflict or agree on the same mutation if no conflict.*

Proof. If a contract, say AB, is transferred to $MutateLockContestable(H(C_1), H(C_2))$ at time t, since the compliant Bob relays it, whatever happens before $t-\Delta$ to AB contract, that means, at time $t-\Delta$, the other chain is in $Ready2Claim(H(A_1))$. If there is any change happening on BA contract, it would happen between $t-\Delta$ to $t+\Delta$.

Then, by $t + 2\Delta$, AB contract will learn what happens on BA contract between $t - \Delta$ to $t + \Delta$ by Bob. If AB contract does not receive any transaction from Bob, then no conflicting changes happened on BA from $t - \Delta$ to $t + \Delta$. Bob can update BA contract to $MutateLockContestable(H(C_1), H(C_2))$ by relaying Alice's mutation to BA contract, and AB contract can be updated to $MutateLockNonContestable(H(C_1), H(C_2))$ after $t + 2\Delta$. If there is any state change between $t - \Delta$ to $t + \Delta$ on BA contract, then by $t + 2\Delta$, AB contract would be informed, and its own state update is also sent to BA contract. Then the conflicting state update would revert back their state change to $Ready2Claim(H(A_1))$ when it receives the conflicting change. □

Theorem C.0.8. *Conforming Bob never ends up in UNDERWATER.*

Proof. Starting from $Ready2Claim(H(A_1))$,

1. If a contract is transferred to $MutateLockContestable(H(C_1), H(C_2))$ at t, and the other contract is not updated to $Claimed(H(A_1))$, then by Lemma C.0.7, we know that either both contracts eventually reach $MutateLockNonContestable(H(C_1), H(C_2))$ or are reverted back to $Ready2Claim(H(A_1))$.

2. If a contract BA is transferred to $Claimed(H(A_1))$ state at t, then, if the other contract makes temporary $MutateLockContestable(H(C_1), H(C_2)$, it will be revert back to $Ready2Claim(H(A_1))$ by $t + \Delta$, and then transfer to $Claimed(H(A_1))$. Bob does not end up with *UNDERWATER*.

3. If a contract is transferred directly to $MutateLockNonContestable(H(C_1), H(C_2))$ at t, then the other contract must be $MutateLockContestable(H(C_1), H(C_2))$ at $t' \in [t - \Delta, t]$, and then after $t + 2\Delta$ or sooner, the other contract will become $MutateLockNonContestable(H(C_1), H(C_2))$.

When $MutateLockNonContestable(H(C_1), H(C_2))$ is reached on both contracts, by Theorem C.0.6, Bob will either be involved in the swap with Alice by Alice's secret A_1 or involved in the swap with Carol by Carol's secret C_2. By the guarantee of atomic swap, Bob will not end up with *UNDERWATER*.

□

The details for showing Protocol 6.3.3 satisfies *No UNDERWATER* are omitted since they mirror the previous arguments shown for 6.3.2.