A Formal Approach to AMM Fee Mechanisms with Lean 4

Marco Dessalvi
Technical University of Denmark, Lyngby, Denmark

Massimo Bartoletti
University of Cagliari, Italy

Alberto Lluch-Lafuente
Technical University of Denmark, Lyngby, Denmark

Abstract

Decentralized Finance (DeFi) has revolutionized financial markets by enabling complex asset-exchange protocols without trusted intermediaries. Automated Market Makers (AMMs) are a central component of DeFi, providing the core functionality of swapping assets of different types at algorithmically computed exchange rates. Several mainstream AMM implementations are based on the constant-product model, which ensures that swaps preserve the product of the token reserves in the AMM – up to a *trading fee* used to incentivize liquidity provision. Trading fees substantially complicate the economic properties of AMMs, and for this reason some AMM models abstract them away in order to simplify the analysis. However, trading fees have a non-trivial impact on users' trading strategies, making it crucial to develop refined AMM models that precisely account for their effects. In this work, we extend a foundational model of AMMs by introducing a new parameter, the trading fee $\phi \in (0, 1]$, into the swap rate function. Fee amounts increase inversely proportional to ϕ. When $\phi = 1$, no fee is applied and the original model is recovered. We analyze the resulting fee-adjusted model from an economic perspective. We show that several key properties of the swap rate function, including output-boundedness and monotonicity, are preserved. At the same time, other properties – most notably additivity – no longer hold. We precisely characterize this deviation by deriving a generalized form of additivity that captures the effect of swaps in the presence of trading fees. In particular, we prove that when $\phi < 1$, executing a single large swap yields strictly greater profit than splitting the trade into smaller ones. Finally, we derive a closed-form solution to the arbitrage problem in the presence of trading fees and prove its uniqueness. All results are formalized and machine-checked in the Lean 4 proof assistant.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Smart contracts, Decentralized Finance, Verification, Blockchain

Digital Object Identifier 10.4230/OASIcs.FMBC.2026.4

Related Version *Full Version (Thesis)*: `https://fulltext-gateway.cvt.dk/oafilestore?oid=6865ca59ee615523af43fd05&targetid=6865ca59c998ab10401d3143`

Supplementary Material *Software*: `https://github.com/Mamboleano/lean4-amm-fees` archived at `swh:1:dir:32aac5519a4e9f3d4d3d68159636cf19cfb932f1`

Funding *Massimo Bartoletti*: Partially supported by project SERICS (PE00000014) and PRIN 2022 DeLiCE (F53D23009130001) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.
Alberto Lluch-Lafuente: Partially supported by Copenhagen Fintech project CODeM – Challenges and Opportunities of Defi Models.

Acknowledgements Special thanks to Troels Damgaard, Espen Højsgaard, and Lasse Nisted for fruitful discussions on DeFi fee mechanisms.

1 Introduction

Decentralized Finance (DeFi) has transformed the financial landscape by enabling peer-to-peer transactions without a central intermediary. One of the core mechanisms of many DeFi applications is the Automated Market Maker (AMM), which enable token swaps through algorithmic pricing mechanisms. Among the various kinds of AMMs, the constant-product Market Makers (CPMMs) are the most used ones, with implementations such as Uniswap v2/v3 [4, 5], Sushiswap [2], *etc.* These implementations have a huge economic relevance in the crypto market: for example, as of January 2026, Uniswap stands out with an approximate Total Value Locked (TVL) of $4 billion [18].

While AMMs have been extensively studied, the role of *trading fees* – which is critical in real-world AMM implementations – has often been overlooked in formal analyses of AMM models [9, 21, 22]. Trading fees play a central role in incentivizing liquidity provision, while also significantly influencing users' strategies and overall market dynamics. A formal understanding of how trading fees impact AMM behavior is essential both from a theoretical standpoint and practical applications such as constructing of optimal automatic trading bots.

Contributions

We provide a formal analysis of AMMs incorporating trading fees, extending a Lean 4 library for no-fee AMMs [21]. More specifically, we summarize our main contributions as follows:

1. **Formal Modeling**: We develop a formal model of AMMs that integrates trading fees, capturing their effect on swap outcomes and users' incentives. We introduce this trading fee in the *constant-product* swap rate function, and discuss the similarities between this model and the implementation of the swap in Uniswap v2.
2. **Economic Analysis**: We analyse core economic properties of AMMs with trading fees, such as how they impact user gain on token swaps, arbitrage opportunities and exchange rates. Besides re-proving the structural properties known from the fee-less analysis of [9] in our fee-adjusted setting (Lemmas 4 and 5), we generalise the additivity property of the constant-product swap rate function (Lemma 6). We leverage this generalised additivity result to show that, in the presence of trading fees, a single large swap achieves a strictly greater gain than two smaller swaps (Theorem 8) – whereas the two strategies yield equal gains in fee-less AMMs. Our main economic result is a closed-form solution to the arbitrage problem for fee-adjusted AMMs, namely the problem of determining the swap that maximizes a trader's gain by exploiting discrepancies between the AMM-implied and external exchange rates. More specifically, we start by determining a swap value that brings the AMM to an equilibrium state where its internal exchange rate is aligned with the exchange rate given by an external price oracle (Theorem 9), and establish its uniqueness (Lemma 10). We then show that this value yields a suboptimal gain, determining an interval of swap values that guarantee a strictly higher gain to the sender (Theorem 12). Our main technical result is a closed-formula solution to the arbitrage problem, namely a swap transaction that obtains the maximum gain (Theorem 13). We conclude by showing that this solution is unique (Lemma 14).
3. **Lean 4 Formalization**: We formalize and prove in Lean 4 [12] all the results presented in this work, extending the library presented in [21]. Our formalization is available online at `https://mamboleano.github.io/lean4-amm-fees`, and it consists of ∼3500 lines of Lean code. We additionally provide detailed pen-and-paper proofs, that help outlining the proof strategies implemented in Lean [13].

By providing a machine-checked analysis of the role of trading fees in AMMs, this work contributes to the ongoing effort to improve the robustness and security of DeFi systems. Because of space constraints, we include several ancillary results in [13], and we relegate the discussion of the differences between our model and Uniswap v2 to Appendix A.

2 AMM model

In this Section, we refine the operational model of AMMs in [9] to introduce the trading fees in the swap rate function, which brings this model closer to the Uniswap v2 protocol [4].

We assume a set of *token types* $\tau, \tau', \ldots$ representing crypto-assets, and a set of *accounts* $\mathsf{A}, \mathsf{B}, \ldots$, representing the users involved in token exchanges. An AMM consists of two token reserves, written $\{r_0 : \tau_0, r_1 : \tau_1\}$, where the amounts r_0, r_1 are non-negative reals. Users interact with AMMs by sending swap transactions. Intuitively, a swap transaction $\mathsf{A} : \mathsf{swap}(x, \tau_0, \tau_1)$ represents the atomic action by which the user A pays an amount x of τ_0 to the AMM and receives a quantity $y = x \cdot SX_\phi(x, r_0, r_1)$ of τ_1 in return. A central component of the model is therefore the swap rate function SX_ϕ. Uniswap v2 [3] and other mainstream AMM implementations [1, 2] use the ***constant-product swap rate*** function:

$$SX_\phi(x, r_0, r_1) = \frac{\phi\, r_1}{r_0 + \phi\, x} \qquad \text{where } \phi \in (0, 1] \tag{1}$$

where r_0 and r_1 are, respectively, the reserves of the tokens τ_0 and τ_1 in the current AMM state, and the parameter ϕ is the ***trading fee***. Despite the name "constant-product", swaps preserve the product $r_0 \cdot r_1$ of the AMM reserves only in the fee-less case, i.e. when $\phi = 1$. Instead, when $\phi < 1$, the product of the reserves strictly increases upon each swap. More specifically, if an AMM $\{r_0 : \tau_0, r_1 : \tau_1\}$ evolves into $\{r_0 + x : \tau_0, r_1 - y : \tau_1\}$ upon a swap, then $(r_0 + x)(r_1 - y) > r_0 r_1$. Therefore, the amount y of tokens τ_1 sent to the user performing the swap is reduced compared to the fee-less case.[1] The intuition is that when the transaction $\mathsf{A} : \mathsf{swap}(x, \tau_0, \tau_1)$ is fired, the actual number of tokens that are used to re-balance the AMM pair is not x, but $\phi \cdot x$ instead. This means that A receives fewer tokens and some units of τ_1 token are "kept" by the AMM in order to benefit its liquidity providers (LPs). The incentives to LPs are briefly discussed at the end of this section.

▶ **Example 1.** Consider a system Γ containing an AMM $\{40 : \tau_0, 60 : \tau_1\}$ and user A's wallet holding $30 : \tau_0$ and $20 : \tau_1$. We write such state as $\Gamma = \mathsf{A}[30 : \tau_0,\ 20 : \tau_1] \mid \{40 : \tau_0, 60 : \tau_1\}$. Suppose that A wants to swap $10 : \tau_0$, and that the trading fee in (1) is set to $\phi = 0.997$. Then, A expects to receive $y : \tau_1$, where:

$$y = 10 \cdot \frac{0.997 \cdot 60}{40 + 0.997 \cdot 10} \approx 11.97$$

Hence, firing the transaction $\mathsf{T} = \mathsf{A} : \mathsf{swap}(10, \tau_0, \tau_1)$ triggers a state transition:

$$\Gamma \xrightarrow{\mathsf{T}} \Gamma' = \mathsf{A}[20 : \tau_0,\ 31.97 : \tau_1] \mid \{50 : \tau_0, 48.03 : \tau_1\}$$

By contrast, by firing the same transaction in an AMM with no trading fee, A would obtain:

$$y = 10 \cdot \frac{1 \cdot 60}{40 + 1 \cdot 10} = 10 \cdot \frac{60}{40 + 10} = 12$$

[1] These conditions are also specified in Uniswap v2's implementation [3], see also Listing 5 at page 18.

and the resulting state would be $\Gamma'' = \mathsf{A}[20 : \tau_0,\ 32 : \tau_1] \mid \{50 : \tau_0, 48 : \tau_1\}$. As we can see, in both Γ' and Γ'' the AMM has 50 units of τ_0 tokens. However, in Γ', the AMM retains a small portion of the τ_1 tokens compared to Γ'', which results in a slightly worse outcome for the trader A and a better one for the LPs. ⌟

Token prices and exchange rates

We assume a price oracle that determines the price of each token type. Technically, we model this oracle as a function $P : \mathbb{T}_0 \to \mathbb{R}_{>0}$, where $\mathbb{T}_0$ is the universe of token types.

Based on the prices provided by this price oracle, we define the ***external exchange rate*** $X(\tau_0, \tau_1)$ between two token types τ_0 and τ_1 as the ratio between their prices:

$$X(\tau_0, \tau_1) = \frac{P(\tau_0)}{P(\tau_1)}$$

This represents the number of units of τ_1 tokens a user can buy with 1 unit of τ_0 using e.g. a centralized exchange platform. Note that the external exchange rate only depends on the prices given by the oracle, neglecting the state of AMMs.

The ***internal exchange rate*** $X_\Gamma(\tau_0, \tau_1)$, by contrast, is determined by the swap rate function SX_ϕ and by the reserves of an AMM pair in Γ handling the token pair $\{\tau_0, \tau_1\}$. We assume that each state Γ can contain at most one AMM for each token pair. Its interpretation mirrors that of the external exchange rate: for a *very small* amount of input tokens x, a user swapping x units of τ_0 expects to receive $x \cdot X_\Gamma(\tau_0, \tau_1)$ units of τ_1 tokens. More formally:

$$X_\Gamma(\tau_0, \tau_1) = \lim_{x \to 0} SX_\phi(x, r_0, r_1) \qquad \text{if } \Gamma \text{ contains } \{r_0 : \tau_0, r_1 : \tau_1\}$$

▶ **Example 2.** Let $\Gamma = \mathsf{A}[30 : \tau_0,\ 20 : \tau_1] \mid \{40 : \tau_0, 60 : \tau_1\}$. Suppose that the external prices of τ_0 and τ_1 are $P(\tau_0) = 4$ and $P(\tau_1) = 5$. Hence, the external exchange rate is:

$$X(\tau_0, \tau_1) = \frac{4}{5} = 0.8$$

Assuming that we are using the constant-product swap rate with trading fee $\phi = 0.997$, the internal exchange rate computed in the state Γ is:

$$X_\Gamma(\tau_0, \tau_1) = \lim_{x \to 0} SX_\phi(x, r_0, r_1) = \lim_{x \to 0} SX_\phi(x, 40, 60) = \lim_{x \to 0} \frac{0.997 \cdot 60}{40 + 0.997 \cdot x}$$
$$= \frac{0.997 \cdot 60}{40} = 1.4955$$

Therefore, in this state the external and internal exchange rates are not aligned. We will show later that rational users are incentivized to execute swaps that align internal and external exchange rates, a practice called ***arbitrage***. ⌟

Users' Gain

The ***gain*** of a user A from executing a transaction T in a state Γ, denoted by $G_\mathsf{A}(\Gamma, \mathsf{T})$, is defined as the difference between the wealth of A after and before the transaction is executed. Wealth is measured as the weighted sum of the token balances in A's wallet, where the weights are given by the external token prices.

▶ **Example 3.** Recall the scenario from Example 1. The wealth of A in Γ is $30 \cdot 4 + 20 \cdot 5 = 220$, while that in Γ' is $20 \cdot 4 + 31.97 \cdot 5 = 239.85$. Therefore, A's gain upon performing the transaction $\mathsf{T} = \mathsf{A} : \mathsf{swap}(10, \tau_0, \tau_1)$ in Γ is:

$$G_{\mathsf{A}}(\Gamma, \mathsf{T}) = 239.85 - 220 = 19.85$$

Note that A had a profit from the swap, and the internal exchange rate shifted from $X_\Gamma(\tau_0, \tau_1) = 1.4955$ to $X_{\Gamma'}(\tau_0, \tau_1) \approx 0.958$, making it closer to the external rate $X(\tau_0, \tau_1) = 0.8$. In general, we expect that swaps that reduce the gap between internal and external exchange rate always yield a profit. We will formalize this intuition in Section 3. ⌟

Liquidity provision

We implicitly assume the presence of *minted tokens*. For example, the minted token $\{\tau_0, \tau_1\}$ represents a share of the AMM $\{r_0 : \tau_0, r_1 : \tau_1\}$. Normally, a user would gain these shares by providing liquidity to the AMM; however, here we only model trading users, leaving the analysis of the impact of fees on liquidity providers as possible future work. Hence, in the following statements, we assume that the supply of the minted tokens is always positive (i.e., that at some point liquidity has been provided to the AMM), and that no trader holds all of the minted token supply. These assumptions rule out unrealistic scenarios in which, for example, the trader has complete control over the AMM shares.

3 Economic properties of AMMs with trading fees

In this Section, we study the economic properties of AMMs with trading fees. Our goal is to analyze how trading fees play a role on the incentives of users and the behavior and properties of swap functions, with keen focus on the constant-product swap rate function (1), which is the function used in several real-world AMMs implementations [1, 2, 3].

Properties of the costant-product swap rate function

AMMs using the constant-product swap rate function (hereaafter, referred to as *CPMMs*) are *output-bounded* [9]. Namely, they always have enough output tokens τ_1 to send to the user who performs a $\mathsf{swap}(x, \tau_0, \tau_1)$, for any value of x. In particular, output-boundedness guarantees that the AMM will always hold a strictly positive amount of each token type.

∃∀ ▶ **Lemma 4.** *CPMMs are* output-bounded*, i.e. for all* $x \geq 0$ *and* $r_0, r_1 > 0$*:*

$$x \cdot SX_\phi(x, r_0, r_1) < r_1$$

CPMMs are also *strictly monotonic* [9]. Namely, for a swap action $\mathsf{swap}(x, \tau_0, \tau_1)$, the swap rate (strictly) increases if we (strictly) decrease the input amount x or the reserves of τ_0, or if we (strictly) increase the reserves of τ_1. This monotonicity property plays a crucial role in establishing the existence and optimality of arbitrage transactions.

∃∀ ▶ **Lemma 5.** *CPMMs are* strictly monotonic*, i.e. for* $i \in \{0, 1, 2\}$ *and* $\lhd_i \in \{<, \leq\}$*:*

$$x' \lhd_0 x,\ r'_0 \lhd_1 r_0,\ r_1 \lhd_2 r'_1 \implies SX_\phi(x, r_0, r_1) \lhd_3 SX_\phi(x', r'_0, r'_1)$$

where: $\lhd_3 = \leq$ *if* $\lhd_i = \leq$ *for* $i \in \{0, 1, 2\}$*, and* $\lhd_3 = <$ *otherwise.*

The following property is useful to measure the effect of splitting a single swap in two swaps of smaller amounts. In particular, it determines the factor $\zeta_\phi(x, y, r_0, r_1)$ between the swap rate of a single transaction swapping $x + y$ tokens and the rates α and β of two transactions (swapping first x, then y) in a CPMM with reserves r_0, r_1.

LEAN

▶ **Lemma 6** (Additivity of swap rate). *CPMMs are* additive*, i.e.:*

$$SX_\phi(x + y, r_0, r_1) = \frac{\alpha x + \beta y}{x + y} \cdot \zeta_\phi(x, y, r_0, r_1)$$

where $\alpha = SX_\phi(x, r_0, r_1)$*,* $\beta = SX_\phi(y, r_0 + x, r_1 - \alpha x)$*, and:*

$$\zeta_\phi(x, y, r_0, r_1) = \frac{\Big((\phi r_1 x)(r_0 + \phi x + \phi y) + (\phi r_1 r_0 y)\Big) \cdot (r_0 + x + \phi y)}{(r_0 + \phi x + \phi y) \cdot \Big((\phi r_1 x)(r_0 + x + \phi y) + (\phi r_1 r_0 y)\Big)}$$

In particular, if $\phi = 1$ *then* $\zeta_\phi(x, y, r_0, r_1) = 1$*.*

In CPMMs without trading fees, the gain obtained by swapping $x+y$ tokens is equal to the sum of the gains obtained from two consecutive swaps of x and y tokens, respectively [9]. In the presence of a trading fee, however, this additivity property no longer holds: technically, this follows by the fact that the factor $\zeta_\phi(x, y, r_0, r_1)$ is strictly positive when $\phi < 1$. Intuitively, this implies that a user achieves a strictly higher gain by executing a single large swap rather than splitting the transaction into multiple smaller swaps.

▶ **Example 7.** Let $\Gamma = \mathsf{A}[300 : \tau_0, 200 : \tau_1] \mid \{400 : \tau_0, 600 : \tau_1\}$. Now suppose that the prices of τ_0 and τ_1 are $P(\tau_0) = 4$ and $P(\tau_1) = 5$. Let $\mathsf{T}_0 = \mathsf{A} : \mathsf{swap}(100, \tau_0, \tau_1)$ and let $\phi = 0.997$. By executing T_0 in Γ, A would obtain $y = 100 \cdot SX_\phi(100, 400, 600) = 119.712 : \tau_1$, and so:

$$\Gamma \xrightarrow{\mathsf{T}_0} \Gamma' = \mathsf{A}[200 : \tau_0, 319.712 : \tau_1] \mid \{500 : \tau_0, 480.288 : \tau_1\}$$

Now, suppose that A – rather than firing T_0 – splits this transaction into two transactions, $\mathsf{T}_1 = \mathsf{A} : \mathsf{swap}(40, \tau_0, \tau_1)$ and $\mathsf{T}_2 = \mathsf{A} : \mathsf{swap}(60, \tau_0, \tau_1)$. We would have that:

$$\Gamma \xrightarrow{\mathsf{T}_1} \Gamma_1 = \mathsf{A}[260 : \tau_0, 254.397 : \tau_1] \mid \{460 : \tau_0, 545.603 : \tau_1\}$$
$$\xrightarrow{\mathsf{T}_2} \Gamma_2 = \mathsf{A}[200 : \tau_0, 319.696 : \tau_1] \mid \{500 : \tau_0, 480.304 : \tau_1\}$$

We see that the split did not benefit A, as she received less tokens than what she would have received by firing the transaction T_0 directly instead of splitting it in T_1 and T_2. ⌟

The impact on a user's gain of splitting a large swap into two smaller swaps is characterized by the following theorem:

LEAN

▶ **Theorem 8** (Additivity of swap gain). *Consider a CPMM* $\{r_0 : \tau_0, r_1 : \tau_1\}$ *in state* Γ *and let* $\mathsf{T}(x) = \mathsf{A} : \mathsf{swap}(x, \tau_0, \tau_1)$*. Let* Γ' *and* Γ'' *be such that* $\Gamma \xrightarrow{\mathsf{T}(x_0)} \Gamma' \xrightarrow{\mathsf{T}(x_1)} \Gamma''$*. If* $\phi < 1$ *then there exists* $\varepsilon_\phi > 0$ *such that:*

$$G_\mathsf{A}(\Gamma, \mathsf{T}(x_0 + x_1)) \;=\; G_\mathsf{A}(\Gamma, \mathsf{T}(x_0)) + G_\mathsf{A}(\Gamma', \mathsf{T}(x_1)) + \varepsilon_\phi$$

The term ε_ϕ captures the correction induced by the presence of trading fees. Since its definition is algebraically heavy and does not provide any additional insight, we refer to [13] and to our Lean library for its formalization. What is relevant, though, is that in the presence of a fee ($\phi < 1$), the factor ε_ϕ is strictly positive, reflecting the fact that splitting a swap into multiple transactions results in a decreased gain for the trader. On the other hand, if $\phi = 1$, then also $\varepsilon_\phi = 0$. Therefore, our result is coherent with [9] in the fee-less case.

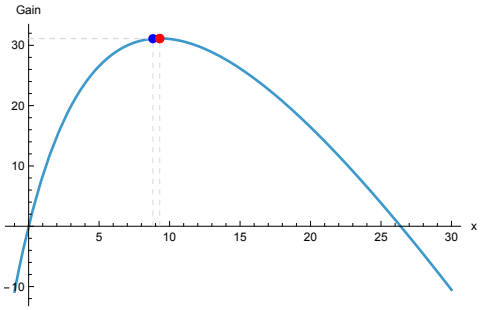

Figure 1 $G_{\mathsf{A}}(\Gamma, \mathsf{T}(x))$.

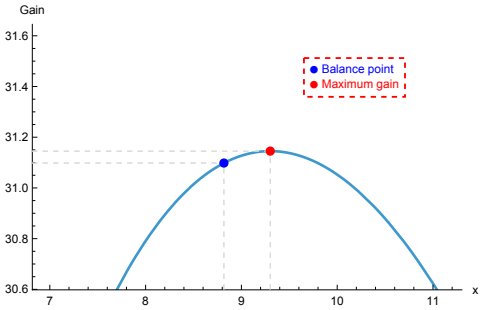

Figure 2 $G_{\mathsf{A}}(\Gamma, \mathsf{T}(x))$ zoomed.

Arbitrage

We now address the problem of finding the optimal amount of tokens that a trader should swap to maximize their gain. In CPMMs without trading fees, the optimal strategy consists in executing a swap that aligns the internal exchange rate to the external one [9], thereby bringing the AMM to a so-called *equilibrium* state. Trying to follow the same strategy in the presence of trading fees, the first step is determining the swap value that brings the AMM to the equilibrium. This is given by the following Theorem:

▶ **Theorem 9** (Equilibrium value). *Consider a CPMM* $\{r_0 : \tau_0, r_1 : \tau_1\}$ *in state* Γ*, and let:*

$$x_0 \;=\; \frac{-\sqrt{P(\tau_0)}r_0(1+\phi) + \sqrt{r_0}\sqrt{P(\tau_0)r_0(-1+\phi)^2 + 4P(\tau_1)r_1\phi^2}}{2\sqrt{P(\tau_0)}\phi} \tag{2}$$

If $\mathsf{T}(x_0) = \mathsf{A} : \mathsf{swap}(x_0, \tau_0, \tau_1)$ *is enabled in* Γ *and* $\phi < 1$*, then* $X(\tau_0, \tau_1) = X_{\Gamma'}(\tau_0, \tau_1)$*, where* Γ' *is the state reached from* Γ *upon executing* $\mathsf{T}(x_0)$*.*

The following lemma states that, when the value x_0 given by Theorem 9 satisfies the enabledness condition, it is the only one that brings the AMM to the equilibrium.

▶ **Lemma 10** (Uniqueness of equilibrium swap value). *Consider a CPMM* $\{r_0 : \tau_0, r_1 : \tau_1\}$ *in state* Γ*, and let* $\mathsf{T}(x) = \mathsf{A} : \mathsf{swap}(x, \tau_0, \tau_1)$*. If* $\mathsf{T}(x_0)$ *and* $\mathsf{T}(x_1)$ *bring the AMM to the equilibrium state from* Γ*, then* $x_0 = x_1$*.*

We now address the problem of determining if the swap value that brings an AMM to the equilibrium also maximizes the user's gain. While this equivalence holds for CPMMs without trading fees [9], the following example shows that it fails when $\phi < 1$.

▶ **Example 11.** Consider a CPMM in the state $\Gamma = \mathsf{A}[30 : \tau_0, 20 : \tau_1] \mid \{10 : \tau_0, 30 : \tau_1\}$. Assume a trading fee $\phi = 0.9$, and external token prices $P(\tau_0) = 4$ and $P(\tau_1) = 5$. Let $\mathsf{T}(x) = \mathsf{A} : \mathsf{swap}(x, \tau_0, \tau_1)$, and let x_0 be the swap amount determined by (2), i.e. the value that brings the AMM to equilibrium. The gain obtained by A from executing $\mathsf{T}(x_0)$ in Γ is:

$$G_{\mathsf{A}}(\Gamma, \mathsf{T}(x_0)) \approx 31.098 \tag{3}$$

In the fee-less setting of [9], this value would coincide with the maximum achievable gain from any swap executed in state Γ. In contrast, this is no longer the case in the presence of trading fees. Figure 1 plots the gain function $G_{\mathsf{A}}(\Gamma, \mathsf{T}(x))$. The function is concave, i.e. it has a critical point which is also the absolute maximum for the function. However, a closer inspection (see Figure 2) reveals that the gain at x_0 (in blue) is not exactly at the top of the curve. Indeed, evaluating the gain at a slightly larger swap amount $x_1 = x_0 + 0.3$ yields:

$$G_{\mathsf{A}}(\Gamma, \mathsf{T}(x_1)) \approx 31.138$$

which is strictly greater than the gain in (3). This example demonstrates that, when $\phi < 1$, the equilibrium-inducing swap does not necessarily maximize the user's gain. ⌟

Although the equilibrium-inducing swap is suboptimal, it remains very close to the optimum. It is therefore still meaningful to compare the gain achieved by this swap amount with that obtained from any other swap value, as formalized by the Theorem below.

∃∀ℕ ▶ **Theorem 12** (Equilibrium value vs gain). *Let* $\Gamma = \mathsf{A}[\sigma_{\mathsf{A}}] \mid \{r_0 : \tau_0, r_1 : \tau_1\} \mid \Delta$ *be such that* $S_\Gamma\{\tau_0, \tau_1\} > 0$ *and* $\sigma_{\mathsf{A}}\{\tau_0, \tau_1\} < S_\Gamma\{\tau_0, \tau_1\}$ *and let* $\mathsf{T}(x) = \mathsf{A} : \mathsf{swap}(x, \tau_0, \tau_1)$. *Let* x_0 *be the value determined in* (2). *If* $\phi < 1$, *then, whenever* $\mathsf{T}(x_0)$ *and* $\mathsf{T}(x)$ *are enabled in* Γ*:*

$$\forall x < x_0 \ : \ G_{\mathsf{A}}(\Gamma, \mathsf{T}(x_0)) \ > \ G_{\mathsf{A}}(\Gamma, \mathsf{T}(x))$$
$$\forall x > x_0 \ : \ \left(G_{\mathsf{A}}(\Gamma, \mathsf{T}(x_0)) \ > \ G_{\mathsf{A}}(\Gamma, \mathsf{T}(x)) \iff x > \frac{x_0}{\phi} \right)$$

This behaviour is consistent with Example 11 and the plots in Figures 1 and 2. Specifically, there exists a small interval immediately to the right of x_0 where A's gain increases relative to that obtained by swapping x_0 tokens. This is the interval $(x_0, {}^{x_0}/_{\phi}]$. Consequently, the remaining task is to identify the swap amount within this interval which maximizes the gain.

∃∀ℕ ▶ **Theorem 13** (Max Gain Value). *Consider a CPMM* $\{r_0 : \tau_0, r_1 : \tau_1\}$ *in a state* Γ *where* A *does not hold all minted tokens* $\{\tau_0, \tau_1\}$. *Let* $\mathsf{T}(x) = \mathsf{A} : \mathsf{swap}(x, \tau_0, \tau_1)$. *Let* x_0 *be a swap value bringing the AMM to equilibrium, and let:*

$$x_{max} = x_0 + \frac{-\sqrt{P(\tau_0)}r_0 - \sqrt{P(\tau_0)}\phi x_0 + \sqrt{P(\tau_1)\phi r_0 r_1}}{\sqrt{P(\tau_0)}\phi} \qquad \begin{array}{l} \text{where } \Gamma \xrightarrow{\mathsf{T}(x_0)} \Gamma' \text{ and} \\ X_{\Gamma'}(\tau_0, \tau_1) = X(\tau_0, \tau_1) \end{array}$$

If $\mathsf{T}(x_{max})$ *is enabled in* Γ *and* $\phi < 1$, *then* $\mathsf{T}(x_{max})$ *maximizes* A*'s gain:*

$$\forall x \neq x_{max} \ : \ G_{\mathsf{A}}(\Gamma, \mathsf{T}(x_{max})) \ > \ G_{\mathsf{A}}(\Gamma, \mathsf{T}(x))$$

Finally, it is natural to ask if x_{max} is the unique value that maximizes the trader's gain. This is established by the following lemma, proved by contradiction.

∃∀ℕ ▶ **Lemma 14** (Optimal swap value uniqueness). *Consider a CPMM* $\{r_0 : \tau_0, r_1 : \tau_1\}$ *in state* Γ *and let* $\mathsf{T}(x) = \mathsf{A} : \mathsf{swap}(x, \tau_0, \tau_1)$. *If* $\phi < 1$ *and* $\mathsf{T}(x_{max})$ *and* $\mathsf{T}(x_{max'})$ *both maximize the gain of* A *from* Γ, *then* $x_{max} = x_{max'}$.

4 Related Work

The literature on Automated Market Makers is extensive, as AMMs are one of the core building blocks of Decentralized Finance [15, 23, 24]. In this section, we focus primarily on comparing our work with the foundational studies on which it builds, as well as with other works that explicitly address fee mechanisms in AMMs.

The work [9] introduces an abstract operational model of AMMs that forms the foundation of our work. While most our formalization and results focus on the constant-product swap rate function, the analysis in [9] deliberately abstracts from any specific instantiation of the swap rate function, instead identifying conditions that guarantee ideal structural properties of AMMs, such as additivity and reversibility of swaps. The main difference between our work and [9] lies in the treatment of trading fees, which are not modelled or analysed in [9]. While the structural properties in [9] are preserved in our model in the fee-less case $\phi = 1$, when $\phi < 1$ some key properties need to be adapted. In particular:

1. **Additivity**: Consider a user executing a swap of $x : \tau_0$ followed by another swap of $y : \tau_0$ on the same AMM. The exchange rates corresponding to these swaps are given by:

$$\alpha = SX_\phi(x, r_0, r_1) \text{ and } \beta = SX_\phi(y, r_0 + x, r_1 - \alpha x)$$

where r_0 and r_1 are the reserves of the AMM pair when the first swap is performed. In the framework of [9], a swap rate function SX_ϕ is defined to be additive if:

$$SX_\phi(x + y, r_0, r_1) = \frac{\alpha x + \beta y}{x + y}$$

When this equation holds, swapping an amount of tokens $(x + y) : \tau_0$ is equivalent to swapping first $x : \tau_0$ and subsequently $y : \tau_0$. Indeed, the output amount of the single swap with input $(x + y) : \tau_0$ is:

$$(x + y) \cdot SX_\phi(x + y, r_0, r_1) = (x + y) \cdot \frac{\alpha x + \beta y}{x + y} = \alpha x + \beta y$$

which is the sum of the output amounts of two subsequent swaps of $x : \tau_0$ and $y : \tau_0$. We have shown in Example 7 that the fee-adjusted constant-product swap rate does not respect this additivity property. Then, in Lemma 6 we devised a generalized additivity notion that takes into account the trading fee, by multiplying the previous formula by the factor $\zeta_\phi(x, y, r_0, r_1)$. Building upon this lemma, Theorem 8 generalises Lemma 5.7 in [9], stating that, the presence of trading fees, performing two subsequent swaps on the same AMM pair yields a *smaller* gain than performing a single large swap.

2. **Reversibility**: Let $\alpha = SX_\phi(x, r_0, r_1)$, where r_0 and r_1 are the current reserves of an AMM pair. The work [9] defines a swap rate function to be reversible if, for all $x > 0$:

$$SX_\phi(\alpha x, r_1 - \alpha x, r_0 + x) = \frac{1}{\alpha}$$

When this condition holds, a swap of $x : \tau_0$ for $y : \tau_1$ immediately followed by a swap of all the received tokens τ_1 in the opposite direction, restores the AMM to the original state. This reversibility property holds in the fee-less setting (Theorem 5.9 in [9]), but it is no longer valid in the presence of a trading fee $\phi < 1$. Since trading fees make the product of the AMM reserves strictly increase after each swap, a trader attempting to reverse the effect of a swap always incurs in a net loss. The absence of reversibility has significant methodological implications. Several results in [9] rely critically on this property, including Theorem 6.3, which establishes the optimality of the arbitrage transaction.
In contrast, our corresponding results (Theorem 13 and Lemma 14) are proved using different proof strategies that do not rely on reversibility of the swap rate function, and are therefore applicable also in the presence of trading fees.

3. **Equilibrium *vs.* Arbitrage**: Recall that an AMM is in an equilibrium state when its internal exchange rate equals the external rate given by a price oracle. In the fee-less setting, the swap amount that brings an AMM to equilibrium coincides with the arbitrage solution, i.e., it is also the value that maximizes the trader's profit (Theorem 6.3 in [9]). This equivalence no longer holds in the presence of a trading fee. We have shown that the equilibrium-inducing swap amount, determined in Theorem 9, differs from the profit-maximising amount, which is instead characterised in Theorem 13. To the best of our knowledge, ours is the first work that, in a fee-adjusted AMM model, formally compares the swap amount that aligns internal to external prices versus the swap amount that maximizes the individual arbitrage profit.

Our Lean 4 implementation builds on the AMM library presented in [21], which formalizes the core functionalities of AMMs and provides machine-checked proofs of key economic properties of CPMMs studied in [9], including the arbitrage solution in the fee-less setting. We extend this library with trading fees and provide machine-checked proofs of the corresponding fee-adjusted economic properties. In contrast, the formalization of [21] also models liquidity provision and withdrawal, enabling the analysis of liquidity providers' incentives.

The work [19] proposes a framework for developing and formally verifying AMMs in the Rocq proof assistant. In this approach, an AMM is modeled as a composition of smart contracts, formalized as Rocq functions on top of ConCert [7], a mechanized framework for blockchain and smart contract verification.

Although both [19] and our work formalise AMM within a proof assistant, the objectives and level of abstraction differ substantially. The formalization in [19] closely follows a concrete AMM implementation (Dexter2) and yields a deployable contract that is provably consistent with its Rocq specification, ensuring functional correctness and implementation-level safety. By contrast, we adopt a more abstract model, aimed at identifying and formally proving properties that must hold for any implementation conforming to the specification. This abstraction allows us to avoid complexities that would arise by trying to prove complex economic theorems in a concrete framework. As a result, while both works model trading fees, only ours analyses their impact on key economic properties such as arbitrage and equilibrium.

A different formalization of AMMs is proposed in [11], within a general Lean 4 library aimed at the formalization of Maximal Extractable Value (MEV). This is a class of attacks where adversaries with transaction sequencing privileges can "extract" value from honest users' transactions by strategically reordering them in the blockchain, potentially front-running or back-running the victim's transactions with adversarial ones [8]. AMMs are notoriously prone to MEV attacks. In particular, the so-called *sandwich attack* allows the adversary to extract value from virtually any swap transaction [25, 10]. The Lean formalization in [11] provides a machine-checked proof of the optimality of sandwich attacks in an AMM model without trading fees. Extending this result to account for trading fees would likely require a substantial amount of additional work, since – even in the fee-less setting – the space of attack strategies that an adversary must consider is surprisingly large.

The work [16] studies "constant-function" AMMs Makers [6], whereas we focus on "constant-product" AMMs, a subclass of this broader family. The model in [16] differs substantially from ours: building on [17], trading fees are analyzed in a stochastic setting with discrete Poisson block arrivals, leading to results that emphasize the long-term impact of fees on arbitrageurs' behavior. By contrast, our analysis focusses on single-swap incentives and equilibrium properties. Finally, unlike our work, the results in [16] are established via pen-and-paper proofs and are not supported by a machine-checked formalization.

Several works address the role of fees in AMMs from perspectives complementary to ours. In particular, [14] investigates the so-called *protocol fee*, namely the portion of the trading fee that is diverted to the protocol and used to generate revenue for governance token holders. That work also analyzes the incentives of both liquidity providers and traders. The modeling assumptions in [14] differ significantly from those adopted here: their framework incorporates features such as sticky liquidity and trade volume, whereas our work focuses on a minimal operational model of AMMs amenable to formal, machine-checked reasoning.

In summary, our work extends this line of research by incorporating trading fees into both an operational AMM model and its Lean 4 implementation, thereby bringing the framework closer to real-world deployments such as Uniswap v2 and contributing to the development of more secure and reliable DeFi systems.

5 Conclusions

We have presented a rigorous and machine-checked analysis of AMMs with trading fees, built on the abstract operational model presented in [9] and its Lean 4 formalization in [21], here extended with the following main contributions. We have started in Section 2 by introducing the trading fee ϕ as a parameter of the constant-product swap rate function (1). We have then shown that, in the presence of a trading fee $\phi < 1$, the product of the AMM reserves strictly increases after a swap, i.e. $(r_0 + x)(r_1 - y) > r_0 r_1$. This implies that the liquidity providers' shares of the AMM also increase compared to the fee-less model.

We have studied the impact of trading fees on key economic properties of AMMs. In particular, we show that the fee-adjusted constant-product swap rate remains *output-bounded* and *strictly monotonic*. These properties ensure that the AMM reserves cannot be fully drained by a swap action and that the swap rate behaves monotonically w.r.t. its inputs (e.g., increasing the input amount in a swap results in an increase of the output amount). We generalise the *additivity* property of [9], showing that a single large swap yields a strictly greater gain than splitting the same total amount into multiple smaller trades (Theorem 8).

Finally, we have investigated the arbitrage problem and how to solve it. We first have determined the swap amount bringing an AMM to equilibrium (Theorem 9). We then have shown that in the presence of fees, this value is not the optimal one, as (Theorem 12), and we find a closed formula for such optimal value (Theorem 13). Additionally, we have established the uniqueness of both the equilibrium value and the optimal one (Lemmas 10 and 14).

All results presented in this work have been formalised and machine-checked in the Lean 4 proof assistant. This guarantees the mathematical correctness of our development and extends the existing AMM library, thereby providing a foundation for future work on the formal verification of AMMs. For readability, we additionally provide (human-checked) pen-and-paper proofs of our results in [13].

Challenges of formalizing AMMs in Lean 4

Although we were able to prove all of the lemmas and theorems above, the current Lean 4 library that implements this model and all the relative proofs has some limitations.

The type used to represent positive amounts in the model, i.e. some $x \in \mathbb{R}_{>0}$, poses significant challenges. This type is defined as a *subtype* of the already existing – and well supported – Reals $\mathbb{R}$. This approach has advantages, such as guaranteeing that the results of some lemmas are strictly positive (for example, it ensures that for an initialized AMM the reserves are always strictly positive), but it also comes with a few challenges. For example, positive reals by definition do not have a zero element, hence they cannot form a ring, since it lacks additive identity and inverse. This might seem a minor issue, but it greatly impacts some of the Lean 4 tactics that are typically used to perform algebraic manipulation (e.g. `ring-nf`). This means that automation in proofs is limited and not always useful.

Another issue that arose during the development of the Lean proofs concerns the use of simplification lemmas. There are a number of already existing simplification lemmas that are not always meant to be used by the Lean 4 simplifier. This can negatively affect proof automation, as the prover may get stuck in infinite rewriting loops caused by inappropriate simplification rules. Investigating more principled criteria for selecting and organizing simplification lemmas could therefore lead to significant improvements in automation.

Finally, in the proof of Theorem 13, we note that a natural proof strategy would be to follow a classical function-analysis argument. In particular, since the gain function is concave, one could establish optimality by showing that a local maximum is also a global maximum.

This reasoning is formalized in Mathlib 4 by the theorem `of_isLocalMaxOn_of_concaveOn`[2]. However, to the best of our knowledge, this approach is currently not directly applicable within our Lean 4 formalization. The problem is that the Mathlib 4 theorem cited before specifically requires the function domain to be an additive commutative monoid, while the gain function in our model is defined in the domain $\mathbb{R}_{>0}$. As a result, the required concavity-based argument cannot be applied directly. While alternative encodings or workarounds may be possible, we preferred to find an alternative proof strategy.

All the challenges described above can be thought of as trade-offs associated with using Lean 4. On the positive side, Lean 4 allows one to formalize models and their properties in a precise and uniform way, and to verify that all stated properties hold without gaps. This makes it very well suited to formalize state-based machines such as AMMs, where it is crucial to check the correctness of the imposed constraints across sequences of transactions. On the other hand, as we previously explained, some of the mathematical libraries might be less mature compared to other proof assistants and may require additional effort to formalize certain proof strategies. While other proof assistants such as Isabelle/HOL [20] provide strong support for real analysis and automation, the present formalization builds directly on an existing Lean 4 development and follows an operational, state-based modeling style that is well supported in Lean. Reproducing the same results in another proof assistant would have required re-implementing the base model from scratch and might have involved different, and currently unexplored, technical challenges.

Discussion and Future Work

We envision the following directions for future research:

1. **Guarded transactions and Slippage Protection**: A natural extension of our model, bringing it closer to the actual implementation of Uniswap v2, would be to formalise the parameters `amountOutMin` and `deadline` of the function `swapExactTokensForTokens` (Appendix A), which are not currently represented in our model. The former enforces a minimum acceptable amount of output tokens to the user executing the swap, while the latter prevents transactions from remaining pending for unbounded period, reducing the risk of adverse price movements and slippage.
2. **Concentrated Liquidity and Multi-Fee tiers**: Uniswap v3 [5] introduces so-called *concentrated liquidity*: instead of depositing liquidity across the whole range of the internal exchange rate, in Uniswap v3 LPs can specify an internal exchange rate range in which their liquidity earn fees. This allows for different strategies for LPs which can increase their gain but also the risk. Another feature that was introduced in Uniswap v3 is the one of multiple fee tiers. Uniswap v2 has a flat fee of 0.3% on every pool, while in Uniswap v3 it is possible to decide on four discrete tiers upon the creation of the pool, which are 0.01%, 0.05%, 0.3% and 1%. Clearly, a higher fee tier benefits the LPs, but could also result in a less attractive pool to trade on for arbitrageurs. Formalizing these features in the current model would bring it closer to recent AMM implementations such as v3.
3. **Incentives for Liquidity Providers**: This work mainly focuses on the arbitrageur's perspective, i.e. determining the optimal swap amount that maximise an arbitrageur's gain. We do not, however, analyse the impact of trading fees on liquidity providers. Namely, while we know that trading fees increase their share of the pool's reserves compared to the fee-less case, a quantitative assessment of this effect and an analysis of trading fees from the liquidity providers' perspective remain open and relevant directions.

[2] `https://leanprover-community.github.io/mathlib4_docs/Mathlib/Analysis/Convex/Extrema.html#IsMaxOn.of_isLocalMaxOn_of_concaveOn`

4. **Lean 4 Automation**: The implementation developed in this work could be significantly simplified by making it more amenable to proof automation. As discussed earlier (see "Challenges of formalizing AMMs in Lean 4" at page 11), automation is currently hindered by several factors. One promising direction would be to replace the subtype of positive real numbers – which is used throughout the library – with $\mathbb{R}$, as well as to systematically review the associated simplification lemmas.

References

1 Mooniswap implementation, 2020. URL: https://github.com/1inch-exchange/mooniswap/blob/02dccfab2ddbb8a409400288cb13441763370350/contracts/Mooniswap.sol.

2 SushiSwap token pair implementation, 2021. URL: https://github.com/sushiswap/sushiswap/blob/94ea7712daaa13155dfab9786aacf69e24390147/contracts/uniswapv2/UniswapV2Pair.sol.

3 Uniswap token pair implementation, 2021. URL: https://github.com/Uniswap/uniswap-v2-core/blob/4dd59067c76dea4a0e8e4bfdda41877a6b16dedc/contracts/UniswapV2Pair.sol.

4 Hayden Adams, Noah Zinsmeister, and Dan Robinson. Uniswap v2 core. https://app.uniswap.org/whitepaper.pdf, 2020. Accessed: January 20, 2026.

5 Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core. https://app.uniswap.org/whitepaper-v3.pdf, 2021. Accessed: January 20, 2026.

6 Guillermo Angeris and Tarun Chitra. Improved price oracles: Constant function market makers. In *ACM Conference on Advances in Financial Technologies (AFT)*, pages 80–91. ACM, 2020. doi:10.1145/3419614.3423251.

7 Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: a smart contract certification framework in Coq. In *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 215–228. ACM, 2020. doi:10.1145/3372885.3373829.

8 K. Babel, P. Daian, M. Kelkar, and A. Juels. Clockwork finance: Automated analysis of economic security in smart contracts. In *IEEE Symposium on Security and Privacy*, pages 622–639. IEEE, 2023. doi:10.1109/SP46215.2023.00036.

9 Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. A theory of Automated Market Makers in DeFi. *Log. Methods Comput. Sci.*, 18(4), 2022. doi:10.46298/LMCS-18(4:12)2022.

10 Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. Maximizing extractable value from Automated Market Makers. In *Financial Cryptography*, volume 13411 of *LNCS*, pages 3–19. Springer, 2022. doi:10.1007/978-3-031-18283-9_1.

11 Massimo Bartoletti, Riccardo Marchesin, and Roberto Zunino. Certifying optimal MEV strategies with Lean. *CoRR*, abs/2510.14480, 2025. doi:10.48550/arXiv.2510.14480.

12 Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, volume 12699 of *LNCS*, pages 625–635. Springer, Cham, 2021. doi:10.1007/978-3-030-79876-5_37.

13 Marco Dessalvi. A Formal Approach to AMM Fee Mechanisms with Lean 4. Master's thesis, Technical University of Denmark, Denmark, 2025. URL: https://fulltext-gateway.cvt.dk/oafilestore?oid=6865ca59ee615523af43fd05&targetid=6865ca59c998ab10401d3143.

14 Robin Fritsch, Samuel Käser, and Roger Wattenhofer. The Economics of Automated Market Makers. In *Advances in Financial Technologies (AFT)*, pages 102–110. ACM, 2022. doi:10.1145/3558535.3559790.

15 Stefan Kitzler, Friedhelm Victor, Pietro Saggese, and Bernhard Haslhofer. Disentangling Decentralized Finance (DeFi) compositions. *ACM Trans. Web*, 17(2):10:1–10:26, 2023. doi:10.1145/3532857.

16 Jason Milionis, Ciamac C. Moallemi, and Tim Roughgarden. Automated Market Making and Arbitrage Profits in the Presence of Fees. In *Financial Cryptography and Data Security*, volume 14744 of *LNCS*, pages 159–171. Springer, Cham, 2025. doi:10.1007/978-3-031-78676-1_9.

17 Jason Milionis, Ciamac C. Moallemi, Tim Roughgarden, and Anthony Lee Zhang. Automated market making and loss-versus-rebalancing. *arXiv preprint arXiv:2208.06046*, 2022. doi:10.48550/arXiv.2208.06046.

18 Fatemeh (Nazanin) Mottaghi and Bertram I. Steininger. Total Value Locked Volatility in DeFi: Empirical Evidence from Market Crashes. https://ssrn.com/abstract=5161601, 2025.

19 Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Formalising decentralised exchanges in Coq. In *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 290–302. ACM, 2023. doi:10.1145/3573105.3575685.

20 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, 2002. doi:10.1007/3-540-45949-9.

21 Daniele Pusceddu and Massimo Bartoletti. Formalizing Automated Market Makers in the Lean 4 Theorem Prover. In *International Workshop on Formal Methods for Blockchains (FMBC)*, volume 118 of *OASIcs*, pages 5:1–5:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/OASIcs.FMBC.2024.5.

22 Margherita Renieri, Letterio Galletta, Alberto Lluch Lafuente, Aleksander Junge, and James Hsin yu Chiang. Netting-based liquidity-saving automated market makers. *Blockchain: Research and Applications*, 2025. doi:10.1016/j.bcra.2025.100361.

23 Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William J. Knottenbelt. SoK: Decentralized Finance (DeFi). In *ACM Conference on Advances in Financial Technologies, (AFT)*, pages 30–46. ACM, 2022. doi:10.1145/3558535.3559780.

24 Jiahua Xu, Nazariy Vavryk, Krzysztof Paruch, and Simon Cousaert. SoK: Decentralized Exchanges (DEX) with automated market maker (AMM) protocols. *ACM Comput. Surv.*, November 2022. doi:10.1145/3570639.

25 Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. High-Frequency Trading on Decentralized On-Chain Exchanges. In *IEEE Symposium on Security and Privacy*, pages 428–445. IEEE, 2021. doi:10.1109/SP40001.2021.00027.

A Comparison with Uniswap v2

In this section we analyze the differences and similarities between the swap used in this work's model, and the swap used in Uniswap v2's implementation. The Uniswap one is slightly more complicated than the one presented in this model, and the main differences are:

1. Uniswap allows to swap both tokens τ_0 and τ_1 at the same time, while our model allows only one atomic token type as input.
2. In Uniswap, when calling the function, the input is not the amount of tokens that the user wants to swap, but instead the maximum amount of tokens that the user expects to receive. The input token amount is then computed internally in the swap function.

Uniswap v2 is architected in two main modules, the *core* and the *periphery*. The core module only contains the minimal and critical contracts, while the periphery holds all the helper and router contracts that integrate and extend the core functionalities. It is worth mentioning that, according to Uniswap v2's guide on implementing a swap in a contract[3], it is recommended not to use the `swap` primitive from the core module, but instead to use the Router[4] from the periphery module. This is because the Router provides a set of methods that allow users to safely swap the desired tokens. For example, in our model, a swap action $\mathsf{A} : \mathsf{swap}(x, \tau_0, \tau_1)$ means that A wants to swap exactly an amount x of τ_0 tokens, and expects

[3] https://docs.uniswap.org/contracts/v2/guides/smart-contract-integration/trading-from-a-smart-contract
[4] https://github.com/Uniswap/v2-periphery/blob/master/contracts/UniswapV2Router02.sol

Listing 1 Method `swapExactTokensForTokens` in Router02.

```
1  function swapExactTokensForTokens(
2          uint amountIn,
3          uint amountOutMin,
4          address[] calldata path,
5          address to,
6          uint deadline
7      ) external virtual override ensure(deadline) returns (uint[] memory amounts) {
8          amounts = UniswapV2Library.getAmountsOut(factory, amountIn, path);
9          require(amounts[amounts.length - 1] >= amountOutMin, 'UniswapV2Router:
               INSUFFICIENT_OUTPUT_AMOUNT');
10         TransferHelper.safeTransferFrom(
11             path[0], msg.sender, UniswapV2Library.pairFor(factory, path[0], path[1]),
                   amounts[0]
12         );
13         _swap(amounts, path, to);
14 }
```

$y = x \cdot SX_\phi(x, r_0, r_1)$ of τ_1 tokens as output. For this scenario, the Router provides a function `swapExactTokensForTokens` which does exactly what we want. To help the reader understand the similarities and differences from our implementation to the one of Uniswap v2, suppose that we want to perform a swap $\mathsf{A} : \mathsf{swap}(x, \tau_0, \tau_1)$ in a state $\Gamma = \{r_0 : \tau_0, r_1 : \tau_1\} \mid \Delta$. Then, we explain how this swap is executed using Uniswap Router02's `swapExactTokensForTokens`, analyzing the variables and function calls being made, up until the swap primitive presented in Listing 5.

Listing 1 shows `swapExactTokensForTokens`'s implementation. In particular, we can see that it takes as input:

1. `amountIn`: this corresponds to our x in the swap
2. `amountOutMin`: this specifies the minimum amount of output tokens that the user A expects. In this work's model, we assume that this value is zero, as the rule [Swap] does not express a minimum amount of tokens that the user can expect upon a swap.
3. `path`: this is the array of token addresses that the trade goes through. The first of these is the input token, and the last one is the output token. So, in our case this is going to be simply $[\tau_0, \tau_1]$.
4. `to`: this is the address that receives the final output tokens. In our case, this address coincides with A, the user performing the swap.
5. `deadline`: this is a timestamp after which the router reverts the swap if it has not been included in a block yet. This is mainly to prevent bad exchange rates if the transaction takes too long to be approved. Since the current model does not represent the transaction mempool, this feature is omitted.

To better understand what `swapExactTokensForTokens` does in its body, we first explain how the output amount y is computed, then break down how the transfer of the input tokens amount x happens, and finally analyze the swap primitive call.

A.1 Output amount

In line 8, the function calls the library function `getAmountsOut` [5], the code of which is reported in Listing 2. This function computes an array of amounts of output tokens depending on how many there are specified in the parameter `path`, where the first element of the array is the input token amount `amountIn`. In our case, since we are trading in the pool $\{r_0 : \tau_0, r_1 : \tau_1\}$ and we

[5] https://github.com/Uniswap/v2-periphery/blob/master/contracts/libraries/UniswapV2Library.sol#L62-L70

Listing 2 Method `getAmountsOut`.

```
function getAmountsOut(address factory, uint amountIn, address[] memory path) internal
    view returns (uint[] memory amounts) {
        require(path.length >= 2, 'UniswapV2Library: INVALID_PATH');
        amounts = new uint[](path.length);
        amounts[0] = amountIn;
        for (uint i; i < path.length - 1; i++) {
            (uint reserveIn, uint reserveOut) = getReserves(factory, path[i], path[i +
                1]);
            amounts[i + 1] = getAmountOut(amounts[i], reserveIn, reserveOut);
        }
    }
```

already explained that `path` $= [\tau_0, \tau_1]$, we will have `amounts[0]` $=$ `amountIn` and `amounts[1]` $=$ `getAmountOut(amounts[0], reserveIn, reserveOut)` where `reserveIn` and `reserveOut` are exactly our r_0 and r_1.

The function `getAmountOut` [6](Listing 3) is where the constant product swap rate is implemented. In comparison with our model, this returns exactly an amount $y = x \cdot SX_\phi(x, r_0, r_1)$ of τ_1 tokens. Analyzing the code line-by-line, the variables being computed are `amountInWithFee` $= x \cdot 997$, `numerator` $= x \cdot 997 \cdot r_1$, `denominator` $= r_0 \cdot 1000 + x \cdot 997$ and `amountOut` $= y$. Putting all together:

$$y = \frac{\texttt{numerator}}{\texttt{denominator}} = \frac{x \cdot 997 \cdot r_1}{r_0 \cdot 1000 + x \cdot 997} = \frac{1000 \cdot (x \cdot 0.997 \cdot r_1)}{1000 \cdot (r_0 + 0.997 \cdot x)} = \frac{x \cdot 0.997 \cdot r_1}{r_0 + 0.997 \cdot x}$$

which is exactly the constant product swap rate from Definition 1 with $\phi = 0.997$, the trading fee hard-coded in Uniswap v2.

Going back to line 8 of Listing 1, `amounts` will have the input amount x for the first element, and the output amount $y = x \cdot SX_\phi(x, r_0, r_1)$ that we just computed as the second one. In line 9, this safety check ensures that the output amount y is at least what the user A expects to receive as a minimum amount of output tokens. If not, the transaction is reverted.

A.2 Tokens transfer

In lines 10-11, we see the transfer of x τ_0 tokens into the contract of the pair $\{\tau_0, \tau_1\}$ before calling the function `_swap`. This is done because the `_swap` function expects the input tokens to be already transferred into the pair. The code of this function is in Listing 4. This is the function that prepares the parameters to call the `swap` primitive, illustrated in Listing 5. In particular, we can see that in line 6 the function instantiates `amount0Out` and `amount1Out`, which will be the parameters passed to the `swap` primitive. In this case, depending on the ordering of the tokens τ_0 and τ_1 in the AMM pair, one of `amount0Out` and `amount1Out` will be instantiated with our output amount y, and the other one to zero, since we do not expect two output amounts, but only one of τ_1 tokens. For simplicity, suppose from now on that in our case `amount0Out` $= 0$, since we are swapping τ_0 tokens.

Next, since we are dealing with only one token pair, when the function computes `to`, it is immediately set to `_to` (which in our case is A's wallet). The final line represents the call to the `swap` primitive, where

1. `amount0Out` is zero, as already explained, since we are swapping τ_0 tokens, and we don't expect to receive any τ_0 tokens as output.
2. `amount1Out` is equal to $y = SX_\phi(x, r_0, r_1)$.
3. `to` represents A's wallet.

[6] https://github.com/Uniswap/v2-periphery/blob/master/contracts/libraries/UniswapV2Library.sol#L43-L50

Listing 3 Method `getAmountOut`.

```
function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) internal pure
    returns (uint amountOut) {
        require(amountIn > 0, 'UniswapV2Library: INSUFFICIENT_INPUT_AMOUNT');
        require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library:
            INSUFFICIENT_LIQUIDITY');
        uint amountInWithFee = amountIn.mul(997);
        uint numerator = amountInWithFee.mul(reserveOut);
        uint denominator = reserveIn.mul(1000).add(amountInWithFee);
        amountOut = numerator / denominator;
    }
```

Listing 4 Method `_swap` in Router02.

```
function _swap(uint[] memory amounts, address[] memory path, address _to) internal
    virtual {
        for (uint i; i < path.length - 1; i++) {
            (address input, address output) = (path[i], path[i + 1]);
            (address token0,) = UniswapV2Library.sortTokens(input, output);
            uint amountOut = amounts[i + 1];
            (uint amount0Out, uint amount1Out) = input == token0 ? (uint(0), amountOut)
                : (amountOut, uint(0));
            address to = i < path.length - 2 ? UniswapV2Library.pairFor(factory, output
                , path[i + 2]) : _to;
            IUniswapV2Pair(UniswapV2Library.pairFor(factory, input, output)).swap(
                amount0Out, amount1Out, to, new bytes(0)
            );
        }
    }
```

A.3 Swap primitive and Invariant check

Finally, we investigate the function `swap` in Listing 5. As previously stated, this function is slightly complicated, so we only want to give an intuition on its behavior for our example case. We begin by locating where our input and output amounts x and y are located/computed inside the function:

1. The input amount x is computed inside the function in line 18, called `amount0In`. Remember that inside the Router's `swapExactTokensForTokens` we have already performed a safe transfer of this input amount x. Hence, the function can simply compute `amount0In` as new reserves of τ_0 called `balance0` minus the old reserves, called `_reserve0`. Intuitively, since we did not transfer any τ_1 token to the pair, then `amount1In` is zero, as expected.
2. The output amount y is passed as the parameter `amount1Out` like previously said. Also, remember that the other parameter `amount0Out` is zero, from the computations made in the function `_swap`.

There is only one thing left to check: the invariant. So far, we have not found anything that checks that $(r_0 + x)(r_1 - y) \geq r_0 r_1$. This is done in an even stricter way inside the `swap` function. In line 22-23 the function declares `balance0Adjusted` and `balance1Adjusted`, which are computed as:

$$\texttt{balance0Adjusted} = (r_0 + x) \cdot 1000 - 3x$$

$$\texttt{balance1Adjusted} = (r_1 - y) \cdot 1000 - 3 \cdot 0 = (r_1 - y) \cdot 1000$$

Recall that at this point of the contract, `balance0` and `balance1` are respectively r_0 and r_1 after the transfers have occurred, i.e. $r_0 + x$ and $r_1 - y$. Then in line 24, the `require` specifies the invariant check:

$$\texttt{balance0Adjusted} \cdot \texttt{balance1Adjusted} \geq r_0 r_1 \cdot 1000^2$$

Listing 5 Uniswap v2's method `swap`.

```
function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data)
    external lock {
    require(amount0Out > 0 || amount1Out > 0, 'UniswapV2: INSUFFICIENT_OUTPUT_AMOUNT');
    (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
    require(amount0Out < _reserve0 && amount1Out < _reserve1, 'UniswapV2:
        INSUFFICIENT_LIQUIDITY');

    uint balance0;
    uint balance1;
    { // scope for _token{0,1}, avoids stack too deep errors
    address _token0 = token0;
    address _token1 = token1;
    require(to != _token0 && to != _token1, 'UniswapV2: INVALID_TO');
    if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically
        transfer tokens
    if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically
        transfer tokens
    if (data.length > 0) IUniswapV2Callee(to).uniswapV2Call(msg.sender, amount0Out,
        amount1Out, data);
    balance0 = IERC20(_token0).balanceOf(address(this));
    balance1 = IERC20(_token1).balanceOf(address(this));
    }
    uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 -
        amount0Out) : 0;
    uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 -
        amount1Out) : 0;
    require(amount0In > 0 || amount1In > 0, 'UniswapV2: INSUFFICIENT_INPUT_AMOUNT');
    { // scope for reserve{0,1}Adjusted, avoids stack too deep errors
    uint balance0Adjusted = balance0.mul(1000).sub(amount0In.mul(3));
    uint balance1Adjusted = balance1.mul(1000).sub(amount1In.mul(3));
    require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1).
        mul(1000**2), 'UniswapV2: K');
    }

    _update(balance0, balance1, _reserve0, _reserve1);
    emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
}
```

At first sight this check might seem very different from the one of our model. However, we can show that it is almost identical:

$$
\begin{aligned}
& \texttt{balance0Adjusted} \cdot \texttt{balance1Adjusted} \geq r_0 r_1 \cdot 1000^2 \\
\iff & ((r_0 + x) \cdot 1000 - 3x)((r_1 - y) \cdot 1000) \geq r_0 r_1 \cdot 1000^2 \\
\iff & (r_0 + x - 0.003x)(r_1 - y) \geq r_0 r_1 \\
\iff & (r_0 + 0.997x)(r_1 - y) \geq r_0 r_1 \\
\iff & (r_0 + \phi x)(r_1 - y) \geq r_0 r_1 \qquad \text{with } \phi = 0.997
\end{aligned}
$$

This check is even more strict than the one shown in Section 2, as it only takes into account the fee-reduced input ϕx. This check clearly implies the previous one, as $r_0 + \phi x < r_0 + x$.