Formalizing Automated Market Makers in the Lean 4 Theorem Prover

Daniele Pusceddu
ETH Zurich, Switzerland
University of Cagliari, Italy

Massimo Bartoletti
University of Cagliari, Italy

Abstract

Automated Market Makers (AMMs) are an integral component of the decentralized finance (DeFi) ecosystem, as they allow users to exchange crypto-assets without the need for trusted authorities or external price oracles. Although these protocols are based on relatively simple mechanisms, e.g. to algorithmically determine the exchange rate between crypto-assets, they give rise to complex economic behaviours. This complexity is witnessed by the proliferation of models that study their structural and economic properties. Currently, most of theoretical results obtained on these models are supported by pen-and-paper proofs. This work proposes a formalization of constant-product AMMs in the Lean 4 Theorem Prover. To demonstrate the utility of our model, we provide mechanized proofs of key economic properties like arbitrage, that at the best of our knowledge have only been proved by pen-and-paper before.

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

Keywords and phrases Smart contracts, Ethereum, Verification, Blockchain

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.5

Supplementary Material *Software*: `https://github.com/danielepusceddu/lean4-amm/tree/paper`, archived at `swh:1:dir:090eb97d11848e5615172481ca6d55537974261c`

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.

1 Introduction

Automated Market Makers (AMMs) are one of the key applications in the Decentralized Finance (DeFi) ecosystem, as they allow users to trade crypto-assets without the need for trusted intermediaries [13]. Unlike traditional order-book exchanges, where buyers and sellers must find a counterpart, AMMs enable traders to autonomously swap assets deposited in liquidity pools contributed by other users, who are incentivized to provide liquidity by a complex reward mechanism. At the time of writing, there are multiple AMM protocols controlling several billions of dollars worth of assets[1]. This has made AMMs an appealing target for attacks, resulting in losses worth billions of dollars over time[2].

The security of AMMs depends on several factors: besides the absence of traditional programming bugs, it is crucial that their economic mechanism gives rise to a rational behaviour of its users that aligns with the AMM ideal functionality, i.e. providing an algorithmic exchange rate coherent with the one given by trusted price oracles. Therefore, it

[1] `https://defillama.com/protocols/Dexes`
[2] `https://chainsec.io/defi-hacks/`

is important to obtain strong guarantees about the economic mechanisms of these protocols. While formal verification tools for smart contracts based on model-checking are useful in detecting programming bugs and even in proving some structural properties of AMMs [6, 8], they are not suitable for verifying, or even expressing more complex properties regarding the economic mechanism of AMMs. These economic mechanisms have been studied in several research works, which, in most cases, provide pen-and-paper proofs of the obtained properties. Given the complexity of the studied models, it would be desirable to also provide machine-verified proofs, so that we may rely on the proven properties beyond any reasonable doubt. To the best of our knowledge, existing mechanized formalizations [11] focus on verifying relevant structural properties of AMMs like their state consistency, and not on studying the economic mechanism of AMMs (see Section 4 for a detailed comparison).

Contributions

In this paper we formalize Automated Market Makers in the Lean 4 theorem prover. Our model is based on a slightly simplified version of the Uniswap v2 protocol (one of the leading AMMs), which was studied in [5] with a pen-and-paper formalization. We provide a Lean specification of blockchain states, abstracted from any factors that are immaterial to the study of AMMs. Then, we model the fundamental interactions that users may have with AMMs as well as the economic notions of price, networth and gain. Finally, we build machine-checked proofs of economic properties of constant-product AMMs. In particular, we derive an explicit formula for the economic gain obtained by a user after an exchange with an AMM. Building upon this formula we prove that, from a trader's perspective, aligning a constant-product AMM's internal exchange rate with the rate given by the trader's price oracle implies the optimal gain from that AMM. This results in the fundamental property of AMMs acting as price oracles themselves [1]. We then construct the optimal swap transaction that a rational user can perform to maximize their gain, solving the arbitrage problem. Our formalization and proofs[3] are made available in a public GitHub repository. At the best of our knowledge, this is the first mechanized formalization of the economic mechanism of AMMs. We finally discuss some open issues, and alternative design choices for formalizing AMMs.

2 Formalization

An Automated Market Maker implements a decentralized exchange between two different token types. The exchange rate is determined by a smart contract, which also takes care of performing the exchange itself: namely, the contract receives from a trader some amount of the input token type, and sends back the correct amount of the output token type, which is taken from the AMM reserves. A single smart contract can control many instances of AMMs (also called *AMM pairs*): we may have a pair for each possible unordered pair of token types. To create an AMM instance, a user must provide the initial liquidity for the reserves of that pair of tokens. Liquidity providers are rewarded with a type of token that specifically represents shares in that AMM's reserves: we call these *minted* token types, while any other token type will be called *atomic*.

[3] `https://github.com/danielepusceddu/lean4-amm`

Blockchain state

We begin by formalizing the blockchain state, abstracting from the details that are immaterial to the study of AMMs. Then, our model includes the users' wallets, the AMMs and their reserves (see Listing 1). We formalize the universes of users and atomic token types as the types `A` and `T`, respectively, as structures that encapsulate a natural number. Hereafter, we use $a, b, \ldots$ to denote users in `A`, and $\tau, \tau_0, \tau_1, \ldots$ to denote tokens in `T`. Minted token types are pairs of `T`. We represent the funds owned by a user by a *wallet* that maps token types to non-negative reals. To rule out wallets with infinite tokens, we use Mathlib's finitely supported functions[4]: in general, given any type α and any type M with a 0 element, $f \in \alpha \to_0 M$ if $\operatorname{supp}(f) = \{x \in \alpha \mid f(x) \neq 0\}$ is finite.

We define the type $\mathtt{W}_0$ of wallets of atomic tokens as a structure encapsulating $\mathtt{T} \to_0 \mathbb{R}_{\geq 0}$. This definition induces an element $0 \in \mathtt{W}_0$ such that $\operatorname{supp}(0) = \emptyset$: this is the *empty wallet*, which enables us to form the type $\mathtt{T} \to_0 \mathtt{W}_0$. We define the type $\mathtt{W}_1$ of wallets of minted tokens as a structure that encapsulates a function $\mathtt{bal} \in \mathtt{T} \to_0 \mathtt{W}_0$. The intuition is that $\mathtt{bal}\ \tau_0\ \tau_1$ gives the owned amount of the minted token type created by the AMM pair with tokens τ_0 and τ_1. Consistently, the function `bal` must satisfy two conditions: $\mathtt{bal}\ \tau_0\ \tau_1 = \mathtt{bal}\ \tau_1\ \tau_0$, meaning that the order of atomic tokens is irrelevant, and $\mathtt{bal}\ \tau\ \tau = 0$, meaning that the two token types in an AMM must be distinct. Our definition of $\mathtt{W}_1$ encapsulates proofs of these two properties, called `unord` and `distinct`, respectively.

We map users to their wallets with the types $\mathtt{S}_0$ and $\mathtt{S}_1$, which account for the atomic tokens and for the minted tokens, respectively. Finally, we formalize sets of AMM pairs with the type `AMMs`. The definition is strikingly similar to that of $\mathtt{W}_1$, but with a changed constraint. The intuition is that $\mathtt{res}\ \tau_0\ \tau_1$ gives the reserves of τ_0 in the AMM pair (τ_0, τ_1), while $\mathtt{res}\ \tau_1\ \tau_0$ gives the reserves of τ_1 in the same AMM pair. For uninitialized AMM pairs, the obtained reserves must be 0. The property `posres` ensures that either an AMM pair has no reserves of both token types (i.e., the AMM has not been created yet), or both token types have strictly positive reserves (i.e., one cannot deplete the reserves of a single token type in an AMM pair). We combine the previous definitions in the type Γ, which represents the state of a blockchain (note that Γ abstracts from all the details immaterial for AMMs).

Token supply

Given a blockchain state $s \in \Gamma$, we define the supply of an atomic token type τ_0 as:

$$\mathtt{atomsupply}_s(\tau_0) \quad = \sum_{a \in \operatorname{supp}(s.\mathtt{atoms})} (s.\mathtt{atoms}\ a)\ \tau_0 \ + \sum_{\tau_1 \in \operatorname{supp}(s.\mathtt{amms}\ \tau_0)} (s.\mathtt{amms}\ \tau_0)\ \tau_1$$

where the partial application $s.\mathtt{amms}\ \tau_0$ gives a map with all AMM pairs with τ_0 as one of their token types. We define the supply of a minted token type (τ_0, τ_1) as follows:

$$\mathtt{mintsupply}_s(\tau_0, \tau_1) \quad = \sum_{a \in \operatorname{supp}(s.\mathtt{mints})} (s.\mathtt{mints}\ a)\ \tau_0\ \tau_1$$

The corresponding Lean definitions (in Listing 2) have been split in order to facilitate theorem proving and, in particular, the use of Lean's simplifier.

[4] https://leanprover-community.github.io/mathlib4_docs/Mathlib/Data/Finsupp/Defs.html

Listing 1 Fundamental Lean definitions for the state of an AMM system.

```
structure A where
  n: ℕ

structure T where
  n: ℕ

structure W₀ where
  bal: T →₀ ℝ≥0

structure S₀ where
  map: A →₀ W₀
```

```
structure W₁ where
  bal: T →₀ W₀
  unord: ∀ (τ₀ τ₁:
    T),
    bal τ₀ τ₁ = f τ₁
    τ₀
  distinct: ∀ (τ: T),
    bal τ τ = 0

structure S₁ where
  map: A →₀ W₁
```

```
structure AMMs where
  res: T →₀ W₀
  distinct: ∀ (τ: T),
    res τ τ = 0
  posres: ∀ (τ₀ τ₁: T),
    res τ₀ τ₁ ≠ 0  ↔ f τ₁ τ₀ ≠
     0

structure Γ where
  atoms: S₀
  mints: S₁
  amms: AMMs
```

Listing 2 Supply of atomic token types and of minted token types.

```
noncomputable def S₀.supply (s: S₀) (τ: T): ℝ≥0 := s.map.sum (λ _ w => w τ)

noncomputable def S₁.supply (s: S₁) (τ₀ τ₁: T): ℝ≥0 :=
    s.map.sum (λ _ w => w.get τ₀ τ₁)

noncomputable def AMMs.supply (amms: AMMs) (τ: T): ℝ≥0 :=
    (amms.res τ).sum λ _ x => x

noncomputable def Γ.atomsupply (s: Γ) (τ: T): ℝ≥0 :=
    (s.atoms.supply τ) + (s.amms.supply τ)

noncomputable def Γ.mintsupply (s: Γ) (τ₀ τ₁: T): ℝ≥0 := s.mints.supply τ₀ τ₁
```

AMM reserves

Given a blockchain state s and two token types τ_0, τ_1, the terms s.amms τ_0 τ_1 and s.amms τ_1 τ_0 denote, respectively, the reserves of τ_0 and τ_1 in the AMM pair (τ_0, τ_1). This way of accessing the AMM reserves is a bit impractical: when writing proofs, using s.amms τ_0 τ_1 carries an obligation to provide the functions distinct and posres. In particular, this requires the user to explicitly add, in any theorem using the reserves, the assumption that the reserves are strictly positive to indicate the AMM pair has been created. Furthermore, this way of accessing the reserves hides the fact that when one of the reserves is strictly positive, also the other one is such, which again should be made explicit when writing proofs.

To cope with these issues, we build a Lean API that allows for hiding these implementation details (see Listing 3). For example, given an AMM pair (τ_0, τ_1) in a state s, the expression s.amms.r$_0$ τ_0 τ_1 init gives the reserves of token τ_0 in the AMM, which are guaranteed to be strictly positive under the initialization precondition init $\in$ (s.amms.init).

Transactions

Our model encompasses all the main types of transactions supported by AMMs: creating an AMM, adding/removing liquidity, and swapping a token for another. Swaps are parameterised by a *swap rate function*, which determines the exchange rate. We use the formalization of swap transactions (Listing 4) to exemplify the scheme we used for all the transaction types.

Listing 3 Fragment of the AMM API: AMM reserves.

```
def AMMs.init (amms: AMMs) (τ₀ τ₁: T): Prop := amms.res τ₀ τ₁ ≠ 0

def AMMs.r₀ (amms: AMMs) (τ₀ τ₁: T) (h: amms.init τ₀ τ₁): ℝ>0 := ⟨ amms.res τ₀ τ₁,
    by unfold init at h; exact NNReal.neq_zero_imp_gt h ⟩

def AMMs.r₁ (amms: AMMs) (τ₀ τ₁: T) (h: amms.init τ₀ τ₁): ℝ>0 := ⟨ amms.res τ₁ τ₀,
    by unfold init at h; exact NNReal.neq_zero_imp_gt ((amms.posres τ₀ τ₁).mp h) ⟩
```

The type $\texttt{Swap}\,(sx, s, a, \tau_0, \tau_1, x)$ represents valid swap transactions in a blockchain state s, with the swap rate function sx, performed by user a to exchange x amount of the input token τ_0 for a certain amount of the output token τ_1 (Line 2). Each element of this type is a structure containing a proof of the validity of the transaction. For example, for swap transactions we must prove that the user has enough amount of τ_0 (condition `enough`), that the AMM pair with tokens τ_0 and τ_1 exists (condition `exi`), and it has enough reserves of τ_1 to give as output (condition `nodrain`). Since the type $\texttt{Swap}\,(\cdots)$ is empty when the parameters do not satisfy the above conditions, invalid transactions are not really expressible in our model. Instead, if $\texttt{Swap}\,(\cdots)$ represents a valid transaction, it will be a singleton type due to proof irrelevance (i.e., any two proofs of the same proposition are equal).

Each transaction is equipped with an `apply` function that yields the state reached by executing the transaction in the given state. For example, for $sw \in \texttt{Swap}\,(sx, s, a, \tau_0, \tau_1, x)$, `apply` sw yields a state where:

- a's atomic tokens wallet is updated by removing x units of τ_0 and adding $sw.\texttt{y}$ units of τ_1 (Line 12), where $sw.\texttt{y}$ is the amount of tokens outputted by the AMM pair;
- accordingly, the AMM reserves are updated by removing $sw.\texttt{y}$ units of τ_1 and adding x units of τ_0 (Line 14);
- the minted token wallets is unchanged (Line 13).

These definitions use functions and proofs not included in Listing 4 for brevity, such as `sub` and `sub_r1`. These are designed with the same spirit of allowing only valid operations, and so require suitable proofs. For example, `sub_r1` requires a proof of the existence of the AMM we are removing liquidity from, and a proof that the AMM pair has enough liquidity to retain a positive amount of reserves. We build these proofs inline using those contained in the structure: for instance, the parameter $sw.\texttt{exi}$ passed to `sub_r1` at line 11 is a proof that the AMM pair exists. Then, at line 16 we define the constant-product swap rate function, that is the swap rate function used by Uniswap v2. From Lemma 5 onwards, our results will focus on AMMs using this swap rate function.

Price, networth and gain

An important aim of our model is to state and prove economic properties of AMMs related to the networth of their users. The fundamental definitions are in Listing 5. Given a wallet $w \in \texttt{W}_0$ and an atomic token price oracle $o \in \texttt{T} \to \mathbb{R}_{>0}$, we define the *value* of w in Line 2 as:

$$\texttt{value}\,(w, o) = \sum_{\tau \in \mathrm{supp}(w)} w\,(\tau) \cdot o\,(\tau)$$

The value of a wallet of minted tokens $w \in \texttt{W}_1$ is defined similarly, except that:

- the summation ranges over $\mathrm{supp}\,(w.\texttt{u})$, with $w.\texttt{u} \in \texttt{T}^2 \to_0 \mathbb{R}_{\geq 0}$ representing the uncurrying of w;
- the summation is divided by 2 since, if (τ_0, τ_1) is in the support of w, then also (τ_1, τ_0) is in its support.

Listing 4 Definition of the swap transaction type and of its application, as well as the constant-product swap rate function.

```
abbrev SX := ℝ>0 → ℝ>0 → ℝ>0 → ℝ>0
structure Swap (sx: SX) (s: Γ) (a: A) (τ₀ τ₁: T) (x: ℝ>0) where
  enough: x ≤ s.atoms.get a τ₀        -- user a has at least x τ₀
  exi: s.amms.init τ₀ τ₁              -- AMM pair τ₀ τ₁ exists in s
  nodrain: x*(sx x (s.amms.r₀ τ₀ τ₁ exi) (s.amms.r₁ τ₀ τ₁ exi))
           < (s.amms.r₁ τ₀ τ₁ exi)   -- AMM has enough output tokens

def Swap.y (sw: Swap sx s a τ₀ τ₁ x): ℝ>0 :=
    x*(sx x (s.amms.r₀ τ₀ τ₁ sw.exi) (s.amms.r₁ τ₀ τ₁ sw.exi))

noncomputable def Swap.apply (sw: Swap sx s a τ₀ τ₁ x): Γ := {
  atoms := (s.atoms.sub a τ₀ x sw.enough).add a τ₁ sw.y,
  mints := s.mints,
  amms  := (s.amms.sub_r₁ τ₀ τ₁ sw.exi sw.y sw.nodrain).add_r₀ τ₀ τ₁ (by
    simp[sw.exi]) x }

noncomputable def SX.constprod: SX := λ (x r₀ r₁: ℝ+) => r₁/(r₀ + x)
```

For uniformity with the definition of value of atomic wallets, also here we assume an oracle that gives the price of (minted) tokens. However, while for pricing atomic tokens we indeed resort to an oracle, for minted tokens this oracle is instantiated to a specific function, coherently with [5]:

$$\texttt{mintedprice}_s(o, \tau_0, \tau_1) = \frac{(s.\texttt{amms.r}_0\ \tau_0\ \tau_1) \cdot o(\tau_0) + (s.\texttt{amms.r}_1\ \tau_0\ \tau_1) \cdot o(\tau_1)}{\texttt{mintsupply}_s(\tau_0, \tau_1)}$$

where we have omitted the initialization precondition `h` for brevity.

We then define the *networth* of a user as the sum of the value of their two types of wallets (Line 12). The *gain* of a user upon an update of the blockchain state is the difference between the networth in the new state and that in the old state (Line 16).

Reachable states

To formalize reachable states, we begin by defining sequences of transactions (Listing 6). `Tx` (sx, s, s') is the type of sequences of valid transactions (of any kind) starting from state s and leading to s'. The parameter sx is the swap rate function used in swap transactions. Technically, `Tx` (sx, s, s') is an instance of the indexed family of dependent types `Tx`, dependent on sx and s, and indexed by s'. In practice, this means that the constructors must preserve the values of sx and s (building upon the sequence of transactions does not change the swap rate function being used nor the originating state), while s' may change after each construction (the state resulting from the sequence changes with each transaction that is added to it). A state s' is *reachable* if there exists a valid sequence of transactions that reaches s' starting from a valid *initial* state s, i.e. a state with no initialized AMMs or minted token types in circulation.

Listing 5 Users' networth and gain.

```
noncomputable def W₀.value (w: W₀) (o: T → ℝ>0): ℝ≥0 :=
  w.sum (λ τ x => x*(o τ))

noncomputable def W₁.value (w: W₁) (o: T → T → ℝ≥0): ℝ≥0 :=
  (w.u.sum (λ p x => x*(o p.fst p.snd))) / 2

noncomputable def Γ.mintedprice (s: Γ) (o: T → ℝ>0) (τ₀ τ₁: T): ℝ≥0 :=
  if h:s.amms.init τ₀ τ₁ then
  ((s.amms.r₀ τ₀ τ₁ h)*(o τ₀)+(s.amms.r₁ τ₀ τ1 h)*(o τ₁)) / (s.mints.supply τ₀ τ₁)
  else 0 -- price is zero if AMM is not initialized

noncomputable def Γ.networth (s: Γ) (a: A) (o: T → ℝ>0): ℝ≥0 :=
  (W₀.value (s.atoms.get a) o) + (W₁.value (s.mints.get a) (s.mintedprice o))

noncomputable def A.gain (a: A) (o: T → ℝ>0) (s s': Γ): ℝ :=
  ((s'.networth a o): ℝ) - ((s.networth a o): ℝ)
```

Listing 6 Sequences of transactions and reachable states.

```
inductive Tx (sx: SX) (init: Γ): Γ → Type where
  | empty: Tx sx init init

  | swap (s': Γ) (rs: Tx sx init s')
         (sw: Swap sx s' a τ₀ τ₁ v0):
      Tx sx init sw.apply
  -- Other constructors omitted for brevity

def validInit (s: Γ): Prop :=
  (s.amms = AMMs.empty ∧ s.mints = S₁.empty)

def reachable (sx: SX) (s: Γ): Prop :=
  ∃ (init: Γ) (tx: Tx sx init s), validInit init
```

3 Results

We now present some noteworthy properties of AMMs that we have proven in Lean.

Proposition 1 ensures that, in any reachable state, there exists an AMM with token types τ_0 and τ_1 if and only if the minted token type (τ_0, τ_1) is in circulation, i.e. it has a strictly positive supply. This result showcases the validity of our model with regards to reasoning about reachable states. Technically, it also allows us to prove that $\texttt{mintedprice}_s(o, \tau_0, \tau_1)$ is strictly positive for any initialized AMM pair (τ_0, τ_1) in any reachable state s.

▶ **Proposition 1** (Existence of AMMs *vs.* minted token supply). *Let $s' \in \Gamma$ be a reachable blockchain state. Then, for any minted token type (τ_0, τ_1), its supply in s' is strictly positive if and only if s' has an AMM with token types τ_0 and τ_1.*

Proof. By induction on the length of the sequence of transactions leading to s':

- Base case: empty transaction sequence. The proof is trivial for both directions, since a valid starting state has no initialized AMMs and no minted tokens in circulation.

- Inductive case: there are several subcases depending on the last transaction fired in the sequence. Here we consider the creation of the AMM (τ_0', τ_1') in reachable state s'. We proceed by cases on the truth of the equality $\{\tau_0, \tau_1\} = \{\tau_0', \tau_1'\}$. If it is a different token pair, then the supply remains unchanged along with the initialization status, and we can conclude by the induction hypothesis. If it is the same token pair, then we just incremented its minted token supply (which is non-negative, so after incrementing it, it must be strictly positive), and we just initialized the AMM.
- See source code for the other cases. AMMLib/Transaction/Trace.lean:275 ◀

Lemma 2 allows us to determine the change in the value of a wallet after it has been updated in some way: the resulting equality is the basis for all the subsequent proofs. To illustrate it, we briefly introduce two definitions that have been omitted before: $\texttt{drain}_0(w_0, \tau_0)$ is the atomic token wallet such that $\texttt{drain}_0(w_0, \tau_0)(\tau_0) = 0$ and $\texttt{drain}_0(w_0, \tau_0)(\tau_1) = w_0(\tau_1)$ for every other token $\tau_1 \neq \tau_0$. We define $\texttt{drain}_1(w_1, (\tau_0, \tau_0)) \in \texttt{W}_1$ similarly.

▶ **Lemma 2** (Value expansion). *Let $w_0 \in \texttt{W}_0$, $o_0 \in \texttt{T} \to \mathbb{R}_{>0}$, and $\tau_0, \tau_1 \in \texttt{T}$. Then,*

$$\texttt{value}\,(w_0, o_0) \;=\; w_0(\tau_0) \cdot o(\tau_0) + \texttt{value}\,(\texttt{drain}_0(w_0, \tau_0), o_0)$$

and, with $w_1 \in \texttt{W}_1$ and $o_1 \in \texttt{T}^2 \to \mathbb{R}_{>0}$ such that $o_1(\tau_0, \tau_1) = o_1(\tau_1, \tau_0)$,

$$\texttt{value}\,(w_1, o_1) \;=\; w_1(\tau_0, \tau_1) \cdot o_1(\tau_0, \tau_1) + \texttt{value}\,(\texttt{drain}_1(w_1, \tau_0, \tau_1), o_1)$$

Proof. By definition of value and by properties of the sum over a finite support. Full Lean proof at AMMLib/State/AtomicWall.lean:116 ◀

Lemma 3 gives an explicit formula for the gain obtained by a user upon firing a swap transaction. It is fundamental to all proofs involving gain.

▶ **Lemma 3** (Gain of a swap). *Let $sw \in \texttt{Swap}\,(sx, s, a, \tau_0, \tau_1, x)$ and let $o \in \texttt{T} \to \mathbb{R}_{>0}$. Then,*

$$\texttt{gain}\,(a, o, s, \texttt{apply}\ sw) = (sw.\mathrm{y} \cdot o\,(\tau_1) - x \cdot o\,(\tau_0)) \cdot \left(1 - \frac{(s.\texttt{mints}\ a\ \tau_0\ \tau_1)}{\texttt{mintsupply}_s\,(\tau_0, \tau_1)}\right)$$

Proof. By repeated application of Lemma 2 in order to isolate the value contributed by the token types involved in the swap, and by use of Mathlib's `ring_nf` simplifier tactic. AMMLib/Transaction/Swap/Networth.lean:55 ◀

Lemma 4 establishes a correspondence between the profitability of a swap transaction (i.e., a positive or negative gain) and the order between the swap rate and the exchange rate given by the price oracle. In particular, assuming a trader a who is not a liquidity provider (i.e., a has no minted tokens for the AMM pair targeted by the swap), Lemma 4 states that:

- $\texttt{gain}\,(a, o, s, \texttt{apply}\ sw) < 0 \iff sx\,(x, r_0, r_1) < {}^{o(\tau_0)}/_{o(\tau_1)}$
- $\texttt{gain}\,(a, o, s, \texttt{apply}\ sw) = 0 \iff sx\,(x, r_0, r_1) = {}^{o(\tau_0)}/_{o(\tau_1)}$
- $\texttt{gain}\,(a, o, s, \texttt{apply}\ sw) > 0 \iff sx\,(x, r_0, r_1) > {}^{o(\tau_0)}/_{o(\tau_1)}$

Technically, to formalize this result it is convenient to use Mathlib's `cmp`[5], which gives the order between the two parameters.

[5] `https://leanprover-community.github.io/mathlib4_docs/Mathlib/Init/Data/Ordering/Basic.html#cmp`

▶ **Lemma 4** (Swap rate *vs.* exchange rate). *Let* $sw \in \texttt{Swap}\,(sx, s, a, \tau_0, \tau_1, x)$ *be a swap transaction, and let* $o \in \texttt{T} \to \mathbb{R}_{>0}$ *be a price oracle. For* $i \in \{0,1\}$*, let* $r_i = s.\texttt{amms.r}_i\,(\tau_0, \tau_1)$*. If* $s.\texttt{mints}\,(a)\,(\tau_0,\tau_1) = 0$*, then*

$$\texttt{cmp}\,(\texttt{gain}\,(a, o, s, \texttt{apply}\ sw)\,, 0) \;=\; \texttt{cmp}\left(sx\,(x, r_0, r_1)\,, \frac{o\,(\tau_0)}{o\,(\tau_1)}\right)$$

Proof. By term manipulation and Lemma 3. AMMLib/Transaction/Swap/Networth.lean:86 ◀

Lemma 5 establishes that, in a constant-product AMM, there exists only one profitable direction for a swap. Namely, if swapping τ_0 for τ_1 gives a positive gain, then swapping in the other direction (i.e., τ_1 for τ_0) will give a negative gain. Note that the inverse does not hold: a negative gain in a direction does not imply a positive gain in the other direction.

▶ **Lemma 5** (Unique direction for swap gain). *Let* $sw \in \texttt{Swap}\,(\texttt{constprod}, s, a, \tau_0, \tau_1, x)$ *and* $sw' \in \texttt{Swap}\,(\texttt{constprod}, s, a, \tau_1, \tau_0, x')$ *be two swap transactions in opposite directions, and let* $o \in \texttt{T} \to \mathbb{R}_{>0}$*. If* $\texttt{gain}\,(a, o, s, \texttt{apply}\ sw) > 0$*, then* $\texttt{gain}\,(a, o, s, \texttt{apply}\ sw) < 0$*.*

Proof. By Lemma 4. AMMLib/Transaction/Swap/Constprod.lean:160 ◀

▶ **Example 6.** Consider a blockchain state s and atomic tokens τ_0 and τ_1, and assume that:
- a is a trader with no minted tokens, i.e. $(s.\texttt{mints}\ a)\ \tau_0\ \tau_1 = 0$;
- the AMM pair for (τ_0, τ_1) has been initialized in s and has reserves $r_0 = (s.\texttt{amms}\ \tau_0\ \tau_1) = 18$ and $r_1 = (s.\texttt{amms}\ \tau_1\ \tau_0) = 6$;
- all the AMMs in s use the constant-product swap rate function;
- o is a price oracle such that $o\ \tau_0 = 3$ and $o\ \tau_1 = 4$.

Assume that a wants to sell 6 units of τ_1 for some units of τ_0 with the swap transaction $sw \in \texttt{Swap}(\texttt{constprod}, s, a, \tau_1, \tau_0, 6)$. Then, by Lemma 3, a's gain is given by $9 \cdot 3 - 24 = 3 > 0$. Coherently with Lemma 5, any swap in the opposite direction would give a negative gain: e.g., if a sells 6 units of τ_0, her gain would be $3/2 \cdot 4 - 18 = -12$.

We say that a swap transaction $sw \in \texttt{Swap}\,(\texttt{constprod}, s, a, \tau_0, \tau_1, x)$ is *optimal* for a given price oracle o when, for all $sw' \in \texttt{Swap}\,(\texttt{constprod}, s, a, \tau_0, \tau_1, x')$ with $x' \neq x$ we have

$$\texttt{gain}\,(a, o, s, \texttt{apply}\ sw') < \texttt{gain}\,(a, o, s, \texttt{apply}\ sw)$$

Theorem 7 gives a sufficient condition for the optimality of swaps in constant-product AMMs: it suffices that the ratio of the AMM's reserves after the swap is equal to the exchange rate given by the price oracle. Intuitively, the condition in Theorem 7 means that the exchange rate between the two token types induced by the AMM (i.e., the ration between the token reserves) is aligned with the exchange rate given by the price oracle, and so further swaps would yield a negative gain. Note that, by definition, if a swap is optimal then it is also unique, i.e. swapping any other amount would yield a suboptimal gain.

▶ **Theorem 7** (Sufficient condition for optimal swaps). *Let* $sw \in \texttt{Swap}\,(\texttt{constprod}, s, a, \tau_0, \tau_1, x)$ *be a swap transaction on a constant-product AMM, and let* $o \in \texttt{T} \to \mathbb{R}_{>0}$ *be a price oracle. For* $i \in \{0,1\}$*, let* $r'_i = (\texttt{apply}\ sw)\,.\texttt{amms.r}_i\,(\tau_0, \tau_1)$ *be the AMM reserves after the swap. If* ${r'_1}/{r'_0} = {o(\tau_0)}/{o(\tau_1)}$*, then* sw *is optimal.*

Proof. By cases on $x < x'$ and by application of Lemma 4. AMMLib/Transaction/Swap/-Constprod.lean:184 ◀

▶ **Example 8.** Under the assumptions of Example 6, the exchange rate between τ_1 and τ_0 given by the price oracle is $(o\ \tau_0)/(o\ \tau_1) = 3/4$, while the exchange rate induced by the AMM (i.e., the ratio between the reserves) is $r_1/r_0 = 1/3$. Hence, to satisfy the equality in Theorem 7 we must perform a swap that increases r_1 and decreases r_0, i.e. a must sell units of τ_1 to buy units of τ_0, coherently with the necessary condition for a positive gain in Example 6. Theorem 9 below will establish exactly how many units of τ_1 must be traded.

Theorem 9 gives an explicit formula for the input amount x that yields an optimal swap transaction for a given AMM pair, under the assumption that the user firing the transaction does not hold the AMM's minted token type. The other implicit assumption is that the user a firing the swap has the needed amount x of units of the sold token, i.e. $x \leq (s.\texttt{atoms}\ a)\ \tau_1$. In practice, this assumption can always be satisfied with *flash loans*, which allow a to borrow the amount x, perform the swap, and then return the loan in a single, atomic transaction.

▶ **Theorem 9** (Arbitrage for constant-product AMMs). *Let* $sw \in \texttt{Swap}\,(\texttt{constprod}, s, a, \tau_0, \tau_1, x)$ *be a swap transaction on a constant-product AMM, and let* $o \in \texttt{T} \to \mathbb{R}_{>0}$ *be a price oracle. For* $i \in \{0,1\}$*, let* $r_i = s.\texttt{amms}.\texttt{r}_i\,(\tau_0, \tau_1)$ *be the AMM reserves. If* a *has no minted tokens (i.e.,* $s.\texttt{mints}\,(a)\,(\tau_0, \tau_1) = 0$*) then* sw *is optimal if the amount of traded units of* τ_0 *is:*

$$x \;=\; \sqrt{\frac{o(\tau_1)\cdot r_0 \cdot r_1}{o(\tau_0)}} - r_0$$

Proof. By algebraic manipulation and Theorem 7. AMMLib/Transaction/Swap/Constprod.lean:316 ◀

▶ **Example 10.** Under the assumptions of Example 8, we know that to perform a profitable swap the trader must sell units of τ_1, i.e. fire a transaction $\texttt{Swap}\,(\texttt{constprod}, s, a, \tau_1, \tau_0, x)$. Theorem 9 gives the optimal input value x, i.e., the number of sold units of τ_1:

$$x \;=\; \sqrt{\frac{3\cdot 6\cdot 18}{4}} - 6 \;=\; 3$$

Then, the output amount is given by $sw.\text{y} = 3\cdot 18/(6+3) = 6$ and, by Lemma 3, the gain of a is $6\cdot 3 - 3\cdot 4 = 6$, which maximizes it. Note that, in the new state, the exchange rate given by the AMM coincides with that given by the price oracle: $(r_1 + x)(r_0 - sw.\text{y}) = (o\ \tau_0)/(o\ \tau_1)$.

4 Related work

The closest work to ours is [11], which proposes a methodology for developing and verifying AMMs in the Coq proof assistant. In this approach, an AMM is decomposed into multiple interacting smart contract: e.g., each minted token is modelled as a single smart contract, following the way fungible tokens are encoded in blockchains platforms that do not provide custom tokens natively, as e.g. in Ethereum and Tezos. These smart contracts are then implemented as Coq functions on top of ConCert [3], a generic model of blockchain platforms and smart contracts mechanized in Coq. Concert is used to verify behavioural properties of smart contracts, either in isolation or composed with other smart contracts: this is fundamental for [11], where AMMs are specified as compositions of multiple contracts.

More specifically, [11] applies the proposed methodology to the Dexter2 protocol, which implements a constant-product AMM based on Uniswap v1 on the Tezos blockchain[6]. The main properties of AMMs proved in [11] are correspondences between the state of AMMs and the sequences of transactions executed on the blockchain. In particular, they prove that:

- the balance recorded in the main AMM contract is coherent with the actual balance resulting from the execution of the sequence of transactions;
- the supply of the minted token recorded in the main AMM contract is equal to the actual supply resulting from the execution;
- the state of the minted token contract is coherent with the execution.

Furthermore, starting from the Coq specification of the Dexter2 AMM, [11] extracts verified CamlLigo code, which is directly deployable on the Tezos blockchain.

Although both our work and [11] involve AMM formalizations within a proof assistant, the ultimate goals are quite different. The formalization in [11] closely follows the concrete implementation of a particular AMM instance (Dexter2) and produces a deployable implementation that is provably coherent with the proposed Coq specification. By contrast, we start from a more abstract specification of AMM, with the goal of studying the properties that must be satisfied by any implementation coherent with the specification. An advantage of our approach over [11] is that it provides a suitable level of abstraction where proving properties about the economic mechanisms of AMMs, i.e. properties about the gain of users and of equilibria among users' strategies. In particular, Theorem 9 establishes a paradigmatic property of AMMs, which explains the economic mechanism underlying their design.

Besides these main differences, our AMM model and that in [11] have several differences. A notable difference is that our model is based on Uniswap v2, while [11] is based on Uniswap v1. In particular, this means that our AMMs can handle arbitrary token pairs, while in Dexter2 any AMM pairs a token with the blockchain native crypto-currency.

At the best of our knowledge, [11] and ours are currently the only mechanized formalizations of AMMs in a proof assistant. Many other works study economic properties of AMMs that go beyond those proved in this paper. The works [2] and [1] study, respectively, an AMM model based on Uniswap similar to ours, and a generalization of the model where the AMM is parameterised over a *trading function* of the AMM reserves, which must remain constant before and after any swap transactions (in Uniswap v1 and v2, the trading function is just the product between the AMM reserves). The work [5] studies another generalization of Uniswap v2, where the relation between input and output tokens of swap transactions is determined by an arbitrary *swap rate function*, studying the properties of this function that give rise to a sound economic mechanism of AMMs. While both [1] and [5] share the common goal of providing general models of AMMs wherein to study their economic behaviour, they largely diverge on the formalization: [1] is based on concepts related to convex optimization problems, while [5] borrows formalization and reasoning techniques from concurrency theory. The Lean 4 model proposed in this paper follows the formalization in [5], which has the advantage of requiring far less mathematical dependencies: although Mathlib is equipped to reason about convex sets and functions[7], it is currently lacking in advanced convex optimization definitions and results as those used in [1].

Our AMM model is based on Uniswap v2, one of the most successful AMMs so far. We briefly discuss some alternative AMM constructions. Balancer [4] generalizes the constant-product function used by Uniswap to a constant (weighted geometric) mean $f(r_1, \cdots, r_n) =$

[6] `https://gitlab.com/dexter2tz/dexter2tz/-/tree/master/`
[7] `https://leanprover-community.github.io/mathlib4_docs/Mathlib/Analysis/Convex/Function`

$\prod_{i=1}^{n} r_i^{w_i}$, where the weight w_i reflects the relevance of a token τ_i in a tuple $(\tau_1, \cdots, \tau_n)$. Curve [7] mixes constant-sum and constant-product functions, aiming at a swap rate with small fluctuations for large amounts of swapped tokens. The work [9] studies a variant of the constant-product swap rate invariant, where the rate adjusts dynamically based on oracle prices, with the goal of reducing the need for arbitrage transactions. Other approaches aiming at the same goal are studies in [12, 9], and implemented in [10]. Extending our Lean formalization and results to these alternative AMM designs would require a substantial reworking of our model and proofs.

5 Conclusions

In this work we have provided a formalization of AMMs in Lean. Blockchain states are represented as structures containing wallets and AMMs, and transactions as dependent types equipped with a function that defines the state resulting from firing the transaction. Based on this, we have modeled the key economic notions of price, networth and gain. We have then focused on the economic properties of AMMs, constructing machine-checkable proofs. In Lemma 3 we have given an explicit formula for the economic gain of a user after firing a swap transaction. In Theorem 7 we have proved that the rational strategy for traders leads to the alignment between the AMM internal exchange rate and that given by price oracles. Finally, in Theorem 9 we have derived the amount of tokens that a trader should sell to maximize the gain from a constant-product AMM.

Design choices

Before coming up with our Lean formalization, we have experimented with a few alternative definitions. Currently, we use the two pairs of atomic token types (τ_0, τ_1) and (τ_1, τ_0) to represent the same minted token type. Initially, we used the type M of sets of atomic tokens of cardinality 2, and modeled wallets of minted tokens by the type $M \to_0 \mathbb{R}_{\geq 0}$. However, using M in the definition of AMMs turned out not to be as easy. The type $M \to_0 \mathbb{R}^2_{\geq 0}$ would obviously not work since we would not know which value corresponds to the reserve of which token. On the other hand, the dependent type $(m : M) \to_0$ `Option` $(m \to \mathbb{R}_{>0})$ would work after defining $0 :=$ `None`, at the cost of losing the straightforward definition for supply of atomic tokens. We have opted for the custom subtype $\mathbb{R}_{>0}$ to represent the positive reals, since they simplify writing certain definitions and proof passages (e.g., avoiding the use of garbage outputs in the definitions where negative inputs would not make sense). This choice however turned out to have some cons, since it makes using Mathlib more complex, and in some cases we have to coerce back to the reals anyway (e.g. when reasoning about the gain). Using Mathlib's reals would perhaps lead to a smoother treatment.

Limitations

Compared to real-world AMM implementations, our Lean formalization introduces a few simplifications, that overall contribute to keeping our proofs manageable. Bridging the gap with real AMMs would require several extensions, which we discuss below as directions for future work. AMMs typically implement a trading fee $\phi \in [0, 1]$ that represents the portion of the swap amount kept by the AMM. While modeling the fee would be easy (ϕ would be an additional parameter to `SX` functions, to `Swap` types, and to transactions `Tx`), it would require a major reworking of all the results that deal with swaps. Our swaps have *zero-slippage*, in that either a swap gives exactly the amount of tokens required by a user, or they are aborted.

While on the one hand this is desirable (e.g., it rules out sandwich attacks), on the other hand it has drawbacks related to liveness, since a user may need to repeatedly send swap transactions until one is accepted. Real-world AMM implementations allow users to specify a slippage tolerance in the form of the minimum amount of tokens they expect from a swap. Extending our model to encompass slippage tolerance would require to add a parameter to each transaction type, and a minor reworking of the results. Some AMM implementations allow users to create AMMs pairs involving *minted* token types. Consequently, the tokens minted by these AMMs in general are "nestings" of token types. Extending our model in this direction would require to replace price oracles in our results with a suitable price function.

References

1 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.

2 Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An analysis of Uniswap markets. *Cryptoeconomic Systems*, 1(1), 2021. doi:10.21428/58320208.c9738e64.

3 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.

4 Balancer whitepaper, 2019. URL: https://balancer.finance/whitepaper/.

5 Massimo Bartoletti, James Hsin yu Chiang, and Alberto Lluch-Lafuente. A theory of automated market makers in DeFi. *Logical Methods in Computer Science*, Volume 18, Issue 4, dec 2022. doi:10.46298/lmcs-18(4:12)2022.

6 Certora. Formal verification of Compound's open-oracle with Uniswap anchor. https://files.safe.de.fi/safe/files/audit/pdf/CompoundUniswapAnchoredOpenOracleAug2020.pdf, 2020.

7 Michael Egorov. Stableswap - efficient mechanism for stablecoin, 2019. URL: https://curve.fi/files/stableswap-paper.pdf.

8 Uri Kirstein. Detecting corner cases in Compound V3 with formal specifications. https://medium.com/certora/detecting-corner-cases-in-compound-v3-with-formal-specifications-b7abf137fb15, 2022.

9 Bhaskar Krishnamachari, Qi Feng, and Eugenio Grippo. Dynamic curves for decentralized autonomous cryptocurrency exchanges. In *International Symposium on Foundations and Applications of Blockchain (FAB)*, volume 92 of *OASIcs*, pages 5:1–5:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/OASIcs.FAB.2021.5.

10 Mooniswap whitepaper, 2020. URL: https://mooniswap.exchange/docs/MooniswapWhitePaper-v1.0.pdf.

11 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.

12 Improving frontrunning resistance of x*y=k market makers, 2018. URL: https://ethresear.ch/t/improving-front-running-resistance-of-x-y-k-market-makers/1281.

13 Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. Sok: Decentralized exchanges (DEX) with automated market maker (AMM) protocols. *ACM Comput. Surv.*, 55(11):238:1–238:50, 2023. doi:10.1145/3570639.