Draft Version

Meta-Theoretic Reasoning in Formal Verification of Financial Smart Contracts

Derek Sorensen

May 23, 2023

I will replace you with a tiny smart contract.

Contents

Chapter 1

Introduction

Smart contracts are programs stored on a blockchain that automatically execute when certain conditions are met. *Financial smart contracts* are broadly defined as smart contracts that serve as a digital intermediary between financial parties. These include contracts collectively referred to as decentralized finance (DeFi), and come in many forms, including decentralized exchanges (DEXs), automated market makers (AMMs), crypto lending, synthetics (including stablecoins), yield farming, crypto insurance protocols, and cross-chain bridges [194]. Financial smart contracts frequently manage huge quantities of money, making it essential for the underlying code to be rigorously tested and verified to ensure its correctness and security [169, 205].

A defining characteristic of smart contracts is that once deployed, they are immutable. Thus if a contract has vulnerabilities, the victims of an attack are helpless to stop the attacker if the contract wasn't designed with the foresight sufficient to respond. Due to the high financial cost of exploits, it can be worth the large overhead cost to formally verify a smart contract before deployment.

1.1 Formal Verification of Smart Contracts

Much work has been done in formal verification of smart contracts [50, 73, 82, 104, 143, 158, 164, 167, 175, 184]. This work is to formally prove that a contract is correct with regards to a specification. However, financial smart contracts have complicated specifications, and it is not at all straightforward to write one which has all of the intended behaviors. One reason for this is that specifications of financial contracts inevitably include *economic behaviors* which are expressed at a level of abstraction higher than a contract specification.

Let us illustrate this point with the formal verification work on Dexter2, an AMM on the Tezos blockchain.

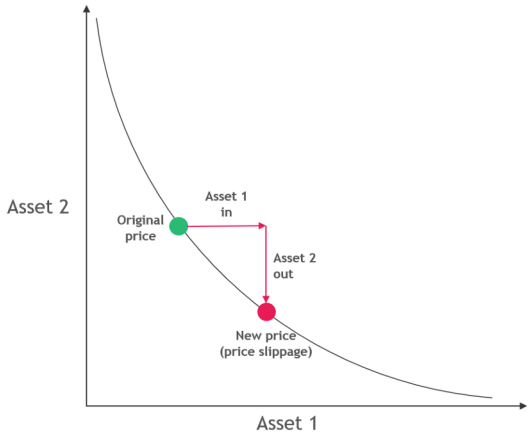


Figure 1.1: A trade along the indifference curve $xy = k$.[1]

1.1.1 Dexter2 Verification

Dexter2 has been formally verified by three different groups using three different formal verification tools [7, 21, 148]. They all based their work on the same (informal) specification [6].

The cited specification specifies the contract interface, including its entrypoint functions, error messages, outgoing transactions, the contents of the storage, some invariants of the storage (including that its store of tokens never fully depletes), fees, and the logic of each of the entrypoint functions. This is a standard and detailed contract specification. Note, however, that while the specification is detailed on the contract design and interface, it doesn't include anything about expected or desired economic behavior.

Consider what Vitalik Buterin, co-founder of Ethereum, wrote in March 2018 about AMMs on the online forum Ethereum Research [19], which informed the design and implementation of Uniswap, likely the first most popular AMM [31]. Buterin proposed that AMMs trade between a pair of tokens along a so-called *indifference curve*

$$xy = k, \tag{1.1}$$

where x represents the quantity held by the contract of the token being traded in, y represents the quantity held by the contract of the token being traded out, and k is constant. The tokens held by the contract come from liquidity providers, which are investors who deposit tokens into the AMM in exchange for a reward, most often a share of transaction fees. That k is constant means that a trade of Δ_x of one token yields Δ_y of another such that the product from (1.1) stays constant at k:

$$(x + \Delta_x)(y - \Delta_y) = k.$$

Buterin argues that an AMM that trades along (1.1) features efficient price discovery. He also argues that it

[1] Image from [9].

can properly incentivize liquidity providers by charging a 0.3% fee on each trade to give to them. Notably, he states that this design is vulnerable to front-running attacks.

Intuitively, we can deduce that the AMM design of Dexter2, specified in [6], is supposed to feature these economic qualities described by Buterin, including efficient price discovery and some suitable incentive mechanism so that investors deposit tokens into the AMM contract and provide liquidity to the market. We might also anticipate front-running attacks and try to mitigate them in some way.

However, concluding that the informal specification of Dexter2 [6] or its formal counterparts [7, 21, 148] actually imply any of these economic behaviors is no more than a matter of intuition. AMM fees and liquidity provision alone are highly complexs topics, with economic studies on choosing optimal transaction fees [96, 102, 103]; on how liquidity providers react to market changes [108]; and how all of that relates to the *curvature* of $xy = k$ [30, 191]. It was also shown that front-running attacks can warp the incentive scheme of the blockchain itself in such a way that could compromise its underlying security [81].

If we are to conclude that the specification of Dexter2 *does* imply any of these intended, economic properties, we need to incorporate the learnings from these studies and somehow reason about the *specification itself*.

1.2 A Meta-Theoretic Approach

As it stands, formal verification of smart contracts reasons about smart contracts, proving a contract correct with regards to a formal specification. However, it does not reason about the specification itself. In this thesis, we propose a *meta-theoretic* approach to the formal verification of financial smart contracts, which is a framework to reason about contract specifications and their associated proofs of correctness. Our fundamental contribution is the notion of a *metaspecification*, or a specification of a contract specification, in the context of smart contract verification.

There is good reason for doing this. As we will see in the coming chapter, economic attacks on financial smart contracts due to incorrect specifications are common and costly. Meta-theoretic proof techniques will help us tame the complexity of financial contract specifications, thus mitigating risk of contract exploits due to incorrect contract specifications.

In short, our thesis is that:

> *We can mitigate vulnerabilities in financial smart contracts due to incorrect specifications through formal verification using meta-theoretic techniques.*

Chapter 2

Related Work

Our work sits at the intersection of studies to formally understand financial smart contracts and their desirable properties, formal verification of smart contracts, and metaprogramming. Before reviewing each of those topics, we first discuss smart contract vulnerabilities which we hope to mitigate by reasoning about contract specifications.

2.1 Smart Contract Vulnerabilities

We have identified three classes of vulnerabilities in financial contracts which give context to the meta-theoretic techniques presented in this thesis. The first are vulnerabilities to economic attacks, which are attacks of a smart contract's design or specification that targets unexpected but correct—*i.e.* functioning as specified—smart contract behavior. The second are vulnerabilities introduced as a contract evolves through upgrades. The last are vulnerabilities due to difficult-to-specify behaviors of a system of contracts.

For each of these classes of vulnerabilities, we will give examples of successful attacks from within the last few years and discuss how we might mitigate issues like this by reasoning about contract specifications, at least in principle. While the core chapters of this thesis are formalized in the proof assistant Coq [162], the discussion here is illustrative and thus informal.

2.1.1 Vulnerabilities to Economic Attacks

There is an important distinction in the literature on smart contract vulnerabilities between tecnical attacks and economic attacks. As noted by Werner *et al.*, technical attacks have a healthy literature, and can

be addressed formally by verifying that a contract matches its specification. On the other hand, economic attacks are largely unexplored, and we do not yet have ways to address them with formal verification because existing tools "lack the capability to recover and understand high-level DeFi semantics" [196].

To illustrate what an economic attack is, let us look first at Beanstalk, a decentralized stablecoin protocol built on Ethereum which uses a decentralized governance protocol. The governance protocol features a function called `emergencyCommit()`, which is designed to allow governance to respond quickly to an emergency. It gives a supermajority of governance votes power to approve and execute a proposal in one vote, so long as the proposal has been active for more than 24 hours. The Beanstalk contract also holds a considerable amount of funds. On April 17, 2022, Beanstalk was attacked using a flash loan and drained of its funds [92].

Flash loans are loans mediated by a smart contract, issued for the duration of a single transaction. Due to their atomicity, flash loans remove the creditor's risk of debt default, and enable enormous, uncollateralized loans. For example, the Aave flash loan pool has at times had in excess of $1 billion which can be loaned out [165]. Flash loans can introduce unintuitive contract behavior which can be difficult to reason about and which have been extensively studied [26, 102, 105, 106, 165, 188].

The Beanstalk attacker proposed two malicious proposals, which if approved by governance would send all of the contract's funds to the attacker's address. After waiting 24 hours, the attacker used a flash loan to buy a supermajority of governance tokens and immediately invoked `emergencyCommit()` to pass their proposals and disburse the contract's funds. After paying back their flash loan, they made away with the equivalent of about 77 million USD [2].

This all happened with the contract functioning as specified [2]. Presumably, the 24-hour waiting period between a proposal and calling `emergency_commit()` was designed to protect the governance process from malicious users, the idea being that governance would be sufficiently attentive to reject a malicious proposal within that period. Delays like this are fairly standard anti-flash-loan tactics [165].

The Beanstalk contract's 24-hour waiting period was insufficient to prevent a flash loan attack on governance—but what measures would be sufficient? To know this, we must reason about the contract specification's economic properties with respect to flash loans. Because flash loans need to be repaid in the same transaction in which they are taken out, most flash loan attacks are made possible because they are profitable within that same transaction [165]. Thus we might protect against a flash loan attack on contract governance by requiring that the contract balance be invariant under all transactions involving voting. For example, rather than giving a supermajority of governance the right to execute an emergency proposal, we can give them the right to pause contract functionality while a new proposal makes its way through the standard governance procedure.

The Spartan Protocol, an AMM on BSC, is another good example of vulnerabilities to economic attacks

due to incorrect contract design. It had flawed economic logic in how it calculated the liquidity share for users burning LP tokens in exchange for underlying funds in the liquidity pool. According to Peckshield, a blockchain security company, the Spartan Protocol contract used the real-time price, rather than a cached price, to calculate liquidity shares [118]. This meant that an attacker could use a flash loan to deposit liquidity, manipulate the price oracle of certain assets, and then withdraw liquidity at a more favorable rate, which happened in May 2021 [67] for a profit of roughly the equivalent of 30 million USD at the time.

A very similar attack was launched against Pancake Bunny, a yield aggregator on BSC. This vulnerability was due to a similar issue of calculating liquidity shares incorrectly using an oracle that updates in real-time, allowing the attacker to mint more than a billion USD worth of BUNNY tokens after manipulating a price oracle [115, 155]. After repaying the flash loans, the attacker left with 114k WBNB and 697k BUNNY tokens, amounting to about 45 million USD at the time in lost funds [12, 66].

We again argue that such an issue can in principle be detected and mitigated by reasoning about the economic properties of the contract specification. As the attack was another flash loan attack, which are commonly used to manipulate oracles [165], we might design a contract specification such that a flash loan cannot manipulate the oracle anytime it is used by the protocol. The specification would then require that the price given by the oracle and used by the contract be constant for the duration of any transaction. Peckshield's prescription for the Spartan Protocol, that the cached price should be used rather than the real-time price, serves the purpose of preventing a holder of a flash loan from manipulating a price oracle, so long as the cached price remains constant within the atomic transaction of the flash loan.

Finally, we look at Mango Markets, a decentralized exchange (DEX) on Solana that lets investors lend, borrow, swap, and use leverage to trade crypto. In October 2022, Mango Markets was attacked by a complex set of transactions, which manipulated an oracle and culminated in the attacker taking out a large, undercollateralized loan [177]. The attack diagnosis made by Beosin, a smart contract auditor, points to a complicated and subtle trading strategy, made possible by the contract design, which only a sophisticated trader would be able to see and exploit [54]. Notably, CoinDesk reported that the attacker "did everything within the parameters of how the platform was designed" [134]. Avraham Eisenberg, the exploiter, also wrote on Twitter, "all of our actions were legal open market actions, using the protocol as designed, even if the development team did not fully anticipate all the consequences of setting parameters the way they are." [42].

The nature of the Mango Markets attack is more subtle than the ones that we've encountered previously in this section, and so the reasoning required to identify it and prevent it will also be more subtle and complex. This is out of the scope of the present discussion, but we include it to point out the complexity of economic behaviors which a smart contract can have. We will see in coming sections that there are various studies and theories that have emerged, *e.g.* [48, 105, 108, 165, 183], which seek to understand and reason

about the complex economic behaviors of smart contracts. These will help us reason formally about contract specifications in Chapter 4.

2.1.2 Unsafe Upgrades and Forks

As its desired economic behaviors change over time, a financial contract will need to undergo upgrades. Upgrades are notoriously difficult and error-prone [45]. In an upgrade, one changes a contract's specification and thus its economic properties. This can lead to unexpected vulnerabilities, which we highlight here.

Let us first look at Nomad, a cross-chain bridge protocol which allows users to deposit funds in a smart contract on one chain and withdraw funds on another. Cross-chain bridges have been subject to several high-profile attacks [125], including this one. On August 1, 2022, more than 500 hacker addresses exploited a bug introduced by an upgrade to one of the Nomad smart contracts [168]. The upgrade incorrectly added the null address, `0x000...000`, as a trusted root, which turned off a smart contract check that ensured withdrawals were made to valid addresses only. This meant that anyone could copy the attacker's original transaction and withdraw funds from the Nomad contract to their own wallet address. The attack resulted in $190 million in lost funds [95, 117].

Similarly, we look at Uranium Finance, an AMM which also suffered a costly exploit after a contract upgrade. The original contract contained a constant, `K`, equal to `1,000` in three different places, which was used to price trades. The update to the code changed this value to `10,000` in two places but not the third, presumably to calculate trades with higher precision. The result of this was that the attacker could swap virtually nothing in for 98% of the total balance of any output token, which resulted in a loss of $50 million of contract funds [98]. NowSwap, a nearly identical application, had the same error and incurred a loss of $1 million [53].

In each of these cases we see a vulnerability introduced because a contract upgrade does not retain important properties, either of safety or of the contract's functionality, of the previous contract. In the first case, the property that withdrawals are only made to valid addresses does not persist in the upgrade. In the latter cases, trades are priced in a radically different way when they should have been priced very similarly.

We can avoid these kinds of errors by precisely specifying what the upgraded changes should look like. We could be clear on what properties should remain constant (*e.g.* safety checks are the same through the upgrade for Nomad), and what properties should change (*e.g.* now trades are calculated with more precision for Uranium). This helps us understand what precisely is changing and being upgraded, what should remain the same, and whether or not the upgrade is backwards compatible. This can be done informally, of course, but in Chapter 5 we introduce methods to do this kind of reasoning formally.

2.1.3 Vulnerabilities and Complex System Behavior

Finally, we will look at economic vulnerabilities introduced through the sheer complexity of systems of smart contracts. We first look at Harvest Finance, a yield aggregator on Ethereum. On October 26, 2020, an attacker used a flash loan to trade about $17.2 million USDT for USDC on Curve, which temporarily increased the price of USDC in the Curve Y pool. Due to the fact that Harvest Finance uses the Curve Y pool as a price oracle in real-time to calculate the vault shares for a deposit, this allowed the attacker to get into a Harvest vault at an advantageous rate. In the same transaction, the attacker reversed the trade on Curve, after which the price of USDC returned to normal in the Curve Y pool, which increased the value of the attacker's shares in terms of the now less expensive USDC. The hacker then exited the vault at this new exchange for a profit of $33 million in lost user funds [153].

Let us also look at CREAM finance (short for *Crypto Rules Everything Around Me*), a multi-purpose DeFi protocol that brands itself as a one-stop shop for decentralized finance. It offers crypto lending, borrowing, yield farming, and trading services, and has several connected implementations across multiple chains. An October 2021 attack drained the pool of roughly 260 million USD in assets [93]. The attack was extremely complex, involving 68 assets and over 9 ETH in gas, roughly 36k USD at the time [93]. Immunefi, a bug bounty platform for smart contracts, diagnoses that the exploit was due to an easily manipulable price oracle, and uncapped supply of the token yUSD [116]. Even so, the attack is complex enough that only experts such as Immunefi can give a comprehensive diagnosis.

Both of these examples are economic attacks which leverage the complexity of interacting systems of smart contracts. The complexity of the Harvest Finance contract behavior comes in part because it relies on another smart contract—Curve—which the Harvest Finance team did not design and which may have unexpected bugs or behaviors which will be very difficult for them to be aware of or predict [183]. The complexity of the CREAM finance specification comes in part because it is a modular system of contracts, spread over many blockchains, which all interact [116].

Systems of contracts are extremely difficult to reason about, formally or informally [149]. There have been recent efforts to formally and rigorously study systems of interacting DeFi contracts [183], but nothing comprehensive has yet been formalized. The work of Chapter 6 seeks to address this by developing methods to formally reason about a system of contracts in terms of a single, monolithic contract, which is substantially easier to reason about [149]. This will relate the specification of a system of contracts to that of a single contract, which falls within our meta-theoretic goals.

Smart Contract	Vulnerability	Funds Lost (USD)	dApp Type	Exploit	Year
Mango Markets	Contract Design [134]	115M	DEX	[177]	2022
Beanstalk	Contract Design [2]	77M	Stablecoin	[92]	2022
Pancake Bunny	Contract Design [12]	45M	Yield aggregator	[66]	2021
Spartan Protocol	Contract Design [118]	30M	AMM	[67]	2021
Nomad	Unsafe Upgrade [117]	190M	Cross-chain bridge	[95]	2022
Uranium	Unsafe Upgrade [98]	50M	AMM	[68]	2021
Cream Finance	Complex System [116]	130M	DeFi protocol	[93]	2021
Harvest Finance	Complex System [97]	34M	Yield aggregator	[94]	2020

Figure 2.1: A small sample of recent attacks, totalling to about USD 776M in lost funds.

2.1.4 Addressing Economic Attacks

Blockchain-based applications lost 2.44 billion USD in 2021 and 3.6 billion USD in 2022 due to attacks [54]. In 2022, DeFi applications were the most attacked type of blockchain-based application, making up about two-thirds of all attacks [5, 54], and attacks which exploit improper business logic or function design are in the top three causes of loss [54, p.10].

The work of this thesis seeks to set theoretical foundations to address issues of incorrect specifications within a formal setting. We do so by developing tools to reason formally about contract specifications, from formally understanding economic behaviors implied by a contract specification (Chapter 4), to formally linking contract specifications as contracts undergo upgrades (Chapter 5), to reasoning formally about systems of smart contracts in terms of (simpler) monolithic contracts (Chapter 6). Over time, through these techniques of formal reasoning we hope to see a reduction in funds lost due to exploits of unexpected contract behavior.

2.2 Understanding Financial Smart Contracts

Much work has been done to understand smart contracts and their behavior, and to mitigate exploits like the ones we saw in the previous section. Studies of smart contract behavior range from formal and theoretical to practical and data-driven studies. They also range in their focus and scope, from a focused study of things like flash loans or AMMs, to broad systemizations of knowledge of topics like various types of attacks [41, 125, 91, 90], decentralized finance [194, 197], blockchain interoperability and bridging [189, 203, 139], or other areas of DeFi [46, 79]. Figure 2.2 shows a graphical representation of some of these studies, where they range left-to-right from focused in scope to broad, and they range bottom-to-top from formal and theoretical

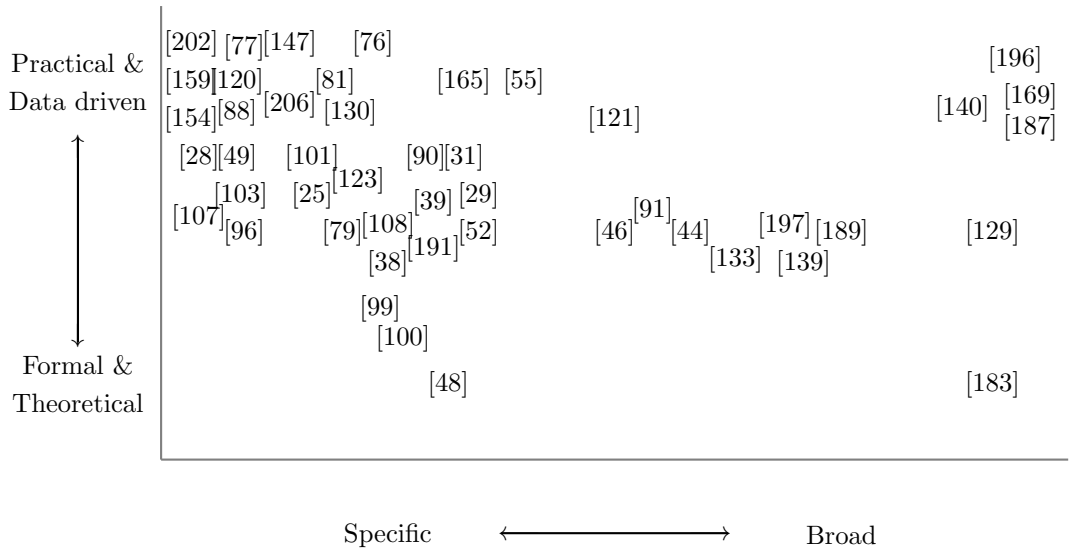


Figure 2.2: A graphical representation of studies of smart contracts and their behavior.

to practical and data driven.

Importantly, as we noted before these studies have not yet been formally incorporated into formal verification of smart contracts [196], which is what we seek to start in this thesis. In the remainder of this section, we will highlight the studies that we incorporate into our work in various degrees later on.

Firstly, we look at and A^2MM [206] and FairAMM [77], two AMMs which are designed to resist front-running attacks, the vulnerability of AMMs originally pointed out by Buterin [19]. Consider the contract specification given in the A^2MM paper. It includes a formal, abstracted theory with a system model, a threat model, and an abstract notion of an AMM [206, §III.A-D]. In this model, an AMM is a contract satisfying the properties of *liquidity sensitivity*, *path independence*, and *market independence*, each of which are defined in the paper in the context of this abstract model of smart contracts [206, §III.C]. Using this abstract and formal model, the authors prove two theorems [206, §III.E, G] about A^2MM as an AMM which relate to its ability to mitigate frontrunning, transaction reordering, etc. In the context of the paper, these theorems justify the notion that the specified AMM has the desired economic behavior as implied by the contract's specification.

FairAMM [77] is similar, in that it defines an AMM specification in the context of a formally abstracted model, and proves three theorems about the contract's implied behavior. The final one is particularly game-theoretic in nature:

> **Theorem 3 (Incentive compatibility).** *A rational profit-seeking market maker has no incentive to manipulate the price given knowledge that some trader wishes to place a trade and the direction (buy/sell) of the trade being known.*

Because both of these AMMs are designed specifically to mitigate economic attacks, including front-running attacks, the models and theorems are core to the correct behavior of both of these AMMs and to any assertion that their specification actually solves those problems. As each of these theorems has a formal proof within the context of the model, there is no reason why it could not be formalized in a proof assistant.

Zooming out from these specific AMMs, the literature also includes broad-scale studies which attempt to distill desirable properties of a "well-behaved AMM" more generally. Among these are Angeris *et al.*'s work on constant function market makers (CFMMs) [28], their work on analyzing Uniswap markets [31], and Bartoletti *et al.*'s work on developing a theory of decentralized finance [48]. Each of these take what is considered to be "good" market behavior and distill it to qualities that could be proved about a specification in the context of some formal model, similar to what we saw with A^2MM [206] and FairAMM [77]. Bartoletti *et al.* do so within the context of a formal model [48].

Some examples of these properties include:

- *Demand sensitivity*, or the property that the price of an asset increase with demand [48, §4.2] [31, §2.3]
- *Non-depletion*, or the property that the balance of a token held by an AMM cannot be zeroed through token trades, also called swaps [48, §4.2] [31, §2.3]
- That trading cost be positive, meaning that a swap of one token for another, and then back again has a positive cost [31, §2], [28, §3 §4.1]
- And that market prices converge through arbitrage [48, §4.3] [31, §3.1]

Were we writing a specification for a new AMM, each of these are properties that we could formalize and prove about our specification. We would do so to prove that the AMM is "well-behaved" in an economic sense.

Zooming out further, when reasoning about financial smart contracts, one quickly has to reason about systems of interacting smart contracts. This is inevitable, since financial smart contracts are financial intermediaries, so their desirable properties inevitably relate to their behavior within a system of interacting contracts. We can see this in the A^2MM [206] and FairAMM [77] specifications, the former of which defined an abstract AMM to interact with, and the latter of which proved game-theoretic properties about incentives and interactions on-chain. We also see this in discussions about arbitrage above.

Systems of smart contracts are substantially more difficult to reason about than monolithic contracts, owing to the number of ways contracts can interact with each other [149]. Even so, there are developments in the literature of formal models of systems of smart contracts, including Bartoletti *et al.*'s work with lending pools [46], and Tolmach *et al.*'s process-algebraic approach to reasoning about composable DeFi protocols

[183].

In a real sense the work of this thesis is to begin the difficult process of introducing the learnings from this corpus of work into a formal setting. In Chapter 4 we will draw on some of the work cited here to form a contract specification and metaspecification of a pool contract and AMM in Coq. We will also build tools to reason formally about systems of contracts and contract upgrades.

2.3 Smart Contract Verification

Let us now consider the landscape of formal verification with the goal of expressing and reasoning about meta-theoretic properties of smart contracts. We will draw on several surveys of formal verification applied to smart contracts, including [50, 73, 82, 104, 143, 158, 164, 167, 175, 184]. From among the plethora of tools available to formally verify smart contracts, we will focus on those which use proof assistants, which we do for a few reasons.

Firstly, [184, §4.2] makes the point that proof assistants are particularly well-suited for meta-theoretic reasoning. While this is generally used to reason meta-theoretically about the smart contract language itself, mostly used to verify code extraction from the verification process, we make use of this property extensively to reason meta-theoretically about smart contracts, their specifications, and proofs of correctness with regard to those specifications. Verification tools based on proof assistants also tend to use the proof assistant itself as a specification language, and so smart contracts are treated as mathematical objects, about which any mathematical statement can be made. In principle, this means that proof assistants can be used to verify any correct design [167]. Finally, we use proof assistants because they are recognized in the literature to be of the highest quality for formal verification [143].

Among the many options, our primary concern when looking at a suitable verification framework is its ability to support meta-theoretic reasoning about smart contracts and their specifications. As we will see, most formal verification in theorem provers involves embedding a smart contract language and (to varying degrees) its semantics into a proof assistant language like Coq, Isabelle/HOL, or Agda. The semantics are usually limited to the semantics of the contract execution, but not the underlying blockchain itself. To our knowledge, only in the case of ConCert [35] do we also get the full semantics of the underlying blockchain as well as the smart contract language, which we argue makes ConCert particularly well-suited for the theory of this thesis.

2.3.1 Smart Contract Language Embeddings

Most verification tools which use proof assistants come in the form of an embedding of a smart contract language into the proof assistant. Smart contracts in that embedded language are then reasoned about through the embedding. We will survey these briefly, starting with embeddings of low-level languages and moving to those of intermediate- and high-level languages.

There are several examples of low-level language embeddings into proof assistants. For EVM bytecode alone we have embeddings in Isabelle [27, 112]; in Coq [201]; in the K Framework [111, 156]; in Why3 [204, 145]; and in Z3 [131]. There are similar embeddings for other smart contract languages, including for the Tezos smart contract language Michelson, which is a low-level, stack-based language. These include Mi-Cho-Coq, an embedding of Michelson into Coq [59]; K Michelson [22], an embedding of Michelson into the K Framework; and WhyISon, which transpiles Michelson into WhyML, the programming and specification language of the Why3 framework [80]. There are examples on other chains as well, including an embedding of the low-level, Bitcoin-based contract language Simplicity in Coq [152].

Due to their proximity in language and semantics of the lowest-level executing environment to each of these blockchains, low-level language embeddings are more likely to be faithful to the actual execution environment [127]. On the other hand, higher-level languages tend to be more human-readable, and can be more straightforward to reason about [127], especially if they are statically typed functional programming languages [35]. Naively, this higher level of abstraction seems like it would make it more straightforward to reason about the correctness of a specification. However, the abstraction can come at the cost of rigor. As they sit at a higher level of abstraction, they require a rigorous language embedding as well as a correct compiler down to the low-level, executable code which preserves the proven properties [127].

We have many examples of intermediate- and high-level languages which have been embedded into proof assistants. For Ethereum smart contracts, we have Lolisa [200] and FEther [199], embeddings of Solidity into Coq; as well as embeddings of Solidity into Event-B [207]; into Isabelle [127]; into the K Framework [119]; and into F* [61]. We also have custom implementations of finite state machines used by FSolidM [136, 137] and VeriSolid [138] that verify Solidity code. For Tezos, we have an embedding of Albert [60], an intermediate-level language which compiles down to Michelson, and which targets Mi-Cho-Coq; we also have ConCert, which has a certified extraction mechanism from Coq code into multiple smart contract languages, including into CameLIGO, an intermediate-level language which compiles down to Michelson [35, 36]. On other chains we have Plutus in Agda [72], and verification for BNB in Coq [181].

At higher levels of abstraction, we also have some DSLs written in proof assistants which target various smart contract languages. Scilla is intermediate-level, and can be used to reason about temporal properties of smart contracts, which targets Solidity [170]. Archetype is a Tezos-based DSL which targets business logic

and uses Why3 [58]. Multi is a framework, written in Coq, which targets reasoning about smart contract interactions [71]. At an even higher level of abstraction, we have TLA+, a tool for reasoning about concurrent and distributed systems, which was used to verify the a cross-chain swap protocol [146].

Ideally, to reason meta-theoretically about smart contracts and their specifications, we would have a language sufficiently high-level to ease reasoning about it, but not so high-level that we sacrifice rigor.

2.3.2 ConCert

We will do the work of this thesis in ConCert [35] for three reasons. Firstly, while ConCert has not yet been used to reason meta-theoretically about smart contracts, it is extremely well-suited to do so. The specification language is in Coq and so it is unrestricted, except by limitations of the blockchain model itself, in the types of statements that can be made and proved about smart contracts. It also formalizes blockchain execution semantics underlying the execution of a smart contract, which means that there is a well-defined smart contract type, `Contract`, in the context of the model, which has a specific semantics within the blockchain and which can thus be reasoned about abstractly and meta-theoretically. This, to our knowledge, is unique to ConCert. As we will see throughout this thesis, the combination of these characteristics allows us to reason unrestrictedly about smart contracts abstractly, as mathematical objects, which is key to the goal of this thesis.

Finally, ConCert's extraction mechanism from Coq code into its target smart contract languages is verified, so despite it not being a low-level embedding like Mi-Cho-Coq, we have a high-fidelity translation into executing code. This is made possible by MetaCoq [32, 34, 180], which is a tool for reasoning about the extraction mechanism. This gives us the advantages of verification at a higher level of abstraction without compromising the integrity of the verification results.

2.4 Metaprogramming

Finally, our work in meta-theoretic reasoning falls within a broader category of metaprogramming. Metaprogramming is the practice of writing programs that treat programs as data, allowing them to reason about or transform existing programs or generate new ones [128]. It has a long history, starting with Lisp macros, and is now arguably mainstream with a variety of applications [57, 128, 173].

Metaprogramming is an important topic in software verification, *e.g.* [56, 57], primarily as a method to specify and verify meta-programs—especially those which generate code. MetaCoq [180], a formalization of Coq in Coq, is particularly relevant to us as it is used to verify ConCert's extraction mechanism into

executable smart contracts [33]. In our work, we will contribute to metaprogramming as it relates to smart contract verification with various notions of equivalences of smart contracts in Chapters 5 and 6.

Metaprogramming shows up most prominently in our work through the notion of a *metaspecification*, or a specification of a specification. The concept of a metaspecification exists in the literature, *e.g.* in [192, 193], which uses metaspecifications to catalogue software patterns, not dissimilar to the use of a macro but to generate program specifications rather than code.

Our use of the notion of a metaspecification is different. While previous work uses metaspecifications to identify patterns in boilerplate specifications, we use metaspecifications to specify complex contract behavior which is generally not fully captured by specifications in the wild. Firstly, we specify how a contract specification behaves economically in relation to its execution environment. This involves reasoning about contract behavior implied by a contract's specification, and will make explicit use of ConCert's formalization of the blockchain execution semantics. We also will reason about specifications as they evolve through upgrades or as they relate to other contract specifications.

To our knowledge, our use of metaspecifications and metaprogramming is novel in the context of smart contract formal verification.

2.5 Summary

The work of this thesis draws on previous work to systematically understand smart contract behavior and attempts to introduce its findings into formal verification of smart contracts. We do this with various metaprogramming techniques, with the primary aim to reason meta-theoretically about smart contracts and their specifications as abstract mathematical objects. Ultimately, our goal is to be able to formally address economic vulnerabilities of smart contracts, making smart contracts safer to design, deploy, and use as critical financial infrastructure.

Chapter 3

Background

Before proceeding to the core work of this thesis, we will give some background on Coq and ConCert.

3.1 A Very Brief Introduction to Coq

Coq is a language in which one can state and prove mathematical facts, as well as being a functional programming language [162]. It has deep connections to OCaml with the Coq-of-OCaml verification tool [126, 185]. Additionally, there has been extensive work on formalizing pure mathematics, including in the UniMath library and others [51, 85, 186]. For the interested reader, we cite some tutorials and introductions to Coq as a tool for program verification [75, 114, 162].

In Coq, the main concepts that will be relevant to us are record and inductive types and type classes, as well as proofs and tactics. We give a short introduction to each.

Record Types, Inductive Types, and Type Classes

Coq is built on the Calculus of Inductive Constructions [157], so unsurprisingly, centers on inductive types. We will cover the Coq syntax of inductive types, as well as a brief introduction to the notion of an induction principle. For more details and a deeper introduction, see [78, 160].

First let us look at the Coq syntax of inductive types. An inductive type is defined by *constructors*. Take for example the inductive definition of a list in Coq.

```
1 Inductive list (A : Type) : Type :=
2 | nil : list A
```

```
| cons : A -> list A -> list A.
```

This definition introduces a new type `list A` with two constructors: `nil`, which represents the empty list, and `cons`, which represents a non-empty list with a head element of type `A` and a tail of type `list A`. To construct something of type `list A`, we can either take `nil : list A`, or append something of type `A` onto something of type `list A`. For example, the list `[1, 2, 3]` could be written as

```
my_list : list nat := cons 3 (cons 2 (cons 1 nil)).
```

For any inductive type, the constructors give us a way to construct functions out of the inductive type, which is its induction principle. In our case, to construct a function out of the type `list A`, we need to specify the image of `nil`, and the image of `cons h t`, for `h : A` and `t : list A`. Take, for example, a function which computes the length of a list:

```
Fixpoint length (A : Type) (l : list A) : nat :=
match l with
| nil => 0
| cons h t => S (length A t)
end.
```

The function `length` uses pattern-matching to specify the image of each constructor of `list A`.

Record types, common in many languages, are inductive types with one constructor. We can write a record type as follows

```
Record point2D := construct_2D_point {
  x : nat ;
  y : nat ;
}.
```

to represent a coordinate of natural numbers in the x-y plane, or as an inductive type with one constructor

```
Inductive point2D :=
| construct_2D_point (x : nat) (y : nat).
```

Finally, type classes are mechanisms for defining abstract concepts and relations that can be shared among many types. For example, here is the definition of a Coq typeclass `Show`, which characterizes types which can be converted into strings.

```
Class Show (A : Type) := {
    show : A -> string
  }.
```

Here is another typeclass for types which have an equivalence relation `eq` defined on them.

```
Class Eq (A : Type) := {
  eq : A -> A -> Prop;
  eq_refl : forall x : A, eq x x;
  eq_sym : forall x y : A, eq x y -> eq y x;
  eq_trans : forall x y z : A, eq x y -> eq y z -> eq x z
}.
```

Proofs and Tactics

Coq is an interactive theorem prover. When we state and prove theorems in Coq, we get a view into our hypotheses and our goals while we are proving results. Coq has tactics which we use to resolve our goals and prove our stated theorems.

Take for example the Modus Ponens theorem, which is that for all propositions `P` and `Q`, if `P` implies `Q`, and we know `P` to be true, then we know that `Q` is true. In Coq, this is stated on the left-hand side of this snapshot:

```
Theorem modus_ponens : forall (P Q : Prop),
  (P -> Q) -> P -> Q.
Proof.
  intros P Q P_implies_Q P_holds.
```

```
1 subgoal
P, Q : Prop
P_implies_Q : P -> Q
P_holds : P
------------------(1/1)
Q
```

Take note of the right-hand-side of the snapshot. Above the dotten line, we have the context, which includes our hypotheses and results we have already proved. We have that `P` and `Q` are propositions, that `P` implies `Q`, and that `P` is true. We can apply the implication `P_implies_Q` to our goal with the tactic `apply`, which transforms it to a goal of knowing that `P` is true.

```
Theorem modus_ponens : forall (P Q : Prop),
  (P -> Q) -> P -> Q.
Proof.
  intros P Q P_implies_Q P_holds.
  apply P_implies_Q.
```

```
1 subgoal
P, Q : Prop
P_implies_Q : P -> Q
P_holds : P
------------------(1/1)
P
```

We already assumed that `P` is true, so this can be solved by the tactic `assumption`.

There is a wide variety of tactics specialized for different scenarios which can be used to prove theorems [1]. There is also a tactic language, `Ltac`, for writing custom tactics. For the interested reader, there are good tutorials for learning Coq, including [114].

3.2 Introduction to ConCert

Here we will introduce the types and tactics of ConCert which are most relevant to the work of this thesis. We will first look at the `Contract` type, and discuss what a specification of a smart contract looks like in ConCert. Then we will look at the underlying semantics, which abstracts at two levels: the `Environment` type, and the `ChainState` type, each of which can be acted on, respectively, by the `Action` and `ChainStep` types which model the progression of an executing blockchain.

We will then discuss what proofs of contract specifications look like in ConCert, covering one of ConCert's main tactics, `contract_induction`. For any interested reader, the codebase and thorough documentation can be found at the ConCert GitHub repositroy [70].

Smart Contracts in ConCert

In ConCert, smart contracts are abstracted as a pair of functions: the initializing function, `init`, which governs how a contract initializes, and the receive function, `receive`, which governs how a contract handles calls to its entrypoints.

```
Record Contract
       (Setup Msg State Error : Type)
       `{Serializable Setup}
       `{Serializable Msg}
       `{Serializable State}
       `{Serializable Error} :=
  build_contract {

    init :
      Chain ->
      ContractCallContext ->
      Setup ->
      result State Error;

    receive :
      Chain ->
      ContractCallContext ->
      State ->
      option Msg ->
      result (State * list ActionBody) Error;
  }.
```

To understand how smart contracts are modeled, let us briefly look at the `Chain`, `ContractCallContext`,

`Setup`, `State`, `Msg`, `Error`, and `ActionBody` types. In brief,

- The `Chain` type carries data about the current state of the chain, such as the block height (we will see this type again shortly).
- The `ContractCallContext` type carries information about the context of a contract call, including the transaction sender and origin (which are distinct concepts), the contract's balance, the amount of layer-one token (*e.g.* ETH or XTZ) sent in the transaction, *etc.*
- The `Setup` type indicates what information is needed to deploy the contract.
- The `State` type is contract's storage type.
- The `Msg` type is the type of messages a contract can receive.
- The `Error` type is the type of errors a contract can throw.
- Finally, the `ActionBody` type is ConCert's type of actions which can be emitted by a contract.

In ConCert, then, to define a smart contract one must define the `Setup`, `State`, `Msg`, and `Error` types and give `init` and `receive` functions. As we will see, a call to a smart contract modifies the state of the blockchain by updating the contract state according to the `receive` functions, and by emitting transactions of type `ActionBody`. If a call to a contract results in something of type `Error`, the execution rolls back and the `Environment` remains unchanged.

There is also the notion of a `WeakContract`, which is the serialized form of the `Contract` type and which is used internally to ConCert. It is serialized to deal with Coq's polymorphism. Importantly, anyone doing contract verification work in ConCert should never encounter the `WeakContract` type explicitly. In ConCert, the `WeakContract` type is defined coinductively with the `ActionBody` type, so we'll write it here as an inductive, rather than a record, type.

```
1  Inductive WeakContract :=
2      | build_weak_contract
3          (init :
4            Chain ->
5            ContractCallContext ->
6            SerializedValue (* setup *) ->
7            result SerializedValue SerializedValue)
8          (receive :
9            Chain ->
10           ContractCallContext ->
11           SerializedValue (* state *) ->
12           option SerializedValue (* message *) ->
13           result (SerializedValue * list ActionBody) SerializedValue).
```

The Blockchain in ConCert

In ConCert, the blockchain is modelled at a several levels of abstraction, which we will go through here. Underlying everything is a typeclass, `ChainBase`, which represents basic assumptions made of any blockchain. This is almost always abstracted away in any reasoning about smart contracts; we will also abstract over it as we develop our metatheory.

```
Class ChainBase :=
  build_chain_base {
    Address : Type;
    address_eqb : Address -> Address -> bool;
    address_eqb_spec :
      forall (a b : Address), Bool.reflect (a = b) (address_eqb a b);
    address_eqdec :> stdpp.base.EqDecision Address;
    address_countable :> countable.Countable Address;
    address_serializable :> Serializable Address;
    address_is_contract : Address -> bool;
  }.

```

The typeclass `ChainBase` carries basic assumptions, such that there is an address type `Address`, which is countable and has decidable equality, and which has a distinction between wallet address and contract addresses. For example, on Tezos, this distinction can be seen in the format of the public keys, where contract addresses are of the form `KT...` and wallet addresse are of the form `tz...`.

At the next level of abstraction, we have the record type `Chain`, which is similarly minimalist.

```
Record Chain :=
  build_chain {
    chain_height : nat;
    current_slot : nat;
    finalized_height : nat;
  }.
```

The only information this type carries is about the chain height, the current slot of a given block, and the finalized height (which can be different from the chain height).

From here, we have two of our most important types: the `Environment` type, which augments the `Chain` type, and the `ChainState` type, which augments the `Environment` type. Let us take a look first at the `Environment` type.

```
Record Environment :=
  build_env {
```

```
    env_chain :> Chain;
    env_account_balances : Address -> Amount;
    env_contracts : Address -> option WeakContract;
    env_contract_states : Address -> option SerializedValue;
  }.
```

We can see that, in addition to the data carried by the `Chain` type, the `Environment` type carries data about account balances, which contracts are at which addresses, and what those contract states are.

Moving up, we have `ChainState`, which just augments the `Environment` type by adding a queue of actions to be evaluated. In terms of a blockchain, this shifts the view from the chain's environment at any given moment to the state of the chain itself, executing transactions in a block. As we will see, the semantics of adding a block are to add transactions to the chain state queue.

```
Record ChainState :=
  build_chain_state {
    chain_state_env :> Environment;
    chain_state_queue : list Action;
  }.
```

Finally, we have `ChainBuilderType`, which is a typeclass representing implementations of blockchains. Part of the trust base of ConCert, then, is that the blockchain in question satisfies the semantics of the `ChainBuilderType`.

```
Class ChainBuilderType :=
  build_builder {
    builder_type : Type;

    builder_initial : builder_type;

    builder_env : builder_type -> Environment;

    builder_add_block
      (builder : builder_type)
      (header : BlockHeader)
      (actions : list Action) :
      result builder_type AddBlockError;

    builder_trace (builder : builder_type) :
      ChainTrace empty_state (build_chain_state (builder_env builder) []);
  }.
```

Blockchain Semantics in ConCert

The types which model the blockchain can be acted on by transactions, which represents the blockchain making progress by executing transactions in a block. Some of these can be initiated by users, and others relate to the blockchain's execution semantics. The possible actions that a user can initiate are modeled by the `Action` and `ActionBody` types.

```
Record Action :=
  build_act {
    act_origin : Address;
    act_from : Address;
    act_body : ActionBody;
  }.
```

```
Inductive ActionBody :=
  | act_transfer (to : Address) (amount : Amount)
  | act_call (to : Address) (amount : Amount) (msg : SerializedValue)
  | act_deploy (amount : Amount) (c : WeakContract) (setup : SerializedValue).
```

Every action carries with it the origin, as `act_origin`, the sender, as `act_from`, and what kind of action it is, whether it be a transfer, a contract call, or a contract deployment. From these we can build the types which act on the `Environment` and `ChainState` types to model the blockchain making progress.

First, let us look at the `ActionEvaluation` type, which acts on the `Environment` type. The definition of `ActionEvaluation` uses sixty-six lines of code, so we will give a shortened version here.

```
Inductive ActionEvaluation
          (prev_env : Environment) (act : Action)
          (new_env : Environment) (new_acts : list Action) : Type :=
  | eval_transfer :
      forall (origin from_addr to_addr : Address)
             (amount : Amount),
        (* some omitted checks *)
        ActionEvaluation prev_env act new_env new_acts
  | eval_deploy :
      forall (origin from_addr to_addr : Address)
             (amount : Amount)
             (wc : WeakContract)
             (setup : SerializedValue)
             (state : SerializedValue),
      (* some omitted checks *)
      ActionEvaluation prev_env act new_env new_acts
  | eval_call :
      forall (origin from_addr to_addr : Address)
```

```
19              (amount : Amount)
20              (wc : WeakContract)
21              (msg : option SerializedValue)
22              (prev_state : SerializedValue)
23              (new_state : SerializedValue)
24              (resp_acts : list ActionBody),
25       (* some omitted checks *)
26       ActionEvaluation prev_env act new_env new_acts.
```

Note first that the type is parameterized by a previous environment `prev_env`, and action `act`, a new environment `new_env`, and a list of actions `new_acts`. This means that an inhabitant of `ActionEvaluation` links a pair of successive environments, the action which links them, and a list of actions which is emitted by executing the transaction. This type parameterization is how the *chain* part of the blockchain is modeled in ConCert, as we will see again shortly with the `ChainStep` type.

Next, an environment can be updated in three ways, given by the three constructors of `ActionEvaluation` as an inductive type. These are transfers, given by `eval_transfer`, deploy actions, given by `eval_deploy`, and contract calls, given by `eval_call`. This indicates that the chain's environment can only be updated by the kinds of actions that users can initiate in practice, as would be expected.

Moving up to the `ChainState` type, we have the `ChainStep` which acts on `ChainState` similar to how `ActionEvaluation` acts on `Environment`, forming a chain. As before, we'll give a shortened version of the type definition.

```
1  Inductive ChainStep (prev_bstate : ChainState) (next_bstate : ChainState) :=
2    | step_block :
3        forall (header : BlockHeader),
4          (* some omitted checks *)
5          ChainStep prev_bstate next_bstate
6    | step_action :
7        forall (act : Action)
8               (acts : list Action)
9               (new_acts : list Action),
10         ActionEvaluation prev_bstate act next_bstate new_acts ->
11         (* some omitted checks *)
12         ChainStep prev_bstate next_bstate
13   | step_action_invalid :
14       forall (act : Action)
15              (acts : list Action),
16         (* some omitted checks *)
17         ChainStep prev_bstate next_bstate
18   | step_permute :
19         EnvironmentEquiv next_bstate prev_bstate ->
```

```
20          Permutation (chain_state_queue prev_bstate) (chain_state_queue next_bstate) ->
21          ChainStep prev_bstate next_bstate.
```

Similar to before, note first that the `ChainStep` type is parameterized by two chain states, the previous state `prev_bstate`, and the new state, `next_bstate`. The chain's state can be updated by evaluating actions, as given by `step_action`, which we note requires an inhabitant of an `ActionEvaluation` type. However, it can also be updated by adding a block, given by `step_block` or by showing an action to be invalid, given by `setp_action_invalid`.

It can also be updated by reordering the blockchain's transaction queue. This is part of the semantics for the sake of generality, so that proofs are independent of depth-first or breadth-first transaction execution orderings, which can vary among chains.

Finally, the actual chained history of a blockchain is modeled through the `ChainTrace` type, which is a linked list of inhabitants of `ChainState`, linked by inhabitants of `ChainStep`.

```
1   Definition ChainTrace := ChainedList ChainState ChainStep.
```

As we will see, the semantics of blockchain execution makes it possible for us to reason along execution traces of blockchains in a general way. We will exploit this strength throughout this thesis.

Specifications and Proofs

Now that we have smart contracts and their full execution semantics, let us look at what a specification and proof look like in ConCert.

A specification is simply a statement, written in Coq, about a smart contract. For practical verification work, a specification will reference a specific smart contract; however, there is nothing stopping us from abstracting over smart contracts, which we will do in later chapters.

For now, let us look at a simple example of contract definition and specification. The contract in question will simply be a counter contract, which can increment and decrement a counter held in storage. We start by defining the `State`, `Msg`, `Setup`, and `Error` types.

```
1 Record State :=
2   build_state {
3       stor : nat
4   }.
5
6 Inductive Msg :=
```

```
| incr (n : nat)
| decr (n : nat).

Definition Error : Type := N.

Definition Setup := unit.
```

We then define the entrypoint contracts and the contract's main functionality.

```
(* entrypoint functions *)
Definition incr_funct (n : nat) (st : State) :=
    {| stor := st.(stor) + n |}.
Definition decr_funct (n : nat) (st : State) :=
    {| stor := sub st.(stor) n |}.

(* main contract functionality *)
Definition counter_funct (st : State) (msg : Msg) : option State :=
    match msg with
    | incr n => Some (incr_funct n st)
    | decr n => Some (decr_funct n st)
    end.
```

And finally, we can construct an inhabitant of `Contract`.

```
  (* init and recv functions *)
  Definition counter_init
      (_ : Chain)
      (_ : ContractCallContext)
      (_ : Setup) :
      option State :=
          Some ({| stor := 0 |}).

  Definition counter_recv
      (_ : Chain)
      (_ : ContractCallContext)
      (st  : State)
      (op_msg : option Msg) :
      option (State * list ActionBody) :=
          match op_msg with
          | Some msg =>
              match counter_funct st msg with
              | Some rslt => Some (rslt, [])
              | None => None
              end
          | None => None
```

```
22          end.
23
24  Definition counter_contract : Contract Setup Msg State Error :=
25    build_contract counter_init counter_recv.
```

Now that we have a contract defined, `counter_contract`, we can prove things about it.

For example, we may wish to verify the property that at any given blockchain state, the value of `stor` in the state of `counter_contract` will equal the sum of the `incr` calls, minus the sum of the `decr` calls. In ConCert, we would write that statement like this:

```
1   Theorem counter_correct : forall bstate caddr (trace : ChainTrace empty_state bstate),
2     env_contracts caddr = Some (counter_contract : WeakContract) ->
3     exists cstate inc_calls,
4       contract_state bstate caddr = Some cstate /\
5       incoming_calls entrypoint trace caddr = Some inc_calls ->
6       (let sum_incr :=
7           sumN get_incr_qty inc_calls in
8       let sum_decr :=
9           sumN get_decr_qty inc_calls in
10      cstate.(stor) = sum_incr - sum_decr).
```

The theorem uses two functions, `get_incr_qty` and `get_decr_qty`, whose definition we omit here but which extract from an incoming call the quantity to be incremented or decremented. Translating this theorem in a prose format, we would say something like:

> Forall blockchain states `bstate`, contract addresses `caddr`, and chain traces `trace` from the genesis block to `bstate`, such that `caddr` is the contract address of `counter_contract`, there exists a contract state `cstate` and incoming calls `inc_calls`, such that `cstate` is the state of `counter_contract` at `bstate`, and `inc_calls` is all the incoming calls found in `trace`, such that: the value of `stor` in `cstate` is the sum of all the values of calls to the `incr` entrypoint, minus the sum of all the values of calls to the `decr` entrypoint.

Shortened from there, this theorem states that at any reachable state, the value of `stor` in the storage of `counter_contract` is the sum of all the `incr` calls minus the sum of all the `decr` calls.

Now that we have our specification formally stated as a Coq theorem, we can prove it using various Coq tactics. In ConCert, principal among those is the custom tactic `contract_induction`, which divides the proof of a contract invariant into seven subgoals, each of which relates to the various ways a blockchain can make progress, as defined in the blockchain semantics.

Metaspecifications

Metaspecifications are a contribution of this work, but we will describe how they work in relation to specifications we've just given here.

In practice, specifications are written in a Coq module as a list of lemmas and theorems. In order to codify a specification of a specification, we need to create an interface for the specification so we can reason about an arbitrary contract `C` satisfying some specification `S`. We will introduce a predicate-style specification, which allows us to do this.

The metaspecification is then analogous to the specification—a list of lemmas and theorems, proved about an arbitrary contract `C` with a proof of the specification predicate `S`.

3.2.1 Contributions

In Chapter 4

- ...

In Chapter 5

- ...

In Chapter 6

- ...

Chapter 4

Specifications and Metaspecifications

In this chapter we develop the notion of correctness of a contract specification in order to address economic vulnerabilities like those in §2.1.1. A contract specification is correct if conformance to the specification implies the desired properties (*e.g.* economic properties) indended by the specification. This could include a resilience against front-running attacks, or economic properties such as incentive compatibility or demand sensitivy which we saw in §2.2, and is specified with a *metaspecification*—a specification of a specification. We will illustrate with an example economic smart contract, called a structured pool, by first specifying the contract and then reasoning about its emergent economic behaviors with a metaspecification.

In contrast to later chapters, this work does not require any explicit additions or modifications to ConCert. Instead, it proposes conventions in how we write and reason about specifications when we verify financial smart contracts.

We proceed by first introducing our example, structured pools, in §4.1. In §4.2, we give our contract specification, first informally and then formally. In §4.3, we give our contract metaspecification, first informally and then formally. In §4.4 we include a short discussion, and we conclude in §4.5.

4.1 Introducing Structured Pools

This chapter revolves around the specification of a novel financial smart contract called a *structured pool*, which is designed to address issues in pooling and trading tokenized carbon credits on the blockchain [178]. The specification of this contract will describe the contract interface and entrypoints, storage, and entrypoint functions, while the metaspecification will include economic properties characteristic of well-behaved AMMs in the wild, derived from the literature.

4.1.1 Introducing Structured Pools

Tokenized carbon credits, which are of growing relevance to voluntary carbon markets [15] [83] [86] [176], have unique metadata and are typically tokenized as non-fungible tokens (NFTs). However, this can lead to low liquidity and high price volatility, so there is a push within the industry to make carbon credits as fungible as possible [87] [151] [179].

The current solution is to pool carbon credits which have similar features, such as a specific vintage or crediting methodology, and to value each pooled token equally within the pool [179]. From a valuation perspective, this discards the differences in constituent credits.

For example, Toucan, perhaps the most prominent provider of tokenized carbon credits [176], tokenizes carbon credits from the Verra registry as NFTs on Polygon [14]. Each NFT can then be fractionalized as an ERC20 token using a so-called TCO2 token contract. Distinct TCO2 contracts are not mutually fungible because they carry the metadata of the carbon credit they fractionalize. To achieve mutual fungibility, Toucan launched the Base Carbon Tonne (BCT) and the Nature Carbon Tonne (NCT) pools. Any TCO2 token which satisfies the acceptance criteria of one of these pools can be pooled, one-for-one, in exchange for BCT or NCT tokens, respectively. While these pools do increase token fungibility, they do so at the cost of valuing individiaul metadata.

Structured pools go further than the status quo by removing the required one-for-one exchange rate, thereby increasing liquidity without ignoring differences in constituent credits. Drawing on commonly-used AMMs, they value constituent tokens relative to each other by enabling trading between them [178].

4.2 The Contract Specification

Our specification defines required parameters of the storage and entrypoint types, and gives functional specifications of each entrypoint function. This follows the style of previous work on novel AMMs, in particular A^{2}MM [206] and Fair AMM [77] which were designed to mitigate front-running attacks, as well as the Dexter2 specification [6]. The specification is somewhat low-level and is intended to be as concise and straightforward to implement as possible.

All of this is formalized in ConCert, so as we go along we will first describe the specification using prose, and then present its corresponding formalization.

4.2.1 Storage Specification

The structured pool contract pools tokens and facilitates trades between them, so it must keep track of:

- a **family T of constituent tokens** which can be pooled by the contract, where the data of a constituent token t_x is its contract address of type `address` and ID of type `nat`,
- an **exchange rate** r_x for each token t_x, assumed here to be strictly positive when deployed, which indicates that a quantity q of tokens in t_x can be pooled for $q \cdot r_x$ pool tokens,
- the **balance** x that the structured pool contract holds for each constituent token t_x in the family T,
- the **address of the pool token contract**, for which the structured pool contract has minting and burning privileges,
- and k, the **total number of pool tokens in circulation**.

This can be formalized in ConCert with a typeclass which calls for four functions which retrieve: the exchange rates for each token, the balances the contract holds in each token, the pool token data, and the number of outstanding tokens. The family T of constituent tokens is defined implicitly to be the tokens which have an entry in the rates map `stor_rates`.

```
Class State_Spec (T : Type) :=
    build_state_spec {
      (* the exchange rates *)
      stor_rates : T -> FMap token exchange_rate ;
      (* token balances *)
      stor_tokens_held : T -> FMap token N ;
      (* pool token data *)
      stor_pool_token : T -> token ;
      (* number of outstanding pool tokens *)
      stor_outstanding_tokens : T -> N ;
  }.
```

Here, `token` and `exchange_rate` are two types both given by

```
(* token data type *)
Record token := { token_address : Address; token_id : N }.

(* exchange rates in N.
   a typical implementation may exchange rates by 1_000_000 to approximate rational numbers *)
Definition exchange_rate := N.
```

For example, the contract state could be implemented as a record type with four entries such as the following.

```
Record storage := {
    rates : FMap token exchange_rate ;
    tokens_held : FMap token N ;
    pool_token : token ;
    outstanding_tokens : N ;
}.
```

This gives an implementation of the typeclass `State_Spec` simply with the constructors of the record type. However, the typeclass does not require that it be this way, and storage can be implemented in any way such that we have access to the data specified. In particular, the storage could keep track of more data, such as an admin or contract metadata.

Regarding the contract storage at initialization, we formally specify that the contract initialize with strictly positive exchange rates, zero token balances (*i.e.* no tokens yet pooled), and zero outstanding tokens (*i.e.* no pool tokens yet issued).

```
Definition initialized_with_nonzero_rates (contract : Contract Setup Msg State Error) :=
    forall chain ctx setup cstate,
    (* If the contract initializes successfully *)
    init contract chain ctx setup = Ok cstate ->
    (* then all rates are nonzero *)
    forall t r,
    FMap.find t (stor_rates cstate) = Some r ->
    r > 0.

Definition initialized_with_zero_balance (contract : Contract Setup Msg State Error) :=
    forall chain ctx setup cstate,
    (* If the contract initializes successfully *)
    init contract chain ctx setup = Ok cstate ->
    (* then all token balances initialize to zero *)
    forall t,
    get_bal t (stor_tokens_held cstate) = 0.

Definition initialized_with_zero_outstanding (contract : Contract Setup Msg State Error) :=
    forall chain ctx setup cstate,
    (* If the contract initializes successfully *)
    init contract chain ctx setup = Ok cstate ->
    (* then there are no outstanding tokens *)
    stor_outstanding_tokens cstate = 0.
```

4.2.2 Entrypoint Specification

The structured pool contract features at least three entrypoints: `POOL`, for pooling tokens, which deposits liquidity for trading; `UNPOOL`, for unpooling tokens, which withdraws liquidity from trading; and `TRADE`, for trading between constitutent, pooled tokens. Each of these entrypoints are governed by predetermined equations which price trades and which update the pooling exchange rates which we give below.

We can again formalize this with a typeclass, where `pool` corresponds to the `POOL` entrypoint, `unpool` corresponds to the `UNPOOL` entrypoint, and `trade` corresponds to the `TRADE` entrypoint.

```
Class Msg_Spec (T : Type) :=
  build_msg_spec {
    pool : pool_data -> T ;
    unpool : unpool_data -> T ;
    trade : trade_data -> T ;
    (* any other potential entrypoints *)
    other : other_entrypoint -> option T ;
}.
```

The entrypoint interfaces are defined by the `pool_data`, `unpool_data`, and `trade_data` types, which are the types of messages accepted by the `POOL`, `UNPOOL`, and `TRADE` entrypoints, respectively.

The `POOL` entrypoint accepts token data of some t_x and some quantity q of tokens in t_x to be pooled.

```
Record pool_data := {
    token_pooled : token ;
    qty_pooled : N ; (* the qty of tokens to be pooled *)
}.
```

The `UNPOOL` entrypoint accepts token data of some t_x and some quantity q of pool tokens the user wishes to burn in exchange for the token t_x.

```
Record unpool_data := {
    token_unpooled : token ;
    qty_unpooled : N ; (* the qty of pool tokens being turned in *)
}.
```

The `TRADE` entrypont accepts token data of some t_x to be traded in, token data of some t_y to be traded out, and the quantity Δ_x of t_x to be traded in.

```
Record trade_data := {
    token_in_trade : token ;
    token_out_trade : token ;
```

```
    qty_trade : N ; (* the qty of token_in going in *)
}.
```

We require that the token data of each of these entrypoints must correspond to tokens in T. This is formalized in three propositions of the specification.

```
(* A successful call to POOL means that token_pooled has an exchange rate (=> is in T) *)
Definition pool_entrypoint_check (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate cstate' chain ctx msg_payload acts,
    (* a successful call *)
    receive contract chain ctx cstate (Some (pool (msg_payload))) = Ok(cstate', acts) ->
    (* an exchange rate exists *)
    exists r_x,
    FMap.find msg_payload.(token_pooled) (stor_rates cstate) = Some r_x.

(* A successful call to UNPOOL means that token_pooled has an exchange rate (=> is in T) *)
Definition unpool_entrypoint_check (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate cstate' chain ctx msg_payload acts,
    (* a successful call *)
    receive contract chain ctx cstate (Some (unpool (msg_payload))) = Ok(cstate', acts) ->
    (* an exchange rate exists *)
    exists r_x,
    FMap.find msg_payload.(token_unpooled) (stor_rates cstate) = Some r_x.

(* A successful call to TRADE means that token_in_trade and token_out_trade have exchange
    rates (=> are in T) *)
Definition trade_entrypoint_check (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate' acts,
    (* a successful call *)
    receive contract chain ctx cstate (Some (trade (msg_payload))) = Ok(cstate', acts) ->
    (* exchange rates exist *)
    exists y r_x r_y,
    ((FMap.find msg_payload.(token_out_trade) (stor_tokens_held cstate) = Some y) /\
    (FMap.find msg_payload.(token_in_trade) (stor_rates cstate) = Some r_x) /\
    (FMap.find msg_payload.(token_out_trade) (stor_rates cstate) = Some r_y)).
```

Finally, the `Msg_Spec` typeclass includes a function `other`, which represents all other possible entrypoints an implementation could have, *e.g.* to perform admin duties, or some upgradeability or governance features. We include this so that we can reason about contract invariants despite not having all the entrypoints at hand. To do so, we specify that anything in a type satisfying `Msg_Spec` can be characterized by the associated constructors. This will allow us to mimic proof by induction on our entrypoint type.

```
Definition msg_destruct (contract : Contract Setup Msg State Error) : Prop :=
    forall (m : Msg),
```

```
3      (exists p, m = pool p) \/
4      (exists u, m = unpool u) \/
5      (exists t, m = trade t) \/
6      (exists o, Some m = other o).
```

4.2.3 The Pool Entrypoint

The `POOL` entrypoint accepts the token data of some t_x from the token family and a quantity q of tokens in t_x to be pooled. The pool contract checks that t_x is in the token family. The pool contract then transfers q tokens of t_x to itself, wich is done by calling the `transfer` entrypoint of the token t_x, which is a standard entrypoint of token contracts. It simultaneously mints $q * r_x$ pool tokens and transfers them to the sender's wallet. This transaction is atomic, meaning that if any of the `transfer` or `mint` operations fail, the entire transaction fails.

```
1  (* pseudocode of the POOL entrypoint function *)
2  fn POOL t_x q =
3      if (is_in_family t_x)
4      then
5          <atomic>
6              transfer (q) of (t_x) from (sender) to (self) ;
7              mint (q * r_x) of (pool_token) and transfer to (sender) ;
8          </atomic>
9      else
10         fail ;
```

Listing 4.1: the `POOL` entrypoint function.

The Pool entrypoint, formalized

We now distill this into a formal specification.

We already saw that the contract accepts data of a token t_x and a quantity q by the definition of `pool_data`:

```
1  Record pool_data := {
2      token_pooled : token ;
3      qty_pooled : N ; (* the qty of tokens to be pooled *)
4  }.
```

We also already saw with `pool_entrypoint_check` that the `POOL` entrypoint checks that the token in question is in our token family T, failing otherwise. What remains is to formally specify the outgoing transactions of

a successful call to POOL, as well as the resulting changes to the storage.

First we specify that a successful call to POOL results in the appropriate transactions with the proposition in the specification.

```
(* When the POOL entrypoint is successfully called, it emits a TRANSFER call to the
    token in storage, with q tokens in the payload of the call *)
Definition pool_emits_txns (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate' acts,
    (* the call to POOL was successful *)
    receive contract chain ctx cstate (Some (pool (msg_payload))) = Ok(cstate', acts) ->
    (* in the acts list there is a transfer call with q tokens as the payload *)
    exists transfer_to transfer_data transfer_payload mint_data mint_payload,
    (* there is a transfer call *)
    let transfer_call := (act_call
        (* calls the pooled token address *)
        (msg_payload.(token_pooled).(token_address))
        (* with amount 0 *)
        0
        (* and payload transfer_payload *)
        (serialize (FA2Spec.Transfer transfer_payload))) in
    (* with a transfer in it *)
    In transfer_data transfer_payload /\
    (* which itself has transfer data *)
    In transfer_to transfer_data.(FA2Spec.txs) /\
    (* whose quantity is the quantity pooled *)
    transfer_to.(FA2Spec.amount) = msg_payload.(qty_pooled) /\
    (* there is a mint call in acts *)
    let mint_call := (act_call
        (* calls the pool token contract *)
        (stor_pool_token cstate).(token_address)
        (* with amount 0 *)
        0
        (* and payload mint_payload *)
        (serialize (FA2Spec.Mint mint_payload))) in
    (* with has mint_data in the payload *)
    In mint_data mint_payload /\
    (* and the mint data has these properties: *)
    let r_x := get_rate msg_payload.(token_pooled) (stor_rates cstate) in
    mint_data.(FA2Spec.qty) = msg_payload.(qty_pooled) * r_x /\
    mint_data.(FA2Spec.mint_owner) = ctx.(ctx_from) /\
    (* these are the only emitted transactions *)
    (acts = [ transfer_call ; mint_call ] \/
    acts = [ mint_call ; transfer_call ]).
```

The specified atomicity of these two actions is given by the blockchain semantics as encoded in ConCert: the entire transaction fails if either of the `mint_call` or `transfer_call` fail.

Finally, we characterize changes to the storage with three properties. These are that the ledger of token balances updates appropriately, that the rates remain unchanged, and that the outstanding tokens counter updates appropriately.

```
(* When the POOL entrypoint is successfully called, tokens_held goes up appropriately *)
Definition pool_increases_tokens_held (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate' acts,
    (* the call to POOL was successful *)
    receive contract chain ctx cstate (Some (pool msg_payload)) = Ok(cstate', acts) ->
    (* in cstate', tokens_held has increased at token *)
    let token := msg_payload.(token_pooled) in
    let qty := msg_payload.(qty_pooled) in
    let old_bal := get_bal token (stor_tokens_held cstate) in
    let new_bal := get_bal token (stor_tokens_held cstate') in
    new_bal = old_bal + qty /\
    forall t,
    t <> token ->
    get_bal t (stor_tokens_held cstate) =
    get_bal t (stor_tokens_held cstate').

(* And the rates don't change *)
Definition pool_rates_unchanged (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate cstate' chain ctx msg_payload acts,
    (* the call to POOL was successful *)
    receive contract chain ctx cstate (Some (pool msg_payload)) = Ok(cstate', acts) ->
    (* rates all stay the same *)
    forall t,
    FMap.find t (stor_rates cstate) = FMap.find t (stor_rates cstate').

(* The outstanding tokens change appropriately *)
Definition pool_outstanding (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate cstate' chain ctx msg_payload acts,
    (* the call to POOL was successful *)
    receive contract chain ctx cstate (Some (pool msg_payload)) = Ok(cstate', acts) ->
    (* rates all stay the same *)
    let rate_in := get_rate msg_payload.(token_pooled) (stor_rates cstate) in
    let qty := msg_payload.(qty_pooled) in
    stor_outstanding_tokens cstate' =
    stor_outstanding_tokens cstate + rate_in / 1000000 * qty.
```

4.2.4 The Unpool Entrypoint

The UNPOOL entrypoint accepts token data of some t_x from the token family and a quantity q of pool tokens the user wishes to burn in exchange for tokens in t_x. The pool contract checks that t_x is in the token family, and checks that it has sufficient tokens in t_x to execute the withdrawal transaction. The pool contract then transfers q pool tokens from the sender to itself and burns them by calling the `burn` entrypoint, a standard entrypoint of token contracts. It simultaneously transfers $\frac{q}{r_x}$ tokens in t_x from itself to the sender's wallet. As before, the transaction is atomic, so if any of the `transfer` or `burn` operations fail, the entire transaction fails.

```
(* pseudocode of the UNPOOL entrypoint function *)
fn UNPOOL t_x q =
    if (is_in_family t_x) && (self_balance t_x >= q / r_x)
    then
        <atomic>
            transfer (q) of (pool_token) from (sender) to (self) and burn ;
            transfer (q / r_x) of (t_x) from (self) to (sender) ;
        </atomic>
    else
        fail ;
```

Listing 4.2: the UNPOOL entrypoint function.

The Unpool entrypoint, formalized

We now distill this into a formal specification.

We already saw that the contract accepts data of a token t_x and a quantity q by the definition of `unpool_data`:

```
Record unpool_data := {
    token_unpooled : token ;
    qty_unpooled : N ; (* the qty of pool tokens being turned in *)
}.
```

We also already saw with `unpool_entrypoint_check` that the UNPOOL entrypoint checks that the token in question is in our token family T, failing otherwise. What remains is to formally specify the check that the contract has enough in reserves to execute the unpool action, the outgoing transactions of a successful call to UNPOOL, as well as the resulting changes to the storage.

First we add an entrypoint check to the specification to ensure that the contract has sufficient funds to execute the unpool action.

```
Definition unpool_entrypoint_check_2 (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate cstate' chain ctx msg_payload acts,
    (* a successful call *)
    receive contract chain ctx cstate (Some (unpool (msg_payload))) = Ok(cstate', acts) ->
    (* we don't unpool more than we have in reserves *)
    qty_unpooled msg_payload <=
    get_rate (token_unpooled msg_payload) (stor_rates cstate) * get_bal (token_unpooled
    msg_payload) (stor_tokens_held cstate).
```

We then specify that a successful call to `UNPOOL` results in the appropriate transactions with the proposition in the specification.

```
(* When the UNPOOL entrypoint is successfully called, it emits a BURN call to the
    pool_token, with q in the payload *)
Definition unpool_emits_txns (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate' acts,
    (* the call to UNPOOL was successful *)
    receive contract chain ctx cstate (Some (unpool msg_payload)) = Ok(cstate', acts) ->
    (* in the acts list there are burn and transfer transactions *)
    exists burn_data burn_payload transfer_to transfer_data transfer_payload,
    (* there is a burn call in acts *)
    let burn_call := (act_call
        (* calls the pool token address *)
        (stor_pool_token cstate).(token_address)
        (* with amount 0 *)
        0
        (* with payload burn_payload *)
        (serialize (FA2Spec.Retire burn_payload))) in
    (* with has burn_data in the payload *)
    In burn_data burn_payload /\
    (* and burn_data has these properties: *)
    burn_data.(FA2Spec.retire_amount) = msg_payload.(qty_unpooled) /\
    (* the burned tokens go from the unpooler *)
    burn_data.(FA2Spec.retiring_party) = ctx.(ctx_from) /\
    (* there is a transfer call *)
    let transfer_call := (act_call
        (* call to the token address *)
        (msg_payload.(token_unpooled).(token_address))
        (* with amount = 0 *)
        0
        (* with payload transfer_payload *)
        (serialize (FA2Spec.Transfer transfer_payload))) in
    (* with a transfer in it *)
    In transfer_data transfer_payload /\
    (* which itself has transfer data *)
```

```
    In transfer_to transfer_data.(FA2Spec.txs) /\
    (* whose quantity is the quantity pooled *)
    let r_x := get_rate msg_payload.(token_unpooled) (stor_rates cstate) in
    transfer_to.(FA2Spec.amount) = msg_payload.(qty_unpooled) / r_x /\
    (* and these are the only emitted transactions *)
    (acts = [ burn_call ; transfer_call ] \/
     acts = [ transfer_call ; burn_call ]).
```

The specified atomicity of these two actions is given by the blockchain semantics as encoded in ConCert: the entire transaction fails if either of the `burn_call` or `transfer_call` fail.

Finally, we characterize changes to the storage with three properties. These are that the ledger of token balances updates appropriately, that the rates remain unchanged, and that the outstanding tokens counter updates appropriately.

```
(* When the UNPOOL entrypoint is successfully called, tokens_held goes down appropriately *)
Definition unpool_decreases_tokens_held (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate' acts,
    (* the call to POOL was successful *)
    receive contract chain ctx cstate (Some (unpool msg_payload)) = Ok(cstate', acts) ->
    (* in cstate', tokens_held has increased at token *)
    let token := msg_payload.(token_unpooled) in
    let r_x := get_rate token (stor_rates cstate) in
    let qty := calc_rx_inv r_x msg_payload.(qty_unpooled) in
    let old_bal := get_bal token (stor_tokens_held cstate) in
    let new_bal := get_bal token (stor_tokens_held cstate') in
    new_bal = old_bal - qty /\
    forall t,
    t <> token ->
    get_bal t (stor_tokens_held cstate) =
    get_bal t (stor_tokens_held cstate').

(* When the UNPOOL entrypoint is successfully called, tokens_held goes down appropriately *)
Definition unpool_rates_unchanged (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate cstate' chain ctx msg_payload acts,
    (* the call to POOL was successful *)
    receive contract chain ctx cstate (Some (unpool msg_payload)) = Ok(cstate', acts) ->
    (* rates all stay the same *)
    forall t,
    FMap.find t (stor_rates cstate) = FMap.find t (stor_rates cstate').

(* Defines how the UNPOOL entrypoint updates outstanding tokens *)
Definition unpool_outstanding (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate cstate' chain ctx msg_payload acts,
    (* the call to POOL was successful *)
```

```
31      receive contract chain ctx cstate (Some (unpool msg_payload)) = Ok(cstate', acts) ->
32      (* rates all stay the same *)
33      let rate_in := get_rate msg_payload.(token_unpooled) (stor_rates cstate) in
34      let qty := msg_payload.(qty_unpooled) in
35      stor_outstanding_tokens cstate' =
36      stor_outstanding_tokens cstate - qty.
```

Finally, note that the pseudocode of Listing 4.2, we divide by the exchange rate `r_x`. This is meant to approximate division of the rational numbers. In particular, it is meant to calculate the inverse of the calculation of the pool entrypoint, such that pool and unpool are inverse operations. However, smart contracts typically do integer division, which carries a margin of error. Rather than using integer division, to calculate the quantity of tokens unpool, we use a function which is meant to approximate rational division by the exchange rate.

```
1 (* given an exchange rate and a quantity of pool tokens, calculate the quantity unpooled *)
2 calc_rx_inv : forall (r_x : N) (q : N), N
```

This function must act as a right inverse on multiplication by pooling exchange rates, which we include in the specification.

```
1 (* the inverse rate function is a right inverse of r_x *)
2 Definition rates_balance :=
3     forall q t rates prev_state,
4     let r_x := get_rate t rates in
5     let x := get_bal t (stor_tokens_held prev_state) in
6     r_x * calc_rx_inv r_x q = q.
```

4.2.5 The Trade Entrypoint

The `TRADE` entrypoint takes the token data of some token t_x in T to be traded in, the token data of some token t_y in T to be traded out, and the quantity Δ_x to be traded. It checks that both t_x and t_y are in the token family, that $k > 0$, and that $\Delta_x > 0$. It calculates Δ_y using formulae we will give below, and checks that it has a sufficient balance y in t_y to execute the trade action. Then in an atomic transaction, the contract updates the exchange rate r_x in response to the trade, transfers Δ_x of tokens t_x from the sender's wallet to itself, and transfers Δ_y of tokens t_y from itself to the sender's wallet.

The contract prices trades by simulating trading along the curve $xy = k$ (for some generic x and y), where k is the total number of outstanding pool tokens. A trade of Δ_x yields Δ_y tokens such that the following

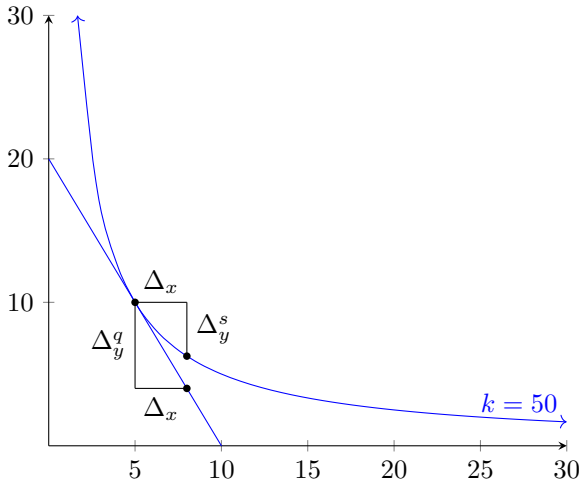


Figure 4.1: A trade of $\Delta_x = 3$ for Δ_y^q and Δ_y^s, respectively, at $k = 50$. $\Delta_y^q = p_q \Delta_x$ is the trade priced at the *quoted price* p_q and $\Delta_y^s = p_s \Delta_x$ is the trade priced at the *swap price* p_s.

equation holds:

$$(x + \Delta_x)(y - \Delta_y) = k, \tag{4.1}$$

giving

$$\Delta_y = y - \frac{k}{x + \Delta_x}. \tag{4.2}$$

This is how trades are priced in the wild for liquidity pools of fungible tokens [31]. We call $p_s = \frac{\Delta_y}{\Delta_x}$ the *swap price*.

An important consequence to (4.1) is that the smaller Δ_x is compared to k, the closer the exchange happens at a rate of $p_q = \frac{y}{x}$. This is because the derivative of $xy = k$, or $f(x) = \frac{k}{x}$, is

$$f'(x) = \frac{-k}{x^2} = \frac{-y}{x},$$

and the smaller Δ_x is relative to k, the more accurately the tangent line at some (x_0, y_0) approximates the convex curve $xy = k$. We call $p_q = \frac{y}{x}$ the *quoted price*.

The difference $p_q - p_s$ is called the *price slippage* [197, §3.2.4]. It is important to note that p_s is always less than p_q because p_q is calculated by moving Δ_x along the tangent line from a starting point (x_0, y_0) representing the current state of the contract's funds available for trading, and p_s is calculated by moving Δ_x along $xy = k$. Since $xy = k$ is convex, moving Δ_x along the tangent line always results in a larger Δ_y than moving along $xy = k$. See Figure 4.1 for a graphical illustration, where Δ_y^q is the output of a trade priced at p_q and Δ_y^s is the output of a trade priced at p_s.

In particular, this means that

$$\Delta_y^s < p_q \Delta_x \tag{4.3}$$

always holds, since $\Delta_y^s = p_s \Delta_x$. This fact is crucial to the mechanics of how structured pools update relative prices in response to trading activity.

We use the pooling exchange rates to inform quoted prices between tokens, and then simulate trades along the curve $xy = k$ (for some generic x and y). If the token t_x pools at a rate of r_x, meaning r_x is the value of t_x in terms of pool tokens, and the token t_y pools at a rate of r_y, then t_x can be valued relative to t_y at a rate of

$$r_{x,y} := \frac{r_x}{r_y}. \tag{4.4}$$

It is perhaps counterintuitive that r_x is in the numerator and not the denominator of $r_{x,y}$, considering that $p_q = \frac{y}{x}$ in the generic case, but this is due to the fact that r_x indicates pool tokens per t_x, and we want $r_{x,y}$ to indicate t_y valued in terms of t_x.

To price a trade we begin by finding ℓ such that

$$(\ell r_y)(\ell r_x) = k,$$

where k is the total number of outstanding pool tokens in the contract's storage. The trade then yields Δ_y^s tokens such that

$$(\ell r_y + \Delta_x)(\ell r_x - \Delta_y^s) = k. \tag{4.5}$$

This formula yields the quoted price of this trade as

$$p_q = \frac{\ell r_x}{\ell r_y} = \frac{r_x}{r_y} = r_{x,y}, \tag{4.6}$$

as desired. The swap price, then, is

$$p_s = \frac{\Delta_y^s}{\Delta_x} \tag{4.7}$$

where

$$\Delta_y^s = \ell r_x - \frac{k}{\ell r_y + \Delta_x}. \tag{4.8}$$

For the rest of this document, we will write Δ_y^s simply as Δ_y unless explicitly stated otherwise.

After executing a trade, if we do not adjust pooling exchange rates, the pool token is now overcollateralized. We can see this because by (4.3),

$$r_y \Delta_y < r_x \Delta_x,$$

so a trade deposits *slightly more* in terms of pool tokens ($r_x \Delta_x$) than it removes ($r_y \Delta_y$). Thus the sum of the value of all the constituent tokens at their current valuation is now greater than the total number of outstanding pool tokens.

To avoid this, we adjust the values of the constituent tokens so that their sum at the new valuation is equal to the total number of outstanding pool tokens. In a trade t_x to t_y, because it is possible to deplete t_y from

the pool, we cannot reliably regain pooled consistency by adjusting the value of the token being traded for in the pool. We know, however, that we have a supply of t_x because that was the deposited token. Thus to regain pooled consistency, we have to slightly devalue t_x in relation to the rest of the pool tokens. To do so, we divide the quantity of pool tokens by its collateral in t_x to get an updated exchange rate r'_x as follows.

$$r'_x := \frac{r_x x + r_y \Delta_y}{x + \Delta_x}. \tag{4.9}$$

Equation (4.9) updates the pooling exchange rate of t_x so that the pool token is neither under- nor over-collateralized.

Consider as an example a pool with three constituent tokens t_x, t_y, and t_z, and pooling exchange rates $r_x = 2$, $r_y = 1$, and $r_z = 1$. That is, t_x is valued at two pool tokens for one token, and each of t_y and t_z are valued at one pool token for one token. Suppose we have 10 t_x, 15 t_y, and 15 t_z pooled, thus having $20 + 15 + 15 = 50$ outstanding pool tokens.

Now suppose that we trade 1 t_x for slightly less than 2 t_y (the quoted price would be exactly 2). Using our formulae, $\ell = \sqrt{\frac{50}{2}} = 5$ and $\Delta_y \approx 1.67$ (slippage is high because of the small amount of liquidity). Post trade, we have in our pool 11 t_x, 13.33 t_y, and 15 t_z, giving us in the pool the equivalent in constituent tokens as

$$2 * 11 + 1 * 13.33 + 1 * 15 = 50.33$$

pool tokens with our unadjusted pooling exchange rates. To rectify this, bringing the pool back down to the value of 50 pool tokens, we slightly devalue t_x relative to the other tokens in the pool. We use the formula (4.9)

$$r'_x = \frac{\#\text{pool tokens}}{\#\text{tokens of } t_x} = \frac{r_x x + r_y \Delta_y}{x + \Delta_x} \approx 1.97,$$

which adjusts r_x so that the 11 t_x are now worth about 21.67 pool tokens instead of 22. This gives us our desired

$$1.97 * 11 + 1 * 13.33 + 1 * 15 = 50.$$

After this update, t_y is valued more in relation to t_x, which makes sense because t_x was sold to buy t_y. One t_y used to be worth half of t_x, and now it is valued at

$$\frac{r_y}{r'_x} \approx 0.508.$$

We need to make sure that the relative price of t_y didn't rise so much that if we trade back for t_x, we have more in t_x than we started with. If this were the case, we would have an opportunity for arbitrage within the structured pool, something we wish to avoid. The quoted price for trading our roughly 1.67 t_y back to t_x would give us about $0.508 * 1.67 \approx 0.848$, which is less than 1, as desired.

Finally, a note that in these calculations, we implicitly assumed exchange rates r_x to be rational numbers by which we can multiply and divide freely so long as $r_x > 0$. Of course, implementations (including our

formal specification) will include rounding error, and so we add to the specification that the UPDATE_RATE function return a positive number if the numerator and denominator of the quotient are positive. We also specify that the CALCULATE_TRADE function return a positive number if k, r_x, r_y, and Δ_x are positive, and that $\Delta_y < \frac{r_x}{r_y}\Delta_x$ always be true for successful trades.

The specification is summarized in Listing 4.3.

```
(* two auxiliary functions *)
fn CALCULATE_TRADE r_x r_y delta_x k =
    let l = sqrt(k / (r_x r_y)) ;
    l * r_x - k / (l * r_y + delta_x) ;

fn UPDATE_RATE x delta_x delta_y r_x r_y =
    (r_x x + r_y * delta_y)/(x + delta_x);

(* pseudocode of the TRADE entrypoint function *)
fn TRADE t_x t_y delta_x =
    let delta_y = CALCULATE_TRADE r_x r_y delta_x k ;
    if (is_in_family t_x) && (is_in_family t_y) && (self_balance t_y >= delta_y)
    then
        <atomic>
            r_x <- UPDATE_RATE x delta_x delta_y r_x r_y ;
            transfer (delta_x) of (t_x) from (sender) to (self) ;
            transfer (delta_y) of (t_y) from (self) to (sender) ;
        </atomic>
    else
        fail ;
```

Listing 4.3: the TRADE entrypoint function.

Trades, formalized

We now distill this into a formal specification.

Recall from our specification, that the pool contract checks that both t_x and t_y are in the token family, and the contract calculates the quantity Δ_y of token t_y to be traded out, and checks that it has a sufficient balance in t_y to execute the trade action. Then in an atomic transaction, the contract updates the exchange rate r_x in response to the trade, transfers Δ_x of tokens t_x from the sender's wallet to itself, and transfers Δ_y of tokens t_y from itself to the sender's wallet.

As before, we saw that the contract accepts data of tokens t_x and t_y, and a quantity q by the definition of unpool_data:

```
Record trade_data := {
    token_in_trade : token ;
    token_out_trade : token ;
    qty_trade : N ; (* the qty of token_in going in *)
}.
```

And we saw with `trade_entrypoint_check` that the `TRADE` entrypoint checks that the tokens in question are in our token family T, failing otherwise.

To proceed, we need the auxiliary functions from the pseudocode in Listing 4.3. Rather than define them explicitly, we introduce them into the context, and give a specification that any implementation must satisfy. We do this because, as in the case of the `UNPOOL` entrypoint, the functions of the pseudocode implicitly assume that division behave as it does with rational numbers, and there are a number of implementations that could satisfy the specification of Listing 4.3.

```
Context
    { calc_delta_y : forall (rate_in : N) (rate_out : N) (qty_trade : N) (k : N) (x : N), N }
    { calc_rx' : forall (rate_in : N) (rate_out : N) (qty_trade : N) (k : N) (x : N), N }.
```

The specification of these functions includes several properties, including what we saw in Equation (4.3).

```
(* r_x * delta_y <= r_x * delta_x *)
Definition trade_slippage :=
    forall r_x r_y delta_x k x,
    let delta_y := calc_delta_y r_x r_y delta_x k x in
    r_y * delta_y <= r_x * delta_x.
```

As before, we first specify that a successful call to `TRADE` results in the appropriate emitted transfer transactions.

```
(* When TRADE is successfully called, it emits two TRANSFER actions *)
Definition trade_emits_transfers (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate cstate' chain ctx msg_payload acts,
    (* the call to TRADE was successful *)
    receive contract chain ctx cstate (Some (trade (msg_payload))) = Ok(cstate', acts) ->
    (* the acts list consists of two transfer actions, specified as follows: *)
    exists transfer_to_x transfer_data_x transfer_payload_x
           transfer_to_y transfer_data_y transfer_payload_y,
    (* there is a transfer call for t_x *)
    let transfer_tx := (act_call
        (* call to the correct token address *)
        (msg_payload.(token_in_trade).(token_address))
        (* with amount = 0 *)
```

```
        0
        (* and payload transfer_payload_x *)
        (serialize (FA2Spec.Transfer transfer_payload_x))) in
    (* with a transfer in it *)
    In transfer_data_x transfer_payload_x /\
    (* which itself has transfer data *)
    In transfer_to_x transfer_data_x.(FA2Spec.txs) /\
    (* whose quantity is the quantity traded, transferred to the contract *)
    transfer_to_x.(FA2Spec.amount) = msg_payload.(qty_trade) /\
    transfer_to_x.(FA2Spec.to_) = ctx.(ctx_contract_address) /\
    (* there is a transfer call for t_y *)
    let transfer_ty := (act_call
        (* call to the correct token address *)
        (msg_payload.(token_out_trade).(token_address))
        (* with amount = 0 *)
        0
        (* and payload transfer_payload_y *)
        (serialize (FA2Spec.Transfer transfer_payload_y))) in
    (* with a transfer in it *)
    In transfer_data_y transfer_payload_y /\
    (* which itself has transfer data *)
    In transfer_to_y transfer_data_y.(FA2Spec.txs) /\
    (* whose quantity is the quantity traded, transferred to the contract *)
    let t_x := msg_payload.(token_in_trade) in
    let t_y := msg_payload.(token_out_trade) in
    let rate_in := (get_rate t_x (stor_rates cstate)) in
    let rate_out := (get_rate t_y (stor_rates cstate)) in
    let k := (stor_outstanding_tokens cstate) in
    let x := get_bal t_x (stor_tokens_held cstate) in
    let q := msg_payload.(qty_trade) in
    transfer_to_y.(FA2Spec.amount) = calc_delta_y rate_in rate_out q k x  /\
    transfer_to_y.(FA2Spec.to_) = ctx.(ctx_from) /\
    (* acts is only these two transfers *)
    (acts = [ transfer_tx ; transfer_ty ] \/
     acts = [ transfer_ty ; transfer_tx ]).
```

The specified atomicity of this entrypoint is given by the blockchain semantics as encoded in ConCert: the entire transaction fails if either of the transfer transactions fail.

Finally, we characterize changes to the storage with four properties, which characterize: how the token balances update after a trade,

```
Definition trade_tokens_held_update (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate' acts,
    (* the call to TRADE was successful *)
```

```
    receive contract chain ctx cstate (Some (trade (msg_payload))) = Ok(cstate', acts) ->
    (* in the new state *)
    let t_x := msg_payload.(token_in_trade) in
    let t_y := msg_payload.(token_out_trade) in
    let rate_in := (get_rate t_x (stor_rates cstate)) in
    let rate_out := (get_rate t_y (stor_rates cstate)) in
    let k := (stor_outstanding_tokens cstate) in
    let x := get_bal t_x (stor_tokens_held cstate) in
    let delta_x := msg_payload.(qty_trade) in
    let delta_y := calc_delta_y rate_in rate_out delta_x k x in
    let prev_bal_y := get_bal t_y (stor_tokens_held cstate) in
    let prev_bal_x := get_bal t_x (stor_tokens_held cstate) in
    (* balances update appropriately *)
    get_bal t_y (stor_tokens_held cstate') = (prev_bal_y - delta_y) /\
    get_bal t_x (stor_tokens_held cstate') = (prev_bal_x + delta_x) /\
    forall t_z,
        t_z <> t_x ->
        t_z <> t_y ->
        get_bal t_z (stor_tokens_held cstate') =
        get_bal t_z (stor_tokens_held cstate).
```

how the outstanding tokens update after a trade,

```
Definition trade_outstanding_update (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate' acts,
    (* the call to TRADE was successful *)
    receive contract chain ctx cstate (Some (trade (msg_payload))) = Ok(cstate', acts) ->
    (* in the new state *)
    (stor_outstanding_tokens cstate') = (stor_outstanding_tokens cstate).
```

and how the trade is priced.

```
Definition trade_pricing (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate' acts,
    (* the call to TRADE was successful *)
    receive contract chain ctx cstate (Some (trade (msg_payload))) = Ok(cstate', acts) ->
    (* balances for t_x change appropriately *)
    FMap.find (token_in_trade msg_payload) (stor_tokens_held cstate') =
    Some (get_bal (token_in_trade msg_payload) (stor_tokens_held cstate) + (qty_trade
    msg_payload)) /\
    (* balances for t_y change appropriately *)
    let t_x := token_in_trade msg_payload in
    let t_y := token_out_trade msg_payload in
    let delta_x := qty_trade msg_payload in
    let rate_in := (get_rate t_x (stor_rates cstate)) in
```

```
    let rate_out := (get_rate t_y (stor_rates cstate)) in
    let k := (stor_outstanding_tokens cstate) in
    let x := get_bal t_x (stor_tokens_held cstate) in
    (* in the new state *)
    FMap.find (token_out_trade msg_payload) (stor_tokens_held cstate') =
    Some (get_bal (token_out_trade msg_payload) (stor_tokens_held cstate)
        - (calc_delta_y rate_in rate_out delta_x k x)).
```

4.2.6 All Other Entrypoints

Finally, we include three propositions in the specification which require that any other entrypoints that the contract may have must not update rates, token balances, or the number of outstanding tokens.

```
(* Specification of all other entrypoints *)
Definition other_rates_unchanged (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate cstate' chain ctx o acts,
    (* the call to POOL was successful *)
    receive contract chain ctx cstate (other o) = Ok(cstate', acts) ->
    (* rates all stay the same *)
    forall t,
    FMap.find t (stor_rates cstate) = FMap.find t (stor_rates cstate').

Definition other_balances_unchanged (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate cstate' chain ctx o acts,
    (* the call to POOL was successful *)
    receive contract chain ctx cstate (other o) = Ok(cstate', acts) ->
    (* balances all stay the same *)
    forall t,
    FMap.find t (stor_tokens_held cstate) = FMap.find t (stor_tokens_held cstate').

Definition other_outstanding_unchanged (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate cstate' chain ctx o acts,
    (* the call to POOL was successful *)
    receive contract chain ctx cstate (other o) = Ok(cstate', acts) ->
    (* balances all stay the same *)
    (stor_outstanding_tokens cstate) = (stor_outstanding_tokens cstate').
```

4.2.7 The Structured Pools Specification Predicate

So far our specification is a list of propositions which characterize the contract types with typeclasses, and the entrypoint functions with propositions. We can summarize this with a predicate on smart contracts

which defines, precisely, the specification a structured pool. This is `is_structured_pool`.

This predicate axiomatizes the notion of a structured pool contract, which allows us to reason abstractly about the consequences of this contract specification, and to easily import or assume the existence of a structured pools contract in any other situation.

The specification predicate all the propositions we've mentioned thus far, as well as some others necessitated by the metaspecification. While not explicitly stated below, the contract types `Setup`, `Msg`, `State`, and `Error` are all implicitly required to conform to their respective typeclasses.

```
Definition is_structured_pool
    (C : Contract Setup Msg State Error) : Prop :=
    none_fails C /\
    msg_destruct C /\
    (* pool entrypoint specification *)
    pool_entrypoint_check C /\
    pool_emits_txns C /\
    pool_increases_tokens_held C /\
    pool_rates_unchanged C /\
    pool_outstanding C /\
    (* unpool entrypoint specification *)
    unpool_entrypoint_check C /\
    unpool_entrypoint_check_2 C /\
    unpool_emits_txns C /\
    unpool_decreases_tokens_held C /\
    unpool_rates_unchanged C /\
    unpool_outstanding C /\
    (* trade entrypoint specification *)
    trade_entrypoint_check C /\
    trade_entrypoint_check_2 C /\
    trade_pricing_formula C /\
    trade_update_rates C /\
    trade_update_rates_formula C /\
    trade_emits_transfers C /\
    trade_tokens_held_update C /\
    trade_outstanding_update C /\
    trade_pricing C /\
    trade_amounts_nonnegative C /\
    (* specification of all other entrypoints *)
    other_rates_unchanged C /\
    other_balances_unchanged C /\
    other_outstanding_unchanged C /\
    (* specification of calc_rx' and calc_delta_y *)
    update_rate_stays_positive /\
    rate_decrease /\
```

```
36      rates_balance /\
37      rates_balance_2 /\
38      trade_slippage /\
39      trade_slippage_2 /\
40      arbitrage_lt /\
41      arbitrage_gt /\
42      (* initialization specification *)
43      initialized_with_positive_rates C /\
44      initialized_with_zero_balance C /\
45      initialized_with_zero_outstanding C /\
46      initialized_with_init_rates C /\
47      initialized_with_pool_token C.
```

Listing 4.4: The `is_structured_pool` contract specification predicate

With this predicate in hand, we can reason abstractly about an arbitrary contract `C` which satisfies our specification by assuming a proof of the proposition `is_structured_pool C`.

4.3 The Metaspecification: Correctness of a Specification

4.3.1 The Definition of a Correct Specification

Let us reflect again on the economic attacks we encountered in Figure 2.1.4. The vulnerabilities were not because contracts were not correct with regards to their specifications, but rather that their specifications implied pathological economic behaviors. The Mango Markets attacker claimed that all his actions "were legal open market actions, using the protocol as designed, even if the development team did not fully anticipate all the consequences of setting parameters the way they are."[1]

A correct specification, then, is one that adequately captures the indended contract behavior. As we will explore throughout this thesis, the scope of intended behaviors is both vast and nuanced, and need not be limited to economic behaviors. However, in the remainder of this chapter, we will reason about the correctness of a specification by reasoning about the economic behaviors from the literature that a contract conforming to the specification will exhibit.

For what follows, a metaspecification is a list of properties that can be proved for any contract `C` satisfying the `is_structured_pool` specification. These properties are proved to justify that the specification does what it sets out to do. In our case, the metaspecification will be used to prove that the structured pools specification exhibits various desirable properties of a financial smart contract.

[1] https://twitter.com/av_eisen/status/1581326199682265088

4.3.2 Correctness of the Structured Pools Specification

Since the structured pool contract is designed to imitate AMMs, we draw on work by Angeris *et al.* [28, 31], Bartoletti *et al.* [47, 48], and Xu *et al.* [197] on AMMs and DeFi, from which we derive six properties indicative of desirable market behavior from game-theoretic and economic perspectives. These are as follows.

Demand sensitivity—A trade for a given token increases its price relative to other constituent tokens, so that higher relative demand corresponds to a higher relative price. Likewise, trading one token in for another decreases the first's relative price in the pool, corresponding to slackened demand. This enforces the classical notion of supply and demand and is important to the proper functioning of an AMM, as we see in [48, §4.2].

Nonpathological prices—As relative prices shift in response to trading activity, a price that starts out nonzero never goes to zero or to a negative value. This is to avoid pathological behavior of zero or negative prices, and is true for standard AMMs like Uniswap [31, §2]. However, like most AMMs, prices can still get arbitrarily close to zero, so a constituent token which loses its value due to external factors can still become arbitrarily devalued within the pool. This is an important property so that the formulae which price trades never divide by zero.

Swap rate consistency—For a token t_x in T and for any $\Delta_x > 0$, there is no sequence of trades, beginning and ending with t_x, such that $\Delta'_x > \Delta_x$, where Δ'_x is the output quantity of the sequence of trades. Swap rate consistency means that it is never profitable to trade in a loop, *e.g.* t_x to t_y, and back to t_x, which is important so that there are never any opportunities for arbitrage internal to the pool. This is similar to the assertion that trading cost be positive [31, §2], that trading from t_x to t_y, and back to t_x not be profitable in [28, §3, §4.1].

Zero-impact liquidity change—The quoted price of trades is unaffected by depositing or withdrawing liquidity [197, §3.3.1]. Typically, for AMMs such as Uniswap [16] or Curve [4] this is implemented by requiring that liquidity providers provide liquidity in pairs such that the quoted price of the AMM does not change by depositing or withdrawing liquidity. Liquidity provision works differently for structured pools, but depositing or withdrawing liquidity still does not impact quoted prices.

Arbitrage sensitivity—If an external, demand-sensitive market prices a constituent token differently from the structured pool, a sufficiently large arbitrage transaction will equalize the prices of the external market and the structured pool, or deplete the pool. In our case, this happens because prices adapt through trades due to demand sensitivity or the pool depletes in that particular token. This is generally considered to be an important property so that prices adjust in line with supply and demand, see [48, §4.3] and [31].

Pooled Consistency—The number of outstanding pool tokens is equal to the value, in pool tokens, of all constituent tokens held by the contract. Mathematically, the sum of all the constituent, pooled tokens,

multiplied by their value in terms of pooled tokens, always equals the total number of outstanding pool tokens. This means that the pool token is never under- or over-collateralized, and is similar to standard AMMs, where the LP token is always fully backed, representing a percentage of the liquidity pool, and is encoded in the literature as *preservation of net worth* [48, §3].

In what follows, we will show that a contract satisfying `is_structured_pool` also exhibits these six properties. For each property, we will first mathematically prove, by hand, that the result holds for a contract satisfying our specification. We will then formalize the results and discuss the formal proof.

4.3.3 Demand Sensitivity

Recall our informal definition of Demand Sensitivity.

> *A trade for a given token increases its price relative to other constituent tokens, so that higher relative demand corresponds to a higher relative price.*

We prove this first by hand as follows.

Property 1 (Demand Sensitivity)**.** Let t_x and t_y, $t_x \neq t_y$, be tokens in our family with nonzero pooled liquidity and exchange rates $r_x, r_y > 0$. In a trade t_x to t_y, as r_x is updated to r'_x, it decereases relative to r_z for all $t_z \neq t_x$, and r_y strictly increases relative to r_x.

Proof. First we prove that $r'_x < r_x$. We must prove:

$$r'_x = \frac{r_x x + r_y \Delta_y}{x + \Delta_x} < \frac{r_x x + r_x \Delta_x}{x + \Delta_x} = \frac{r_x (x + \Delta_x)}{x + \Delta_x} = r_x,$$

which holds if $r_y \Delta_y < r_x \Delta_x$. By (4.3) and (4.6):

$$\Delta_y < \frac{r_x}{r_y} \Delta_x = p_q \Delta_x,$$

so $r_y \Delta_y < r_x \Delta_x$ as desired. By the specification, r_z remains constant for all $t_z \neq t_x$ under `TRADE`, so as r_x is updated to r'_x it decreases relative to r_z. That r_y strictly increases relative to r_x is due to the fact that $r'_x < r_x$ and r_y stays constant. □

In the formalization below, the comments indicate which formalisms correspond to which parts of the statement of the theorem.

```
Theorem demand_sensitivity cstate :
    (* For all tokens t_x t_y, rates r_x r_y, and quantities x and y, where *)
    forall t_x r_x x t_y r_y y,
```

```
    (* t_x is a token with nonzero pooled liquidity and with rate r_x > 0, and *)
    FMap.find t_x (stor_tokens_held cstate) = Some x /\ x > 0 /\
    FMap.find t_x (stor_rates cstate) = Some r_x /\ r_x > 0 ->
    (* t_y is a token with nonzero pooled liquidity and with rate r_y > 0 *)
    FMap.find t_y (stor_tokens_held cstate) = Some y /\ y > 0 /\
    FMap.find t_y (stor_rates cstate) = Some r_y /\ r_y > 0 ->
    (* In a trade t_x to t_y ... *)
    forall chain ctx msg msg_payload acts cstate',
        (* i.e.: a successful call to the contract *)
        receive contract chain ctx cstate (Some msg) = Ok(cstate', acts) ->
        (* which is a trade *)
        msg = trade msg_payload ->
        (* from t_x to t_y *)
        msg_payload.(token_in_trade) = t_x ->
        msg_payload.(token_out_trade) = t_y ->
        (* with t_x <> t_y *)
        t_x <> t_y ->
    (* ... as r_x is updated to r_x': ... *)
    let r_x' := get_rate t_x (stor_rates cstate') in
    (* (1) r_x decreases relative to all rates r_z, for t_z <> t_x, and *)
    (forall t_z,
        t_z <> t_x ->
        let r_z  := get_rate t_z (stor_rates cstate) in
        let r_z' := get_rate t_z (stor_rates cstate') in
        rel_decr r_x r_z r_x' r_z') /\
    (* (2) r_y strictly increases relative to r_x *)
    let t_y := msg_payload.(token_out_trade) in
    let r_y  := get_rate t_y (stor_rates cstate) in
    let r_y' := get_rate t_y (stor_rates cstate') in
    rel_incr r_y r_x r_y' r_x'.
```

Listing 4.5: The formalization of Property 1 in ConCert.

The last lines of the theorem statement call on two technical definitions of what it means for natural numbers to increase or decrease relative to each other, which we have called `rel_incr` and `rel_decr`, respectively. We encode these as follows.

```
(* x decreases relative to z as x => x', z => z' : z - x < z' - x' *)
Definition rel_decr (x z x' z' : N) :=
    ((Z.of_N z) - (Z.of_N x) <= (Z.of_N z') - (Z.of_N x'))%Z.

(* y increases relative to x as y => y', x => x' : y - x < y' - x' *)
Definition rel_incr (y x y' x' : N) :=
    ((Z.of_N y) - (Z.of_N x) <= (Z.of_N y') - (Z.of_N x'))%Z.
```

These definitions are accompanied by two lemmas, which say that for all `x x' y : N`, `x' <= x` implies that `x` decreases relative to `y`, `rel_decr x y x' y`, and that `y` increases relative to `x`, `rel_incr y x y x'`.

```
Lemma rel_decr_lem : forall x x' z : N,
    x' <= x ->
    rel_decr x z x' z.

Lemma rel_incr_lem : forall x x' y : N,
    x' <= x ->
    rel_incr y x y x'.
```

Note that the strict inequality from Property 1 changed to a weak inequality in the formalization. This is, again, a symptom of implicitly assuming rational exchange rates. Because calculations are done with natural numbers in smart contracts, we cannot always guarantee, *e.g.* that for a given rate r_x, there is some rate r'_x which is both greater than zero and less than r_x.

Now onto the proof. Following the strategy of the proof of Property 1, the proof begins by proving that $r'_x \leq r_x$ and for all t_z, $t_z \neq t_x$, $r'_z = r_z$. This is expressed in the Coq proof as an assertion which we called `change_lemma`, which we introduce within the proof of `demand_sensitivity` with the `assert` tactic.

```
assert (r_x' <= r_x /\
    forall t,
    t <> t_x ->
    get_rate t (stor_rates cstate') = get_rate t (stor_rates cstate))
as change_lemma.
```

Once we have this result, we use `rel_decr_lem` to show that, since `r_x' <= r_x` and `r_z` is unchanged for all `t_z <> t_x`, `r_x` decreases relative to `r_z`. We use `rel_incr_lem` to show that, since `r_x' <= r_x` and `r_y` is unchanged, `r_x` increases relative to `r_z`. This gives our result.

The full Coq proof can be found in Appendix A.1.1.

4.3.4 Nonpathological Prices

Recall our informal definition of Nonpathological Prices.

> *As relative prices shift over time, a price that starts out nonzero never goes to zero or to a negative value.*

which we now prove about our specification.

Property 2 (Nonpathological prices)**.** For a token t_x in T, if there is a contract state such that $r_x > 0$, then $r_x > 0$ holds for all future states of the contract.

Proof. We only need to show that $r_x > 0$ implies $r'_x > 0$, since `TRADE` is the only entrypoint that updates exchange rates. Consider a contract state such that $r_x > 0$, and an incoming trade from t_x to some t_y of quantity $\Delta_x > 0$. Because Δ_y is calcluated such that

$$(\ell r_y + \Delta_x)(\ell r_x - \Delta_y) = k,$$

and since r_x, r_y, and Δ_x are all positive, we know that Δ_y is positive so long as k is not zero. If k is zero, the transaction fails as we specified for the `TRADE` entrypoint, so we know that $\Delta_y > 0$. Since $r_y\Delta_y < r_x\Delta_x$ and x cannot be negative we have that

$$0 < r_y\Delta_y < r_x\Delta_x < r_x(x + \Delta_x),$$

rendering the numerator of r'_x,

$$r_x x + r_y\Delta_y,$$

always positive. Since Δ_x is positive and x cannot be negative, the denominator of r'_x,

$$x + \Delta_x,$$

is also positive, which gives our result. Our result holds, then, so long as the `UPDATE_RATE` function return a positive number if the numerator and denominator of the quotient are positive, which we specified for the `TRADE` entrypoint. □

This is formalized as follows. Because by the specification, all rates initialize to be positive, we only need to say that given a reachable contract state `cstate`, if there is an entry in the `rates` map in storage, then it must be positive.

```
Theorem nonpathological_prices bstate caddr :
    (* reachable state with contract at caddr *)
    reachable bstate ->
    env_contracts bstate caddr = Some (contract : WeakContract) ->
    (* the statement *)
    exists (cstate : State),
    contract_state bstate caddr = Some cstate /\
    (* For a token t_x in T and rate r_x, *)
    forall t_x r_x,
    (* if r_x is the exchange rate of t_x, then r_x > 0 *)
    FMap.find t_x (stor_rates cstate) = Some r_x -> r_x > 0.
```

Listing 4.6: The formalization of Property 2 in ConCert

This is proved as a contract invariant, so we use the contract induction tactic, `contract_induction`, to prove the result. This divides the proof of the invariant into six cases. We have to first prove the invariant holds when the contract is deployed, and then we reestablish the invariant after:

1. addition of a block,
2. an outgoing action,
3. a nonrecursive call,
4. a recursive call, and
5. permutation of the action queue.

Each of five cases includes an inductive hypothesis, `IH` in the formal proof, which (as the inductive step) states that the invariant holds before each of these chain steps. In the final, and seventh, case of the proof, we prove any facts introduced at any of the previous six cases. This can be solved by the ConCert tactic `solve_facts` if no extra facts were introduced.

In the proof, found in Appendix A.1.2 each case of contract induction is labelled with comments.

4.3.5 Swap Rate Consistency

Recall our informal definition of Swap Rate Consistency:

> *For tokens t_x, t_y, and t_z, the trade of Δ_x from t_x to t_y, and then to t_z, must result in fewer tokens in t_z than a trade of Δ_x directly from t_x to token t_z.*

which we now prove about our specification.

Property 3 (Swap rate consistency)**.** Let t_x be a token in our family with nonzero pooled liquidity and $r_x > 0$. Then for any $\Delta_x > 0$ there is no sequence of trades, beginning and ending with t_x, such that $\Delta'_x > \Delta_x$, where Δ'_x is the output quantity of the sequence of trades.

Proof. Consider tokens t_x, t_y, and t_z with nonzero liquidity and with $r_x, r_y, r_z > 0$. First, we claim that the following inequality holds for all $x \geq 0$ and all trades from t_x to t_y:

$$r_y \Delta_y \leq r'_x \Delta_x. \tag{4.10}$$

Since

$$r'_x = \frac{r_x x + r_y \Delta_y}{x + \Delta_x}, \tag{4.9}$$

(4.10) simplifies to

$$r_y \Delta_y (x + \Delta_x) \leq \Delta_x (r_x x + r_y \Delta_y),$$

which in turn simplifies to

$$r_y \Delta_y x \leq r_x \Delta_x x.$$

Since we know that $r_y \Delta_y \leq r_x \Delta_x$ from (4.3), we can see that our inequality holds for all $x \geq 0$, as desired.

Now we consider sequences of trades beginning and ending with t_x. For a trade t_x to t_x, we have our result because

$$\Delta'_x < \frac{r_x}{r_x} \Delta_x = \Delta_x$$

by (4.3). Now consider a trading loop from t_x to t_y, and back to t_x, for $t_y \neq t_x$. We have our result if we can show

$$\frac{r_y}{r'_x} \Delta_y \leq \Delta_x$$

is satisfied, because $\frac{r_y}{r'_x} \Delta_y$ is an upper bound on the quantity that Δ_y can be traded for as $p_s < p_q$. This, of course, is given by (4.10) and the fact that $r'_x > 0$ from Property 2.

Finally, consider a trade from t_x to t_y, to t_z, and back to t_x. Similar to before we need to show that

$$\frac{r_z}{r'_x} \Delta_z \leq \Delta_x$$

is satisfied. But we have from (4.10) that

$$r_z \Delta_z \leq r'_y \Delta_y \leq r_y \Delta_y \leq r'_x \Delta_x, \tag{4.11}$$

as desired. This proof can be easily seen to apply to trading loops of arbitrary length, which proves our result. □

We have formalized this property as follows, where the comments indicate which part of the formal statement correspond to each part of the statement of Property 3.

```
Theorem swap_rate_consistency bstate cstate :
    (* Let t_x be a token with nonzero pooled liquidity and rate r_x > 0 *)
    forall t_x r_x x,
    FMap.find t_x (stor_rates cstate) = Some r_x /\ r_x > 0 ->
    FMap.find t_x (stor_tokens_held cstate) = Some x /\ x > 0 ->
    (* then for any delta_x > 0 and any sequence of trades, beginning and ending with t_x *)
    forall delta_x (trade_sequence : list trade_sequence_type) t_fst t_last,
    delta_x > 0 ->
    (* trade_sequence is a list of successive trades *)
    are_successive_trades trade_sequence ->
    (* with a first and last trade, t_fst and t_last respectively, *)
    (hd_error trade_sequence) = Some t_fst ->
```

```
    (hd_error (rev trade_sequence)) = Some t_last ->
    (* starting from our current bstate and cstate *)
    seq_chain t_fst = bstate ->
    seq_cstate t_fst = cstate ->
    (* the first trade is from t_x *)
    token_in_trade (seq_trade_data t_fst) = t_x ->
    qty_trade (seq_trade_data t_fst) = delta_x ->
    (* the last trade is to t_x *)
    token_out_trade (seq_trade_data t_last) = t_x ->
    FMap.find t_x (stor_rates cstate) = FMap.find t_x (stor_rates (seq_cstate t_last)) ->
    (* delta_x', the output of the last trade, is never larger than delta_x. *)
    let delta_x' := trade_to_delta_y t_last in
    delta_x' <= delta_x.
```

Listing 4.7: The formalization of Property 3 in ConCert

The theorem definition calls some definitions, which codify the notion of successive trades.

```
(* first a type to describe successive trades *)
Record trade_sequence_type := build_trade_sequence_type {
    seq_chain : ChainState ;
    seq_ctx : ContractCallContext ;
    seq_cstate : State ;
    seq_trade_data : trade_data ;
    seq_res_acts : list ActionBody ;
}.

(* a proposition that indicates a list of trades are successive, successful trades *)
Fixpoint are_successive_trades (trade_sequence : list trade_sequence_type) : Prop :=
    match trade_sequence with
    | [] => True
    | t1 :: l =>
        match l with
        | [] =>
            (* if the list has one element, it just has to succeed *)
            exists cstate' acts,
            receive contract
                (seq_chain t1)
                (seq_ctx t1)
                (seq_cstate t1)
                (Some (trade (seq_trade_data t1)))
            = Ok(cstate', acts)
        | t2 :: l' =>
            (* the trade t1 goes through, connecting t1 and t2 *)
            receive contract
                (seq_chain t1)
```

```
                (seq_ctx t1)
                (seq_cstate t1)
                (Some (trade (seq_trade_data t1)))
            = Ok(seq_cstate t2, seq_res_acts t2) /\
            (qty_trade (seq_trade_data t2) = trade_to_delta_y t1) /\
            (token_in_trade (seq_trade_data t2) = token_out_trade (seq_trade_data t1)) /\
            (are_successive_trades l)
        end
    end.
```

The formal proof mirrors the proof of Property 3. We first establish Equation (4.10) as a lemma as follows.

```
Lemma swap_rate_lemma : forall trade_sequence,
    (* if this is a list of successive trades *)
    are_successive_trades trade_sequence ->
    (* then  *)
    geq_list (map trade_to_ry_delta_y trade_sequence).
```

We then show that this is sufficient to prove the result with another lemma.

```
Lemma geq_list_is_sufficient : forall trade_sequence t_x t_fst t_last cstate r_x,
    (* more assumptions to be able to call swap_rate_lemma *)
    (hd_error trade_sequence) = Some t_fst ->
    (hd_error (rev trade_sequence)) = Some t_last ->
    token_in_trade (seq_trade_data t_fst) = t_x ->
    token_out_trade (seq_trade_data t_last) = t_x ->
    seq_cstate t_fst = cstate ->
    FMap.find t_x (stor_rates cstate) = Some r_x /\ r_x > 0 ->
    FMap.find t_x (stor_rates cstate) = FMap.find t_x (stor_rates (seq_cstate t_last)) ->
    (* the statement *)
    geq_list (map trade_to_ry_delta_y trade_sequence) ->
    let delta_x := qty_trade (seq_trade_data t_fst) in
    let delta_x' := trade_to_delta_y t_last in
    delta_x' <= delta_x.
```

The lemma `geq_list_is_sufficient` shows the output quantity of the list of trades, `delta_x'`, is less than or equal to the original input quantity of the list of trades, `delta_x`. For the sake of brevity, those quantities are calculated by the functions `qty_trade` and `trade_to_delta_y`.

The full Coq proof can be found in Appendix A.1.3.

4.3.6 Zero-Impact Liquidity Change

Recall our informal definition of Zero-Impact Liquidity Change from §4.3,

> *The quoted price of trades is unaffected by depositing or withdrawing liquidity.*

which we now prove about our specification.

Property 4 (Zero-Impact Liquidity Change)**.** The quoted price of trades is unaffected by calling `POOL` and `UNPOOL`.

Proof. We have this result because the quoted price depends only on the pooling exchange rates, as we saw in (4.4), and as per the specification, only the `TRADE` entrypoint alters pooling exchange rates. □

We have formalized this property as follows.

```
Theorem zero_impact_liquidity_change :
    (* Consider the quoted price of a trade t_x to t_y at cstate, *)
    forall cstate t_x t_y r_x r_y,
    FMap.find t_x (stor_rates cstate) = Some r_x ->
    FMap.find t_y (stor_rates cstate) = Some r_y ->
    let quoted_price := r_x / r_y in
    (* and a successful POOL or UNPOOL action. *)
    forall chain ctx msg payload_pool payload_unpool acts cstate' r_x' r_y',
        receive contract chain ctx cstate (Some msg) = Ok(cstate', acts) ->
        msg = pool payload_pool \/
        msg = unpool payload_unpool ->
    (* Then take the (new) quoted price of a trade t_x to t_y at cstate'. *)
    FMap.find t_x (stor_rates cstate') = Some r_x' ->
    FMap.find t_y (stor_rates cstate') = Some r_y' ->
    let quoted_price' := r_x' / r_y' in
    (* The quoted price is unchanged. *)
    quoted_price = quoted_price'.
```

Listing 4.8: The formalization of Property 4 in ConCert

Just like the proof of Property 4, the formal proof is relatively straightforward, and draws on the specification, and can be found in Appendix A.1.4.

4.3.7 Arbitrage Sensitivity

Recall our informal definition of Arbitrage Sensitivity from §4.3,

If an external, demand-sensitive market prices a constituent token differently from the structured pool, a sufficiently large arbitrage transaction will equalize the prices of the external market and the structured pool, or deplete the pool.

which we now prove about our specification.

Property 5 (Arbitrage sensitivity)**.** Let t_x be a token in our family with nonzero pooled liquidity and $r_x > 0$. If an external, demand-sensitive market prices t_x differently from the structured pool, then assuming sufficient liquidity, with a sufficiently large transaction either the price of t_x in the structured pool converges with the external market, or the trade depletes the pool of t_x.

Proof. Suppose the structured pool prices a constituent token t_x higher than an external market. Then an arbitrageur can buy t_x elsewhere and sell them into the structured pool. Doing so devalues t_x relative to the other tokens, as we have shown. Recall that $0 < r'_x < r_x$, so to prove our result we just need to show that 0 is the greatest lower bound of r'_x. Note that by definition, $\Delta_y = \Delta^s_y$, so substituting (4.8)

$$\Delta^s_y = \ell r_x - \frac{k}{\ell r_y + \Delta_x},$$

$$r'_x = \frac{r_x x + r_y \Delta_y}{x + \Delta_x} = \frac{r_x x + \ell r_x r_y - \frac{r_y k}{\ell r_y + \Delta_x}}{x + \Delta_x}.$$

Then

$$r'_x < \frac{r_x x + \ell r_x r_y}{x + \Delta_x} \tag{4.12}$$

and since x, r_x, r_y, and ℓ are constants for a trade, for any r, $0 < r < r_x$, by choosing a sufficiently large Δ_x we can make $r'_x < r$. Thus assuming sufficient external liquidity, we have our result.

Now suppose the structured pool prices a constituent token t_x lower than an external market. Then an arbitrageur can buy t_x from the structured pool and sell them elsewhere. Doing so does not change r_x, as per the specification. However, the external market is demand sensitive, so the price of t_x will decrease on that market. Then we know that after a trade of $\Delta_x = x$, either the external market now prices t_x lower than the structured pool contract, meaning there was some

$$\Delta'_x < \Delta_x$$

which gives our result, or the trade depletes the pool of t_x, giving our result. □

We have formalized this property as follows.

```
1 Theorem arbitrage_sensitivity :
2     forall cstate t_x r_x x,
3     (* t_x is a token with nonzero pooled liquidity *)
```

```
    FMap.find t_x (stor_rates cstate) = Some r_x /\ r_x > 0 /\
    FMap.find t_x (stor_tokens_held cstate) = Some x /\ x > 0 ->
    (* we consider some external price *)
    forall external_price,
    0 < external_price ->
    (* and a trade of trade_qty succeeds *)
    forall chain ctx msg msg_payload cstate' acts,
    receive contract chain ctx cstate msg = Ok(cstate', acts) ->
    msg = Some(trade msg_payload) ->
    t_x = (token_in_trade msg_payload) ->
    (* the arbitrage opportunity is resolved *)
    let r_x' := get_rate t_x (stor_rates cstate') in
    (* first the case that the external price was lower *)
    (external_price < r_x ->
        exists trade_qty,
        msg_payload.(qty_trade) = trade_qty ->
        external_price >= r_x') /\
    (* second the case that the external price is higher *)
    (external_price > r_x ->
    exists trade_qty,
        msg_payload.(qty_trade) = trade_qty ->
        external_price <= r_x' \/
    let t_y := token_out_trade msg_payload in
    let r_y := get_rate t_y (stor_rates cstate) in
    let x := get_bal t_x (stor_tokens_held cstate) in
    let y := get_bal t_y (stor_tokens_held cstate) in
    let balances := (stor_tokens_held cstate) in
    let k := (stor_outstanding_tokens cstate) in
    get_bal t_y balances <= calc_delta_y r_x r_y trade_qty k x).
```

Listing 4.9: The formalization of Property 5 in ConCert

One weakness of our approach is that we are, as of yet, unable to explicitly model an arbitrary, external, demand-sensitive market. Instead, we have proved our result for some external price, expressed in terms of the exchange rate `r_x`. Of course, the external price will likely not be in terms of pool tokens, but rather in terms of some other token such as XTZ or USDT. Even in these cases, there is a corresponding rate `external_price` which can be compared to `r_x`, so our model assumes some external rate and proves arbitrage sensitivity in relation to it. In Chapter 6, we will look more deeply at rigorously verifying systems of contracts.

The proof of this result relies on two critical properties of the specification of `calc_rx'`.

```
(* rates have no positive lower bound *)
Definition arbitrage_lt :=
```

```
    forall rate_x rate_y ext k x,
    0 < ext ->
    ext < rate_x ->
    exists delta_x,
    calc_rx' rate_x rate_y delta_x k x <= ext.

(* calc_delta_y has no positive upper bound *)
Definition arbitrage_gt :=
    forall rate_x rate_y ext_goal k x,
    rate_x > 0 /\
    rate_y > 0 /\
    x > 0 /\
    k > 0 ->
    exists delta_x,
    ext_goal <= calc_delta_y rate_x rate_y delta_x k x.
```

The full Coq proof, which follows the proof of Property 5, can be found in Appendix A.1.5.

4.3.8 Pooled Consistency

Finally, recall our informal definition of Pooled Consistency from §4.3,

> *The sum of all the constituent, pooled tokens, multiplied by their value in terms of pooled tokens, always equaling the total number of outstanding pool tokens.*

which we now prove about our specification.

Property 6 (Pooled Consistency)**.** The following equation always holds:

$$\sum_{t_x} r_x x = k \tag{4.13}$$

Proof. As a base case, by the specification, at the time of contract deployment $k = 0$ and we have no pooled liquidity, so (4.13) holds trivially because $x = 0$ for all t_x. For our inductive step, consider a contract state for which (4.13) holds. If we call POOL, (4.13) holds by definition because for a deposit of d_x of t_x, we mint $r_x d_x$ pool tokens. The same is true if we call UNPOOL. Finally, if we call TRADE from tokens t_x to t_y, then there is an excess number of tokens in t_x, violating (4.13). This excess is quantified in (4.9) and remedied by adjusting r_x to r'_x as we saw before. □

We have formalized this property as follows.

```
Theorem pooled_consistency bstate caddr :
    reachable bstate ->
    env_contracts bstate caddr = Some (contract : WeakContract) ->
    exists (cstate : State),
    contract_state bstate caddr = Some cstate /\
    (* The sum of all the constituent, pooled tokens, multiplied by their value in terms
    of pooled tokens, always equals the total number of outstanding pool tokens. *)
    suml (tokens_to_values (stor_rates cstate) (stor_tokens_held cstate)) =
        (stor_outstanding_tokens cstate).
```

Listing 4.10: The formalization of Property 6 in ConCert

which relies on an auxiliary function that captures the sum of Equation 4.13 in Proposition 6.

```
Definition tokens_to_values (rates : FMap token exchange_rate) (tokens_held : FMap token N) :=
    List.map
        (fun k =>
            let rate := get_rate k rates in
            let qty_held := get_bal k tokens_held in
            rate * qty_held)
        (FMap.keys rates).
```

The difficulty and length of this proof is largely algebraic, requiring us to tediously manipulate algebraic expressions which are easy to see when proving by hand, but not obvious when proving about our code. Furthermore, while the definition of r'_x in relation to r_x can be easily seen to preserve Equation 4.13, due to rounding error it is not obvious by definition that the same holds. The proof, then, of this proposition is lengthy relative to our other proofs.

As with the formalization of Property 2, this proof is done by contract induction. The full Coq proof can be found in Appendix A.1.6.

4.4 Reflections on the Metaspecification

The six properties of the metaspecification are intended to describe desirable contract behavior from an economic perspective. As mentioned previously, these were derived from work by Angeris *et al.* [28, 31], Bartoletti *et al.* [47, 48], and Xu *et al.* [197] on AMMs and DeFi. All this work either empirically studies contract behavior from an economic lens or creates a theory to describe it from first principles. The goal of each, however, seems to be roughly the same, which is to distil key economic properties of well-behaved AMMs and other DeFi contracts. To our knowledge, our work is the first example of using such properties in a formal setting to reason about a contract specification.

There are other properties of a specification that we might verify to justify its strength and robustness, including properties which we can find by more conventional software testing techniques. For example, Phipathananunth's recent work proposes using mutation testing to evaluate the strength of a contract specification [161]. The idea is to test the robustness of a contract specification using tools similar to how one might test the robustness of a contract. This uses Certora, a formal verification tool for Solidity contracts [3].

Djed, a formally verified stablecoin [**?**], takes a similar approach. Before deploying the stablecoin contracts, they formally verified several properties of the specification, in Isabelle and using SMT solvers, to justify the robustness of the contract specification. The test cases were generated using practical tools, such as mutation and unit testing, to identify potentially pathological behavior.

In all likelihood, deriving properties of the metaspecification both by reasoning from first principles and using more conventional testing techniques is probably the most robust.

Future work could include developing a more well-rounded approach to metaspecification formation, including incorporating more conventional software testing tools to develop the metaspecification, as well as formally embedding the abstract theories of Bartoletti *et al.* [47, 48] into ConCert to have a fully formal theory of AMMs and DeFi.

4.5 Conclusion

This chapter presents a formal, systematic way of reasoning about economic properties and vulnerabilities of smart contracts, something largely unexplored in formal verification of smart contracts despite its relevance to smart contract security. We do this by formally separating the notion of a specification and a metaspecification. The metaspecification is to a specification what a specification is to software: like a specification does for software, a metaspecification abstracts and describes the goals of the specification and can be used to prove its correctness.

Moving forward, we will expand on the notion of a metaspecification to reason about other difficult-to-specify, desirable properties of a contract specification.

Chapter 5

Morphisms of Smart Contracts

The goal of this chapter is to formally reason about properties of smart contracts by studying how contracts can relate to each other in a formal, structural sense. There are good reasons to do this. In §2.1.2, we argued that there are vulnerabilities and costly exploits in contract upgrades and forks which can be reasoned about if we are able to structurally compare a contract upgrade or fork to its previous version.

To this end we introduce a theoretical tool called a *contract morphism*, which is a formal mechanism to structurally compare smart contracts in ConCert. We show how it can be used in proof and specification, targetting upgradeability properties in particular. We show that contract morphisms help us reason about difficult-to-define aspects of a contract specification, this time targetting upgradeability properties rather than the economic properties of Chapter 4.

This chapter is organized as follows: In §5.1, we motivate contract morphisms with a discussion on formal specification and verification of contract upgrades. In §5.2, we introduce a morphism of contracts, which is a formal, structural relationship between smart contracts, as a tool to specify contract upgrades. In §5.3, we discuss strategies in proof and specification using morphisms of contracts. In §5.4 we reason about contract upgrades, drawing again on the example of structured pools from the previous chapter.

5.1 Contract Upgrades

Like the economic properties of a contract specification, contract upgradeability involves complex contract behavior which can be difficult to specify correctly. Generally speaking, there are no in-built methods for upgrading a smart contract once it has been deployed. Indeed, contract immutability is part of their exceptional vulnerability, which motivates using formal methods in the first place. If one wishes to encode the

ability to upgrade into a smart contract, one has to write that into the original contract, before deployment. As we saw in §2.1.2 and Figure 2.1.4, this is hard to do well and can lead to costly bugs.

The alternative to encoding upgradeability into a smart contract is to use some sort of hard fork. This involves deploying a new contract and convincing users to migrate to the new one, for example with each of the Uniswap upgrades [24, 107]. Especially for financial smart contracts such as AMMs which rely on liquidity providers, this transition can be expensive, difficult, and best avoided if possible.

Thus we are faced with the question of how to safely encode upgradeability into a smart contract. There are resources in the form of best practices [17, 18] and established upgrade frameworks [10, 13], one of these being the EIP-2535 Diamond upgrade framework [10], which is a specification of a system of smart contracts which use a proxy framework to enable upgradeability. However, being able to upgrade a contract is not enough to prevent vulnerabilities—if one wishes to achieve the assurance guarantees of formal methods on upgrades, one has to also somehow be able to formally specify and verify an upgradeable contract as it iterates through upgrades [37].

5.1.1 Specifying Upgradeability: The Diamond Framework

Consider the EIP-2535 Diamond upgrade framework [10], which is a specification of a generic and flexible upgrade protocol. The specification describes contract storage and entrypoints, defining a system of contracts that all interact with each other. Reading the specification, one can convince one's self that it does in fact specify an upgradeable contract. However, there is no property of the specification which precisely characterizes what it means to be upgradeable, and *how* upgradeable or mutable this defined structure is.

By the phrase *how upgradeable*, we mean a formal understanding of what functionality can (and is meant to) be updated through some predefined process, and what is unchangeable through upgrades. The Diamond standard, for example, allows one to add to the storage, but one cannot change the type of the storage previously defined. For example, a ledger in storage `balances : FMap Address nat` cannot be upgraded to a ledger of type `Fmap (Address * nat) nat`, which we might wish to do if one address were allowed multiple separate entries in the ledger. If we initialize the contract with a ledger `balances : FMap Address nat`, we know that the contract storage will always `balances` in its storage, with that same map type, no matter the upgrade. We also refer to this as the contract's *bounds of upgradeability*.

Rather than a precise notion of upgradeability, the intuition behind what it means to be upgradeable, and the bounds of upgradeability, are communicated through pictures, diagrams, the motivation section of the specification, and the analogy of a diamond in naming conventions for different parts of the specification. Each of these are rhetorical tools to communicate what it means to be "upgradeable" but they are not

mathematically or technically rigorous. In particular, if we were to formally verify this standard, we would have no independent, mathematical notion to verify the claim that this is, indeed, upgradeable in some precise sense.

How do we mathematically capture the notion of upgradeability? The answer to this question is not obvious. By definition, upgradeability denotes some built-in mutability. As contracts are immutable once deployed, our contention is that understanding contract upgradeability requires being able to decompose a smart contract into its immutable and mutable parts. The immutable denotes the upgradeability framework, or *skeleton*, and the mutable denotes that contract functionality which can be upgraded, or the *mutable functionality*.

5.1.2 Evolving Specifications Through Upgrades

There is another important aspect of contract upgradeability, which is that when executing a contract upgrade, we alter the contract specification as we replace it with a new one. We argue that we can avoid vulnerabilities if we are able to reason about the new specification in relation to the old.

Consider a contract upgrade from the perspective of a formal specification. Generally speaking, an upgrade happens with a goal in mind, whether it be patching a bug, adding functionality, or improving features. In each of these cases, the specification evolves to various degrees and in various ways, whether it is just to add a property or condition that fixes a bug, or to alter specific propositions from the previous specification. It is rare that we would wish to upgrade a contract into something totally unrelated to the previous version.

Instead, the new specification likely relates to the old: the new should eliminate a vulnerability of the old, or the new should be backwards compatible while adding functionality to the old, or the new makes improvements on the old, for example to be more gas-efficient.

The actual intention of the upgrade, and therefore its specification, might be best written in relation to the old contract. Informally, if the upgrade is to patch a bug, we might specify that the new contract behave identically to the old, except that it patches the buggy functionality. If the upgrade adds functionality, we might specify that new functionality as normal, and then specify that all the functionality of the old contract hold for the new. If it does not hold, we might wish to specify exactly how it changes in relation to the old. Finally if we are optimizing, we might specify that the new contract behave exactly like the old, except that it improve in some metric like gas efficiency or precision.

If one wishes to avoid specifying an upgrade in terms of the old contract, the other option is to simply make alterations to the old contract specification into the new. This is hardly an exact science; indeed, doing so led to the costly exploits we saw in §2.1.2.

Thus to reason rigorously about contract upgrades we need to introduce formal tools to be able to compare contract specifications. For this and the decomposition discussed in the previous section, we will introduce *contract morphisms*, an extension of ConCert which enables formal, structural comparisons between smart contracts and their specifications.

5.1.3 Related Work

Little work has gone into understanding contract upgradeability from a formal perspective. There are two notable exceptions, both of which have similarities to what we propose here.

The first is work by Antonino *et al.* [37], which seeks to address vulnerabilities in contract upgrades by proposing a novel *systematic deployment framework* that requires contracts to be formally verified before they are deployed or upgraded. The framework relies on a *trusted deployer*, which is an off-chain service that vets contract creations and updates. Under this framework, at deployment of an upgradeable contract one can specify the bounds of upgradeability by requiring from the trusted deployer that certain invariants hold through all upgrades.

The second is work by Dickerson *et al.* [84], which proposes a paradigm shift for blockchains and smart contracts so that smart contracts can carry with them proofs of correctness. These are called *proof-carrying smart contracts*. These contracts can be upgradeable, so long as the upgraded contract carries a proof that it conforms to a specification by the original deployed contract. This work relates to previous work outside the context of blockchains on dynamic software updating [110] and proof-carrying code [144].

Both of these seek to limit the bounds of upgradeability by requiring that upgrades conform to some kind of a specification, where Antonino *et al.* rely on a trusted deployer to verify that an upgrade meets a specification before deploying the upgrade, and Dickerson *et al.* require a new paradigm of proof-carrying smart contracts so that the blockchain itself can verify a proof that a contract upgrade conforms to certain specified standards. In both cases, there is a desire to be able to specify certain invariants at the time of deployment that cannot be altered through upgrades.

Our approach is distinct. It requires no trusted third party, nor a fundamental upgrade to smart contracts to carry proofs. Rather, it mathematically characterizes the bounds upgradeability of a smart contract through a decomposition into its mutable and immutable parts, so as to rigorously encode these kinds of upgradeability limitations implicitly into the contract itself. If one wishes to impose invariants that cannot be changed through upgrades, one can do so by encoding those into the immutable part of the contract and formally proving the invariants hold.

5.2 Morphisms of Smart Contracts

The goal of morphisms of contracts is to be able to reason about a contract `C` and its specification S in relation to another contract `C'` with specification S', a notion we make precise in §5.3. As we will see, contract morphisms can be used both to prove and specify properties of one contract in terms of another.

Recall that the `Contract`, parameterized by `Setup`, `Msg`, `State`, and `Error` types, is a record type with two constructors: the `init` function, which dictates how a contract is initialized, and the `receive` function, which dictates the semantics of calling a contract entrypoint. In the context of a given state of the blockchain, the `init` function takes something of type `Setup` and, if successful, deploys the contract with an initial storage of type `State`.

```
1 init : Chain -> ContractCallContext -> Setup -> result State Error;
```

The `receive` function then takes the current state of the contract and something of type `Msg`, and, if successful, produces an updated storage of type `State` and a list of emitted transactions.

```
1 receive : Chain -> ContractCallContext -> State -> option Msg ->
2                 result (State * list ActionBody) Error;
```

Now consider contracts `C` and `C'` with init and receive functions `init`, `init'` and `receive`, `receive'`. We can conceive of a function `f : C -> C'` type theoretically as a transformation of init and receive functions,

`f_init : init -> init'` and `f_receive : receive -> receive'`.

Mathematically, such a transformation from $f : A \to B$ to $g : C \to D$ is a *commutative square*, also called a *natural transformation*, consisting of functions $h_1 : A \to C$ and $h_2 : B \to D$ such that in the diagram below, $g \circ h_1 = h_2 \circ f$ by functional extensionality.

$$\begin{array}{ccc} A & \xrightarrow{h_1} & C \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle g} \\ B & \xrightarrow{h_2} & D \end{array} \tag{5.1}$$

To encode these notions into ConCert, we define the types `InitCM C C'`, which is the type of a function from `init` to `init'`, `RecvCM C C'`, the type of a function from `receive` to `receive'`, and `ContractMorphism C C'`, the type of a function from `C` to `C'`. We may informally write `f : C -> C'` as shortand for writing `f : ContractMorphism C C'`.

For `InitCM`, `init_inputs` plays the role of h_1 in Diagram 5.1, `init_outputs` plays the role of h_2, and `init_commutes` is a proof that the square commutes. Analogously for `RecvCM`, `recv_inputs` plays the role

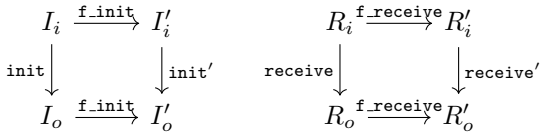


Figure 5.1: A visual representation of the `InitCM` and `RecvCM` components of a contract morphism, where I_i and I_o (resp. I'_i and I'_o) are the input and output types of the `(init C)` (resp. `(init C')`), and R_i and R_o (resp. R'_i and R'_o) are the input and output types of `(receive C)` (resp. `(receive C')`).

of h_1 in Diagram 5.1, `recv_outputs` plays the role of h_2, and `recv_commutes` is a proof that the square commutes. The `ContractMorphism` type is then a record type with an init component of type `InitCM` and receive component of type `RecvCM`.

```
(* The init component *)
Record InitCM (C : Contract Setup Msg State Error) (C' : Contract Setup' Msg' State' Error')
    := build_init_cm {
        (* transform inputs/outputs of the init function *)
        init_inputs : Chain * ContractCallContext * Setup ->
            Chain * ContractCallContext * Setup' ;
        init_outputs : result State Error -> result State' Error' ;
        (* proof of commutativity *)
        init_commutes :
            forall (c, ctx, s) : Chain * ContractCallContext * Setup,
                C'.(init) (init_inputs (c, ctx, s)) =
                init_outputs (C.(init) c ctx s) ;
    }.

(* The receive component *)
Record RecvCM (C : Contract Setup Msg State Error) (C' : Contract Setup' Msg' State' Error')
    := build_recv_cm {
        (* transform inputs/outputs of the receive function *)
        recv_inputs : Chain * ContractCallContext * State * option Msg ->
            Chain * ContractCallContext * State' * option Msg' ;
        recv_outputs : result (State * list ActionBody) Error ->
            result (State' * list ActionBody) Error' ;
        (* proof of commutativity *)
        recv_commutes :
            forall (c, ctx, st, op_msg : Chain * ContractCallContext * State * option Msg),
                C'.(receive) (recv_inputs (c, ctx, st, op_msg)) =
                recv_outputs (C.(receive) c ctx st op_msg) ;
    }.

(* The type of Contract Morphisms *)
Record ContractMorphism
```

```
32      (C  : Contract Setup  Msg  State  Error)
33      (C' : Contract Setup' Msg' State' Error') :=
34      build_cm {
35          cm_init : InitCM C C' ;
36          cm_recv : RecvCM C C' ;
37      }.
```

Listing 5.1: The definition of contract morphisms in ConCert

5.2.1 Composition of Morphisms

Contract morphisms can be composed. We define composition of contract morphisms via a function `composition_cm`, which takes morphisms `f : ContractMorphism C C'` and `g : ContractMorphism C' C''` and returns `composition_cm g f : ContractMorphism C C''`. As we see in the definition below, we compose contract morphisms by composing their component functions.

```
1  Definition composition_cm
2      (g : ContractMorphism C' C'')
3      (f : ContractMorphism C  C') : ContractMorphism C C'' :=
4      let compose_init := {|
5          init_inputs  := compose (init_inputs  C' C'' (init_cm g)) (init_inputs  C C' (init_cm
       f)) ;
6          init_outputs := compose (init_outputs C' C'' (init_cm g)) (init_outputs C C' (init_cm
       f)) ;
7          (* proof of commutativity *)
8          init_commutes := compose_commutes_init g f ;
9      |} in
10     let compose_recv := {|
11         recv_inputs  := compose (recv_inputs  C' C'' (recv_cm g)) (recv_inputs  C C' (recv_cm
       f)) ;
12         recv_outputs := compose (recv_outputs C' C'' (recv_cm g)) (recv_outputs C C' (recv_cm
       f)) ;
13         (* proof of commutativity *)
14         recv_commutes := compose_commutes_recv g f ;
15     |} in
16     {|
17         cm_init := compose_init ;
18         cm_recv := compose_recv ;
19     |}.
```

Listing 5.2: Composition of morphisms of contracts

The function `composition_cm` relies on two lemmas,

`compose_commutes_init g f` and `compose_commutes_recv g f`,

which prove the commutativity results of the `InitCM C C''` and `RecvCM C C''` components of the contract morphism `composition_cm g f`, using the corresponding commutativity result of `f` and `g`, respectively.

These lemmas effectively show that commuting diagrams compose. That is, if we have the diagram below such that each of the left and right squares commute, then the outer rectangle also commutes.

$$\begin{array}{ccccc} A & \xrightarrow{i_1} & C & \xrightarrow{j_1} & E \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle h} \\ B & \xrightarrow{i_2} & D & \xrightarrow{j_2} & F \end{array}$$

```
Lemma compose_commutes_init (g : ContractMorphism C' C'') (f : ContractMorphism C  C') :
    init_morphs_commute
        C.(init)
        C''.(init)
        (compose (init_inputs  C' C'' (init_cm g)) (init_inputs  C C' (init_cm f)))
        (compose (init_outputs C' C'' (init_cm g)) (init_outputs C C' (init_cm f))).
Proof.
    unfold init_morphs_commute. intro. simpl.
    rewrite (init_commutes C' C'' (init_cm g)).
    rewrite (init_commutes C C' (init_cm f)).
    reflexivity.
Qed.

Lemma compose_commutes_recv (g : ContractMorphism C' C'') (f : ContractMorphism C  C') :
    recv_morphs_commute
        C.(receive)
        C''.(receive)
        (compose (recv_inputs  C' C'' (recv_cm g)) (recv_inputs  C C' (recv_cm f)))
        (compose (recv_outputs C' C'' (recv_cm g)) (recv_outputs C C' (recv_cm f))).
Proof.
    unfold recv_morphs_commute. intro. simpl.
    rewrite (recv_commutes C' C'' (recv_cm g)).
    rewrite (recv_commutes C C' (recv_cm f)).
    reflexivity.
Qed.
```

Listing 5.3: Two lemmas to define morphism composition.

5.2.2 Equality of Morphisms

Morphisms `f g : ContractMorphism C C'` are equal if their respective `InitCM` and `RecvCM` components are equal, which we prove this with the `is_eq_cm` lemma from Listing 5.4. To prove that the `InitCM` and `RecvCM` components are equal, we use function extensionality, as shown in the `is_eq_cm_init` and `is_eq_cm_recv` lemmas also in Listing 5.4. The lemmas derive the associated commutativity results for us using proof irrelevance.

```
Lemma is_eq_cm :
    forall (f g : ContractMorphism C C'),
    init_cm f = init_cm g ->
    recv_cm f = recv_cm g ->
    f = g.
Proof.
    intros f g init_eq recv_eq.
    destruct f. destruct g. f_equal.
    - unfold init_cm in init_eq. unfold cm_init in init_eq.
      exact init_eq.
    - unfold recv_cm in recv_eq. unfold cm_recv in recv_eq.
      exact recv_eq.
Qed.

Lemma is_eq_cm_init :
    forall (f g : ContractMorphism C C'),
    (init_inputs C C' (init_cm f)) = (init_inputs C C' (init_cm g)) ->
    (init_outputs C C' (init_cm f)) = (init_outputs C C' (init_cm g)) ->
    init_cm f = init_cm g.
Proof.
    (* proof omitted except for the last line *)
    apply proof_irrelevance.
Qed.

Lemma is_eq_cm_recv :
    forall (f g : ContractMorphism C C'),
    (recv_inputs C C' (recv_cm f)) = (recv_inputs C C' (recv_cm g)) ->
    (recv_outputs C C' (recv_cm f)) = (recv_outputs C C' (recv_cm g)) ->
    recv_cm f = recv_cm g.
Proof.
    (* proof omitted except for the last line *)
    apply proof_irrelevance.
Qed.
```

Listing 5.4: Equality of contract morphisms

5.2.3 Composition is Associative

With the notions of morphism composition and equality defined, we prove that composition is associative.

```
Proposition composition_assoc_cm :
    forall
    (f : ContractMorphism C C')
    (g : ContractMorphism C' C'')
    (h : ContractMorphism C'' C'''),
    composition_cm h (composition_cm g f) =
    composition_cm (composition_cm h g) f.
Proof.
    intros.
    unfold composition_cm. simpl. apply is_eq_cm.
    -   apply
        (is_eq_cm_init
            (composition_cm h (composition_cm g f))
            (composition_cm (composition_cm h g) f)); auto.
    -   apply
        (is_eq_cm_recv
            (composition_cm h (composition_cm g f))
            (composition_cm (composition_cm h g) f)); auto.
Qed.
```

Listing 5.5: Composition of contract morphisms is associative

5.2.4 Examples of Contract Morphisms

Example 5.2.1 (Identity Morphism)**.** Our first example is the identity morphism `id_cm`, which is defined for any contract `C` and inhabits `ContractMorphism C C`. We give the `InitCM` and `RecvCM` components, and define the morphism. The definition relies on a pair of lemmas `init_commutes_id` and `recv_commutes_id` which give our commutativity results:

```
Lemma init_commutes_id (C : Contract Setup Msg State Error) :
    init_morphs_commute
        C.(init) C.(init)
        (id (Chain * ContractCallContext * Setup))
        (id (result State Error)).
(* proof omitted *)

Lemma recv_commutes_id (C : Contract Setup Msg State Error) :
    recv_morphs_commute
        C.(receive) C.(receive)
        (id (Chain * ContractCallContext * State * option Msg))
```

```
12          (id (result (State * list ActionBody) Error)).
13 (* proof omitted *)
```

With these lemmas we define the identity contract morphism for any contract `C`.

```
1  (* the init component *)
2  Definition id_cm_init (C : Contract Setup Msg State Error) :
3      InitCM C C := {|
4          init_inputs  := id (Chain * ContractCallContext * Setup) ;
5          init_outputs := id (result State Error) ;
6          (* proof of commutativity *)
7          init_commutes := init_commutes_id C ; |}.
8
9  (* the receive component *)
10 Definition id_cm_recv (C : Contract Setup Msg State Error) : RecvCM C C := {|
11         recv_inputs := id (Chain * ContractCallContext * State * option Msg) ;
12         recv_outputs := id (result (State * list ActionBody) Error) ;
13         (* proof of commutativity *)
14         recv_commutes := recv_commutes_id C ; |}.
15
16 (* the identity morphism *)
17 Definition id_cm (C : Contract Setup Msg State Error) : ContractMorphism C C := {|
18         cm_init := id_cm_init C ;
19         cm_recv := id_cm_recv C ; |}.
```

Listing 5.6: The identity morphism of contracts

An important property of an identity morphism is that it be trivial under composition. That is, we should have the results that for all `f : ContractMorphism C C'`,

```
composition_cm f (id_cm C) = f = composition_cm (id_cm C') f,
```

which we give here:

```
1  (* Composition with the Identity morphism is trivial *)
2  Proposition composition_id_cm_left :
3      forall (f : ContractMorphism C C'),
4      (composition_cm (id_cm C') f) = f.
5  Proof.
6      intros. unfold composition_cm. unfold id_cm. simpl.
7      apply is_eq_cm; simpl.
8      -   apply (is_eq_cm_init (composition_cm (id_cm C') f) f); auto.
9      -   apply (is_eq_cm_recv (composition_cm (id_cm C') f) f); auto.
10 Qed.
```

```
11
12 Proposition composition_id_cm_right :
13     forall (f : ContractMorphism C C'),
14     (composition_cm f (id_cm C)) = f.
15 Proof.
16     intros. unfold composition_cm. unfold id_cm. simpl.
17     apply is_eq_cm; simpl.
18     + apply (is_eq_cm_init (composition_cm f (id_cm C)) f); auto.
19     + apply (is_eq_cm_recv (composition_cm f (id_cm C)) f); auto.
20 Qed.
```

Listing 5.7: Left and right composition of the identity morphism is trivial.

Example 5.2.2 (Null Morphism)**.** Our next example is a *terminal contract*, also called a *null contract*. The terminal contract is a contract for which there are exactly two ways to interact with the contract. One succeeds and one fails. Because of this, every contract `C` has an essentially unique morphism into the null contract, which takes successful executions to the success call, and failed executions to the fail call.

First, the definition of the contract.

```
1  (** State *)
2  Record Null_State := { null_state : unit }.
3
4  (** Msg *)
5  Inductive Null_Msg :=
6  | null_msg (n : unit).
7
8  (** Setup *)
9  Definition Null_Setup := option unit.
10
11 (* one canonical error message *)
12 Definition Null_Error := unit.
13
14 (** Init/Recv Functions *)
15 Definition null_init
16     (_ : Chain)
17     (_ : ContractCallContext)
18     (null_init : Null_Setup) : result Null_State Null_Error :=
19         match null_init with
20         | Some _ => Ok {| null_state := tt |}
21         | None => Err tt
22         end.
23
24 Definition null_recv
25     (_ : Chain)
```

```
    (_ : ContractCallContext)
    (_ : Null_State)
    (op_msg : option Null_Msg) :
    result (Null_State * list ActionBody) Null_Error :=
        match op_msg with
        | Some _ => Ok ({| null_state := tt |}, [])
        | None => Err tt
        end.

(** the Terminal contract *)
Definition null_contract : Contract Null_Setup Null_Msg Null_State Null_Error :=
    build_contract null_init null_recv.
```

Listing 5.8: The Terminal Contract

The terminal contract is designed to contain only the most bare-bones structure of a smart contract possible. Thus the state has one constructor of type `unit`, and so can have no variation. It has one entrypoint, which accepts a message of type `unit`. The `receive` function succeeds if and only if it receives a message when called, and fails if it receives no message when called.

Consider a contract `C : ContractMorphism Setup Msg State Error` for arbitrary `Setup`, `Msg`, `State`, and `Error`. For each of the `init` and `receive` functions of `C`, we can divide inputs, including states of the chain and the call context, into those which lead to a success and those which lead to an error. This is all we need to construct a morphism from `C` into our contract `null_contract`.

We begin with the `init` component:

```
Definition morph_init_i (x : Chain * ContractCallContext * Setup) : Chain *
    ContractCallContext * Null_Setup :=
    let (x', s) := x in let (c, ctx) := x' in
    match (C.(init) c ctx s) with
    | Ok  _ => (c, ctx, Some tt)
    | Err _ => (c, ctx, None)
    end.

Definition morph_init_o (x : result State Error) : result Null_State Null_Error :=
    match x with
    | Ok  _ => Ok {| null_state := tt |}
    | Err _ => Err tt
    end.

Lemma null_init_commutes : init_morphs_commute C.(init) null_contract.(init) morph_init_i
    morph_init_o.
Proof.
```

```
    unfold init_morphs_commute.
    intro. destruct x as [x' s]. destruct x' as [c ctx]. simpl.
    unfold uncurry_fun3. unfold null_init. unfold morph_init_o.
    now destruct (init C c ctx s).
Qed.
```

Listing 5.9: The `init` component of a morphism in `ContractMorphism C null_contract`

And then move to the `receive` components:

```
Definition morph_recv_i (x : Chain * ContractCallContext * State * option Msg) :
    Chain * ContractCallContext * Null_State * option Null_Msg :=
    let (x', op_msg) := x in
    let (x'', st) := x' in
    let (c, ctx) := x'' in
    match (C.(receive) c ctx st op_msg) with
    | Ok  _ => (c, ctx, {| null_state := tt |}, (Some (null_msg tt)))
    | Err _ => (c, ctx, {| null_state := tt |}, None)
    end.

Definition morph_recv_o (x : result (State * list ActionBody) Error) :
    result (Null_State * list ActionBody) Null_Error :=
    match x with
    | Ok  _ => Ok ({| null_state := tt |}, [])
    | Err _ => Err tt
    end.

Lemma null_recv_commutes : recv_morphs_commute C.(receive) null_contract.(receive)
    morph_recv_i morph_recv_o.
Proof.
    unfold recv_morphs_commute. intro.
    destruct x as [x' op_msg]. destruct x' as [x'' st]. destruct x'' as [c ctx]. simpl.
    unfold uncurry_fun4. unfold null_recv. unfold morph_recv_o.
    now destruct (receive C c ctx st op_msg).
Qed.
```

Listing 5.10: The `receive` component of a morphism in `ContractMorphism C null_contract`

Our morphism is then given as follows:

```
(* the terminal morphism *)
Definition null_morphism : ContractMorphism C null_contract :=
    let morph_init := {|
        init_inputs   := morph_init_i ;
        init_outputs  := morph_init_o ;
        init_commutes := null_init_commutes ;
```

```
    |} in
    let morph_recv := {|
        recv_inputs   := morph_recv_i ;
        recv_outputs  := morph_recv_o ;
        recv_commutes := null_recv_commutes ;
    |} in {|
        cm_init := morph_init ;
        cm_recv := morph_recv ;
    |}.
```

Listing 5.11: The canonical morphism in `ContractMorphism C null_contract`

This morphism characterizes all possible calls to our contract `C`, by simple success and error. The formal, structural relationship that is expressed by this morphism is simply that `C` is a smart contract, and thus contains some storage, accepts some (or no) kind of message, and each call to `init` or `receive` results in either success or failure.

Example 5.2.3 (Isomorphism)**.** In our final example, we look at *isomorphisms* of contracts. These are morphisms for which the formal, structural relationship expressed between the contracts is that of an equivalence. We define these via a predicate:

```
Definition is_iso_cm (f : ContractMorphism C C') (g : ContractMorphism C' C) : Prop :=
    composition_cm g f = id_cm C /\
    composition_cm f g = id_cm C'.
```

Listing 5.12: An isomorphism of contracts

That is, `f` and `g` together form an isomorphism of contracts if, when composed in either direction, their composition is equal to the identity morphism. If we think of `f` and `g` as defining a binary relation of state transition systems, an isomorphism is, by definition, a bisiumlation of contracts.

5.2.5 Simple Morphisms

Contract morphisms have nice category-theoretic properties that we might expect, like associativity, a null morphism, the notion of equality of morphisms, and isomorphisms. Even so, when conceiving of a morphism in the wild, between actual smart contracts, and with the aim of codifying some structural relationship that will itself be useful for the specification and verification process of smart contracts, the definition we've given might be slightly too general to be of use in all cases.

Before we discuss the relationship of contract morphisms to proof and specification, we will introduce a subset of contract morphisms, which we call *simple morphisms*, so named because their construction is

slightly simpler than a generic contract morphism.

To construct a simple morphism we need functions between each of the contract types `Setup`, `Msg`, `State`, and `Error`, *etc.* and our commutativity result `init_coherence`. The definition of the receive component is similar. As before, we construct the `InitCM` and `RecvCM` components of our morphism, and then construct the morphism itself.

```
Definition simple_cm_init
    (* the components of f *)
    (setup_morph : Setup -> Setup')
    (state_morph : State -> State')
    (error_morph : Error -> Error')
    (* coherence conditions *)
    (init_coherence : forall c ctx s,
        (init_result_transform state_morph error_morph) ((init C1) c ctx s) = (init C2) c ctx
    (setup_morph s)) :
    InitCM C1 C2 := {|
        init_inputs := (fun (x : Chain * ContractCallContext * Setup) =>
        let '(c, ctx, s) := x in (c, ctx, setup_morph s)) ;
        init_outputs := (init_result_transform state_morph error_morph) ;
        init_commutes := init_commutes_simple setup_morph state_morph error_morph
    init_coherence ;
    |}.

Definition simple_cm_recv
    (* the components of f *)
    (msg_morph   : Msg   -> Msg')
    (state_morph : State -> State')
    (error_morph : Error -> Error')
    (* coherence conditions *)
    (recv_coherence : forall c ctx st op_msg,
        (recv_result_transform state_morph error_morph) ((receive C1) c ctx st op_msg) =
        (receive C2) c ctx (state_morph st) (option_fun msg_morph op_msg)) :
    RecvCM C1 C2 := {|
        recv_inputs :=
            (fun (x : Chain * ContractCallContext * State * option Msg) =>
                let '(c, ctx, st, op_msg) := x in (c, ctx, state_morph st, option_fun
    msg_morph op_msg)) ;
        recv_outputs :=
            (recv_result_transform state_morph error_morph) ;
        recv_commutes := recv_commutes_simple msg_morph state_morph error_morph recv_coherence
     ;
    |}.

(* the simple contract morphism *)
```

```
35 Definition simple_cm
36     (* the components of f *)
37     (msg_morph   : Msg   -> Msg')
38     (setup_morph : Setup -> Setup')
39     (state_morph : State -> State')
40     (error_morph : Error -> Error')
41     (* coherence conditions *)
42     (init_coherence : forall c ctx s,
43         (init_result_transform state_morph error_morph) ((init C1) c ctx s) = (init C2) c ctx
       (setup_morph s))
44     (recv_coherence : forall c ctx st op_msg,
45         (recv_result_transform state_morph error_morph) ((receive C1) c ctx st op_msg) =
46         (receive C2) c ctx (state_morph st) (option_fun msg_morph op_msg)) :
47     ContractMorphism C1 C2 := {|
48         cm_init := simple_cm_init setup_morph state_morph error_morph init_coherence ;
49         cm_recv := simple_cm_recv msg_morph   state_morph error_morph recv_coherence ;
50     |}.
```

Listing 5.13: The simple contract morphism

We finish with two lemmas, that simple morphisms are closed under composition and that the identity morphism is simple.

```
1  Lemma composition_simple_cm
2      (g : ContractMorphism C' C'')
3      (f : ContractMorphism C  C') :
4      is_simple_cm g ->
5      is_simple_cm f ->
6      is_simple_cm (composition_cm g f).
7  Proof.
8      (* omitted for length *)
9  Qed.
10
11 Lemma identity_simple
12     (C : Contract Setup Msg State Error) :
13     is_simple_cm (id_cm C).
14 Proof.
15     (* omitted for length *)
16 Qed.
```

Listing 5.14: Simple morphisms are closed under composition

5.3 Reasoning with Morphisms: Specification and Proof

Morphisms can be used in a variety of ways in proof and specification of smart contracts. We will highlight some of them here, including the examples we mentioned in §5.1.2 of specifying a contract upgrade in relation to the old (§5.3.1), patching a bug (§5.3.2), proving backwards compatibility (§5.3.3), and optimizing for gas efficiency (§5.3.4). We will also demonstrate a generic proof technique, which uses morphisms to prove Hoare-like properties of partial correctness (§5.3.5).

We will refer to the heuristics of these examples several times through the remainder of this thesis.

5.3.1 Specifying a Contract Upgrade With Morphisms

Example 5.3.1 (Uranium Finance Upgrade Specification)**.** Recall from §2.1.2 Uranium Finance, a contract which was exploited because developers replaced a constant `k` set at `1,000` with `10,000` in all but one of its instances during an upgrade. The result was wildly incorrect pricing, which rapidly drained their liquidity pools.

We do not have access to their source code, but suppose `C` is the Uranium Finance contract pre-upgrade, and `C'` is the contract post-upgrade, and suppose the upgrade was only to adjust how the contract prices trades. In this examplae we go further and assume that the upgrade was to increase the decimal precision of this calculation ten times, meaning that the internal token balances in storage have one more decimal place, and the trade calculation is able to calculate at one decimal place greater in precision.

The original contract `C` will have a `storage` type, then, which keeps track of internal token balances, as well as a `TRADE` entrypoint which accepts some message of type `entrypoint` whose payload type `trade_data` includes a desired trade quantity. We will also have some function `calculate_trade` which calculates how many tokens will be traded out for a given contract call to the `TRADE` entrypoint. The trade quantity, internal token balances, and the `calculate_trade` function will all be accurate up to some decimal place, commonly 6 or 9 in the wild.

When we upgrade `C` to `C'` with the express purpose of increasing the accuracy by one decimal place for internal token balances and trades, we will alter the storage type to `storage'` and the entrypoint type to `entrypoint'` where `TRADE` now accepts something of type `trade_data'`. We will also update `calculate_trade` to `calculate_trade'` which calculates at one higher decimal place of accuracy. This update consists of of just adding one decimal place to the internal token balances and the quantity parameter of the trade message payload.

If we had gone through the trouble of formally verifying `C`, and wished to do the same for `C'`, we might do so

by altering the specification of `C` until it now correctly specified the contract `C'` and then verify `C'` from the ground up. As most of the contract functionality did not change, this would require re-proving many results already proved about `C`, where the slight changes we made to `C` would likely make it impossible to simply copy/paste the proofs. Furthermore, as we saw in the previous chapter, specifications are difficult to write correctly, and small changes to the specification may have unintended consequences. Moreover, in altering the specification we might make the same error that the Uranium Finance engineers did and unintentionally create a vulnerability.

Instead, we could specify `C'` in relation to `C`: that `C'` should in some sense "do everything that `C` does," except that internal balances and trades should be calculated at one decimal place higher of precision. We can specify this rigorously through a simple morphism `f: ContractMorphism C' C` defined by:

- a function `msg_morph: entrypoint' -> entrypoint` which is the identity on every entrypoint except for on `TRADE`, where it rounds the traded quantity of the trade payload type `trade_data'` into something of type `trade_data`,
- a function `state_morph: state' -> state` which is the identity on everything in storage except internal token balances, which it also rounds the decimal place,
- a function `setup_morph: setup' -> setup` which is the identity unless it needs to round any data corresponding to token balances in the storage,
- a function `error_morph: error' -> error` which is the identity,
- and two proofs of coherence, showing that rounding the decimal precision is consistent with contract functionality.

The two coherence results show that the upgrade from `C` to `C'` was done as intended: we increased precision successfully without changing how trades were priced. We know this because we have that, after rounding, a trade on `C'` is the same as a trade on `C`. In particular, this is where the Uranium Finance engineers would have realized the pricing mechanism of the upgraded contract did not conform to that of the old, avoiding the catastrophic mistake and subsequent exploitation.

5.3.2 Specifying a Bug Fix With Morphisms

Example 5.3.2 (Fixing a Bug)**.** In this example we will illustrate how a bug fix can be understood and specified in with morphisms. Recall Nomad (from §2.1.2), a cross-chain bridge protocol which suffered a catastrophic attack because of a bug, introduced during an upgrade, which added the null address as a trusted root, enabling anyone to withdraw funds from the liquidity pool. As in Example 5.3.1, we do not

have access to the Nomad source code, but let us consider a simplified example of a bug due to incorrect permissions.

Consider a contract `C` which keeps a counter in storage `n:N`, which can be incremented by a wallet that has permission to do so. Permissions are tracked by a map in storage, `permissions:FMap address bool`, and a wallet increments the counter by calling the `incr` entrypoint. Those who have permissions can also add other addresses to the permissioned list. Furthermore, there are some wallets which we wish to prevent from ever incrementing `n`, so in the permissions map their entry reads `false`.

The code in ConCert is given here:

```
(* contract types definition *)
Inductive entrypoint :=
    | incr
    | add_permissions (addr : Address).
Record storage := build_storage { n : N ; permissions : FMap Address bool ; }.
Definition setup := FMap Address bool.
Definition error := N.
Definition result : Type := ResultMonad.result (storage * list ActionBody) error.

(* an auxiliary function to see if an address has permissions *)
Definition has_permissions (addr : Address) (permissions : FMap Address bool) : bool :=
    match FMap.find addr permissions with
    | Some b => b
    | None => false
    end.

(* init function definition *)
Definition init (_ : Chain)
                (_ : ContractCallContext)
                (init_permissions : setup)
                : ResultMonad.result storage N :=
    let init_state := {| n := 0 ; permissions := init_permissions ; |} in
    Ok (init_state).

(* receive function definition *)
Definition receive (_ : Chain)
                (ctx : ContractCallContext)
                (storage : storage)
                (msg : option entrypoint)
                : result :=
    match msg with
    | Some incr =>
        if has_permissions ctx.(ctx_from) storage.(permissions) then
            let st := {| n := storage.(n) + 1 ; permissions := storage.(permissions) ; |} in
```

```
            Ok (st, [])
        else Err 0
    | Some (add_permissions new_addr) =>
        if has_permissions ctx.(ctx_from) storage.(permissions) then
            let st := {|
                n := storage.(n) ;
                permissions :=
                    FMap.update new_addr (Some true) storage.(permissions) ; |} in
            Ok (st, [])
        else Err 0
    | None => Err 0
    end.

(* construct the contract *)
Definition C : Contract setup entrypoint storage error :=
    build_contract init receive.
```

`C` has a bug: a wallet with permissions can call `update_permissions` on a blacklisted wallet and turn its entry from `false` to `true`. The correct behavior would be, if a user with permissions wishes to update permissions for a blacklisted address, that it result in an error. Of course, in this simple case we can implement this straightforwardly by adding a permissions check, but as we saw in this case of Nomad in §2.1.2, it is not always obvious that a simply-stated condition, like that the null address not be a trusted root, holds for the contract.

How do we specify this bug fix with a contract morphism? We need a contract `C'` which behaves identically to `C`, except that now,

```
receive C' c ctx st msg = Err 0
```

if `FMap.find addr storage.(permissions) = Some false`. In the code above, this holds only if the address `ctx.(ctx_from)` does not have permissions.

To address this we will define a contract `C'` which implements this bug fix, and then show that it does so via a contract morphism. As `C'` only implements minor changes to `C`, most of the contract code is the same.

```
(* contract types definition *)
Definition entrypoint' := entrypoint.
Definition storage' := storage.
Definition setup' := setup.
Definition error' := error.
Definition result' := result.

(* init function definition *)
```

```
Definition init' (c : Chain)
                 (ctx : ContractCallContext)
                 (init_permissions : setup')
                 : ResultMonad.result storage' N :=
    init c ctx init_permissions.

(* receive function definition *)
Definition receive' (_ : Chain)
                 (ctx : ContractCallContext)
                 (storage : storage')
                 (msg : option entrypoint')
                 : result :=
    match msg with
    | Some incr =>
        if has_permissions ctx.(ctx_from) storage.(permissions) then
            let st := {| n := storage.(n) + 1 ; permissions := storage.(permissions) ; |} in
            Ok (st, [])
        else Err 0
    | Some (add_permissions new_addr) =>
        (* begin bug fix *)
        match FMap.find new_addr storage.(permissions) with
        | Some _ => Err 0
        | None =>
        (* end bug fix *)
            if has_permissions ctx.(ctx_from) storage.(permissions) then
                let st := {|
                    n := storage.(n) ;
                    permissions :=
                        FMap.update new_addr (Some true) storage.(permissions) ; |} in
                Ok (st, [])
            else Err 0
        end
    | None => Err 0
    end.

(* construct the contract *)
Definition C' : Contract setup' entrypoint' storage' error' :=
    build_contract init' receive'.
```

A close reading of the code reveals that the only difference between `C` and `C'` is that when a wallet calls `update_permissions` with payload `new_addr`, `C'` checks if `new_addr` already has an entry in its permissions records `storage.(permissions)`. If it does, the contract call fails. In particular, this means that a wallet with a `false` entry in `storage.(permissions)` will always have a `false` entry.

The bug fix takes the problematic execution—a call to `update_permissions` when `new_addr` already has an entry in `storage.(permissions)`—and diverts it from resulting in a success to resulting in `Err 0`. We can define a morphism which expresses a new structural relationship between our two contracts: `C'` behaves identically to `C` except in the one case that the bug fix targets.

```
(* InitCM component definition *)
Definition init_inputs : Chain * ContractCallContext * setup'
    -> Chain * ContractCallContext * setup := id.
Definition init_outputs : ResultMonad.result storage' error'
    -> ResultMonad.result storage error := id.
Lemma init_commutes : forall x : Chain * ContractCallContext * setup',
    uncurry_3 (Blockchain.init C) (init_inputs x) =
    init_outputs (uncurry_3 (Blockchain.init C') x).
Proof. auto. Qed.

Definition cm_init : InitCM C' C := {|
    ContractMorphisms.init_inputs := init_inputs ;
    ContractMorphisms.init_outputs := init_outputs ;
    ContractMorphisms.init_commutes := init_commutes ; |}.

(* RecvCM component definition *)
Definition recv_inputs
    (x : Chain * ContractCallContext * storage' * option entrypoint') :
        Chain * ContractCallContext * storage * option entrypoint :=
    let '(c, ctx, storage, msg) := x in
    match msg with
    | Some (add_permissions new_addr) =>
        match FMap.find new_addr storage.(permissions) with
        | Some _ => (c, ctx, storage, None)
        | _ => x
        end
    | _ => x
    end.

Definition recv_outputs : ResultMonad.result (storage' * list ActionBody) error' ->
    ResultMonad.result (storage * list ActionBody) error := id.


Lemma recv_commutes :
    forall (x : Chain * ContractCallContext * storage' * option entrypoint'),
        uncurry_4 (Blockchain.receive C) (recv_inputs x) =
        recv_outputs (uncurry_4 (Blockchain.receive C') x).
Proof.
    intros. cbn.
    destruct x as [x msg']. destruct x as [x state']. destruct x as [c ctx].
```

```
    induction msg'; cbn; auto.
    induction a; cbn; auto.
    induction (FMap.find addr (permissions state')); cbn; auto.
Qed.

Definition cm_recv := {|
    ContractMorphisms.recv_inputs := recv_inputs ;
    ContractMorphisms.recv_outputs := recv_outputs ;
    ContractMorphisms.recv_commutes := recv_commutes ;
|}.

(* contract morphism definition *)
Definition f : ContractMorphism C' C := {|
    ContractMorphisms.cm_init := cm_init ;
    ContractMorphisms.cm_recv := cm_recv ;
|}.
```

The morphism `f` defines the logic of the bug fix. We were able to precisely indicate *what* functionality we expected to change, and *how* we expected it to change, through the contract update. We isolated the problematic contract execution and showed with `f` that they now result in an error.

Our morphism `f` also tells us that our patch isolated the issue, and that none of the other functionality of `C` was inadvertently changed. This methodology is thus relevant to the Nomad hack, which exposed a vulnerability by inadvertently changing too much with their upgrade.

5.3.3 Adding Features and Backwards Compatibility

Example 5.3.3 (Backwards Compatibility)**.** Consider two contracts

```
C : Contract setup entrypoint storage error
C': Contract setup' entrypoint' storage' error'
```

Suppose in the course of an upgrade from contracts `C` to `C'`, we wish to show that `C'` is backwards compatible with `C`, meaning that all the functionality of `C` is preserved in `C'`.

This can be expressed via a contract morphism in a very precise way: one can not only prove that `C'` is backwards compatible with `C`, but also indicate exactly *how*—we can indicate what entrypoints and actions in the new contract correspond to the functionality of the old.

We do this via a simple morphism. To indicate which entrypoints in `C'` correspond to the old, we need to define a function `msg_morph : entrypoint -> entrypoint'`, where `e : entrypoint` is sent to its

corresponding entrypoint `msg_morph e`. Similarly, functions `state_morph : storage -> storage'` and `error_morph : error -> error'` indicate how the storage and errors of `C` take form in the new storage and error types.

We illustate with an example of a counter contract `C` which keeps `n : N` in storage and has one entrypoint `incr`, which is updated to `C'` that includes a `decr` entrypoint and is backwards compatible with `C`.

```
(** The initial contract C *)
(* contract types definition *)
Inductive entrypoint := | incr (u : unit).
Definition storage := N.
Definition setup := N.
Definition error := N.
Definition result : Type := ResultMonad.result (storage * list ActionBody) error.

(* init function definition *)
Definition init (_ : Chain) (_ : ContractCallContext) (n : setup) :
    ResultMonad.result storage N :=
    Ok (n).

(* receive function definition *)
Definition receive (_ : Chain) (_ : ContractCallContext) (n : storage)
    (msg : option entrypoint) : result :=
    match msg with
    | Some (incr _) => Ok (n + 1, [])
    | None => Err 0
    end.

(* construct the contract *)
Definition C : Contract setup entrypoint storage error :=
    build_contract init receive.


(** The updated contract C' *)
(* contract types definition *)
Inductive entrypoint' := | incr' (u : unit) | decr (u : unit).

(* receive function definition *)
Definition receive' (_ : Chain) (_ : ContractCallContext) (n : storage)
    (msg : option entrypoint') : result :=
    match msg with
    | Some (incr' _) => Ok (n + 1, [])
    | Some (decr _) => Ok (n - 1, [])
    | None => Err 0
    end.
```

```
39
40 (* construct the contract *)
41 Definition C' : Contract setup entrypoint' storage error :=
42     build_contract init receive'.
43
44
45 (** The contract morphism confirming backwards compatibility *)
46 Definition msg_morph (e : entrypoint) : entrypoint' :=
47     match e with | incr _ => incr' tt end.
48 Definition setup_morph : setup -> setup := id.
49 Definition state_morph : storage -> storage := id.
50 Definition error_morph : error -> error := id.
51
52 (* the coherence results *)
53 Lemma init_coherence : forall c ctx s,
54     (init_result_transform state_morph error_morph) ((Blockchain.init C) c ctx s) =
55     (Blockchain.init C') c ctx (setup_morph s).
56 Proof. auto. Qed.
57
58 Lemma recv_coherence : forall c ctx st op_msg,
59     (recv_result_transform state_morph error_morph) ((Blockchain.receive C) c ctx st op_msg) =
60     (Blockchain.receive C') c ctx (state_morph st) (option_map msg_morph op_msg).
61 Proof.
62     intros. cbn. unfold recv_result_transform.
63     unfold msg_morph. unfold state_morph. unfold error_morph.
64     cbn. induction op_msg; cbn; auto.
65     destruct a. auto.
66 Qed.
67
68
69 (* construct the morphism *)
70 Definition f : ContractMorphism C C' :=
71     simple_cm msg_morph setup_morph state_morph error_morph init_coherence recv_coherence.
```

The coherence proofs in the definition of `f` prove that `incr'` as an entrypoint behaves exactly as `init` did in `C`, and `f` tells us how to go from interacting with `C` to interacting with its update `C'`: any call to `incr` should now simply be made to `incr'`, and we have the exact same functionality preserved.

5.3.4 Improving Gas Efficiency

Example 5.3.4 (Optimizing Performance)**.** Suppose that we have a contract `C` which we know is correct, but we would like it to be more gas efficient. The goal of this (and any other optimization) is to make

improvements to the contract's performance without changing its behavior. Since ConCert doesn't model gas consumption, any changes we make to optimize `C` should result in a contract `C'` that has identical behavior from the perspective of ConCert.

This can be expressed by a bisimulation, so we can prove that contract optimizations preserve the contract's functionality by producing an isomorphism of contracts. That is, a pair of simple morphisms `f : ContractMorphism C C'` and `g : ContractMorphism C' C` such that the `msg_morph` component of each of `f` and `g` is the identity function, and `is_iso_cm f g`.

In particular, this could help us first prove desired properties of a contract optimized for readability and formal proof, and then optimize the contract for performance, something which often stands in conflict with intelligibility and readability.

5.3.5 Transporting Hoare-Like Properties Over a Morphism

Finally, we introduce a generic proof technique which uses the coherence proofs of a contract morphism to prove Hoare-like properties of a given contract `C` using contract morphisms.

Recall from our specification from §4.2 the property `unpool_emits_transfer` which we saw in Listing **??**. On line 11 of Listing **??**, we assume that the contract executes without error

```
receive contract chain ctx state (Some msg) = Ok(cstate', acts)
```

where `msg` is of the form `unpool(msg_payload)` and `cstate'` is the updated state after the transaction. The remainder of `unpool_emits_transfer` is various properties of the list of emitted transactions, `acts`, stating that there has to be a `transfer` transaction in the list and specifying what the payload of that transaction needs to look like. A proof of `unpool_emits_transfer` consists of constructing an outgoing transaction which is emitted by the contract.

More fundamentally, `unpool_emits_transfer` is a Hoare-like assertion of partial correctness, reasoning about a successful call to the structured pool contract, where the preconditions are that the state of the chain be reachable with contract address `caddr` and state `cstate`, and the postconditions are statements about `acts` and `cstate'` including that `acts` contain an appropriate `TRANSFER` transaction. Indeed, a reflection on the formal specification of §4.2 reveals that most of the properties of the specification are of this form.

A proof of `unpool_emits_transfer` and properties like it require constructing a particular execution of the `receive` function which satisfies the appropriate postconditions. Consider contracts `C` and `C'` and morphism `f : ContractMorphism C C'`, and suppose that `C` satisfies `unpool_emits_transfer`. To prove that `C'`

also satisfies `unpool_emits_transfer`, we can either directly construct an execution of `receive C'`, the receive function of `C'`, satisfying the appropriate conditions, or we can construct one of `receive C`, the receive function of `C`, which is sent to an execution satisfying `unpool_emits_transfer` under `f`.

Following the latter strategy, we must find inputs to `receive C` which, under `f`, satisfy the preconditions of `unpool_emits_transfer` of giving a reachable state and an `unpool` message. Because of the coherence results which come with `f`, namely that

```
receive C' (f chain) (f ctx) (f state) (f (Some msg)) = f (Ok(cstate', acts))
```

where

```
receive C chain ctx state (Some msg) = Ok(cstate', acts),
```

and we know that `acts` satisfies the postconditions of `unpool_emits_transfer`, depending on `f` we may be able to take advantage of the proofs of coherence of the morphism `f`, and use the proof about `cstate` or `acts` to show the same about `f cstate` and `f acts`.

We can also go the other way, going from an execution of `C` to `C'` to prove something about `C` using knowledge that a similar result holds for `C'`. In both cases, we take advantage of the coherence results given in a contract morphism.

Example 5.3.5 (Transporting a Property From the Specification)**.** Consider contracts

```
C  : Contract setup  entrypoint  storage  error
C' : Contract setup' entrypoint' storage' error'
```

and assume `(is_sp : is_structured_pool C')`. Suppose further that

```
setup = setup', storage = storage', error = error',
```

and that `entrypoint` and `entrypoint'` are defined as follows:

```
Inductive entrypoint :=
| Pool : pool_data -> entrypoint
| Unpool : unpool_data -> entrypoint.

Inductive entrypoint' :=
| Pool' : pool_data -> entrypoint'
| Unpool' : unpool_data -> entrypoint'
| Trade' : trade_data -> entrypoint'.
```

This gives us a function between entrypoint types

```
Definition embed_entrypoint (e : entrypoint) : entrypoint' :=
match e with
| Pool p => Pool' p
| Unpool p => Unpool' p
end.
```

Finally, assume that the functionality of `C` regarding its `Pool` and `Unpool` entrypoints is identical to that of `C'` regarding its `Pool'` and `Unpool'` entrypoints. That is,

```
Definition init_coherence_prop : Prop :=
    forall (c : Chain) (ctx : ContractCallContext) (s : setup),
    init C c ctx s = init C' c ctx s.
Axiom init_coherence_pf : init_coherence_prop.

Definition recv_coherence_prop : Prop :=
    forall (c : Chain) (ctx : ContractCallContext) (st : storage) (op_msg : option entrypoint)
    ,
    receive C  c ctx st op_msg =
    receive C' c ctx st (option_map embed_entrypoint op_msg).
Axiom recv_coherence_pf : recv_coherence_prop.
```

Then we can construct a simple morphism `f : ContractMorphism C C'` as follows:

```
(* simple morphism constructors *)
Definition msg_morph : entrypoint -> entrypoint' := embed_entrypoint.
Definition setup_morph : setup -> setup' := id.
Definition state_morph : state -> state' := id.
Definition error_morph : error -> error' := id.

(* coherence results, for which we used the assumed results above *)
Lemma init_coherence : forall c ctx s,
    (init_result_transform state_morph error_morph) ((Blockchain.init C) c ctx s) =
    (Blockchain.init C') c ctx (setup_morph s).
Proof.
    intros.
    unfold setup_morph, state_morph, error_morph, init_result_transform. simpl.
    rewrite init_coherence_pf.
    destruct (init C' c ctx s); auto.
Qed.

Lemma recv_coherence : forall c ctx st op_msg,
    (recv_result_transform state_morph error_morph) ((Blockchain.receive C) c ctx st op_msg) =
    (Blockchain.receive C') c ctx (state_morph st) (option_map msg_morph op_msg).
```

```
Proof.
    intros.
    unfold state_morph, error_morph, msg_morph, recv_result_transform. simpl.
    rewrite recv_coherence_pf.
    destruct (receive C' c ctx st (option_map embed_entrypoint op_msg)); auto.
    destruct t; auto.
Qed.

(* the simple morphism *)
Definition f : ContractMorphism C C' :=
    simple_cm msg_morph setup_morph state_morph error_morph init_coherence recv_coherence.
```

Now suppose that we wish to prove `unpool_emits_transfer C`.

```
Theorem pullback_unpool_emits_transfer : unpool_emits_transfer C.
```

Recall that we already have a proof of `unpool_emits_transfer C'`. Furthermore, we know that for all messages `msg` into `C`,

```
receive C c ctx st (Some msg) = receive C' c ctx st (Some (embed_entrypoint msg))
```

Since in proving `unpool_emits_transfer C` we assume a successful execution

```
receive C c ctx st (Some msg) = Ok(cstate', acts)
```

we know that

```
receive C' c ctx st (Some embed_entrypoint msg) = Ok(cstate', acts)
```

also holds. Because we know `unpool_emits_transfer C'`, we know that `acts` satisfies the postconditions of `unpool_emits_transfer`, and thus we have a proof of `unpool_emits_transfer C`.

The coherence results are easier to access in a proof the more `C` and `C'` are similar to each other, as we might see in a contract fork or upgrade. In particular, if a contract is altered slightly to be upgraded, it could save tedious work of re-proving specified properties or re-specifying by simply using what is already known about the old contract to prove things about the new.

5.4 Revisiting Contract Upgrades

Recall from §5.1 that reasoning formally about contract upgradeability has at least two important aspects: understanding the bounds of a contract's upgradeability—what *exactly* is mutable and what is not?—and reasoning about new contract versions, possibly in relation to the old. In this section we will use contract morphisms to mathematically characterize a contract's upgradeability and discuss the practicalities of specifying iterated contract upgrades. Before doing so, we go into greater detail on upgradeable contracts.

5.4.1 The Varieties of Upgradeable Contracts

Upgradeable contracts vary, ranging from limited upgradeability to highly flexible frameworks like we saw in the Diamond standard.

Starting from the limited end, take for example MakerDAO, the contract that governs the stablecoin DAI. Certain contract parameters can be updated through a vote by MKR token holders, such as the *stability rate*, which is an interest rate that affects the price of DAI [122]. While MakerDAO cannot radically change the contract features and functionality, it can be updated by governance token holders in these limited ways.

Upgradeable contracts commonly come in the form of decentralized autonomous organizations (DAOs), which are smart contracts that are governed collectively by holders of a governance token [89]. In some DAOs, governance token holders can vote to disburse contract funds or engage in other kinds of corporate governance [190]. In others, governance token holders can also vote to alter, and in some cases fully upgrade, contract functionality. These contracts are governed through a complex web of tokens and incentives, perhaps best-described game-theoretically, which is a fully-fledged research topic in its own right [62].

Moving toward more upgradeability features, take for example Murmuration, a generic DAO template built on Tezos [11]. Murmuration includes a governance token contract and a DAO contract which is governed by governance token holders. A user submits a proposal which consists of an anonymous lambda function and various metadata. If the proposal passes and executes, which is determined through a voting procedure by governance token holders, the lambda of the proposal becomes the new contract functionality, replacing an entrypoint function. By definition, a lambda has few restrictions, if any, so the upgradeability afforded by Murmuration is substantially greater than that of MakerDAO.

Finally, as we saw before the Diamond upgrade framework [10] is an extremely flexible system of proxy contracts, which essentially allows for fully altering contract functionality. In addition to changing entrypoint functions like Murmuration, contracts conforming to the Diamond standard can also add entrypoints to the contract. Again, the Diamond framework is in some sense more upgradeable than Murmuration.

Each of these examples lend themselves to an intuitive notion of "how upgradeable" they are, and we can say with confidence that the Diamond framework is affords greater upgradeability features than Murmuration, which in turn is more flexible than MakerDAO. But what does this mean, formally?

5.4.2 Mutable and Immutable Parts: Splitting an Upgradeable Smart Contract

To formally understand upgradeable contracts, we need language which can characterize its upgradeability properties—by precisely identifying its immutable and mutable parts—as well as language for the governance process and how the upgrade process relates to the actual contract functionality.

To that end, let us now consider an upgradeable contract in relation to its mutable and immutable parts.

First, the immutable part, which we informally call its *skeleton*. The immutable part of an upgradeable contract is what governs the upgradeability framework and corresponding incentive and game-theoretic structure of the upgrade process. It indicates what about the contract can change, and through what process it can change, similar to constitutional rules which govern legislative rule-making: the United States Congress is free to make laws so long as they are made in accordance with and do not breach the United States Constitution. The Constitution guarantees a certain legal structure that all reachable legistlative states have in common.

How do we express this mathematically? Recall Example 5.3.1, where we showed that all the trades in an upgraded contract `C'` are priced the same as those in the previous version `C`, after rounding. Effectively, the morphism `f : ContractMorphism C' C` compresses executions of `C'` to those in `C`: any two trades which round to the same prices are sent to the same execution. We say that they are *identified* under `f`. One way to interpret this would be that `f` shows that `C'` conforms to a pattern set by `C` in how it prices and executes trades. Another way to look at it is that `f` *forgets* some information—the extra decimal places of precision—of `C'`.

This type of morphism which compresses and categorizes executions of a contract, called a *surjective* morphism, can help us see structure of a contract which persists over any reachable state. We will show in coming examples that a surjective morphism can help us identify the immutable part of an upgradeable contract, and reason about the contract in terms of its skeleton.

Before giving a formal definition of a surjective morphism, we define a the notion of a surjective function for a generic type:

```
1 Definition is_surj {A B : Type} (f : A -> B) : Prop :=
2     forall (b : B), exists (a : A), f a = b.
```

Definition 5.4.1 (Surjection of Contracts)**.** Consider contracts `C: Contract Setup Msg State Error` and `C': Contract Setup' Msg' State' Error'`. A contract morphism `f : ContractMorphism C C'` is a *surjection* if both the `init_inputs` component of `InitCM f` and the `recv_inputs` component of `RecvCm f` satisfy `is_surj`.

Formalized in ConCert, we have a predicate `is_surj_cm` which is defined as follows:

```
Definition is_surj_cm (f : ContractMorphism C C') : Prop :=
    (* init component surjects *)
    is_surj (init_inputs C C' (init_cm f)) /\
    (* recv component surjects *)
    is_surj (recv_inputs C C' (recv_cm f)).
```

Example 5.4.1 (Surjection Onto a Skeleton)**.** Consider a contract `C` whose storage contains some natural number `n : N` and a function `s : N -> N`, and which has two entrypoints: a `next` entrypoint, which applies `s` to `n`, and updates the number in storage with `(s n)`; and an `upgrade_fun` entrypoint, which accepts a parameter `s' : N -> N`, and replaces `s` in storage with `s'`. This contract is upgradeable in the sense that the functionality of `next`, the primary way in which a user can act on the number in contract storage, can be changed by calling `upgrade_fun`.

We define the contract in ConCert as follows:

```
(* contract types definition *)
Inductive entrypoint :=
    | next (u : unit)
    | upgrade_fun (s' : N -> N).
Record storage := build_storage {
    n : N ;
    s : N -> N ; }.
Definition setup := storage.
Definition error := N.
Definition result : Type := ResultMonad.result (storage * list ActionBody) error.

(* init function definition *)
Definition init (_ : Chain)
                (_ : ContractCallContext)
                (init_state : setup)
                : ResultMonad.result storage N :=
    Ok (init_state).

(* receive function definition *)
Definition receive (_ : Chain)
                (_ : ContractCallContext)
                (storage : storage)
```

```
                (msg : option entrypoint)
                : result :=
    match msg with
    | Some (next _) =>
        let st := {| n := storage.(s) storage.(n) ; s := storage.(s) ; |} in
        Ok (st, [])
    | Some (upgrade_fun s') =>
        let st := {| n := storage.(n) ; s := s' ; |} in
        Ok (st, [])
    | None => Err 0
    end.

(* construct the contract *)
Definition C : Contract setup entrypoint storage error :=
    build_contract init receive.
```

Now, let's consider what the immutable part of `C` is. Because the upgradeability refers to the functionality of the `upgrade_fun` entrypoint rather than `next`, we would say that the skeleton of upgradeable `C` has some natural number (it doesn't matter which) and some function `s : N -> N` (again, it doesn't matter which) in storage. It only has two entrypoints, `next` and `upgrade_fun`.

To express this, we define a skeleton contract of `C`, which is identical to `C` except that the value of `n` in storage is always `0 : N`, the value of `s` is always the constant function at zero `(fun (n : N) => 0)`, and the `upgrade_fun` entrypoint does nothing. This removes the details of precisely *which* number `n` and function `s` are in storage by fixing them at trivial values, but preserves the rest of the structure of the contract as an immutable skeleton. Importantly, invariance results about `C` which are agnostic to the number `n` and function `s` in storage will also hold for `C_s`; given the appropraite morphism we can prove them about `C` in relation to `C_s`.

We define our skeleton contract `C_s` in ConCert as follows.

```
(* contract types definition *)
Definition setup_s := unit.
Inductive entrypoint_s :=
    | next_s (u : unit)
    | upgrade_fun_s (s' : N -> N).
Record storage_s := build_storage_s { n_s : N ; s_s : N -> N ; }.
Definition error_s := N.
Definition result_s : Type := ResultMonad.result (storage_s * list ActionBody) error_s.

(* init function definition *)
Definition init_s (_ : Chain)
                (_ : ContractCallContext)
```

```
                (_ : setup_s)
                : ResultMonad.result storage_s N :=
    let storage_s := {|
        n_s := 0 ;
        s_s := (fun (n : N) => 0) ; |} in
    Ok (storage_s).

(* receive function definition *)
Definition receive_s (_ : Chain)
                (_ : ContractCallContext)
                (storage_s : storage_s)
                (msg : option entrypoint_s)
                : result_s :=
    match msg with
    | Some (next_s _) =>
        let st := {| n_s := storage_s.(s_s) storage_s.(n_s) ; s_s := storage_s.(s_s) ; |} in
        Ok (st, [])
    | Some (upgrade_fun_s _) => Ok (storage_s, [])
    | None => Err 0
    end.

(* construct the contract *)
Definition C_s : Contract setup_s entrypoint_s storage_s error_s :=
    build_contract init_s receive_s.
```

Note that the function `receive` still applies `s` to `n` when it receives a `next` message. However, because `C_s` is agnostic to the choice of `n` and `s`, `n` always remains constant at `0` and `s` remains constant at the function $\lambda x.0$. Similarly, the entrypoint `upgrade_fun` can be called, but it does nothing to the function in storage.

Now let us construct a surjective morphism of contracts `f_s : ContractMorphism C C_s`.

```
(* using the simple morphism construction *)
Definition msg_morph (e : entrypoint) : entrypoint_s :=
    match e with
    | next _ => next_s tt
    | upgrade_fun s' => upgrade_fun_s s'
    end.
Definition setup_morph : setup -> setup_s := (fun (_ : setup) => tt).
Definition state_morph (st : storage) : storage_s :=
    {| n_s := 0 ; s_s := (fun (n : N) => 0) ; |}.
    (* {| n_s := st.(n) ; s_s := st.(s) |}. *)
Definition error_morph : error -> error_s := id.

(* the coherence results *)
Lemma init_coherence : forall c ctx s,
```

```
    (init_result_transform state_morph error_morph) ((Blockchain.init C) c ctx s) =
    (Blockchain.init C_s) c ctx (setup_morph s).
Proof. auto. Qed.

Lemma recv_coherence : forall c ctx st op_msg,
    (recv_result_transform state_morph error_morph) ((Blockchain.receive C) c ctx st op_msg) =
    (Blockchain.receive C_s) c ctx (state_morph st) (option_map msg_morph op_msg).
Proof.
    intros. cbn.
    destruct op_msg; auto.
    destruct e; auto.
Qed.

(* construct the morphism *)
Definition f_s : ContractMorphism C C_s :=
    simple_cm msg_morph setup_morph state_morph error_morph init_coherence recv_coherence.
```

The morphism `f_s` is primarily defined through `msg_morph`, `state_morph`, and `setup_morph`, all of which forget the details of their respective types. More precisely, `state_morph` is simply the constant function which sends `n` to `0` and `s` to $\lambda x.0$; `setup_morph` is the constant function at `tt : nil`; and `msg_morph` though remembers the message and payload, we can see by the function `receive C_s` that the message payload is essentially discarded. Only the `error` type is preserved.

The coherence results are trivial to prove because because the morphisms of contract types are either identity or they force entrypoint or storage values into trivial values. The functions `init_s` and `receive_s` compress all the possible ways to initialize and call the contract into one option for initializing and only a few for calling contract entrypoints.

Proving surjectivity is similarly straightforward, because there are only three possible executions of `receive C_s`: calling `next_s`, which takes `0 : N` to `0 : N`, calling `upgrade_fun_s`, which does nothing to contract storage; or calling the contract with no message, which results in the same error as `C`. Each of these three executions has a preimage under `f_s`: an arbitrary call to `next`, an arbitrary call to `upgrade_fun`, and a call with no message. Similarly, there is only one execution for calling `init C_s`, which has a preimage under `f_s` being any initialization of `C`.

```
Theorem f_s_is_surj : is_surj_cm f_s.
Proof.
    (* proof omitted *)
Qed.
```

Now that we have the skeleton contract `C_s` and surjection `f_s`, what have we gained? Intuitively, because

`f_s` is a surjection, we have in `C_s` a classification of possible executions of `C` in terms of `C_s`: a contract call is to one of two entrypoints, `next` and `upgrade_fun`; since both of these act on storage and don't emit any transactions, `C_s` is agnostic to the functionality of both of these because it is agnostic to what is in storage.

However, `C_s` does give us a generic structure of the contract which holds no matter what upgrade the contract is on. Indeed, surjective functions are good for proving invariants. Recall from §?? the formal specification of our structured pool contract. Many of the propositions in the specification had the condition

```
...
(* reachable bstate *)
reachable bstate ->
...
```

in the hypothesis, generally used to show an invariant which holds for all reachable states of the contract. Because surjections simplify and categorize contract executions, given the right morphism it is possible to prove invariants about `C` using its skeleton `C_s`, using techniques we saw in §5.3.

Let us now move to the mutable part, which we informally call the *mutable functionality*. This is the contract's functionality which stands apart from the upgradeability framework skeleton. Were it not implemented as an upgradeable contract, it could in principle be implemented as a standalone, non-upgradeable contract. The current state of the upgradeable contract in some real sense emulates this contract from within the upgradeability skeleton.

The morphism that denotes this phenomenon is an *embedding*, or an `injection`, which is a structure-preserving morphism. This means that if `f : ContractMorphism C C'` is an embedding, then all the functionality of `C` is fully preserved in `C'`. Another way of thinking about it is that `C'` has a *copy* of `C` inside of it.

Before giving a formal definition, we define the notion of an injective function for a generic type:

```
Definition is_inj {A B : Type} (f : A -> B) : Prop :=
    forall (a a' : A), f a = f a' -> a = a'.
```

Definition 5.4.2 (Embedding of Contracts)**.** Consider contracts `C: Contract Setup Msg State Error` and `C':Contract Setup' Msg' State' Error'`. A contract morphism `f:ContractMorphism C C'` is an *embedding* if both the `init_inputs` component of `InitCM f` and the `recv_inputs` component of `RecvCm f` satisfy `is_inj`.

Formalized in ConCert, we have a predicate `is_inj_cm` which is defined as follows:

```
Definition is_inj_cm (f : ContractMorphism C C') : Prop :=
```

```
    (* init component injects *)
    is_inj (init_inputs  C C' (init_cm f)) /\
    (* recv component injects *)
    is_inj (recv_inputs  C C' (recv_cm f)).
```

Example 5.4.2 (Contract Embedding)**.** Consider `C` from Example 5.4.1 which, through any number of upgrades, has some `n:N` and `s_next:N -> N` in storage. Now we wish to isolate a contract which describes the functionality that is current to this particular upgraded version of the contract.

Rather than distilling the upgradeability skeleton as we did in Example 5.4.1, we remove the upgradeability features and isolate the current functionality of the contract. That is, we remove the `upgrade_fun` entrypoint and `s` from storage, using a (fixed) `s_next:N -> N` when the `next` entrypoint is called.

```
(* contract types definition *)
Definition setup_i := N.
Inductive entrypoint_i := | next_i (u : unit).
Record storage_i := build_storage_i { n_i : N ; }.
Definition error_i := N.
Definition result_i : Type := ResultMonad.result (storage_i * list ActionBody) error_i.

(* init function definition *)
Definition init_i (_ : Chain)
                  (_ : ContractCallContext)
                  (n : setup_i)
                  : ResultMonad.result storage_i N :=
    let storage_i := {| n_i := n ; |} in
    Ok (storage_i).

(* receive function definition *)
Axiom s_next : N -> N.

Definition receive_i (_ : Chain)
                  (_ : ContractCallContext)
                  (storage_i : storage_i)
                  (msg : option entrypoint_i)
                  : result_i :=
    match msg with
    | Some (next_i _) =>
        let st := {| n_i := s_next storage_i.(n_i) ; |} in
        Ok (st, [])
    | None => Err 0
    end.

(* construct the contract *)
Definition C_i : Contract setup_i entrypoint_i storage_i error_i :=
```

```
33      build_contract init_i receive_i.
```

The contract `C_i` can be called to update its internal natural number in storage, but it is unable to upgrade in the sense that `s_next` is fixed. In contrast to `C_s`, `C_i` can take on any values in its storage which can be reached through `s_next`. Importantly, were we to initialize `C` with `s_next` and never call `upgrade_fun`, then `C` and `C_i` would be identical behaviorally. It is only through upgrades that `C` and `C_i` would have diverging behavior.

In contrast to the surjection `f_s : ContractMorphism C C_s` from Example 5.4.1, let us construct an embedding `f_i : ContractMorphism C_i C`.

```
1  (* using the simple morphism construction *)
2  Definition msg_morph' (e : entrypoint_i) : entrypoint :=
3      match e with
4      | next_i _ => next tt
5      end.
6  Definition setup_morph' (st_i : setup_i) : setup := {|
7      n := st_i ;
8      s := s_next ; |}.
9  Definition state_morph' (st_i : storage_i) : storage :=
10     {| n := st_i.(n_i) ; s := s_next ; |}.
11 Definition error_morph' : error_i -> error := id.
12
13 (* the coherence results *)
14 Lemma init_coherence' : forall c ctx s,
15     (init_result_transform state_morph' error_morph') ((Blockchain.init C_i) c ctx s) =
16     (Blockchain.init C) c ctx (setup_morph' s).
17 Proof. auto. Qed.
18
19 Lemma recv_coherence' : forall c ctx st op_msg,
20     (recv_result_transform state_morph' error_morph') ((Blockchain.receive C_i) c ctx st
       op_msg) =
21     (Blockchain.receive C) c ctx (state_morph' st) (option_map msg_morph' op_msg).
22 Proof.
23     intros. cbn.
24     destruct op_msg; auto.
25     destruct e; auto.
26 Qed.
27
28 (* construct the morphism *)
29 Definition f_i : ContractMorphism C_i C :=
30     simple_cm msg_morph' setup_morph' state_morph' error_morph'
31         init_coherence' recv_coherence'.
```

Proving injectivity is straightforward because each of the component morphisms of the simple morphism are injections.

```
Theorem f_s_is_inj : is_inj_cm f_s.
Proof.
    (* proof omitted *)
Qed.
```

This morphism and proof of embedding indicates that any execution of `C_i` can be emulated in `C` by simply initializing the contract correctly and not calling the upgrade function. There is a *copy* of `C_i` in `C`, though we should point out that because `C` is upgradeable, it includes "copies" of many more contracts similar to `C_i`, but with different functions `s : N -> N` in storage. It is precisely because of `C`'s upgradeability that it can move between all these instances of static contracts, emulating each in turn.

In Example 5.4.1 we argued that the surjection `f_s : ContractMorphism C C_s` can help us prove invariants about `C`, using techniques from §5.3. In contrast, `f_i : ContractMorphism C_i C` does not help us with invariants—it does not categorize contract executions of `C`, but rather models a set of possible contract executions if `C` is upgraded to have the appropriate morphism `s : N -> N` in storage.

Rather, embeddings of contracts can be used to prove the existence of a reachable state. For example, think again back to our formal specification of the structured pool contract (§**??**), and recall the formulation of Property 5, the Arbitrage Sensitivity property of the metaspecification. Property 5 states that under certain preconditions, there exists a reachable contract state that equalizes prices between the structured pool contract and an external market.

Applied to our case here, to prove that `C` can take on any value of `n:N` in its storage, we could show an embedding of `C_i` which initializes at `1:N` and for which `s_next` is $\lambda x.x + 1$. Because `C_i` can take on the value of any `n:N`, and `C_i` embeds into `C`, `C` can also take on any value of `n:N` in storage.

The information that `C_i` isolates and embeds through `f_i` is *precisely* the information about `C` that `C_s` forgets through `f_s`. `C_s` completely nullifies any particular value of `n : N` or `s : N -> N` in storage, instead isolating the invariant structure of the contract. The nullified values are precisely captured by the non-upgradeable contract `C_i`.

In choosing `C_s` and `C_i`, we were guided by intuition rather than a precise notion of what it means for a contract `C_s` to be the *skeleton* of `C`, and what contract precisely represents the contract's *functionality* `C_f`. Importantly, however, choosing `C_s` and `C_i` gave us precise contracts which we can use to describe the immutable and mutable parts of smart contracts. They are useful insofar as they facilitate proofs of an upgradeable contract—both to prove invariants or the existence of certain reachable states—and to formally

describe the upgrade functionality.

Iterated Upgrades

Finally, the specification of an upgradeable contract includes a specification of *how* upgrades are executed. In our case, the upgrade mechanism can be characterized by the `upgrade_fun` entrypoint, which can be called by any user. Importantly, through this functionality we can derive a relationship between the mechanics of calling an upgrade and the resulting contract functionality. We know that, for any reachable state of `C`, calling the `upgrade_fun` entrypoint with payload `s':N -> N` results in an embedding of a contract `C_i`, where `n_i` matches the `n` in the storage of `C`, and for which calling `next_i` results in updating `n_i` with `s_next := s'`. We can state this in a theorem:

```
Theorem upgrade_mechanism :
    forall bstate caddr cstate msg s' cstate' acts chain ctx,
    (* state is reachable *)
    reachable bstate ->
    (* the contract address is caddr with state cstate *)
    env_contracts bstate caddr = Some (C : WeakContract) ->
    contract_state bstate caddr = Some cstate ->
    (* a call to upgrade_fun was successful *)
    msg = upgrade_fun s' ->
    receive C chain ctx cstate (Some msg) = Ok(cstate', acts) ->
    (* exists a contract C_i and an embedding *)
    exists C_i (f_i : ContractMorphism C_i C),
    is_inj_cm f_i /\
    (* with storage as desired *)
    cstate' =
    (state_morph f_i) {| n := cstate'.(n) ; |}.
Proof.
    (* proof omitted *)
Qed.
```

This precisely characterizes the exact relationship between the upgrade mechanism (calling `upgrade_fun`) and the resulting contract functionality, and does so via an embedding of contracts. Furthermore, if `C_i` is the current contract functionality and we wish to upgrade to `C_i'`, we can reason about that upgrade rigorously, before deploying it, by reasoning about the corresponding embedded contracts `C_i` and `C_i'` in relation to each other using techniques from §5.3.

On the other hand, the existence of a surjection no matter the state of `C`, the proof of which is trivial because `C_s` is defined independent of the state of `C`.

```
Theorem upgradeability_invariance :
```

```
    forall bstate caddr cstate msg s' cstate' acts,
    (* state is reachable *)
    reachable bstate ->
    (* the contract address is caddr with state cstate *)
    env_contracts bstate caddr = Some (C : WeakContract) ->
    contract_state bstate caddr = Some cstate ->
    (* exists a surjection onto C_s *)
    exists (f_s : ContractMorphism C C_s).
Proof. intros. exact f_s. Qed.
```

5.4.3 Mathematical Characterization: A Digression

Given an upgradeable contract C we've demonstrated a potentially useful proof technique of isolating its immutable and mutable parts, C_s and C_i respecitvely, via an embedding $f_i : C_i \hookrightarrow C$ and a surjection $f_x : C \twoheadrightarrow C_s$. This gives us a sequence of functions:

$$C_i \hookrightarrow C \twoheadrightarrow C_s.$$

Sequences of functions of the form $A \hookrightarrow B \twoheadrightarrow C$ are common in mathematics. For example, a group G with normal subgroup $H \triangleleft G$ admits the following sequence:

$$H \hookrightarrow G \twoheadrightarrow G' \equiv G/H \tag{5.2}$$

Some sequences *split*, which means that

$$G \cong H \oplus G'$$

and $H \hookrightarrow G$ is the left inclusion of the direct sum, and $G \twoheadrightarrow G'$ is the projection onto the right direct summand. If the sequence (5.2) splits, this indicates that in some sense G can be thought of as a clean "gluing together" of H and G'. However, something like a smash product is a more interesting way to combine groups, which admits a sequence like (5.2), but not a splitting sequence.

Moving from group theory into topology, some quotients have more structure than others, for example a *fibration*, which is a surjective map with added properties. The *homotopy lifting property* states that for a fibration

$$\begin{array}{ccc} X & \hookrightarrow & E \\ & & \downarrow \\ & & B \end{array}$$

and a homotopy $X \times [0,1] \to B$ such that the image of $X \times \{0\} \subset S \times [0,1]$ in B is equal to the image of X in B, the homotopy can be lifted uniquely to a homotopy $X \times [0,1] \to E$ as follows:

$$\begin{array}{ccc} X \times \{0\} & \hookrightarrow & E \\ \downarrow & \nearrow & \downarrow \\ X \times [0,1] & \longrightarrow & B \end{array}$$

Fibrations play a key role in category theory, type theory, and theoretical computer science [40]. Like the sequence (5.2), it describes a "gluing together" of topological spaces. There has been recent work on fibrations and Hoare-like partial and total correctness assertions [195].

The sequence of contracts that we've just demonstrated doesn't seem to have as much algebraic structure as the examples we've given above (*e.g.* it is not clear if there is an analogue to the kernel of a morphism of contracts), but the idea behind this decomposition of upgradeability through morphisms is meant to be analagous. In each of these mathematical examples, a sequence

$$A \hookrightarrow B \twoheadrightarrow C$$

indicates that in some sense B is constructed as a combination of A and C—not dissimilar to how we wish to see `C` as made up of its mutable and immutable parts, `C_i` and `C_s`. If the sequence does something analogous to splitting, then the gluing together is very clean, but most often the delineation between the immutable and mutable parts of a smart contract are not at all obvious.

This type of decomposition has many applications in pure mathematics, not least of which are exact sequences in homological algebra, which are used to define the ubiquitous and highly influential homology functor [166].

5.5 Conclusion

By way of contract morphisms, in this chapter we were able to mathematically characterize contract upgradeability, a property of contracts which can lead to severe vulnerabilities if not well-understood. To do so we introduced a new theoretical tool, contract morphisms, which can be used to reason about smart contracts in relation to each other. Through various examples, we showed how morphisms can apply to formal proof and specification of smart contracts, including reasoning about an upgradeable contract `C` in relation to contracts `C_i` and `C_s`, an embedding `f_i : ContractMorphism C_i C`, and a surjection `f_s : ContractMorphism C C_s`.

Chapter 6

Equivalences of Contracts

Having treated economic and upgradeability properties of smart contracts, in our final research chapter we turn our attention to the complex behavior of a system of contracts. Part of the power of smart contracts is that they are *composable*—so much so that decentralized finance is commonly referred to as "money legos" in the wild [183], which refers to building new DeFi protocols by composing existing ones. Because contracts can interact via transactions, systems of contracts can have highly complex emergent behavior and are difficult to reason about. Furthermore, reasoning formally about systems of contracts poses challenges in itself, as we will see shortly.

We aim to introduce formal techniques to simplify the complex behavior of systems of contracts in the context of formal reasoning. Our main tool will be to rigorously develop various notions of equivalences of contracts, starting from the strict bisimulation we saw in Chapter 5, to looser notions which will allow us to reason about a system of contracts in terms of an equivalent, monolithic contract. All in all, we define three notions of equivalence: equivalences (bisimulations), weak equivalences, and multi-chain weak equivalences. These notions will help make reasoning about systems of contracts more tractable, and lay foundations for a formal, process-algebraic approach to reasoning about smart contracts in ConCert.

We proceed as follows: in §6.1, we motivate the theory of this chapter by looking closely at the difficulty of formal reasoning about systems of contracts; in §6.2, we formally introduce equivalences and weak equivalences; in §6.3, we introduce multi-chain weak equivalences; and in §6.4, we discuss applications to process-algebraic approaches of formal reasoning.

6.1 Systems of Contracts

Financial smart contracts are ubiquitously implemented as systems of contracts rather than as monolithic contracts. One reason for this is purely practical. A blockchain is a highly resource-constrained platform on which to write programs due to gas fees, which are paid by the user [209]. Depending on the contract, it can be more gas-efficient to implement modularly rather than as a monolithic contract, for example to prevent the storage of a contract from getting too large and thus expensive to access. Other reasons are more endemic. Systems of contracts show up ubiquitously in financial contracts because new applications typically build off of existing contracts [183].

We've already seen several examples of this. Dexter2, like other AMMs including structured pools from Chapter 4, is implemented as a main, trading contract, and a secondary token contract for the LP token [148]. The functionality of the main depends on that of the token contract, and conversely the token contract depends on the main trading contract. Furthermore, Dexter2 and AMMs like Uniswap and Curve facilitate trading between tokens, each of which has its own tokenomics which are managed independently of the AMM. For example, governance tokens can be traded on AMMs and used to govern upgrades of an upgradeable contract. As we saw in the Beanstalk attack (§2.1.1), this exposed a vulnerability to a flash loan attack (again using another contract) to gain a majority governance vote and drain the Beanstalk contract's funds.

The examples continue. Yield aggregators like Harvest or Yearn [79] optimize over a set of existing DeFi protocols so that users can maximize yield farming. DEX aggregators like 1inch or Matcha minimize trading fees over various DEXes. Synthetics, including stablecoins like DAI, rely on price oracles to manage their tokenomics and maintain their peg [182]. Finally, financial contracts that use cross-chain bridges, such as cross-chain swaps, are systems of contracts over multiple chains and have received a lot of attention in the literature [150, 63, 132, 109]. In each of these examples, contract security and even correctness depends intimately on the behavior of other contracts [183].

This inevitably forces the specification of a financial contract to include interactions with other contracts. We saw this in the Arbitrage Sensitivity property (Property 5) of the metaspecification in §**??**, and in the fact that the structured pools specification assumes an abstract FA2 contract. We also saw this in Chapter 5, where contract upgrades can be governed by token holders, meaning that a contract's upgrade functionality effectively relies on the balances kept in the ledger of another contract.

Reasoning about systems of contracts in a formal setting is thus inevitable if we wish to be fully rigorous.

6.1.1 Challenges to Formal Reasoning About Systems of Contracts

Reasoning about the behavior of a system of smart contracts is highly complex, and largely not done with full rigor in proof-assistant-based verification. Most of the previous work to verify systems of contracts has been limited to properties involving incoming or outgoing traffic from arbitrary contracts, and have focused on the perspective of one contract. As noted by Nielsen *et al.*, this is not enough to fully prove safety of interacting contracts [148].

Take again the example of verification work done on Dexter2. Neither Mi-Cho-Coq nor K Framework formalize the execution model of a blockchain, and so cannot reason about the semantics of a blockchain executing with multiple contracts like ConCert can. The formalization in K Framework includes specific caveats in the report on the assumptions about contract execution semantics and external contracts [20]. The list is fairly extensive and not formally proved to be correct. The formalization in Mi-Cho-Coq does include an accompanying FA2 contract [8], but is unable to reason rigorously the contracts' interaction. Instead, both focus on verification from the perspective of the main Dexter2 contract and arbitrary incoming messages.

Because ConCert formalizes the execution model of a blockchain, it is possible to reason rigorously about systems of contracts using general induction over chain traces. However, proving a multi-contract invariant explodes the number of cases needed to check because one needs induct over each contract, and reason about all possible transactions in which they could interact. By design, `contract_induction` simplifies reasoning about one contract and is not sufficiently general to reason about multiple contracts. To simplify this process, Nielsen *et al.* introduce a new proof technique which allows one to reason about multi-contract invariants by reasoning from each contract's perspective individually, and then composing them. The composition is facilitated by a theorem which shows that the "outgoing" perspective of one contract is equivalent to the "incoming" perspective of the other [148].

6.1.2 An Approach With the Notion of Equivalence

This new technique is effective for reasoning about multi-contract invariants, but as before we are interested in reasoning abstractly about the correctness of a specification. Like we saw in the Diamond upgradeability standard in §5.1.1, a specification of a system of contracts is a specification of each contract, their interfaces, and how they interact with each other—but it is not always obvious from the specification what one wishes the emergent behavior of the system to be, or indeed whether the specification actually captures that behavior. The tools we have thus far developed are for individual contracts, so if possible we would like to reason about the emergent behavior of a system of contracts in terms of one single, monolithic contract which is equivalent to the system.

To do so, we explore in depth various notions of *equivalence* of smart contracts. The motivations for this are twofold. First, we wish to understand when contract optimizations are property-preserving. As we mentioned in Example 5.3.4, a contract that is optimized for performance is not always optimized for formal reasoning and intelligibility. We mentioned this in the context of gas-efficiency, but the same applies to contracts which could in principle be implemented as monolithic contracts are implemented as a system of contracts for convenience or efficiency. Doing so could make them more difficult to reason about formally—but if we could reason instead about a single contract which represents the "monolithic version" of the contract, in principle we could transport those properties over an equivalence with the "modular version" of the contract. Intuitively, we know what the distinction between a modular and monolithic version of the same contract means, and that if done right from the perspective of the blockchain the two are equivalent. We would like to make this formal.

The second is to enable a process-algebraic approach to formal reasoning about systems of contracts, in particular financial contracts. There are emerging efforts to understand DeFi from a process-algebraic perspective, *e.g.* Tolmach *et al.*'s work to formally analyze composable DeFi protocls [183], and Bartoletti *et al.*'s work to formally understand DeFi through a theory of DeFi [47, 48]. Neither of these models have been formalized, but in principle could be formalized into ConCert if we have a thorough understanding of equivalence on the blockchain. We are interested in enabling these efforts in ConCert, but any process algebra must rely on a concrete notion of what it means for two processes to be equivalent.

6.1.3 Related Work

We are not aware of any embedded process-algebraic semantics in theorem-prover based formal verification of smart contracts. The only efforts in this direction we are aware of are building *Multi*, a formal playground to model interacting contracts which is in progress at the time of writing [71]. At the time of writing, this project is not yet completed. While their ArXiv paper discusses the need to formalize blockchain semantics, it is not obvious how they plan to do so.

Our approach of formalizing the notion of equivalence is similar recent work in Coq for machine-checked state-separating cryptographic proofs, in particular SSProve [23, 65], which combines high-level modular proofs about composed protocols. Our work is analogous, except rather than cryptographic proofs we focus on formal proofs about smart contracts.

6.2 Equivalences and Weak Equivalences

In Chapter 5, two contracts are considered equivalent if there is an isomorphism, or bisimulation, of contracts. But as we mentioned in Example 5.3.4, isomorphic contracts are not identical. For one, they can vary in gas efficiency. The degree to which two contracts are equivalent depends on what is true of both contracts by way of the isomorphism. We did not make any specific claims about this in Chapter 5, but it can discovered empirically by using the proof techniques of §5.3.

There are other ways that contracts can be equivalent. Suppose we have a contract `C` with several entrypoints; `C` forwards all entrypoint calls to a contract `C'` which has identical entrypoints but can only receive messages from `C`. If a call to `C'` results in an emitted transaction, that goes to `C` which forwards it to the appropriate address. We can clearly see that `C` is just an interface for `C'`, and that we could have simply implemented `C` and `C'` as a standalone contract `C''` that doesn't separate the interface from the contract functionality. Of course, the contracts `C` and `C'` taken together are not identical to `C''`. Nor are they isomorphic. But from the perspective of the blockchain's execution semantics we can say intuitively that they are equivalent in some sense. We will call this notion of equivalence a *weak equivalence* of contracts, which indicates that there is a bisimulation of chains in which the weakly equivalent contracts are interchangeable.

Going even further, since ConCert can formally model multiple concurrent chains (and extract executable code to multiple chains), we might consider two contracts that communicate over a cross-chain bridge and form a system of contracts. In ConCert, a blockchain is defined by the `ChainState` type which can be acted on by something of type `ChainStep`. As we will see, we can model concurrent chains with a *product of chains*, and model such a system of contracts as a single, monolithic contract. These contracts will be called *multi-chain weakly equivalent*, which induce bisimulations of concurrent chains.

In this section we will treat equivalences and weak equivalences, and we will treat multi-chain weak equivalences in §6.3. We first we need to formalize the notion of a bisimulation of chains, which we again do by defining morphisms.

6.2.1 Morphisms of Chains

In ConCert, the type `ChainState` extends the `Environment` type, `chain_state_env :> Environment`, with a queue of transactions `chain_state_queue : list Action`. We can progress through chain states by the inductive type `ChainStep prev_bstate next_bstate`, which is parameterized by two inhabitants of `ChainState`, and indicates that `next_bstate` is reachable through `prev_bstate` by either adding a block, executing an action, failing to execute an invalid action, or permuting the queue. Intimately related to the `ChainStep` type is the `ChainTrace` type, which is simply a chained list of inhabitants of `ChainState`,

connected by inhabitants of `ChainTrace`. A chained list is defined in ConCert as follows:

```
Context {Point : Type} {Link : Point -> Point -> Type}.

Inductive ChainedList : Point -> Point -> Type :=
  | clnil : forall {p}, ChainedList p p
  | snoc : forall {from mid to},
      ChainedList from mid -> Link mid to -> ChainedList from to.
```

A morphism of chains, then, is a function `chainstate_morph : ChainState -> ChainState` which respects the algebraic relation of `ChainState` induced by `ChainStep`.

```
Record ChainMorphism := build_chain_morphism {
    chainstate_morph : ChainState -> ChainState ;
    empty_fixpoint : chainstate_morph empty_state = empty_state ;
    chainstep_morph :
        forall bstate1 bstate2,
        ChainStep bstate1 bstate2 ->
        ChainStep (chainstate_morph bstate1) (chainstate_morph bstate2) ;
}.
```

Listing 6.1: A morphism of chains.

A morphism of chains is an endomorphism on `ChainState` which takes a chain trace to a chain trace. The empty state `empty_state` is a fixed point, so in particular it takes reachable states to reachable states.

```
Lemma reachable_to_reachable : forall bstate (chain_morph : ChainMorphism),
    reachable bstate -> reachable (chainstate_morph chain_morph bstate).
Proof.
    intros.
    unfold reachable in H. destruct H. unfold reachable.
    pose proof (chm_trace_to_trace empty_state bstate X chm) as X'.
    rewrite (empty_fixpoint chm) in X'.
    constructor.
    exact X'.
Qed.
```

As with morphisms of contracts, morphisms of chains can be composed and composition is associative.

```
(* an auxiliary lemma *)
Lemma compose_fixpoint_result (chm1 chm2 : ChainMorphism) :
    compose (chainstate_morph chm2) (chainstate_morph chm1) empty_state =
    empty_state.
Proof.
    unfold compose.
```

```
    rewrite (empty_fixpoint chm1).
    rewrite (empty_fixpoint chm2).
    reflexivity.
Qed.

(* composition of chain morphisms *)
Definition composition_chm (chm2 chm1 : ChainMorphism) : ChainMorphism := {|
    chainstate_morph := compose (chainstate_morph chm2) (chainstate_morph chm1) ;
    empty_fixpoint   := compose_fixpoint_result (chm1) (chm2) ;
    chainstep_morph  := (fun b1 b2 =>
        compose
            (chainstep_morph chm2
                (chainstate_morph chm1 b1)
                (chainstate_morph chm2 b2))
            (chainstep_morph chm1 b1 b2)) ;
|}.

(* composition is associative *)
Lemma composition_assoc_chm : forall chm1 chm2 chm3,
    composition_chm (composition_chm chm3 chm2) chm1 =
    composition_chm chm3 (composition_chm chm2 chm1).
Proof.
    intros. unfold composition_chm.
    f_equal. apply proof_irrelevance.
Qed.
```

We also have an identity morphism, which is trivial under composition.

```
(* an auxiliary lemma *)
Lemma empty_fixpoint_id :
    @id ChainState empty_state = empty_state.
Proof. auto. Qed.

(* the identity chain morphism *)
Definition id_chm : ChainMorphism := {|
    chainstate_morph := @id ChainState ;
    empty_fixpoint := empty_fixpoint_id ;
    chainstep_morph := (fun b1 b2 => @id (ChainStep b1 b2)) ;
|}.

(* left and right composition results *)
Lemma composition_chm_left : forall chm, composition_chm id_chm chm = chm.
Proof.
    intro. unfold composition_chm.
    destruct chm. f_equal. apply proof_irrelevance.
```

```
Qed.

Lemma composition_chm_right : forall chm, composition_chm chm id_chm = chm.
Proof.
    intro. unfold composition_chm.
    destruct chm. f_equal. apply proof_irrelevance.
Qed.
```

Finally, an isomorphism is defined analogously to isomorphisms of contracts, as a bisimulation of chains.

```
(* chain isomoprhisms, which are bisimulations of blockchains *)
Definition is_iso_chm (f g : ChainMorphism) : Prop :=
    composition_chm g f = id_chm /\
    composition_chm f g = id_chm.
```

We argue that this is the correct notion for a bisimulation of chains because a morphism of chains preserves the algebraic structure of moving between chain states via chain steps, preserving key properties like reachability.

Example 6.2.1 (Null Chain Morphism)**.** Like in the case of contract morphisms, we have a canonical null morphism, which sends all chain states to the emtpy state `empty_state` and all chain steps to the trivial permutation of the empty queue of `empty_state`.

```
(* some auxiliary lemmas *)
Lemma empty_self_equiv : EnvironmentEquiv empty_state empty_state.
Proof. unfold EnvironmentEquiv. auto. Qed.

(* function definitions *)
Definition chainstate_morph (_ : ChainState) : ChainState := empty_state.

Lemma empty_fixpoint : chainstate_morph empty_state = empty_state.
Proof. unfold chainstate_morph. auto. Qed.

chainstep_morph (bstate1 bstate2 : ChainState) (_ : ChainStep bstate1 bstate2) : ChainStep
    emtpy_state empty_state :=
    step_permute empty_self_equiv id_permute perm_nil.

(* definition of morphism of chains *)
Definition null_morphism : ChainMorphism :=
    build_chain_morphism chainstate_morph empty_fixpoint chainstep_morph.
```

6.2.2 Simple Morphisms of Contracts Lift to Morphisms of Chains

A morphism of contracts `f:ContractMorphism C C'` takes an execution trace of `C` and sends to one of `C'`. If `f` is simple, then we can isolate a morphism of contract storage, setup, and even emitted actions. In principle, a morphism like `f` could induce a morphism of chains which replaces the contract `C` for `C'`. In fact, this is the case for simple morphisms.

```
Theorem cm_lift :
    forall bstate caddr,
    (* bstate is reachable with a trace *)
    reachable bstate,
    (* the contract address is caddr with state cstate *)
    env_contracts bstate caddr = Some (C : WeakContract) ->
    contract_state bstate caddr = Some cstate ->
    (* there is a contract morphism f : C -> C' *)
    (f : ContractMorphism C C') ->
    (* then there exists a reachable bstate' *)
    exists bstate' (f : ChainMorphism),
    f bstate = bstate' /\
    (* bstate' is reachable *)
    reachable bstate' /\
    (* the contract address is caddr with state cstate' *)
    env_contracts bstate' caddr = Some (C' : WeakContract) /\
    contract_state bstate' caddr = Some (state_morph f cstate).
Proof. (* proof omitted *) Qed.
```

The proof of `cm_lift` relies on a function

```
cm_lift_morph : ContractMorphism C C' -> ChainMorphism.
```

Importantly, compositions of morphisms lift to compositions of morphisms, and the identity lifts to the identity.

```
Lemma composition_cm_to_composition_chm :
    forall (f : ContractMorphism C C') (g : ContractMorphism C' C''),
    cm_lift_morph (composition_cm g f) =
    composition_chm (cm_lift_morph g) (cm_lift_morph f).
Proof. (* proof omitted *) Qed.
```

```
Lemma id_cm_to_id_chm :
    forall C,
    cm_lift_morph (id_cm C) = id_chm.
Proof. (* proof omitted *) Qed.
```

In particular, a bisimulation of contracts lifts to a bisimulation of chains.

```
Corollary iso_cm_to_iso_chm :
    forall (f : ContractMorphism C C') (g : ContractMorphism C' C''),
    is_iso_cm f g ->
    is_iso_chm (cm_lift_morph f) (cm_lift_morph g).
Proof. (* proof omitted *) Qed.
```

6.2.3 Equivalences of Contracts

Our first notion of equivalence is the strong notion we defined as an isomorphism of contracts in Chapter 5.

Definition 6.2.1 (Equivalence of Contracts)**.** Contracts `C` and `C'` are *equivalent* if there are morphisms `f : ContractMorphism C C'` and `g : ContractMorphism C' C` such that `is_iso f g`.

```
Definition are_equiv
    (C : Contract Setup State Msg Error)
    (C' : Contract Setup' State' Msg' Error') : Prop :=
    exists (f : ContractMorphism C C') (g : ContractMorphism C' C),
    is_iso_cm f g.
```

The notion of equivalence is justified, at least for simple morphisms, because equivalent morphisms lift to a bisimulation of chains. Semantically, the only real difference between equivalent contracts is a renaming of contract entrypoints, storage, and types because they induce a bisimulation of blockchains.

Importantly, equivalences of contracts form an equivalence relation on contracts.

```
(* reflexivity *)
Lemma are_equiv_refl : forall C,
    are_equiv C C.
Proof. (* proof omitted *) Qed.

(* symmetry *)
Lemma are_equiv_sym : forall C C',
    are_equiv C C' -> are_equiv C' C.
Proof. (* proof omitted *) Qed.

(* transitivity *)
Lemma are_equiv_trans : forall C C' C'',
    are_equiv C C' -> are_equiv C' C'' -> are_equiv C C''.
Proof. (* proof omitted *) Qed.
```

6.2.4 Weak Equivalences of Contracts

We now define a weaker notion of equivalence which only requires the existence of a bisimulation of blockchains.

Definition 6.2.2 (Weak Equivalence of Contracts)**.** Contracts `C` and `C'` are *weakly equivalent* if there is an isomorphism of chains `f,g:ChainMorphism` with `H:is_iso f g` such that for all blockchain states `bstate:ChainState` with `C` deployed at an address `caddr`, `f bstate:ChainState` has `C'` deployed at `caddr`. Conversely, for all `bstate':ChainState` with `C'` deployed at some `caddr'`, we have `C` deployed at `caddr'` in `f bstate':ChainState`.

```
Definition are_weakly_equiv
    (C : Contract Setup State Msg Error)
    (C' : Contract Setup' State' Msg' Error') : Prop :=
    exists (f, g : ChainMorphism),
    (env_contracts bstate  caddr  = Some C  -> env_contracts (f bstate)  caddr  = Some C
    ') /\
    (env_contracts bstate' caddr' = Some C' -> env_contracts (g bstate') caddr' = Some C)
    .
```

As before with equivalences, weak equivalences give us an equivalence relation.

```
(* reflexivity *)
Lemma are_weakly_equiv_refl : forall C,
    are_weakly_equiv C C.
Proof. (* proof omitted *) Qed.

(* symmetry *)
Lemma are_weakly_equiv_sym : forall C C',
    are_weakly_equiv C C' -> are_weakly_equiv C' C.
Proof. (* proof omitted *) Qed.

(* transitivity *)
Lemma are_weakly_equiv_trans : forall C C' C',
are_weakly_equiv C C' -> are_weakly_equiv C' C'' -> are_weakly_equiv C C''.
Proof. (* proof omitted *) Qed.
```

This will be the sense in which we called the contract `C'` and its interface `C` "equivalent" to `C''` at the beginning of this section. However, it is not immediately obvious how to do this because a bisimulation of chains requires a bijection on contract address type `Address`, and if we wish to formalize the intuitive notion that `C` and `C'` together are somehow equivalent to `C''`, we need a way to bijectively map `C` and `C'` onto `C''`. `Address` is an arbitrary countable serializable type with decidable equality and a distinction between

contract and wallet addresses, so this would be impossible if *e.g.* `Address` is finite.

Disjoint Sums of Contracts

To remedy this issue we introduce the notion of a *disjoint sum of contracts*, which models two concurrent contracts, compressed to operate disjointly with a shared address. This allows us to consider a

Definition 6.2.3. A *disjoint sum* of contracts `C : Contract Setup Msg State Error` and `C' : Contract Setup' Msg' State' Error'`, written `C ⊔ C'`, is a contract of types `(Setup + Setup')`, `(Msg + Msg')`, `(State + State')`, and `(Error + Error' + unit)`, which disjointly combines the functionality of `C` and `C'` by executing `C` when `C`'s entrypoint is called, and executing `C'` when `C'`'s entrypoint is called.

```
Definition init_sum
    (init  : Chain -> ContractCallContext -> Setup  -> result State Error)
    (init' : Chain -> ContractCallContext -> Setup' -> result State' Error') :
    (Chain -> ContractCallContext -> Setup + Setup' -> result (State + State') (Error +
    Error' + unit)) :=
    (fun (c : Chain) (ctx : ContractCallContext) (x : Setup + Setup') =>
        match x with
        | inl s =>
            match init c ctx s with
            | Ok s' => Ok (inl s')
            | Err e => Err (inl (inl e))
            end
        | inr s =>
            match init' c ctx s with
            | Ok s' => Ok (inr s')
            | Err e => Err (inl (inr e))
            end
        end).

Definition recv_sum
    (recv  : Chain -> ContractCallContext -> State  -> option Msg  -> result (State  *
    list ActionBody) Error)
    (recv' : Chain -> ContractCallContext -> State' -> option Msg' -> result (State' *
    list ActionBody) Error') :
    Chain -> ContractCallContext -> State + State' -> option (Msg + Msg') -> result ((
    State + State') * list ActionBody) (Error + Error' + unit) :=
    (fun (c : Chain) (ctx : ContractCallContext) (st : State + State') (op_msg : option (
    Msg + Msg')) =>
        match st with
        | inl s =>
```

```
            match op_msg with
            | Some msg =>
                match msg with
                | inl m =>
                    match recv c ctx s (Some m) with
                    | Ok (s', actns) => Ok (inl s', actns)
                    | Err e => Err (inl (inl e))
                    end
                (* fails if state/msg don't pertain to the same contract *)
                | inr m => Err (inr tt)
                end
            | None =>
                match recv c ctx s None with
                | Ok (s', actns) => Ok (inl s', actns)
                | Err e => Err (inl (inl e))
                end
        end
        | inr s =>
            match op_msg with
            | Some msg =>
                match msg with
                | inr m =>
                    match recv' c ctx s (Some m) with
                    | Ok (s', actns) => Ok (inr s', actns)
                    | Err e => Err (inl (inr e))
                    end
                (* fails if state/msg don't pertain to the same contract *)
                | inl m => Err (inr tt)
                end
            | None =>
                match recv' c ctx s None with
                | Ok (s', actns) => Ok (inr s', actns)
                | Err e => Err (inl (inr e))
                end
            end
        end).

Definition sum_contract
    (C : Contract Setup Msg State Error)
    (C' : Contract Setup' Msg' State' Error') :
    Contract (Setup + Setup') (Msg + Msg') (State + State') (Error + Error' + unit) :=
    build_contract
        (init_sum C.(init) C'.(init))
        (recv_sum C.(receive) C'.(receive)).
```

The direct sum operation `sum_contract` is associative and commutative.

```
Lemma sum_contract_associative
    { C : Contract Setup Msg State Error }
    { C' : Contract Setup' Msg' State' Error' }
    { C'' : Contract Setup'' Msg'' State'' Error'' } :
    are_equiv
        (sum_contract C (sum_contract C' C''))
        (sum_contract (sum_contract C C') C'').
Proof. (* proof omitted *) Qed.

Lemma sum_contract_commutative
    { C : Contract Setup Msg State Error }
    { C' : Contract Setup' Msg' State' Error' } :
    are_equiv
        (sum_contract C C')
        (sum_contract C' C).
Proof. (* proof omitted *) Qed.
```

A disjoint sum of contracts lets us model a deployment of two distinct contracts at one single address. This will let us now talk of weak equivalences between systems of contracts.

Definition 6.2.4 (Weakly Equivalent Systems of Contracts)**.** Two systems of contracts `C_1, C_2, ... , C_n` and `C'_1, C'_2, ... , C'_n'` are weakly equivalent if the corresponding sum contracts `C_1 ⊔ C_2 ⊔ ... ⊔ C_n` and `C'_1 ⊔ C'_2 ⊔ ... ⊔ C'_n'` are weakly equivalent.

```
(* a left fold over a list of contracts which sums them *)
Definition foldl_sum_contract (L : list Contract) : Contract :=
    foldl sum_contract null_contract L.

(* weakly equivalent systems of contracts *)
Definition are_weakly_equiv_system (L1 L2 : list Contract) : Prop :=
    are_weakly_equiv (foldl_sum_contract L1) (foldl_sum_contract L2).
```

6.2.5 Examples of Weakly Equivalent Systems of Contracts

Example 6.2.2 (Monoid of Contracts)**.** In the formal definition Definition 6.2.4 we used the null contract as an accumulator to fold over for a sum, so the formal definition departs slightly from the informal statement. However, for any contract `C`, summing with the null contract results in a weak equivalence.

```
Theorem null_contract_monoid_identity :
    forall (C : Contract Setup Msg State Error),
    are_weakly_equiv C (sum_contract C null_contract) /\
```

```
    are_weakly_equiv C (sum_contract null_contract C).
Proof. (* proof omitted *) Qed.
```

Since disjoint sums are associative and commutative, contracts form a monoid under the operation $\sqcup$, or `sum_contract`, using weak equivalences as the notion of equivalence. The null contract `null_contract` is the identity element of the monoid.

Example 6.2.3 (Basic System of Contracts)**.** In this example, we give a basic system of contracts like the contracts `C`, `C'`, and `C''` we discussed at the beginning of this chapter. We will first define a pair of contracts, `C` and `C'`, which form a simple system of interacting contracts. We then define a contract `C''` which is the "monolithic version" of the "modular version" of `C` and `C'`. We then show that `C ⊔ C'` and `C''` are weakly equivalent.

```
(* contract types definition *)
Definition setup := Address.
Inductive  entrypoint := | incr (n : N) | decr (n : N) | reset.
Definition storage := Address.
Definition error := N.
Definition result : Type := ResultMonad.result (storage * list ActionBody) error.

(* init function definition *)
Definition init (_ : Chain)
                (_ : ContractCallContext)
                (storage_contract : setup)
                : ResultMonad.result storage N :=
    Ok (storage_contract).

(* receive function definition *)
Definition receive (_ : Chain)
                (_ : ContractCallContext)
                (storage_contract : storage)
                (msg : option entrypoint)
                : result :=
    match msg with
    | Some (incr n) =>
        let act_fwd := act_call storage_contract 0 (serialize (incr' n)) in
        Ok (storage_contract, [ act_fwd ; ])
    | Some (decr n) =>
        let act_fwd := act_call storage_contract 0 (serialize (decr' n)) in
        Ok (storage_contract, [ act_fwd ; ])
    | Some reset =>
        let act_fwd := act_call storage_contract 0 (serialize (reset')) in
        Ok (storage_contract, [ act_fwd ; ])
    | None => Err 0
```

```
32      end.
33
34  (* construct the contract *)
35  Definition C : Contract setup entrypoint storage error :=
36      build_contract init receive.
37
38
39  (* contract types definition *)
40  Definition setup' := Address.
41  Inductive  entrypoint' := | incr' (n : N) | decr' (n : N) | reset'.
42  Definition storage' := { interface_address : Address ; counter : N ; }.
43  Definition error' := N.
44  Definition result' : Type := ResultMonad.result (storage' * list ActionBody) error'.
45
46  (* init function definition for C' *)
47  Definition init' (_ : Chain)
48                   (_ : ContractCallContext)
49                   (interface_address : setup')
50                   : ResultMonad.result storage' error' :=
51      let init_storage := {|
52          interface_address := interface_address ;
53          counter := 0 ;
54      |} in
55      Ok (init_storage).
56
57  (* receive function definition for C' *)
58  Definition receive' (_ : Chain)
59                   (_ : ContractCallContext)
60                   (st : storage')
61                   (msg : option entrypoint')
62                   : result :=
63      (* check the call comes from the interface contract *)
64      if ctx.(ctx_from) <> st.(interface_address) then Err 0 else
65      (* call the  *)
66      match msg with
67      | Some (incr' n) =>
68          let st' := {|
69              interface_address := st.(interface_address) ;
70              counter := st.(counter) + 1 ;
71          |} in
72          Ok (st', [])
73      | Some (decr' n) =>
74          let st' := {|
75              interface_address := st.(interface_address) ;
76              counter := st.(counter) - 1 ;
```

```
        |} in
        Ok (st', [])
    | Some reset' =>
        let st' := {|
            interface_address := st.(interface_address) ;
            counter := 0 ;
        |} in
        Ok (st', [])
    | None => Err 0
    end.

(* construct the contract *)
Definition C' : Contract setup' entrypoint' storage' error' :=
    build_contract init' receive'.
```

Listing 6.2: Definition of `C` and `C'`

```
(* contract types definition for C'' *)
Definition setup'' := Address.
Inductive  entrypoint'' := | incr'' (n : N) | decr'' (n : N) | reset''.
Definition storage'' := { interface_address : Address ; counter : N ; }.
Definition error'' := N.
Definition result'' : Type := ResultMonad.result (storage'' * list ActionBody) error''.

(* init function definition for C'' *)
Definition init'' (_ : Chain)
                  (_ : ContractCallContext)
                  (interface_address : setup'')
                  : ResultMonad.result storage'' error'' :=
    let init_storage := {|
        interface_address := interface_address ;
        counter := 0 ;
    |} in
    Ok (init_storage).

(* receive function definition for C' *)
Definition receive'' (_ : Chain)
                  (_ : ContractCallContext)
                  (st : storage'')
                  (msg : option entrypoint'')
                  : result :=
    match msg with
    | Some (incr'' n) =>
        let st'' := {|
            interface_address := st.(interface_address) ;
            counter := st.(counter) + 1 ;
```

```
        |} in
        Ok (st'', [])
    | Some (decr'' n) =>
        let st'' := {|
            interface_address := st.(interface_address) ;
            counter := st.(counter) - 1 ;
        |} in
        Ok (st'', [])
    | Some reset'' =>
        let st'' := {|
            interface_address := st.(interface_address) ;
            counter := 0 ;
        |} in
        Ok (st'', [])
    | None => Err 0
    end.

(* construct the contract *)
Definition C' : Contract setup' entrypoint' storage' error' :=
    build_contract init' receive'.
```

Listing 6.3: Definition of the "monolithic version" `C''`

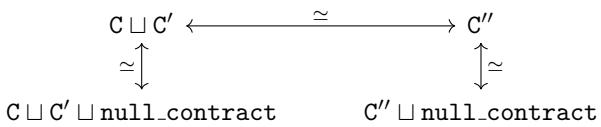
To prove weak equivalence, we prove both of the following equivalent results.

```
Theorem are_weakly_equiv (sum_contract C C') C''.
Proof. (* proof omitted *) Qed.

Theorem are_weakly_equiv_system [ C ; C' ; ] [ C'' ; ].
Proof. (* proof omitted *) Qed.
```

Listing 6.4: Weak equivalence between `C ⊔ C'` and `C''`.

There are two proofs for `are_weakly_equiv_system`, one which we gave, and one which leverages the fact that the null contract is the identity in the monoid of contracts under the operation $\sqcup$ and weak equivalence, and that weak equivalences compose.



6.2.6 Applications to Contract Optimization

In principle one immediate application for these results is similar to one we posited for equivalences: A team wishing to verify a system of contracts could formally reason about a monolithic implementation rather than the system, and then prove an equivalence of the system and the monolithic contract. This may be more practical than it sounds, as it may make sense to develop a contract first as a monolithic contract, and then split it up into modules for various optimizations. Using weak equivalences, this process of optimization could be proven to be property-preserving by showing the bisimulation of chains associated to the weak equivalence.

Certainly, if one is able to construct the corresponding monolithic contract, verifying a large system of contracts can be simplified by compressing it to one single contract, even if one uses the theorem from [148]. This is because one can reason about multi-contract invariants in terms of one contract, rather than from each contract individually.

6.3 Multi-chain Systems of Contracts

We now turn our attention to systems of contracts which span multiple chains. Because ConCert models the execution semantics of a blockchain, we can model mutliple blockchains executing concurrently. Furthermore, because contract code in ConCert can be extracted to executable code for multiple chains, under some assumptions about a cross-chain bridge we can specify, verify, and deploy a multi-chain system of contracts in ConCert.

6.3.1 Motivation

Multi-chain systems of contracts are prolific and growing. For example, both the Ethereum and Celo blockchains feature Toucan protocol smart contracts[1]. Uniswap has deployed contracts on Polygon, Arbitrum, Optimism, and Ethereum mainnet[2]. Tether, the largest stablecoin by market cap, is deployed on Ethereum, Avalanche, Polygon, Tron, Algorand, Solana, Tezos, and others[3]. Furthermore, there is an extensive list of tokens, including NFTs, which can be transferred between chains through a process called *wrapping* [69].

To varying degrees, each of these multi-chain systems of contracts communicates over a cross-chain bridge, which have themselves been subject to multiple attacks and are an active area of research. We already

[1] https://blog.toucan.earth/toucans-carbon-ecosystem-celo/
[2] https://blog.uniswap.org/multichain-uniswap
[3] https://tether.to/en/supported-protocols/

saw Nomad, a cross-chain bridge which was exploited for $190 million due to an unsafe upgrade (§2.1.2). Wormhole, a cross-chain bridge between Ethereum and Solana, was attacked in February 2022, where hackers made off with about $320 million USD[4]. Chain interoperability protocols and cross-chain bridges are prolific and include Interledger [5], IBC protocol [6], ICON[7], the Binance bridge[8], the Celer cBridge[9], the Cross-Chain Bridge[10], the Umbria Narni Bridge[11], the Multichain bridge[12], and London bridge [13].

From a computer scientific perspective, chain interoperability essentially amounts to a problem of extending state machine replication past a single blockchain, verified through consensus of distinct pools of validators [198]. Specialist protocols have been developed for atomic cross-chain swaps [109], token transfers [64], and to support more general cross-chain transactions [163, 208, 174, 172].

We seek to specify and verify multi-chain systems of contracts, and in this work forego the issue of specifying and verifying bridges themselves. Instead, we will assume the existence of a cross-chain messaging protocol which allows a contract on one chain to call a contract on another, but without the atomicity guarantee one gets on a single chain. That is, if a cross-chain call to a contract fails, the transaction that made the call may still succeed. We assert that this is a reasonable assumption, supported by recent work on generic cross-chain transactions such as [163, 208, 174, 172].

6.3.2 Formal Multi-Chain Model in ConCert

We model two blockchains, `Chain1` and `Chain2`, executing concurrently with the type `MultiChainState`, which consists of two concurrent blockchain states. `MultiChainState` can be acted on by `MultiChainStep`, which advances the state of `Chain1` only, `Chain2` only, or both concurrently. We can reason about a combined trace `MultiChainTrace`, which is chained list of `MultiChainState` and `MultiChainStep`.

In the ConCert code, we assume two separate inhabitants of `ChainBase`. This is normally abstracted over when verifying a contract on one single chain, but as we look to support multiple chains we cannot abstract over them.

```
1 (* we assume two separate inhabitants of ChainBase, one for each modelled chain *)
2 Context { Base1 Base2 : ChainBase }.
3
```

[4]https://www.investopedia.com/crypto-theft-of-usd320-million-wormhole-hack-5218062
[5]https://interledger.org/developer-tools/get-started/overview/
[6]https://ibcprotocol.org/
[7]https://iconographicx.medium.com/
[8]https://www.bnbchain.world/en/bridge
[9]https://cbridge.celer.network/
[10]https://www.crosschainbridge.org/
[11]https://bridge.umbria.network/
[12]https://multichain.org/
[13]https://londonbridge.io/

```
(* some auxiliary definitions *)
Definition ChainState1 := @ChainState Base1.
Definition ChainState2 := @ChainState Base2.
Definition ChainStep1  := @ChainStep Base1.
Definition ChainStep2  := @ChainStep Base2.

(* a model of two executing chains Chain1 and Chain2 *)
Definition MultiChainState : Type := ChainState1 * ChainState2.
Definition MultiChainStep
    (prev_multi_bstate : MultiChainState) (next_multi_bstate : MultiChainState) : Type :=
        ChainStep1 (fst prev_multi_bstate) (fst next_multi_bstate) +
        ChainStep2 (snd prev_multi_bstate) (snd next_multi_bstate) +
       (ChainStep1 (fst prev_multi_bstate) (fst next_multi_bstate) *
        ChainStep2 (snd prev_multi_bstate) (snd next_multi_bstate)).
Definition MultiChainTrace := ChainedList MultiChainState MultiChainStep.
```

Listing 6.5: The Definition of Two Concurrent Executing Chains

A contract on `Chain1` can emit a transaction targetted at a contract or wallet on `Chain1` or `Chain2`. In this way we abstract away the specific bridging mechanism, assuming that there is some generic one that allows contracts to communicate cross-bridge. In this model, transaction atomicity only depends on transactions targetted at the same chain as the emitting contract—if a cross-chain call fails, but the the emitted transactions succeed on the local chain, then the resulting inhabitant of `MultiChainStep prev_multi_bstate next_multi_bstate` will be a success on the local chain and a failed transaction on all the cross-chain calls. Since many cross-chain bridges include elaborate timelock protocols, we note that this may be a weakness of our current approach.

6.3.3 Multi-Chain Contracts

We have a notion of a contract on a single chain, but here we will define multi-chain contracts.

Definition 6.3.1 (Multi-Chain Contract)**.** A *multi-chain contract* `S` consists of a contract on `Chain1` and one on `Chain2`.

```
Definition MContract (B1 B2 : ChainBase) := @Contract B1 * @Contract B2.
```

Example 6.3.1 (Multi-Chain Null Contract)**.** For example, we have the *multi-chain null contract* which is a null contract on each chain.

```
Definition multichain_null_contract : MContract Base1 Base2 :=
    (@null_contract Base1, @null_contract Base2).
```

Finally, we can define a multi-chain system of contracts.

Definition 6.3.2. A *multi-chain contract* consists of a list of contracts on `Chain1` and a list of contracts on `Chain2`.

```
Definition Sys_MContract (B1 B2 : ChainBase) :=
    (list @Contract B1) * (list @Contract B2).
```

6.3.4 Multi-Chain Morphisms

As before, multi-chain weak equivalence is defined in terms of a bisimulation of multiple chains executing concurrently. To codify this, we need a multi-chain morphism. Before doing so, we define two auxiliary notions: the multi-chain empty state, and multi-chain reachability.

```
Definition multichain_empty_state := (empty_state, empty_state).

Definition multichain_reachable (mstate : MultiChainState) : Prop :=
    reachable (fst mstate) /\ reachable (snd mstate).
```

Now for our definition of multi-chain morphisms.

```
(* A multi-chain morphism is a morphism of linked lists *)
Record MultiChainMorphism := build_multi_chain_morphism {
    multichainstate_morph : MultiChainState -> MultiChainState ;
    multi_empty_fixpoint :
        multichainstate_morph multichain_empty_state = multichain_empty_state ;
    multichainstep_morph :
        forall mstate1 mstate2,
        MultiChainStep mstate1 mstate2 ->
        MultiChainStep (multichainstate_morph mstate1) (multichainstate_morph mstate2) ;
}.
```

As before, a multi-chain morphism takes reachable states to reachable states.

```
Lemma multichain_reachable_to_reachable :
    forall mstate (multichain_morph : MultiChainMorphism),
    multichain_reachable mstate -> multichain_reachable (multichain_morph mstate).
Proof. (* omitted *) Qed.
```

Multi-chain morphisms can be composed, and composition is associative.

```
(* an auxiliary lemma *)
Lemma multichain_compose_fixpoint_result (mch1 mch2 : MultiChainMorphism) :
```

```
    compose (multichainstate_morph mch2) (multichainstate_morph mch1) multichain_empty_state =
    multichain_empty_state.
Proof. (* proof omitted *) Qed.

(* composition of multi-chain morphisms *)
Definition composition_mch (mch1 mch2 : MultiChainMorphism) : MultiChainMorphism := {|
    multichainstate_morph :=
        compose (multichainstate_morph mch1) (multichainstate_morph mch2) ;
        empty_fixpoint := multichain_compose_fixpoint_result mch1 mch2 ;
        multichainstep_morph := (fun b1 b2 =>
            compose
                (multichainstep_morph mch2
                    (multichainstep_morph mch1 b1)
                    (multichainstep_morph mch2 b2))
                (multichainstep_morph mch1 b1 b2)) ;
|}.

(* composition is associative *)
Lemma composition_assoc_mch : forall mch1 mch2 mch3,
    composition_mch (composition_mch mch3 mch2) mch1 =
    composition_mch mch3 (composition_mch mch2 mch1).
Proof. (* proof omitted *) Qed.
```

We also have the identity morphism, which is trivial under composition.

```
(* an auxiliary lemma *)
Lemma multi_empty_fixpoint_id :
    @id MultiChainState multichain_empty_state = multichain_empty_state.
Proof. auto. Qed.

(* the identity multi-chain morphism *)
Definition id_mch : MultiChainMorphism := {|
    multichainstate_morph := @id MultiChainState ;
    multi_empty_fixpoint := multi_empty_fixpoint_id ;
    multichainstep_morph := (fun b1 b2 => @id (MultiChainStep b1 b2)) ;
|}.

(* left and right composition results *)
Lemma composition_mch_left : forall mch, composition_mch id_mch mch = mch.
Proof. (* proof omitted *) Qed.

Lemma composition_mch_right : forall mch, composition_mch mch id_mch = mch.
Proof. (* proof omitted *) Qed.
```

Finally, an multi-chain isomorphism is a bisimulation, and defined as before.

```
Definition is_iso_mch (f g : MultiChainMorphism) : Prop :=
    composition_mch g f = id_mch /\
    composition_mch f g = id_mch.
```

Example 6.3.2 (Null Multi-Chain Morphism)**.** As before, we have a canonical null morphism, which sends all multi-chain states to the multi-chain empty state, and all multi-chain steps to the trivial permutation pair of the empty queue pair of the multi-chain empty state.

```
(* function definitions *)
Definition multichainstate_morph (_ : MultiChainState) : MultiChainState :=
    multichain_empty_state.

Lemma multichain_empty_fixpoint : multichainstate_morph multichain_empty_state =
    multichain_empty_state.
Proof. unfold multichainstate_morph. auto. Qed.

Definition multichainstep_morph (mstate1 mstate2 : MultiChainState)
    (_ : MultiChainStep mstate1 mstate2) :
    MultiChainStep multichain_empty_state multichain_empty_state :=
    (step_permute empty_self_equiv id_permute perm_nil,
     step_permute empty_self_equiv id_permute perm_nil).

(* definition of the null multi-chain morphism of chains *)
Definition null_multichain_morphism : MultiChainMorphism :=
    build_multi_chain_morphism multichainstate_morph multichain_empty_fixpoint
    multichainstep_morph.
```

6.3.5 Multi-Chain Weak Equivalences

Definition 6.3.3 (Mult-Chain Weakly Equivalent Systems of Contracts)**.** Let `Chain1` and `Chain2` be concurrent chains. Consider two multi-chain systems of contracts `S_1` and `S_2`, where `S_1` is given by `C_1, C_2, ... , C_n` deployed on `Chain1` and `D_1, D_2, ... , D_m` deployed on `Chain2`, and `S_2` is given by `C'_1, C'_2, ... , C'_n'` deployed on `Chain1` and `D'_1, D'_2, ... , D'_m'` deployed on `Chain2`.

Then `S1` and `S2` are multi-chain weakly equivalent if the corresponding sum contracts on each chain are weakly equivalent. That is, if `C_1 ⊔ C_2 ⊔ ... ⊔ C_n` and `C'_1 ⊔ C'_2 ⊔ ... ⊔ C'_n'` are weakly equivalent on `Chain1`, and `D_1 ⊔ D_2 ⊔ ... ⊔ D_n` and `D'_1 ⊔ D'_2 ⊔ ... ⊔ D'_n'` are weakly equivalent on `Chain2`.

```
(* weakly equivalent systems of contracts *)
Definition are_multichain_weakly_equiv_system (C C' : list @Contract Base1) (D D' : list
```

```
      @Contract Base2) : Prop :=
    are_weakly_equiv (foldl_sum_contract C) (foldl_sum_contract C') /\
    are_weakly_equiv (foldl_sum_contract D) (foldl_sum_contract D').
```

Because multi-chain weak equivalence depends on weak equivalence on each constituent chain, the multi-chain null contract is the identity element of a monoid.

```
Theorem multichain_null_contract_monoid_identity :
    forall (S : MultiChainContract),
    are_multichain_weakly_equiv_system S (multichain_sum S multichain_null_contract) /\
    are_multichain_weakly_equiv_system (multichain_sum S multichain_null_contract) S.
```

Furthermore, multi-chain weak equivalences are an equivalence relation on multi-chain contracts.

```
(* relfexivity *)
Lemma are_multichain_equiv_refl : forall S,
    are_multichain_equiv S S.
Proof. (* proof omitted *) Qed.

(* symmetry *)
Lemma are_multichain_equiv_sym : forall S S',
    are_multichain_equiv S S' ->
    are_multichain_equiv S' S.
Proof. (* proof omitted *) Qed.

(* transitivity *)
Lemma are_multichain_equiv_trans : forall S S' S''
    are_multichain_equiv S S' ->
    are_multichain_equiv S' S'' ->
    are_multichain_equiv S S''.
Proof. (* proof omitted *) Qed.
```

6.4 Applications to Process-Algebraic Approaches

Weak equivalences give us a formal mechanism to embed process-algebraic semantics because they can be used to show an equivalence between systems of interacting contracts (processes) and a single, equivalent contract (process). We also now have a strong notion of various bisimulations, each of which targets different levels of equivalence.

In particular consider Tolmach *et al.*'s process-algebraic approach to formally analyzing composable DeFi protocols [183], which is formalized in CSP# [113], a high-level formal language similar to TLA+ which

models process algebras and concurrent and distributed systems. If we had a formalization in ConCert, we could in principle reason process-algebraically about executable code, which is a stronger assertion than reasoning in CSP#.

Tolmach *et al.* formally model components of DeFi protocols, in particular tokens and pools (as we have done in Chapter 4), and use these to model Curve and Compound, two widely used decentralized exchanges, and the stablecoin USDC. In §2.2, the authors define various abstract processes that can be composed; inherent to this is a notion of equivalence of contracts. With the notions of equivalence defined in this chapter, the semantics of Tolmach *et al.*'s theory can in principle be embedded into ConCert. The advantage of doing so would be to remove the assumptions necessary to the abstractions they made in their model of blockchain execution semantics.

Let us also look at Bartoletti *et al.*'s theory of AMMs and DeFi [48]. In their formal model of a blockchain as a state transition machine, they have a notion of a *reachable state*, *equivalence of states*, and state transitions which themselves can be composed and equal. In principle, Bartoletti *et al.*'s theory can also be embedded into ConCert using our notions of equivalence. Because the goal of Chapter 4 was to reason about economic properties of smart contracts specifications, and Bartoletti *et al.* do this using process-algebraic methods, the work of this chapter has immediate applications to furthering the work of Chapter 4.

6.4.1 Modeling Systems of Contracts With Bigraphs

Bigraphs are a computational modelling formalism which extend process algebras and which have not yet been explored in the context of smart contract verification, despite having been successfully used to formalize a wide range of models and situations [74]. A *bigraph* is a superposition of a hypergraph (the *link graph*) and a forest (the *place graph*). Both graphs share a node set, where nodes model processes. The link graph models how processes interact, where processes can be linked via *ports*, and the place graph models a spacial notion of how processes can be nested. Bigraphs evolved from various process calculi, including the π calculus, and are meant to subsume the Calculus of Communicating Systems [141]. For a short introduction, see [142].

We argue that (directed) bigraphs model systems of smart contracts in a natural way. We treat individual contracts, deployed at an address, as a process, and thus as a vertex in a bigraph. Each contract has entrypoints, which are described by the *ports* of a vertex. The link graph then describes contract calls, where there is a link between two ports if calling one of those ports can result in a call of the other. Using a directed bigraph allows us to indicate the direction of the contract call. This kind of interconnectivity based on message-passing is ubiquitous in financial smart contracts. Furthermore, the emergent behavior of systems connected in this way is precisely what we wish to understand.

The place graph gives us a natural grouping for smart contracts. Most systems of financial contracts have a main contract, with which every component interacts, and which we would indicate as a region of the place graph. In the case we're modeling various interacting systems, each region (or subtree and root of the place graph) represents the system corresponding to a particular smart contract.

Interoperability between these systems of contracts behaves precisely as modeled by bigraphs: systems of contracts implicitly contain interfaces which specify how other contracts can interact with the system and in what way. Internal workings of the contract, including oracles and auxiliary token contracts, will require permissions to operate. The resulting algebraic relations make a bigraph a natural model for financial contracts; composability is the key property of DeFi which gives it the colloquial name "money legos."

Finally, the notions we developed here of direct sums of contracts and of equivalence are good starting places to embed the algebraic relations of bigraphs in ConCert. The advantage to embedding the semantics of bigraphs into ConCert is that the abstraction semantics can be defined explicitly and manipulate executable code, as opposed to a system like BigraphER [171] or jLibBig [74, 135] which are designed, similar to TLA+ [124], to reason about abstract systems from a high level.

6.5 Conclusion

The purpose of this chapter was to develop the theory of equivalence of contracts from various perspectives to enable process-algebraic approaches to reasoning about systems of smart contracts. The notions of equivalence presented in this chapter range from strong equivalence to weak, and from local—equivalence only at the contract level—to global—systems of multi-chain contracts—in scope.

As we've seen in the literature, process-algebraic approaches have already been developed abstractly to reason about complex systems of interacting financial smart contracts—what is colloquially referred to as DeFi. Some of these efforts have direct implications on desirable economic properties of smart contracts, so in furthering the research of this chapter we can also extend the research of Chapter 4. Insofar as upgradeable contracts interact in complex systems of contracts, in principle this chapter also lays the foundation to extend the research of Chatper 5.

Chapter 7

Conclusion

The goal of this thesis has been to mathamatically codify complex behavior of smart contracts in order to reason about the correctness of contract specifications. Three principle areas, economic properties, upgradeability properties, and the emergent behavior of complex systems of contracts, are implicit to, but in general out of scope of, the formation of a contract specification. In this thesis we have introduced mathematical tools to reason about a contract specification, with the aim of precisely defining these implicit properties and rigorously showing that a given specification satisfies them.

In Chatper 4, we introduced the notion of a metaspecification and used it to reason about the economic properties of a contract's specification. In Chapter 5, we introduced the notion of a morphism of contracts and used it to classify the bounds of a contract's upgradeability, isolating its upgradeability skeleton and its interchangeable functionality. In Chapter 6, we introduced various notions of equivalences of contracts, from the strong notion of contract equivalence to weaker notions such as weak equivalences and multi-chain weak equivalences. This final chapter lays the foundations necessary to rigorously embed process-algebraic approaches to reasoning about systems of contracts in ConCert. This, in turn, can be applied to extend the research of Chapters 4 and 5.

7.1 Limitations and Future Work

Each chapter of this thesis has limitations that we hope to address in future work.

Firstly, the economic properties from Chapter 4 were derived from related work but the six results we proved about the contract specification of structured pools did not prove conclusively or precisely that the structured pool contract satisfied *all* desirable economic properties. We showed how this might in principle be done,

establishing the abstraction of a specification and subsequent reasoning about its properties, but in order to conclusively prove that a contract is well-behaved economically we need to embed an economic theory into ConCert. Previous efforts to do this (*e.g.* [48, 183]) have been process-algebraic in nature, so in order to address the limitation of Chapter 4, we propose that embedding a theory of DeFi into ConCert, building off of Chapter 6, is the next important step to take.

Secondly, in Chapter 5 we did not classify what results of a contract can be proved in relation to another contract via a morphism. Instead, we gave some generic heuristics and techniques in §5.3 so that these properties can be discovered empirically. Future research related to Chapter 5 should then include deeper results on the kinds of properties that are provable with contract morphisms.

Another related weakness of Chapter 5 is that there are no theoretical results to indicate that a particular decomposition of an upgradeable contract via an embedding and a surjection is "correct." Instead, we left this to be discovered empirically, again using techniques introduced in §5.3, where a particular decomposition is correct if it is useful to prove the desired results. Decompositions are not unique, and as it stands reasoning about an upgradeable contract in this way is more of an art of choosing a correct decomposition, rather than a precise science. Future research related to upgradeability should include an attempt to mathematically classify various splittings of contracts similar to those we explored in §5.4.3. We conjecture that there is stronger mathematical structure, perhaps like a fibration, in these decompositions.

Finally, the notions of equivalence in Chapter 6 are not accompanied with corresponding proofs of propositional indistinguishability. In other words, we do not have strong results that precisly indicate in to what degree and in what sense contracts are equivalent. Rather, as before we give tools which can be used to discover this empirically. This is a deep theoretical topic, relevant in many branches of mathematics, which we hope to explore; for example, it might be appropriate to introduce univalence—a logical axiom that says that equivalent contracts are equal—into ConCert [43]. Future work related to this topic could include developing a library of isomorphism-preserving transformations.

Of course, as the work of Chapter 6 is to lay foundations for process-algebraic verification methods, which have not yet been implemented. Doing so is an obvious and pressing matter of future work.

Appendix A

Proofs of Theorems

A.1 Metaspecification Proofs

Here we include the Coq proofs of each property of the metaspecification in Chapter 4.

A.1.1 Demand Sensitivity

The full Coq proof of demand sensitivity from §4.3.3.

```
(** Demand Sensitivity :

    A trade for a given token increases its price relative to other constituent tokens,
    so that higher relative demand corresponds to a higher relative price.
    Likewise, trading one token in for another decreases the first's relative price in
    the pool, corresponding to slackened demand. This enforces the classical notion of
    supply and demand

    We prove that r_x' > r_x and forall t_z, r_z' = r_z.
*)

(* x decreases relative to z as x => x', z => z' :
    z - x <= z' - x' *)
Definition rel_decr (x z x' z' : N) :=
    ((Z.of_N z) - (Z.of_N x) <= (Z.of_N z') - (Z.of_N x'))%Z.

Lemma rel_decr_lem : forall x x' z : N,
    x' <= x ->
```

```
    rel_decr x z x' z.
Proof.
    intros * x_leq.
    unfold rel_decr.
    lia.
Qed.

(* y increases relative to x as y => y', x => x' :
    y - x <= y' - x' *)
Definition rel_incr (y x y' x' : N) :=
    ((Z.of_N y) - (Z.of_N x) <= (Z.of_N y') - (Z.of_N x'))%Z.

Lemma rel_incr_lem : forall x x' y : N,
    x' <= x ->
    rel_incr y x y x'.
Proof.
    intros * x_leq.
    unfold rel_incr.
    lia.
Qed.

(** Theorem (Demand Sensitivity):
    Let t_x and t_y, t_x \neq t_y, be tokens in our family with nonzero pooled liquidity and
    exchange rates r_x, r_y > 0.
    In a trade t_x to t_y, as r_x is updated to r_x', it decereases relative to r_z for all
    t_z \neq t_x, and r_y strictly increases relative to r_x.
*)

Theorem demand_sensitivity cstate :
    (* For all tokens t_x t_y, rates r_x r_y, and quantities x and y, where *)
    forall t_x r_x x t_y r_y y,
    (* t_x is a token with nonzero pooled liquidity and with rate r_x > 0, and *)
    FMap.find t_x (stor_tokens_held cstate) = Some x /\ x > 0 /\
    FMap.find t_x (stor_rates cstate) = Some r_x /\ r_x > 0 ->
    (* t_y is a token with nonzero pooled liquidity and with rate r_y > 0 *)
    FMap.find t_y (stor_tokens_held cstate) = Some y /\ y > 0 /\
    FMap.find t_y (stor_rates cstate) = Some r_y /\ r_y > 0 ->
    (* In a trade t_x to t_y ... *)
    forall chain ctx msg msg_payload acts cstate',
        (* i.e.: a successful call to the contract *)
        receive contract chain ctx cstate (Some msg) = Ok(cstate', acts) ->
        (* which is a trade *)
        msg = trade msg_payload ->
        (* from t_x to t_y *)
        msg_payload.(token_in_trade) = t_x ->
```

```
62          msg_payload.(token_out_trade) = t_y ->
63          (* with t_x <> t_y *)
64          t_x <> t_y ->
65      (* ... as r_x is updated to r_x': ... *)
66      let r_x' := get_rate t_x (stor_rates cstate') in
67      (* (1) r_x decreases relative to all rates r_z, for t_z <> t_x, and *)
68      (forall t_z,
69          t_z <> t_x ->
70          let r_z  := get_rate t_z (stor_rates cstate) in
71          let r_z' := get_rate t_z (stor_rates cstate') in
72          rel_decr r_x r_z r_x' r_z') /\
73      (* (2) r_y strictly increases relative to r_x *)
74      let t_y := msg_payload.(token_out_trade) in
75      let r_y  := get_rate t_y (stor_rates cstate) in
76      let r_y' := get_rate t_y (stor_rates cstate') in
77      rel_incr r_y r_x r_y' r_x'.
78  Proof.
79      intros ? ? ? ? ? ? t_x_data t_y_data ? ? ? ? ? ? successful_txn msg_is_trade
80          token_in_tx token_out_ty tx_neq_ty r_x'.
81      destruct t_x_data as [stor_tx t_x_data].
82      destruct t_x_data as [x_geq0 t_x_data].
83      destruct t_x_data as [rate_rx rx_geq_0].
84      (* first, prove that r_x' < r_x while r_z = r_z' for all other rates *)
85      assert (r_x' <= r_x /\
86          forall t,
87          t <> t_x ->
88          get_rate t (stor_rates cstate') = get_rate t (stor_rates cstate))
89      as change_lemma.
90      {   intros.
91          is_sp_destruct.
92          rewrite msg_is_trade in successful_txn.
93          pose proof (trade_entrypoint_check_pf cstate chain ctx msg_payload
94                  cstate' acts successful_txn)
95          as token_in_qty.
96          do 3 destruct token_in_qty as [* token_in_qty].
97          destruct token_in_qty as [token_in_qty rates_exist].
98          destruct rates_exist as [rate_in_exists rate_out_exists].
99          (* get the new rates *)
100         rewrite <- token_out_ty in tx_neq_ty.
101         rewrite <- msg_is_trade in successful_txn.
102         rewrite msg_is_trade in successful_txn.
103         pose proof (trade_update_rates_formula_pf cstate chain ctx msg_payload cstate' acts
        successful_txn) as updating_rates.
104         destruct updating_rates as [_ updating_rates].
105         destruct updating_rates as [calc_new_rate_x other_rates_equal].
```

```
106         (* split into cases *)
107         split.
108         -   unfold r_x'. unfold get_rate.
109             rewrite token_in_tx in calc_new_rate_x.
110             replace (FMap.find t_x (stor_rates cstate'))
111             with (Some
112                 (calc_rx'
113                     (get_rate t_x (stor_rates cstate))
114                     (get_rate (token_out_trade msg_payload) (stor_rates cstate))
115                     (qty_trade msg_payload)
116                     (stor_outstanding_tokens cstate)
117                     (get_bal t_x (stor_tokens_held cstate)))).
118             assert (r_x = (get_rate t_x (stor_rates cstate))) as r_x_is_r_x.
119             {   unfold get_rate.
120                 now replace (FMap.find t_x (stor_rates cstate))
121                 with (Some r_x). }
122             now rewrite r_x_is_r_x.
123         -   intros t t_neq_tx.
124             unfold get_rate.
125             replace (FMap.find t (stor_rates cstate')) with (FMap.find t (stor_rates cstate)).
126             2:{ rewrite <- token_in_tx in t_neq_tx.
127                 apply (eq_sym (other_rates_equal t t_neq_tx)). }
128             auto. }
129     (* now use change_lemma to prove the result *)
130     destruct change_lemma as [rx_change tz_change].
131     split.
132     (* prove: r_x decreases relative to all rates r_z, for t_z <> t_x *)
133     -   intros t_z z_neq_x.
134         pose proof (tz_change t_z z_neq_x) as tz_change'.
135         clear tz_change.
136         rewrite tz_change'.
137         apply rel_decr_lem.
138         exact rx_change.
139     (* prove: r_y strictly increases relative to r_x *)
140     -   unfold rel_incr.
141         rewrite <- token_out_ty in tx_neq_ty.
142         pose proof (tz_change (token_out_trade msg_payload) (not_eq_sym tx_neq_ty)) as
    tz_change'.
143         clear tz_change.
144         rewrite tz_change'.
145         apply rel_incr_lem.
146         exact rx_change.
147 Qed.
```

A.1.2 Nonpathological Prices

The full Coq proof of nonpathological prices from §4.3.4.

```
(** Nonpathological prices

    As relative prices shift in response to trading activity, a price that starts out nonzero
    never goes to zero or to a negative value.

    This is to avoid pathological behavior of zero or negative prices.

    Note, however, that prices can still get arbitrarily close to zero, like in the case
    of CPMMs.
*)

(** Theorem (Nonpathological Prices):
    For a token t_x in T, if there is a contract state such that r_x > 0,
    then r_x > 0 holds for all future states of the contract.
*)

Theorem nonpathological_prices bstate caddr :
    (* reachable state with contract at caddr *)
    reachable bstate ->
    env_contracts bstate caddr = Some (contract : WeakContract) ->
    (* the statement *)
    exists (cstate : State),
    contract_state bstate caddr = Some cstate /\
    (* For a token t_x in T and rate r_x, *)
    forall t_x r_x,
    (* if r_x is the exchange rate of t_x, then r_x > 0 *)
    FMap.find t_x (stor_rates cstate) = Some r_x -> r_x > 0.
Proof.
    intros reachable_st contract_at_caddr.
    contract_induction; intros.
    (* Please reestablish the invariant after addition of a block *)
    -   now apply (IH t_x r_x).
    (* Please establish the invariant after deployment of the contract *)
    -   is_sp_destruct.
        pose proof (initialized_with_positive_rates_pf chain ctx setup result init_some) as
    init_rates.
        now apply (init_rates t_x r_x).
    (* Please reestablish the invariant after an outgoing action *)
    -   now apply (IH t_x r_x).
    (* Please reestablish the invariant after a nonrecursive call *)
    -   destruct msg.
        (* msg = Some m *)
```

```
42          +   is_sp_destruct.
43              pose proof (msg_destruct_pf m) as msg_destruct.
44              destruct msg_destruct as [pool | [unpool | [trade | other]]].
45              (* m = pool p : by the specification, pooling does not change exchange rates *)
46              *   destruct pool as [p msg_pool].
47                  rewrite msg_pool in receive_some.
48                  pose proof (pool_rates_unchanged_pf prev_state new_state chain ctx p new_acts
    receive_some t_x) as rates_unchanged.
49                  now replace (FMap.find t_x (stor_rates new_state))
50                  with (FMap.find t_x (stor_rates prev_state))
51                  in H7.
52              (* m = unpool p : by the specification, pooling does not change exchange rates *)
53              *   destruct unpool as [u msg_unpool].
54                  rewrite msg_unpool in receive_some.
55                  pose proof (unpool_rates_unchanged_pf prev_state new_state chain ctx u
    new_acts receive_some t_x) as rates_unchanged.
56                  now replace (FMap.find t_x (stor_rates new_state))
57                  with (FMap.find t_x (stor_rates prev_state))
58                  in H7.
59              (* m = trade p *)
60              *   destruct trade as [t msg_trade].
61                  rewrite msg_trade in receive_some.
62                  (* We split into cases (token_in_trade t) = t_x and (token_in_trade t) <> t_x
    *)
63                  destruct (token_eq_dec (token_in_trade t) t_x).
64                  (* first we treat unequality, which is the simpler case *)
65                  2:{ (* first use the spec to prove that the updated r_x is equal to the old
    r_x *)
66                      pose proof (trade_update_rates_formula_pf prev_state chain ctx t
67                          new_state new_acts receive_some)
68                      as rx_update.
69                      simpl in rx_update.
70                      destruct rx_update as [_ rates_update].
71                      simpl in rates_update.
72                      destruct rates_update as [_ tz_update].
73                      pose proof (tz_update t_x (not_eq_sym n)).
74                      now replace (FMap.find t_x (stor_rates new_state))
75                      with (FMap.find t_x (stor_rates prev_state))
76                      in H7. }
77                  (* now we have (token_in_trade t) = t_x *)
78                  (* we need to get r_x from prev_state *)
79                  assert (exists prev_rx, FMap.find t_x (stor_rates prev_state) = Some prev_rx)
80                  as exists_prevrx.
81                  {   pose proof (trade_entrypoint_check_pf prev_state chain ctx t
82                          new_state new_acts receive_some)
```

```
83                    as prev_rate_lem.
84                    do 3 destruct prev_rate_lem as [* prev_rate_lem].
85                    destruct prev_rate_lem as [_ prev_rate_lem].
86                    destruct prev_rate_lem as [in_lem _].
87                    rewrite e in in_lem.
88                    exists x0.
89                    apply in_lem. }
90                destruct exists_prevrx as [prev_rx *].
91                (* by IH prev_rx is positive *)
92                pose proof (IH t_x prev_rx H8) as prev_pos.
93                (* assert that there is some rate r_y of the  *)
94                assert (exists r_y, r_x = calc_rx' prev_rx r_y (qty_trade t) (
      stor_outstanding_tokens prev_state) (get_bal t_x (stor_tokens_held prev_state)))
95                as rx_update.
96                {   pose proof (trade_update_rates_formula_pf prev_state chain ctx t
97                        new_state new_acts receive_some)
98                    as rx_update.
99                    destruct rx_update as [_ rates_update].
100                   simpl in rates_update.
101                   destruct rates_update as [tx_update _].
102                   rewrite <- e in *.
103                   rewrite H7 in tx_update.
104                   inversion tx_update.
105                   exists (get_rate (token_out_trade t) (stor_rates prev_state)).
106                   f_equal.
107                   unfold get_rate.
108                   now replace (FMap.find (token_in_trade t) (stor_rates prev_state))
109                   with (Some prev_rx). }
110               destruct rx_update as [r_y rx_update].
111               (* we call on the result prev_rx > 0 => r_x > 0 *)
112               pose proof (update_rate_stays_positive_pf prev_rx r_y (qty_trade t) (
      stor_outstanding_tokens prev_state) (get_bal t_x (stor_tokens_held prev_state)))
113               as pos_new_rate.
114               rewrite <- rx_update in pos_new_rate.
115               cbn in pos_new_rate.
116               apply (pos_new_rate prev_pos).
117           (* finally, now for any other entrypoints *)
118           *   destruct other as [o msg_other_op].
119               rewrite msg_other_op in receive_some.
120               pose proof (other_rates_unchanged_pf prev_state new_state chain ctx o new_acts
       receive_some t_x) as rates_unchanged.
121               now replace (FMap.find t_x (stor_rates new_state))
122               with (FMap.find t_x (stor_rates prev_state))
123               in H7.
124       (* msg = None *)
```

```
125         +   is_sp_destruct. clear is_sp.
126             pose proof (none_fails_pf prev_state chain ctx) as none_fail.
127             destruct none_fail as [err none_fail].
128             rewrite none_fail in receive_some.
129             inversion receive_some.
130     (* Please reestablish the invariant after a recursive call *)
131     -   destruct msg.
132         (* msg = Some m *)
133         +   is_sp_destruct.
134             pose proof (msg_destruct_pf m) as msg_destruct.
135             destruct msg_destruct as [pool | [unpool | [trade | other]]].
136             (* m = pool p : by the specification, pooling does not change exchange rates *)
137             *   destruct pool as [p msg_pool].
138                 rewrite msg_pool in receive_some.
139                 pose proof (pool_rates_unchanged_pf prev_state new_state chain ctx p new_acts
        receive_some t_x) as rates_unchanged.
140                 now replace (FMap.find t_x (stor_rates new_state))
141                 with (FMap.find t_x (stor_rates prev_state))
142                 in H7.
143             (* m = unpool p : by the specification, pooling does not change exchange rates *)
144             *   destruct unpool as [u msg_unpool].
145                 rewrite msg_unpool in receive_some.
146                 pose proof (unpool_rates_unchanged_pf prev_state new_state chain ctx u
        new_acts receive_some t_x) as rates_unchanged.
147                 now replace (FMap.find t_x (stor_rates new_state))
148                 with (FMap.find t_x (stor_rates prev_state))
149                 in H7.
150             (* m = trade p *)
151             *   destruct trade as [t msg_trade].
152                 rewrite msg_trade in receive_some.
153                 (* We split into cases (token_in_trade t) = t_x and (token_in_trade t) <> t_x
        *)
154                 destruct (token_eq_dec (token_in_trade t) t_x).
155                 (* first we treat unequality, which is the simpler case *)
156                 2:{ (* first use the spec to prove that the updated r_x is equal to the old
        r_x *)
157                     pose proof (trade_update_rates_formula_pf prev_state chain ctx t
158                         new_state new_acts receive_some)
159                     as rx_update.
160                     simpl in rx_update.
161                     destruct rx_update as [_ rates_update].
162                     simpl in rates_update.
163                     destruct rates_update as [_ tz_update].
164                     pose proof (tz_update t_x (not_eq_sym n)).
165                     now replace (FMap.find t_x (stor_rates new_state))
```

```
166                     with (FMap.find t_x (stor_rates prev_state))
167                     in H7. }
168                 (* now we have (token_in_trade t) = t_x *)
169                 (* we need to get r_x from prev_state *)
170                 assert (exists prev_rx, FMap.find t_x (stor_rates prev_state) = Some prev_rx)
171                 as exists_prevrx.
172                 {   pose proof (trade_entrypoint_check_pf prev_state chain ctx t
173                         new_state new_acts receive_some)
174                     as prev_rate_lem.
175                     do 3 destruct prev_rate_lem as [* prev_rate_lem].
176                     destruct prev_rate_lem as [_ prev_rate_lem].
177                     destruct prev_rate_lem as [in_lem _].
178                     rewrite e in in_lem.
179                     exists x0.
180                     apply in_lem. }
181                 destruct exists_prevrx as [prev_rx *].
182                 (* by IH prev_rx is positive *)
183                 pose proof (IH t_x prev_rx H8) as prev_pos.
184                 (* assert that there is some rate r_y of the  *)
185                 assert (exists r_y, r_x = calc_rx' prev_rx r_y (qty_trade t) (
        stor_outstanding_tokens prev_state) (get_bal t_x (stor_tokens_held prev_state)))
186                 as rx_update.
187                 {   pose proof (trade_update_rates_formula_pf prev_state chain ctx t
188                         new_state new_acts receive_some)
189                     as rx_update.
190                     destruct rx_update as [_ rates_update].
191                     simpl in rates_update.
192                     destruct rates_update as [tx_update _].
193                     rewrite <- e in *.
194                     rewrite H7 in tx_update.
195                     inversion tx_update.
196                     exists (get_rate (token_out_trade t) (stor_rates prev_state)).
197                     f_equal.
198                     unfold get_rate.
199                     now replace (FMap.find (token_in_trade t) (stor_rates prev_state))
200                     with (Some prev_rx). }
201                 destruct rx_update as [r_y rx_update].
202                 (* we call on the result prev_rx > 0 => r_x > 0 *)
203                 pose proof (update_rate_stays_positive_pf prev_rx r_y (qty_trade t) (
        stor_outstanding_tokens prev_state) (get_bal t_x (stor_tokens_held prev_state)))
204                 as pos_new_rate.
205                 rewrite <- rx_update in pos_new_rate.
206                 cbn in pos_new_rate.
207                 apply (pos_new_rate prev_pos).
208             (* finally, now for any other entrypoints *)
```

```
209             *   destruct other as [o msg_other_op].
210                 rewrite msg_other_op in receive_some.
211                 pose proof (other_rates_unchanged_pf prev_state new_state chain ctx o new_acts
        receive_some t_x) as rates_unchanged.
212                 now replace (FMap.find t_x (stor_rates new_state))
213                 with (FMap.find t_x (stor_rates prev_state))
214                 in H7.
215         (* msg = None *)
216         +   is_sp_destruct. clear is_sp.
217             pose proof (none_fails_pf prev_state chain ctx) as none_fail.
218             destruct none_fail as [err none_fail].
219             rewrite none_fail in receive_some.
220             inversion receive_some.
221     (* Please reestablish the invariant after permutation of the action queue *)
222     -   now apply (IH t_x r_x).
223     -   solve_facts.
224 Qed.
```

A.1.3 Swap Rate Consistency

The full Coq proof of swap rate consistency from §4.3.5.

```
1  (** Swap rate consistency :
2      For a token t_x in T and for any delta_x > 0, there is no sequence of trades, beginning
       and
3      ending with t_x, such that delta_x' > delta_x, where delta_x' is the output quantity of
       the
4      sequence of trades.
5
6      Swap rate consistency means that it is never profitable to trade in a loop, e.g. t_x to
       t_y,
7      and back to t_x, which is important so that there are never any opportunities for
       arbitrage
8      internal to the pool.
9  *)
10
11 (* Some results on successive trades *)
12
13 (* first a type to describe successive trades *)
14 Record trade_sequence_type := build_trade_sequence_type {
15     seq_chain : ChainState ;
16     seq_ctx : ContractCallContext ;
17     seq_cstate : State ;
18     seq_trade_data : trade_data ;
```

```
    seq_res_acts : list ActionBody ;
}.

(* a function to calculate the the trade output of the final trade, delta_x' *)
Definition trade_to_delta_y (t : trade_sequence_type) :=
    let cstate := seq_cstate t in
    let token_in := token_in_trade (seq_trade_data t) in
    let token_out := token_out_trade (seq_trade_data t) in
    let rate_in := get_rate token_in (stor_rates cstate) in
    let rate_out := get_rate token_out (stor_rates cstate) in
    let delta_x := qty_trade (seq_trade_data t) in
    let k := stor_outstanding_tokens cstate in
    let x := get_bal token_in (stor_tokens_held cstate) in
    (* the calculation *)
    calc_delta_y rate_in rate_out delta_x k x.

Definition trade_to_rx' (t : trade_sequence_type) :=
    let cstate := seq_cstate t in
    let token_in := token_in_trade (seq_trade_data t) in
    let token_out := token_out_trade (seq_trade_data t) in
    let rate_in := get_rate token_in (stor_rates cstate) in
    let rate_out := get_rate token_out (stor_rates cstate) in
    let delta_x := qty_trade (seq_trade_data t) in
    let k := stor_outstanding_tokens cstate in
    let x := get_bal token_in (stor_tokens_held cstate) in
    (* the calculation *)
    calc_rx' rate_in rate_out delta_x k x.

(* a proposition that indicates a list of trades are successive, successful trades *)
Fixpoint are_successive_trades (trade_sequence : list trade_sequence_type) : Prop :=
    match trade_sequence with
    | [] => True
    | t1 :: l =>
        match l with
        | [] =>
            (* if the list has one element, it just has to succeed *)
            exists cstate' acts,
            receive contract
                (seq_chain t1)
                (seq_ctx t1)
                (seq_cstate t1)
                (Some (trade (seq_trade_data t1)))
            = Ok(cstate', acts)
        | t2 :: l' =>
            (* the trade t1 goes through, connecting t1 and t2 *)
```

```
            receive contract
                (seq_chain t1)
                (seq_ctx t1)
                (seq_cstate t1)
                (Some (trade (seq_trade_data t1)))
            = Ok(seq_cstate t2, seq_res_acts t2) /\
            (qty_trade (seq_trade_data t2) = trade_to_delta_y t1) /\
            (token_in_trade (seq_trade_data t2) = token_out_trade (seq_trade_data t1)) /\
            (are_successive_trades l)
        end
    end.

Fixpoint leq_list (l : list N) : Prop :=
    match l with
    | [] => True
    | h :: tl =>
        match tl with
        | [] => True
        | h' :: tl' => (h <= h') /\ leq_list tl
        end
    end.

Fixpoint geq_list (l : list N) : Prop :=
    match l with
    | [] => True
    | h :: tl =>
        match tl with
        | [] => True
        | h' :: tl' => (h >= h') /\ geq_list tl
        end
    end.

Lemma rev_injective : forall {A : Type} (l l' : list A), rev l = rev l' -> l = l'.
Proof.
  intros A l l' H_rev.
  rewrite <- (rev_involutive l).
  rewrite <- (rev_involutive l').
  apply f_equal, H_rev.
Qed.

Lemma geq_cons : forall a l,
    geq_list (a :: l) ->
    geq_list l.
Proof.
    intros * H_geq.
```

```
109     destruct l as [| a' l]; auto.
110     unfold geq_list in *.
111     now destruct H_geq as [geq ind_l].
112 Qed.
113 
114 Lemma geq_app : forall l l',
115     geq_list (l ++ l') ->
116     geq_list l /\ geq_list l'.
117 Proof.
118     intros * H_geq.
119     split.
120     -   induction l.
121         +   now unfold geq_list.
122         +   rewrite <- app_comm_cons in H_geq.
123             pose proof (IHl (geq_cons a (l ++ l') H_geq)) as geq_l.
124             unfold geq_list.
125             destruct l; auto.
126             unfold geq_list in geq_l.
127             split; try assumption.
128             clear geq_l.
129             rewrite <- app_comm_cons in H_geq.
130             unfold geq_list in H_geq.
131             now destruct H_geq as [g_geq_n _].
132     -   induction l; auto.
133         rewrite <- app_comm_cons in H_geq.
134         now apply geq_cons in H_geq.
135 Qed.
136 
137 Lemma geq_cons_remove_simpl : forall a a' l,
138     geq_list (a :: a' :: l) ->
139     geq_list (a :: l).
140 Proof.
141     intros * H_geq.
142     destruct l as [| a'' l].
143     -   now unfold geq_list.
144     -   unfold geq_list in *.
145         destruct H_geq as [a_geq_a' [a'_geq_a'' H_geq]].
146         split; auto.
147         apply N.ge_le in a_geq_a', a'_geq_a''.
148         apply N.le_ge.
149         now apply (N.le_trans a'' a' a).
150 Qed.
151 
152 Lemma list_decompose {A : Type} : forall (l : list A), l = [] \/ exists l' b, l = (l' ++ [b])%
        list.
```

```
Proof.
  intros.
  destruct (rev l) eqn:H_rev.
  - left.
    now apply rev_injective.
  - right.
    exists (rev l0), a.
    apply rev_injective.
    rewrite H_rev.
    rewrite rev_unit.
    now rewrite rev_involutive.
Qed.

Lemma geq_transitive : forall l fst lst,
    hd_error l = Some fst ->
    hd_error (rev l) = Some lst ->
    geq_list l ->
    lst <= fst.
Proof.
    intros * hd_fst tl_lst H_geq_list.
    destruct l as [| a l]; try rewrite hd_error_nil in *; try discriminate.
    pose proof (list_decompose l) as l_decompose.
    destruct l_decompose as [one_txn | l_decompose].
    -   rewrite one_txn in *.
        cbn in hd_fst. injection hd_fst. intro a_fst.
        cbn in tl_lst. injection tl_lst. intro a_lst.
        now rewrite <- a_fst, <- a_lst.
    -   destruct l_decompose as [l' [b l_decompose]].
        rewrite l_decompose in *. clear l_decompose.
        cbn in hd_fst. injection hd_fst. intro a_fst.
        assert (b = lst) as b_lst.
        {   cbn in tl_lst.
            rewrite rev_unit in tl_lst.
            rewrite <- app_comm_cons in tl_lst.
            now simpl in tl_lst. }
        (* by induction on l' *)
        induction l'.
        +   rewrite app_nil_l in *.
            unfold geq_list in H_geq_list.
            destruct H_geq_list as [qeg _].
            apply (N.ge_le a b) in qeg.
            now rewrite <- a_fst, <- b_lst.
        +   apply IHl'.
            *   cbn.
                rewrite rev_unit.
```

```
                rewrite <- app_comm_cons.
                now simpl.
            *   rewrite <- app_comm_cons in H_geq_list.
                now apply geq_cons_remove_simpl in H_geq_list.
Qed.

Definition trade_to_ry_delta_y (t : trade_sequence_type) :=
    let delta_y := trade_to_delta_y t in
    let rate_y := get_rate (token_out_trade (seq_trade_data t)) (stor_rates (seq_cstate t)) in
    rate_y * delta_y.

Lemma mult_strict_mono : forall x y z : N, x <> 0 -> x * y <= x * z -> y <= z.
Proof.
  intros x y z Hx Hz.
  now apply N.mul_le_mono_pos_l in Hz.
Qed.

Lemma hd_transitive { A B : Type } : forall (l : list A) (f : A -> B) fst,
    hd_error l = Some fst ->
    hd_error (map f l) = Some (f fst).
Proof.
    intros * fst_is_fst.
    destruct l; try rewrite hd_error_nil in *; try discriminate.
    cbn in fst_is_fst.
    cbn.
    injection fst_is_fst.
    intro a_eq_fst.
    now rewrite a_eq_fst.
Qed.

Lemma geq_list_is_sufficient : forall trade_sequence t_x t_fst t_last cstate r_x,
    (* more assumptions *)
    (hd_error trade_sequence) = Some t_fst ->
    (hd_error (rev trade_sequence)) = Some t_last ->
    token_in_trade (seq_trade_data t_fst) = t_x ->
    token_out_trade (seq_trade_data t_last) = t_x ->
    seq_cstate t_fst = cstate ->
    FMap.find t_x (stor_rates cstate) = Some r_x /\ r_x > 0 ->
    FMap.find t_x (stor_rates cstate) = FMap.find t_x (stor_rates (seq_cstate t_last)) ->
    (* the statement *)
    geq_list (map trade_to_ry_delta_y trade_sequence) ->
    let delta_x := qty_trade (seq_trade_data t_fst) in
    let delta_x' := trade_to_delta_y t_last in
    delta_x' <= delta_x.
Proof.
```

```
243     intros * fst_txn lst_txn from_tx to_tx start_cstate rx_exists one_tx_txn H_geq_list *.
244     is_sp_destruct.
245     (* a lemma *)
246     assert (forall t,
247         let r_x := get_rate (token_in_trade (seq_trade_data t)) (stor_rates (seq_cstate t)) in
248         let r_y := get_rate (token_out_trade (seq_trade_data t)) (stor_rates (seq_cstate t))
        in
249         let delta_x := qty_trade (seq_trade_data t) in
250         let delta_y := trade_to_delta_y t in
251         r_y * delta_y <= r_x * delta_x)
252     as trade_slippage.
253     {   intro.
254         cbn.
255         unfold trade_to_delta_y.
256         apply trade_slippage_pf. }
257     destruct trade_sequence; inversion fst_txn.
258     destruct trade_sequence.
259     (* the case that this is a trade t_x -> t_x *)
260     {   (* delta_x' < r_x / r_x delta_x => delta_x' < delta_x *)
261         rename H8 into tx_is_fst.
262         assert (t = t_last) as tx_is_last.
263         { cbn in lst_txn. injection lst_txn. auto. }
264         unfold delta_x', delta_x.
265         rewrite <- tx_is_fst, <- tx_is_last.
266         pose proof (trade_slippage t) as trade_lt. cbn in trade_lt.
267         rewrite <- from_tx in to_tx.
268         rewrite <- tx_is_fst, <- tx_is_last in *.
269         rewrite to_tx in trade_lt.
270         rewrite <- from_tx in rx_exists.
271         destruct rx_exists as [rx_exists rx_gt0].
272         rewrite <- start_cstate in rx_exists.
273         unfold get_rate in trade_lt.
274         replace (FMap.find (token_in_trade (seq_trade_data t)) (stor_rates (seq_cstate t)))
275         with (Some r_x)
276         in trade_lt.
277         assert (r_x <> 0) as rate_neq_0.
278         {   apply N.neq_0_lt_0.
279             now apply N.gt_lt. }
280         eapply mult_strict_mono; now try exact rate_neq_0. }
281     (* now, the case of t_x -> t_y -> [...] -> t_x *)
282     (* deduce that r_x' delta_x' <= r_y delta_y *)
283     set (r_x' := get_rate t_x (stor_rates (seq_cstate t_last))).
284     set (t_y := token_out_trade (seq_trade_data t_fst)).
285     set (r_y := get_rate t_y (stor_rates (seq_cstate t_fst))).
286     set (delta_y := trade_to_delta_y t_fst).
```

```
287     assert (r_x' * delta_x' <= r_y * delta_y)
288     as H_y_x'.
289     {   pose proof (hd_transitive (rev (t :: t0 :: trade_sequence)) trade_to_ry_delta_y t_last
         lst_txn)
290         as rev_map_head.
291         rewrite map_rev in rev_map_head.
292         pose proof (geq_transitive
293             (map trade_to_ry_delta_y (t :: t0 :: trade_sequence))
294             (trade_to_ry_delta_y t_fst)
295             (trade_to_ry_delta_y t_last)
296             (hd_transitive (t :: t0 :: trade_sequence) trade_to_ry_delta_y t_fst fst_txn)
297             rev_map_head
298             H_geq_list)
299         as geq_trans.
300         unfold trade_to_ry_delta_y in geq_trans.
301         unfold r_x', delta_x', r_y, delta_y, t_y.
302         now rewrite <- to_tx. }
303     (* recall r_y delta_y <= r_x' delta_x *)
304     assert (r_y * delta_y <= r_x' * delta_x)
305     as H_x_y.
306     {   destruct rx_exists as [rx_exists rx_gt0].
307         replace (FMap.find t_x (stor_rates cstate))
308         with (FMap.find t_x (stor_rates (seq_cstate t_last)))
309         in rx_exists.
310         unfold r_x', get_rate.
311         replace (FMap.find t_x (stor_rates (seq_cstate t_last)))
312         with (Some r_x).
313         pose proof (trade_slippage t_fst) as trade_lemma.
314         cbn in trade_lemma.
315         unfold r_y, delta_y, delta_x.
316         replace r_x
317         with (get_rate (token_in_trade (seq_trade_data t_fst)) (stor_rates (seq_cstate t_fst))
        ).
318         2:{ unfold get_rate.
319             replace (seq_cstate t_fst) with cstate.
320             replace (token_in_trade (seq_trade_data t_fst)) with t_x.
321             replace (FMap.find t_x (stor_rates cstate))
322             with (FMap.find t_x (stor_rates (seq_cstate t_last))).
323             now replace (FMap.find t_x (stor_rates (seq_cstate t_last)))
324             with (Some r_x). }
325         assumption. }
326     (* so r_x' delta_x' <= r_x' delta_x *)
327     pose proof (N.le_trans (r_x' * delta_x') (r_y * delta_y) (r_x' * delta_x) H_y_x' H_x_y)
328     as H_x'_x.
329     (* giving that delta_x' <= delta_x *)
```

```
330     assert (r_x' <> 0) as rate_neq_0.
331     {   destruct rx_exists as [rx_exists rx_gt0].
332         replace (FMap.find t_x (stor_rates cstate))
333         with (FMap.find t_x (stor_rates (seq_cstate t_last)))
334         in rx_exists.
335         unfold r_x', get_rate.
336         now replace (FMap.find t_x (stor_rates (seq_cstate t_last)))
337         with (Some r_x). }
338     now eapply mult_strict_mono.
339 Qed.
340
341 Lemma swap_rate_lemma : forall trade_sequence,
342     (* if this is a list of successive trades *)
343     are_successive_trades trade_sequence ->
344     (* then  *)
345     geq_list (map trade_to_ry_delta_y trade_sequence).
346 Proof.
347     intros * trades_successive.
348     induction trade_sequence as [| t1 trade_sequence IHtrade_sequence]; auto.
349     assert (forall a l, are_successive_trades (a :: l) -> are_successive_trades l)
350     as successive_iter.
351     {   intros *.
352         destruct l.
353         -   intro.
354             unfold are_successive_trades.
355             auto.
356         -   intro outer_successive.
357             unfold are_successive_trades in outer_successive.
358             destruct outer_successive as [H_recv [H_qty H_iter]].
359             now unfold are_successive_trades. }
360     apply successive_iter in trades_successive as successive_itered.
361     apply IHtrade_sequence in successive_itered.
362     rewrite map_cons.
363     destruct trade_sequence as [| t2 trade_sequence]; auto.
364     rewrite map_cons.
365     assert (forall a b l, a >= b /\ geq_list (b :: l) -> geq_list (a :: b :: l))
366     as geq_conds.
367     { auto. }
368     apply geq_conds.
369     rewrite map_cons in successive_itered.
370     split; auto.
371     (* some simplifying notation *)
372     set (t_x := token_in_trade (seq_trade_data t1)).
373     set (r_x := get_rate t_x (stor_rates (seq_cstate t1))).
374     set (r_x' := trade_to_rx' t1).
```

```
375     set (delta_x := qty_trade (seq_trade_data t1)).
376     set (t_y := token_out_trade (seq_trade_data t1)).
377     set (r_y := get_rate t_y (stor_rates (seq_cstate t1))).
378     set (r_y' := trade_to_rx' t2).
379     set (delta_y := trade_to_delta_y t1).
380     set (t_z := token_out_trade (seq_trade_data t2)).
381     set (r_z := get_rate t_z (stor_rates (seq_cstate t2))).
382     set (delta_z := trade_to_delta_y t2).
383     (* a lemma *)
384     assert (forall t,
385         let t_x := token_in_trade (seq_trade_data t) in
386         let t_y := token_out_trade (seq_trade_data t) in
387         let r_x' := trade_to_rx' t in
388         let r_y := get_rate t_y (stor_rates (seq_cstate t)) in
389         let delta_x := qty_trade (seq_trade_data t) in
390         let delta_y := trade_to_delta_y t in
391         r_y * delta_y <= r_x' * delta_x)
392     as trade_slippage_2.
393     {   is_sp_destruct.
394         intros.
395         unfold r_y0, delta_y0, r_x'0, delta_x0, trade_to_delta_y, trade_to_rx'.
396         apply trade_slippage_2_pf. }
397     (* recall r_z * delta_z <= r_y' * delta_y (for t_2) *)
398     assert (r_z * delta_z <= r_y' * delta_y) as le_1.
399     {   pose proof (trade_slippage_2 t2) as trade_leq.
400         cbn in trade_leq.
401         unfold r_z, t_z, delta_z, r_y', t_y, delta_y.
402         cbn in trades_successive. destruct trades_successive as [recv_some [qty_traded [
        in_out_eq _]]].
403         now rewrite <- qty_traded. }
404     (* recall r_y * delta_y <= r_x' * delta_x (for t_1) *)
405     assert (r_y * delta_y <= r_x' * delta_x) as le_2.
406     {   pose proof (trade_slippage_2 t1) as trade_leq.
407         cbn in trade_leq.
408         unfold r_y, t_y, delta_y, r_x', t_x, delta_x.
409         cbn in trades_successive. destruct trades_successive as [recv_some [qty_traded [
        in_out_eq _]]].
410         now rewrite <- qty_traded. }
411     (* recall r_y' < r_y *)
412     assert (r_y' <= r_y) as rates_decr.
413     {   cbn in trades_successive.
414         destruct trades_successive as [recv_some [qty_traded [in_out_eq _]]].
415         assert (get_rate t_y (stor_rates (seq_cstate t1))
416                 = get_rate t_y (stor_rates (seq_cstate t2)))
417         as ry_unchanged_in_t1.
```

```
418         {   is_sp_destruct.
419             pose proof (trade_update_rates_formula_pf (seq_cstate t1) (seq_chain t1) (seq_ctx
    t1) (seq_trade_data t1) (seq_cstate t2)(seq_res_acts t2) recv_some)
420             as rates_change.
421             destruct rates_change as [token_neq [_ rates_unchange]].
422             unfold t_y, get_rate.
423             pose proof (rates_unchange (token_out_trade (seq_trade_data t1)) (not_eq_sym
    token_neq))
424             as rates_unchange.
425             now replace (FMap.find (token_out_trade (seq_trade_data t1)) (stor_rates (
    seq_cstate t2)))
426             with (FMap.find (token_out_trade (seq_trade_data t1)) (stor_rates (seq_cstate t1))
    ). }
427         unfold t_y in ry_unchanged_in_t1.
428         rewrite <- in_out_eq in ry_unchanged_in_t1.
429         unfold r_y, r_y', t_y, trade_to_rx'.
430         rewrite <- in_out_eq.
431         rewrite ry_unchanged_in_t1.
432         is_sp_destruct.
433         apply rate_decrease_pf. }
434     (* so r_z * delta_z <= r_y' * delta_y <= r_y * delta_y *)
435     assert (r_y' * delta_y <= r_y * delta_y) as le_mid.
436     {   apply (N.mul_le_mono_nonneg_r r_y' r_y); try assumption.
437         cbn in trades_successive.
438         destruct trades_successive as [recv_some [qty_traded [in_out_eq _]]].
439         is_sp_destruct.
440         pose proof (trade_amounts_nonnegative_pf (seq_cstate t1) (seq_chain t1) (seq_ctx t1)
441         (seq_trade_data t1) (seq_cstate t2) (seq_res_acts t2) recv_some)
442         as trades_nonneg.
443         destruct trades_nonneg as [_ dy_nonneg].
444         now unfold delta_y, trade_to_delta_y. }
445     assert (r_z * delta_z <= r_y * delta_y) as le_comp.
446     {   apply (N.le_trans (r_z * delta_z) (r_y' * delta_y) (r_y * delta_y) le_1 le_mid). }
447     unfold trade_to_ry_delta_y.
448     unfold r_z, t_z, delta_z, r_y, t_y, delta_y in le_comp.
449     now apply N.le_ge.
450 Qed.
451
452 (** Theorem (Swap Rate Consistency):
453     Let t_x be a token in our family with nonzero pooled liquidity and r_x > 0.
454     Then for any delta_x > 0 there is no sequence of trades, beginning and ending with
455     t_x, such that delta_x' > delta_x, where delta_x' is the output quantity
456     of the sequence of trades.
457 *)
458
```

```
459 Theorem swap_rate_consistency bstate cstate :
460     (* Let t_x be a token with nonzero pooled liquidity and rate r_x > 0 *)
461     forall t_x r_x x,
462     FMap.find t_x (stor_rates cstate) = Some r_x /\ r_x > 0 ->
463     FMap.find t_x (stor_tokens_held cstate) = Some x /\ x > 0 ->
464     (* then for any delta_x > 0 and any sequence of trades, beginning and ending with t_x *)
465     forall delta_x (trade_sequence : list trade_sequence_type) t_fst t_last,
466     delta_x > 0 ->
467     (* trade_sequence is a list of successive trades *)
468     are_successive_trades trade_sequence ->
469     (* with a first and last trade, t_fst and t_last respectively, *)
470     (hd_error trade_sequence) = Some t_fst ->
471     (hd_error (rev trade_sequence)) = Some t_last ->
472     (* starting from our current bstate and cstate *)
473     seq_chain t_fst = bstate ->
474     seq_cstate t_fst = cstate ->
475     (* the first trade is from t_x *)
476     token_in_trade (seq_trade_data t_fst) = t_x ->
477     qty_trade (seq_trade_data t_fst) = delta_x ->
478     (* the last trade is to t_x *)
479     token_out_trade (seq_trade_data t_last) = t_x ->
480     FMap.find t_x (stor_rates cstate) = FMap.find t_x (stor_rates (seq_cstate t_last)) ->
481     (* delta_x', the output of the last trade, is never larger than delta_x. *)
482     let delta_x' := trade_to_delta_y t_last in
483     delta_x' <= delta_x.
484 Proof.
485     intros * H_rate H_held * dx_geq_0 trades_successive fst_txn lst_txn start_bstate
        start_cstate from_tx trade_delta_x to_tx one_tx_txn *.
486     unfold delta_x'. rewrite <- trade_delta_x.
487     apply (geq_list_is_sufficient trade_sequence t_x t_fst t_last cstate r_x); auto.
488     now apply swap_rate_lemma.
489 Qed.
```

A.1.4 Zero-Impact Liquidity Change

The full Coq proof of zero-impact liquidity change from §4.3.6.

```
1 (** Zero-Impact Liquidity Change :
2     The quoted price of trades is unaffected by depositing or withdrawing liquidity
3 *)
4
5 (** Theorem (Zero-Impact Liquidity Change)
6     The quoted price of trades is unaffected by calling `pool` and `unpool`.
7 *)
```

```
Theorem zero_impact_liquidity_change :
    (* Consider the quoted price of a trade t_x to t_y at cstate, *)
    forall cstate t_x t_y r_x r_y,
    FMap.find t_x (stor_rates cstate) = Some r_x ->
    FMap.find t_y (stor_rates cstate) = Some r_y ->
    let quoted_price := r_x / r_y in
    (* and a successful POOL or UNPOOL action. *)
    forall chain ctx msg payload_pool payload_unpool acts cstate' r_x' r_y',
        receive contract chain ctx cstate (Some msg) = Ok(cstate', acts) ->
        msg = pool payload_pool \/
        msg = unpool payload_unpool ->
    (* Then take the (new) quoted price of a trade t_x to t_y at cstate'. *)
    FMap.find t_x (stor_rates cstate') = Some r_x' ->
    FMap.find t_y (stor_rates cstate') = Some r_y' ->
    let quoted_price' := r_x' / r_y' in
    (* The quoted price is unchanged. *)
    quoted_price = quoted_price'.
Proof.
    intros * rx_rate ry_rate * successful_txn msg_pool_or_unpool new_rx_rate new_ry_rate *.
    destruct msg_pool_or_unpool as [msg_pool | msg_unpool].
    (* the successful transaction was a deposit *)
    -   is_sp_destruct.
        assert (FMap.find t_x (stor_rates cstate') = FMap.find t_x (stor_rates cstate))
        as rate_unchanged_x.
        {   rewrite msg_pool in successful_txn.
            exact (eq_sym (pool_rates_unchanged_pf cstate cstate' chain ctx payload_pool acts
            successful_txn t_x)). }
        assert (FMap.find t_y (stor_rates cstate') = FMap.find t_y (stor_rates cstate))
        as rate_unchanged_y.
        {   rewrite msg_pool in successful_txn.
            exact (eq_sym (pool_rates_unchanged_pf cstate cstate' chain ctx payload_pool acts
                successful_txn t_y)). }
        replace (FMap.find t_x (stor_rates cstate'))
        with (FMap.find t_x (stor_rates cstate))
        in new_rx_rate.
        replace (FMap.find t_y (stor_rates cstate'))
        with (FMap.find t_y (stor_rates cstate))
        in new_ry_rate.
        replace (FMap.find t_x (stor_rates cstate))
        with (Some r_x')
        in rx_rate.
        replace (FMap.find t_y (stor_rates cstate))
        with (Some r_y')
        in ry_rate.
        injection rx_rate as rx_eq.
```

```
53          injection ry_rate as ry_eq.
54          unfold quoted_price, quoted_price'.
55          now rewrite rx_eq, ry_eq.
56      (* the successful transaction was a withdraw *)
57      -   is_sp_destruct.
58          assert (FMap.find t_x (stor_rates cstate') = FMap.find t_x (stor_rates cstate))
59          as rate_unchanged_x.
60          {   rewrite msg_unpool in successful_txn.
61              exact (eq_sym (unpool_rates_unchanged_pf cstate cstate' chain ctx payload_unpool
        acts
62              successful_txn t_x)). }
63          assert (FMap.find t_y (stor_rates cstate') = FMap.find t_y (stor_rates cstate))
64          as rate_unchanged_y.
65          {   rewrite msg_unpool in successful_txn.
66              exact (eq_sym (unpool_rates_unchanged_pf cstate cstate' chain ctx payload_unpool
        acts
67                  successful_txn t_y)). }
68          replace (FMap.find t_x (stor_rates cstate'))
69          with (FMap.find t_x (stor_rates cstate))
70          in new_rx_rate.
71          replace (FMap.find t_y (stor_rates cstate'))
72          with (FMap.find t_y (stor_rates cstate))
73          in new_ry_rate.
74          replace (FMap.find t_x (stor_rates cstate))
75          with (Some r_x')
76          in rx_rate.
77          replace (FMap.find t_y (stor_rates cstate))
78          with (Some r_y')
79          in ry_rate.
80          injection rx_rate as rx_eq.
81          injection ry_rate as ry_eq.
82          unfold quoted_price, quoted_price'.
83          now rewrite rx_eq, ry_eq.
84  Qed.
```

A.1.5 Arbitrage Sensitivity

The full Coq proof of arbitrage sensitivity from §4.3.7.

```
1  (** Arbitrage Sensitivity :
2
3      If an external, demand-sensitive market prices a constituent token differently from
4      the structured pool, a sufficiently large arbitrage transaction will equalize
5      the prices of the external market and the structured pool, or deplete the pool.
```

```
6
7       In our case, this happens because prices adapt through trades due to demand
8       sensitivity or the pool depletes in that particular token.
9   *)
10
11
12  (** Theorem (Arbitrage Sensitivity):
13      Let t_x be a token in our family with nonzero pooled liquidity and r_x > 0.
14      If an external, demand-sensitive market prices t_x differently from the structured pool,
15      then assuming sufficient liquidity,  with a sufficiently large transaction either the
16      price of t_x in the structured pool converges with the external market, or the trade
17      depletes the pool of t_x.
18  *)
19  Theorem arbitrage_sensitivity :
20      forall cstate t_x r_x x,
21      (* t_x is a token with nonzero pooled liquidity *)
22      FMap.find t_x (stor_rates cstate) = Some r_x /\ r_x > 0 /\
23      FMap.find t_x (stor_tokens_held cstate) = Some x /\ x > 0 ->
24      (* we consider some external price *)
25      forall external_price,
26      0 < external_price ->
27      (* and a trade of trade_qty succeeds *)
28      forall chain ctx msg msg_payload cstate' acts,
29      receive contract chain ctx cstate msg = Ok(cstate', acts) ->
30      msg = Some(trade msg_payload) ->
31      t_x = (token_in_trade msg_payload) ->
32      (* the arbitrage opportunity is resolved *)
33      let r_x' := get_rate t_x (stor_rates cstate') in
34      (* first the case that the external price was lower *)
35      (external_price < r_x ->
36          exists trade_qty,
37          msg_payload.(qty_trade) = trade_qty ->
38          external_price >= r_x') /\
39      (* second the case that the external price is higher *)
40      (external_price > r_x ->
41      exists trade_qty,
42          msg_payload.(qty_trade) = trade_qty ->
43          external_price <= r_x' \/
44      let t_y := token_out_trade msg_payload in
45      let r_y := get_rate t_y (stor_rates cstate) in
46      let x := get_bal t_x (stor_tokens_held cstate) in
47      let y := get_bal t_y (stor_tokens_held cstate) in
48      let balances := (stor_tokens_held cstate) in
49      let k := (stor_outstanding_tokens cstate) in
50      get_bal t_y balances <= calc_delta_y r_x r_y trade_qty k x).
```

```
Proof.
    intros * liquidity * ext_pos * successful_trade msg_is_trade tx_trade_in.
    (* show that the trade has equalized arbitrage *)
    split.
    (* the external market prices lower than the contract *)
    -   is_sp_destruct.
        intro ext_lt_rx.
        (* the strategy is to buy low (externally), sell high (internally) *)
        pose proof arbitrage_lt_pf r_x (get_rate (token_out_trade msg_payload) (stor_rates
    cstate)) external_price  (stor_outstanding_tokens cstate) (get_bal t_x
            (stor_tokens_held cstate)) ext_pos ext_lt_rx
        as arb_lemma_lt.
        destruct arb_lemma_lt as [trade_qty new_rate_lt].
        exists trade_qty.
        (* use the spec to get r_x' *)
        intro traded_qty.
        rewrite msg_is_trade in successful_trade.
        pose proof trade_update_rates_formula_pf cstate chain ctx msg_payload cstate' acts
            successful_trade as new_rate_calc.
        destruct new_rate_calc as [_ rates].
        destruct rates as [new_rx _].
        unfold get_rate.
        rewrite tx_trade_in.
        replace (FMap.find (token_in_trade msg_payload) (stor_rates cstate'))
        with (Some
        (calc_rx' (get_rate (token_in_trade msg_payload) (stor_rates cstate))
           (get_rate (token_out_trade msg_payload) (stor_rates cstate)) (qty_trade msg_payload
    )
           (stor_outstanding_tokens cstate) (get_bal (token_in_trade msg_payload) (
    stor_tokens_held cstate)))).
        assert (r_x = (get_rate (token_in_trade msg_payload) (stor_rates cstate)))
        as rx_got_rate.
        {   unfold get_rate.
            destruct liquidity as [rx_l _].
            rewrite tx_trade_in in rx_l.
            now replace (FMap.find (token_in_trade msg_payload) (stor_rates cstate))
            with (Some r_x). }
        rewrite rx_got_rate in new_rate_lt.
        rewrite <- traded_qty in new_rate_lt.
        rewrite tx_trade_in in new_rate_lt.
        apply (N.le_ge (calc_rx' (get_rate (token_in_trade msg_payload) (stor_rates cstate))
        (get_rate (token_out_trade msg_payload) (stor_rates cstate)) (qty_trade msg_payload)
        (stor_outstanding_tokens cstate) (get_bal (token_in_trade msg_payload) (
    stor_tokens_held cstate))) external_price) in new_rate_lt.
        exact new_rate_lt.
```

```
92      (* the external market prices higher than the contract *)
93      -   is_sp_destruct.
94          intro ext_gt_rx.
95          (* the strategy is to buy low (internally), sell high (externally) *)
96          set (t_y := token_out_trade msg_payload).
97          rewrite msg_is_trade in successful_trade.
98          destruct (trade_entrypoint_check_pf cstate chain ctx msg_payload cstate'
99          acts successful_trade) as [y [_ [r_y [bal_y [_ rate_y]]]]].
100         destruct liquidity as [rate_x [rx_gt_0 [bal_x bal_gt_0]]].
101         destruct (trade_entrypoint_check_2_pf cstate chain ctx msg_payload cstate'
102         acts successful_trade)
103         as [x0 [rx0 [ry0 [k0 [is_x [is_rx [is_ry [is_k gt0]]]]]]]].
104         replace (FMap.find (token_out_trade msg_payload) (stor_rates cstate))
105         with (Some r_y) in is_ry.
106         rewrite tx_trade_in in rate_x, bal_x.
107         replace (FMap.find (token_in_trade msg_payload) (stor_rates cstate))
108         with (Some r_x) in is_rx.
109         replace (FMap.find (token_in_trade msg_payload) (stor_tokens_held cstate))
110         with (Some x) in is_x.
111         replace k0 with (stor_outstanding_tokens cstate) in gt0.
112         replace x0 with x in gt0 by (injection is_x; auto).
113         replace ry0 with r_y in gt0 by (injection is_ry; auto).
114         replace rx0 with r_x in gt0 by (injection is_rx; auto).
115         destruct (arbitrage_gt_pf r_x r_y y (stor_outstanding_tokens cstate) x gt0)
116         as [dx calc_dy].
117         exists dx.
118         intro deplete_store.
119         right.
120         destruct (trade_tokens_held_update_pf cstate chain ctx msg_payload cstate' acts
        successful_trade) as [new_bal_y [_ _]].
121         unfold get_bal, get_rate in new_bal_y.
122         unfold get_bal, t_y.
123         replace (FMap.find (token_out_trade msg_payload) (stor_tokens_held cstate))
124         with (Some y) in new_bal_y.
125         replace (FMap.find (token_out_trade msg_payload) (stor_rates cstate))
126         with (Some r_y) in new_bal_y.
127         replace (FMap.find (token_in_trade msg_payload) (stor_rates cstate))
128         with (Some r_x) in new_bal_y.
129         replace (FMap.find (token_in_trade msg_payload) (stor_tokens_held cstate))
130         with (Some x) in new_bal_y.
131         replace (FMap.find (token_out_trade msg_payload) (stor_tokens_held cstate))
132         with (Some y).
133         rewrite tx_trade_in.
134         replace (FMap.find (token_in_trade msg_payload) (stor_tokens_held cstate))
135         with (Some x).
```

```
        unfold get_rate.
        now replace (FMap.find (token_out_trade msg_payload) (stor_rates cstate))
        with (Some r_y).
Qed.
```

A.1.6 Pooled Consistency

The full Coq proof of pooled consistency from §4.3.8.

```
(* Pooled Consistency:
    The number of outstanding pool tokens is equal to the value, in pool tokens, of all
    constituent tokens held by the contract.

    Mathematically, the sum of all the constituent, pooled tokens, multiplied by their value
    in
    terms of pooled tokens, always equals the total number of outstanding pool tokens.

    This means that the pool token is never under- or over-collateralized, and is similar to
    standard AMMs, where the LP token is always fully backed, representing a percentage of the
    liquidity pool.
*)

(* map over the keys *)
Definition tokens_to_values (rates : FMap token exchange_rate) (tokens_held : FMap token N) :
    list N :=
    List.map
        (fun k =>
            let rate := get_rate k rates in
            let qty_held := get_bal k tokens_held in
            rate * qty_held)
        (FMap.keys rates).

(* some lemmas *)
Lemma token_keys_invariant_pool : forall p new_state new_acts prev_state ctx chain,
    receive contract chain ctx prev_state (Some (pool p)) =
        Ok (new_state, new_acts) ->
    (FMap.keys (stor_rates new_state)) =
    (FMap.keys (stor_rates prev_state)).
Proof.
    intros * receive_some.
    is_sp_destruct.
    (* pool p *)
    pose proof (pool_rates_unchanged_pf prev_state new_state chain ctx p new_acts receive_some
    ) as rates_unchanged.
```

```
33      assert ((stor_rates prev_state) = (stor_rates new_state)) as rates_eq.
34      { now apply FMap.ext_eq. }
35      now rewrite <- rates_eq.
36  Qed.
37
38  Lemma token_keys_invariant_unpool : forall u new_state new_acts prev_state ctx chain,
39      receive contract chain ctx prev_state (Some (unpool u)) =
40          Ok (new_state, new_acts) ->
41      (FMap.keys (stor_rates new_state)) =
42      (FMap.keys (stor_rates prev_state)).
43  Proof.
44      intros * receive_some.
45      is_sp_destruct.
46      (* unpool u *)
47      pose proof (unpool_rates_unchanged_pf prev_state new_state chain ctx u new_acts
        receive_some) as unrates_unchanged.
48      assert ((stor_rates prev_state) = (stor_rates new_state)) as rates_eq.
49      { now apply FMap.ext_eq. }
50      now rewrite <- rates_eq.
51  Qed.
52
53  Lemma token_keys_invariant_trade : forall t new_state new_acts prev_state ctx chain,
54      receive contract chain ctx prev_state (Some (trade t)) =
55          Ok (new_state, new_acts) ->
56      Permutation
57      (FMap.keys (stor_rates new_state))
58      (FMap.keys (stor_rates prev_state)).
59  Proof.
60      intros * receive_some.
61      is_sp_destruct.
62      (* trade t *)
63      destruct (trade_update_rates_formula_pf prev_state chain ctx t new_state new_acts
        receive_some) as [tokens_neq update_rates_formula].
64      destruct update_rates_formula as [old_rate_in other_rates_unchanged].
65      (* ... *)
66      pose proof (trade_entrypoint_check_pf prev_state
67      chain ctx t new_state new_acts receive_some)
68      as trade_check.
69      do 3 destruct trade_check as [* trade_check].
70      destruct trade_check as [_ trade_check].
71      destruct trade_check as [trade_check _].
72      clear x1 x.
73      pose proof (FMap.keys_already
74          (token_in_trade t) x0
75          (calc_rx'
```

```
76              (get_rate (token_in_trade t)
77                  (stor_rates prev_state))
78              (get_rate (token_out_trade t)
79                  (stor_rates prev_state))
80              (qty_trade t)
81              (stor_outstanding_tokens prev_state)
82              (get_bal (token_in_trade t)
83                  (stor_tokens_held prev_state)))
84          (stor_rates prev_state) trade_check)
85      as keys_permuted.
86      destruct (trade_update_rates_pf prev_state chain ctx t new_state new_acts receive_some)
87      as [_ update_rates].
88      simpl in update_rates.
89      (* rewrite <- update_rates in H6 *)
90      replace (FMap.add (token_in_trade t)
91          (calc_rx' (get_rate (token_in_trade t) (stor_rates prev_state))
92          (get_rate (token_out_trade t) (stor_rates prev_state)) (qty_trade t)
93          (stor_outstanding_tokens prev_state) (get_bal (token_in_trade t) (stor_tokens_held
        prev_state)))
94          (stor_rates prev_state))
95      with (stor_rates new_state)
96      in keys_permuted.
97      exact keys_permuted.
98  Qed.
99
100 Definition suml l := fold_right N.add 0 l.
101
102 (* and suml commutes with the permutation *)
103 Lemma map_extract { A B : Type } : forall (a : A) (l : list A) (f : A -> B),
104     map f (a :: l) =
105     (f a) :: map f l.
106 Proof. auto. Qed.
107
108 Lemma suml_extract : forall a l,
109     suml (a :: l) =
110     a + (suml l).
111 Proof. auto. Qed.
112
113 Lemma in_map : forall { A B : Type } (f : A -> B) l a,
114     In a l -> In (f a) (map f l).
115 Proof.
116     intros * H_in.
117     induction l as [|x l IH]; auto.
118     simpl. simpl in H_in. destruct H_in.
119     + left. now subst.
```

```
    + right. now apply IH.
Qed.

Lemma remove_permute : forall { A : Type } a b (l : list A)
    (eq_dec : forall a b, { a = b } + { a <> b }),
    a <> b ->
    (a :: remove eq_dec b l) = (remove eq_dec b (a :: l)).
Proof.
    intros * a_neq_b.
    unfold remove.
    apply not_eq_sym in a_neq_b.
    now destruct (eq_dec b a); try contradiction.
Qed.

Lemma nodup_permute : forall { A : Type } a (l : list A)
    (eq_dec : forall a b, { a = b } + { a <> b }),
    NoDup l ->
    In a l ->
    Permutation l (a :: remove eq_dec a l).
Proof.
    intros * nodup_l in_a_l.
    induction l; cbn in in_a_l; try contradiction.
    (* establish NoDup l *)
    apply NoDup_cons_iff in nodup_l as nodup_l'.
    destruct nodup_l' as [not_in_a0_l nodup_l'].
    destruct in_a_l as [a0_eq_a | in_a_l].
    (* a0 = a *)
    -   rewrite a0_eq_a in *.
        replace (remove eq_dec a (a :: l))
        with l.
        2:{ unfold remove.
            destruct (eq_dec a a); try contradiction.
            pose proof (notin_remove eq_dec l a not_in_a0_l) as remove_eq.
            unfold remove in remove_eq.
            now rewrite remove_eq. }
        reflexivity.
    (* In a l *)
    -   destruct (eq_dec a a0) as [a_eq_a0 | a_neq_a0].
        +   rewrite <- a_eq_a0 in *.
            replace (remove eq_dec a (a :: l))
            with l.
            2:{ unfold remove.
                destruct (eq_dec a a); try contradiction. }
            apply Permutation_refl.
        +   rewrite <- remove_permute; try exact (not_eq_sym a_neq_a0).
```

```
            pose (perm_swap a a0 (remove eq_dec a l)) as perm_intermediate1.
            pose (perm_skip a0 (IHl nodup_l' in_a_l)) as perm_intermediate2.
            apply (perm_trans perm_intermediate2 perm_intermediate1).
Qed.

Lemma not_in_split : forall {A : Type} (a : A) (l1 l2 : list A),
  ~ In a (l1 ++ l2) -> ~ In a l1 /\ ~ In a l2.
Proof.
  intros A a l1 l2 H_nin.
  split; intros Hcontra; apply H_nin; apply in_or_app; [left|right]; assumption.
Qed.

Lemma nodup_remove_mid : forall { A : Type } a (l1 l2 : list A)
(eq_dec : forall a b, { a = b } + { a <> b }),
    NoDup (l1 ++ l2) /\ ~ In a (l1 ++ l2) ->
    ((remove eq_dec a (l1 ++ a :: l2)) = l1 ++ l2)%list.
Proof.
    intros * hyp.
    destruct hyp as [nodup_concat not_in].
    rewrite remove_app.
    apply not_in_split in not_in. destruct not_in as [not_in_l1 not_in_l2].
    rewrite (notin_remove eq_dec l1 a not_in_l1).
    rewrite remove_cons.
    now rewrite (notin_remove eq_dec l2 a not_in_l2).
Qed.

Lemma nodup_remove : forall { A : Type } a (l : list A)
    (eq_dec : forall a b, { a = b } + { a <> b }),
    NoDup l ->
    NoDup (remove eq_dec a l).
Proof.
    intros * nodup_l.
    destruct (in_dec eq_dec a l).
    -   apply in_split in i.
        destruct i as [l1 [l2 i]].
        rewrite i.
        rewrite i in nodup_l.
        apply NoDup_remove in nodup_l.
        rewrite nodup_remove_mid.
        now destruct nodup_l.
        assumption.
    -   now rewrite (notin_remove eq_dec l a n).
Qed.

Lemma remove_permute_2 : forall { A : Type } a (l1 l2 : list A)
```

```
210     (eq_dec : forall a b, { a = b } + { a <> b }),
211     NoDup l1 ->
212     NoDup l2 ->
213     Permutation l1 l2 ->
214     Permutation (remove eq_dec a l1) (remove eq_dec a l2).
215 Proof.
216     intros * nodup_l1 nodup_l2 perm.
217     destruct (in_dec eq_dec a l1) as [in_l1 | not_in_l1].
218     (* a is in l1 *)
219     -   (* prove in l2 *)
220         pose proof (Permutation_in a perm in_l1)
221         as in_l2.
222         (* decompose l1 and l2 because in and nodup *)
223         apply in_split in in_l1 as l1_decompose.
224         destruct l1_decompose as [l1' [l1'' l1_split]].
225         apply in_split in in_l2 as l2_decompose.
226         destruct l2_decompose as [l2' [l2'' l2_split]].
227         rewrite l1_split, l2_split.
228         (* remove a explicitly *)
229         rewrite l1_split in nodup_l1.
230         rewrite l2_split in nodup_l2.
231         apply NoDup_remove in nodup_l1, nodup_l2.
232         (* retain permutation *)
233         repeat rewrite nodup_remove_mid; try assumption.
234         apply (Permutation_app_inv l1' l1'' l2' l2'' a).
235         rewrite <- l1_split.
236         now rewrite <- l2_split.
237     (* a not in l1 *)
238     -   (* prove not in l2 *)
239         destruct (in_dec eq_dec a l2) as [in_l2 | not_in_l2].
240         apply Permutation_sym in perm.
241         pose proof (Permutation_in a perm in_l2)
242         as in_l1.
243         contradiction.
244         (* show equality with remove *)
245         rewrite (notin_remove eq_dec l1 a not_in_l1).
246         now rewrite (notin_remove eq_dec l2 a not_in_l2).
247 Qed.
248
249 Lemma mapped_eq : forall (A B : Type) (f f' : A -> B) (l : list A),
250     (forall a, In a l -> f a = f' a) ->
251     map f l = map f' l.
252 Proof.
253     intros * H_in.
254     induction l; auto.
```

```
255     repeat rewrite map_extract.
256     rewrite (H_in a (in_eq a l)).
257     assert (forall a : A, In a l -> f a = f' a) as H_in_ind.
258     {   intros * in_a0.
259         exact (H_in a0 (in_cons a a0 l in_a0)). }
260     now rewrite (IHl H_in_ind).
261 Qed.
262
263 Lemma suml_permute_commutes : forall l l',
264     Permutation l l' -> suml l = suml l'.
265 Proof.
266     intros. induction H7; auto.
267     -   repeat rewrite (suml_extract x).
268         now rewrite IHPermutation.
269     -   rewrite (suml_extract y). rewrite (suml_extract x).
270         rewrite (suml_extract x). rewrite (suml_extract y).
271         repeat rewrite N.add_assoc.
272         now rewrite (N.add_comm x y).
273     -   rewrite IHPermutation1.
274         now rewrite IHPermutation2.
275 Qed.
276
277 (** Theorem (Pooled Consistency):
278     The following equation always holds:
279         Sum_{t_x} r_x x = k
280 *)
281
282 Theorem pooled_consistency bstate caddr :
283     reachable bstate ->
284     env_contracts bstate caddr = Some (contract : WeakContract) ->
285     exists (cstate : State),
286     contract_state bstate caddr = Some cstate /\
287     (* The sum of all the constituent, pooled tokens, multiplied by their value in terms
288     of pooled tokens, always equals the total number of outstanding pool tokens. *)
289     suml (tokens_to_values (stor_rates cstate) (stor_tokens_held cstate)) =
290         (stor_outstanding_tokens cstate).
291 Proof.
292     contract_induction; intros; try exact IH.
293     is_sp_destruct.
294     (* Please establish the invariant after deployment of the contract *)
295     -   (* we use the fact that balances initialize at zero *)
296         pose proof (initialized_with_zero_balance_pf chain ctx setup result init_some)
297         as zero_bal.
298         pose proof (initialized_with_zero_outstanding_pf chain ctx setup result init_some)
299         as zero_outstanding.
```

```
300         rewrite zero_outstanding.
301         clear zero_outstanding.
302         unfold tokens_to_values.
303         induction (FMap.keys (stor_rates result)); auto.
304         rewrite map_cons.
305         rewrite zero_bal.
306         now rewrite N.mul_0_r.
307     (* Please reestablish the invariant after a nonrecursive call *)
308     -   destruct msg.
309         (* first, msg = None *)
310         2:{ is_sp_destruct.
311             destruct (none_fails_pf prev_state chain ctx) as [err failed].
312             rewrite receive_some in failed.
313             inversion failed. }
314         (* msg = Some m *)
315         is_sp_destruct.
316         destruct (msg_destruct_pf m) as [pool | [unpool | [trade | other]]].
317         (* m = pool *)
318         +   destruct pool as [p msg_pool].
319             rewrite msg_pool in receive_some.
320             (* understand how tokens_held has changed *)
321             destruct (pool_increases_tokens_held_pf prev_state chain ctx p new_state new_acts
322             receive_some) as [bal_update bals_unchanged].
323             (* show all rates are the same *)
324             pose proof (pool_rates_unchanged_pf prev_state new_state chain ctx p new_acts
325                 receive_some) as rates_update.
326             (* show how the outstanding tokens update *)
327             pose proof (pool_outstanding_pf prev_state new_state chain ctx p new_acts
328                 receive_some) as out_update.
329             simpl in out_update.
330             (* because the rates are unchanged, summing over them is unchanged *)
331             assert (tokens_to_values (stor_rates new_state)  (stor_tokens_held new_state) =
332                     tokens_to_values (stor_rates prev_state) (stor_tokens_held new_state))
333             as rates_unchanged.
334             {   unfold tokens_to_values.
335                 rewrite (token_keys_invariant_pool p new_state new_acts prev_state ctx chain
    receive_some).
336                 apply map_ext.
337                 intro t.
338                 unfold get_rate.
339                 now rewrite rates_update. }
340             rewrite rates_unchanged. clear rates_unchanged.
341             (* now separate the sum *)
342             assert (suml (tokens_to_values (stor_rates prev_state) (stor_tokens_held new_state
    )) =
```

```
343                     suml (tokens_to_values (stor_rates prev_state) (stor_tokens_held
    prev_state)) + get_rate (token_pooled p) (stor_rates prev_state) * qty_pooled p)
344             as separate_sum.
345             {   unfold tokens_to_values.
346                 (* some simplifying notation *)
347                 set (keys_minus_p := (remove token_eq_dec (token_pooled p)
348                     (FMap.keys (stor_rates prev_state)))).
349                 set (tokens_to_values_new := fun k : token =>
350                     get_rate k (stor_rates prev_state) * get_bal k (stor_tokens_held new_state
    )).
351                 set (tokens_to_values_prev := fun (k : token) =>
352                     get_rate k (stor_rates prev_state) * get_bal k (stor_tokens_held
    prev_state)).
353                 (* 1. show there's a Permutation with (token_pooled p) at the front *)
354                 assert (Permutation
355                     (FMap.keys (stor_rates prev_state))
356                     ((token_pooled p) :: keys_minus_p))
357                 as keys_permuted.
358                 {   unfold keys_minus_p.
359                     (* In (token_pooled p) (FMap.keys (stor_rates prev_state)) *)
360                     destruct (pool_entrypoint_check_pf prev_state new_state chain
361                     ctx p new_acts receive_some) as [r_x rate_exists_el].
362                     apply FMap.In_elements in rate_exists_el.
363                     pose proof (in_map fst
364                         (FMap.elements (stor_rates prev_state))
365                         (token_pooled p, r_x)
366                         rate_exists_el)
367                     as rate_exists_key.
368                     clear rate_exists_el.
369                     simpl in rate_exists_key.
370                     (* recall NoDup keys *)
371                     pose proof (FMap.NoDup_keys (stor_rates prev_state))
372                     as nodup_keys.
373                     (* since NoDup keys, you can permute *)
374                     exact (nodup_permute
375                         (token_pooled p)
376                         (FMap.keys (stor_rates prev_state))
377                         token_eq_dec
378                         nodup_keys
379                         rate_exists_key). }
380                 (* Use the permutation to rewrite the LHS first, extracting (token_pooled p)
    *)
381                 pose proof
382                     (Permutation_map tokens_to_values_new)
383                         (* list 1 *)
```

```
384                         (FMap.keys (stor_rates prev_state))
385                         (* list 2 *)
386                         ((token_pooled p) :: keys_minus_p)
387                         (* the previous permutation *)
388                         keys_permuted
389                 as lhs_permute_map.
390                 (* now that we have that permutation, rewrite in the suml *)
391                 rewrite (suml_permute_commutes
392                     (map tokens_to_values_new (FMap.keys (stor_rates prev_state)))
393                     (map tokens_to_values_new (token_pooled p :: keys_minus_p))
394                     lhs_permute_map).
395                 (* now extract (token_pooled p) *)
396                 rewrite (map_extract
397                     (token_pooled p)
398                     keys_minus_p
399                     tokens_to_values_new).
400                 rewrite (suml_extract (tokens_to_values_new (token_pooled p))).
401                 (* Now separate the sum on the RHS *)
402                 pose proof
403                     (Permutation_map tokens_to_values_prev)
404                         (* list 1 *)
405                         (FMap.keys (stor_rates prev_state))
406                         (* list 2 *)
407                         ((token_pooled p) :: keys_minus_p)
408                         (* the previous permutation *)
409                         keys_permuted
410                 as rhs_permute_map.
411                 (* now that we have that permutation, rewrite in the suml *)
412                 rewrite (suml_permute_commutes
413                     (map tokens_to_values_prev (FMap.keys (stor_rates prev_state)))
414                     (map tokens_to_values_prev (token_pooled p :: keys_minus_p))
415                     rhs_permute_map).
416                 (* now extract (token_pooled p) *)
417                 rewrite (map_extract
418                     (token_pooled p)
419                     keys_minus_p
420                     tokens_to_values_prev).
421                 rewrite (suml_extract (tokens_to_values_prev (token_pooled p))).
422
423                 (* Now proceed in two parts *)
424                 (* first the sum of all the unchanged keys *)
425                 assert
426                     (suml (map tokens_to_values_new keys_minus_p) =
427                      suml (map tokens_to_values_prev keys_minus_p))
428                 as old_keys_eq.
```

```
429                 {   assert (forall t, In t keys_minus_p -> t <> (token_pooled p))
430                     as t_neq.
431                     {   intros * H_in.
432                         unfold keys_minus_p in H_in.
433                         apply in_remove in H_in.
434                         destruct H_in as [_ res].
435                         exact res. }
436                     assert (forall t, In t keys_minus_p ->
437                         tokens_to_values_new t = tokens_to_values_prev t)
438                     as tokens_to_vals_eq.
439                     {   intros * H_in.
440                         apply t_neq in H_in.
441                         apply bals_unchanged in H_in.
442                         unfold tokens_to_values_new, tokens_to_values_prev.
443                         now rewrite H_in. }
444                     now rewrite (mapped_eq
445                         token N
446                         tokens_to_values_new
447                         tokens_to_values_prev
448                         keys_minus_p
449                         tokens_to_vals_eq). }
450
451                 (* then to the key that did change *)
452                 assert
453                     (tokens_to_values_new (token_pooled p) =
454                     tokens_to_values_prev (token_pooled p) +
455                     get_rate (token_pooled p) (stor_rates prev_state) * qty_pooled p)
456                 as new_key_eq.
457                 {   unfold tokens_to_values_new, tokens_to_values_prev.
458                     rewrite bal_update.
459                     apply N.mul_add_distr_l. }
460                 rewrite old_keys_eq. rewrite new_key_eq.
461                 rewrite <- N.add_assoc.
462                 rewrite (N.add_comm
463                     (get_rate (token_pooled p) (stor_rates prev_state) * qty_pooled p)
464                     (suml (map tokens_to_values_prev keys_minus_p))).
465                 now rewrite N.add_assoc. }
466             rewrite separate_sum. clear separate_sum.
467             rewrite out_update.
468             now rewrite IH.
469         (* m = unpool *)
470         +   destruct unpool as [u msg_unpool].
471             rewrite msg_unpool in receive_some.
472             (* understand how tokens_held has changed *)
```

```
473             destruct (unpool_decreases_tokens_held_pf prev_state chain ctx u new_state
        new_acts
474             receive_some) as [bal_update bals_unchanged].
475             (* show all rates are the same *)
476             pose proof (unpool_rates_unchanged_pf prev_state new_state chain ctx u new_acts
477                 receive_some) as rates_update.
478             (* show how the outstanding tokens update *)
479             pose proof (unpool_outstanding_pf prev_state new_state chain ctx u new_acts
480                 receive_some) as out_update.
481             simpl in out_update.
482             rewrite out_update. clear out_update.
483             (* because the rates are unchanged, summing over them is unchanged *)
484             assert (tokens_to_values (stor_rates new_state)  (stor_tokens_held new_state) =
485                     tokens_to_values (stor_rates prev_state) (stor_tokens_held new_state))
486             as rates_unchanged.
487             {   unfold tokens_to_values.
488                 rewrite (token_keys_invariant_unpool u new_state new_acts prev_state ctx
489                 chain receive_some).
490                 apply map_ext.
491                 intro t.
492                 unfold get_rate.
493                 now rewrite rates_update. }
494             rewrite rates_unchanged. clear rates_unchanged.
495             (* now separate the sum *)
496             assert (suml (tokens_to_values (stor_rates prev_state) (stor_tokens_held new_state
        ))
497             = suml (tokens_to_values (stor_rates prev_state) (stor_tokens_held prev_state)) -
498             qty_unpooled u)
499             as separate_sum.
500             {   unfold tokens_to_values.
501                 (* some simplifying notation *)
502                 set (keys_minus_u := (remove token_eq_dec (token_unpooled u)
503                     (FMap.keys (stor_rates prev_state)))).
504                 set (tokens_to_values_new := fun k : token =>
505                     get_rate k (stor_rates prev_state) * get_bal k (stor_tokens_held new_state
        )).
506                 set (tokens_to_values_prev := fun k : token =>
507                     get_rate k (stor_rates prev_state) * get_bal k (stor_tokens_held
        prev_state)).
508                 (* 1. show there's a Permutation with (token_pooled p) at the front *)
509                 assert (Permutation
510                     (FMap.keys (stor_rates prev_state))
511                     ((token_unpooled u) :: keys_minus_u))
512                 as keys_permuted.
513                 {   unfold keys_minus_u.
```

```
514                     (* In (token_pooled p) (FMap.keys (stor_rates prev_state)) *)
515                     destruct (unpool_entrypoint_check_pf prev_state new_state chain ctx u
516                     new_acts receive_some) as [r_x rate_exists_el].
517                     apply FMap.In_elements in rate_exists_el.
518                     pose proof (in_map fst
519                         (FMap.elements (stor_rates prev_state))
520                         (token_unpooled u, r_x)
521                         rate_exists_el)
522                     as rate_exists_key. clear rate_exists_el.
523                     simpl in rate_exists_key.
524                     (* recall NoDup keys *)
525                     pose proof (FMap.NoDup_keys (stor_rates prev_state))
526                     as nodup_keys.
527                     (* since NoDup keys, you can permute *)
528                     exact (nodup_permute
529                         (token_unpooled u)
530                         (FMap.keys (stor_rates prev_state))
531                         token_eq_dec
532                         nodup_keys
533                         rate_exists_key). }
534                 (* Use the permutation to rewrite the LHS first, extracting (token_pooled p)
        *)
535                 pose proof
536                     (Permutation_map tokens_to_values_new)
537                         (* list 1 *)
538                         (FMap.keys (stor_rates prev_state))
539                         (* list 2 *)
540                         ((token_unpooled u) :: keys_minus_u)
541                         (* the previous permutation *)
542                         keys_permuted
543                 as lhs_permute_map.
544                 (* now that we have that permutation, rewrite in the suml *)
545                 rewrite (suml_permute_commutes
546                     (map tokens_to_values_new (FMap.keys (stor_rates prev_state)))
547                     (map tokens_to_values_new (token_unpooled u :: keys_minus_u))
548                     lhs_permute_map).
549                 (* now extract (token_pooled p) *)
550                 rewrite (map_extract
551                     (token_unpooled u)
552                     keys_minus_u
553                     tokens_to_values_new).
554                 rewrite (suml_extract (tokens_to_values_new (token_unpooled u))).
555                 (* Now separate the sum on the RHS *)
556                 pose proof
557                     (Permutation_map tokens_to_values_prev)
```

```
                        (* list 1 *)
                        (FMap.keys (stor_rates prev_state))
                        (* list 2 *)
                        ((token_unpooled u) :: keys_minus_u)
                        (* the previous permutation *)
                        keys_permuted
                as rhs_permute_map.
                (* now that we have that permutation, rewrite in the suml *)
                rewrite (suml_permute_commutes
                    (map tokens_to_values_prev (FMap.keys (stor_rates prev_state)))
                    (map tokens_to_values_prev (token_unpooled u :: keys_minus_u))
                    rhs_permute_map).
                (* now extract (token_pooled p) *)
                rewrite (map_extract
                    (token_unpooled u)
                    keys_minus_u
                    tokens_to_values_prev).
                rewrite (suml_extract (tokens_to_values_prev (token_unpooled u))).

                (* Now proceed in two parts *)
                (* first the sum of all the unchanged keys *)
                assert
                    (suml (map tokens_to_values_new keys_minus_u) =
                     suml (map tokens_to_values_prev keys_minus_u))
                as old_keys_eq.
                {   assert (forall t, In t keys_minus_u -> t <> (token_unpooled u))
                    as t_neq.
                    {   intros * H_in.
                        unfold keys_minus_u in H_in.
                        apply in_remove in H_in.
                        destruct H_in as [_ res].
                        exact res. }
                    assert (forall t, In t keys_minus_u ->
                        tokens_to_values_new t = tokens_to_values_prev t)
                    as tokens_to_vals_eq.
                    {   intros * H_in.
                        apply t_neq in H_in.
                        apply bals_unchanged in H_in.
                        unfold tokens_to_values_new, tokens_to_values_prev.
                        now rewrite H_in. }
                    now rewrite (mapped_eq
                        token N
                        tokens_to_values_new
                        tokens_to_values_prev
                        keys_minus_u
```

```
603                         tokens_to_vals_eq). }
604                 rewrite old_keys_eq. clear old_keys_eq.
605 
606                 (* then to the key that did change *)
607                 assert
608                     (tokens_to_values_new (token_unpooled u) =
609                     tokens_to_values_prev (token_unpooled u) - qty_unpooled u)
610                 as new_key_eq.
611                 {   unfold tokens_to_values_new, tokens_to_values_prev.
612                     (* get the calculation from the specification *)
613                     rewrite bal_update.
614                     rewrite (N.mul_sub_distr_l
615                         (get_bal (token_unpooled u) (stor_tokens_held prev_state))
616                         (calc_rx_inv (get_rate (token_unpooled u) (stor_rates prev_state))
617                         (qty_unpooled u)) (get_rate (token_unpooled u) (stor_rates prev_state)
    )).
618                     now rewrite rates_balance_pf. }
619                 rewrite new_key_eq.
620 
621                 (* this is an additional condition required here so that commutativity applies
     *)
622                 assert (qty_unpooled u <= tokens_to_values_prev (token_unpooled u))
623                 as unpooled_leq. {
624                     now unfold tokens_to_values_prev. }
625                 now rewrite (N.add_sub_swap
626                     (tokens_to_values_prev (token_unpooled u))
627                     (suml (map tokens_to_values_prev keys_minus_u))
628                     (qty_unpooled u)
629                     unpooled_leq). }
630             rewrite separate_sum. clear separate_sum.
631             now rewrite IH.
632         (* m = trade *)
633         +   destruct trade as [t msg_trade].
634             rewrite msg_trade in receive_some.
635             (* understand how tokens_held has changed *)
636             destruct (trade_tokens_held_update_pf prev_state chain ctx t new_state new_acts
637             receive_some)
638             as [tokens_held_update_ty [tokens_held_update_tx tokens_held_update_tz]].
639             (* how calling trade updates rates *)
640             destruct (trade_update_rates_formula_pf prev_state chain ctx t new_state new_acts
641             receive_some) as [tx_neq_ty [rx_change rz_unchanged]].
642             (* how outstanding_tokens has changed by calling trade *)
643             pose proof (trade_outstanding_update_pf prev_state chain ctx t new_state new_acts
644                 receive_some) as out_update.
645             rewrite out_update.
```

```
646             clear out_update.
647
648             (* LHS equals sum over tokens except for tx and ty, plus r_x' * etc ,
649             minus r_y * delta_y *)
650
651             (* some notation *)
652             unfold tokens_to_values.
653             set (keys_minus_trade_in :=
654                 (remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates new_state)))).
655             set (keys_minus_trades :=
656                 (remove token_eq_dec (token_out_trade t) keys_minus_trade_in)).
657             set (tokens_to_values_new := fun k : token =>
658                 get_rate k (stor_rates new_state) * get_bal k (stor_tokens_held new_state)).
659
660             (* LHS *)
661             (* permute FMap.keys (stor_tokens_held new_state) to have tx :: ty :: keys' *)
662                 (* for these, rates and balances are all the same as before *)
663             assert (Permutation
664                 (FMap.keys (stor_rates new_state))
665                 ((token_in_trade t) :: (token_out_trade t) :: keys_minus_trades))
666             as keys_permuted.
667             {   unfold keys_minus_trades.
668                 unfold keys_minus_trade_in.
669                 assert (Permutation
670                     (FMap.keys (stor_rates new_state))
671                     (token_in_trade t ::
672                     remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates new_state)))
    )
673                 as permute_1.
674                 {   apply (nodup_permute
675                         (token_in_trade t)
676                         (FMap.keys (stor_rates new_state))
677                         token_eq_dec
678                         (FMap.NoDup_keys (stor_rates new_state))).
679                     assert (exists x, FMap.find (token_in_trade t) (stor_rates new_state)
680                     = Some x) as rate_exists.
681                     {   destruct (trade_update_rates_pf prev_state chain ctx t new_state
682                         new_acts receive_some) as [_ update_rates].
683                         simpl in update_rates.
684                         rewrite update_rates.
685                         set (x := (calc_rx' (get_rate (token_in_trade t) (stor_rates
    prev_state))
686                         (get_rate (token_out_trade t) (stor_rates prev_state)) (qty_trade t)
687                         (stor_outstanding_tokens prev_state) (get_bal (token_in_trade t)
688                         (stor_tokens_held prev_state)))).
```

```
689                         exists x.
690                         apply FMap.find_add. }
691                     destruct rate_exists as [x rate_exists].
692                     apply (FMap.In_elements (token_in_trade t) x (stor_rates new_state))
693                     in rate_exists.
694                     now apply (in_map fst
695                         (FMap.elements (stor_rates new_state))
696                         (token_in_trade t, x)). }
697                 assert (Permutation
698                     (token_in_trade t ::
699                     remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates new_state)))
700                     ((token_out_trade t) :: (token_in_trade t) ::
701                     remove token_eq_dec (token_out_trade t)
702                     (remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates new_state))
       )))
703                 as permute_2.
704                 {   assert (Permutation
705                         (token_out_trade t :: token_in_trade t ::
706                             remove token_eq_dec (token_out_trade t)
707                             (remove token_eq_dec (token_in_trade t)
708                                 (FMap.keys (stor_rates new_state))))
709                         (token_out_trade t ::
710                             remove token_eq_dec (token_out_trade t)
711                             (token_in_trade t :: remove token_eq_dec (token_in_trade t)
712                                 (FMap.keys (stor_rates new_state)))))
713                     as permute_step_1.
714                     {   assert (Permutation
715                             (token_in_trade t
716                             :: remove token_eq_dec (token_out_trade t) (remove token_eq_dec
717                             (token_in_trade t) (FMap.keys (stor_rates new_state))))
718                             (remove token_eq_dec (token_out_trade t)
719                             (token_in_trade t :: remove token_eq_dec (token_in_trade t)
720                             (FMap.keys (stor_rates new_state)))))
721                         as inner_permute.
722                         {   set (rates_remove_in := (remove token_eq_dec (token_in_trade t)
723                             (FMap.keys (stor_rates new_state)))).
724                             rewrite remove_permute; try assumption.
725                             apply Permutation_refl. }
726                         exact (perm_skip (token_out_trade t) inner_permute). }
727                     assert (Permutation
728                     (token_in_trade t ::
729                     remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates new_state)))
730                     (token_out_trade t ::
731                             remove token_eq_dec (token_out_trade t)
732                             (token_in_trade t :: remove token_eq_dec (token_in_trade t)
```

```
                                (FMap.keys (stor_rates new_state)))))
                    as permute_step_2.
                    {   apply nodup_permute.
                        (* prove the NoDup result *)
                        -   apply NoDup_cons.
                            +   apply remove_In.
                            +   apply nodup_remove.
                                exact (FMap.NoDup_keys (stor_rates new_state)).
                        (* prove the In result *)
                        -   apply in_cons.
                            apply in_in_remove.
                            +   now apply not_eq_sym.
                            +   assert (exists x,
                                    FMap.find (token_out_trade t) (stor_rates new_state)
                                    = Some x)
                                as rate_exists.
                                {   pose proof (trade_entrypoint_check_pf prev_state
                                    chain ctx t new_state new_acts receive_some)
                                    as trade_check.
                                    (* get rate from prev_state *)
                                    do 3 destruct trade_check as [* trade_check].
                                    destruct trade_check as [_ trade_check].
                                    destruct trade_check as [_ prev_rate].
                                    clear x x0.
                                    (* get the rate from the new state *)
                                    rewrite <- (rz_unchanged (token_out_trade t)
                                    (not_eq_sym tx_neq_ty)) in prev_rate.
                                    now exists x1. }
                                destruct rate_exists as [x rate_exists].
                                apply FMap.In_elements in rate_exists.
                                now apply (in_map fst
                                    (FMap.elements (stor_rates new_state))
                                    (token_out_trade t, x)). }
                    exact (Permutation_trans permute_step_2 (Permutation_sym permute_step_1)).
        }
                exact (Permutation_trans permute_1
                    (Permutation_trans permute_2
                        (perm_swap (token_in_trade t) (token_out_trade t)
                        (remove token_eq_dec (token_out_trade t)
                        (remove token_eq_dec (token_in_trade t)
                        (FMap.keys (stor_rates new_state)))))))). }
            (* Now separate LHS into ^^, plus new rate * t_x, minus rate * t_y *)
            pose proof
                (Permutation_map tokens_to_values_new)
                    (* list 1 *)
```

```
                    (FMap.keys (stor_rates new_state))
                    (* list 2 *)
                    ((token_in_trade t) :: (token_out_trade t) :: keys_minus_trades)
                    (* the previous permutation *)
                    keys_permuted
            as lhs_permute_map.
            (* now that we have that permutation, we can decompose the suml *)
            rewrite (suml_permute_commutes
                (map tokens_to_values_new (FMap.keys (stor_rates new_state)))
                (map tokens_to_values_new
                    ((token_in_trade t) :: (token_out_trade t) :: keys_minus_trades))
                lhs_permute_map).
            (* now extract (token_in_trade t) and (token_out_trade t) *)
            rewrite (map_extract
                (token_in_trade t)
                (token_out_trade t :: keys_minus_trades)
                tokens_to_values_new).
            rewrite (map_extract
                (token_out_trade t)
                keys_minus_trades
                tokens_to_values_new).
            rewrite (suml_extract (tokens_to_values_new (token_in_trade t))).
            rewrite (suml_extract (tokens_to_values_new (token_out_trade t))).

            (* permute FMap.keys for the OLD suml to MANIPULATE the inductive hypothesis IH *)
            unfold tokens_to_values in IH.
            set (keys_minus_trade_in_prev :=
                (remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates prev_state)))).
            set (keys_minus_trades_prev :=
                (remove token_eq_dec (token_out_trade t) keys_minus_trade_in_prev)).
            set (tokens_to_values_prev := fun k : token =>
                get_rate k (stor_rates prev_state) * get_bal k (stor_tokens_held prev_state))
            in IH.

            assert (Permutation
                (FMap.keys (stor_rates prev_state))
                ((token_in_trade t) :: (token_out_trade t) :: keys_minus_trades_prev))
            as keys_permuted_prev.
            {   unfold keys_minus_trades_prev.
                unfold keys_minus_trade_in_prev.
                assert (Permutation
                    (FMap.keys (stor_rates prev_state))
                    (token_in_trade t ::
                    remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates prev_state))
        ))
```

```
821                 as permute_1.
822                 {   apply (nodup_permute
823                         (token_in_trade t)
824                         (FMap.keys (stor_rates prev_state))
825                         token_eq_dec
826                         (FMap.NoDup_keys (stor_rates prev_state))).
827                     assert (exists x, FMap.find (token_in_trade t) (stor_rates prev_state)
828                     = Some x) as rate_exists.
829                     {   pose proof (trade_entrypoint_check_pf prev_state
830                             chain ctx t new_state new_acts receive_some)
831                         as trade_check.
832                         do 3 destruct trade_check as [* trade_check].
833                         destruct trade_check as [_ trade_check].
834                         destruct trade_check as [prev_rate _].
835                         clear x x1.
836                         now exists x0. }
837                     destruct rate_exists as [x rate_exists].
838                     apply (FMap.In_elements (token_in_trade t) x (stor_rates prev_state))
839                     in rate_exists.
840                     now apply (in_map fst
841                         (FMap.elements (stor_rates prev_state))
842                         (token_in_trade t, x)). }
843                 assert (Permutation
844                     (token_in_trade t ::
845                     remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates prev_state))
       )
846                     ((token_out_trade t) :: (token_in_trade t) ::
847                     remove token_eq_dec (token_out_trade t)
848                     (remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates prev_state)
       ))))
849                 as permute_2.
850                 {   assert (Permutation
851                         (token_out_trade t :: token_in_trade t ::
852                             remove token_eq_dec (token_out_trade t)
853                             (remove token_eq_dec (token_in_trade t)
854                                 (FMap.keys (stor_rates prev_state))))
855                         (token_out_trade t ::
856                             remove token_eq_dec (token_out_trade t)
857                             (token_in_trade t :: remove token_eq_dec (token_in_trade t)
858                                 (FMap.keys (stor_rates prev_state)))))
859                     as permute_step_1.
860                     {   assert (Permutation
861                             (token_in_trade t
862                             :: remove token_eq_dec (token_out_trade t) (remove token_eq_dec
863                             (token_in_trade t) (FMap.keys (stor_rates prev_state))))
```

```
                            (remove token_eq_dec (token_out_trade t)
                            (token_in_trade t :: remove token_eq_dec (token_in_trade t)
                            (FMap.keys (stor_rates prev_state)))))
                        as inner_permute.
                        {   set (rates_remove_in := (remove token_eq_dec (token_in_trade t)
                            (FMap.keys (stor_rates prev_state)))).
                            rewrite remove_permute; try assumption.
                            apply Permutation_refl. }
                        exact (perm_skip (token_out_trade t) inner_permute). }
                    assert (Permutation
                    (token_in_trade t ::
                    remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates prev_state))
    )
                    (token_out_trade t ::
                            remove token_eq_dec (token_out_trade t)
                            (token_in_trade t :: remove token_eq_dec (token_in_trade t)
                                (FMap.keys (stor_rates prev_state)))))
                    as permute_step_2.
                    {   apply nodup_permute.
                        (* prove the NoDup result *)
                        -   apply NoDup_cons.
                            +   apply remove_In.
                            +   apply nodup_remove.
                                exact (FMap.NoDup_keys (stor_rates prev_state)).
                        (* prove the In result *)
                        -   apply in_cons.
                            apply in_in_remove.
                            +   now apply not_eq_sym.
                            +   assert (exists x,
                                    FMap.find (token_out_trade t) (stor_rates prev_state) =
    Some x)
                                as rate_exists.
                                {   pose proof (trade_entrypoint_check_pf prev_state
                                        chain ctx t new_state new_acts receive_some)
                                    as trade_check.
                                    do 3 destruct trade_check as [* trade_check].
                                    destruct trade_check as [_ trade_check].
                                    destruct trade_check as [_ prev_rate].
                                    clear x x0.
                                    now exists x1. }
                                destruct rate_exists as [x rate_exists].
                                apply FMap.In_elements in rate_exists.
                                now apply (in_map fst
                                    (FMap.elements (stor_rates prev_state))
                                    (token_out_trade t, x)). }
```

```
907                     exact (Permutation_trans permute_step_2 (Permutation_sym permute_step_1)).
        }
908                 exact (Permutation_trans permute_1
909                     (Permutation_trans permute_2
910                         (perm_swap (token_in_trade t) (token_out_trade t)
911                         (remove token_eq_dec (token_out_trade t)
912                         (remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates
    prev_state))))))). }
913             (* Now separate LHS into ^^, plus new rate * t_x, minus rate * t_y *)
914             pose proof
915                 (Permutation_map tokens_to_values_prev)
916                     (* list 1 *)
917                     (FMap.keys (stor_rates prev_state))
918                     (* list 2 *)
919                     ((token_in_trade t) :: (token_out_trade t) :: keys_minus_trades_prev)
920                     (* the previous permutation *)
921                     keys_permuted_prev
922             as lhs_permute_map_prev.
923             (* now that we have that permutation, we can decompose the suml *)
924             rewrite (suml_permute_commutes
925                 (map tokens_to_values_prev (FMap.keys (stor_rates prev_state)))
926                 (map tokens_to_values_prev ((token_in_trade t) :: (token_out_trade t) ::
    keys_minus_trades_prev))
927                 lhs_permute_map_prev)
928             in IH.
929             (* now extract (token_in_trade t) and (token_out_trade t) *)
930             rewrite (map_extract
931                 (token_in_trade t)
932                 (token_out_trade t :: keys_minus_trades_prev)
933                 tokens_to_values_prev)
934             in IH.
935             rewrite (map_extract
936                 (token_out_trade t)
937                 keys_minus_trades_prev
938                 tokens_to_values_prev)
939             in IH.
940             rewrite (suml_extract (tokens_to_values_prev (token_in_trade t))) in IH.
941             rewrite (suml_extract (tokens_to_values_prev (token_out_trade t))) in IH.
942             rewrite <- IH.
943
944             (* Proceed in two parts *)
945             (* excluding the two tokens involved in the trade, all else is equal *)
946             assert (suml (map tokens_to_values_new  keys_minus_trades) =
947                     suml (map tokens_to_values_prev keys_minus_trades_prev))
948             as unchanged_keys_eq.
```

```
949             {   apply suml_permute_commutes.
950                 assert ((map tokens_to_values_prev keys_minus_trades_prev)
951                     = (map tokens_to_values_new keys_minus_trades_prev))
952                 as calc_eq.
953                 {   (* We use rz_unchanged and tokens_held_update_tz *)
954                     apply mapped_eq.
955                     intros a in_keys.
956                     (* first prove that a <> token_in_trade t and a <> token_out_trade t *)
957                     assert (a <> (token_in_trade t) /\ a <> (token_out_trade t))
958                     as a_neq_traded.
959                     {   unfold keys_minus_trades_prev in in_keys.
960                         destruct (in_remove token_eq_dec keys_minus_trade_in_prev a  (
        token_out_trade t) in_keys) as [a_in_prev a_neq_out].
961                         unfold keys_minus_trade_in_prev in a_in_prev.
962                         destruct (in_remove token_eq_dec
963                             (FMap.keys (stor_rates prev_state))
964                             a (token_in_trade t)
965                             a_in_prev)
966                         as [_ a_neq_in].
967                         split;
968                         try exact a_neq_out;
969                         try exact a_neq_in. }
970                     destruct a_neq_traded as [a_neq_in a_neq_out].
971                     pose proof (rz_unchanged a a_neq_in) as rates_eq.
972                     pose proof (tokens_held_update_tz a a_neq_in a_neq_out) as bal_eq.
973                     unfold keys_minus_trades_prev, tokens_to_values_prev, tokens_to_values_new
        .
974                     unfold get_rate.
975                     replace (FMap.find a (stor_rates new_state))
976                     with (FMap.find a (stor_rates prev_state)).
977                     now replace (get_bal a (stor_tokens_held new_state))
978                     with (get_bal a (stor_tokens_held prev_state)). }
979                 rewrite calc_eq.
980                 apply Permutation_map.
981                 unfold keys_minus_trades, keys_minus_trades_prev.
982                 unfold keys_minus_trade_in, keys_minus_trade_in_prev.
983                 do 2 (apply remove_permute_2;
984                 try apply nodup_remove;
985                 try apply FMap.NoDup_keys).
986                 (* ... *)
987                 destruct (trade_update_rates_pf prev_state chain ctx t new_state new_acts
        receive_some) as [_ update_rates].
988                 rewrite update_rates.
989                 clear update_rates.
990                 set (x := (calc_rx' (get_rate (token_in_trade t) (stor_rates prev_state))
```

```
991                 (get_rate (token_out_trade t) (stor_rates prev_state)) (qty_trade t) (
    stor_outstanding_tokens prev_state)
992                 (get_bal (token_in_trade t) (stor_tokens_held prev_state)))).
993                 assert (exists x', FMap.find (token_in_trade t) (stor_rates prev_state) = Some
     x')
994                 as prev_x.
995                 {   pose proof (trade_entrypoint_check_pf prev_state
996                         chain ctx t new_state new_acts receive_some)
997                     as trade_check.
998                     do 3 destruct trade_check as [* trade_check].
999                     destruct trade_check as [_ trade_check].
1000                    destruct trade_check as [prev_rate _].
1001                    clear x0 x2.
1002                    now exists x1. }
1003                destruct prev_x as [x' prev_x].
1004                exact (FMap.keys_already (token_in_trade t) x' x (stor_rates prev_state)
    prev_x). }
1005
1006            assert (tokens_to_values_new (token_in_trade t) + tokens_to_values_new (
    token_out_trade t)
1007                = tokens_to_values_prev (token_in_trade t) + tokens_to_values_prev (
    token_out_trade t))
1008            as changed_keys_eq.
1009            {   unfold tokens_to_values_new, tokens_to_values_prev.
1010                set (r_x' := (calc_rx' (get_rate (token_in_trade t) (stor_rates prev_state))
1011                (get_rate (token_out_trade t) (stor_rates prev_state)) (qty_trade t) (
    stor_outstanding_tokens prev_state)
1012                (get_bal (token_in_trade t) (stor_tokens_held prev_state)))).
1013                rewrite tokens_held_update_tx.
1014                rewrite tokens_held_update_ty.
1015                set (delta_y := calc_delta_y (get_rate (token_in_trade t) (stor_rates
    prev_state))
1016                (get_rate (token_out_trade t) (stor_rates prev_state)) (qty_trade t) (
    stor_outstanding_tokens prev_state)
1017                (get_bal (token_in_trade t) (stor_tokens_held prev_state))).
1018                set (r_x := get_rate (token_in_trade t) (stor_rates prev_state)).
1019                set (x := get_bal (token_in_trade t) (stor_tokens_held prev_state)).
1020                set (y := get_bal (token_out_trade t) (stor_tokens_held prev_state)).
1021                assert (get_rate (token_out_trade t) (stor_rates prev_state) =
1022                    get_rate (token_out_trade t) (stor_rates new_state))
1023                as ry_unchanged.
1024                {   unfold get_rate.
1025                    pose proof (rz_unchanged (token_out_trade t) (not_eq_sym tx_neq_ty)).
1026                    now replace (FMap.find (token_out_trade t) (stor_rates new_state))
1027                    with (FMap.find (token_out_trade t) (stor_rates prev_state)). }
```

```
1028                rewrite <- ry_unchanged.
1029                set (r_y := get_rate (token_out_trade t) (stor_rates prev_state)).
1030                unfold get_rate.
1031                replace (FMap.find (token_in_trade t) (stor_rates new_state))
1032                with (Some r_x').
1033                (* finally, from the specification *)
1034                apply rates_balance_2_pf. }
1035            rewrite unchanged_keys_eq.
1036            rewrite N.add_assoc.
1037            rewrite N.add_assoc.
1038            now rewrite changed_keys_eq.
1039        (* now for all other entrypoints *)
1040        +   destruct other as [o msg_other_op].
1041            rewrite msg_other_op in receive_some.
1042            rewrite <- (FMap.ext_eq (stor_rates prev_state) (stor_rates new_state) (
        other_rates_unchanged_pf prev_state new_state chain ctx o new_acts receive_some)).
1043            rewrite <- (FMap.ext_eq (stor_tokens_held prev_state) (stor_tokens_held new_state)
         (other_balances_unchanged_pf prev_state new_state chain ctx o new_acts receive_some)).
1044            rewrite <- (other_outstanding_unchanged_pf prev_state new_state chain ctx o
        new_acts receive_some).
1045            assumption.
1046    (* Please reestablish the invariant after a recursive call *)
1047    -   destruct msg.
1048        (* first, msg = None *)
1049        2:{ is_sp_destruct.
1050            pose proof (none_fails_pf prev_state chain ctx) as failed.
1051            destruct failed as [err failed].
1052            rewrite receive_some in failed.
1053            inversion failed. }
1054        (* msg = Some m *)
1055        is_sp_destruct.
1056        pose proof (msg_destruct_pf m) as msg_destruct.
1057        destruct msg_destruct as [pool | [unpool | [trade | other]]].
1058        (* m = pool *)
1059        +   destruct pool as [p msg_pool].
1060            rewrite msg_pool in receive_some.
1061            (* understand how tokens_held has changed *)
1062            pose proof (pool_increases_tokens_held_pf prev_state chain ctx p new_state
        new_acts
1063                receive_some) as bal_update.
1064            destruct bal_update as [bal_update bals_unchanged].
1065            (* show all rates are the same *)
1066            pose proof (pool_rates_unchanged_pf prev_state new_state chain ctx p new_acts
1067                receive_some) as rates_update.
1068            (* show how the outstanding tokens update *)
```

```
1069            pose proof (pool_outstanding_pf prev_state new_state chain ctx p new_acts
1070                receive_some) as out_update.
1071            simpl in out_update.
1072            (* because the rates are unchanged, summing over them is unchanged *)
1073            assert (tokens_to_values (stor_rates new_state)  (stor_tokens_held new_state) =
1074                    tokens_to_values (stor_rates prev_state) (stor_tokens_held new_state))
1075            as rates_unchanged.
1076            {   unfold tokens_to_values.
1077                rewrite (token_keys_invariant_pool p new_state new_acts prev_state ctx chain
    receive_some).
1078                apply map_ext.
1079                intro t.
1080                unfold get_rate.
1081                now rewrite rates_update. }
1082            rewrite rates_unchanged. clear rates_unchanged.
1083            (* now separate the sum *)
1084            assert (suml (tokens_to_values (stor_rates prev_state) (stor_tokens_held new_state
    )) =
1085                    suml (tokens_to_values (stor_rates prev_state) (stor_tokens_held
    prev_state)) + get_rate (token_pooled p) (stor_rates prev_state) * qty_pooled p)
1086            as separate_sum.
1087            {   unfold tokens_to_values.
1088                (* some simplifying notation *)
1089                set (keys_minus_p := (remove token_eq_dec (token_pooled p) (FMap.keys (
    stor_rates prev_state)))).
1090                set (tokens_to_values_new := fun k : token => get_rate k (stor_rates
    prev_state) * get_bal k (stor_tokens_held new_state)).
1091                set (tokens_to_values_prev := fun (k : token) => get_rate k (stor_rates
    prev_state) * get_bal k (stor_tokens_held prev_state)).
1092                (* 1. show there's a Permutation with (token_pooled p) at the front *)
1093                assert (Permutation
1094                    (FMap.keys (stor_rates prev_state))
1095                    ((token_pooled p) :: keys_minus_p))
1096                as keys_permuted.
1097                {   unfold keys_minus_p.
1098                    (* In (token_pooled p) (FMap.keys (stor_rates prev_state)) *)
1099                    pose proof (pool_entrypoint_check_pf prev_state new_state chain ctx p
    new_acts receive_some) as rate_exists.
1100                    destruct rate_exists as [r_x rate_exists_el].
1101                    apply FMap.In_elements in rate_exists_el.
1102                    pose proof (in_map fst
1103                        (FMap.elements (stor_rates prev_state))
1104                        (token_pooled p, r_x)
1105                        rate_exists_el)
1106                    as rate_exists_key. clear rate_exists_el.
```

```
1107                    simpl in rate_exists_key.
1108                    (* recall NoDup keys *)
1109                    pose proof (FMap.NoDup_keys (stor_rates prev_state))
1110                    as nodup_keys.
1111                    (* since NoDup keys, you can permute *)
1112                    exact (nodup_permute
1113                        (token_pooled p)
1114                        (FMap.keys (stor_rates prev_state))
1115                        token_eq_dec
1116                        nodup_keys
1117                        rate_exists_key). }
1118                (* Use the permutation to rewrite the LHS first, extracting (token_pooled p)
        *)
1119                pose proof
1120                    (Permutation_map tokens_to_values_new)
1121                        (* list 1 *)
1122                        (FMap.keys (stor_rates prev_state))
1123                        (* list 2 *)
1124                        ((token_pooled p) :: keys_minus_p)
1125                        (* the previous permutation *)
1126                        keys_permuted
1127                as lhs_permute_map.
1128                (* now that we have that permutation, rewrite in the suml *)
1129                rewrite (suml_permute_commutes
1130                    (map tokens_to_values_new (FMap.keys (stor_rates prev_state)))
1131                    (map tokens_to_values_new (token_pooled p :: keys_minus_p))
1132                    lhs_permute_map).
1133                (* now extract (token_pooled p) *)
1134                rewrite (map_extract
1135                    (token_pooled p)
1136                    keys_minus_p
1137                    tokens_to_values_new).
1138                rewrite (suml_extract (tokens_to_values_new (token_pooled p))).
1139                (* Now separate the sum on the RHS *)
1140                pose proof
1141                    (Permutation_map tokens_to_values_prev)
1142                        (* list 1 *)
1143                        (FMap.keys (stor_rates prev_state))
1144                        (* list 2 *)
1145                        ((token_pooled p) :: keys_minus_p)
1146                        (* the previous permutation *)
1147                        keys_permuted
1148                as rhs_permute_map.
1149                (* now that we have that permutation, rewrite in the suml *)
1150                rewrite (suml_permute_commutes
```

```
                    (map tokens_to_values_prev (FMap.keys (stor_rates prev_state)))
                    (map tokens_to_values_prev (token_pooled p :: keys_minus_p))
                    rhs_permute_map).
                (* now extract (token_pooled p) *)
                rewrite (map_extract
                    (token_pooled p)
                    keys_minus_p
                    tokens_to_values_prev).
                rewrite (suml_extract (tokens_to_values_prev (token_pooled p))).

                (* Now proceed in two parts *)
                (* first the sum of all the unchanged keys *)
                assert
                    (suml (map tokens_to_values_new keys_minus_p) =
                     suml (map tokens_to_values_prev keys_minus_p))
                as old_keys_eq.
                {   assert (forall t, In t keys_minus_p -> t <> (token_pooled p))
                    as t_neq.
                    {   intros * H_in.
                        unfold keys_minus_p in H_in.
                        apply in_remove in H_in.
                        destruct H_in as [_ res].
                        exact res. }
                    assert (forall t, In t keys_minus_p ->
                        tokens_to_values_new t = tokens_to_values_prev t)
                    as tokens_to_vals_eq.
                    {   intros * H_in.
                        apply t_neq in H_in.
                        apply bals_unchanged in H_in.
                        unfold tokens_to_values_new, tokens_to_values_prev.
                        now rewrite H_in. }
                    now rewrite (mapped_eq
                        token N
                        tokens_to_values_new
                        tokens_to_values_prev
                        keys_minus_p
                        tokens_to_vals_eq). }

                (* then to the key that did change *)
                assert
                    (tokens_to_values_new (token_pooled p) =
                    tokens_to_values_prev (token_pooled p) +
                    get_rate (token_pooled p) (stor_rates prev_state) * qty_pooled p)
                as new_key_eq.
                {   unfold tokens_to_values_new, tokens_to_values_prev.
```

```
1196                    rewrite bal_update.
1197                    apply N.mul_add_distr_l. }
1198                rewrite old_keys_eq. rewrite new_key_eq.
1199                rewrite <- N.add_assoc.
1200                rewrite (N.add_comm
1201                    (get_rate (token_pooled p) (stor_rates prev_state) * qty_pooled p)
1202                    (suml (map tokens_to_values_prev keys_minus_p))).
1203                now rewrite N.add_assoc. }
1204            rewrite separate_sum. clear separate_sum.
1205            rewrite out_update.
1206            now rewrite IH.
1207        (* m = unpool *)
1208        +   destruct unpool as [u msg_unpool].
1209            rewrite msg_unpool in receive_some.
1210            (* understand how tokens_held has changed *)
1211            pose proof (unpool_decreases_tokens_held_pf prev_state chain ctx u new_state
    new_acts
1212                receive_some) as bal_update.
1213            destruct bal_update as [bal_update bals_unchanged].
1214            (* show all rates are the same *)
1215            pose proof (unpool_rates_unchanged_pf prev_state new_state chain ctx u new_acts
1216                receive_some) as rates_update.
1217            (* show how the outstanding tokens update *)
1218            pose proof (unpool_outstanding_pf prev_state new_state chain ctx u new_acts
1219                receive_some) as out_update.
1220            simpl in out_update.
1221            rewrite out_update. clear out_update.
1222            (* because the rates are unchanged, summing over them is unchanged *)
1223            assert (tokens_to_values (stor_rates new_state)  (stor_tokens_held new_state) =
1224                    tokens_to_values (stor_rates prev_state) (stor_tokens_held new_state))
1225            as rates_unchanged.
1226            {   unfold tokens_to_values.
1227                rewrite (token_keys_invariant_unpool u new_state new_acts prev_state ctx chain
     receive_some).
1228                apply map_ext.
1229                intro t.
1230                unfold get_rate.
1231                now rewrite rates_update. }
1232            rewrite rates_unchanged. clear rates_unchanged.
1233            (* now separate the sum *)
1234            assert (suml (tokens_to_values (stor_rates prev_state) (stor_tokens_held new_state
    )) = suml (tokens_to_values (stor_rates prev_state) (stor_tokens_held prev_state)) -
    qty_unpooled u)
1235            as separate_sum.
1236            {   unfold tokens_to_values.
```

```
1237                (* some simplifying notation *)
1238                set (keys_minus_u := (remove token_eq_dec (token_unpooled u) (FMap.keys (
        stor_rates prev_state)))).
1239                set (tokens_to_values_new := fun k : token => get_rate k (stor_rates
        prev_state) * get_bal k (stor_tokens_held new_state)).
1240                set (tokens_to_values_prev := fun k : token => get_rate k (stor_rates
        prev_state) * get_bal k (stor_tokens_held prev_state)).
1241                (* 1. show there's a Permutation with (token_pooled p) at the front *)
1242                assert (Permutation
1243                    (FMap.keys (stor_rates prev_state))
1244                    ((token_unpooled u) :: keys_minus_u))
1245                as keys_permuted.
1246                {   unfold keys_minus_u.
1247                    (* In (token_pooled p) (FMap.keys (stor_rates prev_state)) *)
1248                    pose proof (unpool_entrypoint_check_pf prev_state new_state chain ctx u
        new_acts receive_some) as rate_exists.
1249                    destruct rate_exists as [r_x rate_exists_el].
1250                    apply FMap.In_elements in rate_exists_el.
1251                    pose proof (in_map fst
1252                        (FMap.elements (stor_rates prev_state))
1253                        (token_unpooled u, r_x)
1254                        rate_exists_el)
1255                    as rate_exists_key. clear rate_exists_el.
1256                    simpl in rate_exists_key.
1257                    (* recall NoDup keys *)
1258                    pose proof (FMap.NoDup_keys (stor_rates prev_state))
1259                    as nodup_keys.
1260                    (* since NoDup keys, you can permute *)
1261                    exact (nodup_permute
1262                        (token_unpooled u)
1263                        (FMap.keys (stor_rates prev_state))
1264                        token_eq_dec
1265                        nodup_keys
1266                        rate_exists_key). }
1267                (* Use the permutation to rewrite the LHS first, extracting (token_pooled p)
        *)
1268                pose proof
1269                    (Permutation_map tokens_to_values_new)
1270                        (* list 1 *)
1271                        (FMap.keys (stor_rates prev_state))
1272                        (* list 2 *)
1273                        ((token_unpooled u) :: keys_minus_u)
1274                        (* the previous permutation *)
1275                        keys_permuted
1276                as lhs_permute_map.
```

```
                (* now that we have that permutation, rewrite in the suml *)
                rewrite (suml_permute_commutes
                    (map tokens_to_values_new (FMap.keys (stor_rates prev_state)))
                    (map tokens_to_values_new (token_unpooled u :: keys_minus_u))
                    lhs_permute_map).
                (* now extract (token_pooled p) *)
                rewrite (map_extract
                    (token_unpooled u)
                    keys_minus_u
                    tokens_to_values_new).
                rewrite (suml_extract (tokens_to_values_new (token_unpooled u))).
                (* Now separate the sum on the RHS *)
                pose proof
                    (Permutation_map tokens_to_values_prev)
                        (* list 1 *)
                        (FMap.keys (stor_rates prev_state))
                        (* list 2 *)
                        ((token_unpooled u) :: keys_minus_u)
                        (* the previous permutation *)
                        keys_permuted
                as rhs_permute_map.
                (* now that we have that permutation, rewrite in the suml *)
                rewrite (suml_permute_commutes
                    (map tokens_to_values_prev (FMap.keys (stor_rates prev_state)))
                    (map tokens_to_values_prev (token_unpooled u :: keys_minus_u))
                    rhs_permute_map).
                (* now extract (token_pooled p) *)
                rewrite (map_extract
                    (token_unpooled u)
                    keys_minus_u
                    tokens_to_values_prev).
                rewrite (suml_extract (tokens_to_values_prev (token_unpooled u))).

                (* Now proceed in two parts *)
                (* first the sum of all the unchanged keys *)
                assert
                    (suml (map tokens_to_values_new keys_minus_u) =
                     suml (map tokens_to_values_prev keys_minus_u))
                as old_keys_eq.
                {   assert (forall t, In t keys_minus_u -> t <> (token_unpooled u))
                    as t_neq.
                    {   intros * H_in.
                        unfold keys_minus_u in H_in.
                        apply in_remove in H_in.
                        destruct H_in as [_ res].
```

```
                        exact res. }
                    assert (forall t, In t keys_minus_u ->
                        tokens_to_values_new t = tokens_to_values_prev t)
                    as tokens_to_vals_eq.
                    {   intros * H_in.
                        apply t_neq in H_in.
                        apply bals_unchanged in H_in.
                        unfold tokens_to_values_new, tokens_to_values_prev.
                        now rewrite H_in. }
                    now rewrite (mapped_eq
                        token N
                        tokens_to_values_new
                        tokens_to_values_prev
                        keys_minus_u
                        tokens_to_vals_eq). }
                rewrite old_keys_eq. clear old_keys_eq.

                (* then to the key that did change *)
                assert
                    (tokens_to_values_new (token_unpooled u) =
                    tokens_to_values_prev (token_unpooled u) - qty_unpooled u)
                as new_key_eq.
                {   unfold tokens_to_values_new, tokens_to_values_prev.
                    (* get the calculation from the specification *)
                    rewrite bal_update.
                    rewrite (N.mul_sub_distr_l
                        (get_bal (token_unpooled u) (stor_tokens_held prev_state))
                        (calc_rx_inv (get_rate (token_unpooled u) (stor_rates prev_state)) (
    qty_unpooled u))
                        (get_rate (token_unpooled u) (stor_rates prev_state))).
                    now rewrite rates_balance_pf. }
                rewrite new_key_eq.

                (* this is an additional condition required here so that commutativity applies
     *)
                assert (qty_unpooled u <= tokens_to_values_prev (token_unpooled u))
                as unpooled_leq. {
                    unfold tokens_to_values_prev.
                    exact (unpool_entrypoint_check_2_pf prev_state new_state chain ctx u
    new_acts receive_some). }
                now rewrite (N.add_sub_swap
                    (tokens_to_values_prev (token_unpooled u))
                    (suml (map tokens_to_values_prev keys_minus_u))
                    (qty_unpooled u)
                    unpooled_leq). }
```

```
1364            rewrite separate_sum. clear separate_sum.
1365            now rewrite IH.
1366        (* m = trade *)
1367        +   destruct trade as [t msg_trade].
1368            rewrite msg_trade in receive_some.
1369            (* understand how tokens_held has changed *)
1370            destruct (trade_tokens_held_update_pf prev_state chain ctx t new_state new_acts
1371            receive_some) as [tokens_held_update_ty [tokens_held_update_tx
    tokens_held_update_tz]].
1372            (* how calling trade updates rates *)
1373            destruct (trade_update_rates_formula_pf prev_state chain ctx t new_state new_acts
1374            receive_some) as [tx_neq_ty [rx_change rz_unchanged]].
1375            (* how outstanding_tokens has changed by calling trade *)
1376            pose proof (trade_outstanding_update_pf prev_state chain ctx t new_state new_acts
1377                receive_some) as out_update.
1378            rewrite out_update. clear out_update.
1379
1380            (* LHS equals sum over tokens except for tx and ty, plus r_x' * etc , minus r_y *
    delta_y *)
1381
1382            (* some notation *)
1383            unfold tokens_to_values.
1384            set (keys_minus_trade_in := (remove token_eq_dec (token_in_trade t) (FMap.keys (
    stor_rates new_state)))).
1385            set (keys_minus_trades := (remove token_eq_dec (token_out_trade t)
    keys_minus_trade_in)).
1386            set (tokens_to_values_new := fun k : token => get_rate k (stor_rates new_state) *
    get_bal k (stor_tokens_held new_state)).
1387
1388            (* LHS *)
1389            (* permute FMap.keys (stor_tokens_held new_state) to have tx :: ty :: keys' *)
1390                (* for these, rates and balances are all the same as before *)
1391            assert (Permutation
1392                (FMap.keys (stor_rates new_state))
1393                ((token_in_trade t) :: (token_out_trade t) :: keys_minus_trades))
1394            as keys_permuted.
1395            {   unfold keys_minus_trades.
1396                unfold keys_minus_trade_in.
1397                assert (Permutation
1398                    (FMap.keys (stor_rates new_state))
1399                    (token_in_trade t ::
1400                    remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates new_state)))
    )
1401                as permute_1.
1402                {   apply (nodup_permute
```

```
1403                        (token_in_trade t)
1404                        (FMap.keys (stor_rates new_state))
1405                        token_eq_dec
1406                        (FMap.NoDup_keys (stor_rates new_state))).
1407                    assert (exists x, FMap.find (token_in_trade t) (stor_rates new_state) =
    Some x) as rate_exists.
1408                    {   destruct (trade_update_rates_pf prev_state chain ctx t new_state
    new_acts receive_some) as [_ update_rates].
1409                        simpl in update_rates.
1410                        rewrite update_rates.
1411                        set (x := (calc_rx' (get_rate (token_in_trade t) (stor_rates
    prev_state))
1412                        (get_rate (token_out_trade t) (stor_rates prev_state)) (qty_trade t)
1413                        (stor_outstanding_tokens prev_state) (get_bal (token_in_trade t) (
    stor_tokens_held prev_state)))).
1414                        exists x.
1415                        apply FMap.find_add. }
1416                    destruct rate_exists as [x rate_exists].
1417                    apply (FMap.In_elements (token_in_trade t) x (stor_rates new_state))
1418                    in rate_exists.
1419                    now apply (in_map fst
1420                        (FMap.elements (stor_rates new_state))
1421                        (token_in_trade t, x)). }
1422                assert (Permutation
1423                    (token_in_trade t ::
1424                    remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates new_state)))
1425                    ((token_out_trade t) :: (token_in_trade t) ::
1426                    remove token_eq_dec (token_out_trade t)
1427                    (remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates new_state))
    )))
1428                as permute_2.
1429                {   assert (Permutation
1430                        (token_out_trade t :: token_in_trade t ::
1431                            remove token_eq_dec (token_out_trade t)
1432                            (remove token_eq_dec (token_in_trade t)
1433                                (FMap.keys (stor_rates new_state))))
1434                        (token_out_trade t ::
1435                            remove token_eq_dec (token_out_trade t)
1436                            (token_in_trade t :: remove token_eq_dec (token_in_trade t)
1437                                (FMap.keys (stor_rates new_state)))))
1438                    as permute_step_1.
1439                    {   assert (Permutation
1440                            (token_in_trade t
1441                            :: remove token_eq_dec (token_out_trade t) (remove token_eq_dec (
    token_in_trade t) (FMap.keys (stor_rates new_state))))
```

```
1442                            (remove token_eq_dec (token_out_trade t)
1443                            (token_in_trade t :: remove token_eq_dec (token_in_trade t) (FMap.
        keys (stor_rates new_state)))))
1444                        as inner_permute.
1445                        {   set (rates_remove_in := (remove token_eq_dec (token_in_trade t) (
        FMap.keys (stor_rates new_state)))).
1446                            rewrite remove_permute; try assumption.
1447                            apply Permutation_refl. }
1448                        exact (perm_skip (token_out_trade t) inner_permute). }
1449                    assert (Permutation
1450                    (token_in_trade t ::
1451                    remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates new_state)))
1452                    (token_out_trade t ::
1453                            remove token_eq_dec (token_out_trade t)
1454                            (token_in_trade t :: remove token_eq_dec (token_in_trade t)
1455                                (FMap.keys (stor_rates new_state)))))
1456                    as permute_step_2.
1457                    {   apply nodup_permute.
1458                        (* prove the NoDup result *)
1459                        -   apply NoDup_cons.
1460                            +   apply remove_In.
1461                            +   apply nodup_remove.
1462                                exact (FMap.NoDup_keys (stor_rates new_state)).
1463                        (* prove the In result *)
1464                        -   apply in_cons.
1465                            apply in_in_remove.
1466                            +   now apply not_eq_sym.
1467                            +   assert (exists x,
1468                                    FMap.find (token_out_trade t) (stor_rates new_state) =
        Some x)
1469                                as rate_exists.
1470                                {   pose proof (trade_entrypoint_check_pf prev_state
1471                                    chain ctx t new_state new_acts receive_some)
1472                                    as trade_check.
1473                                    (* get rate from prev_state *)
1474                                    do 3 destruct trade_check as [* trade_check].
1475                                    destruct trade_check as [_ trade_check].
1476                                    destruct trade_check as [_ prev_rate].
1477                                    clear x x0.
1478                                    (* get the rate from the new state *)
1479                                    rewrite <- (rz_unchanged (token_out_trade t) (not_eq_sym
        tx_neq_ty)) in prev_rate.
1480                                    now exists x1. }
1481                                destruct rate_exists as [x rate_exists].
1482                                apply FMap.In_elements in rate_exists.
```

```
1483                                 now apply (in_map fst
1484                                     (FMap.elements (stor_rates new_state))
1485                                     (token_out_trade t, x)). }
1486                     exact (Permutation_trans permute_step_2 (Permutation_sym permute_step_1)).
     }
1487                 exact (Permutation_trans permute_1
1488                     (Permutation_trans permute_2
1489                         (perm_swap (token_in_trade t) (token_out_trade t)
1490                         (remove token_eq_dec (token_out_trade t)
1491                         (remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates
    new_state))))))). }
1492             (* Now separate LHS into ^^, plus new rate * t_x, minus rate * t_y *)
1493             pose proof
1494                 (Permutation_map tokens_to_values_new)
1495                     (* list 1 *)
1496                     (FMap.keys (stor_rates new_state))
1497                     (* list 2 *)
1498                     ((token_in_trade t) :: (token_out_trade t) :: keys_minus_trades)
1499                     (* the previous permutation *)
1500                     keys_permuted
1501             as lhs_permute_map.
1502             (* now that we have that permutation, we can decompose the suml *)
1503             rewrite (suml_permute_commutes
1504                 (map tokens_to_values_new (FMap.keys (stor_rates new_state)))
1505                 (map tokens_to_values_new ((token_in_trade t) :: (token_out_trade t) ::
    keys_minus_trades))
1506                 lhs_permute_map).
1507             (* now extract (token_in_trade t) and (token_out_trade t) *)
1508             rewrite (map_extract
1509                 (token_in_trade t)
1510                 (token_out_trade t :: keys_minus_trades)
1511                 tokens_to_values_new).
1512             rewrite (map_extract
1513                 (token_out_trade t)
1514                 keys_minus_trades
1515                 tokens_to_values_new).
1516             rewrite (suml_extract (tokens_to_values_new (token_in_trade t))).
1517             rewrite (suml_extract (tokens_to_values_new (token_out_trade t))).
1518
1519             (* permute FMap.keys for the OLD suml to MANIPULATE the inductive hypothesis IH *)
1520             unfold tokens_to_values in IH.
1521             set (keys_minus_trade_in_prev := (remove token_eq_dec (token_in_trade t) (FMap.
    keys (stor_rates prev_state)))).
1522             set (keys_minus_trades_prev := (remove token_eq_dec (token_out_trade t)
    keys_minus_trade_in_prev)).
```

```
1523            set (tokens_to_values_prev := fun k : token => get_rate k (stor_rates prev_state)
        * get_bal k (stor_tokens_held prev_state)) in IH.
1524
1525            assert (Permutation
1526                (FMap.keys (stor_rates prev_state))
1527                ((token_in_trade t) :: (token_out_trade t) :: keys_minus_trades_prev))
1528            as keys_permuted_prev.
1529            {   unfold keys_minus_trades_prev.
1530                unfold keys_minus_trade_in_prev.
1531                assert (Permutation
1532                    (FMap.keys (stor_rates prev_state))
1533                    (token_in_trade t ::
1534                    remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates prev_state))
        ))
1535                as permute_1.
1536                {   apply (nodup_permute
1537                        (token_in_trade t)
1538                        (FMap.keys (stor_rates prev_state))
1539                        token_eq_dec
1540                        (FMap.NoDup_keys (stor_rates prev_state))).
1541                    assert (exists x, FMap.find (token_in_trade t) (stor_rates prev_state) =
        Some x) as rate_exists.
1542                    {   pose proof (trade_entrypoint_check_pf prev_state
1543                            chain ctx t new_state new_acts receive_some)
1544                        as trade_check.
1545                        do 3 destruct trade_check as [* trade_check].
1546                        destruct trade_check as [_ trade_check].
1547                        destruct trade_check as [prev_rate _].
1548                        clear x x1.
1549                        now exists x0. }
1550                    destruct rate_exists as [x rate_exists].
1551                    apply (FMap.In_elements (token_in_trade t) x (stor_rates prev_state))
1552                    in rate_exists.
1553                    now apply (in_map fst
1554                        (FMap.elements (stor_rates prev_state))
1555                        (token_in_trade t, x)). }
1556                assert (Permutation
1557                    (token_in_trade t ::
1558                    remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates prev_state))
        )
1559                    ((token_out_trade t) :: (token_in_trade t) ::
1560                    remove token_eq_dec (token_out_trade t)
1561                    (remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates prev_state)
        ))))
1562                as permute_2.
```

```
1563                {   assert (Permutation
1564                        (token_out_trade t :: token_in_trade t ::
1565                            remove token_eq_dec (token_out_trade t)
1566                            (remove token_eq_dec (token_in_trade t)
1567                                (FMap.keys (stor_rates prev_state))))
1568                        (token_out_trade t ::
1569                            remove token_eq_dec (token_out_trade t)
1570                            (token_in_trade t :: remove token_eq_dec (token_in_trade t)
1571                                (FMap.keys (stor_rates prev_state)))))
1572                    as permute_step_1.
1573                    {   assert (Permutation
1574                            (token_in_trade t
1575                            :: remove token_eq_dec (token_out_trade t) (remove token_eq_dec (
        token_in_trade t) (FMap.keys (stor_rates prev_state))))
1576                            (remove token_eq_dec (token_out_trade t)
1577                            (token_in_trade t :: remove token_eq_dec (token_in_trade t) (FMap.
        keys (stor_rates prev_state)))))
1578                        as inner_permute.
1579                        {   set (rates_remove_in := (remove token_eq_dec (token_in_trade t) (
        FMap.keys (stor_rates prev_state)))).
1580                            rewrite remove_permute; try assumption.
1581                            apply Permutation_refl. }
1582                        exact (perm_skip (token_out_trade t) inner_permute). }
1583                    assert (Permutation
1584                    (token_in_trade t ::
1585                    remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates prev_state))
        )
1586                    (token_out_trade t ::
1587                            remove token_eq_dec (token_out_trade t)
1588                            (token_in_trade t :: remove token_eq_dec (token_in_trade t)
1589                                (FMap.keys (stor_rates prev_state)))))
1590                    as permute_step_2.
1591                    {   apply nodup_permute.
1592                        (* prove the NoDup result *)
1593                        -   apply NoDup_cons.
1594                            +   apply remove_In.
1595                            +   apply nodup_remove.
1596                                exact (FMap.NoDup_keys (stor_rates prev_state)).
1597                        (* prove the In result *)
1598                        -   apply in_cons.
1599                            apply in_in_remove.
1600                            +   now apply not_eq_sym.
1601                            +   assert (exists x,
1602                                    FMap.find (token_out_trade t) (stor_rates prev_state) =
        Some x)
```

```
                                as rate_exists.
                                {   pose proof (trade_entrypoint_check_pf prev_state
                                        chain ctx t new_state new_acts receive_some)
                                    as trade_check.
                                    do 3 destruct trade_check as [* trade_check].
                                    destruct trade_check as [_ trade_check].
                                    destruct trade_check as [_ prev_rate].
                                    clear x x0.
                                    now exists x1. }
                                destruct rate_exists as [x rate_exists].
                                apply FMap.In_elements in rate_exists.
                                now apply (in_map fst
                                    (FMap.elements (stor_rates prev_state))
                                    (token_out_trade t, x)). }
                    exact (Permutation_trans permute_step_2 (Permutation_sym permute_step_1)).
 }
                exact (Permutation_trans permute_1
                    (Permutation_trans permute_2
                        (perm_swap (token_in_trade t) (token_out_trade t)
                        (remove token_eq_dec (token_out_trade t)
                        (remove token_eq_dec (token_in_trade t) (FMap.keys (stor_rates
prev_state))))))). }
            (* Now separate LHS into ^^, plus new rate * t_x, minus rate * t_y *)
            pose proof
                (Permutation_map tokens_to_values_prev)
                    (* list 1 *)
                    (FMap.keys (stor_rates prev_state))
                    (* list 2 *)
                    ((token_in_trade t) :: (token_out_trade t) :: keys_minus_trades_prev)
                    (* the previous permutation *)
                    keys_permuted_prev
            as lhs_permute_map_prev.
            (* now that we have that permutation, we can decompose the suml *)
            rewrite (suml_permute_commutes
                (map tokens_to_values_prev (FMap.keys (stor_rates prev_state)))
                (map tokens_to_values_prev ((token_in_trade t) :: (token_out_trade t) ::
keys_minus_trades_prev))
                lhs_permute_map_prev)
            in IH.
            (* now extract (token_in_trade t) and (token_out_trade t) *)
            rewrite (map_extract
                (token_in_trade t)
                (token_out_trade t :: keys_minus_trades_prev)
                tokens_to_values_prev)
            in IH.
```

```
            rewrite (map_extract
                (token_out_trade t)
                keys_minus_trades_prev
                tokens_to_values_prev)
            in IH.
            rewrite (suml_extract (tokens_to_values_prev (token_in_trade t))) in IH.
            rewrite (suml_extract (tokens_to_values_prev (token_out_trade t))) in IH.
            rewrite <- IH.

            (* Proceed in two parts *)
            (* excluding the two tokens involved in the trade, all else is equal *)
            assert (suml (map tokens_to_values_new  keys_minus_trades) =
                    suml (map tokens_to_values_prev keys_minus_trades_prev))
            as unchanged_keys_eq.
            {   apply suml_permute_commutes.
                assert ((map tokens_to_values_prev keys_minus_trades_prev)
                    = (map tokens_to_values_new keys_minus_trades_prev))
                as calc_eq.
                {   (* We use rz_unchanged and tokens_held_update_tz *)
                    apply mapped_eq.
                    intros a in_keys.
                    (* first prove that a <> token_in_trade t and a <> token_out_trade t *)
                    assert (a <> (token_in_trade t) /\ a <> (token_out_trade t))
                    as a_neq_traded.
                    {   unfold keys_minus_trades_prev in in_keys.
                        destruct (in_remove token_eq_dec
                            keys_minus_trade_in_prev
                            a (token_out_trade t)
                            in_keys)
                        as [a_in_prev a_neq_out].
                        unfold keys_minus_trade_in_prev in a_in_prev.
                        destruct (in_remove token_eq_dec
                            (FMap.keys (stor_rates prev_state))
                            a (token_in_trade t)
                            a_in_prev)
                        as [_ a_neq_in].
                        split;
                        try exact a_neq_out;
                        try exact a_neq_in. }
                    destruct a_neq_traded as [a_neq_in a_neq_out].
                    pose proof (rz_unchanged a a_neq_in) as rates_eq.
                    pose proof (tokens_held_update_tz a a_neq_in a_neq_out) as bal_eq.
                    unfold keys_minus_trades_prev, tokens_to_values_prev, tokens_to_values_new
        .
                    unfold get_rate.
```

```
                    replace (FMap.find a (stor_rates new_state))
                    with (FMap.find a (stor_rates prev_state)).
                    now replace (get_bal a (stor_tokens_held new_state))
                    with (get_bal a (stor_tokens_held prev_state)). }
                rewrite calc_eq.
                apply Permutation_map.
                unfold keys_minus_trades, keys_minus_trades_prev.
                unfold keys_minus_trade_in, keys_minus_trade_in_prev.
                do 2 (apply remove_permute_2;
                try apply nodup_remove;
                try apply FMap.NoDup_keys).
                (* ... *)
                pose proof (trade_update_rates_pf prev_state chain ctx t new_state new_acts receive_some)
                as update_rates.
                destruct update_rates as [_ update_rates].
                rewrite update_rates. clear update_rates.
                set (x := (calc_rx' (get_rate (token_in_trade t) (stor_rates prev_state))
                (get_rate (token_out_trade t) (stor_rates prev_state)) (qty_trade t) (stor_outstanding_tokens prev_state)
                (get_bal (token_in_trade t) (stor_tokens_held prev_state)))).
                assert (exists x', FMap.find (token_in_trade t) (stor_rates prev_state) = Some x')
                as prev_x.
                {   pose proof (trade_entrypoint_check_pf prev_state
                        chain ctx t new_state new_acts receive_some)
                    as trade_check.
                    do 3 destruct trade_check as [* trade_check].
                    destruct trade_check as [_ trade_check].
                    destruct trade_check as [prev_rate _].
                    clear x0 x2.
                    now exists x1. }
                destruct prev_x as [x' prev_x].
                exact (FMap.keys_already (token_in_trade t) x' x (stor_rates prev_state) prev_x). }

            assert (tokens_to_values_new (token_in_trade t) + tokens_to_values_new (token_out_trade t)
                = tokens_to_values_prev (token_in_trade t) + tokens_to_values_prev (token_out_trade t))
            as changed_keys_eq.
            {   unfold tokens_to_values_new, tokens_to_values_prev.
                set (r_x' := (calc_rx' (get_rate (token_in_trade t) (stor_rates prev_state))
                (get_rate (token_out_trade t) (stor_rates prev_state)) (qty_trade t) (stor_outstanding_tokens prev_state)
```

```
1727                (get_bal (token_in_trade t) (stor_tokens_held prev_state)))).
1728                rewrite tokens_held_update_tx.
1729                rewrite tokens_held_update_ty.
1730                set (delta_y := calc_delta_y (get_rate (token_in_trade t) (stor_rates
        prev_state))
1731                (get_rate (token_out_trade t) (stor_rates prev_state)) (qty_trade t) (
        stor_outstanding_tokens prev_state)
1732                (get_bal (token_in_trade t) (stor_tokens_held prev_state))).
1733                set (r_x := get_rate (token_in_trade t) (stor_rates prev_state)).
1734                set (x := get_bal (token_in_trade t) (stor_tokens_held prev_state)).
1735                set (y := get_bal (token_out_trade t) (stor_tokens_held prev_state)).
1736                assert (get_rate (token_out_trade t) (stor_rates prev_state) =
1737                    get_rate (token_out_trade t) (stor_rates new_state))
1738                as ry_unchanged.
1739                {   unfold get_rate.
1740                    pose proof (rz_unchanged (token_out_trade t) (not_eq_sym tx_neq_ty)).
1741                    now replace (FMap.find (token_out_trade t) (stor_rates new_state))
1742                    with (FMap.find (token_out_trade t) (stor_rates prev_state)). }
1743                rewrite <- ry_unchanged.
1744                set (r_y := get_rate (token_out_trade t) (stor_rates prev_state)).
1745                unfold get_rate.
1746                replace (FMap.find (token_in_trade t) (stor_rates new_state))
1747                with (Some r_x').
1748                (* finally, from the specification *)
1749                apply rates_balance_2_pf. }
1750            rewrite unchanged_keys_eq.
1751            rewrite N.add_assoc.
1752            rewrite N.add_assoc.
1753            now rewrite changed_keys_eq.
1754        (* now for all other entrypoints *)
1755        +   destruct other as [o msg_other_op].
1756            rewrite msg_other_op in receive_some.
1757            rewrite <- (FMap.ext_eq (stor_rates prev_state) (stor_rates new_state) (
        other_rates_unchanged_pf prev_state new_state chain ctx o new_acts receive_some)).
1758            rewrite <- (FMap.ext_eq (stor_tokens_held prev_state) (stor_tokens_held new_state)
         (other_balances_unchanged_pf prev_state new_state chain ctx o new_acts receive_some)).
1759            rewrite <- (other_outstanding_unchanged_pf prev_state new_state chain ctx o
        new_acts receive_some).
1760            assumption.
1761    (* solve remaining facts *)
1762    -   solve_facts.
1763 Qed.
```

Glossary

address In cryptocurrency, an address is a unique identifier for a specific wallet or account on a blockchain network. Addresses are used to send and receive cryptocurrency transactions and can be thought of as similar to an email address or bank account number.

Typically, an address consists of a string of characters and is generated by the wallet software or exchange that the user is using. The format of an address varies depending on the blockchain network it is on, but they all serve the same purpose: to securely and uniquely identify a specific wallet or account. 11, 13, 92

arbitrage In the context of cryptocurrency, arbitrage refers to the practice of taking advantage of price differences between different decentralized exchanges (DEXs) or different trading pairs within the same DEX to make a profit. In an AMM, users can buy and sell tokens at prices that are determined algorithmically, based on the supply and demand for the tokens in question. If a token is priced differently on different DEXs, an arbitrage trader can buy the token on the DEX where it is underpriced, and then sell it on the DEX where it is overpriced. By doing so, the trader can make a profit from the price difference between the two DEXs, while also contributing to a more efficient market by bringing the prices closer together. 17, 59, 69

automated market maker An automated market maker (AMM) is a type of decentralized exchange (DEX) that uses a mathematical formula to determine the price of assets being traded on the platform. Unlike traditional centralized exchanges, which match buyers and sellers and take a cut of the transaction as a fee, AMMs use algorithms to set the price of assets based on the supply and demand of those assets on the platform. This allows for near-instant trades without the need for an intermediary to match buyers and sellers, and it provides users with more control over their trades. AMMs are commonly used in decentralized finance (DeFi) applications, and they play an important role in providing liquidity and enabling the trading of digital assets. 7, 227, 229

Binance Smart Chain Binance Smart Chain (BSC) is a high-performance blockchain developed by the Binance exchange. It is designed to support decentralized applications and decentralized finance (DeFi)

use cases with fast and low-cost transactions. BSC is built on top of the Ethereum Virtual Machine (EVM) and uses a similar smart contract language to Ethereum. However, it is optimized for faster and cheaper transactions than the Ethereum network. BSC is compatible with Ethereum-based tools and applications, but it offers faster confirmation times and lower fees. This allows for the creation of new DeFi applications and the migration of existing ones from Ethereum to Binance Smart Chain. 229

collateralized In the context of decentralized finance (DeFi), a collateralized asset is used to secure a loan or a borrowing agreement. The borrower puts up the collateral as a guarantee that they will be able to repay the loan. If the borrower defaults, the lender can seize the collateral to recover their funds. The value of the collateral must be equal to or greater than the value of the loan in order to provide sufficient security to the lender. 60

constant function market maker A constant function market maker (CFMM) is a type of smart contract used in decentralized finance (DeFi) to provide liquidity to a market. It operates by allowing users to deposit and trade assets through the contract, and it uses a pre-defined set of rules to manage the supply and demand for the assets in the market. The CFMM acts as a market maker, providing liquidity and facilitating trades by constantly adjusting the price of assets based on the current supply and demand. The constant function in its name refers to the fact that the rules for price determination and liquidity management are pre-programmed and executed automatically by the smart contract, rather than being subject to changes based on the decisions of any individual or group. 17, 229

constituent token A constituent token of a pool is a (usually non-fungible) token which can be pooled in exchange for a (usually fungible) token, called a pool token. The pool token can usually be redeemed for underlying constituent tokens at an exchange rate set by the pool contract. 38, 59, 223

cross-chain bridge A cross-chain bridge is a system that allows for the transfer of assets or information between different blockchain networks. This enables the exchange of tokens, coins, and other digital assets between different blockchains, even if they have different consensus algorithms, security models, and underlying infrastructure. Cross-chain bridges are designed to facilitate interoperability between different blockchain networks and create new use cases for decentralized applications. They can also help to address scalability and liquidity issues, as they allow users to access a broader pool of assets and to move their assets between different blockchains as needed. 7, 13, 15, 92, 118, 121, 135, 136

cross-chain swaps Cross-chain swaps are a type of decentralized exchange that enables users to trade cryptocurrencies between different blockchain networks without the need for intermediaries or centralized exchanges. Unlike traditional exchanges that operate within a single blockchain network, cross-chain swaps allow users to exchange cryptocurrencies across multiple blockchains, including those with different consensus mechanisms or smart contract languages. This is achieved through the use of atomic

swaps, which enable users to exchange cryptocurrencies in a trustless and secure manner, without the need for a third party to facilitate the trade. Cross-chain swaps are becoming an increasingly popular tool in the decentralized finance (DeFi) ecosystem, as they enable users to access liquidity across different blockchain networks and participate in a wider range of trading opportunities. 118

crypto insurance protocols Crypto insurance protocols are decentralized financial applications that provide insurance coverage for assets in the crypto space. These protocols are designed to mitigate the risk of loss for investors in the event of unexpected events such as hacking, smart contract vulnerabilities, or market crashes. They typically work by pooling funds from multiple investors, who then share the risk of loss. The coverage offered by crypto insurance protocols is typically purchased using the same cryptocurrency that is being insured, and the price of coverage is based on the level of risk associated with the asset being insured. The decentralized nature of these protocols ensures that the funds are secure and that claims can be processed quickly and transparently. 7

crypto lending Crypto lending refers to the practice of lending or borrowing digital assets, typically cryptocurrencies, in a decentralized manner, typically through a decentralized finance (DeFi) platform. In a crypto lending system, users can deposit their digital assets as collateral and then borrow other digital assets, typically at a higher value than the collateral deposited. The borrowed assets can then be used for various purposes, such as trading, investment, or speculative activities.

Lenders, on the other hand, earn interest on their digital assets by lending them out to borrowers. They also benefit from the potential appreciation of the digital assets they hold, as well as the interest earned on the loans they issue.

Crypto lending platforms typically use smart contracts to manage the lending and borrowing process, allowing for automatic and transparent execution of loan agreements and interest payments. This eliminates the need for intermediaries and enables crypto lending to operate in a trustless and decentralized manner.

The crypto lending market has grown rapidly in recent years, with many DeFi protocols offering lending and borrowing services for a wide range of digital assets. These services have become increasingly popular due to their potential for high returns, their accessibility, and the increasing number of cryptocurrencies and other digital assets available for lending and borrowing. 7, 14

Curve Curve is a decentralized exchange (DEX) on the Ethereum blockchain that specializes in stablecoins. Curve allows users to trade a variety of stablecoins, such as USDC, DAI, and USDT, in a decentralized and trustless manner. It provides a platform for users to exchange stablecoins with low slippage, meaning that the price of the asset that a user receives in a trade is close to the expected price, resulting in minimal losses. The exchange operates through smart contracts on the Ethereum network, and the exchange rate is determined by a liquidity pool of the various stablecoins. The platform aims to provide a secure, efficient, and user-friendly environment for trading stablecoins. 14

DAI DAI is a stablecoin that is pegged to the value of the US dollar. It's designed to maintain a stable value, regardless of the volatility of other cryptocurrencies. DAI is implemented as an Ethereum-based smart contract, and it is created through a process called "locking" or "staking" ETH, where the ETH is held in a smart contract as collateral. The amount of DAI created is determined by the amount of ETH staked and the value of the ETH collateral. DAI is designed to be used as a medium of exchange in the decentralized finance (DeFi) ecosystem. 104, 118, 216

decentralized application A decentralized application (dApp) is a software application that runs on a decentralized network and is not controlled by any central authority. Instead of relying on a centralized server, dApps run on a peer-to-peer network, using blockchain technology to secure transactions and ensure that the underlying code is tamper-proof. This makes dApps more secure, transparent, and resistant to censorship, and can help to remove intermediaries, reducing costs and increasing efficiency. Decentralized applications can be used for a variety of purposes, ranging from financial applications like exchanges, to gaming platforms and social media sites. 224, 229

decentralized autonomous organization A decentralized autonomous organization (DAO) is a type of organization that is run using rules encoded as computer programs on a blockchain network. DAOs are designed to be transparent, autonomous, and decentralized, meaning that they operate without the need for intermediaries or a central authority. Instead, the rules and decision-making processes of a DAO are encoded into smart contracts on the blockchain, and the operations of the DAO are executed automatically according to these rules. Members of a DAO participate by holding and voting with tokens that represent ownership in the organization. This allows DAOs to operate in a decentralized manner, with decisions being made through a consensus mechanism, such as voting or staking. DAOs are often used in the decentralized finance (DeFi) ecosystem as a way to create and manage decentralized investment funds, decentralized exchanges, or other types of decentralized organizations. They offer a way to create organizations that are transparent, borderless, and operate without the need for intermediaries, which has the potential to disrupt traditional organizational structures and business models. 104, 226, 229

decentralized exchange A decentralized exchange (DEX) is a type of cryptocurrency exchange that operates on a decentralized network, typically built on blockchain technology. Unlike centralized exchanges, which are operated by a single entity and hold users' assets on their servers, decentralized exchanges allow for peer-to-peer trading of cryptocurrencies without the need for a central authority.

In a decentralized exchange, users retain control of their private keys and assets, as transactions are executed directly between users on the network. This eliminates the need for intermediaries, such as centralized exchanges, and reduces the risk of theft or loss of funds.

Decentralized exchanges typically use smart contracts to automate the trading process, enabling the seamless and transparent execution of trades without the need for intermediaries. They also provide

users with increased privacy and security, as the transactions are recorded on a public ledger that is transparent and tamper-proof.

Overall, decentralized exchanges offer a more secure and transparent way to trade cryptocurrencies and other digital assets, and are becoming increasingly popular as the demand for decentralized financial services continues to grow. 7, 12, 142, 214, 216, 225, 226, 229

decentralized finance Decentralized Finance (DeFi) refers to a growing ecosystem of financial applications and services built on decentralized, open-source blockchain technology, such as Ethereum. DeFi applications aim to offer financial services that are accessible to everyone, regardless of geographical location or financial status, by leveraging the transparency, security, and immutability of blockchain technology.

DeFi services include a wide range of financial products and services, such as lending and borrowing platforms, stablecoins, decentralized exchanges, tokenized assets, yield farming, insurance, and more. These services are designed to operate in a decentralized, trustless, and permissionless manner, meaning that users can access these services without relying on intermediaries, such as traditional banks or financial institutions.

The DeFi movement is driven by the belief that financial services should be accessible to everyone, and that the traditional financial system is not serving the needs of a large portion of the global population. By leveraging blockchain technology, DeFi services aim to offer financial services that are transparent, secure, and accessible to all.

Overall, DeFi represents a new and rapidly evolving paradigm in the world of finance, offering innovative financial services and opportunities for growth, while also challenging the traditional financial system and the role of intermediaries in financial services. 7, 15, 17, 117, 215, 226, 229

decentralized governance Decentralized governance refers to a system of decision-making and administration in which power is distributed among a network of individuals or entities, rather than being centralized in a single governing body. In the context of blockchain technology and cryptocurrencies, decentralized governance typically refers to a system in which stakeholders collectively make decisions about the direction and management of a particular network or protocol, often through the use of decentralized voting mechanisms or on-chain proposals. This type of governance aims to be more transparent, fair, and secure than traditional centralized governance models, as it allows for more equal participation and reduces the risk of single points of failure or control. 11

DEX aggregator A DEX aggregator is a platform that enables users to trade cryptocurrencies across multiple decentralized exchanges (DEXs) at once, using complex algorithms to find the best prices across all supported exchanges. DEX aggregators offer a range of benefits, including improved pricing, access to larger pools of liquidity, better user interfaces, and faster trade execution times. By providing a convenient and efficient way to access the benefits of decentralized exchanges, DEX aggregators are

becoming increasingly popular in the decentralized finance (DeFi) ecosystem. 118

economic attack An economic attack on a smart contract refers to an attempt to exploit a vulnerability in the contract's design or implementation that results in financial gain for the attacker at the expense of the contract's users or the underlying network. This can take many forms, such as a front-running attack, where an attacker submits a transaction that takes advantage of an existing transaction's delay in being processed on the blockchain, or flash loan attack, where an attacker borrows a large amount of funds for a short period of time and uses them to manipulate the contract's state in a way that generates profits. Economic attacks on smart contracts can have significant consequences, as they can disrupt the normal functioning of the contract, compromise the security of user funds, and undermine the overall trust in the decentralized ecosystem. 10, 17

entrypoint function A contract entrypoint function is a function in a smart contract that can be called directly by an external caller. These functions serve as the entry point for users or other contracts to interact with the smart contract and execute its logic. In other words, they are the public facing functions that allow users to interact with the smart contract and make changes to its state.
Examples of entrypoint functions in a smart contract might include functions to transfer funds, mint new tokens, vote on proposals, and access information about the state of the contract. In general, the entrypoint functions are defined by the contract developer and are intended to provide a way for external users to interact with the contract in a meaningful way. 8, 37

Ethereum Ethereum is a decentralized, open-source blockchain platform that enables the creation and execution of smart contracts and decentralized applications (dapps). It was created in 2015 by Vitalik Buterin and has since become one of the largest and most widely used blockchain platforms in the world. Ethereum has its own cryptocurrency called Ether (ETH), which is used to pay for transactions and computational services on the network. The platform supports a Turing-complete programming language, which allows developers to build a wide variety of applications, from decentralized exchanges and prediction markets to games and social networks. One of Ethereum's main goals is to create a decentralized and transparent computing platform, which can be used to build trust in digital systems and eliminate the need for intermediaries. 14, 19, 222, 229

flash loan A flash loan is a type of loan in decentralized finance (DeFi) that allows a user to borrow funds for a short period of time, typically within seconds, without collateral and with a very low interest rate. The loan is repaid automatically at the end of the short time period. Flash loans are used for a variety of purposes in DeFi, including exploiting market inefficiencies and executing arbitrage strategies. They are considered to be highly risky due to the short repayment window and the lack of collateral, and are only available on decentralized lending protocols that support the feature. 11, 12, 14, 15

flash loan attack A flash loan attack is a type of exploit that allows an attacker to borrow funds from a DeFi protocol for a very short period of time, typically just a few milliseconds, without having to put up any collateral. The attacker can then use these funds to execute a trade, perform an arbitrage, or carry out some other type of transaction that takes advantage of the momentary imbalance in the market. Since the loan is only for a brief period of time, the attacker can return the borrowed funds before they become due, effectively "borrowing" the funds for free. If the attack is successful, the attacker can make a profit while leaving the DeFi protocol and its users with the losses. 11, 12, 118, 219

front-running attack A front-running attack on a blockchain occurs when a malicious user discovers a swap transaction after it has been broadcasted but before it has been finalized and reorders transactions to their benefit. 9, 16, 17, 36, 37, 219

fungible See fungible token. 37, 215, 223

fungible token A fungible token is a type of digital asset that represents a unit of value that is interchangeable with other units of the same value. This means that each unit of the token is identical and interchangeable with any other unit. An example of a fungible token is a cryptocurrency, such as Bitcoin or Ethereum, where each unit of the token represents a certain amount of value, and all units are interchangeable and have the same value. 220, 223, 229

gas Gas is a term used to describe the fee required to process a transaction on the Ethereum blockchain. In the Ethereum network, all transactions and smart contract executions must be processed by nodes, which are collectively known as the network's infrastructure. When a user submits a transaction, they must include a fee in the form of gas, which is paid to the nodes that process the transaction. The amount of gas required for a transaction depends on the computational complexity of the operation being performed. The purpose of requiring gas is to ensure that the network is not congested by too many transactions, and to incentivize nodes to process transactions in a timely manner. 14, 76, 118

governance token A governance token is a type of cryptocurrency that is used to facilitate decision-making in a decentralized organization, such as a decentralized autonomous organization (DAO). Holders of governance tokens are given voting rights that allow them to participate in important decisions, such as changes to the organization's protocol or the allocation of funds. Governance tokens can also be used to propose and vote on changes to the organization's governance structure, as well as to elect or recall members of the organization's leadership. By giving holders a say in the organization's operations, governance tokens help to promote a more democratic and decentralized decision-making process within decentralized organizations. 104, 118

indifference curve An indifference curve, also referred to as an isoutility curve, is a graphical representation of combinations of two goods or services that yield the same level of satisfaction or utility to an

individual or economic agent. It is typically used in microeconomic theory to model preferences and choice behavior. On a graph, each indifference curve is a curve that represents all the combinations of goods or services that the individual would be indifferent between, meaning that they would be equally satisfied with any combination of goods or services that lie on that curve. The slope of the curve indicates the rate at which the individual is willing to trade one good or service for another, and the distance between curves represents the level of satisfaction or utility associated with each curve. Indifference curves are used to analyze consumer behavior and to determine the optimal consumption bundle for an individual given their preferences and budget constraints. 8

initial coin offering An initial coin offering (ICO) is a type of fundraising mechanism in the cryptocurrency and blockchain space. An ICO involves the creation and sale of a new cryptocurrency or token, with the funds raised being used to develop a new blockchain project or product. The tokens sold in an ICO can typically be traded on cryptocurrency exchanges, and their value is tied to the success of the project or product being developed.

ICOs were popular in the early days of the cryptocurrency space, but the regulatory environment has since become more stringent, and many countries have placed restrictions on ICOs or banned them outright. Despite this, some projects continue to use ICOs as a means of raising funds, and investors are still attracted to ICOs due to the potential for high returns. However, ICOs are considered to be high-risk investments, and there have been numerous cases of ICOs that have turned out to be scams or have failed to deliver on their promises. As a result, potential investors in ICOs should exercise caution and perform due diligence before investing. 226

liquidity In the context of decentralized finance (DeFi), liquidity refers to the availability of assets to be bought and sold in a market. A market is considered to have high liquidity when there are many buyers and sellers, and the assets can be bought and sold easily and quickly, with minimal price slippage. In DeFi, liquidity is often provided by individuals or entities known as liquidity providers, who deposit assets into a liquidity pool in exchange for a share of the pool, known as a liquidity token. This liquidity makes it easier for other users to trade assets on decentralized exchanges, as there are always buyers and sellers available to take the other side of the trade. 12, 37

liquidity pool A liquidity pool is a collection of assets that are combined to provide greater liquidity for the underlying assets. In the context of decentralized finance (DeFi), a liquidity pool is typically created by pooling together cryptocurrencies, stablecoins, or other assets in a smart contract on a blockchain. This smart contract allows users to trade the underlying assets with each other and the price of each asset is determined by supply and demand. By providing liquidity to the pool, users can earn rewards in the form of fees from trades that are executed on the platform. The amount of rewards earned is proportional to the share of the pool's liquidity that a user provides. These rewards can be in the

form of governance tokens, which give users a say in how the platform is governed and how fees are distributed, or in the form of yield generated by the assets in the pool. 12

liquidity provider an investor (individual or institution) who funds a liquidity pool with crypto assets she owns to facilitate trading on the platform and earn passive income on her deposit. 8, 75

liquidity share Liquidity share refers to the proportional ownership of a pool of assets within a financial market, such as a cryptocurrency exchange. In decentralized finance (DeFi), liquidity is provided by users who deposit their assets into a common pool, which is then used to facilitate trades and provide stability to the market. Each user who contributes liquidity to the pool is entitled to a share of the pool proportional to the amount they have deposited. These shares are often represented as tokens, which can be traded on exchanges or used to participate in governance decisions for the pool. The value of the liquidity share is directly tied to the performance of the assets in the pool and can increase or decrease in value as the market moves. 12

LP token An LP token is a Liquidity Pool token. It is a type of token that represents ownership in a decentralized liquidity pool. Liquidity pools are an important component of many decentralized finance (DeFi) applications and are used to ensure that users can trade digital assets quickly and efficiently. The tokens are typically used to incentivize users to provide liquidity to the pool by allowing them to earn a share of the fees generated by the trades that take place within the pool. LP tokens are usually tradeable and can be bought and sold on various decentralized exchanges. They are often used as a way to invest in the liquidity of specific DeFi protocols or as a way to diversify one's portfolio within the DeFi ecosystem. 12, 60, 118

minting and burning privileges Minting and burning privileges refer to the ability to create (mint) new tokens or to destroy (burn) existing tokens in a token-based system. In the context of cryptocurrencies, minting privileges are often granted to the creators of a token or to certain trusted entities, such as a central authority or a set of validators, who can issue new tokens as needed. Burning privileges, on the other hand, give the ability to destroy tokens, which is often used to control the supply of a token and to maintain its value. For example, a token creator might choose to burn tokens that are lost or stolen to reduce the overall supply of tokens and prevent their value from being diluted. 38

non-fungible See non-fungible token. 215, 223

non-fungible token A non-fungible token (NFT) is a unique, indivisible digital asset that is recorded on a blockchain. Unlike fungible tokens, such as cryptocurrencies like Bitcoin and Ethereum, which are interchangeable and have a uniform value, NFTs are one-of-a-kind and cannot be divided or replicated. NFTs are typically used to represent ownership of digital assets such as artwork, collectibles, and virtual real estate, and they can be bought, sold, and traded like traditional assets. The ownership

and authenticity of NFTs is maintained through the use of cryptographic proof and immutability of the blockchain, making them ideal for representing unique, valuable assets in the digital world. See also fungible token. 222, 229

peg A peg in the context of digital assets refers to a mechanism that is used to maintain a fixed exchange rate between a cryptocurrency and a real-world asset, such as the US dollar or gold. This allows the cryptocurrency to maintain a stable value relative to the pegged asset, which can reduce price volatility and make it more useful for everyday transactions. Pegged cryptocurrencies are typically backed by a reserve of the asset they are pegged to, which helps to ensure the stability of the peg even during market fluctuations. Pegged cryptocurrencies can be used for a variety of purposes, including cross-border transactions, remittances, and as a store of value. 118

Polygon The Polygon Sidechain is a layer 2 scaling solution built on Ethereum, offering a framework for creating sidechains or Polygon chains. These sidechains are interoperable with Ethereum, addressing scalability issues by enabling faster, cost-effective transactions. Utilizing technologies like Proof of Stake (PoS) consensus, Polygon Sidechain provides high throughput and lower fees. It empowers developers to build DApps and smart contracts with the security of Ethereum while enjoying improved scalability. Users benefit from faster confirmations, reduced congestion, and enhanced scalability, making it ideal for DeFi, NFTs, gaming, and more.. 37

pool In the context of blockchains, the term "pool" can have a few different meanings.

One common use of the term "pool" in crypto refers to a "mining pool." In a mining pool, several miners work together to mine a cryptocurrency and share the rewards. By pooling their resources, the miners can increase their chances of successfully mining blocks and earning rewards.

Another use of the term "pool" in crypto refers to a "liquidity pool." In a decentralized finance (DeFi) context, a liquidity pool is a shared pool of assets that provides liquidity for a specific asset or market. Users can add their own assets to the pool and receive a proportional share of the pool's tokens, called "liquidity provider" or "LP" tokens. These tokens give users a share of the fees generated by the trades that occur in the pool.

In other contexts, the term "pool" may be used to refer to a shared collection of resources or services that are provided or managed by a network of computers, as opposed to a central authority. 18, 36–38, 215, 223

pool token The pool token of a pool is a (usually fungible) token which can be minted in exchange for pooling a (usually non-fungible) constituent token. The pool token can usually be redeemed for underlying constituent tokens at an exchange rate set by the pool contract. 38, 59, 60, 215, 223

price oracle In decentralized finance (DeFi), a price oracle is a source of truth for the current price of a cryptocurrency or other asset. A price oracle is necessary because DeFi applications are built on

decentralized, trustless networks like Ethereum, and they need a way to obtain accurate and up-to-date information about the prices of assets in order to function properly.

A price oracle is essentially a smart contract that retrieves information about the price of an asset from an external data source, such as an exchange, and provides it to other smart contracts in a standardized format. For example, a DeFi application might use a price oracle to determine the value of a collateral asset, or to calculate the interest rate on a loan.

In order for a price oracle to be effective, it must be reliable and secure, as well as resistant to tampering and manipulation. There are several different approaches to building price oracles, and different DeFi applications use different price oracles, depending on their specific needs. 12, 14, 118

smart contract A computer program that is stored on a blockchain and automatically executes the terms of a contract when certain conditions are met. Smart contracts do not require intermediaries to enforce their terms. They allow for the automatic and trustless exchange of assets, such as cryptocurrencies, without the need for intermediaries such as banks or lawyers. 7, 11

Solana Solana is a high-performance blockchain platform designed for decentralized applications and decentralized finance (DeFi) use cases. It was created with the goal of providing fast and scalable blockchain infrastructure for decentralized applications, enabling them to handle a high volume of transactions per second. Solana is designed to be energy-efficient and cost-effective, making it an attractive option for developers and users who want to participate in the decentralized finance ecosystem. Solana has its own native cryptocurrency, called SOL, which is used to pay for transaction fees and to participate in governance decisions on the network. 12

Solidity Solidity is a contract-oriented, high-level programming language for writing smart contracts that run on the Ethereum Virtual Machine (EVM). It was developed by the Ethereum Foundation and is influenced by C++, Python, and JavaScript. Solidity enables developers to write decentralized applications (dApps) on the Ethereum blockchain, which automatically execute when certain conditions are met. Solidity provides a syntax for defining data structures and functions, as well as handling errors, security, and program execution. The code written in Solidity can be compiled into bytecode that can be executed on the EVM and stored on the Ethereum blockchain. This makes Solidity a crucial part of the Ethereum ecosystem and a key tool for building decentralized financial (DeFi) applications and other blockchain-based systems. 19

stablecoin A stablecoin is a type of cryptocurrency in the class of synthetics that is designed to maintain a stable value relative to a specific asset or basket of assets. The most common type of stablecoin is pegged to the US dollar, so that each stablecoin unit is worth exactly one US dollar. The goal of a stablecoin is to provide a more stable store of value compared to other cryptocurrencies, which are often subject to significant price swings.

Stablecoins achieve price stability through various mechanisms, including being backed by assets such as US dollars held in reserve, or through algorithmic mechanisms that adjust the supply of the stablecoin in response to changes in demand. For example, if demand for a stablecoin increases, its algorithm may automatically issue more units to meet that demand, while if demand decreases, the algorithm may automatically redeem units, effectively reducing the supply.

Stablecoins are used in various applications, including as a medium of exchange in decentralized finance (DeFi) protocols, as a hedge against cryptocurrency volatility, or as a way to move funds between different blockchain networks. They provide a way for users to transfer value and participate in financial transactions without being exposed to the price volatility of cryptocurrencies like Bitcoin or Ethereum. 7, 11, 15, 104, 216

swap A swap in the context of cryptocurrency refers to an exchange of one token for another token, either on a decentralized exchange (DEX) or on a centralized exchange (CEX). In DeFi, the most common type of swap is a token-to-token swap, which allows users to exchange one token for another token directly, without the need for intermediaries such as brokers. The price of the swap is determined by the current market price of the tokens being traded, and users can take advantage of price differentials between different exchanges or pools to earn a profit. Additionally, some protocols offer more sophisticated swaps that allow users to trade one token for a basket of other tokens, or to swap one token for another token with a leveraged position, among other things. 17

synthetics Synthetics are digital assets that are designed to track the price of another asset, such as a traditional financial instrument, a commodity, or another cryptocurrency. Synthetics allow traders to gain exposure to the price movements of these underlying assets without actually owning them, which can be useful for hedging against price volatility or for speculative purposes. Synthetic assets can be created and traded on decentralized exchanges using decentralized finance (DeFi) protocols. These protocols typically use algorithms and smart contracts to create and manage the synthetic assets, which are typically tokenized versions of the underlying assets. Synthetics have become a popular tool in the DeFi space for users who want to trade and invest in a wide range of assets, and they offer a way to access exposure to traditional financial instruments in a decentralized and permissionless manner. 7, 118, 224

Tezos Tezos is a decentralized blockchain platform that was created with a focus on governance and upgradeability. It uses a proof-of-stake consensus mechanism and is designed to be self-amending, meaning that its protocol can be updated without the need for a hard fork. Tezos also has a unique governance mechanism that allows token holders to participate in the decision-making process and propose and vote on protocol upgrades. Additionally, Tezos supports smart contracts and decentralized applications, making it a platform for a wide range of use cases, from digital asset management to decentralized finance. 19, 229

token A token is a digital asset that represents a unit of value and can be traded on a blockchain platform. Tokens can serve a variety of purposes, such as representing ownership in a company, access to a particular product or service, or as a medium of exchange. In the context of cryptocurrency, tokens are often used to raise funds through initial coin offering (ICOs), or as a way to reward network participants for performing certain tasks or contributing resources. Tokens can be created using smart contracts on blockchain platforms, such as Ethereum, and they are typically stored and traded using a digital wallet. Tokens can be used for a wide range of applications, from creating decentralized autonomous organizations (DAOs) to enabling new forms of micropayments, and they are a key component of the decentralized finance (DeFi) ecosystem. Tokens are standardized on most blockchains, for example the ERC20 and ERC721 standards on the Ethereum blockchain, and the FA1.2 and FA2 standards on the Tezos blockchain. 12, 17, 38, 104, 118, 135, 220, 222, 225, 229

tokenized carbon credits Tokenized carbon credits are digital assets that represent a unit of carbon dioxide emissions reduction. These tokens can be bought and sold on a blockchain, and their ownership can be easily verified and transferred. The idea behind tokenized carbon credits is to provide a market-based mechanism for reducing carbon emissions, where individuals and businesses can purchase credits to offset their carbon footprint. The carbon credits can be generated through a variety of methods, such as investing in renewable energy projects, planting trees, or improving energy efficiency. The creation and transfer of these tokens is managed by smart contracts on a blockchain, which ensures that the carbon credits are verifiable and tamper-proof. 36

tokenomics Tokenomics, a portmanteau of "token" and "economics," refers to the study and design of the economic system and incentives behind a cryptocurrency or digital token. It encompasses the rules, policies, and mechanisms that determine the creation, distribution, usage, and value of the tokens within a blockchain or decentralized ecosystem. Tokenomics plays a crucial role in determining the success and sustainability of a cryptocurrency or digital token project. 118

transaction A transaction on a blockchain is a record of a transfer of data or value between parties that is recorded on a decentralized ledger. Transactions are verified and processed by network participants, also known as nodes, using consensus algorithms. The transactions are packaged into blocks and linked to previous blocks, forming a chain of blocks, which creates an immutable and tamper-proof record of all transactions on the network. The verified transactions are then added to the blockchain and can be viewed by anyone on the network. The term "transaction" is often used to refer specifically to financial transactions on a blockchain, but the concept can be applied to any type of data exchange that is recorded on the blockchain. 11

Uniswap Uniswap is a decentralized exchange (DEX) built on the Ethereum blockchain. It allows users to trade cryptocurrencies directly with each other, without the need for a centralized intermediary.

Uniswap is unique in that it operates as an automated market maker (AMM) rather than as a traditional order book-based exchange. This means that instead of relying on buyers and sellers to match their orders, Uniswap uses a mathematical formula to determine the price of assets based on their supply and demand. Uniswap has become popular among cryptocurrency traders and investors due to its ease of use, high liquidity, and low fees. 75

USDC USDC (USD Coin) is a stablecoin, which is a type of cryptocurrency that is pegged to the value of the US dollar. The aim of USDC is to provide a stable and secure store of value that is backed by the US dollar. The idea is to create a digital asset that is less volatile than other cryptocurrencies, and that can be used to facilitate transactions and payments in a fast, secure, and decentralized manner. USDC is issued by regulated financial institutions and is fully collateralized, which means that the amount of USDC in circulation is always backed by an equivalent amount of US dollars held in reserve. This makes USDC a popular choice for individuals and organizations looking for a stable form of digital currency for use in financial transactions and investments. 14, 142, 216

USDT USDT is a stablecoin, a type of cryptocurrency that is pegged to the value of the US dollar. The idea behind stablecoins is to provide a stable store of value that can be used in place of traditional fiat currencies in digital currency exchanges and transactions. USDT is issued by Tether Limited and is designed to maintain a 1-to-1 value with the US dollar. This means that for every USDT token in circulation, there is a corresponding US dollar held in reserve. This helps to minimize the volatility that is often associated with other cryptocurrencies, making USDT a popular choice for traders and investors who want to minimize risk. 14, 70, 216

wallet A wallet in crypto refers to a software application or device that provides a secure way to store and manage cryptocurrency assets, such as Bitcoin, Ethereum, and others. A wallet can be used to receive, send, and manage these assets. The wallet is associated with a unique public address, which is used to receive funds, and a private key, which is used to send funds and control access to the assets stored in the wallet. The private key must be kept confidential and should not be shared with anyone. The security of the crypto assets stored in a wallet depends on the security measures in place, such as the use of strong passwords, two-factor authentication, and hardware wallets. 42, 45, 52, 214

yield aggregator A yield aggregator in cryptocurrency is a type of smart contract that allows users to automatically aggregate their crypto assets into various yield-generating pools. The purpose of a yield aggregator is to maximize the yield received by the user on their cryptocurrency investments. Yield aggregators typically work by automatically re-allocating funds to the highest yielding pools, taking into account various factors such as fees, liquidity, and performance. By using a yield aggregator, users can simplify the process of earning yield on their crypto assets and potentially earn a higher return than they would by manually selecting yield-generating pools. 12, 14, 15, 118

yield farming Yield farming is the practice of providing liquidity to decentralized finance (DeFi) protocols in exchange for rewards in the form of interest or tokens. The rewards come from the fees generated by the protocols, and they are distributed to liquidity providers as a way of incentivizing them to provide liquidity and ensure the stability of the platform. Yield farming typically involves depositing funds into a liquidity pool, which is then used to provide liquidity for trading activities on the platform. The rewards earned by the liquidity providers are proportional to their share of the total liquidity in the pool. Yield farming has become popular in the DeFi space as a way for users to earn passive income on their digital assets, and it has also been a driving force behind the growth of the DeFi ecosystem. 7, 14, 118

Acronyms

AMM Automated market maker. 7–9, 11, 13, 15–18, 37, 59, 72, 73, 75, 118, 142, 227

BSC Binance Smart Chain. 11, 12

CFMM constant function market maker. 17

DAO decentralized autonomous organization. 104

dApp decentralized application. 224

DeFi decentralized finance. 11, 14, 15, 17, 59, 72, 73, 117, 118, 120, 141–143, 145, 222, 224

DEX Decentralized exchange. 12, 15, 118, 216, 225, 226

ERC20 A token standard for fungible tokens on the Ethereum blockchain. 37, 226

ERC721 A token standard for non-fungible tokens on the Ethereum blockchain. 226

ETH The native token of Ethereum. 14

FA1.2 A token standard for tokens on the Tezos blockchain. FA2 stands for "Financial Asset 1.2.". 226

FA2 A token standard for tokens on the Tezos blockchain. FA2 stands for "Financial Asset 2.". 118, 226

NFT See non-fungible token. 37, 135, 222

XTZ Tezos. 70

Bibliography

[1] 3110 Coq Tactics Cheatsheet. https://www.cs.cornell.edu/courses/cs3110/2018sp/a5/coq-tactics-cheatsheet.html.

[2] Beanstalk Governance Exploit. https://bean.money/blog/beanstalk-governance-exploit.

[3] Certora Prover. https://www.certora.com.

[4] Curve.fi. https://curve.fi/. Accessed March 2023.

[5] DeFi-type projects received the highest number of attacks in 2022: Report. https://cointelegraph.com/news/defi-type-projects-received-the-highest-number-of-attacks-in-2022-report.

[6] Dexter2 Specification. https://gitlab.com/dexter2tz/dexter2tz/-/blob/master/docs/informal-spec/dexter2-cpmm.md.

[7] Dexter2 Specification (Mi-Cho-Coq). https://gitlab.com/nomadic-labs/mi-cho-coq/-/blob/dexter-verification/src/contracts_coq/dexter_spec.v.

[8] Dexter2 Verification (Mi-Cho-Coq). https://gitlab.com/nomadic-labs/mi-cho-coq/-/merge_requests/71.

[9] Disrupting financial markets - Decentralised Exchanges. https://thepaypers.com/expert-opinion/disrupting-financial-markets-decentralised-exchanges–1250818.

[10] EIP-2535: Diamonds, Multi-Facet Proxy. https://eips.ethereum.org/EIPS/eip-2535.

[11] Murmuration: A generalizable dao for tezos; powers kolibri governance. https://github.com/Hover-Labs/murmuration. Accessed March 2023.

[12] PancakeBunny Faces FlashLoan Exploit Resulting In 97% Correction. https://www.bsc.news/post/pancakebunny-faces-flashloan-exploit-resulting-in-97-correction.

[13] Proxy Upgrade Pattern - OpenZeppelin Docs. https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies.

[14] Toucan Whitepaper. https://docs.toucan.earth/. Accessed March 2023.

[15] UN Supports Blockchain Technology for Climate Action — UNFCCC. https://unfccc.int/news/un-supports-blockchain-technology-for-climate-action.

[16] Uniswap protocol. https://uniswap.org/. Accessed March 2023.

[17] Writing Upgradeable Contracts - OpenZeppelin Docs. https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable.

[18] Contract upgrade anti-patterns. https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/, September 2018.

[19] Improving front running resistance of x*y=k market makers - Decentralized exchanges. https://ethresear.ch/t/improving-front-running-resistance-of-x-y-k-market-makers/1281, March 2018.

[20] Formal Verification Report: Tezos Dexter V2 Contract. Runtime Verification Inc., June 2021.

[21] Dexter2 Specification (K Framework). Runtime Verification Inc., August 2022.

[22] K-Michelson: A Michelson Semantics. Runtime Verification Inc., August 2022.

[23] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–15, June 2021. doi:10.1109/CSF51468.2021.00048.

[24] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core, 2021.

[25] Andreas A. Aigner and Gurvinder Dhaliwal. Uniswap: Impermanent loss and risk profile of a liquidity provider. *arXiv preprint arXiv:2106.14404*, 2021. arXiv:2106.14404.

[26] Hamda Al-Breiki, Muhammad Habib Ur Rehman, Khaled Salah, and Davor Svetinovic. Trustworthy blockchain oracles: Review, comparison, and open research challenges. *IEEE Access*, 8:85675–85685, 2020.

[27] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 66–77, New York, NY, USA, January 2018. Association for Computing Machinery. doi:10.1145/3167084.

[28] Guillermo Angeris, Akshay Agrawal, A. Evans, T. Chitra, and Stephen P. Boyd. Constant Function Market Makers: Multi-Asset Trades via Convex Optimization. 2021.

[29] Guillermo Angeris and Tarun Chitra. Improved price oracles: Constant function market makers. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 80–91, 2020.

[30] Guillermo Angeris, Tarun Chitra, and Alex Evans. When Does The Tail Wag The Dog? Curvature and Market Making. *Cryptoeconomic Systems*, 2(1), June 2022. doi:10.21428/58320208.e9e6b7ce.

[31] Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An Analysis of Uniswap markets. *Cryptoeconomic Systems*, 0(1), April 2021. doi:10.21428/58320208.c9738e64.

[32] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. Extracting smart contracts tested and verified in Coq. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, pages 105–121, New York, NY, USA, January 2021. Association for Computing Machinery. doi:10.1145/3437992.3439934.

[33] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. Extracting functional programs from Coq, in Coq. *Journal of Functional Programming*, 32:e11, 2022/ed. doi:10.1017/S0956796822000077.

[34] Danil Annenkov, Mikkel Milo, and Bas Spitters. Code Extraction from Coq to ML-like languages. page 4.

[35] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. ConCert: A smart contract certification framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 215–228, New York, NY, USA, January 2020. Association for Computing Machinery. doi:10.1145/3372885.3373829.

[36] Danil Annenkov and Bas Spitters. Deep and shallow embeddings in Coq. TYPES, 2019.

[37] Pedro Antonino, Juliandson Ferreira, Augusto Sampaio, and A. W. Roscoe. Specification is Law: Safe Creation and Upgrade of Ethereum Smart Contracts. In *Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings*, pages 227–243. Springer, 2022.

[38] Jun Aoyagi. Liquidity Provision by Automated Market Makers, May 2020. doi:10.2139/ssrn.3674178.

[39] Jun Aoyagi and Yuki Ito. Liquidity Implication of Constant Product Market Makers. *SSRN Electronic Journal*, January 2021. doi:10.2139/ssrn.3808755.

[40] Andrea Asperti and Giuseppe Longo. *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist*. MIT press, 1991.

[41] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, Lecture Notes in Computer Science, pages 164–186, Berlin, Heidelberg, 2017. Springer. doi:10.1007/978-3-662-54455-6_8.

[42] Avraham Eisenberg [@avi_eisen]. Mango Markets Exploit, October 2022.

[43] Steve Awodey. Univalence as a principle of logic. *Indagationes Mathematicae*, 29(6):1497–1510, December 2018. doi:10.1016/j.indag.2018.01.011.

[44] Sarah Azouvi and Alexander Hicks. SoK: Tools for Game Theoretic Models of Security for Cryptocurrencies, February 2020. arXiv:1905.08595, doi:10.48550/arXiv.1905.08595.

[45] Morena Barboni, Andrea Morichetta, and Andrea Polini. Smart Contract Testing: Challenges and Opportunities. In *2022 IEEE/ACM 5th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 21–24, May 2022. doi:10.1145/3528226.3528370.

[46] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch Lafuente. SoK: Lending Pools in Decentralized Finance. In Matthew Bernhard, Andrea Bracciali, Lewis Gudgeon, Thomas Haines, Ariah Klages-Mundt, Shin'ichiro Matsuo, Daniel Perez, Massimiliano Sala, and Sam Werner, editors, *Financial Cryptography and Data Security. FC 2021 International Workshops*, Lecture Notes in Computer Science, pages 553–578, Berlin, Heidelberg, 2021. Springer. doi:10.1007/978-3-662-63958-0_40.

[47] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch Lafuente. Towards a Theory of Decentralized Finance. In Matthew Bernhard, Andrea Bracciali, Lewis Gudgeon, Thomas Haines, Ariah Klages-Mundt, Shin'ichiro Matsuo, Daniel Perez, Massimiliano Sala, and Sam Werner, editors, *Financial Cryptography and Data Security. FC 2021 International Workshops*, Lecture Notes in Computer Science, pages 227–232, Berlin, Heidelberg, 2021. Springer. doi:10.1007/978-3-662-63958-0_20.

[48] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. A Theory of Automated Market Makers in DeFi. In Ferruccio Damiani and Ornela Dardha, editors, *Coordination Models and Languages*, Lecture Notes in Computer Science, pages 168–187, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-78142-2_11.

[49] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. Maximizing Extractable Value from Automated Market Makers, July 2022. arXiv:2106.01870, doi:10.48550/arXiv.2106.01870.

[50] Massimo Bartoletti and Roberto Zunino. Formal models of bitcoin contracts: A survey. *Frontiers in Blockchain*, 2:8, 2019.

[51] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT library: A formalization of homotopy type theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 164–172, 2017.

[52] Carsten Baum, James Hsin-yu Chiang, Bernardo David, Tore Kasper Frederiksen, and Lorenzo Gentile. SoK: Mitigation of Front-running in Decentralized Finance, 2021.

[53] Rob Behnke. Explained: The NowSwap Protocol Hack (September 2021). https://halborn.com/explained-the-nowswap-protocol-hack-september-2021/, September 2021.

[54] Beosin. Beosin — Global Web3 Security Report 2022, January 2023.

[55] Jan Arvid Berg, Robin Fritsch, Lioba Heimbach, and Roger Wattenhofer. An Empirical Study of Market Inefficiencies in Uniswap and SushiSwap. *arXiv preprint arXiv:2203.07774*, 2022. arXiv:2203.07774.

[56] Martin Berger. Specification and verification of meta-programs. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, pages 3–4, Philadelphia Pennsylvania USA, January 2012. ACM. doi:10.1145/2103746.2103750.

[57] Martin Berger and Laurence Tratt. Program Logics for Homogeneous Meta-programming. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science, pages 64–81, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-17511-4_5.

[58] Bruno Bernardo, Raphaël Cauderlier, Guillaume Claret, Arvid Jakobsson, Basile Pesin, and Julien Tesson. Making Tezos Smart Contracts More Reliable with Coq. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, Lecture Notes in Computer Science, pages 60–72, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-61467-6_5.

[59] Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-Cho-Coq, a Framework for Certifying Tezos Smart Contracts. In Emil Sekerinski, Nelma Moreira, José N. Oliveira, Daniel Ratiu, Riccardo Guidotti, Marie Farrell, Matt Luckcuck, Diego Marmsoler, José Campos, Troy Astarte, Laure Gonnord, Antonio Cerone, Luis Couto, Brijesh Dongol, Martin Kutrib, Pedro Monteiro, and David Delmas, editors, *Formal Methods. FM 2019 International Workshops*, Lecture Notes in Computer Science, pages 368–379, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-54994-7_28.

[60] Bruno Bernardo, Raphaël Cauderlier, Basile Pesin, and Julien Tesson. Albert, an intermediate smart-contract language for the Tezos blockchain, January 2020. arXiv:2001.02630.

[61] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, October 2016. Association for Computing Machinery. doi:10.1145/2993600.2993611.

[62] Giancarlo Bigi, Andrea Bracciali, Giovanni Meacci, and Emilio Tuosto. Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods. In Chiara Bodei, Gianluigi Ferrari, and Corrado Priami, editors, *Programming Languages with Applications to Biology and Security: Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*, Lecture Notes in Computer Science, pages 142–161. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-25527-9_11.

[63] Michael Borkowski, Daniel McDonald, Christoph Ritzer, and Stefan Schulte. Towards Atomic Cross-Chain Token Transfers: State of the Art and Open Questions within TAST. page 10.

[64] Michael Borkowski, Marten Sigwart, Philipp Frauenthaler, Taneli Hukkinen, and Stefan Schulte. Dextt: Deterministic Cross-Blockchain Token Transfers. *IEEE Access*, 7:111030–111042, 2019. doi:10.1109/ACCESS.2019.2934707.

[65] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In *Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part III 24*, pages 222–249. Springer, 2018.

[66] BscScan.com. Pancake Bunny Exploiter. http://bscscan.com/address/0x158c244b62058330f2c328c720b072d8db2c612f.

[67] BscScan.com. Spartan Protocol 2021 Exploit. http://bscscan.com/tx/0xb64ae25b0d836c25d115a9368319902c972a0215b

[68] BscScan.com. Uranium Finance Exploiter. http://bscscan.com/address/0x2b528a28451e9853f51616f3b0f6d82af8bea6ae

[69] Giulio Caldarelli. Wrapping Trust for Interoperability: A Preliminary Study of Wrapped Tokens. *Information*, 13(1):6, January 2022. doi:10.3390/info13010006.

[70] COBRA Research Center. Concert. URL: https://github.com/AU-COBRA/ConCert.

[71] Martán Ceresa and César Sánchez. Multi: A Formal Playground for Multi-Smart Contract Interaction, July 2022. arXiv:2207.06681.

[72] James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. System F in Agda, for Fun and Profit. In Graham Hutton, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 255–297, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-33636-3_10.

[73] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 53(3):1–43, 2020.

[74] Alessio Chiapperini, Marino Miculan, and Marco Peressotti. Computing (optimal) embeddings of directed bigraphs. *Science of Computer Programming*, 221:102842, September 2022. doi:10.1016/j.scico.2022.102842.

[75] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2022.

[76] Usman W. Chohan. Initial Coin Offerings (ICOs): Risks, Regulation, and Accountability. In Stéphane Goutte, Khaled Guesmi, and Samir Saadi, editors, *Cryptofinance and Mechanisms of Exchange: The Making of Virtual Currency*, Contributions to Management Science, pages 165–177. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-30738-7_10.

[77] Michele Ciampi, Muhammad Ishaq, Malik Magdon-Ismail, Rafail Ostrovsky, and Vassilis Zikas. FairMM: A Fast and Frontrunning-Resistant Crypto Market-Maker, 2021.

[78] Thierry Coquand and Christine Paulin. Inductively defined types. In G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, Per Martin-Löf, and Grigori Mints, editors, *COLOG-88*, volume 417, pages 50–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990. doi:10.1007/3-540-52335-9_47.

[79] Simon Cousaert, Jiahua Xu, and Toshiko Matsui. SoK: Yield Aggregators in DeFi. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–14, May 2022. doi:10.1109/ICBC54727.2022.9805523.

[80] Luís Pedro Arrojado da Horta, João Santos Reis, Simão Melo de Sousa, and Mário Pereira. A tool for proving michelson smart contracts in why3. In *2020 IEEE International Conference on Blockchain (Blockchain)*, pages 409–414. IEEE, 2020.

[81] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 910–927, May 2020. doi:10.1109/SP40000.2020.00040.

[82] Monika Di Angelo and Gernot Salzer. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78. IEEE, 2019.

[83] Andres Diaz-Valdivia and Marta Poblet. Governance of ReFi Ecosystem and the Integrity of Voluntary Carbon Markets as a Common Resource, November 2022. doi:10.2139/ssrn.4286167.

[84] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, Vikram Saraph, and Eric Koskinen. Proof-Carrying Smart Contracts. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 325–338, Berlin, Heidelberg, 2019. Springer. doi:10.1007/978-3-662-58820-8_22.

[85] Christian Doczkal, Damien Pous, and Daniel Severin. Graph Theory. URL: https://github.com/coq-community/graph-theory.

[86] Xiaoqun Dong, Rachel Chi Kiu Mok, Durreh Tabassum, Pierre Guigon, Eduardo Ferreira, Chandra Shekhar Sinha, Neeraj Prasad, Joe Madden, Tom Baumann, Jason Libersky, Eamonn McCormick, and Jefferson Cohen. Blockchain and emerging digital technologies for enhancing post-2020 climate markets. https://tinyurl.com/bdz5wczb.

[87] Gregor Dorfleitner, Franziska Muck, and Isabel Scheckenbach. Blockchain applications for climate protection: A global empirical investigation. *Renewable and Sustainable Energy Reviews*, 149:111378, October 2021. doi:10.1016/j.rser.2021.111378.

[88] Michael Egorov. StableSwap-efficient mechanism for Stablecoin liquidity. *Retrieved Feb*, 24:2021, 2019.

[89] Youssef El Faqir, Javier Arroyo, and Samer Hassan. An overview of decentralized autonomous organizations on the blockchain. In *Proceedings of the 16th International Symposium on Open Collaboration*, OpenSym 2020, pages 1–8, New York, NY, USA, August 2020. Association for Computing Machinery. doi:10.1145/3412569.3412579.

[90] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain. In Andrea Bracciali, Jeremy Clark, Federico Pintore, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 170–189, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-43725-1_13.

[91] Shayan Eskandari, Mehdi Salehi, Wanyun Catherine Gu, and Jeremy Clark. SoK: Oracles from the ground truth to market manipulation. In *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, AFT '21, pages 127–141, New York, NY, USA, September 2021. Association for Computing Machinery. doi:10.1145/3479722.3480994.

[92] etherscan.io. Beanstalk Exploit. http://etherscan.io/tx/0xcd314668aaa9bbfebaf1a0bd2b6553d01dd58899c508d4729fa73

[93] etherscan.io. CREAM Finance Exploit. http://etherscan.io/tx/0x0fe2542079644e107cbf13690eb9c2c65963ccb79089ff96b

[94] etherscan.io. Harvest.Finance: Hacker 1. http://etherscan.io/address/0xf224ab004461540778a914ea397c589b677e27bb.

[95] etherscan.io. Nomad Bridge Exploit. http://etherscan.io/tx/0xa5fe9d044e4f3e5aa5bc4c0709333cd2190cba0f4e7f16bcf73

[96] Alex Evans, Guillermo Angeris, and Tarun Chitra. Optimal Fees for Geometric Mean Market Makers. In Matthew Bernhard, Andrea Bracciali, Lewis Gudgeon, Thomas Haines, Ariah Klages-Mundt, Shin'ichiro Matsuo, Daniel Perez, Massimiliano Sala, and Sam Werner, editors, *Financial Cryptography and Data Security. FC 2021 International Workshops*, Lecture Notes in Computer Science, pages 65–79, Berlin, Heidelberg, 2021. Springer. doi:10.1007/978-3-662-63958-0_6.

[97] Harvest Finance. Harvest Flashloan Economic Attack Post-Mortem, October 2020.

[98] Uranium Finance. Uranium Finance Exploit, April 2021.

[99] Pierluigi Freni, Enrico Ferro, and Roberto Moncada. Tokenization and Blockchain Tokens Classification: A morphological framework. In *2020 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, July 2020. doi:10.1109/ISCC50000.2020.9219709.

[100] Pierluigi Freni, Enrico Ferro, and Roberto Moncada. Tokenomics and blockchain tokens: A design-oriented morphological framework. *Blockchain: Research and Applications*, 3(1):100069, March 2022. doi:10.1016/j.bcra.2022.100069.

[101] Robin Fritsch. Concentrated Liquidity in Automated Market Makers. *DeFi@CCS*, 2021. doi:10.1145/3464967.3488590.

[102] Robin Fritsch, Samuel Kaser, and Roger Wattenhofer. The Economics of Automated Market Makers. 2022.

[103] Robin Fritsch and Roger Wattenhofer. A Note on Optimal Fees for Constant Function Market Makers. *DeFi@CCS*, 2021. doi:10.1145/3464967.3488589.

[104] Ikram Garfatta, Kais Klai, Walid Gaaloul, and Mohamed Graiet. A Survey on Formal Verification for Solidity Smart Contracts. In *2021 Australasian Computer Science Week Multiconference*, ACSW '21, pages 1–10, New York, NY, USA, February 2021. Association for Computing Machinery. doi:10.1145/3437378.3437879.

[105] Florian Gronde. Flash Loans and Decentralized Lending Protocols: An In-Depth Analysis. page 75.

[106] Lewis Gudgeon, Daniel Perez, Dominik Harz, Benjamin Livshits, and Arthur Gervais. The Decentralized Financial Crisis. In *2020 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 1–15, June 2020. doi:10.1109/CVCBT50464.2020.00005.

[107] Lioba Heimbach, Eric Schertenleib, and Roger Wattenhofer. Risks and Returns of Uniswap V3 Liquidity Providers. *arXiv preprint arXiv:2205.08904*, 2022. arXiv:2205.08904.

[108] Lioba Heimbach, Ye Wang, and Roger Wattenhofer. Behavior of Liquidity Providers in Decentralized Exchanges. 2021.

[109] Maurice Herlihy. Atomic Cross-Chain Swaps. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 245–254, New York, NY, USA, July 2018. Association for Computing Machinery. doi:10.1145/3212734.3212736.

[110] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, November 2005. doi:10.1145/1108970.1108971.

[111] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, July 2018. doi:10.1109/CSF.2018.00022.

[112] Yoichi Hirai. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, volume 10323, pages 520–535. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-70278-0_33.

[113] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[114] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant a tutorial. *Rapport Technique*, 178, 1997.

[115] Igor Igamberdiev [@FrankResearcher]. BUNNY Tweet 1, May 2021.

[116] Immunefi. Hack Analysis: Cream Finance Oct 2021, November 2022.

[117] Immunefi. Hack Analysis: Nomad Bridge, August 2022, January 2023.

[118] PeckShield Inc. The Spartan Incident: Root Cause Analysis, May 2021.

[119] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1695–1712. IEEE, 2020.

[120] Aljosha Judmayer, Nicholas Stifter, Philipp Schindler, and Edgar Weippl. Estimating (Miner) Extractable Value is Hard, Let's Go Shopping! *Cryptology ePrint Archive*, 2021.

[121] Roman Kozhan and Ganesh Viswanath-Natraj. Decentralized Stablecoins and Collateral Risk, June 2021. doi:10.2139/ssrn.3866975.

[122] Roman Kozhan and Ganesh Viswanath-Natraj. Fundamentals of the MakerDAO Governance Token. In *3rd International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[123] Bhaskar Krishnamachari, Qi Feng, and Eugenio Grippo. Dynamic Curves for Decentralized Autonomous Cryptocurrency Exchanges, January 2021. arXiv:2101.02778, doi:10.48550/arXiv.2101.02778.

[124] Leslie Lamport. Specifying Concurrent Systems with TLA+. *Calculational System Design*, pages 183–247, April 1999.

[125] Sung-Shine Lee, Alexandr Murashkin, Martin Derka, and Jan Gorzny. SoK: Not Quite Water Under the Bridge: Review of Cross-Chain Bridge Hacks. *arXiv preprint arXiv:2210.16209*, 2022. arXiv:2210.16209.

[126] Pierre Letouzey. Extraction in coq: An overview. In *Logic and Theory of Algorithms: 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008 Proceedings 4*, pages 359–369. Springer, 2008.

[127] Ximeng Li, Zhiping Shi, Qianying Zhang, Guohui Wang, Yong Guan, and Ning Han. Towards Verifying Ethereum Smart Contracts at Intermediate Language Level. In Yamine Ait-Ameur and Shengchao Qin, editors, *Formal Methods and Software Engineering*, Lecture Notes in Computer Science, pages 121–137, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-32409-4_8.

[128] Yannis Lilis and Anthony Savidis. A Survey of Metaprogramming Languages. *ACM Comput. Surv.*, 52(6):113:1–113:39, October 2019. doi:10.1145/3354584.

[129] Alexander Lipton, Aetienne Sardon, Fabian Schär, and Christian Schüpbach. 11. Stablecoins, Digital Currency, and the Future of Money. In *Building the New Economy*. April 2020.

[130] Chen Liu and Haoquan Wang. Initial Coin Offerings: What Do We Know and What Are the Success Factors? In Stéphane Goutte, Khaled Guesmi, and Samir Saadi, editors, *Cryptofinance and Mechanisms of Exchange: The Making of Virtual Currency*, Contributions to Management Science, pages 145–164. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-30738-7_9.

[131] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, October 2016. Association for Computing Machinery. doi:10.1145/2976749.2978309.

[132] Léonard Lys, Arthur Micoulet, and Maria Potop-Butucaru. Atomic cross chain swaps via relays and adapters. In *Proceedings of the 3rd Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, CryBlock '20, pages 59–64, New York, NY, USA, September 2020. Association for Computing Machinery. doi:10.1145/3410699.3413799.

[133] Katya Malinova and Andreas Park. Market Design with Blockchain Technology, July 2017. doi:10.2139/ssrn.2785626.

[134] Shaurya Malwa. How Market Manipulation Led to a $100M Exploit on Solana DeFi Exchange Mango. https://www.coindesk.com/markets/2022/10/12/how-market-manipulation-led-to-a-100m-exploit-on-solana-defi-exchange-mango/, October 2022.

[135] Alessio Mansutti, Marino Miculan, and Marco Peressotti. Multi-agent Systems Design and Prototyping with Bigraphical Reactive Systems. In Kostas Magoutis and Peter Pietzuch, editors, *Distributed Applications and Interoperable Systems*, volume 8460, pages 201–208. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. doi:10.1007/978-3-662-43352-2_16.

[136] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers 22*, pages 523–540. Springer, 2018.

[137] Anastasia Mavridou and Aron Laszka. Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust*, Lecture Notes in Computer Science, pages 270–277, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-89722-6_11.

[138] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 446–465, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-32101-7_27.

[139] Patrick McCorry, Chris Buckland, Bennet Yee, and Dawn Song. Sok: Validating bridges as a scaling solution for blockchains. *Cryptology ePrint Archive*, 2021.

[140] Eva Meyer, Isabell M. Welpe, and Philipp G. Sandner. Decentralized Finance—A Systematic Literature Review and Research Directions, 2022. doi:10.2139/ssrn.4016497.

[141] Robin Milner. The Bigraphical Model. https://www.cl.cam.ac.uk/archive/rm135/uam-theme.html.

[142] Robin Milner. Bigraphs and Their Algebra. *Electronic Notes in Theoretical Computer Science*, 209:5–19, April 2008. doi:10.1016/j.entcs.2008.04.002.

[143] Yvonne Murray and David A. Anisi. Survey of Formal Verification Methods for Smart Contracts on Blockchain. In *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–6, June 2019. doi:10.1109/NTMS.2019.8763832.

[144] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, January 1997. Association for Computing Machinery. doi:10.1145/263699.263712.

[145] Zeinab Nehai and François Bobot. Deductive Proof of Ethereum Smart Contracts Using Why3, August 2019. arXiv:1904.11281.

[146] Zeinab Nehaï, François Bobot, Sara Tucci-Piergiovanni, Carole Delporte-Gallet, and Hugues Fauconnier. A TLA+ Formal Proof of a Cross-Chain Swap. In *23rd International Conference on Distributed Computing and Networking*, ICDCN 2022, pages 148–159, New York, NY, USA, January 2022. Association for Computing Machinery. doi:10.1145/3491003.3491006.

[147] Michael Neuder, Rithvik Rao, Daniel J. Moroz, and David C. Parkes. Strategic liquidity provision in uniswap v3. *arXiv preprint arXiv:2106.12033*, 2021. arXiv:2106.12033.

[148] Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Formalising Decentralised Exchanges in Coq. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 290–302, 2023.

[149] Jakob Botsch Nielsen and Bas Spitters. Smart Contract Interactions in Coq. In Emil Sekerinski, Nelma Moreira, José N. Oliveira, Daniel Ratiu, Riccardo Guidotti, Marie Farrell, Matt Luckcuck, Diego Marmsoler, José Campos, Troy Astarte, Laure Gonnord, Antonio Cerone, Luis Couto, Brijesh Dongol, Martin Kutrib, Pedro Monteiro, and David Delmas, editors, *Formal Methods. FM 2019 International Workshops*, Lecture Notes in Computer Science, pages 380–391, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-54994-7_29.

[150] Markus Nissl, Emanuel Sallinger, Stefan Schulte, and Michael Borkowski. Towards Cross-Blockchain Smart Contracts. In *2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pages 85–94, August 2021. doi:10.1109/DAPPS52256.2021.00015.

[151] Eric Nowak. Voluntary Carbon Markets. https://tinyurl.com/k8sbhemf, March 2022.

[152] Russell O'Connor. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pages 107–120, 2017.

[153] Kris Oosthoek. Flash Crash for Cash: Cyber Threats in Decentralized Finance, June 2021. arXiv:2106.10740.

[154] Abraham Othman, David M. Pennock, Daniel M. Reeves, and Tuomas Sandholm. A practical liquidity-sensitive automated market maker. *ACM Transactions on Economics and Computation (TEAC)*, 1(3):1–25, 2013.

[155] pancakebunny.finance [@PancakeBunnyFin]. BUNNY Tweet 2, May 2021.

[156] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. A formal verification tool for Ethereum VM bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 912–915, Lake Buena Vista FL USA, October 2018. ACM. doi:10.1145/3236024.3264591.

[157] Christine Paulin-Mohring. *Introduction to the Calculus of Inductive Constructions*, volume 55. College Publications, January 2015.

[158] Daniel Perez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care? *arXiv preprint arXiv:1902.06710*, pages 1–15, 2019. arXiv:1902.06710.

[159] Daniel Perez, Sam M. Werner, Jiahua Xu, and Benjamin Livshits. Liquidations: DeFi on a Knife-Edge. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 457–476, Berlin, Heidelberg, 2021. Springer. doi:10.1007/978-3-662-64331-0_24.

[160] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, Lecture Notes in Computer Science, pages 209–228, New York, NY, 1990. Springer-Verlag. doi:10.1007/BFb0040259.

[161] Siraphob Phipathananunth. Using Mutations to Analyze Formal Specifications. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2022, pages 81–83, New York, NY, USA, December 2022. Association for Computing Machinery. doi:10.1145/3563768.3563960.

[162] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 2010.

[163] Babu Pillai, Kamanashis Biswas, and Vallipuram Muthukkumarasamy. Cross-chain interoperability among blockchain-based systems using transactions. *The Knowledge Engineering Review*, 35, 2020.

[164] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph Liu, and Robin Doss. Security analysis methods on ethereum smart contract vulnerabilities: A survey. *arXiv preprint arXiv:1908.08605*, 2019. arXiv:1908.08605.

[165] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 3–32, Berlin, Heidelberg, 2021. Springer. doi:10.1007/978-3-662-64322-8_1.

[166] Joseph J. Rotman. *An Introduction to Homological Algebra*. Springer, New York, NY, 2009. doi:10.1007/b98977.

[167] John Rushby. Theorem Proving for Verification. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, *Modeling and Verification of Parallel Processes: 4th Summer School, MOVEP 2000 Nantes, France, June 19–23, 2000 Revised Tutorial Lectures*, Lecture Notes in Computer Science, pages 39–57. Springer, Berlin, Heidelberg, 2001. doi:10.1007/3-540-45510-8_2.

[168] samczsun is occasionally shitposting [@samczsun]. Nomad Tweet Thread, August 2022.

[169] Fabian Schär. Decentralized Finance: On Blockchain- and Smart Contract-Based Financial Markets, April 2021. doi:10.20955/r.103.153-74.

[170] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Temporal Properties of Smart Contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Lecture Notes in Computer Science, pages 323–338, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-03427-6_25.

[171] Michele Sevegnani and Muffy Calder. BigraphER: Rewriting and analysis engine for bigraphs. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, pages 494–501. Springer, 2016.

[172] Narges Shadab, Farzin Houshmand, and Mohsen Lesani. Cross-chain Transactions. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, May 2020. doi:10.1109/ICBC48266.2020.9169477.

[173] Tim Sheard. Accomplishments and research challenges in meta-programming. In *SAIG*, volume 2196, pages 2–44. Springer, 2001.

[174] Marten Sigwart, Philipp Frauenthaler, Christof Spanring, Michael Sober, and Stefan Schulte. Decentralized Cross-Blockchain Asset Transfers. In *2021 Third International Conference on Blockchain Computing and Applications (BCCA)*, pages 34–41, November 2021. doi:10.1109/BCCA53669.2021.9657007.

[175] Amritraj Singh, Reza M. Parizi, Qi Zhang, Kim-Kwang Raymond Choo, and Ali Dehghantanha. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Computers & Security*, 88:101654, January 2020. doi:10.1016/j.cose.2019.101654.

[176] Adam Sipthorpe, Sabine Brink, Tyler Van Leeuwen, and Iain Staffell. Blockchain solutions for carbon markets are nearing maturity. *One Earth*, 5(7):779–791, July 2022. doi:10.1016/j.oneear.2022.06.004.

[177] solscan.io. Mango Markets Exploiter. https://solscan.io/account/CQvKSNnYtPTZfQRQ5jkHq8q2swJyRsdQLcFcj3Em

[178] Derek Sorensen. Structured pools for tokenized carbon credits. *IEEE ICBC CryptoEx*, 2023.

[179] Derek Sorensen. Tokenized carbon credits. *Preprint*, 2023. URL: https://derekhsorensen.com/docs/sorensen-tokenized-carbon-credits.pdf.

[180] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *J Autom Reasoning*, 64(5):947–999, June 2020. doi:10.1007/s10817-019-09540-0.

[181] Tianyu Sun and Wensheng Yu. A Formal Verification Framework for Security Issues of Blockchain Smart Contracts. *Electronics*, 9(2):255, February 2020. doi:10.3390/electronics9020255.

[182] Huang Teng, Wayneyuan Tian, Haocheng Wang, and Zhiyuan Yang. Applications of the Decentralized Finance (DeFi) on the Ethereum. In *2022 IEEE Asia-Pacific Conference on Image Processing, Electronics and Computers (IPEC)*, pages 573–578. IEEE, 2022.

[183] Palina Tolmach, Yi Li, Shang-Wei Lin, and Yang Liu. Formal Analysis of Composable DeFi Protocols. In Matthew Bernhard, Andrea Bracciali, Lewis Gudgeon, Thomas Haines, Ariah Klages-Mundt, Shin'ichiro Matsuo, Daniel Perez, Massimiliano Sala, and Sam Werner, editors, *Financial Cryptography and Data Security. FC 2021 International Workshops*, Lecture Notes in Computer Science, pages 149–161, Berlin, Heidelberg, 2021. Springer. doi:10.1007/978-3-662-63958-0_13.

[184] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.*, 54(7):148:1–148:38, July 2021. doi:10.1145/3464421.

[185] UniMath. Coq of ocaml. URL: https://github.com/formal-land/coq-of-ocaml.

[186] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath — a computer-checked library of univalent mathematics. available at http://unimath.org. URL: https://github.com/UniMath/UniMath.

[187] Shermin Voshmgir and Michael Zargham. Foundations of Cryptoeconomic Systems. *Foundations of Cryptoeconomic Systems*, 2020.

[188] Dabao Wang, Siwei Wu, Ziling Lin, Lei Wu, Xingliang Yuan, Yajin Zhou, Haoyu Wang, and Kui Ren. Towards A First Step to Understand Flash Loan and Its Applications in DeFi Ecosystem. In

Proceedings of the Ninth International Workshop on Security in Blockchain and Cloud Computing, pages 23–28, May 2021. arXiv:2010.12252, doi:10.1145/3457977.3460301.

[189] Gang Wang. SoK: Exploring Blockchains Interoperability, 2021.

[190] Shuai Wang, Wenwen Ding, Juanjuan Li, Yong Yuan, Liwei Ouyang, and Fei-Yue Wang. Decentralized Autonomous Organizations: Concept, Model, and Applications. *IEEE Transactions on Computational Social Systems*, 6(5):870–878, October 2019. doi:10.1109/TCSS.2019.2938190.

[191] Yongge Wang. Automated Market Makers for Decentralized Finance (DeFi), September 2020. arXiv:2009.01676, doi:10.48550/arXiv.2009.01676.

[192] León Welicki, Juan Cueva Lovelle, and Luis Aguilar. Meta-Specification and Cataloging of Software Patterns with Domain Specific Languages and Adaptive Object Models. pages 359–392, January 2006.

[193] León Welicki, O. San Juan, and J. M. Cueva Lovelle. A model for meta-specification and cataloging of software patterns. *Proceedings of the 12th PLoP*, 2005.

[194] Sam M. Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William J. Knottenbelt. SoK: Decentralized Finance (DeFi), September 2021. arXiv:2101.08778, doi:10.48550/arXiv.2101.08778.

[195] U. E. Wolter, A. R. Martini, and E. H. Häusler. Indexed and fibered structures for partial and total correctness assertions. *Mathematical Structures in Computer Science*, pages 1–31, September 2022. doi:10.1017/S0960129522000275.

[196] Siwei Wu, Dabao Wang, Jianting He, Yajin Zhou, Lei Wu, Xingliang Yuan, Qinming He, and Kui Ren. DeFiRanger: Detecting Price Manipulation Attacks on DeFi Applications, April 2021. arXiv:2104.15068, doi:10.48550/arXiv.2104.15068.

[197] 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, February 2023. doi:10.1145/3570639.

[198] Yingjie Xue and Maurice Herlihy. Cross-Chain State Machine Replication, June 2022. arXiv:2206.07042, doi:10.48550/arXiv.2206.07042.

[199] Zheng Yang and Hang Lei. Fether: An extensible definitional interpreter for smart-contract verifications in coq. *IEEE Access*, 7:37770–37791, 2019.

[200] Zheng Yang and Hang Lei. Lolisa: Formal Syntax and Semantics for a Subset of the Solidity Programming Language in Mathematical Tool Coq. *Mathematical Problems in Engineering*, 2020:e6191537, December 2020. doi:10.1155/2020/6191537.

[201] Zheng Yang, Hang Lei, and Weizhong Qian. A Hybrid Formal Verification System in Coq for Ensuring the Reliability and Security of Ethereum-Based Service Smart Contracts. *IEEE Access*, 8:21411–21436, 2020. doi:10.1109/ACCESS.2020.2969437.

[202] Jimmy Yin and Mac Ren. On Liquidity Mining for Uniswap v3. *arXiv preprint arXiv:2108.05800*, 2021. arXiv:2108.05800.

[203] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J. Knottenbelt. Sok: Communication across distributed ledgers. In *International Conference on Financial Cryptography and Data Security*, pages 3–36. Springer, 2021.

[204] Xiyue Zhang, Yi Li, and Meng Sun. Towards a Formally Verified EVM in Production Environment. *Coordination Models and Languages*, 12134:341–349, May 2020. doi:10.1007/978-3-030-50029-0_21.

[205] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, April 2020. doi:10.1016/j.future.2019.12.019.

[206] Liyi Zhou, Kaihua Qin, and Arthur Gervais. A2mm: Mitigating frontrunning, transaction reordering and consensus instability in decentralized exchanges. *arXiv preprint arXiv:2106.07371*, 2021. arXiv:2106.07371.

[207] Jian Zhu, Kai Hu, Mamoun Filali, Jean-Paul Bodeveix, and Jean-Pierre Talpin. Formal Verification of Solidity contracts in Event-B, May 2020. arXiv:2005.01261, doi:10.48550/arXiv.2005.01261.

[208] Jean-Yves Zie, Jean-Christophe Deneuville, Jérémy Briffaut, and Benjamin Nguyen. Extending Atomic Cross-Chain Swaps. In Cristina Pérez-Solà, Guillermo Navarro-Arribas, Alex Biryukov, and Joaquin Garcia-Alfaro, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, Lecture Notes in Computer Science, pages 219–229, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-31500-9_14.

[209] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering*, 47(10):2084–2106, October 2021. doi:10.1109/TSE.2019.2942301.