Meta-Theoretic Reasoning in Formal Verification of Financial Smart Contracts

Derek Sorensen

March 7, 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 [162]. 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 [142, 171].

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 [42, 60, 68, 88, 120, 133, 137, 139, 145, 153]. 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.

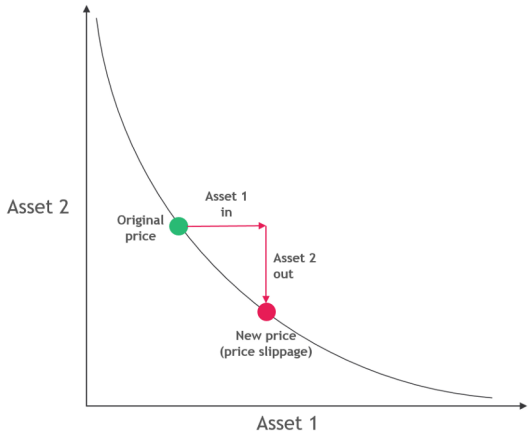

Figure 1.1: A trade along the indifference curve $xy = k$.[1]

Dexter2 Verification

Dexter2 has been formally verified by three different groups using three different formal verification tools [5, 16, 124]. They all based their work on the same (informal) specification [4].

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. Of course, because the contract is financial in nature, it must have *some* expected economic behavior.

Consider what Vitalik Buterin, co-founder of Ethereum, wrote in March 2018 about AMMs on the online forum Ethereum Research [14], which informed the design and implementation of Uniswap, likely the first most popular AMM [25]. 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:

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

[1] Image from [6].

Buterin argues that an AMM that trades along (1.1) features efficient price discovery. He also argues that it 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 [4], 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 [4] or its formal counterparts [5, 16, 124] 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 [80, 86, 87]; on how liquidity providers react to market changes [92]; and how all of that relates to the *curvature* of $xy = k$ [24, 159]. 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 [67].

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 *meta-specification*, 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 [136], 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" [163].

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 [76].

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 [138]. Flash loans can introduce unintuitive contract behavior which can be difficult to reason about and which have been extensively studied [20, 86, 89, 90, 138, 157].

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 [138].

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 [138]. 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 [100]. 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 [55] 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 [97, 130]. 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 [8, 54].

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 [138], 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 [147]. 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 [46]. Notably, CoinDesk reported that the attacker "did everything within the parameters of how the platform was designed" [113]. 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." [35].

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.* [40, 89, 92, 138, 152], 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. As we have noted, contracts are immutable once deployed, so upgrades are notoriously difficult and error-prone [37]. 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 [105], 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 [140]. 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 [79, 99].

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 [82]. NowSwap, a nearly identical application, had the same error and incurred a loss of $1 million [45].

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 [127].

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 [77]. The attack was extremely complex, involving 68 assets and over 9 ETH in gas, roughly 36k USD at the time [77]. 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 [98]. 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 [152]. 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 [98].

Systems of contracts are extremely difficult to reason about, formally or informally [125]. There have been recent efforts to formally and rigorously study systems of interacting DeFi contracts [152], 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 [125]. 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 [113]	115M	DEX	[147]	2022
Beanstalk	Contract Design [2]	77M	Stablecoin	[76]	2022
Pancake Bunny	Contract Design [8]	45M	Yield aggregator	[54]	2021
Spartan Protocol	Contract Design [100]	30M	AMM	[55]	2021
Nomad	Unsafe Upgrade [99]	190M	Cross-chain bridge	[79]	2022
Uranium	Unsafe Upgrade [82]	50M	AMM	[56]	2021
Cream Finance	Complex System [98]	130M	DeFi protocol	[77]	2021
Harvest Finance	Complex System [81]	34M	Yield aggregator	[78]	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 [46]. In 2022, DeFi applications were the most attacked type of blockchain-based application, making up about two-thirds of all attacks [3, 46], and attacks which exploit improper business logic or function design are in the top three causes of loss [46, 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 [34, 105, 75, 74], decentralized finance [162, 164], blockchain interoperability and bridging [158, 169, 118], or other areas of DeFi [38, 65]. 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

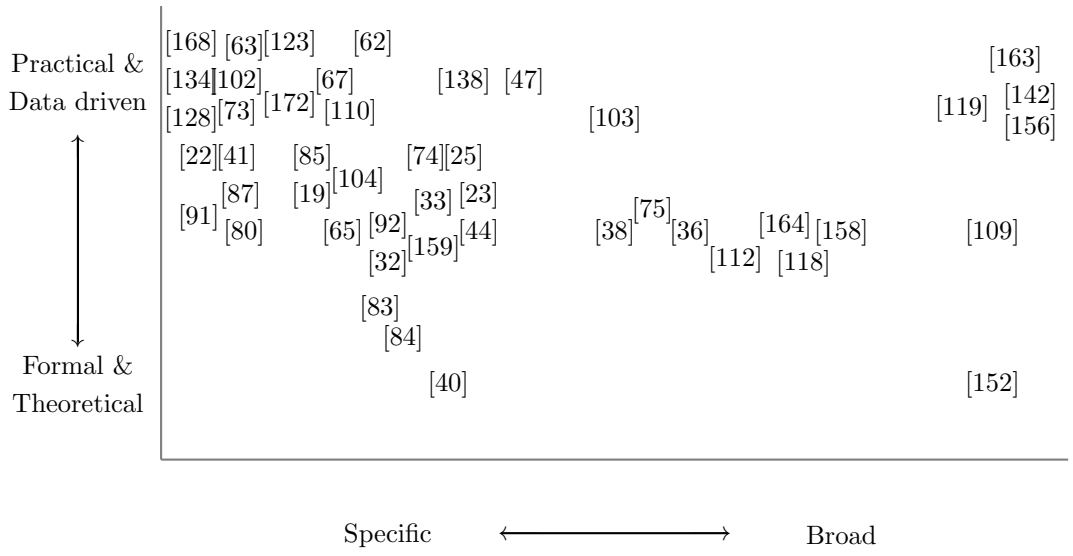

Figure 2.2: A graphical representation of studies of smart contracts and their behavior.

to practical and data driven.

Importantly, as we noted before these studies have not yet been formally incorporated into formal verification of smart contracts [163], 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 [172] and FairAMM [63], two AMMs which are designed to resist front-running attacks, the vulnerability of AMMs originally pointed out by Buterin [14]. 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 [172, §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 [172, §III.C]. Using this abstract and formal model, the authors prove two theorems [172, §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 [63] 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) [22], their work on analyzing Uniswap markets [25], and Bartoletti *et al.*'s work on developing a theory of decentralized finance [40]. 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 [172] and FairAMM [63]. Bartoletti *et al.* do so within the context of a formal model [40].

Some examples of these properties include:

- *Demand sensitivity*, or the property that the price of an asset increase with demand [40, §4.2] [25, §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 [40, §4.2] [25, §2.3]
- That trading cost be positive, meaning that a swap of one token for another, and then back again has a positive cost [25, §2], [22, §3 §4.1]
- And that market prices converge through arbitrage [40, §4.3] [25, §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 [172] and FairAMM [63] 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 [125]. 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 [38], and Tolmach *et al.*'s process-algebraic approach to reasoning about composable DeFi protocols

[152].

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 meta-specification 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 [42, 60, 68, 88, 120, 133, 137, 139, 145, 153]. 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, [153, §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 [139]. Finally, we use proof assistants because they are recognized in the literature to be of the highest quality for formal verification [120].

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 [29] 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 [21, 95]; in Coq [167]; in the K Framework [94, 131]; in Why3 [170, 121]; and in Z3 [111]. 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 [51]; K Michelson [17], an embedding of Michelson into the K Framework; and WhyISon, which transpiles Michelson into WhyML, the programming and specification language of the Why3 framework [66]. There are examples on other chains as well, including an embedding of the low-level, Bitcoin-based contract language Simplicity in Coq [126].

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 [107]. On the other hand, higher-level languages tend to be more human-readable, and can be more straightforward to reason about [107], especially if they are statically typed functional programming languages [29]. 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 [107].

We have many examples of intermediate- and high-level languages which have been embedded into proof assistants. For Ethereum smart contracts, we have Lolisa [166] and FEther [165], embeddings of Solidity into Coq; as well as embeddings of Solidity into Event-B [173]; into Isabelle [107]; into the K Framework [101]; and into F* [53]. We also have custom implementations of finite state machines used by FSolidM [115, 116] and VeriSolid [117] that verify Solidity code. For Tezos, we have an embedding of Albert [52], 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 [29, 30]. On other chains we have Plutus in Agda [59], and verification for BNB in Coq [151].

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 [143]. Archetype is a Tezos-based DSL which targets business logic

and uses Why3 [50]. Multi is a framework, written in Coq, which targets reasoning about smart contract interactions [58]. 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 [122].

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 [29] 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 [26, 28, 150], 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 [108]. It has a long history, starting with Lisp macros, and is now arguably mainstream with a variety of applications [49, 108, 144].

Metaprogramming is an important topic in software verification, *e.g.* [48, 49], primarily as a method to specify and verify meta-programs—especially those which generate code. MetaCoq [150], 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 [27]. 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 *meta-specification*, or a specification of a specification. The concept of a meta-specification exists in the literature, *e.g.* in [160, 161], which uses meta-specifications 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 meta-specification is different. While previous work uses meta-specifications to identify patterns in boilerplate specifications, we use meta-specifications 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 meta-specifications 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 [136]. It has deep connections to OCaml with the Coq-of-OCaml verification tool [106, 154]. Additionally, there has been extensive work on formalizing pure mathematics, including in the UniMath library and others [43, 69, 155]. For the interested reader, we cite some tutorials and introductions to Coq as a tool for program verification [61, 96, 136].

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 [132], 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 [64, 135].

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 :=
```

```
| 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 [96].

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 [57].

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.

```
1 Record Contract
2      (Setup Msg State Error : Type)
3      `{Serializable Setup}
4      `{Serializable Msg}
5      `{Serializable State}
6      `{Serializable Error} :=
7   build_contract {
8
9     init :
10      Chain ->
11      ContractCallContext ->
12      Setup ->
13      result State Error;
14
15    receive :
16      Chain ->
17      ContractCallContext ->
18      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.

```
Inductive WeakContract :=
  | build_weak_contract
    (init :
      Chain ->
      ContractCallContext ->
      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.

```
1  Class ChainBase :=
2    build_chain_base {
3      Address : Type;
4      address_eqb : Address -> Address -> bool;
5      address_eqb_spec :
6        forall (a b : Address), Bool.reflect (a = b) (address_eqb a b);
7      address_eqdec :> stdpp.base.EqDecision Address;
8      address_countable :> countable.Countable Address;
9      address_serializable :> Serializable Address;
10     address_is_contract : Address -> bool;
11   }.
```

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.

```
1  Record Chain :=
2    build_chain {
3      chain_height : nat;
4      current_slot : nat;
5      finalized_height : nat;
6    }.
```

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)
```

```
6           (amount : Amount),
7       (* some omitted checks *)
8       ActionEvaluation prev_env act new_env new_acts
9   | eval_deploy :
10     forall (origin from_addr to_addr : Address)
11          (amount : Amount)
12          (wc : WeakContract)
13          (setup : SerializedValue)
14          (state : SerializedValue),
15     (* some omitted checks *)
16     ActionEvaluation prev_env act new_env new_acts
17  | eval_call :
18     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
```

```
| step_action :
   forall (act : Action)
        (acts : list Action)
        (new_acts : list Action),
     ActionEvaluation prev_bstate act next_bstate new_acts ->
     (* some omitted checks *)
     ChainStep prev_bstate next_bstate
| step_action_invalid :
   forall (act : Action)
        (acts : list Action),
     (* some omitted checks *)
     ChainStep prev_bstate next_bstate
| step_permute :
     EnvironmentEquiv next_bstate prev_bstate ->
     Permutation (chain_state_queue prev_bstate) (chain_state_queue next_bstate) ->
     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`.

```
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.

```
Record State :=
  build_state {
     stor : nat
  }.

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
       end.

Definition counter_contract : Contract Setup Msg State Error :=
  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:

```
Theorem counter_correct : forall bstate caddr (trace : ChainTrace empty_state bstate),
  env_contracts caddr = Some (counter_contract : WeakContract) ->
  exists cstate inc_calls,
    contract_state bstate caddr = Some cstate /\
    incoming_calls entrypoint trace caddr = Some inc_calls ->
    (let sum_incr :=
       sumN get_incr_qty inc_calls in
    let sum_decr :=
       sumN get_decr_qty inc_calls in
    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 statement about a contract into seven subgoals, each of which relates to the various ways a blockchain can make progress, as defined in the blockchain semantics.

Metaspecifications

TODO Ties it back to discussion before and introduces a metaspecification.

3.2.1 Contributions

In Chapter 4

- ...

In Chapter 5

- ...

In Chapter 6

- ...

Chapter 4

Specifications and Metaspecifications

Our goal in this chapter is to formally develop the notion of correctness on the level of a contract specification. We do so by reasoning about the properties of a contract specification by way of a metaspecification, which is a specification of a specification. By using a metaspecification, we can incorporate findings from previous research in contract economic behavior that we saw int §2.2.

We will develop the theory of abstract specifications and metaspecifications around a novel financial smart contract which we call a *structured pool* [148], which facilitates pooling and trading tokenized carbon credits, a blockchain-based financial asset growing in prominence and utility [149]. We will give a contract specification, and then reason about that specification's economic properties first informally, and then formally.

The chapter is organized as follows: In §4.1, we motivate and specify the structured pools contract, deriving and proving by hand certain desirable properties of the contract specification from the relevant literature in §4.1.2. In §4.2, we formalize the specification and its associated properties in ConCert; we formally introduce the notion of an abstract specification in §4.2.1 and that of a metaspecification in §4.2.2.

4.1 Structured Pools

The blockchain is a natural technological platform on which to tokenize, trade, and retire carbon credits, as recognized by the UN [11], the World Bank [70], the World Economic Forum [93], and others [141, 72, 146, 71, 114, 129]. It offers several advantages to legacy systems, including transparency, traceability, and censorship resistance, which lend themselves to robust accounting practices and which can help prevent double counting carbon credits [70, 72].

However, tokenized carbon credits are generally tokenized as non-fungible tokens (NFTs), which are not compatible with trading on automated market makers (AMMs) [149]. As there is demand for trading tokenized carbon credits on automated market makers, some have deployed ways to pool tokenized carbon credits together [10]. These pools allow an individual to deposit a (non-fungible) tokenized carbon credit, called a *constituent token*, into the pool in exchange for a (fungible) token representing the pool, called the *pool token*. As they stand, these pools are limited by the fact that all pools value constitutent tokens equally. This limits the size of pools, which has numerous negative effects, including volatile pricing on AMMs [149].

Structured pools are designed to remove this limitation on pool size, which mitigates the associated negative effects, by pooling tokens together along with a mechanism that values constituent tokens relative to each other and facilitates trades between them [148].

4.1.1 The Contract Specification

We will now give a specification of the structured pools contract. A specification of a smart contract includes a description of its storage, its entrypoints, and each of its entrypoint functions. Our specification here follows the style of previous work on novel AMMs, in particular A^2MM [172] and Fair AMM [63] which were designed to mitigate front-running attacks. That is, we first define required parameters of the storage and entrypoint types, and then give functional specifications of each entrypoint function. In the next section, we reason about the specification to justify that it actually leads to the desired economic behavior.

Storage

As the structured pools contract pools tokens and facilitates trades between them, 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`,
- a nonzero exchange rate r_x for each token t_x, assumed here to be a rational number, 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 pools contract carries of each constituent token t_x in the family T,
- the address of the pool token contract, for which the structured pools contract has minting and burning privileges,
- and k, the total number of pool tokens in circulation.

We will assume that all of this information is contained in contract storage and thus accessible to any entrypoint function.

Entrypoints

The structured pools contract features at least three entrypoints: `DEPOSIT`, for pooling tokens, which deposits liquidity for trading; `WITHDRAW`, 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.

Deposits

The `DEPOSIT` entrypoint accepts the token data of some t_x from the token family and a quantity q of tokens in t_x to be deposited. 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 DEPOSIT entrypoint function *)
2  fn DEPOSIT 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 `DEPOSIT` entrypoint function.

Withdrawals

The `WITHDRAW` 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 WITHDRAW entrypoint function *)
fn WITHDRAW 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 `WITHDRAW` entrypoint function.

Trades

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 in. The pool contract checks that both t_x and t_y are in the token family. Using the exchange rates, the number of outstanding pool tokens, and its balance of the token t_x, 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.

```
(* 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 ;
```

```
16          transfer (delta_x) of (t_x) from (sender) to (self) ;
17          transfer (delta_y) of (t_y) from (self) to (sender) ;
18       </atomic>
19    else
20       fail ;
```

Listing 4.3: the TRADE entrypoint function.

It is not obvious from the CALCULATE_TRADE and UPDATE_RATE pseudocode, but these functions are designed to imitate how a standard AMM prices and executes trades [148]. This will become more clear in the next section of this chapter as we prove that structured pools behave similarly to standard AMMs.

The equations from Listing 4.3, written with formulae, are as follows: For exchange rates r_x and r_y, the contract's balance x of the token t_x, the total number k of pool tokens in circulation, and and a trade of Δ_x, the quantity to be traded out Δ_y is given by

$$\Delta_y := \ell r_x - \frac{k}{\ell r_y + \Delta_x} \tag{4.1}$$

where

$$\ell := \sqrt{\frac{k}{r_x r_y}}. \tag{4.2}$$

The exchange rate r_x is updated to r'_x, calculated as:

$$r'_x := \frac{r_x x + r_y \Delta_y}{x + \Delta_x}. \tag{4.3}$$

We will use these equations and give them greater context in the following section when we reason about the economic properties of structured pools.

4.1.2 Economic Properties of the Specification

Since the structured pools contract is designed to imitate AMMs, we draw on work by Angeris *et al.* [22, 25] and Bartoletti *et al.* [39, 40] on AMMs and DeFi, which studies what might be considered "good" market behaviors of AMMs. They use bespoke, formal models of blockchain execution semantics to reason about game-theoretic properties of smart contracts. The properties we list here emerge from each of these models as indicative of desirable market behavior from game-theoretic and economic perspectives.

These are as follows:

1. *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 [40, §4.2].

2. *Nonpathological prices*—As relative prices shift over time, 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 AMMs, 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 [25, §2].

3. *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. In particular, 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 [25, §2], that trading from t_x to t_y, and back to t_x not be profitable in [22, §3, §4.1].

4. *Functional non-depletion*—No trade can empty the entire pool of tokens. This mimics property of *non-depletion* [40, §4.2] which AMMs generally exhibit, which is that no trade can deplete the liquidity held by the AMM. This property is important so that prices between AMMs equalize through arbitrage [25, 40]. We weaken the property of non-depletion slightly so that some constituent tokens can deplete through trading, but as we will show, we still maintain desirable properties related to arbitrage.

5. *Arbitrage sensitivity*—If an external, demand-sensitive market prices a constituent token differently from the structured pool, through arbitrage the prices of the external market and the structured pool will eventually equalize. 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 [40, §4.3] and [25].

6. *Pooled Consistency*—Pool tokens, which are backed by constituent tokens, are never under- or over-collateralized. This is for definitional integrity of the pool token: the pool tokens are fully backed by the pool, and the pool is fully represented by the pool token. Mathematically, this translates to 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. This 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* [40, §3].

Proof of Economic Properties

The specification of the previous section, §4.1.1, is sufficiently comprehensive so that a contract which satisfies it also satisfies each of the property listed above. In the remainder of this section, we prove this mathematically in six individual results.

Before doing so, note that because these listed properties are true of any contract conforming to the specification of §4.1.1, these are properties of the specification just as much as they are of the contract satisfying the specification. Indeed, the proofs we will give are proofs that the *specification* is correct—it is correct because any contract conforming to it also exhibits the desired economic behaviors, which is the essential purpose of the specification in the first place. Furthermore, if we were to implement a structured pools contract, we would only have to prove our contract correct with regards to the specification, and not these economic properties, since we know that the specification implies these economic behaviors. Because these proofs justify the correctness of the specification, and are proofs *about* the specification, in the coming section the following properties will constitute the *metaspecification*, rather than the contract specification.

Property 1 (Demand Sensitivity)**.** Let t_x and t_y be tokens in our token family with nonzero pooled liquidity and exchange rates $r_x, r_y > 0$. In a trade t_x to t_y, for all $t_z \neq t_x$, r_x decereases relative to r_z and r_z stays constant.

Proof. We need to show that for all r_x, $r'_x < r_x$. That r_z stays constant for all $t_z \neq t_x$ can be seen in Listing 4.3, since of the exchange rates only r_x is updated during a call to `TRADE`. Thus we need to show

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

By definition,

$$r_x = \frac{r_x(x + \Delta_x)}{x + \Delta_x},$$

which by substitution gives us the proposition

$$\frac{r_x x + r_y \Delta_y}{x + \Delta_x} < \frac{r_x(x + \Delta_x)}{x + \Delta_x}.$$

By eliminating the denominator, which is nonzero by assumption, this simplifies to

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

which is true iff

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

or

$$\Delta_y < \frac{r_x}{r_y} \Delta_x. \tag{4.4}$$

The proof of (4.4) comes from the fact that the equations which price trades in Listing 4.3 are designed to imitate trading along the indifference curve $xy = k$, where ℓ from (4.2) is calculated so that

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

and Δ_y from (4.1) is calculated so that

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

The reason for this setup is that, if trades are calculated this way, then the so-called *quoted price* of the trade, or an estimate price given by an AMM before a trade, can be calculated by the exchange rates r_x and r_y, and is in fact $\frac{r_x}{r_y}$ as we see in (4.4).

However, it is a well-known fact that the quoted price is always better than the actual price, in that the so-called *swap price*, or the price that the AMM gives when it actually calculates Δ_y, always yields fewer output tokens than the quoted price estimates. The difference between quoted price and the swap price is called the *price slippage*, and is a well-documented phenomenon [159, 164].

For us, this means that the output of a trade, Δ_y, is always less than the quoted trade, which is $\frac{r_x}{r_y}\Delta_x$, and this proves (4.4), giving us our result. □

Property 2 (Nonpathological prices)**.** If there is a state of the contract such that $r_x > 0$, then $r_x > 0$ always holds.

Proof. Consider (4.3), the formula that updates exchange rates. Because of how Δ_y is calcluated, we know that

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

rendering the numerator of (4.3) always positive. Since x and Δ_x are both positive, the denominator of (4.3) is also positive. Since calling `TRADE` is the only specified function that changes r_x, r_x remains positive so long as there are no other mechanisms in the smart contract that update it. □

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

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:

$$r_y\Delta_y \leq r_x'\Delta_x. \tag{4.5}$$

Since

$$r'_x = \frac{r_x x + r_y \Delta_y}{x + \Delta_x}, \tag{4.3}$$

(4.5) 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 (4.4), we can see that our inequality holds for all $x \geq 0$, as desired.

Now let us consider a trading loop from t_x to t_y, and back to t_x. We have our result if we can show

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

is satisfied, because $\frac{r_y}{r'_x} \Delta_y$ (the quoted price of the trade) is an upper bound on the quantity that Δ_y can be traded for. This, of course, is given by (4.5) and the fact that $r'_x > 0$ from Property 2.

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

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

is satisfied. But we have from (4.5) that

$$r_z \Delta_z \leq r'_y \Delta_y \leq r_y \Delta_y \leq r'_x \Delta_x,$$

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

Property 4 (Functional non-depletion)**.** If there is liquidity held by the structured pool, calling `TRADE` cannot empty the pool entirely.

Proof. First, note that a trade can deplete at most one constituent token at a time, because trades are only for one single input token, and one single output token. Thus we only need to prove this for the case that we have a nonzero x for one token t_x with exchange rate $r_x > 0$. If we execute a trade, we deposit tokens in t_z and withdraw in t_x. First, note that if $t_z = t_x$, then we have our result from (4.4). Thus let $t_x \neq t_z$. If we don't deplete t_x with this trade, we don't have a problem. If we do, we know that we have tokens in t_z, and our pool is not depleted. Furthermore, the trade deposited more tokens than it took out because of how trades are priced, as seen in (4.4). More concretely, this can be seen if we trade back to t_z from t_x sufficient to deplete the t_z, which by Property 3 gives us *more* t_x in the pool than we started with, and so the pool actually stays constant or *grows*, rather than shrinks, with trading. □

Property 5 (Arbitrage sensitivity)**.** Let t_x be a token in our token 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 single 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. First, we make a note that the price of t_x in the structured pool is its price in pool tokens, which presumably are valued on some external AMM (the whole point of pooling is so that we can trade the pool token with deep liquidity on an external AMM). Thus the price of t_x in the structured pool is its price in terms of pool tokens, calculated at the exchange rate of r_x.

We need to consider two cases for this: that where an external market prices a constituent token higher than the structured pool, and one that prices it lower. If the structured pool prices a constituent token t_x higher than an external market, 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. It is straightforward to show that it does so with no positive lower bound (so long as there is sufficient liquidity to execute the trade). So assuming sufficient external liquidity, we have our result.

In the other case, arbitrageurs are incentivized to buy from the structured pool and sell elsewhere. If there is sufficient liquidity, and the market prices an increase in pool token value due to trading, the price of t_x will eventually rise sufficiently to close the gap in prices. If there isn't sufficient liquidity, then the pool will deplete in t_x, removing the opportunity of arbitrage. However, over time the value of t_x in the pool only increases over time relative to other pooled tokens, assuming that there continue to be trades involving other tokens in the structured pool. With enough trades it is thus possible in principle for the gap in prices to close, even making it profitable to eventually trade tokens back into the pool. □

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

$$\sum_x r_x t_x = \#\text{outstanding pool tokens.} \tag{4.6}$$

Proof. As a base case, if we have no pooled liquidity, then (4.6) holds trivially. Then if we call `DEPOSIT`, (4.6) holds by definition. The same is true if we call `WITHDRAW`. Finally, if (4.6) holds and we call `TRADE` from tokens t_x to t_y, then there is an excess number of tokens in t_x, violating (4.6). This excess is quantified in (4.3) and remedied by adjusting r_x to r'_x. □

4.2 The Formalized Specification and Metaspecification

With an example specification in hand, let us consider how we might correctly and rigorously specify this smart contract in a formal setting. Were we to formally verify a smart contract to be correct with regards to this specification, the procedure commonly taken would be to formalize the specification at the time of proving the implementation correct, usually after the implementation is already completed. Because we have no formal proof that the specification of §4.1.1 implies the economic properties of §4.1.2, we would also need to formalize and verify the economic properties if we wanted to be sure they held. We argue that this method of formalization and verification is not fully rigorous, and unnecessarily complicated for those verifying the contract.

Firstly, in formalizing a specification ad hoc for formal verification—instead of when the specification is formulated—we do not have a precise and rigorous notion of what the specification is, or ought to be. Informal specifications, written in prose, inevitably lack in precision compared to their formal counterparts. Thus the translation process from informal to formal inevitably requires choices to be made, which may or may not accurately represent the intent of the informal specification. This compromises the meaning and utility of verifying a contract correct with respect to it.

To illustrate, consider the notion of "safety" formalized about Dexter2 in both ConCert [124] and K Framework [15]. In ConCert, safety is expressed in a theorem that says that the balance of LP tokens held by the main contract can always be calculated correctly by looking at certain kinds of transactions from the blockchain's trace [124, Theorem 4]. In contrast, in K Framework safety is expressed as the property that the *price* of LP tokens, given by the Dexter2 contract, is always correct. If one looks at the details of the Dexter2 mechanics, these two notions of safety can be understood intuitively to be similar, there is no obvious way to tell if these two notions of safety are equivalent to each other, or if they accurately reflect the notion of safety which itself is only alluded to in the informal specification [4]. Reflecting on the claim that Dexter2 is formally verified to be correct with regards to this safety property, are we sure that we even know what that actually means?

Because of these ambiguities, like Nielsen *et al.* [124] we argue that specification ought to be done in hand with formalization, either before or concurrently with implementation. In our case, this allows us to reason about the specification to verify its correctness before the contract need be implemented.

Secondly, if we formalize a specification ad hoc to verify a contract, we will end up verifying more properties about the contract than is necessary. We spent §4.1.2 proving that any contract satisfying the specification of §4.1.1 also satisfies each of those properties, so ideally if we were to formally verify a contract we would only prove the contract correct with regards to the specification of §4.1.2 and nothing more. If the proofs of §4.1.2 were formalized before verification, then we could restrict our verification work to proving the specification

of §4.1.1, a much simpler task. Furthermore, the reasoning of §4.1.2 can be done beforehand by experts, making it easier for practitioners to use the formal specification and verify their contracts correct.

We propose an alternative to the status quo of formal specifications that addresses this issue of rigor while also simplifying the verification process itself. This is through an abstract specification and metaspecification.

The abstract specification is a minimal, functional specification of the contract, and a formal analogue to what we saw in §4.1.1. It consists of a formal description of the contract's types and entrypoint functions, and is written as a Coq module. Its most important feature is that it allows us to talk abstractly about a specification S, and an arbitrary contract `C` which satisfies the specification S.

The goal of the specification is to axiomatize the necessary features of a smart contract so that it exhibits the desirable economic and other behaviors, and do so with formal precision. Its goal is also to be reusable, much like how anyone wishing to implement structured pools might consult the specification of §4.1.1, or anyone wishing to implement Dexter2 might consult its informal specification [4] to guide the implementation. Because it minimizes the properties of the specification, it also minimizes the work of formally verifying a contract correct with regards to this specification. From an abstract specification we can extract prose specifications, like what we have given in §4.1.1 and other market-making contract specifications given in the literature [63, 172].

The metaspecification is a specification of the abstract specification, which we use to reason about the abstract specification itself and prove it correct. It consists of a list of properties which assume in their context a contract `C` which satisfies a given specification S. These properties can be written in the same Coq module as the specification S, or in a module that imports S.

The purpose of the metaspecification is to reason about all the contract's properties which are not directly captured in the implementation details, but which are true of any contract which satisfies the specification. It consists of safety and economic properties, such as those which are proved by hand as in the examples of [63], [172] or §4.1.2, or those which can be implied by the context, such as the economic reasoning used to justify Uniswap's choice the curve $xy = k$ [25]. Because this reasoning is done on the specification itself, it only need be done once, and can be done by experts and then used by software engineers without needing the expertise; any contract proved correct with regards to the (simple and minimal) abstract specification inherits all these additional properties for free.

4.2.1 An Abstract Specification

Our goal in defining an abstract specification is to have a formal and concrete structure in Coq which can be used to write any specification, and which can be abstracted and reasoned about. To do so, we will outline

the structure of a specification here as a predicate S on smart contracts, where $S(\texttt{C})$ indicates that a smart contract `C` satisfies the specification S.

Following the pattern in §4.1.1, an *abstract specification* consists first of specifications of the contract's storage and entrypoint types, for which we use Coq typeclasses to specify constructors of the entrypoint type, as well as required features of the contract storage. It consists second of a functional specification of the entrypoint functions, which is a list of propositions describing nessecary and sufficient conditions for a successful call to the entrypoint, which characterize how a call to an entrypoint updates the contract storage or emitts transactions. Finally and crucially, an abstract specification is written as a Coq module which can be imported to reason about any contract in ConCert, which aggregates the propositions constituting the specification into a predicate S on smart contracts.

We describe the formalized specification of structured pools, but the full Coq module can be found here[1]. What remains of this section is self-contained, but will also serve as an excellent companion to reading the specification itself.

Type Specifications

As we saw in Chapter 3, a the `Contract` type is parameterized by four types: the entrypoint type, `Msg`, the storage type, `State`, the setup type, `Setup`, and the error type, `Error`. To specify a contract we must first specify features of each of these types.

State Specification

Consider first the specification of the contract storage type `State`. In §4.1.1 we specified that the storage had to keep track of the family T of tokens which can be pooled by the contract, an exchange rate r_x for each t_x in T, the balance x carried by the structured pools contract for each token t_x in T, the details of the pool token contract, and k, the total number of pool tokens in circulation. We formalize this via the following typeclass

```
1   build_state_spec {
2     stor_rates : T -> FMap token exchange_rate ;
3     stor_tokens_held : T -> FMap token N ;
4     stor_pool_token : T -> token ;
5     stor_outstanding_tokens : T -> N ;
6   }.
```

where `token` and `exchange_rate` types are both given by

[1] TODO include link to repository

```
Record token := { token_address : Address; token_id : N }.
Definition exchange_rate := N. (* we always divide exchange rates by 1_000_000 *)
```

The `State_Spec` typeclass specifies four functions from our `State` type which retrieve: the rates, a ledger of the token balances it holds, the pool token data, and the number of outstanding tokens. This might 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 ;
}.
```

which yields an implementation of the typeclass `State_Spec` simply by 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.

We note that exchange rates, treated as rational numbers in the specification of §4.1.1, are encoded here as natural numbers; as we will see, any time we use an exchange rate, we divide it by `1_000_000`, which is a standard way to approximate decimals in smart contracts, and thus some rational numbers, using natural numbers.

Entrypoint Type Specification

Our specification of contract entrypoints indicated that there be at least three: `DEPOSIT`, `WITHDRAW`, and `TRADE`, where `DEPOSIT` is used to deposit, or pool tokens, `WITHDRAW` is used to withdraw, or unpool, tokens, and `TRADE` is used to trade. This results in a typeclass specification which requires at least three constructors to the `Msg` type:

```
Class Msg_Spec (T : Type) :=
    build_msg_spec {
        DEPOSIT : pool_data -> T ;
        WITHDRAW : unpool_data -> T ;
        TRADE : trade_data -> T ;
    }.
```

The entrypoint interfaces are defined by the `pool_data`, `unpool_data`, and `trade_data` types: recall that the `DEPOSIT` entrypoint accept a token data of some t_x in T and some quantity q of tokens in t_x to be deposited;

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

the `WITHDRAW` entrypoint accept token data of some t_x in T and some quantity q of pool tokens the user wishes to burn in exchange for tokens in t_x;

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

and the `TRADE` entrypont accept token data of some t_x in T to be traded in, token data of some t_y in T 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 *)
}.
```

Error and Setup Specification

In order to be fully rigorous, we also specify the `Error` and `Setup` types with two typeclasses:

```
Class Setup_Spec (T : Type) :=
    build_setup_spec {
      init_rates : T -> FMap token exchange_rate ;
      init_pool_token : T -> token ;
  }.
```

the first indicating that to initialize the contract we only need the initial rates and the pool token information,

```
Class Error_Spec (T : Type) :=
    build_error_type {
      error_to_Error : error -> T ;
  }.
```

and the second simply giving a labeling to errors in the `Error` type for clarity in error handling. The `error` type is then defined to have certain basic errors.

```
Definition error : Type := N.
Definition error_PERMISSIONS_DENIED : error := 1.
Definition error_CONTRACT_NOT_FOUND : error := 2.
Definition error_TOKEN_NOT_FOUND : error := 3.
Definition error_INSUFFICIENT_BALANCE : error := 4.
Definition error_CALL_VIEW_FAILED : error := 5.
Definition error_FAILED_ASSERTION : error := 6.
Definition error_FAILED_DIVISION : error := 7.
Definition error_FAILED_TO_INITIALIZE : error := 8.
```

These will be useful in proving some of the properties of the metaspecification, but are not salient enough to have come through in the informal specifications of §4.1.1 or §4.1.2, nor critical to the core functionality of the contract.

Entrypoint Function Specifications

Now that we have the contract's types formally specified, we can specify the entrypoint functions.

Deposit

Recall our specification of §4.1.1 of the `DEPOSIT` entrypoint. It included: that the structured pool contract checks that t_x is in the token family; it then transfers q tokens of t_x to itself, which is done by calling the `transfer` entrypoint of t_x; it simultaneously mints $q * r_x$ pool tokens and transfers them to the sender's wallet; and finally, that this transaction be atomic.

Each of these properties, captured in the psudocode of Listing 4.1, needs to be formally specified, and can be done in three propositions. These are formalized as

- `pool_entrypoint_check`, which specifies that the entrypoint check the token to be pooled is in the token family (see line _);
- `pool_emits_transfer`, which specifies that a successful call to the `DEPOSIT` entrypoint results in an emitted transfer transaction of the correct form (see line _);
- `pool_emits_mint`, which specifies that a successful call to the `DEPOSIT` entrypoint results in an emitted mint transaction of the correct form (see line _);
- `pool_atomic`, which specifies that the `transfer` and `mint` transactions are the only transactions emitted by the contract (see line _) which implies their atomicity.

To illustrate, let us look at the proposition `pool_entrypoint_check`, which formalizes the check that the structured pool contract makes that t_x is in the token family.

```
(* When the POOL entrypoint is called, the contract call fails if
   the token to be pooled is not in the family of semi-fungible tokens. *)
Definition pool_entrypoint_check (contract : Contract Setup Msg State Error) : Prop :=
   forall bstate caddr cstate chain ctx msg msg_payload,
   (* reachable bstate *)
   reachable bstate ->
   (* the contract address is caddr with state cstate *)
   env_contracts bstate caddr = Some (contract : WeakContract) ->
   contract_state bstate caddr = Some cstate ->
   (* forall calls to the DEPOSIT entrypoint *)
   msg = DEPOSIT (msg_payload) ->
   (* the receive function returns an error if the token to be pooled is not in the
     rates map held in the storage (=> t_x is not in the token family) *)
   FMap.find msg_payload.(token_pooled) (stor_rates cstate) = None ->
   receive contract chain ctx cstate (Some msg) =
    Err(error_to_Error error_TOKEN_NOT_FOUND).
```

Listing 4.4: One of the propositions in the `DEPOSIT` entrypoint specification

Translated into prose, this reads something like:

Proposition 1. *For all reachable states* `bstate` *of the blockchain with the contract's address* `caddr` *and the contract's state* `cstate`*, if the token to be pooled,* t_x*, does not have an entry in the* `rates` *map of the storage, then the call fails with the error* `error_TOKEN_NOT_FOUND`*.*

The other propositions of the `DEPOSIT` entrypoint specification read similarly, assuming a reachable state of the blockchain `bstate` with contract address `caddr` and state `cstate`, and then reason about what a call to the contract's `receive` function, with a `DEPOSIT` message, can result in. A reachable blockchain state is one for which there is a trace from the genesis block. These often have Hoare-like formulations, for example the `pool_emits_transfer` and `pool_emits_mint` propositions which tie successful calls to the contract's `receive` function with specific resultant transactions.

Withdraw

Similar to the `DEPOSIT` entrypoint, the `WITHDRAW` entrypoint is characterized by four propositions:

- `unpool_entrypoint_check`, which specifies that the entrypoint check the token to be unpooled is in the token family (see line _);
- `unpool_emits_burn`, which specifies that a successful call to the `WITHDRAW` entrypoint results in an

emitted burn transaction of the correct form (see line _);

- `unpool_emits_transfer`, which specifies that a successful call to the `WITHDRAW` entrypoint results in an emitted transfer transaction of the correct form (see line _);
- and `unpool_atomic`, which specifies that the `burn` and `transfer` transactions are the only transactions emitted by the contract (see line _) which implies their atomicity.

Let us look this time at `unpool_emits_transfer`, the proposition that a successful call to the `WITHDRAW` entrypoint results in a call to the `transfer` entrypoint of a token contract, which we assume here is an FA2 token contract.

```
1  Definition unpool_emits_transfer (contract : Contract Setup Msg State Error) : Prop :=
2    forall bstate caddr cstate chain ctx msg msg_payload cstate' acts,
3    (* reachable bstate *)
4    reachable bstate ->
5    (* the contract address is caddr with state cstate *)
6    env_contracts bstate caddr = Some (contract : WeakContract) ->
7    contract_state bstate caddr = Some cstate ->
8    (* the call to UNPOOL was successful *)
9    msg = unpool (msg_payload) ->
10   receive contract chain ctx cstate (Some msg) = Ok(cstate', acts) ->
11   (* in the acts list there is a transfer call with q tokens as the payload *)
12   exists transfer_to transfer_data transfer_payload,
13   (* there is a transfer call *)
14   In
15   (act_call
16      (msg_payload.(token_unpooled).(FA2.token_address)) (* call to the token address *)
17      0 (* with amount = 0 *)
18      (serialize (FA2.Transfer transfer_payload)))
19   acts /\
20   (* with a transfer in it *)
21   In transfer_data transfer_payload /\
22   (* which itself has transfer data *)
23   In transfer_to transfer_data.(FA2.txs) /\
24   (* whose quantity is the quantity pooled *)
25   let r_x := get_rate msg_payload.(token_unpooled) (stor_rates cstate) in
26   transfer_to.(FA2.amount) = msg_payload.(qty_unpooled) * 1_000_000 / r_x.
```

Listing 4.5: One of the propositions in the `WITHDRAW` entrypoint specification

Written in prose, this reads something like:

Proposition 2. *For all reachable states of the blockchain* `bstate` *with the contract's address* `caddr` *and the contract's state* `cstate`*, a successful call to the* `WITHDRAW` *entrypoint with* t_x *and* q *in the payload emits a*

call to the address of the token t_x, calling the `transfer` *entrypoint, instructing it to transfer*

$$\frac{q * 1000000}{r_x}$$

tokens in t_x to the sender's wallet.

Note the use of pre-defined entrypoints and types, imported from the `FA2` module, as in `FA2.Transfer` and `FA2.amount`. This is a previously-formalized token contract conforming to the FA2 token standard of the Tezos blockchain. The token contract of t_x is assumed to be an FA2 token contract.

Trade

Recall our specification of the `TRADE` entrypoint from §4.1.1. It included: that the pool contract checks that both t_x and t_y are in the token family; 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, each of these properties, captured in the psudocode of Listing 4.3, needs to be formally specified, which can be formalized using the following auxiliary functions:

```
1   Definition calc_delta_y (rate_in : N) (rate_out : N) (qty_trade : N) (k : N) (x : N) : N :=
2     (* calculate ell *)
3     let l := N.sqrt (k * (1_000_000 * 1_000_000) / (rate_in * rate_out)) in
4     (* calculate the exchange rate *)
5     l * rate_in / 1_000_000 - k / ((l * rate_out) / 1_000_000 + qty_trade).
6
7   Definition calc_rate_in (rate_in : N) (rate_out : N) (qty_trade : N) (k : N) (x : N) : N :=
8     let delta_y := calc_delta_y rate_in rate_out qty_trade k x in
9     (rate_in * x / 1_000_000 + rate_out * delta_y / 1_000_000) / (x + qty_trade).
```

Listing 4.6: Auxiliary functions in the `TRADE` entrypoin specification

We formalize the specification in four propositions:

- `trade_entrypoint_check`, which specifies that the entrypoint check that the tokens to be traded in and out are both in the token family, and that the contract has sufficient balance to execute the trade (see line _);
- `trade_pricing_formula`, which specifies that the trade be calculated using the formula given by the function `calc_delta_y` given below (see line _);

- `trade_update_rates_formula`, which specifies that the exchange rate r_x be updated and updated using the formula from the function `calc_rate_in` given below (see line _);
- `trade_emits_transfer_tx` and `trade_emits_transfer_ty`, which specify that a successful call to the `TRADE` entrypoint result in two outgoing `transfer` operations (see line _);
- and `trade_atomic`, which specifies that the two `transfer` transactions are the only transactions emitted by the contract (see line _) which implies their atomicity.

This time let us look at the `trade_update_rates_formula` specification, the proposition that a successful call to the `TRADE` entrypoint will result in an updated exchange rate r_x for t_x using the formula given in the `calc_rate_in` function.

```
1  Definition trade_update_rates_formula (contract : Contract Setup Msg State Error) : Prop :=
2    forall bstate caddr cstate chain ctx msg msg_payload t_x t_y q cstate' acts,
3    (* reachable bstate *)
4    reachable bstate ->
5    (* the contract address is caddr with state cstate *)
6    env_contracts bstate caddr = Some (contract : WeakContract) ->
7    contract_state bstate caddr = Some cstate ->
8    (* the TRADE entrypoint was called succesfully *)
9    msg = trade msg_payload ->
10   t_x = msg_payload.(token_in_trade) ->
11   t_y = msg_payload.(token_out_trade) ->
12   t_x <> t_y ->
13   q = msg_payload.(qty_trade) ->
14   receive contract chain ctx cstate (Some msg) =
15     Ok(cstate', acts) ->
16   (* calculate the diffs delta_x and delta_y *)
17   let rate_in := (get_rate t_x (stor_rates cstate)) in
18   let rate_out := (get_rate t_y (stor_rates cstate)) in
19   let k := (stor_outstanding_tokens cstate) in
20   let x := get_bal t_x (stor_tokens_held cstate) in
21   (* the new rate of t_x is correct *)
22   let r_x' := calc_rate_in rate_in rate_out q k x in
23   FMap.find t_x (stor_rates cstate') = Some r_x'.
```

Listing 4.7: One of the propositions in the `TRADE` entrypoint specification

Written in prose, this reads something like:

Proposition 3. *For all reachable states of the blockchain* `bstate` *with the contract's address* `caddr` *and the contract's state* `cstate`*, after a successful call to the* `TRADE` *entrypoint which trades in* Δ_x *of* t_x *for* t_y*, the*

updated state has r'_x as the exchange rate for t_x, where r'_x is given by the formula

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

encoded in `calc_rate_in`.

Note that we require that rates be updated to be equal with this formula, but despite having an example implementation in `calc_rate_in` we do not mandate any specific implementation. The advantage of the implementation we give is that it is clear, and thus amenable to formal reasoning. One might implement a structured pools contract with a more efficient function, which may not be so convenient for formal reasoning, getting both correctness and efficiency so long as Proposition 3 holds.

The Abstract Specification

So far our specification is a list of propositions which characterize the contract types with typeclasses, the entrypoint functions with propositions. We now need a predicate S on smart contracts which abstracts the structured pools contract so that we can reason about it. This is given by `is_structured_pool`.

```
Definition is_structured_pool
    (* Contract type specifications *)
    `{Msg_Spec Msg} `{Setup_Spec Setup} `{State_Spec State} `{Error_Spec Error}
    (C : Contract Setup Msg State Error) : Prop :=
    (* the DEPOSIT specification *)
    pool_entrypoint_check C /\
    pool_emits_transfer C /\
    pool_emits_mint C /\
    pool_atomic /\
    (* the WITHDRAW specification *)
    unpool_entrypoint_check /\
    unpool_emits_burn /\
    unpool_emits_transfer /\
    unpool_atomic /\
    (* the TRADE specification *)
    trade_entrypoint_check /\
    trade_pricing_formula /\
    trade_update_rates_formula /\
    trade_emits_transfer_tx /\
    trade_atomic.
```

Listing 4.8: The `is_structured_pool` contract specification predicate

Our predicate `is_structured_pool` is parameterized by a smart contract, whose types conform to our type specifications, and requires proofs of each proposition in the specification. We can now reason abstractly

about an arbitrary contract `C` which satisfies our specification. This is done by assuming a proof of the proposition `is_structured_pool C`. Crucially, we can now reason about the correctness of the specification `is_structured_pool` by reasoning with a proof of `is_structured_pool C` in the context.

4.2.2 A Metaspecification

We have noted that, as a specification is used to prove that a contract is correct, by definition a metaspecification is used to prove a *specification* is correct. But what does it mean for a specification to be correct?

Let us reflect again on the economic attacks we encountered in Figure 2.1.4. Each of the examples we saw featured contracts that had a specification, and conformed to that specification. The issue was that the specification did not adequately capture the desired economic behaviors. Because the contracts were all behaving as specified, 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."[2]

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 the remainder of this chapter, a metaspecification is a list of properties, further to those we saw in the specification, that can be proved by assuming an arbitrary contract satisfying the specification we just saw. As we saw in the specification of §4.1.2, 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 desirable properties of a financial smart contract from the literature.

We have formalized and proved each of the six properties from §4.1.2 here[3], which justifies the correctness of our specification `is_structured_pools`. Take for example the theorem on demand sensitivity, which was Property 1 in §4.1.2.

```
1 Theorem demand_sensitivity :
2    forall bstate caddr cstate t_x r_x,
3    (* state is reachable *)
4    reachable bstate ->
5    (* the contract address is caddr with state cstate *)
6    env_contracts bstate caddr = Some (contract : WeakContract) ->
```

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

[3] TODO include link to repository

```
7    contract_state bstate caddr = Some cstate ->
8    (* r_x is the rate of token t_x *)
9    (FMap.find t_x (stor_rates cstate)) = Some r_x ->
10   (* a trade of t_x for some t_y happens *)
11   forall chain ctx msg msg_payload acts cstate',
12      receive contract chain ctx cstate (Some msg) = Ok(cstate', acts) ->
13      msg = trade msg_payload ->
14      msg_payload.(token_in_trade) = t_x ->
15      t_x <> msg_payload.(token_out_trade) ->
16   (* new cstate induced by *)
17   let r_x' := get_rate t_x (stor_rates cstate') in
18   (* r_x' goes down while all other rates stay constant *)
19   r_x' < r_x /\
20   forall t,
21   t <> t_x ->
22   get_rate t (stor_rates cstate') = get_rate t (stor_rates cstate).
```

Listing 4.9: The first of six propositions constituting the structured pools metaspecification

Written in prose, this reads something like:

Theorem 1. *Let* `bstate` *be a reachable state of the blockchain with the contract's address* `caddr` *and the contract's state* `cstate` *such that* r_x *is the exchange rate for the token* t_x *in* `cstate`*. Suppose a trade of* t_x *for some token* t_y*,* $t_x \neq t_y$*, succeeds. Then in the updated state* `cstate'` *with exchange rate* r'_x *for the token* t_x*,* $r'_x < r_x$*, and for all tokens* t_z*,* $t_z \neq t_x$*, with exchange rate* r_z *in* `cstate` *and* r'_z *in* `cstate'`*,* $r'_z = r_z$*.*

Crucially, the proof of Theorem 1 relies on property from the specification which gives the formula for how exchange rates are updated after a successful call to the `TRADE` entrypoint, which we saw earlier as Proposition 3. It also relies on a lemma which states that the `calc_rate_in` function is strictly decreasing, which we saw before in Property 1.

```
1 Lemma rate_decrease : forall r_x r_y delta_x k x,
2    calc_rate_in r_x r_y delta_x k x < r_x.
```

The proofs of the other properties of the metaspecification all follow patterns of proof similar to what we saw in §4.1.2.

4.3 Conclusion

In this chapter we introduced the notions of abstract contract specification and metaspecification in order to reason about the economic properties that the structured pools contract exhibits, and prove its specification correct. This allowed us to distil the specification of a structured pools contract into a list of requirements of the contract's entrypoint, storage, setup, and error types, as well as of the entrypoint functions, and prove formally that any contract satisfying those conditions also exhibits the desired economic behaviors we listed informally in §4.1.2 and formally in §4.2.2. This specification is reusable and can be used to reason about any smart contract.

The economic behaviors which we introduced in §4.1.2 were derived from previous work to formally understand AMMs and DeFi, which themselves used variations of formal models of the blockchain. Further work in this area could include formalizing these models within ConCert to make the derivation of these economic properties more precise, rigorous, and broadly applicable to financial smart contracts.

In the following chapters we will use the notion of an abstract specification and build on the notion of a metaspecification to prove that a specification has other desirable properties relating to contract upgrades and systems of contracts.

Chapter 5

Morphisms of Smart Contracts

In this chapter, we broaden the notion of a metaspecification and use it to reason about properties of smart contracts in relation to other contracts. There are good reasons to do this. As we saw in §2.1.2 and Figure 2.1.4 of the Introduction, 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. As we will show, being able to reason about contracts relative to each other can also help us safely optimize code.

This chapter is organized as follows: In §5.1, we discuss issues in formal specification and verification of contract upgrades. In §5.2, we introduce the notion of a morphism of contracts, which is a formal, structural relationship between smart contracts, as a tool to specify contract upgrades. In §5.3, we show how contract specifications can be related between smart contracts through a morphism of contracts, and encode this formally within a metaspecification. In §5.4 we reason about contract upgrades, drawing again on the example of structured pools from the previous chapter. In §5.5 we discuss further applications of morphisms of contracts to formal verification of smart contracts.

5.1 Contract Upgrades

Our previous chapters have been essentially motivated by the claim that there are contract vulnerabilities due to complex contract behavior which are difficult to specify and to reason about. The previous chapter sought to enable reasoning about complex economic properties of a contract specification. In this chapter, we will address the complex behavior of contract upgrades.

Generally speaking, there are no in-built methods for upgrading a smart contract once it has been deployed.

Indeed, contract immutability is part of its 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.

If one wishes to avoid encoding upgradeability into a smart contract, the other option for upgrades is through 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 [18, 91]. Especially for financial smart contracts such as AMMs which rely on liquidity providers, this transition can be expensive, difficult, and best avoided if possible [168].

To make upgrades safer and less unweildy, there are resources in the form of best practices [12, 13] and established upgrade frameworks [7, 9], the most popular of these being the EIP-2535 Diamond upgrade framework [7], which is a specification of a system of smart contracts which use a proxy framework to enable upgradeability. Even so, 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 an upgrade [31].

As we alluded to in §2.1.2 of the Introduction, in order to reason about contract upgrades properly, one must be able to understand how the upgraded version of a contract relates to the old version. Generally speaking, an upgrade happens with a goal in mind, whether it is patching a bug, adding functionality, or improving features. In each of these cases, the goal of the upgrade is best expressed in relation to the old contract: the new contract should eliminate a vulnerability the old contract had, or the new contract should be backwards compatible regarding its old functionality while adding new functions, or the new contract makes improvements on the old, for example it is more gas-efficient than the old.

Thus in our arsenal of formal verification tools, in order to reason in a fully rigorous manner about contract upgrades, we need a formal notion of a structural relationship between smart contracts which we can use to compare them. We present them here, and call them *morphisms of smart contracts*.

5.2 Morphisms of Smart Contracts

A morphism of contracts is a formal, structural relationship between smart contracts.

5.3 Specifications, Metaspecifications, and Morphisms

A contract morphism relates specifications of contracts. This opens a new avenue of expressivity for metaspecifications: we can formally reason about how two contracts relate to each other by reasoning about how one contract's specification should/does relate to another contract's specification.

5.4 Specifying a Contract Upgrade

5.5 Further Applications of Contract Morphisms

Chapter 6

Contract Composability and Weak Equivalences

Another metaspecification technique: From here we consider systems of contracts.

6.1 Reasoning About Systems of Contracts

Reasoning about systems of contracts is hard to do, and introduces complexity. For example, the Dexter2 verification.

6.2 Weak Equivalences

There is a weaker notion of equality than what we saw in the previous chapter.

6.3 Weak Equivalences and Specifications

How do weak equivalences relate to specifications and metaspecifications?

6.4 Specifying a System of Contracts

Dexter2 example.

6.5 Applications to Process-Algebraic Approaches

We can execute the semantics of process algebras with these weak equivalences, actually transforming contracts.

Chapter 7

Conclusion

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. 9, 11

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. 15, 39

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. 5, 35, 76, 78

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. 78

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. 15, 78

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. 35, 39, 72

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. 5, 11, 13

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. 5

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. 5, 12

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. 12

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. 66

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. 73, 78

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. 75

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. 5, 10, 64, 66, 74, 76, 78

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. 5, 13, 15, 65, 75, 78

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. 9

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. 8, 15

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. 6, 35

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. 12, 17, 72, 78

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. 9, 10, 12, 13

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. 9, 10, 68

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. 7, 14, 15, 35, 68

fungible See fungible token. 35, 65, 72

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. 69, 72, 78

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. 12

genesis block A genesis block is the first block in a blockchain. It is a special block that is hardcoded into the software of the blockchain and serves as the starting point for the entire blockchain. The genesis block typically includes a set of initial parameters that define the rules of the blockchain, such as the block reward structure, the difficulty adjustment mechanism, and the initial distribution of tokens. In most blockchains, the genesis block cannot be altered once the blockchain has been launched. The existence of a genesis block is what makes a blockchain a secure, decentralized, and trustless system, as it provides a clear starting point for all participants to agree on. 50

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. 6

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. 75

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. 10

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. 10

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. 6, 59

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. 10

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. 10, 44

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. 35

non-fungible See non-fungible token. 35, 65, 72

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. 35, 72, 78

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. 16, 34, 35, 65, 72

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. 35, 39, 65, 72

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. 10, 12

slippage In the context of decentralized exchanges and automated market maker (AMM) protocols, slippage refers to the difference between the expected price of an asset and the actual price at which it is traded, due to the changing market conditions. When a large trade is executed, it can cause a noticeable change in the price of an asset, leading to slippage. This is particularly relevant in DeFi, where price slippage can have a significant impact on the profitability of trading strategies such as arbitrage. 41

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. 5, 9

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. 10

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. 17

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. 5, 9, 13, 66

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. 15

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. 5, 73

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. 17

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. 10, 15, 35, 51, 52, 69, 72, 74, 78

tokenization Tokenization is the process of converting ownership rights or assets into a digital representation, called a token, which can be traded and managed on a blockchain or other digital platform. The tokenization of assets, such as real estate, commodities, or art, allows for a more efficient, transparent, and secure way of buying, selling, and managing these assets. By creating a digital token that represents an asset, it becomes possible to automate many aspects of ownership and transfer, such as tracking ownership and transfer of rights, and facilitating the buying and selling of the asset on a global marketplace. Tokenization can also bring liquidity to previously illiquid assets, making it possible to trade them in much smaller increments and with a wider range of participants. 75

tokenize See tokenization. 34

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. 34, 35

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. 9

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. 59

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. 12, 66

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. 12, 66

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. 36, 37, 52, 64

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. 10, 12, 13

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. 5, 12

Acronyms

AMM Automated market maker. 5–7, 9, 11, 13–16, 35, 38, 39, 41, 57, 59, 76

BSC Binance Smart Chain. 9, 10

CFMM constant function market maker. 15

dApp decentralized application. 73

DeFi decentralized finance. 9, 12, 13, 15, 38, 57, 71, 73

DEX Decentralized exchange. 10, 13, 66, 74, 76

ERC20 A token standard for fungible tokens on the Ethereum blockchain. 75

ERC721 A token standard for non-fungible tokens on the Ethereum blockchain. 75

ETH The native token of Ethereum. 12

FA1.2 A token standard for tokens on the Tezos blockchain. FA2 stands for "Financial Asset 1.2.". 75

FA2 A token standard for tokens on the Tezos blockchain. FA2 stands for "Financial Asset 2.". 51, 52, 75

NFT See non-fungible token. 35, 72

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] 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.

[4] Dexter2 Specification. https://gitlab.com/dexter2tz/dexter2tz/-/blob/master/docs/informal-spec/dexter2-cpmm.md.

[5] Dexter2 Specification (Mi-Cho-Coq). https://gitlab.com/nomadic-labs/mi-cho-coq/-/blob/dexter-verification/src/contracts_coq/dexter_spec.v.

[6] Disrupting financial markets - Decentralised Exchanges. https://thepaypers.com/expert-opinion/disrupting-financial-markets-decentralised-exchanges–1250818.

[7] EIP-2535: Diamonds, Multi-Facet Proxy. https://eips.ethereum.org/EIPS/eip-2535.

[8] PancakeBunny Faces FlashLoan Exploit Resulting In 97% Correction. https://www.bsc.news/post/pancakebunny-faces-flashloan-exploit-resulting-in-97-correction.

[9] Proxy Upgrade Pattern - OpenZeppelin Docs. https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies.

[10] Toucan Whitepaper. https://docs.toucan.earth/.

[11] UN Supports Blockchain Technology for Climate Action — UNFCCC. https://unfccc.int/news/un-supports-blockchain-technology-for-climate-action.

[12] Writing Upgradeable Contracts - OpenZeppelin Docs. https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable.

[13] Contract upgrade anti-patterns. https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/, September 2018.

[14] 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.

[15] Formal Verification Report: Tezos Dexter V2 Contract. Runtime Verification Inc., June 2021.

[16] Dexter2 Specification (K Framework). Runtime Verification Inc., August 2022.

[17] K-Michelson: A Michelson Semantics. Runtime Verification Inc., August 2022.

[18] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core, 2021.

[19] 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.

[20] 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.

[21] 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.

[22] Guillermo Angeris, Akshay Agrawal, A. Evans, T. Chitra, and Stephen P. Boyd. Constant Function Market Makers: Multi-Asset Trades via Convex Optimization. 2021.

[23] 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.

[24] 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.

[25] Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An analysis of Uniswap markets, February 2021. arXiv:1911.03380, doi:10.48550/arXiv.1911.03380.

[26] 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.

[27] 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.

[28] Danil Annenkov, Mikkel Milo, and Bas Spitters. Code Extraction from Coq to ML-like languages. page 4.

[29] 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.

[30] Danil Annenkov and Bas Spitters. Deep and shallow embeddings in Coq. TYPES, 2019.

[31] 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.

[32] Jun Aoyagi. Liquidity Provision by Automated Market Makers, May 2020. doi:10.2139/ssrn.3674178.

[33] Jun Aoyagi and Yuki Ito. Liquidity Implication of Constant Product Market Makers. *SSRN Electronic Journal*, January 2021. doi:10.2139/ssrn.3808755.

[34] 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.

[35] Avraham Eisenberg [@avi_eisen]. Mango Markets Exploit, October 2022.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] Massimo Bartoletti and Roberto Zunino. Formal models of bitcoin contracts: A survey. *Frontiers in Blockchain*, 2:8, 2019.

[43] 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.

[44] Carsten Baum, James Hsin-yu Chiang, Bernardo David, Tore Kasper Frederiksen, and Lorenzo Gentile. SoK: Mitigation of Front-running in Decentralized Finance, 2021.

[45] Rob Behnke. Explained: The NowSwap Protocol Hack (September 2021). https://halborn.com/explained-the-nowswap-protocol-hack-september-2021/, September 2021.

[46] Beosin. Beosin — Global Web3 Security Report 2022, January 2023.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] Bruno Bernardo, Raphaël Cauderlier, Basile Pesin, and Julien Tesson. Albert, an intermediate smart-contract language for the Tezos blockchain, January 2020. arXiv:2001.02630.

[53] 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.

[54] BscScan.com. Pancake Bunny Exploiter. http://bscscan.com/address/0x158c244b62058330f2c328c720b072d8db2c612f.

[55] BscScan.com. Spartan Protocol 2021 Exploit. http://bscscan.com/tx/0xb64ae25b0d836c25d115a9368319902c972a0215b

[56] BscScan.com. Uranium Finance Exploiter. http://bscscan.com/address/0x2b528a28451e9853f51616f3b0f6d82af8bea6ae

[57] COBRA Research Center. Concert. URL: https://github.com/AU-COBRA/ConCert.

[58] Martán Ceresa and César Sánchez. Multi: A Formal Playground for Multi-Smart Contract Interaction, July 2022. arXiv:2207.06681.

[59] 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.

[60] 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.

[61] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2022.

[62] 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.

[63] Michele Ciampi, Muhammad Ishaq, Malik Magdon-Ismail, Rafail Ostrovsky, and Vassilis Zikas. FairMM: A Fast and Frontrunning-Resistant Crypto Market-Maker, 2021.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] Christian Doczkal, Damien Pous, and Daniel Severin. Graph Theory. URL: https://github.com/coq-community/graph-theory.

[70] 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.

[71] Gregor Dorfleitner and Diana Braun. Fintech, Digitalization and Blockchain: Possible Applications for Green Finance. *Palgrave Studies in Impact Finance*, pages 207–237, 2019.

[72] 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.

[73] Michael Egorov. StableSwap-efficient mechanism for Stablecoin liquidity. *Retrieved Feb*, 24:2021, 2019.

[74] 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.

[75] 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.

[76] etherscan.io. Beanstalk Exploit. http://etherscan.io/tx/0xcd314668aaa9bbfebaf1a0bd2b6553d01dd58899c508d4729fa73

[77] etherscan.io. CREAM Finance Exploit. http://etherscan.io/tx/0x0fe2542079644e107cbf13690eb9c2c65963ccb79089ff96b

[78] etherscan.io. Harvest.Finance: Hacker 1. http://etherscan.io/address/0xf224ab004461540778a914ea397c589b677e27bb.

[79] etherscan.io. Nomad Bridge Exploit. http://etherscan.io/tx/0xa5fe9d044e4f3e5aa5bc4c0709333cd2190cba0f4e7f16bcf73

[80] 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.

[81] Harvest Finance. Harvest Flashloan Economic Attack Post-Mortem, October 2020.

[82] Uranium Finance. Uranium Finance Exploit, April 2021.

[83] 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.

[84] 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.

[85] Robin Fritsch. Concentrated Liquidity in Automated Market Makers. *DeFi@CCS*, 2021. doi:10.1145/3464967.3488590.

[86] Robin Fritsch, Samuel Kaser, and Roger Wattenhofer. The Economics of Automated Market Makers. 2022.

[87] Robin Fritsch and Roger Wattenhofer. A Note on Optimal Fees for Constant Function Market Makers. *DeFi@CCS*, 2021. doi:10.1145/3464967.3488589.

[88] 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.

[89] Florian Gronde. Flash Loans and Decentralized Lending Protocols: An In-Depth Analysis. page 75.

[90] 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.

[91] Lioba Heimbach, Eric Schertenleib, and Roger Wattenhofer. Risks and Returns of Uniswap V3 Liquidity Providers. *arXiv preprint arXiv:2205.08904*, 2022. arXiv:2205.08904.

[92] Lioba Heimbach, Ye Wang, and Roger Wattenhofer. Behavior of Liquidity Providers in Decentralized Exchanges. 2021.

[93] Celine Herweijer, Dominic Waughray, and Sheila Warren. Building block (chain) s for a better planet. In *World Economic Forum.*, 2018.

[94] 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.

[95] 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.

[96] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant a tutorial. *Rapport Technique*, 178, 1997.

[97] Igor Igamberdiev [@FrankResearcher]. BUNNY Tweet 1, May 2021.

[98] Immunefi. Hack Analysis: Cream Finance Oct 2021, November 2022.

[99] Immunefi. Hack Analysis: Nomad Bridge, August 2022, January 2023.

[100] PeckShield Inc. The Spartan Incident: Root Cause Analysis, May 2021.

[101] 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.

[102] Aljosha Judmayer, Nicholas Stifter, Philipp Schindler, and Edgar Weippl. Estimating (Miner) Extractable Value is Hard, Let's Go Shopping! *Cryptology ePrint Archive*, 2021.

[103] Roman Kozhan and Ganesh Viswanath-Natraj. Decentralized Stablecoins and Collateral Risk, June 2021. doi:10.2139/ssrn.3866975.

[104] 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.

[105] 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.

[106] 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.

[107] 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.

[108] 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.

[109] 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.

[110] 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.

[111] 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.

[112] Katya Malinova and Andreas Park. Market Design with Blockchain Technology, July 2017. doi:10.2139/ssrn.2785626.

[113] 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.

[114] Gary E. Marchant, Zachary Cooper, and Philip J. VI Gough-Stone. Bringing Technological Transparency to Tenebrous Markets: The Case for Using Blockchain to Validate Carbon Credit Trading Markets. *Nat. Resources J.*, 62(2):159–182, 2022.

[115] 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.

[116] 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.

[117] 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.

[118] Patrick McCorry, Chris Buckland, Bennet Yee, and Dawn Song. Sok: Validating bridges as a scaling solution for blockchains. *Cryptology ePrint Archive*, 2021.

[119] Eva Meyer, Isabell M. Welpe, and Philipp G. Sandner. Decentralized Finance—A Systematic Literature Review and Research Directions, 2022. doi:10.2139/ssrn.4016497.

[120] 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.

[121] Zeinab Nehai and François Bobot. Deductive Proof of Ethereum Smart Contracts Using Why3, August 2019. arXiv:1904.11281.

[122] 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.

[123] 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.

[124] Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Formalising Decentralised Exchanges in Coq, March 2022. arXiv:2203.08016, doi:10.48550/arXiv.2203.08016.

[125] 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.

[126] 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.

[127] Kris Oosthoek. Flash Crash for Cash: Cyber Threats in Decentralized Finance, June 2021. arXiv:2106.10740.

[128] 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.

[129] Yuting Pan, Xiaosong Zhang, Yi Wang, Junhui Yan, Shuonv Zhou, Guanghua Li, and Jiexiong Bao. Application of Blockchain in Carbon Trading. *Energy Procedia*, 158:4286–4291, February 2019. doi:10.1016/j.egypro.2019.01.509.

[130] pancakebunny.finance [@PancakeBunnyFin]. BUNNY Tweet 2, May 2021.

[131] 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.

[132] Christine Paulin-Mohring. *Introduction to the Calculus of Inductive Constructions*, volume 55. College Publications, January 2015.

[133] Daniel Perez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care? *arXiv preprint arXiv:1902.06710*, pages 1–15, 2019. arXiv:1902.06710.

[134] 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.

[135] 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.

[136] 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.

[137] 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.

[138] 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.

[139] 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.

[140] samczsun is occasionally shitposting [@samczsun]. Nomad Tweet Thread, August 2022.

[141] Soheil Saraji and Mike Borowczak. A Blockchain-based Carbon Credit Ecosystem, June 2021. arXiv:2107.00185, doi:10.48550/arXiv.2107.00185.

[142] Fabian Schär. Decentralized Finance: On Blockchain- and Smart Contract-Based Financial Markets, April 2021. doi:10.20955/r.103.153-74.

[143] 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.

[144] Tim Sheard. Accomplishments and research challenges in meta-programming. In *SAIG*, volume 2196, pages 2–44. Springer, 2001.

[145] 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.

[146] 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.

[147] solscan.io. Mango Markets Exploiter. https://solscan.io/account/CQvKSNnYtPTZfQRQ5jkHq8q2swJyRsdQLcFcj3Em

[148] Derek Sorensen. Structured pools for tokenized carbon credits. *Preprint*, 2023.

[149] Derek Sorensen. Tokenized carbon credits. *Preprint*, 2023. URL: https://derekhsorensen.com/docs/sorensen-tokenized-carbon-credits.pdf.

[150] 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.

[151] 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.

[152] Palina Tolmach, Yi Li, Shang-Wei Lin, and Yang Liu. Formal Analysis of Composable DeFi Protocols, April 2021. arXiv:2103.00540.

[153] 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.

[154] UniMath. Coq of ocaml. URL: https://github.com/formal-land/coq-of-ocaml.

[155] 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.

[156] Shermin Voshmgir and Michael Zargham. Foundations of Cryptoeconomic Systems. *Foundations of Cryptoeconomic Systems*, 2020.

[157] 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.

[158] Gang Wang. SoK: Exploring Blockchains Interoperability, 2021.

[159] Yongge Wang. Automated Market Makers for Decentralized Finance (DeFi), September 2020. arXiv:2009.01676, doi:10.48550/arXiv.2009.01676.

[160] 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.

[161] 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.

[162] 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.

[163] 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.

[164] Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols, January 2022. arXiv:2103.12732, doi:10.48550/arXiv.2103.12732.

[165] Zheng Yang and Hang Lei. Fether: An extensible definitional interpreter for smart-contract verifications in coq. *IEEE Access*, 7:37770–37791, 2019.

[166] 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.

[167] 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.

[168] Jimmy Yin and Mac Ren. On Liquidity Mining for Uniswap v3. *arXiv preprint arXiv:2108.05800*, 2021. arXiv:2108.05800.

[169] 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.

[170] 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.

[171] 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.

[172] 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.

[173] 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.