UNIVERSITY OF CAMBRIDGE

Formal Tools for Specifying Financial Smart Contracts

Derek Sorensen

Clare College

September 2023

This dissertation is submitted for the degree of Doctor of Philosophy

Declaration

Derek Sorensen
September, 2023

Abstract

Formal Tools for Specifying Financial Smart Contracts

Derek Sorensen

Financial smart contracts routinely manage billions of US dollars worth of digital assets, and as a consequence bugs in smart contracts can be extremely costly. Because of this, much work has been done in formal verification of smart contracts to prove a contract correct with regards to its specification. However, financial smart contracts have complicated specifications, and it is not all straightforward to write one which correctly describes its intended behaviors. As a response to this challenge, we develop formal tools for specifying financial smart contracts. We target aspects of contract specification which are typically difficult to address and which can be a source of expensive contract vulnerabilities. In doing so, we hope to expand the capability of formal methods to specify desired contract behavior and thereby prevent catastrophic loss of funds.

Acknowledgements

First to my primary supervisor Anil, for giving me a second chance at a PhD when my first topic did not work out as I'd hoped; for his unbounded enthusiasm in pursuing meaningful and impactful things; for his exceptional leadership; for his wise advice in business and research; for pressing me to find one single, coherent thread to my thinking that transformed this thesis it into what it is now.

To my secondary supervisor Keshav, for his extraordinary and unrelenting ability to find weaknesses in my thinking; for his instructive stories; for his groundedness that forces me to confront the world as it is and do research that is relevant to the world; for forcing me to justify all of my ideas with the utmost of rigor; and for taking the time to meticulously read my work.

To the Choir of Clare College, which gave me new life and love for the city of Cambridge and for my college after the profound isolation and suffering of successive lockdowns, having spent more than four months without getting within two meters of another human soul; for the music that inspires me to commune with the Divine and which has taught me to see research and mathematics in the transcendence that they embody; for the music, the parties, the late nights, the trips, and the unforgettable memories.

To the Rev'd Dr Mark Smith, for our many chats on life, faith, and theology; for our trips to the pub in between lockdowns; for our many Bible study sessions; for making the chapel a sacred space of worship—all of which ultimately culminated in my baptism. To the Chapel of Clare College, for our hilarious lunches, formals, and trivia nights; for the moral support to continue on with a PhD through the depths of lockdown; for making a community of faith in chapel.

To Mike Stay, for giving a wide-eyed and over-confident algebraic topologist his first job in formal methods and giving him the freedom and flexibility to relate to it on his own terms.

To my mother for her quiet but vigorous encouragement to avoid abandoning my goals, and to my father whose unbridled confidence in my abilities has given me new (and perhaps unsubstantiated) confidence when I have found myself discouraged.

And finally to Luke, the love of my life and the missing puzzle piece for this document. You are infused into the beating heart of this work and all of my work to come.

Words cannot express my gratitude; thank you all.

Contents

Chapter 1

Introduction

Smart contracts are programs stored on a blockchain that automatically execute when certain predefined 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 [228]. 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 [244].

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 is frequently worth the large overhead cost in time and expertise to formally verify a smart contract before deployment.

1.1 Formal Specification and Verification of Smart Contracts

Much work has been done in formal verification of smart contracts [29, 54, 64, 90, 139, 208, 218]. Broadly speaking, the goal of formal verification is to prove that a contract is correct with regards to a specification. However, financial smart contracts have complicated specifications, and there are several common and desirable properties of smart contracts which are not at all straightforward to correctly specify.

1.1.1 Dexter2 Verification

To illustrate, consider the formal verification work on Dexter2, an AMM on the Tezos blockchain. Dexter2 has been formally verified by at least three different groups using three different formal verification

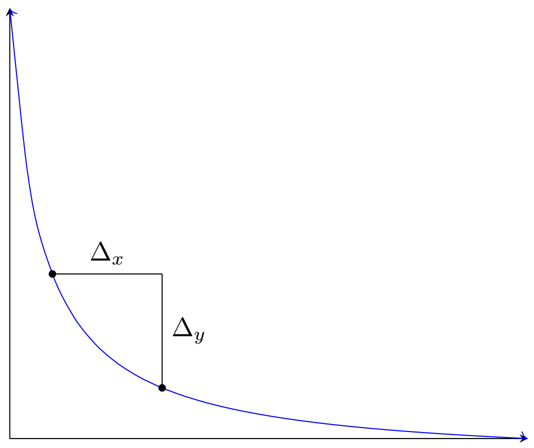

Figure 1.1: A trade of Δ_x for Δ_y along the indifference curve $xy = k$.

tools [50, 111, 146]. Each was based on the same informal specification [27]. The informal specification describes the contract interface, including its entrypoint functions, error messages, outgoing transactions, the contents of its 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.

However, while the specification is detailed on the contract design and interface, it doesn't include anything about expected or desired behavior from an economic perspective. This is not because the expected or desired economic behavior is unknown or uninteresting. It was clearly articulated by Vitalik Buterin, co-founder of Ethereum, in what he wrote in March 2018 about AMMs on the online forum Ethereum Research [49]. Buterin proposed that AMMs trade between a pair of tokens along an *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 a constant. The tokens held by the contract come from liquidity providers, which are investors who deposit tokens into the AMM in exchange for a reward, most often a share of transaction fees. That k is constant means that a trade of Δ_x of one token yields Δ_y of another such that the product from (1.1) stays constant at k:

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

Buterin argued that an AMM that trades along (1.1) features efficient price discovery. He also argued that it can properly incentivize liquidity providers by charging a 0.3% fee on each trade to give to them.

We can probably convince ourselves that the informal specification of Dexter2 [27] features 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; however, concluding that the informal specification [27] or its formal counterparts [50, 111, 146] actually imply any of these economic behaviors is not a given fact. AMM fees and liquidity provision alone are highly complex topics, and are the subject of several economic studies, including on choosing optimal transaction fees [75, 85, 84], how liquidity providers react to market changes [100], and how all of that relates to the *curvature* of the indifference curve $xy = k$ [15]. 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 [62].

There is then an outstanding question of whether any of Dexter2's specifications are *correct* with regards to the desired—indeed, essential—high-level economic and game-theoretic behavior of the resulting AMM. Not only do these important economic and game-theoretic factors not make it into any of Dexter2's specifications, formal or informal, but to complicate matters a brief study of the three formalizations [50, 111, 146] of Dexter2's informal specification [27] reveals that each differs substantively both in how the properties of the informal specification are formalized, and in what assumptions are made in the process of verification. (More on this in Chapter 2.)

Furthermore, were we to try to formally specify some of the aforementioned high-level economic and game-theoretic behaviors on the Dexter2 contract it is not at all obvious how we would go about doing so. Most specification languages and models of computation are too low-level to easily admit clear specifications of such properties. What model or framework would we use, for example, to specify that Dexter2 facilitate an efficient market?

As codebases go, Dexter2 is quite small—only a few thousand lines—but as we've seen much is expected of it in terms of its economic and game-theoretic behavior; because AMMs like Dexter2 routinely manage billions of USD worth of funds, much is also at risk in case of a bug. Furthermore, despite the fact that they are not present in any of Dexter2's specifications, it is difficult to conceptualize a notion of Dexter2's *correctness* that does not include some of the economic properties that we have just mentioned, directly or indirectly. It is, however, not obvious that these properties hold by reading any of the aforementioned specifications. Thus we are left with the problem of how to formally specify that required behavior, which leads us to introduce the problems in contract specification and verification which we will address in this thesis.

1.2 Challenges in Contract Specification

Creating a formal framework in which one can perfectly articulate the intended behavior, from all levels of analysis, of a financial smart contract is far out of scope of what we could ever hope to present here. We can however identify some challenges to doing so and take some modest steps in that direction. In what remains of this chapter we outline three challenges to formally specifying and verifying financial smart contracts which relate to the complex nature of formally specifying said contracts. These are the challenges of reasoning about: a specification's completeness; upgradeable contracts and contract upgrades; and contract optimizations and optimized code. These challenges are the subject of the forthcoming work.

1.2.1 Challenge 1: Reasoning About a Specification's Completeness

Our first challenge is to understand how complete a contract specification is so as to not miss bugs due to underspecification. An ideal formal specification is *consistent* (free of contradiction) and complete (fully descriptive of contract behavior) [55, 230]. One can prove a formal specification consistent by producing an implementation and proving it correct with regards to that specification, but asserting that a specification is complete is far more complex. To do so we must have some way of asserting that the contract specification behaves correctly at all levels of analysis; a specification that permits unintended and pathological behavior would be incorrect by any reasonable standard.

This is often more subtle than it seems, even in the flexible and expressive setting of natural language specifications. Take for example the extensively documented specification of Beanstalk, an Ethereum-based stablecoin protocol which uses a decentralized governance protocol. This governance protocol features an emergency commit function, which gives a supermajority of governance votes power to quickly respond to an emergency by approving and executing a proposal in one single vote. The goal of said emergency commit function is to allow governance to respond quickly to an emergency, providing added security in the face of unforeseen vulnerabilities.

In a turn of grim irony, this very emergency response mechanism turned out to be the source of a catastrophic exploit. On April 17, 2022, an attacker used a flash loan to temporarily buy a supermajority of governance tokens and execute a proposal, draining the contract of approximately 77 million USD in lost contract assets [73]. A flash loan is a loan mediated by a smart contract and issued for the duration of a single, atomic transaction. Due to its atomicity, a flash loan removes the creditor's risk of debt default, and thus enables enormous, uncollateralized loans. For example, the Aave flash loan pool has had in excess of 1 billion USD which can be loaned out [192]. Flash loans can introduce unintuitive contract behavior, deviating from that of traditional markets [94, 96, 192, 226]. Importantly, the Beanstalk attack leveraged the unexpected behavior due to the availability of flash loans, exploiting the faulty design of the governance mechanism rather than incorrect code [78].

Flash loans are often a source of unexpected contract behavior which complicates the task of writing a complete (or rather, a correct) specification. More examples of successful flash loan attacks include attacks on the Spartan Protocol and Pancake Bunny, two DeFi contracts on the Binance Smart Chain (BSC). Attackers used flash loans to make huge trades and temporarily manipulate market prices of certain assets. Both of these contracts used these market prices in the contract logic, and in both cases this lead to pathological—though, again, correct according to the flawed specification—contract behavior. In May 2021, an attacker drained the Spartan Protocol contract for a profit of roughly the equivalent of 30 million USD [48, 110]. Another attacker drained Pancake Bunny of 114k WBNB and 697k BUNNY tokens, amounting to about 45 million USD at the time in lost funds [47, 97, 107, 182].

Finally, Mango Markets, a Solana-based DEX, was attacked in October 2022 for approximately 116 million USD in contract assets [213]. The attack consisted of a complicated and subtle trading strategy which only a sophisticated trader would be able to see and exploit [40]. CoinDesk reported that the attacker did everything within the parameters of the platform's design [127]. Avraham Eisenberg, the attacker, wrote:

> *I believe 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.* [25]

We would recognize each of these exploits as attacks, despite the fact that the contracts were functioning as specified, which tells us that the specifications were erroneous as they did not accurately capture the intended economic behaviors. To the contrary, they permitted unintended and pathological behavior.

Issues of incomplete specifications are common sources of vulnerabilities for financial smart contracts because the implications of a specification can be difficult to understand. Crucially, formal verification is useless to mitigate these vulnerabilities without a formal framework for understanding the completeness or correctness of a specification because they are vulnerabilities of the specification itself. In an attempt to mitigate this challenge, in Chapter 4 we offer a formal framework of contract *axiomatization* and *metaspecificaiton* that could help us address the challenge of formally reasoning about the completeness of a formal specification.

1.2.2 Challenge 2: Reasoning About Contract Upgrades

Our next challenge is to understand how to reason about contracts as they iterate through upgrades. Despite being immutable once deployed, nearly all smart contracts undergo regular versioned upgrades. This most often takes the form of a contract hard fork, where the contract developers urge their users to use the most recently-deployed contract. However, this can also be done via *upgradeable* contracts, or contracts which include a predefined process for proposing and executing changes to the contract functionality—an upgrade. Irrespective of the method of upgrade, smart contract upgrades are costly from a verification perspective and introduce complications that can be a meaningful source of vulnerabilities when done incorrectly.

The particular issue on which we will focus is a common failure to align the *intent* of an upgrade with its actual resulting specification and code, which can result in undiscovered vulnerabilities in the upgraded contract. For example, consider Nomad, a cross-chain bridge protocol. In August 2022, more than 500 hacker addresses exploited a bug introduced by a faulty upgrade to one of the Nomad smart contracts [201]. The upgrade incorrectly added the null address (`0x000...000`) as a trusted root, which turned off a key safety check, allowing anyone to withdraw arbitrary amounts of funds from the Nomad contract to their wallet by calling the contract with a particular payload. The attack resulted in 190 million USD in lost funds [74, 109]. Similarly, Uranium Finance, an AMM, suffered a costly exploit after a faulty 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 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 USD of contract funds [81]. NowSwap, a nearly identical application, upgraded with the same error and incurred a loss of 1 million USD [36].

It is clear that none of these contract upgrades captured the actual intent of the upgrade because for each the new contract version violated critical safety invariants that held for the previous version. The Uranium Finance and NowSwap vulnerabilities had easy solutions, using global constants, but even so these are examples that point to a deeper problem for formal verification. Instead of reusing any work on previous contract versions, one must repeat the full verification process for each contract upgrade. This relies heavily on fallible intuition, can lead to unexpected vulnerabilities, and drives up the cost of formally verifying smart contracts. Furthermore, because verifying software is time, labor, and resource intensive, it can be difficult to justify formally verifying software which may be upgraded quickly or frequently—a problem shared with other verified software [195, 233]. Each of these factors limit the effectiveness of formal methods to address security issues in real-world software, inhibiting verification as business and security propositions [207].

What is needed is a practical and formal framework through which to formally specify and verify contract upgrades. As it stands we have no such framework apart from repeating the formal specification and verification process on a new contract version. We will address this in Chapter 5 by introducing *contract morphisms* as a tool for formally specifying and verifying contract upgrades and upgradeability.

1.2.3 Challenge 3: Reasoning About Optimized and Performant Code

The final challenge we consider here is how to effectively reason about optimized and performant code. The efficacy of formal verification to prevent actual, critical contract vulnerabilities depends on the feasibility of applying formal verification to deployable contract code. However, deployable code, optimized for performance, is typically more difficult to formally reason about than a reference implementation. Code highly optimized for performance thus risks vulnerability due to the difficulty of formal reasoning, while code written for ease of formal reasoning may not be efficient enough for the resource-scarce environment of smart contracts. Ideally, we would reason about contracts in a state optimized for formal reasoning while still deploying them in a state optimized for efficiency and gas consumption.

As it stands no formal framework exists to address this problem for smart contracts. To that end in Chapter 6 we introduce a formal framework of extensional equivalence between smart contracts in Coq, called *contract isomorphisms*. These equivalences will allow us to port proofs between contracts that can be proved to be bisimilar, and to use contracts themselves as specifications. Their eventual goal is to enable reasoning about optimized contracts in terms of more intelligible ones.

1.3 Formal Tools for Specifying Financial Smart Contracts

The approach we take to address these three challenges draws on the robust mathematical setting offered by Coq. Axiomatization, morphisms, and isomorphisms are all classical mathematical techniques that we formalize in ConCert to add to the language of formal specification. We hope to enable practitioners to more accurately and rigorously specify financial smart contracts in a formal setting, ultimately preventing catastrophic loss of funds.

We proceed as follows. In Chapter 2, we survey related work; as our work is in Coq, we focus on formal verification in proof assistants and the advantages of their uniquely mathematical setting. In Chapter 3, we give the necessary background in Coq and ConCert, the formal framework on which our work builds. In Chapter 4, we address the challenge of reasoning about completeness by introducing the formal framework of contract axiomatization and metaspecification. In Chapter 5, we address the challenge of reasoning about contract upgrades by introducing the formal tool of contract morphisms. In Chapter 6, we address the challenge of reasoning about optimized and performant code by introducing the formal tool of contract isomorphisms. In Chapter 7, we conclude.

1.4 Associated Publications

Sorensen, D. *(In)Correct Smart Contract Specifications.* ICBC 2024. (Chapter 4)

Sorensen, D. *Towards Formally Specifying and Verifying Smart Contract Upgrades in Coq.* FMBC 2024. (Chapter 5)

Sorensen, D. *Formally Specifying Contract Optimizations With Bisimulations in Coq.* FMBC 2025. (Chapter 6)

Sorensen, D. *Structured Pools for Tokenized Carbon Credits.* ICBC 2023. (Appendix A)

Sorensen, D. *Tokenized Carbon Credits.* Ledger, 2023. (Appendix B)

Chapter 2

Related Work

The background setting of our work is primarily smart contract verification in interactive theorem provers (ITPs), or proof assistants, such as Agda, Lean, and Coq. We survey ITP-based tools for formally verifying smart contracts, focusing on the advantages offered by the uniquely mathematical verification setting of an ITP. The formal tools of our work here rely on these robust mathematical foundations.

We proceed as follows. In Section 2.1 we survey the most prevalent technique for ITP-based contract verification, a language embedding into a proof assistant. We survey low-, intermediate- and high-level language and virtual machine embeddings into proof assistants. In Section 2.2 we survey high-level verification tools for smart contracts which are not language embeddings, but which are either built on top of language embeddings or can be formally linked to one. In Section 2.3 we consider formal specification in theorem provers, studying specific instances of contracts verified in proof assistants. We consider the issue of *correct specification*, and how proof assistants may offer a solution unique to other proof tools for smart contracts. We conclude in Section 2.5 with challenges and future directions of proof-assistant-based smart contract formal verification.

2.1 Smart Contract Language Embeddings

By far the most common form of ITP-based smart contract verification tool is an embedding—deep, shallow or both—of a smart contract language into the proof assistant. In many ways this makes perfect sense. Smart contracts execute in the siloed environment of a blockchain and execution semantics are both deterministic and well-defined. In contrast with software which interacts with the physical world, in smart contract verification there is seemingly little more to consider to its behavior than the execution from its perspective.

This section consists of a survey of such language embeddings, which can be neatly classified as low-, intermediate- or high-level language embeddings. Seeing as at the time of writing Ethereum is the blockchain with the most active development [120], most of these tools target either the Ethereum virtual machine (EVM) or Solidity, the high-level language most commonly used to develop smart contracts for EVM-comaptible chains [211]. There are other blockchains and languages for which ITP-based verification tools have been made—most prominently, the Tezos blockchain—and we also include those tools.

2.1.1 Low-Level Language Embeddings

The most important set of examples of low-level language or virtual machine embeddings are that of the Ethereum virtual machine (EVM). EVM bytecode, like other low-level languages, is seldom written directly, but it is useful to do formal reasoning at the bytecode level in order to avoid (indeed, discover) any bugs due to a compiler, *e.g.* from Solidity [211], and minimize the trust base of the verification method.

The first formal definition of the EVM that implemented all of its op codes was in Lem [106], a formal language for defining semantic models which can be translated into OCaml for testing, Coq, HOL4, and Isabelle/HOL for proof, and LaTeX and HTML for presentation [138]. The stated goal of this formalization in Lem is to serve as a basis for smart contract verification that targets proof assistants in particular as the verification tool. This is in contrast to a tool like the K Framework [197] and its associated EVM formalization [105], which use alternative forms of proof and verification (though are still interactive).

The great downside to interactive verification at the bytecode level is the unintelligibility of proof goals as well as the sheer tedium of writing proofs. There have been some efforts to mitigate this. The first we mention is ETH-Isabelle [11], a tool based on the embedding of the EVM in Isabelle/HOL due to the EVM semantics in Lem [106]. ETH-Isabelle addresses the intricacy of manually verifying EVM bytecode by abstracting EVM instructions into blocks which can be composed, and on which one can use a Hoare-style logic to derive semantic properties of composed blocks from the properties of its parts. The logic they propose is designed to express complex safety and security requirements at a higher level which makes manual verification of EVM bytecode more intelligible and feasible. This approach is not dissimilar to embeddings of medium- and high-level languages, except that the analogue to a compiler in this case is given by these bytecode blocks and the associated logic. However, these abstractions are not as expressive or intricate as something like Solidity, so the trust basis remains small.

Another good set of examples are formalizations of the EVM into Why3 [242, 143], a tool for formal specification and verification in which proofs can be discharged with the help of both automated and interactive theorem provers [80]. The specification language of Why3, WhyML, is a dialect of ML designed to generate first-order proof obligations. These can then be translated to various tools, including Z3 [63], an SMT solver, or Coq [60]. The idea here is to make verifying EVM bytecode feasible by discharging as many proof obligations as possible to an SMT solver and interactively proving anything else.

The final approach to interactively verifying EVM bytecode that we will see is an embedding of the EVM into Coq [26]. Rather than stating and proving specific results on contracts, this approach uses automated techniques to harden contracts at the bytecode level against known vulnerabilities. The EVM embedding into Coq is used primarily to prove input-output equivalence between the original and hardened bytecode. It also leaves erroneous states undefined so that proved theorems implicitly guarantee that the contract avoided those states.

There are similar embeddings for other smart contract languages, including for the Tezos smart contract language Michelson, a low-level, stack-based language [10]. Mi-Cho-Coq, for example, is an embedding of Michelson into Coq [42]. Like EVM bytecode, Michelson is typically too low-level to read or write in, and similar to before we have WhyISon, a tool which transpiles Michelson into WhyML [61], thus facilitating both manual and automated reasoning on Michelson smart contracts.

Our set of examples of low-level language embeddings has by no means been exhaustive, but it illustrates important points about smart contract verification in proof assistants. The first is that the manifold benefits of interactive verification come at the cost of it being *interactive*. Thus one continually encounters a tradeoff: Low-level embeddings into proof assistants minimize the trust base and maximize rigor, but because these tools require interaction that comes at the cost of intelligibility and feasibility. Strategies to address this tradeoff come either in the form of outsourcing feasible goals to an automated proof assistant, thus minimizing the need for prover interaction, or of some sort of abstraction—be that in the logical framework or in the language itself—so that proof becomes more intelligible and thus more feasible.

Our primary interest here is in those tools of abstraction used to make stating and proving properties of a contract's specification more feasible in the interactive setting. We continue in this section by discussing medium- and high-level language embeddings into ITPs and the associated abstractions. We continue the discussion on abstraction later on in 2.2 and 2.3.

2.1.2 Intermediate- and High-level Language Embeddings

Intermediate- and high-level languages have the advantage of intelligibility and can be more straightforward to reason about [122], especially if they are statically typed functional programming languages [19]. However, the abstraction may 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 [122]. As Li *et al.* observe in [122], higher-level languages generally add to the intuiteveness and manageability of verification, while low-level languages minimize the trust base of verification.

Once again we start with Ethereum-based contract verification, this time with tools that target Solidity [211]. As we will see, to rigorously verify contracts in higher-level languages we need some sort of foundational machinery that connects the higher-level language with bytecode or something analogous.

Our first example is is FSPVM-E [238], a formal symbolic process virtual machine embedded into Coq that targets a subset of Solidity called Lolisa [237]. To formally verify contracts in Lolisa (Solidity), FSPVM-E includes a Lolisa interpreter, FEther, the so-called execution engine of Lolisa which includes executable formal semantics verified in Coq [236]. Finally, FSPVM-E includes a formal memory model, the GERM framework, which virtualizes memory hardware, basic memory operations, and pointer arithmetic in Coq.

Other tools take the perhaps more straightforward approach of a deep or shallow embedding of Solidity into a proof assistant. Take for example Isabelle/Solidity, a deep embedding of (a subset of) Solidity into Isabelle/HOL [130]. Isabelle/Solidity is an executable denotational sematantics for Solidity in Isabelle/HOL [129] which supports domain-specific primitives (*e.g.* money transfer), a gas model, and domain-specific automated proof methods. These semantics were verified as a legimate implementation of Solidity against Ethereum's test suite.

Other embeddings of Solidity into proof asistants follow a similar pattern of formalizing the semantics of Solidity (or something close to it) through a deep or shallow embedding [30, 143, 193]. Others opt for formally verifying contracts at an intermediate language level, *e.g.* Yul, in Isabelle/HOL, targeting EVM bytecode [122], and Albert, in Coq, targeting Mi-Cho-Coq [43], arguing that this approach takes a middle ground of low- and high-level language embeddings and enjoys the benefits of both. Finally, others embed DSLs that target languages like Solidity or Michelson which specialize in the properties that they can express and verify. Two examples of these are Scilla, which can be used to reason about temporal properties of smart contracts, and which targets Solidity [206], and Archetype, a Tezos-based DSL which targets business logic and uses Why3 [41].

2.2 High-Level Tools in Proof Assistants

We have now seen tools that offer some advantages of using proof assistants—rigor, for example—but we could reasonably compare these to competing methodologies outside of proof assistants and would not be faulted for opting to use those due to the fact that, in general, other tools offer more by way of automation. In this section, we present work that highlights the great strengths of theorem provers in formal verification: First is the ability to systematically, rigorously and flexibly build high-level, abstract theories *on top of* lower-level tools such as language embeddings. This lets us do high-level reasoning while keeping assumptions inherent to modeling to a minimum. Second and complementary to this is the mathematically robust nature of the verification environment. We illustrate by example, noting that many of these tools are brand new or still under development.

Take first SSCalc [131], a framework for verifying Solidity contracts that is built on top of Isabelle/Solidity [130]. This work proposes a calculus for verifying Solidity programs, extending traditional calculi. Crucially, because it is developed in a proof assistant the authors are able to formalize this calculus and mechanically prove its soundness. This allowed them to build a verification condition generator to facilitate formal proof about Solidity contracts. Crucially, the entire theoretical foundation of this verification tool—from the language embedding to the high-level embedded calculus—are developed within a formal setting, which allows for not only a formal proof of soundness but also a formalized connection to bytecode from high-level semantics.

On the other hand, there are formalization projects that embed not only the semantics of languages, but of that of the entire blockchain [52]. By far the most well-developed of these is ConCert [19], a Coq-based tool which models the execution semantics of third-generation blockchains and which has a certified extraction mechanism from Coq code into multiple smart contract languages [20], including into CameLIGO, an intermediate-level language which compiles down to Michelson [16]. The embedding of the full semantics of the blockchain enables ConCert to reason more rigorously, and with custom Coq tactics, about smart contract execution by inducting along the execution trace of a blockchain. One can also reason about arbitrary multi-contract systems, instead of reasoning about properties exclusively from the perspective of a single contract; the latter is the standard for language embeddings (see *e.g.* Section 2.3.1) [146, 147].

2.3 Formal Specification and Verification in Proof Assistants

This brings us to the topic of formal specification in interactive theorem provers. To motivate our discussion, we first consider some examples of actual, deployable smart contracts verified in the wild with proof assistants. We see that formal specification is not a precise science. Indeed, one informal specification will inevitably have multiple formal interpretations, and it is not at all obvious how to compare formalizations on their correctness, or faithfulness to the intended contract design.

This problem is, of course, not unique to proof assistants; however, we argue that proof assistants offer a unique solution to the problem of incorrect formal specification. In particular, we argue that by creating high-level tools with low-level semantics (mirroring the high-level language embeddings of Section 2.2), we can evaluate the correctness of high-level smart contract specifications with semantics in low-level proof concepts. This facility is unique to ITPs, and we argue that it presents perhaps the most compelling argument in favor of ITPs as a proof tool for smart contracts.

2.3.1 Smart Contracts Verified Interactively

We now consider various examples of smart contracts verified interactively, focusing on the formal specification itself and our ability to evaluate whether it correctly captures the intended properties, both high- and low-level.

2.3.1.1 Dexter2, a Formally Verified AMM

Dexter2 is a Tezos-based automated market maker (AMM) modeled off of Uniswap V1. It was formally verified by three different groups in three distinct formal settings: in Mi-Cho-Coq, the low-level Michelson embedding into Coq that we saw previously [3]; in K Framework, using the K Michelson formalization [9]; and in ConCert, the Coq-based tool which has an embedding of the execution semantics of a blockchain and verified extraction [146]. Each of these were formalizations of the same informal specification [27].

All three formalizations include a functional specification. The K Framework and ConCert formalizations also include high-level properties which we will discuss later. From the functional perspective, each formal specification differs slightly in the properties formalized and verified. This is in part due to differing formal models of contract execution—K Framework and Mi-Cho-Coq each have to make their own set of assumptions about external contracts and the execution environment which ConCert doesn't need to make. For example, K Framework assumes that the evaluation of operations follows a depth-first search, something that ConCert makes a point not to need to assume. Both K Framework and Mi-Cho-Coq also make assumptions about safety properties of external contracts and their interaction with the main Dexter2 contract, while ConCert formally implements token interfaces and formally models the interacting contracts, avoiding such assumptions. That there are differences between the quality of the three formalizations is made most obvious in the fact that ConCert was the only team to find logical errors in the informal specification which could have lead to critical vulnerabilities, and which were reported to the Dexter2 team and later rectified [146].

Even though the informal specification of Dexter2 is low-level, technical, and clear [27], there was still sufficient nuance that made different formalizations not straightforwardly equivalent. We might ask ourselves how to evaluate any of these formal specifications for correctness.

One way to evaluate a specification's correctness is whether it implies the high-level properties and behaviors. We have one example of a high-level correctness property that was verified by both the K Framework and ConCert teams. The Dexter2 main contract keeps track of token balances in its storage and those balances should reflect actual token balances recorded in the respective token contracts. It is straightforward both to argue and see that this property of consistency between contract states is part of the intended design, and thus qualifies as a property of contract correctness. Under some assumptions, the K Framework team is able to verify a high-level safety property of state consistency between the contracts. The ConCert team also articulated an analogous formal property aiming at the same safety property of state consistency, showing novel formal techniques to reason about smart contract interactions.

Interestingly, the K Framework team articulated an additional high-level safety property which they argue implies that it is impossible to profit by incorrect rounding. The statement of this property as part of the formal specification draws on the reader's intuitive understanding that an AMM should not allow a trader to profit because the contract incorrectly priced a trade. While the property is not explicitly part of the formal specification, most readers would agree that a violation of this rule would be unfair, and thus constitute a bug.

It is proved by showing that the liquidity share price never decreases. Formally, they consider three state variables $XtzPool$, $TokenPool$, and $LqtTotal$ which are updated, respectively, to $XtzPool'$, $TokenPool'$, and $LqtTotal'$ after an arbitrary operation. The formal invariant expressing this high-level property is that the following formula always holds:

$$\frac{XtzPool' \times TokenPool'}{XtzPool \times TokenPool} \geq \left(\frac{LqtTotal'}{LqtTotal}\right)^2$$

The K Framework team argues that, taken in conjunction with the consistency property mentioned previously, the above invariant implies that: when adding or removing liquidity, users cannot mint more liquidity shares or redeem more assets than they should be able to; that users cannot get more than they should in trades; and that updating the token pool cannot be exploited despite the non-atomicity.

Of course, a formal justification for this intuitive argument is impossible within the K Framework without substantially extending the formal model to include cryptoeconomic properties. Even so, the intuitive properties which they claim to be the consequence of the articulated formal property are compelling and arguably part of the inherent intent of Dexter2's design. Providing an example where one of those intuitive properties is violated would be a compelling example of a bug.

The contrast between these formalizations show us that formalizing a specification is not at all straightforward, that some are in some sense more correct than others, and that a specification can include high-level properties which are not actually part of the informal specification, but which can be informally derived by specification's intent and design. In particular, it shows the sizeable role of *intuition* in the formal specification process. These differences in formalization are certainly not unique to proof assistants. Due to the nature of prose, informal specifications, any formalization necessarily requires making choices in the formalization that cannot be deterministically derived from the informal specification. Indeed, this is part of the reason that some promote writing formal specifications, even without verification–the process of formalization forces the one formalizing to be extremely precise.

For us, the natural question is then whether there is any way to formally evaluate the correctness of a formal specification or of contract design. This would consist of evaluating whether or not a functional specification correctly articulates necessary and sufficient conditions for correct execution, as well as articulating correct high-level properties implied by the design of the specification. The answer is tentatively in the affirmative. First, we will see now that previous work has verified standalone specifications in proof assistants, targeting correct design. Second, we argue that a formal analysis of contract design can, in principle, be built inside ITPs without loss of rigor. we expand on both of these points in Chapter 4

2.3.1.2 Djed, a Formally Verified Stablecoin

To that end consider Djed, a stablecoin protocol designed to behave like an autonomous central bank and verified in Isabelle/HOL [240]. The designers of Djed were more interested in the verification of correct design than verifying a particular implementation to be correct. This is because the design of Djed is complex and such that correctness can be expressed in eight so-called *stability properties*, which are precise, mathematical statements of desirable high-level economic behavior. The goal of formalization then was to prove that the functional specification and design actually implied those eight high-level economic properties.

The team reported positive results from the formalization. Despite having proved the eight stability properties on paper about the protocol design before formalization, they reported that the formalization uncovered issues like implicit and/or missing constraints (*e.g.* lower or upper bounds), assumptions in the informal proofs, unnecessary assumptions, missing case distinctions, ambiguous wording, and missing or unnecessary steps in proofs [240]. One of the formulae of the on-paper formulation was also shown to be incorrect and had to be relaxed. It shows that protocol design is very difficult to adequately specify and reason about on paper; the errors typically come in the form of unknown or incorrect foundational assumptions, problems due to approximations, or inexhaustive case reasoning.

This is our first glimpse at the power of ITP-based verification to evaluate the correctness of a contract specification. Because a contract implementation and specification are written in the same language and setting, we can specify and reason about the *specification* in the same way that we reason specify and reason about an implementation. In particular, had this analysis been done on a specific formal specification with regards to which an implementation was proved correct, then that implementation would also inherit the desired high-level properties.

We were fortunate in this case because the desired high-level, cryptoeconomic properties were easy to state in a formulaic and mathematical way, so no additional theory had to be built up in order to rigorously analyze the correctness of this particular contract's design. This won't always be true, and to be able to state a wider range of desirable, high-level properties we will likely need to embed a theory of some kind into our ITPs. Indeed, this is precisely the subject of recent work.

2.3.1.3 A Formally Verified Generic AMM Protocol

The authors of a recent paper [191] formalize a minimal state-machine model of a blockchain in Lean4, with tokens, balances, and transfers as primitives, and study the behavior of an AMM from a game-theoretic and cryptoeconomic standpoint. The main results are economic in nature: they first quantify (and prove) a user's gain from a trade on the AMM with an explicit formula; from there, they prove game-theoretic results about the optimal strategy of a user of the AMM and "construct the optimal swap transaction that a rational user can perform to maximize their gain, solving the arbitrage problem." [191]

Similar to the previously-mentioned work on Djed, the goal of [191] (though not stated in these precise terms) is to verify the *specification*, or design, of Uniswap V2 to be correct with regards to a cryptoeconomic specification—a set of desirable game-theoretic and economic behaviors of the AMM. Principal among these is the property that AMMs like Uniswap V2 respond to arbitrage efficiently, becoming effective price oracles. In contrast to Djed, in order to state these properties rigorously some additional economic primitives and theory had to be formalized into Lean4. Note that were there an underlying language embedding (*e.g.* of Solidity) or a model of a blockchain in Lean4 (*e.g.* something like ConCert [19]) nothing would inherently impede us from formalizing the primitives of this theory in terms of contract storage and transactions. This would give semantics of the high-level theory in terms of something with which we could verify actual contracts; such contracts would inherit any high-level properties we can prove in the high-level theory.

2.4 Proof Assistants and Embedded Theories

These examples have shown us that in an ITP we can formalize contracts, their execution environments, their specifications, and theories through which we can reason about the economic behavior of those specifications all in one single formal setting. Furthermore, the mathematically robust nature of an ITP means that we can state arbitrary properties and formulate arbitrary theories to inform the process of formal specification and verification. The process of reasoning about a contract's high-level economic properties, or indeed properties of its specification, mirrors the spectrum of low-, medium-, and high-level language embeddings into proof assistants. As one moves up in the abstraction of the tool, to be fully rigorous one must certify that the associated assumptions and primitives are themselves correct. We saw in Sections 2.1 and 2.2 that this can be done within ITPs.

We argue that the natural conclusion of recent work to verify contract design within ITPs is to embed high-level economic theories, within which one can specify and verify desirable high-level economic properties of smart contracts, into a setting (*e.g.* one of the tools of Section 2.1) in which one can verify actual contract code. If done correctly, the high-level theory can be given low-level semantics and can be certified to be correct. This would highlight the unique mathematical strength of ITPs, in that in an ITP one can state and prove mathematical theorems and arbitrary properties of programs. To our knowledge no other formal setting can do this rigorously, so we argue that ITPs uniquely offer the capacity to rigorously and interactively reason about the correctness of contracts as well as their specifications.

2.5 Conclusion

Interactive proof tools for smart contracts mostly consist of language embeddings into ITPs. Because in an ITP users construct proofs themselves, proving properties of contracts with a low-level language can be tedious. Embeddings of higher-level languages increase the trust base to include the compiler, but with the benefit that reasoning is more tractable. However, due to the mathematically robust nature of ITPs, one can prove a compiler to be correct, so high-level proof tools can be used rigorously on low-level code so long as the details of the abstraction are certified to be correct.

It is this same property that gives ITPs a capacity to not only formally specify a contract, but to also reason at a high level about the contract's specification (or design). Because smart contracts most frequently manage money, there is typically a set of desired economic properties which are implied by a contract specification but not explicitly stated (and rarely verified), like those we saw in Chapter 1. If done rigorously, high-level ITP-based tools to reason about contract design seen previously could at least in principle be given semantics in a low-level language embedding or something of the kind. The setting of an ITP means then that a contract proved correct with regards to a specification would automatically inherit all of the high-level properties proved of its specification. This, we argued, is the unique potential of ITPs, and is something that we explore in depth in the forthcoming work.

The setting and potential we have described for ITPs is vast, and something that we cannot fully realize in the forthcoming work. However, we will take modest steps in that direction, and leverage the unique mathematical strength of ITPs to introduce mathematical tools for specification and verification that target the high-level and difficult-to-specify properties corresponding to each of the three challenges outlined in Chapter 1. Each of these tools are high-level and mathematically abstract in nature, but because they are implemented in ConCert they have semantics in smart contracts; and due to ConCert's verified extraction mechanism, they can be used to reason about actual, deployable smart contract code.

To our knowledge, no previous work has leveraged the mathematical setting of ITPs in the way we do here for formal specification.

Chapter 3

Background

Before moving on to the core of our work, we give a short introduction to ConCert, the formal verification tool in which this work is built. As we mentioned in Section 2.2, ConCert is a high-level tool which is an embedding of the execution semantics of third-generation blockchains in Coq and which features verified extraction. It is one of the most theoretically mature tools for ITP-based smart contract verification; for the reader interested in the fine details of ConCert there is a maturing literature on the tool, including [18, 51, 17, 145, 148, 20, 21].

In Chapter 2 we studied its most high-profile use case, the verification of Dexter2, an automated market maker on the Tezos blockchain [145]. However, ITP-based verification for smart contracts is still finding its footing [53]. The primary drawback to using ConCert is that one must manually implement a contract to be verified within ConCert as in [145]; while ConCert features verified extraction and could thus could be used for verification-first development, to our knowledge no deployed contract has been developed in this way. Thus far non-ITP tools such as the Move Prover [67], K Framework [197], and the Certora Prover [2] have been applied to the most high-profile use cases [33, 90].

We will give a high-level overview of ConCert in Section 3.1, and more Coq-specific details in Section 3.2. In each, we cover the implementation of the blockchain and of contracts, and we cover proof tactics for properties and invariants of smart contracts. Section 3.2 is intended for readers who are familiar with Coq but not with ConCert, and is designed to prepare the reader to read code blocks included in the main body of this thesis. However, each chapter is prefaced with a theoretical, Coq-independent description of the formalized work, so a working understanding of Coq and the specific details of ConCert are not necessary to understand the work of this thesis.

3.1 ConCert From a High Level

Contracts are formalized in ConCert in the context of the execution semantics of a blockchain. This means that in ConCert there are data types corresponding to the chain state, the execution environment, and the context for any actions taken on chain. Actions include native token transfers, contract calls, and contract deployments. Valid steps for the chain can either be: a valid action execution; an invalid action which results in a revert case; the addition of a block, which adds actions to the action queue to be executed; or a permutation of the action queue. Permutation of the action queue is for the sake of generality, to accommodate blockchain implementations that use depth-first and breadth-first orders of contract call execution [148].

A contract `C` is modeled in ConCert with four types: `Setup`, the data for contract initialization; `Msg`, the type of messages a contract can receive; `State`, the contract's storage; and `Error`, a type describing the contract's errors (typically `N`). Contracts also have a function governing contract initialization, `C.(init)`, and a function which governs calls to the contract, `C.(receive)`. The `init` function takes a chain state, the context of a call to initialize a contract, and something of the contract's setup type. It returns a result of the contract's state after setup or it reverts with an error. The `receive` function takes a chain state, the contract call context, a state, and a message (the payload of the contract call). It returns the resulting contract state and a list of resulting actions to be made by the contract, or it reverts with an error.

Once a contract is formalized, one can state and prove any properties about it that can be expressed in Coq. One can prove a Hoare-style property by assuming a chain state, context, and contract state that has some pre-defined properties and then stepping through valid steps of the blockchain. One can prove an invariant about a contract through *contract induction*, a custom tactic that inducts over the trace of a contract from initialization to an arbitrary reachable contract state. These methods make ConCert extremely strong theoretically, as ConCert is not merely a language embedding, but a language embedding in the context of the semantics of a blockchain.

We will use these proof tactics when reasoning about specific formalized contracts and examples in each of Chapters 4, 5, and 6. We take advantage of the formalized blockchain semantics in Chapter 6 to formally prove equivalences of contract traces.

3.2 Continuing in More Detail

In the remainder of this chapter we introduce the types and tactics of ConCert which are most relevant to the forthcoming work. These are details for readers who are familiar with Coq but not with ConCert, and are designed to prepare the reader to read code blocks included in the main body of this thesis. However, each chapter is prefaced with a theoretical, Coq-independent description of the work to be done. So, for any readers unfamiliar with Coq or ConCert, the main body of this thesis should still be accessible.

We first look at the type of smart contracts in ConCert, the `Contract` type, and at the types which underlie the blockchain's execution semantics. The latter abstracts the execution semantics at two levels: the `Environment` type, and the `ChainState` type, each of which can be acted on, respectively, by the `Action` and `ChainStep` types to model the progression of an executing blockchain.

We then discuss what contract specifications and proofs of contract invariants look like in ConCert, covering ConCert's central custom Coq tactic, `contract_induction`. For any interested reader, the codebase and thorough documentation can be found at the ConCert GitHub repository [51].

3.2.1 Smart Contracts in ConCert

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

```
Record Contract (Setup Msg State Error : Type) :=
  build_contract {
    init :
      Chain -> ContractCallContext -> Setup ->
        result State Error;
    receive :
      Chain -> ContractCallContext -> State -> option Msg ->
        result (State * list ActionBody) Error;
  }.
```

Listing 3.1: The type of smart contracts in ConCert is a record type with two functions: `init`, which governs contract initialization, and `receive`, which governs contract calls.

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.
- The `ContractCallContext` type carries information about the context of a contract call, including the transaction sender, the transaction origin, the contract's balance, the amount of the native token (*e.g.* ETH or XTZ) sent in the transaction.
- The `Setup` type indicates what information is needed to deploy a contract.
- The `State` type is a 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 produce `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 with the `receive` function and emits 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.

To deal with Coq's polymorphism, ConCert also features a serialized contract type `WeakContract`, though anyone doing contract verification work in ConCert should not ever encounter the `WeakContract` type explicitly. We will see the `WeakContract` type briefly in various definitions relevant to the chain's execution semantics later on. Note that, while we omitted it in Listing 3.1, because contracts need to be serialized, all four types parameterizing a contract must be serializable.

```
Inductive WeakContract :=
    | build_weak_contract
        (init :
          Chain ->
          ContractCallContext ->
          SerializedValue (* setup *) ->
          result SerializedValue SerializedValue)
        (receive :
          Chain ->
          ContractCallContext ->
          SerializedValue (* state *) ->
          option SerializedValue (* message *) ->
          result (SerializedValue * list ActionBody) SerializedValue).
```

Listing 3.2: The `WeakContract` type is a serialization of the `Contract` type used interally to ConCert to deal with contract polymorphism. It is defined coinductively with the `ActionBody` type.

3.2.2 The Blockchain in ConCert

In ConCert, the blockchain and its execution semantics are modeled at multiple levels of abstraction, which we go through here. Underlying everything is a typeclass, `ChainBase`, which represents basic assumptions made of any blockchain. This is almost always abstracted away when reasoning about smart contracts.

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

```
      address_countable :> countable.Countable Address;
      address_serializable :> Serializable Address;
      address_is_contract : Address -> bool;
    }.

```

Listing 3.3: The `ChainBase` typeclass, which represents basic assumptions made of any blockchain.

The basic assumptions of the `ChainBase` typeclass include 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 represents the view of the blockchain that a contract can access and interact with. The only information this type carries is the chain height, the current slot of a given block, and the finalized height.

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

Listing 3.4: The `Chain` type, which represents the view of the blockchain that a contract can access and interact with.

From here, we have two types: the `Environment` type, which augments the `Chain` type to model the information that a realistic blockchain needs to implement operations, and the `ChainState` type, which augments the `Environment` type to include a queue of pending transactions that need to be executed.

The `Environment` type includes data about account balances, which contracts are at which addresses, and the states of deployed contracts.

```
  Record Environment :=
    build_env {
      env_chain :> Chain;
      env_account_balances : Address -> Amount;
      env_contracts : Address -> option WeakContract;
      env_contract_states : Address -> option SerializedValue;
    }.
```

Listing 3.5: The `Environment` type augments the `Chain` type to model the information that a realistic blockchain needs to implement operations.

The `ChainState` type augments the `Environment` type to add a queue of outstanding transactions, shifting our view from the chain's internal environment at any given block height to an external view of the chain itself, which executes transactions in a block.

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

Listing 3.6: the `ChainState` type augments the `Environment` type to include a queue of pending transactions that need to be executed.

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) []);
  }.
```

Listing 3.7: The `ChainBUilderType` typeclass characterizes implementations of blockchains.

3.2.3 Blockchain Semantics in ConCert

The `Environment` and `ChainState` types can be acted on by actions which represent 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;
  }.
```

Listing 3.8: The `Action` type, which includes the action's origin, the sender, and the action's body.

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

Listing 3.9: The `ActionBody` type, which specifies that a user can interact with the blockchain by transferring funds, calling a contract, or deploying a contract.

Every action carries with it the origin, `act_origin`, the sender, `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` involves sixty-six lines of code, so we give a shortened version here.

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

Listing 3.10: The `ActionEvaluation` links two inhabitants of the `Environment` type to represent a blockchain making progress by evaluating an action.

The `ActionEvaluation` 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 models a blockchain making progress by evaluating an action, moving from the previous environment to a new environment.

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

```
Inductive ChainStep (prev_bstate : ChainState) (next_bstate : ChainState) :=
  | step_block :
      forall (header : BlockHeader),
        (* some omitted checks *)
        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.
```

Listing 3.11: The ChainStep type links two inhabitants of the ChainState type to represent a blockchain making progress.

The ChainStep type is parameterized by two chain states, the previous state prev_bstate, and the new state, next_bstate, and represents an update to the chain's state. The chain's state can be updated by: updating the environment with an inhabitant of an ActionEvaluation type, as given by step_action; adding a block, given by step_block; showing an action to be invalid, given by setp_action_invalid; or reordering the blockchain's transaction queue. Reordering the transaction queue is 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.
```

Listing 3.12: The ChainTrace type, which models the chained history of a blockchain, and can be used to define the notion of a reachable chain state.

The ChainedList type models the chaining of points in some arbitrary type by a type of links, as follows.

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

Listing 3.13: The ChainedList type, described in the ConCert documentation as a proof-relevant transitive reflexive closure of a relation.

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. In particular, the ChainTrace type gives us the notion of a reachable state of a blockchain, defined as a state to which there is a trace from the empty state, empty_state.

```
Definition reachable (state : ChainState) : Prop :=
  inhabited (ChainTrace empty_state state).
```

Listing 3.14: The definition of a reachable state of a blockchain.

Many proofs of contract invariants begin by assuming a reachable chain state.

3.2.4 Specification and Proof in ConCert

A contract specification is simply a list of propositions, written in Coq, about a smart contract. For practical verification work, a specification typically references 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 Setup, Msg, State, and Error types.

```
Definition Setup := unit.

Inductive Msg :=
| incr (n : N)
| decr (n : N).

Record State :=
  build_state { stor : Z  }.

Definition Error : Type := N.
```

Listing 3.15: The counter contract's four types Setup, Msg, State, and Error.

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

```
(* entrypoint functions *)
Definition incr_funct (n : N) (st : State) :=
    {| stor := st.(stor) + (Z.of_N n) |}.
Definition decr_funct (n : N) (st : State) :=
    {| stor := st.(stor) - (Z.of_N 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.
```

Listing 3.16: The counter contract's main functionality.

Finally, we can construct an inhabitant of `Contract` by defining `init` and `receive` 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.
```

Listing 3.17: An inhabitant of the `Contract` type, defined by the `init` and `receive` functions.

Now that we have our contract `counter_contract` defined, we can prove invariants 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).
```

Listing 3.18: An invariant on `counter_contract`, which says the state of the counter is always the sum of all the `incr` calls minus the sum of the `decr` calls.

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

Theorem 1 (`counter_correct`). *For all 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.

We can prove invariants like `counter_correct` with `contract_induction`, ConCert's custom tactic which inducts over a contract's execution trace. To prove an invariant by contract induction one proves it for a base case, contract deployment, and then for the inductive step, which consists of the various ways a blockchain can make progress.

The `contract_induction` tactic divides the proof of a contract invariant into seven subgoals. In the first six subgoals, one must (re)establish the invariant after:

1. deployment of the contract (the base case),
2. addition of a block,
3. an outgoing action,
4. a nonrecursive call,

5. a recursive call, and
6. permutation of the action queue.

Finally, in each of these steps, one can introduce facts about the contract to help with the proof. These must be proved in the seventh subgoal.

We use contract induction in each of the subsequent chapters to prove invariants in ConCert.

3.3 Conclusion

ConCert is a powerful tool for formally verifying smart contracts in Coq. It formalizes the execution semantics of a blockchain, including the execution environment for smart contracts, valid steps, revert cases, and the addition and permutation of a block. We will make use of these formalizations to not only state and prove properties of contracts in each of the upcoming chapters, but to extend the formal framework of ConCert and reason about relations and equivalences of contracts.

Chapter 4

Axiomatization and Metaspecification

In this chapter we address the first challenge to contract specification, detailed in Section 1.2.1.

4.1 Introduction

Poorly specified smart contracts can be vulnerable to attacks on faulty design [119]. Examples of such attacks, typically targeting poor economic or governance design, are alarmingly common, costing the equivalent of billions of US dollars in cryptocurrency losses each year [40, 245]. (See also Section 1.2.1.)

The nature of these attacks means that they are rarely targeted by formal methods, as smart contracts can be *correct*, but with regards to a faulty specification [46, 187]. Furthermore, many vulnerabilities relating to poor economic or governance design are out of scope of a specification [53]. So, while specifications attempt to target these properties, anyone formally verifying said contracts can only make informal arguments to justify many design choices as correctly capturing the intended properties and behaviors of the smart contract in question.

We are thus in need of a paradigm shift in how we specify and verify smart contracts which allows for a rigorous and accurate notion of a contract specification's *correctness*, especially with regards to properties intended by it, but ultimately out of its scope. We advocate here for an approach to formal contract specification with interactive theorem provers (ITPs) consisting of *axiomatization* and *metaspecification*.

First, we note that for ITP-based verification, a consistent specification forms the basis for an axiomatized theory. This is because a specification is a set of propositions which characterize a contract's design and structure. We can thus state these propositions as a specification and study the behavior of arbitrary contracts satisfying that specification. Importantly, to reduce the burden of formally verifying any given contract, specifications should be minimal [230]. This is contract axiomatization.

From there we formally study the implications of a contract specification via a metaspecification. The metaspecification is to a specification what a specification is to an implementation. It consists of properties which justify the specification as being complete, or implying the correct desired contract behavior, and can include properties intended by, but out of scope of, the contract specification. A contract specification's *correctness*, then, depends on whether it is consistent, admitting a correct implementation, and in some sense complete, conforming to its metaspecification.

We make the specification-metaspecification distinction because the cost of formally verifying software can already be prohibitive, and we wish to address issues of poor specification in formal methods without unnecessarily augmenting the burden of verification on any given smart contract. By treating the specification as a contract axiomatization, we keep it minimal while expanding the formal study of smart contract behavior, adding to the security guarantees of formal methods without increasing the burden of verifying a specific implementation once the specification is formalized.

We proceed in this chapter as follows. In Section 4.2, we give historical context to this problem and discuss related work. In Section 4.3, we discuss the problem of correct specification. In Section 4.4 we discuss our proposed framework of contract axiomatization and metaspecification. In Section 4.5, we illustrate with an example contract. In Section 4.6, we justify this paradigm as a solution to our issue of (in)correct specification. In Section 4.6.1, we discuss the relationship of our work to refinement types. In Section 4.7, we discuss limitations and future work. In Section 4.8, we conclude.

4.2 Related Work

Limitations of formal methods are well-established [82, 119, 215]. We know that formal methods cannot guarantee perfect software [98], in part because of theoretical limits of a causal model of a physical process [44, 79]. As such, formal methods should be used in conjunction with other techniques to ensure software security which compensate for limitations inherent to formal methods [45, 104].

Design and formation of a specification has long been considered in the domain of informal techniques, out of the bounds of formal methods [93, 119]. The literature rightly points out that the infrastructure required to reason about software specification is vast. In order to reason about a specification's correctness, one must have a formal model of its execution model and—intractibly for most software—the social and ecological environment in which that software operates and executes, consisting of different types of users as well as society and the natural environment around them [119]. The limitation of formal methods due to the difficulty of forming a *correct* specification has long been recognized [93].

Even so, there have been efforts of varying formality to address the issue of (in)correct specification. Firstly, it is often argued that formalizing a specification alone, due to the precision required, helps ensure a specification's correctness by clarifying details and preventing inconsistencies [45, 134]. Specification languages are also frequently designed to target certain domain-specific properties in order to ease the translation between prose and formal specification [218].

There is also emerging work which attempts to formally justify the correctness of a contract specification. For example, one study tests the strength of a contract specification by mutation testing to identify any pathological yet correct (per the specification) behavior of Ethereum smart contracts [188]. Similarly, the developers of a formally verified stablecoin, Djed, used techniques such as mutation and unit testing to identify potentially pathological behavior of the specification. They then targeted these behaviors with formal verification, in Isabelle and using SMT solvers, to justify the robustness of the contract specification [239].

These are good examples of developers considering the correctness of their specifications, but crucially the properties they proved about these specifications are articulated and proved *ad hoc*. Testing and intuition ground the conceptual framework from which they derive the results to be proved. In particular, they are not derived systematically via a theory of some kind. Without a systematic framework, one has no rigorous notion of *completeness*—whether the propositions proved about a specification are sufficient to guarantee it to be correct.

Finally, there are some verification efforts which take a stronger theoretical approach to smart contract specifications, but these do not reason about deployable or executable code. Consulting and auditing firms such as Gauntlet [8] and 20squares [1] perform statistical, economic, and game-theoretic analysis on contract specifications [38, 91], but crucially such analyses are not present in any setting of formal verification.

Our work is to lay the theoretical foundations for a systematic framework that can evaluate the correctness of a smart contract specification based on cryptoeconomic analysis, and which brings a high-level approach of cryptoeconomic reasoning into a setting of verification on contracts which can be deployed and executed. The purpose of this work is to improve the efficacy of formal methods against attacks on poor cryptoeconomic design.

4.3 The Problem of (In)Correct Specification

Contract specifications almost never feature economic properties, despite the fact that the primary use case for smart contracts is as economic or financial infrastructure. Instead, the specifier goes through an informal translation process from a high-level, informal business or economic specification into a technical specification [217]. This informal process can be erroneous, resulting in a specification that fails to capture the intended cryptoeconomic properties and contract behaviors—in other words, an *incorrect* specification.

Consider the specification of an automated market maker (AMM). From its inception, its design is to facilitate an efficient market, with efficient price discovery [49]. Bonding curves, *e.g.* the first and most fundamental

$$xy = k, \tag{4.1}$$

were put forward from classical economics as having desirable market properties. However, reading through the specification of an AMM—take for example Dexter2 [27], a Tezos-based AMM, its formal counterparts [50, 111, 146], or a generic formal specification for AMMs using equation 4.1 as the bonding curve [243]—does little to convince us that the resulting smart contracts do indeed exhibit the desired high-level, economic properties of an efficient market-maker.

This is because contract specifications tend to be low-level in nature, focusing on contract interface, storage, and functional descriptions of entrypoint functions. High-level, cryptoeconomic properties are assumed to emerge from the specification, but they are difficult to formally justify. For some examples of such properties, properly incentivizing liquidity providers with fees, without disrupting other cryptoeconomic features of the AMM, is a highly complex topic [15, 76, 84, 85, 100], and not at all obvious to be correct from a typical contract specification. Indeed, assurances of desirable cryptoeconomic behavior for AMMs using equation 4.1 as the bonding curve were largely provided *after* the original Uniswap contracts were specified and deployed, *e.g.* in [14].

We can see that smart contract developers ubiquitously use an informal translation process, from high-level cryptoeconomic or business logic to a technical specification, to specify their smart contracts. In contrast to Uniswap, a resounding success, many instances of contract specification result in catastrophic losses due to incorrect design. Examples include Beanstalk [72], Mango Markets [127, 212], the Spartan Protocol [48, 110], Pancake Bunny [47, 97, 107, 182], and a seemingly countless stream of others [40, 245].

Frustratingly, aside from the benefits of producing a formal specification, formal methods are of limited use to resolve these vulnerabilities because they are not vulnerabilities of incorrect code, but of incorrect specification. Since formal methods are an important avenue toward high-assurance software, and are of particular relevance to smart contracts due to contract immutability [38, 217], we are in need of a paradigm shift in how we specify and verify smart contracts in order to adequately address vulnerabilities due to incorrect specification.

Our goal in this and future work is to develop rigorous tools for reasoning about the correctness of smart contract specifications in an ITP-based formal setting. In this paper, we will focus on this problem as it relates to a contract's cryptoeconomic properties. We call a contract specification *correct* if any contract satisfying that specification also exhibits the associated and desired cryptoeconomic properties. The framework that we put forward is one of contract *axiomatization* and *metaspecification.*

4.4 Contract Axiomatization and Metaspecification

The essential idea of contract axiomatization and metaspecification is to specify a contract's essential features in its specification (a contract axiomatization) and then to formally study the implications, cryptoeconomic or otherwise, of that specification via the metaspecification. This isolates the minimal conditions that must be true of a contract from the properties and behaviors that necessarily follow, emulating standard mathematical reasoning. Importantly, this allows us to minimize the size of a contract specification, and thus the burden of formally verifying any particular implementation, while improving our understanding of that contract's cryptoeconomic behavior. Formal specifications remain low-level and technical in nature, but through the metaspecification we are able to express and reason about the high-level, cryptoeconomic properties of the specification.

4.4.1 Contract Axiomatization

An effective specification abstracts the essential pieces of a contract's design and interface. It should be *consistent* (unambiguous) and *complete* (fully descriptive of contract behavior) [55, 230]. In particular, we should be able to deduce the outputs of any contract call by the specification, given the inputs. If it is well-defined in an ITP, a formal specification should be able to be stated as a predicate on smart contracts. In ConCert, the contract type is parameterized by a contract's setup, message, state, and error types, and a specification `S` then has the following form:

```
S : forall (C : Contract Setup Msg State Error), Prop.
```

The art of specification holds a tension between saying enough, so that implementers do not choose unacceptable implementations, and not saying too much, which can limit the design freedom of the implementer [230]. From the perspective of formal methods, there is further pressure to make the specification as concise as possible, since formal verification is difficult and costly due to the time and expertise required [232].

For ITP-based verification, we can see right away that a specification, a list of propositions we might hope to prove about a particular implementation, mimics the practice of axiomatization in mathematical theories [123]. An example of axiomatization in mathematics, a *group* is defined by a set of axioms: it is a set, with an associative operation, an identity element, and inverses [241]. Given a set with an operation, one can prove or disprove whether or not that set conforms to the axioms of a group by proving the operation to be associative, demonstrating inverses, and producing the identity.

We can make an analogy, where the axioms defining a group are the analogue to a specification, and any particular group is analogous to a specification-compliant implementation. Indeed, in ITP-based verification, these are in actuality the same practice: a specification is a list of propositions (axioms), and an implementation is a well-defined mathematical object which may or may not satisfy those propositions.

We might, then, resolve the tension of specification in an ITP-based formal setting as mathematicians do: study, refine, and minimize the required axioms (specification) by proving theorems about the axioms and studying their formal implications. For this, we have the metaspecification.

4.4.2 Metaspecification

Given a specification, its *metaspecification* is a set of properties either of the specification itself or of the implications of that specification. The goal of a metaspecification is to clearly define what it means for a specification to be complete, thereby giving us a formal way to demonstrate that the formal specification (axiomatization) correctly captures the intended behaviors of a specification from multiple levels of analysis.

We therefore study the implications of a specification. Given a specification `S`, stated as a predicate on contracts, we consider an arbitrary contract `C` and a proof

```
C_conforms_to_S : S(C).
```

By assuming only the witness `C_conforms_to_S` in our context, any theorems we prove apply to all contracts satisfying the specification `S`.

As we will see by example in the upcoming section, a metaspecification can include desired, high-level cryptoeconomic properties. Importantly, proving properties via the metaspecification does not add to the burden of verifying any given implementation, since by definition contracts conforming to a correct specification automatically inherit all the properties of the metaspecification.

For example, consider the standard specification of an ERC20 token contract [224, 184], which defines contract storage, interface, and functionality for a basic token contract. In addition to this standard, every token contract has an associated *tokenomics*, which includes rules governing minting, burning, token issuance or buy-backs, maximum supply, *etc* [88]. A token contract's tokenomics are essential to its correct functionality, since tokens typically attempt to capture value of some kind or regulate the functionality of some other smart contract, *e.g.* the governance tokens for a DAO [227], or the LP tokens for an AMM [234].

Within the framework of axiomatization and metaspecification, we can study a token contract specification by formalizing it as a predicate `P` on contracts and then stating and proving properties relevant to its tokenomics. The specification minimally includes specific rules governing minting and burning, including a maximum supply of tokens (if any). The metaspecification then might include some set of game-theoretic or incentive-based rules governing minting and burning hold, *e.g.* as articulated in [28], proving that the token contract conforms to some given tokenomics. Since the specification languages for ITP-based verification can state arbitrary properties, in principle we could state and attempt to prove anything we wish [198].

Indeed, we might wish to formalize a theory of DeFi and AMMs, a formal counterpart of previous work on the subject by Bartoletti *et al.* [32] and Angeris *et al.* [14]. Bartoletti *et al.* formally derive and prove desirable, high-level, economic properties of AMMs via a labelled state transition system. This work targets the so-called *arbitrage problem*, formally proving that the pricing functions of Uniswap-style AMMs respond, from an economic point of view, appropriately to market actions by rational arbitrageurs. In particular, this is a property explicitly aimed for by the earliest AMM specifications (*e.g.* [49]) for the sake of market efficiency, but to our knowledge has never actually featured in an AMM's specification.

By way of an illustrating example in the following section, our argument is that ITP-based formal methods should consider smart contracts analogously to axiomatized, mathematical objects. Returning to the mathematical analogy, in mathematics, like in formal specification, a set of axioms must be consistent, in that they do not imply a contradiction, and complete, in that they correctly characterize the intended mathematical phenomenon [186]. That the group axioms are correct is confirmed by the emergent behavior of groups, explored mathematically through the resulting theory. Importantly, the axioms of groups were carefully chosen to say enough to capture the intended mathematical structure without overspecifying—precisely the same tension exhibited in specification. To this end we proceed with an example of a formalized AMM specification and metaspecification.

4.5 Example: Formalizing Structured Pools

We illustrate the process and utility of axiomatization and metaspecification with a specific AMM contract. We have formalized[1] a structured pool contract, its specification, and its metaspecification. We also have formal proofs that the contract is correct with respect to its formal specification, and that the formal specification is correct with regards to its metaspecification. With this example, we show the AMM specification to exhibit desirable, high-level cryptoeconomic properties. Furthermore, parts of the formal specification can only be derived in reference to the metaspecification.

We will omit most of the background and details of the structured pool contract, as they are not essential to this case study. But for full, mathematical details of the specification and metaspecification of the structured pool contract, see Appendix A. For the background on the issue of fungibility for tokenized carbon credits which motivates the structured pool contract, see Appendix B.

4.5.1 The Formal Specification, or Contract Axiomatization

The structured pool specification, given in mathematically precise detail in Appendix A, is an AMM specification split in three parts: contract storage, interface, and entrypoint functions. The first two are type specifications, which we handle in Coq by way of typeclasses. The last are functional specifications,

[1]The full formalization of a structured pool contract, its specification, and metaspecification can be found at the FinCert repository: `https://github.com/dhsorens/FinCert`

which we can write using pre- and post-conditions. The formal specification can then be summarized into a predicate on an arbitrary contract C,

```
is_structured_pool :  forall C, Prop.
```

A proof of is_structured_pool indicates that the storage, interface, and entrypoint functions of C all conform to the specification.

4.5.1.1 Storage

According to the specification, contract storage must contain the following data: exchange rates for each constituent token (used for pooling and trading rates), the quantity of each token held in the pool, the address of the pool token contract, and the number of outstanding pool tokens. We can specify this by using a Coq typeclass, requiring that the storage type of a structured pool contract have functions which reveal each of these data points.

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

Listing 4.1: The formal typeclass characterizing the storage type.

4.5.1.2 Interface

The interface consists of at least three entrypoints: POOL, UNPOOL, and TRADE. These are for pooling liquidity, withdrawing (unpooling) liquidity, and trading individual carbon credits, respectively. We formalize the payload data for each entrypoint into three types: pool_data, the payload type for POOL; unpool_data, the payload type for UNPOOL; trade_data, the payload type for TRADE; other_entrypoint, an abstract type representing one, many, or no additional entrypoints.

The typeclass characterizing the interface requires that each of these types are legitimate payload types.

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

Listing 4.2: The typeclass characterizing the interface type.

Finally, we require that any incoming message be either to the `POOL`, `UNPOOL`, `TRADE`, or `other` entrypoints.

```
Definition msg_destruct contract :=
    forall (m : Msg),
    (exists p, m = pool p) ∨
    (exists u, m = unpool u) ∨
    (exists t, m = trade t) ∨
    (exists o, Some m = other o).
```

Listing 4.3: The payload of any legitimate contract call is the image of one of: `pool`, `unpool`, `trade`, or `other`.

4.5.1.3 Entrypoint Functions

Entrypoint functions are characterized with functional specifications. There are twenty-four properties of the full entrypoint specification, encoded as propositions. Some of the key properties are:

1. `pool_increases_tokens_held`, which states that a successful call to `POOL` increases the tokens pooled,
2. `unpool_decreases_tokens_held`, which states that a successful call to `UNPOOL` decreases the tokens pooled,
3. `trade_pricing_formula`, which specifies the formula used to price trades, and
4. `trade_update_rates_formula`, which specifies how exchange rates update in response to trades.

Numbers 3 and 4 listed above are parameterized by functions that calculate trades and update exchange rates, respectively `calc_delta_y` and `calc_rx'`. This is all we need to fully specify the AMM in question, but there are two ambiguities in the formal specification which can only be clarified by the metaspecification.

The first relates to how trades are priced, specified in the pricing formula of `trade_pricing_formula`. As is typical in prose specifications of AMMs that price trades along a convex curve, or indeed for any financial contract involving mathematical calculations, the structured pool specification does arithmetic in rational or real numbers. However, any implementation necessarily uses arithmetic with natural numbers which estimate rational or real numbers (typically at 6 or 9 decimal points of precision) [214]. We must decide, then, whether to estimate from above, below, or some combination of the two depending on the context. At the heart of the question is how to estimate the calculations in such a way that all the desired cryptoeconomic behaviors of the contract are satisfied. This is thus a question for the metaspecification.

The second is the functional specification of the `other` entrypoint, which is a placeholder in the specification for any entrypoints other than the three explicitly specified. The structured pool specification allows for other entrypoints, but none that fundamentally change the functionality of the contract. However, this is only an intuitive requirement, difficult to formalize. From the specification it is not obvious what functionality is and is not permitted of any other entrypoints. We must restrict the `other` entrypoint so that any additional entrypoints, whether they be to add a governance layer or something more inocuous like an entrypoint for updating metadata, do not sabotage the contract's correct cryptoeconomic behavior. Again, we can answer this within the context of the metaspecification, enabling us to give a precise functional specification of the `other` entrypoint.

4.5.2 The Formal Metaspecification

The metaspecification consists of six cryptoeconomic properties derived from previous work which elucidate desirable economic behavior of AMMs [13, 14, 32, 235]. The properties we have formalized here are those proved in the original, informal AMM specification of Appendix A, and are designed to justify the AMM specification to be cryptoeconomically correct. Informally, these are:

1. *Demand sensitivity*: in a trade, the relative price of the token traded in decreases, and that of the token being traded out increases, simulating the principle of supply and demand from classical economics.
2. *Nonpathological prices*: the price of an asset can never reach zero or go negative.
3. *Swap rate consistency*: trading cost must be nonnegative, so that it is impossible to make a sequence of calls to the `TRADE` entrypoint and output more in assets than were traded in initially.
4. *Zero-impact liquidity change*: pooling or unpooling tokens (depositing or withdrawing liquidity) must not affect trade prices.
5. *Arbitrage sensitivity*: if the price of a token differs on an external AMM from this one, a rational arbitrageur will either equalize the prices by trading, or drain the structured pool of that token.
6. *Pooled consistency*: the total value of the outstanding pool tokens is equal to the value of the pool.

Together, these properties are designed to encapsulate the intended cryptoeconomic behavior for this AMM (see Appendix A for full details). In particular, demand and arbitrage sensitivity target the desired property that the AMM facilitate an efficient (*i.e.* price-finding) market. Swap rate consistency ensures that there are no arbitrage opportunities internal to the AMM itself. Pooled consistency and nonpathological prices are the invariants of the contract state, while the rest pertain to specific entrypoint functions. To illustrate, see the formalized statements of properties 2 (nonpathological prices) and 6 (pooled consistency) in listings 4.4 and 4.5, respectively. The correspondence between Coq code and prose is illustrated in the comments.

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

Listing 4.4: The formalization of Property 2, Nonpathological Prices.

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

Listing 4.5: The formalization of Property 6, Pooled Consistency.

To our knowledge, these types of economic properties do not feature in any other contract specifications, informal or formal, but as we have pointed out they are critical to evaluating the correctness of the specification with respect to our cryptoeconomic intent. That they are formally verified to be true of the structured pool specification assures us that the design itself is correct. Importantly, any contract satisfying the functional specification of Section 4.5.1 also satisfies these economic properties without requiring any further proofs.

Furthermore, the two ambiguities in the formal specification of Section 4.5.1 can only be clarified in the context of the specification's cryptoeconomic properties. These are: verifying the pricing formulae to be correct using natural-number arithmetic, rather than rational or real numbers; and formally specifying minimal requirements on any additional entrypoints such that the economic properties of the metaspecification are not violated. We expound on both.

4.5.2.1 Rational to natural-number arithmetic

The aspects of the informal specification (see Appendix A) which require the metaspecification due to the fact that smart contracts use natural-number, rather than rational, arithmetic are these: first, how trades are priced, and second, how token exchange rates are updated by trades. Both of these implicitly use properties of rational numbers which are not true of natural numbers: that between 0 and any positive rational number r, there are infinitely many rational numbers, and that every nonzero rational number has an inverse. See in particular Figure A.1 of the structured pool specification, which specifies how trades are to be calculated.

Because any implementation necessarily uses natural numbers for arithmetic, in the formal, functional specification of the trade and exchange rate functions we must decide which properties of rational arithmetic must be preserved in our formalization into natural-number arithmetic. In the structured pool's formal specification, this resulted in seven formal properties on the abstract functions `calc_delta_y` and `calc_rx'` which are, respectively, the functions that price trades and update token exchange rates (see Listing 4.6). These include theoretical bounds on trade slippage, exchange rates, and that there be no theoretical upper bound on the output of trades. The specification allows for any pricing and rate-updating formulae which conform to those seven formal properties.

```
(* ... *)
(* specification of calc_rx', calc_delta_y *)
update_rate_stays_positive ∧
rate_decrease ∧
rates_balance ∧
rates_balance_2 ∧
trade_slippage ∧
trade_slippage_2 ∧
arbitrage_lt ∧
arbitrage_gt ∧
(* ... *)
```

Listing 4.6: An exerpt of the formal specification of a structured pool contract consisting of the required properties of `calc_rx'` and `calc_delta_y`.

Importantly, this shows that the solutions to issues such as rounding errors in calculating trades and exchange rates have solutions from within a *cryptoeconomic* context. This is particularly relevant considering recent costly attacks due to rounding error in smart contracts, *e.g.* DFX Finance [108] and KyberSwap [59].

4.5.2.2 Specifying the `other` entrypoint

The metaspecification also governs the behavior of any additional entrypoints, such as one for a governance mechanism or something more inocuous like for updating metadata. Two properties—nonpathological prices and pooled consistency—are high-level invariants of the contract, in contrast with the other properties of the metaspecification which are entrypoint-specific. In particular, they are the only invariants on contract state, so they dictate the admissible behavior of any additional entrypoint: We retain the desired cryptoeconomic behavior of our AMM so long as no additional entrypoint does not push prices to a nonpositive value, or make the total value of outstanding pool tokens unequal to the value of the pool. For example, a specification that requires that any additional entrypoints not alter rates, token balances, or outstanding pool tokens satisfies the metaspecification, though the metaspecification may allow for more varied entrypoint behavior.

```
(* ... *)
(* specification of all other entrypoints *)
other_rates_unchanged C ∧
other_balances_unchanged C ∧
other_outstanding_unchanged C ∧
(* ... *)
```

Listing 4.7: An exerpt of the formal specification of a structured pool contract consisting of the required properties of any additional, unspecified entrypoint.

4.6 (In)Correct Contract Specifications

From our example we can observe various benefits to formalizing contract specifications and metaspecifications.

1. Formally specifying the high-level properties intended by the specification gives the benefits of clarity and rigor inherent to formalization, analogous to the benefits of formalizing a specification

on an implementation.

2. The metaspecification can inform, and evolve with, the specification, just as the specification does with an implementation.

3. Choices inevitably made when formalizing a specification can be proved correct with reference to a metaspecification.

4. Once formalized, the metaspecification adds to the security guarantees of the formal specification without increasing the burden of formally verifying any particular implementation, since an implementation proved correct with regards to the specification inherits the properties of the metaspecification without requiring additional proof.

In particular, the metaspecification achieves our goal to develop rigorous tools for reasoning about the correctness of smart contract specifications in an ITP-based formal setting: It forces us to formalize the contract specification as a standalone mathematical object, and then to clearly and formally articulate the intended properties of the specification and contract design.

This example also gives us an initial evaluation metric on the efficacy of a metaspecification to prevent attacks on poor cryptoeconomic design. As we mentioned before, any smart contract facilitating trades must inevitably round when pricing trades. Correct rounding is actually a hard problem and has lead to many vulnerabilities in smart contract design. The industry rule of thumb is to round in favor of the smart contract, but even that breaks sometimes and can be a source of catastrophic loss. In our example, the metaspecification fully clarified which way to round when implementing the pricing function. Indeed, the answer to this engineering question is inevitably rooted in the desired cryptoeconomic behavior.

Even so, any genuine evaluation on the efficacy of a metaspecification to prevent attacks on poor cryptoeconomic design will depend on the sophistication with which we are able to state and verify cryptoeconomic properties, which leads us to current limitations and future work.

4.6.1 Relationship to Refinement Types

The work of this chapter is reminiscent of the practice in type theory of using refinement types, often for ease of formally reasoning about code or for more accurate static analysis [39]. Refinement types have been developed in various languages, *e.g.* ML [83] and Haskell [223], with the goal of improving the language's type system with more precision, enabling function definitions avoid the need to deal with tedious error cases with type definitions instead of function definitions. They can also allow for proof reuse [56], something we will see more of in Chapters 5 and 6.

Refinement types are simply types with an inclusion predicate. For example, the type of natural numbers $\mathbb{N}$ can be refined to the type of even natural numbers with a predicate $\lambda n.((n \bmod 2) == 0)$; and with refinement can define a partial function on $\mathbb{N}$ restricted to even natural numbers. Comparing refinement types to our work here, an implementation can be seen as a refinement of the specification, and the specification as a refinement of the metaspecification. That is, at each level we define a class of possible inhabitants and we narrow that class as we go from metaspecification all the way down to extractible implementation.

While the goals of this chapter resemble those of refinement types—enabling more effective formal reasoning—the analogy seemingly ends there. Refinement types are a method of encoding specification in the type system of a language, enforcing function preconditions at the type level and allowing for the definition of partial functions. Refinements add precision to code. In the case of a metaspecification, the direction of logical travel is reversed: we start more specialized and move to a more general setting with a metaspecification in order to formalize the logical context of the specification and ensure that it is correct.

4.7 Limitations and Future Work

The example given here is preliminary and illustrative. In order to more fully realize these benefits we should lay stronger foundations from which to derive desirable cryptoeconomic properties of smart contracts. We might also consider similar work in other formal settings.

4.7.1 Formal Theories of DeFi and AMMs

We mentioned before that substantial work has already been done to develop theories of DeFi and AMMs. The metaspecification of this paper was informed by the work of Angeris *et al.* [13, 14], Bartoletti *et al.* [32], and Xu *et al.* [235] to characterize the desirable cryptoeconomic properties of AMMs which price trades along a convex curve. However, rather than rigorously deriving them from within a theory if DeFi and AMMs embedded into ConCert, we formalized the statements of the metaspecification ourselves. The process of metaspecification could be made more rigorous if we had a formalized theory of DeFi and AMMs from which to derive our desired cryptoeconomic properties.

The cited studies are not the only attempts to systematically study the cryptoeconomic behavior of blockchains and smart contracts. There is a growing literature on cryptoeconomics more generally, *e.g.* [117, 118, 225]. We are hopeful that the growing literature will provide strong, theoretical foundations of cryptoeconomics which can be applied to specification design and verification.

To aid in the rigorous formation of contract metaspecifications, we hope to start from first principles and develop a Coq-native cryptoeconomic theory in ConCert, which fomralizes token and AMM primitives as abstract specifications [32], and operations for owning, transferring, and trading resources [46]. This is doable in ConCert because it models the semantics of a blockchain embedded in Coq, and so arbitrary theories can be constructed, including those that reason about the cryptoeconomic incentives relating to the blockchain itself. From such a theory we could make a formal study of cryptoeconomics, and provide strong foundations for contract metaspecifications.

4.8 Conclusion

Poorly specified smart contracts are vulnerable to attacks on faulty design for which formal methods typically have no answer. We are in need of a paradigm shift in how we specify and verify contracts so that we can rigorously consider a contract specification's correctness.

We propose a framework for formal specification in interactive theorem provers consisting of contract axiomatization and metaspecification. This framework treats contracts as well-defined mathematical objects, and contract specifications as the axiomatization of a mathematical theory. Our aim was to increase the expressiveness and rigor of ITP-based formal methods, enabling the expression and verification of meta properties.

We illustrated with an example, formal specification of an AMM. Not only were we able to describe high-level, cryptoeconomic properties that target market efficiency and arbitrage, but we showed that a metaspecification can shed light on choices made in the formalization of the specification and justify their correctness.

We hope that this work leads to a more rigorous and formal understanding of the cryptoeconomic properties of smart contracts, which in turn can help us mitigate the near-ubiquitous cryptoeconomic vulnerabilities in contract design.

Chapter 5

Contract Morphisms

In this chapter we address the second challenge to contract specification, detailed in Section 1.2.2.

5.1 Introduction

Faulty upgrades are a meaningful source of smart contract vulnerabilities. Costly attacks such as those on Uranium Finance (2021) [81], NowSwap (2021) [36], and Nomad (2022) [74, 109], totaling 241 million USD in lost assets, are a few of many examples of contracts attacked after an erroneous upgrade. Furthermore, because verifying software is time, labor, and resource intensive, it can be difficult to justify formally verifying software which may be upgraded quickly or frequently—a problem shared with other verified software, *e.g.* [195, 233]. Both of these factors limit the effectiveness of formal methods to address security issues in real-world software, inhibiting verification as business and security propositions [207].

What is needed is a practical and formal framework through which to specify and verify contract upgrades. As it stands we have no such framework apart from repeating the formal specification and verification process on a new contract version. Not only are upgrades costly from a verification perspective, as we have no good way of reusing much of the verification work on previous contract versions, but incorrect specifications are themselves a meaningful source of contract vulnerabilities (see Chapter 4). Thus each time a specification is made from scratch we risk introducing errors of incorrect specification.

To mitigate these issues we introduce a formal framework for specifying and verifying contract upgrades, through which we can reuse formal specification and proof on previous contract versions. This framework relies on the notion of a *contract morphism*, a theoretical tool we introduce that formally encodes structural relationships between smart contracts, and with which we can specify and reason about the structure and behavior of an upgraded contract relative to its previous versions. We argue that this is a natural framework for specifying and verifying contract upgrades, one which could decrease the cost of formally verifying contract upgrades as well as the risk of introducing vulnerabilities due to incorrect specification.

We proceed as follows. In Section 5.2, we survey related work. In Section 5.3, we introduce contract morphisms as a formal tool to specify and verify contract upgrades. In Section 5.4 we give two examples of formally specifying a contract upgrade with contract morphisms. In Section 5.5 we discuss formal verification with contract morphisms. We conclude in Section 5.6.

5.2 Related Work

In the realm of smart contracts there is limited formal work on formal reasoning about contract upgrades. Previous work [22, 66] proposes paradigm-shifting methods to either attach formal proofs to smart contracts and their upgrades, which are verified by the chain, or to trust a canonical third party to verify all contract upgrades before deployment. Unfortunately this work is likely impractical, as both solutions require substantial paradigm shifts or re-engineering of blockchain ecosystems. The latter also arguably contradicts the permissionless ethos of blockchain ecosystems by mandating a trusted third party.

In the context of software more generally, much work has gone into ensuring that software upgrades are carried out safely with formal methods [113, 133, 233]. Recent work has begun to address the issue of adapting formal proofs in a proof assistant to changes in software in order to lower the cost of formally verified software which may undergo regular upgrades [195]. This problem is complicated by the computable nature of proofs in proof assistants like Coq; chosen data types strongly influence the structure of proofs, making adaptation difficult [126]. A notable contribution to this work is Ringer *et al.*'s work on *proof repair* [194, 196], which seeks to relate a new program version to the old—by type equivalences or by comparing inductive structures—and thereby reuse previously-completed proofs on the updated code.

Drawing on this previous work, particularly Ringer *et al.*'s idea of reusing formal proofs by way of structural similarities between programs, our goal is to provide a framework for using formal methods to formally specify and verify smart contract upgrades. Contract morphisms (Section 5.3) will be our primary theoretical tool for specifying and verifying contract upgrades. Their purpose is to formally encode a structural relationship between smart contracts which can be used for both formal specification and proof reuse. With contract morphisms we address the problem of formal reasoning about contract upgrades, but in contrast to previous work on the subject our proposed framework does not require the paradigm-shifting reengineering of blockchain systems in order to be used.

Finally, we note that for smart contracts there is a distinction between contract upgrades and contract *upgradeability*. Some contracts come with a predefined logic to handle upgrades and avoid hard forks, the most popular of these on Ethereum being the Diamond framework [137]. However, they are complicated contracts as their specifications include the upgradeability functionality and governance, as well as the functionality of a given version of the contract. We will only consider upgrades via hard forks in this paper, leaving the question of rigorous formal specification and verification of upgradeable contracts to future work.

5.3 Contract Morphisms

In what follows we define *contract morphisms*, a theoretical tool which codifies formal relationships between smart contracts. In later sections we use them to formally specify and verify contract upgrades. We argue that this provides our desired formal framework.

5.3.1 Morphisms of Pure Functions

Before focusing on the specific case of smart contracts, we consider the more general case of programs formalized as pure functions. Take types A, A' and B, B', and two functions $p : A \to B$ and $q : A' \to B'$. A *morphism* from p to q is a or a pair of functions f_i and f_o which form a commutative square,

$$\begin{tikzcd}
A \arrow[r, "f_i"] \arrow[d, "p"'] \arrow[dr, phantom, "\sslash"] & A' \arrow[d, "q"] \\
B \arrow[r, "f_o"] & B'
\end{tikzcd}$$

i.e. for which

$$q \circ f_i = f_o \circ p.$$

Together, we call f_i and f_o the morphism

$$f : p \to q.$$

Via f_i and f_o, the commutative square like the above maps inputs and outputs of p to inputs and outputs of q. If p and q are programs (in particular, pure functions), we can also interpret this as execution traces of p to execution traces of q, such that transforming the inputs of p into those of q with f_i, and then applying q is the same as applying p first and then transforming the outputs over f_o.

We can define composition of morphisms easily as the composition of commutative squares. That is, given functions p, q, and r, and morphisms

$$f' : p \to q \text{ and } f'' : q \to r,$$

we can define a morphism $f := f'' \circ f' : p \to r$ by the outer square of the following diagram,

$$\begin{tikzcd}
A \arrow[r, "f_i'"] \arrow[d, "p"'] \arrow[dr, phantom, "\mathbin{/\!/}"] & A' \arrow[r, "f_i''"] \arrow[d, "q"] \arrow[dr, phantom, "\mathbin{/\!/}"] & A'' \arrow[d, "r"] \\
B \arrow[r, "f_o'"] & B' \arrow[r, "f_o''"] & B''
\end{tikzcd}$$

which is commutative if each of the inner squares are commutative. Note that composition is associative, assuming the underlying functions are associative, and that we have the obvious identity morphism $f_{\text{id}} : p \to p$ given by $f_i, f_o := id$,

$$\begin{tikzcd}
A \arrow[r, "\text{id}"] \arrow[d, "p"'] \arrow[dr, phantom, "\mathbin{/\!/}"] & A \arrow[d, "p"] \\
B \arrow[r, "\text{id}"] & B
\end{tikzcd}$$

which commutes trivially. Thus given a well-defined class of functions, which in our case will be smart contracts modeled in Coq by pure functions, we have a category on those functions with morphisms given by commutative squares on those pure functions.

In the coming sections, given a morphism $f : p \to q$, we might consider the case that q is an upgraded version of p. Because f relates execution traces of q to those of p, we will see this can be used to reason formally about q in terms of p, both in specification and verification.

5.3.2 Contract Morphisms in ConCert

Consider contracts `C1` and `C2` in ConCert,

```
C1 :  Contract Setup1 Msg1 State1 Error1
C2 :  Contract Setup2 Msg2 State2 Error2.
```

We define a data type of morphisms between contracts `C1` and `C2`,

```
ContractMorphism C1 C2.
```

This data type consists firstly of four *component functions* between the contract types of `C1` and `C2`—the `Setup`, `Msg`, `State`, and `Error` types respectively.

- `setup_morph :  Setup1 -> Setup2`
- `msg_morph   :  Msg1   -> Msg2`
- `state_morph :  State1 -> State2`
- `error_morph :  Error1 -> Error2.`

```
(* functions to form a commutative square on init *)
mA_init :=
    fun (c : Chain) (ctx : ContractCallContext) (s : Setup) ⇒
    (c, ctx, setup_morph s).
mB_init := fun (res : result State Error) ⇒
    match res with
    | Ok init_st ⇒ Ok (state_morph init_st)
    | Err e ⇒ Err (error_morph e)
    end.

(* functions to form a commutative square on receive *)
mA_recv := fun (c : Chain) (ctx : ContractCallContext)
    (st : State) (op_msg : option Msg) ⇒
    (c, ctx, state_morph st, option_map msg_morph op_msg).
mB_recv := fun (res : result (State * list ActionBody) Error) ⇒
    match res with
    | Ok (init_st, nacts) ⇒ Ok (state_morph init_st, nacts)
    | Err e ⇒ Err (error_morph e)
    end.
```

Listing 5.1: We use `f_init : init C1 -> init C2` and `f_recv : receive C1 -> receive C2` for the horizontal arrows of a pair of commutative squares, respectively, in the definition of a contract morphism.

We can use these component functions to make commutative squares like those we saw in Section 5.3.1 for each of the `init` and `receive` functions. For `init`, the horizontal arrows of the squares are given by the functions `mA_init` and `mB_init`. For `receive`, the horizontal arrows are given by the functions `mA_recv` and `mB_recv`. See Listing 5.1 for the definition of these functions in terms of the four component functions given above.

$$\begin{tikzcd}
A_{\texttt{init}} \arrow[r, "\texttt{mA\_init}"] \arrow[d, "\texttt{init}"'] \arrow[dr, phantom, "//"] & A'_{\texttt{init}} \arrow[d, "\texttt{init}'"] \\
B_{\texttt{init}} \arrow[r, "\texttt{mB\_init}"] & B'_{\texttt{init}}
\end{tikzcd}
\qquad
\begin{tikzcd}
A_{\texttt{recv}} \arrow[r, "\texttt{mA\_recv}"] \arrow[d, "\texttt{receive}"'] \arrow[dr, phantom, "//"] & A'_{\texttt{recv}} \arrow[d, "\texttt{receive}'"] \\
B_{\texttt{recv}} \arrow[r, "\texttt{mB\_recv}"] & B'_{\texttt{recv}}
\end{tikzcd}$$

The functions defined above give us squares, but to finish the definition of contract morphisms we need these squares to commute. Thus our definition includes two coherence conditions, one for the `init` square and one for the `receive` square, which are given as follows.

```
(* The coherence condition that makes the init square commute *)
init_coherence: forall c ctx s,
(match (init C1 c ctx s) with
    | Ok init_st ⇒ Ok (state_morph init_st)
    | Err e ⇒ Err (error_morph e)
    end) =
(init C2 c ctx (setup_morph s)).

(* The coherence condition that makes the receive square commute *)
recv_coherence : forall c ctx st op_msg,
Record ContractMorphism
   (C1 : Contract Setup1 Msg1 State1 Error1)
   (C2 : Contract Setup2 Msg2 State2 Error2) :=
   build_contract_morphism {
      (* the components of a morphism *)
      setup_morph : Setup1 → Setup2 ;
      msg_morph   : Msg1   → Msg2   ;
      state_morph : State1 → State2 ;
      error_morph : Error1 → Error2 ;
      (* coherence conditions *)
      init_coherence : forall c ctx s,
         result_functor state_morph error_morph (init C1 c ctx s) =
         init C2 c ctx (setup_morph s) ;
      recv_coherence : forall c ctx st op_msg,
         result_functor (fun '(st, l) ⇒ (state_morph st, l))
            error_morph
            (receive C1 c ctx st op_msg) =
         receive C2 c ctx (state_morph st)
            (option_map msg_morph op_msg) ;
}.
```

Listing 5.2: The formal definition of a contract morphism in ConCert, consisting of four component functions and two coherence conditions, which together give a pair of commutative squares.

```
11 (match (receive C1 c ctx st op_msg) with
12    | Ok (new_st, new_acts) ⇒ Ok (state_morph new_st, new_acts)
13    | Err e ⇒ Err (error_morph e)
14    end) =
15 (receive C2 c ctx (state_morph st) (option_map msg_morph op_msg)).
```

Thus a contract morphism

$$\texttt{m : ContractMorphism C1 C2}$$

is defined as a pair of commutative squares, each of which are morphisms between the respective `init` and `receive` functions of each contract. We give the formal definition of a contract morphism in Listing 5.2.

As the name morphism suggests, we should expect contract morphisms to behave like morphisms in a well-defined category. That is, we should have an associative composition operation on morphisms, and for every contract `C` should have an identity morphism

$$\texttt{id\_C : ContractMorphism C C}$$

with which composition is trivial.

Indeed, this is the case. We can compose morphisms by composing the morphism component functions. We have two results,

$$\texttt{compose\_init\_coh} \quad \text{and} \quad \texttt{compose\_recv\_coh},$$

which show that coherence of the composed morphism follows from the coherence conditions of each individual morphism. These results simply show that commutative squares compose, as we saw in Section 5.3.1, giving us a well-defined composition function `compose_cm`.

```
compose_cm : forall C1 C2 C3
    (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) : ContractMorphism C1 C3.
```

We also have a proof that composition is associative, drawing on the associativity of component functions, and we have the obvious identity morphism, given by four identity component functions, such that composition with the identity is trivial.

```
Definition id_cm (C : Contract Setup Msg State Error) :
  ContractMorphism C C := {|
    (* components *)
    setup_morph := id ;
    msg_morph   := id ;
    state_morph := id ;
    error_morph := id ;
    (* coherence conditions *)
    init_coherence := init_coherence_id C ;
    recv_coherence := recv_coherence_id C ;
  |}.
```

This gives us a well-defined category **Contracts** of smart contracts, with objects given by the `Contract` type and morphisms given by the `ContractMorphism` type.

Note that in many categories, *e.g.* the categories of sets, topological spaces, or groups, morphisms are structure-preserving functions. So too for us. The existence of a morphism

```
f :  ContractMorphism C1 C2
```

indicates a structural and mathematical relationship between contracts `C1` and `C2`, in particular relating their execution traces via the four component morphisms. As we will see, this relationship can be exploited to prove theorems about one contract in terms of another contract, something which we will do here in the case of contract upgrades and upgradeability.

In many categories there are also different classes of morphisms, including injections (embeddings, monomorphisms), surjections (quotients, epimorphisms), and isomorphisms. Injections, or embeddings, typically preserve the structure of the domain faithfully within the codomain, essentially identifying a copy of the domain within the codomain. Surjections typically represent a compression of some kind, and the information lost in the compression can frequently be described by a kernel object. As we will see, we also have injective and surjective contract morphisms, which are given when the four component functions are, respectively, injective or surjective, and which follow analogous intuitions.

5.4 Morphisms to Specify and Verify Contract Upgrades

Our goal now is to use contract morphisms as a tool to formally specify and verify contract upgrades in ConCert. Consider a contract upgrade from the perspective of a formal specification. Contracts are usually upgraded with a goal that relates the new to the previous contract version, whether it be to patch a bug, add functionality, or improve contract features. Thus the new specification relates to the old—it should eliminate a vulnerability but preserve all other functionality, be backwards compatible while adding functionality, or make improvements such as greater gas-efficiency without deviating from the behavior of the previous contract version. Of course, in practice an upgraded contract is not formally specified in relation to an older version, but rather by altering the old specification into the new, or simply starting from scratch and writing a new specification by hand. As discussed in Section 5.1, this can be a source of vulnerabilities.

In this section, we will formally specify contract upgrades in two examples using contract morphisms. The advantage of using morphisms is that we are able to clearly articulate the intent of an upgrade in the formal specification by way of a morphism in such a way that formal verification consists of producing a morphism between the updated contract implementation and a previous version which meets the required specification.

Example 5.4.1 (Swap Contract Upgrade)**.** Consider a smart contract `C1` that prices and executes trades, *e.g.* a decentralized exchange (DEX) or an automated market maker (AMM) [234]. Suppose that we wish to upgrade `C1` to a contract `C2` so that it calculates trades at higher precision by a factor of ten, meaning that the internal token balances in storage have one more decimal place, and the trade calculation is able to calculate at one decimal place greater in precision. Then in ConCert our contract `C1` will have a storage type which keeps track of internal token balances, exposed by a function `get_bal`.

```
Context { storage : Type } { get_bal : storage → N }.
```

It will also have a `TRADE` entrypoint which accepts a payload of some type, `trade_data`, characterized by an entrypoint type, `entrypoint`, and an associated typeclass, `Msg_Spec`.

```
Class Msg_Spec (T : Type) := {
   (* the trade entrypoint *)
   trade : trade_data → T ;
   (* for any other entrypoint types *)
   other : other_entrypoint → option T ;
}.

(* We assume an entrypoint conforming to Msg_Spec *)
Context { entrypoint : Type } `{ e_msg : Msg_Spec entrypoint }.
```

Listing 5.3: We assume an entrypoit type `entrypoint`, characterized by a typeclass `Msg_Spec`, which

includes a trade function trade.

Now assume that C1 has some internal function calculate_trade that it uses to calculate how many tokens will be traded out for a given contract call to the TRADE entrypoint. The trade quantity, internal token balances, and the calculate_trade function will all be accurate up to some decimal place, commonly 9 in the wild, formalized in the following specification, spec_trade, of C1.

```
(* the specification of C1's trading functionality with regards to the
    calculate_trade function *)
Definition spec_trade : Prop :=
   forall cstate chain ctx trade_data cstate' acts,
   (* for any successful call to C1's trade entrypoint, *)
   receive C1 chain ctx cstate (Some (trade trade_data)) =
   Ok(cstate', acts) →
   (* the balance in storage updates according to the
        calculate_trade function *)
   get_bal cstate' =
   get_bal cstate + calculate_trade (trade_qty trade_data).
```

Listing 5.4: The formalized proposition that C1 uses calculate_trade to price trades.

The property of Listing 5.4, spec_trade, is a specification with regards to which C1 is assumed to be correct.

Now we wish to upgrade C1 to a new contract C2 such that C2 calculates trades and keeps balances at one decimal place higher of accuracy. We will first have a specification for C2 which is analogous to spec_trade in Listing 5.4, which says that C2 uses some new function, calc_trade_precise, to calculate its trades.

```
(* The specification of C2's trading functionality with regards to the
    calculate_trade_precise function. This is analogous to spec_trade *)
Definition spec_trade_precise : Prop :=
   forall cstate chain ctx trade_data cstate' acts,
   (* for a successful call to C2's trade entrypoint, *)
   receive C2 chain ctx cstate (Some (trade trade_data)) = Ok (cstate', acts) →
   (* the balance in storage updates according to the
        calculate_trade_precise function *)
   get_bal cstate' =
   get_bal cstate +
   calculate_trade_precise (trade_qty trade_data).
```

Listing 5.5: The formalized proposition that C2 uses calculate_trade_precise to price trades.

Our goal now is to use a contract morphism to complete the formal specification of `C2` in terms of `C1`. Our specification is this: A correct implementation of the upgraded contract `C2` must satisfy `spec_trade_precise` and be accompanied by a contract morphism

```
f :  ContractMorphism C2 C1
```

with the following five properties, stated formally in Listing 5.6:

1. `msg_morph f` rounds down the precision of messages to `trade` by a factor of `10`
2. `msg_morph f` is the identity morphism on all messages aside from messages to `trade`
3. `state_morph f` rounds down on the balances kept in storage exposed by `get_bal`
4. `error_morph f` and `setup_morph f` are the respective identity functions

```
(* FORMAL SPECIFICATION:
    An upgrade C2 must admit a morphism
    f : ContractMorphism C2 C1
    with the following properties: *)

(*1. msg_morph f rounds trades down when it maps inputs of the receive function*)
Definition f_recv_input_rounds_down
   (f : ContractMorphism C2 C1) : Prop :=
   forall t', exists t,
   (msg_morph C2 C1 f) (trade t') = trade t ∧
   trade_qty t = (trade_qty t') / 10.

(* 2. msg_morph f only affects the trade entrypoint *)
Definition f_recv_input_other_equal
   (f : ContractMorphism C2 C1) : Prop :=
   forall msg o,
   (* for calls to all other entrypoints, *)
   msg = other o →
   (* f is the identity *)
   option_map (msg_morph C2 C1 f) (other o) = other o.

(* 3. state_morph f rounds down on the storage *)
Definition f_state_morph (f : ContractMorphism C2 C1) : Prop :=
   forall st, get_bal (state_morph C2 C1 f st) = (get_bal st) / 10.

(* 4. error_morph f and setup_morph f are the identity functions *)
Definition f_recv_output_err (f : ContractMorphism C2 C1) : Prop :=
   (error_morph C2 C1 f) = id.

```

```
Definition f_init_id (f : ContractMorphism C2 C1) : Prop :=
    (setup_morph C2 C1 f) = id.
```

Listing 5.6: The formal specification of the upgrade from `C1` to `C2`.

The meaning of a morphism `f` satisfying the above conditions, as a specification, is in the *coherence conditions* of `f`. We know that every possible execution trace of `C2` has a corresponding execution trace in `C1`, and we know that the input messages are identical except that `C2` accepts trades at a higher level of precision. The coherence conditions also tell us that the state of `C2` is always related to the analogous state of `C1`, expressed in the function `state_morph`. With regards to the trading functionality of our new contract `C2`, we know that the balance kept in the storage of `C2`, which is affected by trades, will always be identical to the analogous balance of `C1` after rounding down, which we can formally prove.

```
Theorem rounding_down_invariant bstate caddr
    (trace : ChainTrace empty_state bstate):
    (* Forall reachable states with contract at caddr, *)
    env_contracts bstate caddr = Some (C2 : WeakContract) →
    (* cstate is the state of the contract AND *)
    exists (cstate' cstate : storage),
    contract_state bstate caddr = Some cstate' ∧
    (* cstate is contract-reachable for C1 AND *)
    cstate_reachable C1 cstate ∧
    (* such that for cstate, the state of C1 in bstate,
        the balance in cstate is rounded-down from the
        balance of cstate' *)
    get_bal cstate = (get_bal cstate') / 10.
```

Listing 5.7: All reachable states of `C2` round down to their corresponding states in `C1`.

Most importantly, `f` guarantees a relationship between the trading functionality of `C2` and that of `C1`: `C2` emulates the exact same trading behavior as `C1` after rounding down one decimal place in precision. This means that `C2` does not introduce any novel vulnerabilities relating to trades and balances not extant to `C1`. In particular, a proof of this fact would have prevented the attacks on Uranium Finance [81], NowSwap [36], and Nomad [74].

Moving on, note that `f` of Example 5.4.1 was directed from `C2` to `C1`. The coherence conditions of `f` forced all execution traces of `C2` to conform to a pattern set by `C1`, which is precisely what lets us make the claim that we haven't introduced any new behaviors regarding trading functionality to `C2` aside from the increase in precision. Morphisms directed in the opposite direction can also be used in specification. Rather than classifying all possible execution traces of the upgrade, in this case a morphism proves that certain desired behavior exists within the contract. We illustrate with an example of specifying backwards compatibility.

Example 5.4.2 (Backwards Compatibility)**.** Consider contracts `C1` and `C2`, where `C2` is again an upgrade of `C1`, and suppose that we wish to show that `C2` is backwards compatible with `C1`. The intent of this upgrade is that the full functionality of `C1` be present within `C2`. We show this by embedding `C1` into `C2` via an injective contract morphism.

We illustate with a simple example of a counter contract `C1` which keeps some `n : N` in storage and has one entrypoint `incr` that increments the natural number in storage by `1`. `C1` is upgraded to `C2`, which in addition to an entrypoint to increment the natural number in storage also includes a `decr` entrypoint to decrement the natural number in storage by `1`.

```
Inductive entrypoint1 := | incr  (u : unit).
Inductive entrypoint2 := | incr' (u : unit) | decr (u : unit).
```

Listing 5.8: The entrypoint types of `C1` and `C2`, respectively.

We prove that `C2` is backwards compatible with `C1` by defining a contract morphism

`f : ContractMorphism C1 C2`

with the following component functions.

```
Definition msg_morph (e : entrypoint1) : entrypoint2 :=
   match e with | incr _ ⇒ incr' tt end.
Definition setup_morph : setup   → setup   := id.
Definition state_morph : storage → storage := id.
Definition error_morph : error   → error   := id.
```

These component functions do the obvious thing—send calls to the increment entrypoint of `C1` to the increment entrypoint of `C2` with the same payload, and do nothing otherwise. And `f` is an embedding since each of its component functions are manifestly injective, which we can formally prove.

```
Lemma f_is_embedding : is_inj_cm f.
```

Again, the meaning of `f` as a specification is in its coherence conditions. Any reachable state of `C1` necessarily has an analagous reachable state of `C2` which is fully structure preserving: if we were to only use the functionality of `C2` which it inherits from `C1`, we would get identical contract behavior to `C1`. We have a formal proof of this result.

```
Theorem injection_invariant bstate caddr
   (trace : ChainTrace empty_state bstate):
   env_contracts bstate caddr = Some (C1 : WeakContract) →
   (* Forall reachable states cstate of C1,
        there's a corresponding reachable state
        cstate' of C2, related by the injection *)
```

```
exists (cstate' cstate : storage),
contract_state bstate caddr = Some cstate ∧
(* cstate' is a contract-reachable state of C2 *)
cstate_reachable C2 cstate' ∧
(* .. equal to cstate *)
cstate' = cstate.
```

Listing 5.9: `C2` is backwards compatible with `C1` via the embedding `f`.

This is a toy example, but in practice specifying a new contract which is backwards compatible to the old in this strong sense may not be straightforward. Via contract embeddings, contract morphisms give us a way of formally specifying and verifying backwards compatibility.

5.5 Further Applications of Morphisms in Formal Verification

Contract morphisms establish a relationship between contracts which makes them suitable for specifying and verifying upgrades. For that same reason, contract morphisms may also have applications in proof reuse, or proof *transport*, more generally. The special case of contract *isomorphism* may also provide a stronger relationship between formal specification and proof on the associated contracts.

5.5.1 Hoare Properties and Contract Morphisms

First we consider properties that *transport* over a morphism, in particular those that we can pull back over a morphism. Hoare properties are a particularly strong example: they relate pre-conditions to post-conditions, which is relevant to morphisms because morphisms relate inputs and outputs of contract executions. As contracts are formalized in ConCert, constraints on on inputs amount to pre-conditions, and constraints on outputs amount to post-conditions. Thus for contracts `C1` and `C2` and a morphism `f : ContractMorphism C1 C2`, we might expect to be able to transport Hoare properties of one contract over `f` to the other.

Indeed, any Hoare property proved for `C2` will always have an analogous result on `C1`, mediated by `f`. We proved this in two results which relate all reachable states of `C1` to those of `C2`, and those of `C2` to those of `C1`, via the `state_morph` component of `f`. These results, `left_cm_induction` and `right_cm_induction`, are collectively called morphism induction, as they allow us to induct along the execution trace of one contract in relation to that of another. In particular, morphism induction says that properties of the state of `C2` which are invariant over `state_morph` must hold for all states of `C1`.

As a toy example of this relationship, suppose that we can prove that if a certain boolean in the storage of `C2` is set at `true`, a given entrypoint `e2` of `C2` can be successfully called, and that it fails otherwise.

Suppose further that the `msg_morph` component of `f` sends all calls to an entrypoint `e1` of `C1` to calls to `e2`, and that the `state_morph` component of `f` sends a state of `C1` with an analogous boolean set at `true` to one of `C2` with the boolean set at `false`, and visa versa. Then by morphism induction on the trace of `C1`, we get for free that calls to `e1` succeed only when the analogous boolean in the state of `C1` is set at `false`, rather than `true`. The relationship encoded by `f` between contracts `C1` and `C2` shows that `C1` and `C2` use opposing, but predictably related, logic for execution, which allows us to reuse proofs on `C2` to prove analogous results on `C1`.

5.5.2 Isomorphisms and Propositional Indistinguishability

This relationship between contracts strengthens when we have a pair of morphisms

`f : ContractMorphism C1 C2` and `g : ContractMorphism C2 C1`

such that `compose_cm g f = id_cm C1` and `compose_cm f g = id_cm C2`. This is an *isomorphism* of contracts. Isomorphisms of contracts are particularly strong; the component functions are equivalences of types and they induce a bisimulation of contracts in ConCert.

Since bisimulation is a strong and mathematically stable notion of equivalence [202], future work could investigate proof transport over contract isomorphisms, building on recent work in Coq-based formal methods. For example, we may wish to prove results on a contract optimized for formal reasoning, and transport those onto a bisimlar, performant contract, similar to the work of Cohen *et al.* [56]. This might include altering certain data types while maintaining an equivalence; chosen data types have a strong influence on the structure of proofs and can be nontrivial to transport [126, 196, 216]. We will explore this in greater depth in Chapter 6.

5.6 Conclusion

Our goal in this paper was to provide a formal framework for formally specifying and verifying smart contract upgrades in Coq. To do so we introduced the notion of a contract morphism, which encodes a formal relationship between execution traces of two contracts. We argued that this was a suitable, formal notion with which to reason about contract upgrades and provided examples of contract upgrades which can be specified and verified with contract morphisms. To our knowledge, this is the first time that the intent of an upgrade has been articulated explicitly in formal specification, and is the first formal attempt at reasoning explicitly about contract upgrades in a formal setting.

This work is intended to be a preliminary framework for reasoning about contract upgrades in Coq. As such, there are practical questions to be asked, such as whether these tools are even feasible on gas-optimized code, which can be difficult to formally reason about. Even so we are optimistic, as the previously-mentioned work by Ringer *et al.* in proof repair is practically useful and resembles our framework from a theoretical standpoint. Since the status quo is to simply update the formal specification of a previous version into the specification of the new, we hope that contract morphisms will be a strong start to efficient and rigorous verification of contract upgrades.

Chapter 6

Contract Bisimulations

In this chapter we address the third challenge to contract specification which we detailed in 1.2.3.

6.1 Introduction

The efficacy of formal verification to prevent actual, critical contract vulnerabilities depends on the feasibility of applying formal verification to deployable contract code. However, deployable code is typically optimized for performance, which typically makes it more difficult to reason about formally. Code highly optimized for performance thus risks vulnerability due to the difficulty of formal reasoning, while code written for ease of formal reasoning may not be efficient enough for the resource-scarce environment of smart contracts. Ideally, we would reason about contracts in a state optimized for formal reasoning while still deploying them in a state optimized for efficiency and gas consumption.

To do so, what is needed is a formal tool that enables us to reason about code in a format optimized for intelligibility, design and formal reasoning, whose results can be applied to a highly optimized and equivalent version of that code. As it stands no such framework exists for smart contracts. To mitigate this we introduce a formal framework of extensional equivalence between smart contracts in Coq, called *contract isomorphisms*. These equivalences will allow us to use a reference implementation as a specification of an optimized contract, as well as to port proofs between contracts that can be proved to be bisimilar.

We proceed as follows. In Section 6.2 we discuss related work. In Section 6.3 we define contract isomorphisms, our notion of formal, extensional equivalence which implements a bisimulation of contracts. In Section 6.4 we show that our definition of contract isomorphisms induces a strong form of equivalence between contracts, a *trace equivalence*. In Section 6.5, we give an example of a contract formally specified by equivalence to an extant contract and port proofs over a bisimulation. In Section 6.6 we conclude.

6.2 Related Work

Bisimulations are a core component of theoretical computer science. They primarily denote an equivalence of state transition systems [136, 203]. They are, for example, central in the study of process algebras, which rely on a notion of equivalence between processes in order to reason algebraically about the behavior of concurrent systems.

One critical role that bisimulations play is in equivalence checking [89, 99]. Equivalence checking is an approach to formal verification that consists in proving that two programs or models are related modulo some equivalence relation, or that one is included in the other modulo some preorder relation [89]. In this case, one uses a bisimulation to prove that a particular program meets its specification, where the specification is defined not in prose but as a program. Areas of formal reasoning and logic, including Hennessy-Milner logic, treat bisimulations as full equality and cannot distinguish between bisimilar processes [121, 136].

To our knowledge none of these techniques have been applied to smart contracts, but one can imagine that with a sufficient notion of contract bisimulation, we can mimic this process and use a contract formally verified and optimized for formal reasoning as a specification for a contract optimized for deployment. Proving the optimized contract correct then consists of producing a contract bisimulation.

One could also conceive of porting proofs over such an equivalence of contracts, *e.g.* in [57, 56, 216]. The strategy of porting proofs over equivalences is used in formal verification elsewhere. For example, work by Ringer *et al.* uses type equivalences to efficiently reuse proofs when updating a formally verified program [196]; work by Cohen *et al.* [57, 56] uses refinement types to optimize code in a proof-invariant way. Our work here is in a similar spirit, and may be applicable to future version of this work, but our equivalence in question is a contract bisimulation instead of a type equivalence.

Previous work has used bisimulations to encode the notion of correct implementation on a UTXO-based blockchain [101], but to our knowledge the work here is the first application of bisimulations as a tool for formally verifying optimized contracts. We build off of contract morphisms, introduced in Chapter 5. Our work here is a special use case of that theoretical tool.

6.3 Contract Isomorphisms

The fundamental contribution of this paper is a formal mechanism for proving equivalence (bisimilarity) between smart contracts in Coq, to be used in formal specification and verification. To present this mechanism, we first give a theoretical definition of contract isomorphisms as bisimulations between contracts (Section 6.3.1), moving onto the details of an implementation in Coq (Section 6.3.2).

6.3.1 Bisimilarity

Bisimilarity is a stable and natural concept that describes equivalence between processes [101, 202, 221]. A standard definition of bisimulation for labelled transition systems is as follows.

Definition 6.3.1 (Bisimulation)**.** Consider a labelled transition system $(S, \Lambda, \rightarrow)$, where S is a set of states, Λ is a set of labels, and $\rightarrow$ is a set of labelled transitions (a subset of $S \times \Lambda \times S$). A bisimulation is a binary relation $R \subseteq S \times S$ such that for every pair of states $(p, q) \in R$ and labels $\alpha, \beta \in \Lambda$,

- if $p \xrightarrow{\alpha} p'$, then there is $q \xrightarrow{\beta} q'$ such that $(p', q') \in R$
- if $q \xrightarrow{\beta} q'$, then there is $p \xrightarrow{\alpha} p'$ such that $(p', q') \in R$.

A bisimulation defines equivalence between transition systems by defining a correspondence between states that is stable under transition: given two equivalent states and a transition on the first, there is a corresponding transition such that the output states are also equivalent.

As we will see in the following section, ConCert models smart contracts as pure functions. Since we wish to capture the notion of bisimulations of contracts by defining equivalences of states that are stable under transitions, our specialized definition of bisimulation is a *natural isomorphism* of pure functions. A natural isomorphism of two pure functions defines a correspondence of function inputs and outputs that is stable under application of the function. In the following definition, we consider the category of contracts defined in Chapter 5.

Definition 6.3.2 (Natural Isomorphism of Pure Functions)**.** Consider pure functions $F : A \rightarrow B$ and $G : A' \rightarrow B'$. A natural isomorphism between F and G is a pair of isomorphisms, $\iota_A : A \cong A'$ and $\iota_B : B \cong B'$ such that the following square commutes:

$$\begin{tikzcd}
A \arrow[r, leftrightarrow, "\iota_A", "\sim"'] \arrow[d, "F"'] & A' \arrow[d, "G"] \\
B \arrow[r, leftrightarrow, "\iota_B", "\sim"'] & B'
\end{tikzcd}$$

Smart contracts modeled as pure functions get their state and entrypoint calls as inputs and as output an updated state with the resulting transactions. Because contract calls result in transitions between contract states, we can consider contracts as state transition systems. The fact that the square commutes is precisely what makes it a bisimulation in this particular interpretation of a state transition system.

$$\begin{tikzcd}
A_{\texttt{C1}} \arrow[r, leftrightarrow, "f_i"] \arrow[d, "\texttt{init}"'] & A'_{\texttt{C2}} \arrow[d, "\texttt{init}"] \\
B_{\texttt{C1}} \arrow[r, leftrightarrow, "f_o"] & B'_{\texttt{C2}}
\end{tikzcd}
\qquad
\begin{tikzcd}
A_{\texttt{C1}} \arrow[r, leftrightarrow, "f_i"] \arrow[d, "\texttt{receive}"'] & A'_{\texttt{C2}} \arrow[d, "\texttt{receive}"] \\
B_{\texttt{C1}} \arrow[r, leftrightarrow, "f_o"] & B'_{\texttt{C2}}
\end{tikzcd}$$

Figure 6.1: A bisimulation of contracts in ConCert is a natural isomorphism of each of the component functions `init` and `receive`, which inductively constructs a contract bisimulation.

6.3.2 Bisimulations in ConCert

We now move on to give details of a specific implementation of contract bisimulations in ConCert.[1] Following the theory in Section 6.3.1, we now formalize bisimulations in ConCert between a pair of contracts `C1` and `C2`,

```
C1 :  Contract Setup1 Msg1 State1 Error1
C2 :  Contract Setup2 Msg2 State2 Error2,
```

by constructing natural isomorphisms between the `init` and `receive` functions, respectively, of `C1` and `C2`. This is made by defining a correspondence of inputs and outputs to each of `init` and `receive` which is stable under contract initialization and contract calls. Constructing this equivalence consists of proving that the respective `init` functions are equivalent (the base case) and then show that each of the steps are also equivalent (the inductive step).

As we will see in Section 6.4, these natural isomorphisms induce a trace equivalence of contracts, or a bisimulation of contracts when considering them as state transition systems. This is an *extensional* equivalence of contracts.

6.3.2.1 Constructing Bisimulations via Contract Isomorphisms

We can encode these extensional equivalences in ConCert using contract morphisms from Chapter 5. Recall that morphisms compose, and there is a canonical identity morphism. These two facts give us everything we need to define an invertible pair of contract morphisms, or a contract *isomorphism*, which will take as our notion of contract bisimulation.

Definition 6.3.3 (Contract Isomorphism)**.** A contract isomorphism between contracts `C1` and `C2` is a pair of morphisms,

```
f :  ContractMorphism C1 C2
g :  ContractMorphism C2 C1,
```

such that `f` and `g` compose each way to the identity morphism.

[1]The definitions and results of this section are available at theories/ContractMorphisms.v

```
Definition is_iso_cm
  (f : ContractMorphism C1 C2) (g : ContractMorphism C2 C1) : Prop :=
  compose_cm g f = id_cm C1 ∧
  compose_cm f g = id_cm C2.
```

Listing 6.1: Contract isomorphisms are defined as a pair of morphisms that compose each way to the identity morphism under the morphism composition function `compose_cm`.

To state this as a formal proposition, we summarize this definition as a proposition in Coq in Listing 6.1.

6.4 Contract Bisimulations Induce Trace Equivalences

Our task now is to prove that contract isomorphisms actually induce the desired, strong notion of equivalence between state transition systems. In fact, they induce an isomorphism of the generated trace graphs of the contracts in question [221]. In this section we give a formal proof in Coq that a contract isomorphism produces a trace equivalence of contracts. We will first formalize a trace equivalence between contracts in ConCert in Section 6.4.1, and then show that contract isomorphisms imply trace equivalence in Section 6.4.2, showing that our definition of contract isomorphisms induces the desired strong equivalence of bisimulation.

6.4.1 Trace Equivalences in ConCert

To codify trace equivalances in ConCert, we formally define contract traces and morphisms betwen contract traces.[2] With morphisms we can formalize equivalence via *trace isomoprhisms*. Similar to contract morhpisms, contract trace morphisms are a formal, structural relationship between the traces of two contracts. As we will see, an equivalence of contract traces is the strong form of extensional equivalence that we are looking for.

6.4.1.1 Contract Traces

We begin by defining some key data types. First, a contract's trace is a chained list of contract states, connected by contract steps.

```
Definition ContractTrace (C : Contract Setup Msg State Error) :=
  ChainedList State (ContractStep C).
```

Listing 6.2: A contract's trace is a chained list of contract states, linked together by contract steps.

Contract steps are a record type of the data for a successful contract call, or a call to the `receive` function, which links two contract states. The record contains data for a successful contract call such as

[2]The definitions and results of this section are available at theories/Bisimulation.v

the contract call context, the incoming message, the resulting actions, as well as a proof that the call to `receive` succeeds.

```
Record ContractStep (C : Contract Setup Msg State Error)
     (prev_cstate : State) (next_cstate : State) := {
   (* data for a successful contract call *)
   seq_chain : Chain ;
   seq_ctx : ContractCallContext ;
   seq_msg : option Msg ;
   seq_new_acts : list ActionBody ;
   (* we can call receive successfully *)
   recv_some_step :
       receive C seq_chain seq_ctx prev_cstate seq_msg =
       Ok (next_cstate, seq_new_acts) ;
}.
```

Listing 6.3: Contract steps are successful calls to the `receive` function.

Contract traces codify the trace of contracts as state transition systems.

6.4.1.2 Contract Trace Morphism

A *contract trace morphism*, analogous to a contract morphism, encodes a formal, structural relationship between the traces of two contracts. For contracts

```
C1:Contract Setup1 Msg1 State1 Error1
C2:Contract Setup2 Msg2 State2 Error2,
```

a morphism of contract traces includes the following data:

- A function between contract state types, `ct_state_morph : State1 -> State2`.
- A proof that `ct_state_morph` sends valid initial states of `C1` to valid initial states of `C2`.
- A function `cstep_morph` that, for states `state1` and `state2` of `C1`, sends a contract step

  ```
  step1 : ContractStep C1 state1 state2,
  ```

  to a corresponding contract step of `C2` between the corresponding states

  ```
  step2 : ContractStep C2 (ct_state_morph state1) (ct_state_morph state2).
  ```

Inductively, this data gives us a relationship between all reachable states: initial states of each contract are related via the function between state types, and from there, any contract step of `C1` is related to a contract step of `C2` that respects the function on states. We codify this with a type `f : ContractTraceMorphism C1 C2`.

Example 6.4.1 (The Identity Contract Trace Morphism)**.** For any contract `C` we can define the identity morphism `id_ctm`, whose component functions are the identity and respective proofs are trivial, and which inhabits the type `ContractTraceMorphism C C`.

Example 6.4.2 (Contract Trace Morphism Composition)**.** We can define composition of contract trace morphisms similar to compision of contract morphisms, via a function `compose_ctm`, which takes morphisms

`f : ContractTraceMorphism C1 C2` and `g : ContractTraceMorphism C2 C3`

and returns a morphism

`compose_ctm g f : ContractTraceMorphism C1 C3`.

To compose contract morphisms, we simply compose their component functions. That composition is associative comes trivially. Similarly, it comes immediately that composition on either side with the identity is a trivial operation, and so composition and identity behave as we might expect in a well-defined category.

6.4.1.3 Contract Trace Isomorphisms

Contract trace isomorphisms are then defined analogously to contract isomorphisms (6.3.2.1).

Definition 6.4.1 (Contract Trace Isomorphism)**.** A contract trace isomorphism between contracts `C1` and `C2` is a pair of trace morphisms,

`f : ContractTraceMorphism C1 C2`

`g : ContractTraceMorphism C2 C1`,

such that `f` and `g` compose each way to the identity morphism `id_ctm`.

To state this as a formal proposition, we summarize this definition in a type in Coq.

```
Definition is_iso_ctm
(m1 : ContractTraceMorphism C1 C2) (m2 : ContractTraceMorphism C2 C1) :=
compose_ctm m2 m1 = id_ctm C1 ∧
compose_ctm m1 m2 = id_ctm C2.
```

Listing 6.4: Contract trace isomorphisms are defined as a pair of morphisms that compose each way to the identity morphism.

By definition, if two contracts are related by a contract trace isomorphism, then there is a one-to-one correspondence between all possible contract states; furthermore, this correspondence respects initial states. Thus extensionally, contracts which are trace isomorphic have identical behavior up to their state isomoprhisms. When considered as a labelled transition system, their execution graphs are necessarily isomorphic. The behavior of a contract is fully defined by its initial state and the steps it can take from there, and so contract trace isomorphisms give us the strong form of extensional equivalence we are looking for.

6.4.2 Contract Morphisms to Contract Trace Morphisms

The final result of this section is that contract bisimulations induce contract trace isomorphisms. We prove this result by defining a function `cm_to_ctm`, which takes a contract morphism

```
f : ContractMorphism C1 C2
```

and returns a contract trace morphism

```
cm_to_ctm f : ContractTraceMorphism C1 C2,
```

which respects identity and compositions. Contract morphisms and contract trace morphisms define a category whose objects are contracts in ConCert, so `cm_to_ctm` is a functor.

To define `cm_to_ctm` for a contract morphism `f : C1 -> C2`, we need a function between the state types of `C1` and `C2` which respects initial states and state transitions. The obvious candidate is, of course, the component function of `f` of contract states, `f.(state_morph)`, which respects initial states state transitions by the coherence conditions of its definition.

Furthermore, the identity contract morphism induces the identity contract trace morphism, and compositions of contract morphisms induce compositions of contract trace morphisms.

```
Theorem cm_to_ctm_id : cm_to_ctm (id_cm C1) = id_ctm C1.
```

Listing 6.5: Identity induces the identity.

```
Theorem cm_to_ctm_compose (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) :
    (* the image of the composition = ... *)
    cm_to_ctm (compose_cm g f) =
    (* composing the image morphisms *)
    compose_ctm (cm_to_ctm g) (cm_to_ctm f).
```

Listing 6.6: Compositions induce compositions.

Since contract isomorphisms and contract trace isomorphisms are both defined as respective morphism pairs which compose each way to the identity, a bisimulation of contracts induces a trace equivalence. We have our desired result.

6.5 Using Bisimulation as a Tool for Formal Specification

Contract isomorphisms (bisimulations) could be considered as a tool in at least two ways: first, to reuse proofs on a different contract version by porting them over the isomorphism and achieve those results on the target contract, *e.g.* as in Chapter 5; and second, to use a contract *as a specification*. To show this, in this section we give an example of a contract whose specification is another contract, *e.g.* a reference implementation, and explore the ways in which proofs transport over a contract bisimulation. This is an example where a common optimization makes a contract more difficult to reason about, and we use a contract bisimulation to formally specify the optimized contract with the intelligible contract.[3]

6.5.1 Linked Lists and Dynamic Arrays

Consider a simple contract `C_arr` that manages an array of owners, *e.g.* for access control, each identified by a natural number. It has functionality to add owners, remove owners, and swap owners. Consider also a second implementation `C_ll` that does the same, except that it stores owner IDs as a linked list instead of a dynamic array, a common contract optimization strategy over arrays in Solidity which introduces nontrivial challenges to verification [112]. The correctness criteria for the second, optimized implementation are that it behave identically to the reference implementation from an extensional standpoint, precisely because the linked list is supposed to emulate a dynamic array (though more efficiently at the bytecode level).

The two contracts `C_arr` and `C_ll` share setup and entrypoint types, but differ in their storage types and the implementation of their entrypoint functions. Both contracts must maintain a set of owners with no duplicates.

```
Inductive entrypoint :=
    | addOwner (a : N) (* to add a as an owner ID *)
    | removeOwner (a : N) (* to remove a as an owner ID *)
    | swapOwners (a_fst a_snd : N). (* to swap a_fst for a_snd as owners *)
```

Listing 6.7: The entrypoint type shared by both `C_arr` and `C_ll`.

The first contract, `C_arr`, keeps track of owners in an array in its storage type `storage_arr`.

```
Record storage_arr := { owners_arr : list N }.
```

The optimized contract `C_ll` keeps track of owners in a linked list implemented (somewhat unconventionally in Coq) via a finite mapping.

```
Record storage_ll := { owners_ll : FMap N N }.
```

[3]The contracts and bisimulations of this section are available at optimization2.v

The mapping emulates an array as follows: Using a global constant `ROOT : N`, the empty list is emulated as the mapping which points `ROOT` to `ROOT`.

```
arr_to_ll := [] ⇒ { ROOT : ROOT }.
```

From here, to insert an element `a` into the mapping, we point `ROOT` to `a`, and `a` to whatever `ROOT` used to point to (`ROOT` if `a` is the first element of the list).

```
arr_to_ll := [a] ⇒ { ROOT : a ; a : ROOT }.
```

This pattern continues such that in the mapping `ROOT` always points to the most recently-added element, and elements form a chain until the last points back to `ROOT`. So for list of the form `[a, b, c]`, the corresponding mapping points `ROOT` to `a`, and `a` to `b`, `b` to `c`, and `c` back to `ROOT`.

```
arr_to_ll := [a, b, c] ⇒ { ROOT : a ; a : b ; b : c ; c : ROOT}
```

The three entrypoints behave analogously for their respective data structures. For our array contract, `C_arr`, calling `(addOwner a)` simply appends `a` to the list of owners (provided `a` is not already an owner).

```
addOwner a := {| owners_arr := l |} ⇒ {| owners_arr := a :: l |}.
```

For our linked list contract, `C_ll`, calling `(addOwner a)` inserts the owner into the linked list.

```
addOwner a := {| owners_ll := { ROOT : a' ; ... } |} ⇒
    {| owners_ll := ROOT : a ; a : a' ; ... |}.
```

Removing an owner behaves similarly: for `C_arr`, `(removeOwner a)` removes `a` from the array,

```
removeOwner a := {| owners_arr := [ ..., b, a, b', ... ] |} ⇒
    {| owners_arr := [ ..., b, b', ... ] |}.
```

while for `C_ll`, `(removeOwner a)` updates the pointers in the mapping to excise `a`.

```
removeOwner a := {| owners_ll := { ... ; b : a ; a : b' ; ... } |} ⇒
    {| owners_ll := ... ; b : b' ; ... |}.
```

Finally, to swap owners in `C_arr`, `(swapOwners a a')` replaces `a` with `a'`,

```
swapOwners a a' := {| owners_arr := [ ..., b, a, b', ... ] |} ⇒
    {| owners_arr := [ ..., b, a', b', ... ] |}.
```

and `C_ll`, does the analogous operation by updating its pointers.

```
swapOwners a a' := {| owners_ll := { ... ; b : a ; a : b' ;  ...  } |} ⇒
    {| owners_ll := ... ; b : a' ; a' : b' ; ...  |}.
```

6.5.2 The Bisimulation

We now explore the consequences of a bisimulation, or a contract isomoprhism, between our reference implementation `C_arr` and its counterpart `C_ll`.

```
Theorem bisim_arr_ll : contracts_isomorphic C_arr C_ll.
```

A witness of the proposition `contracts_isomorphic C_arr C_ll` is a contract isomorphism between `C_arr` and `C_ll`. We first explore how a bisimulation between `C_arr` and `C_ll` lets us use code as a specification (6.5.2.1), and then explore how the specification of each ports over the bisimulation (6.5.2.2). Note that in the following example we assume some key properties about array and map operations and their properties.

6.5.2.1 Contract as a Specification

The purpose of any contract optimization is to improve the performance of the code without changing its behavior within some semantic domain. That domain is, at least in principle, the domain of a formal specification. This almost always means that changes can be made *intentionally*, affecting the inner workings of the contract, but *extensional* behavior—behavior from an outside or semantic perspective—should remain the same. In the case of our contracts `C_arr` and `C_ll`, we expect `C_ll` to behave identically to `C_arr` up to an equivalence of data structures. That precise equivalence, of expected behavior of data structures and contract entrypoints, is exactly the data held in the contract isomorphism.

To illustrate this point, we construct the contract morphism. To do so we need functions between entrypoint, state, error, and setup types. Because `C_arr` and `C_ll` differ only in their entrypoint type, these functions are the identity on all but the entrypoint type; and for the entrypoint type, these are the functions `arr_to_ll` and `ll_to_arr` specified above in Section 6.5.1.

```
(* msg, setup, and error morphisms are all identity *)
Definition msg_morph : entrypoint → entrypoint := id.
Definition setup_morph : setup → setup := id.
Definition error_morph : error → error := id.

(* storage morphisms *)
Definition state_morph : owners_arr → owners_ll := arr_to_ll.
Definition state_morph_inv : owners_ll → owners_arr := ll_to_arr.
```

Listing 6.8: The component functions of morphisms between `C_arr` and `C_ll`.

With these component functions we can prove the corresponding coherence conditions, and we get morphisms:

`f : ContractMorphism C_arr C_ll` and `f_inv : ContractMorphism C_ll C_arr`

The key point of data held in this pair of functions, which form a bisimulation, is in the way that they codify the relationship in functionality between storage and entrypoints in each contract. This is precisely the data of the argument we made in Section 6.5.1 that `C_ll` was indeed an alternative representation of `C_arr`.

Consider in particular the behavior of calling `(addOwner a)`. We know from Section 6.5.1 that in `C_arr` this appends `a` to the list of owners, while in `C_ll` this inserts `a` into the implemented linked list. We have a formal proof of this correspondence in the following two lemmas. The functions `add_owner_arr` and `add_owner_ll` are, respectively, the functions that implement the `addOwner` entrypoint in each of `C_arr` and `C_ll`.

```
Lemma add_owner_coh : forall a st st' acts,
    add_owner_arr a st = Ok (st', acts) →
    add_owner_ll a (state_morph st) = Ok (state_morph st', acts).
```

```
Lemma add_owner_coh' : forall a st e,
    add_owner_arr a st = Err e →
    add_owner_ll a (state_morph st) = Err e.
```

Listing 6.9: Two coherence results which show the correspondence of the `addOwner` entrypoint between `C_arr` and `C_ll`.

These are coherence results à la Listing 5.3.2: adding `a` to the state of `C_arr` and then transforming the state to a linked list is the same as transforming the state to a linked list first and then adding `a` to the state of `C_ll`, and vice versa. They constitute a formal proof that the behavior of the two contracts is the same up to the equivalence of their data structures for the `addOwner` entrypoint.

We have analogous proofs for each of the remaining two entrypoints of `C_arr` and `C_ll`. That they give us a bisimulation of contracts tells us that the behavior of the two contracts is the same up to the equivalence of their data structures for each entrypoint—and the equivalence of their data structures is precisely a formal description of how arrays are emulated as linked lists in the state of `C_ll`. How could you possibly be more precise in formally specifying `C_ll` as an optimization of `C_arr` than by a formal proof like this that the two contracts are extensionally equivalent?

6.5.2.2 Porting Properties Over the Bisimulation

Standard practice for comparing an optimized contract to its reference implementation would be to apply the same test suites or formal specification to the new contract and ensure that it passes all tests and still conforms to the formal specification. If the formal specification includes details of the inner workings of the contract, then relevant alterations are made to the formal specification to accommodate the new setting. This is a translation effort, which can be prone to mistranslation and resulting errrors by underspecification, so instead we would rather see if we can port previously-proved results over a bisimulation.

Indeed, we can and we will do so here with a key property for both contracts: that there be no duplicate owner IDs in storage. This property is important not only because of the intended contract functionality of `C_arr`, but also in the optimization of `C_arr` into `C_ll`. Due to the implementation of `C_ll` as a linked list via a mapping, being able to add a "duplicate" would actually compromise the integrity of the linked list as a model of an array: the mapping only allows for an owner ID to point to one other owner, so adding a "duplicate" would mean altering the pointers and unlinking the data structure. That there not be duplicates is thus an important property both from the perspective of high-level functionality (with respect to contract permissions and control flow) as well as from the perspective of low-level implementation correctness (linked list implementation emulating an array).

We first formally verify the reference implementation, `C_arr`, by proving that all reachable contract states are free of duplicates, codified in the following result.

```
Theorem no_dup_arr (cstate : owners_arr) :
   cstate_reachable C_arr cstate → no_duplicates_arr cstate.
```

Listing 6.10: All reachable states of `C_arr` have no duplicate owners in storage.

Using the bisimulation, we can now prove the analogous result about `C_ll` using `morphism_induction`, a proof technique that leverages contract morphisms to compare the reachable states of contracts related by contract morphisms.

Lemma 1 (Morphism Induction)**.** *Consider contracts* `C1` *and* `C2` *and a contract morphism*

```
f : ContractMorphism C1 C2.
```

Then every reachable state `cstate_1` *of* `C1` *corresponds to a reachable state* `cstate_2` *of* `C2`*, related by the state morphism component of* `f` *such that*

```
cstate_2 == f.(state_morph) cstate_1.
```

This lemma is codified as `left_cm_induction`.[4]

Because we have a bisimulation, not only do we know that the states of `C_arr` and `C_ll` are related by `state_morph` described in Section 6.5.2.1, but we know that `state_morph` has an inverse. Thus using the details of that morphism we can prove that `C_arr` has duplicates in storage if and only if `C_ll` has been unlinked, the analogous property for duplicates in a linked list. By morphism induction, then, we have the analogous result on `C_ll`.

[4]theories/ContractMorphisms.v

```
Theorem no_dup_ll (cstate : owners_ll) :
   cstate_reachable C_ll cstate → no_duplicates_ll cstate.
```

Listing 6.11: The desired result that all reachable states of `C_ll` have no duplicate owners in storage.

6.6 Conclusion

The efficacy of formal verification on smart contracts depends on being able to correctly specify and carry out the verification of optimized code. However, code optimized for performance is rarely optimized for intelligibility, which can make formally verifying optimized code difficult and costly. To remedy this, we introduced contract isomorphisms, a formal tool that establishes a structural equivalence between smart contracts, and we proved that contract isomorphisms give us full trace equivalences of contracts. We then demonstrated how contract isomorphisms can be used to formally specify and verify an optimized smart contract by proving it extensionally equivalent to its reference implementation. Our example illustrates the practical application of this framework to a common optimization technique in smart contract development. It shows how formal proofs of correctness can be ported over a bisimulation and how a bisimulation enables the use of a contract as a specification. We hope that this work paves the way for more robust and reliable smart contract verification, enabling practitioners to more easily reason about optimized contracts in terms of their more intelligible reference implementations.

Chapter 7

Conclusion

Smart contracts are challenging to specify and verify for a variety of reasons, which can compromise their security and is routinely the cause of substantial losses of funds. We identified and targeted three challenges in formal specification and verification of smart contracts which we aimed to address in this thesis. These are challenges of:

1. Reasoning about a specification's completeness (or correctness), which we address by introducing the logical framework of axiomatization and metaspecification in Chapter 4,
2. Reasoning about contract upgrades, which we addressed by introducing contract morphisms as a formal tool for specification in Chapter 5, and
3. Reasoning about optimized and performant code, which we addressed by introducing contract isomorphisms as a tool for proof and specification in Chapter 6.

In each of these cases, we have attempted to address challenges in contract specification and verification by introducing formal tools that leverage the highly mathematical setting of Coq as a proof assistant.

A major weakness of this work is that the tools presented here remain untested on industry-grade smart contracts, so while they have been advancements in the theory of smart contract verification it is yet unclear how practical it would be to use these tools in the wild. This will be the subject of forthcoming work.

The overarching goal, both of this thesis and any future work, is to make the practice of formal verification—stating propositions and supplying proofs—more effectual by adding to its mathematical maturity. Because programs are vulnerable to poor specifications as much as they are to incorrect code, doing so could make formally verified software more secure by grounding the process of formal verification deeper in mathematical theory.

References

[1] 20squares. https://20squares.xyz/. Accessed July 2024.

[2] Certora Prover. https://www.certora.com. Accessed February 2025.

[3] Dexter2 Specification (Mi-Cho-Coq). https://gitlab.com/dexter2tz/dexter2tz/-/tree/8a5792a56e0143042926c3ca8bff7d7068a541c3.

[4] Objkt.com — The largest Digital Art & Collectible marketplace on Tezos. https://www.objkt.com. Accessed March 2023.

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

[6] UN Supports Blockchain Technology for Climate Action — UNFCCC. https://unfccc.int/news/un-supports-blockchain-technology-for-climate-action. Accessed July 2023., .

[7] Editors' Note: Between August 2023 and the time of publication, Likvidity's "Origin Collection" of NFTs appear to have been removed from the company's website. Evidence of their previous existence can be found in the article, "Likvidi launches Origins, the first ever carbon credit yielding NFT collection," at the publication The Tokenizer: `https://thetokenizer.io/NFT/regenerative-finance-company-likvidi-launches-origins-the-first-ever-carbon-credit-`

[8] Gauntlet. https://www.gauntlet.xyz/. Accessed December 2023.

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

[10] Victor Allombert, Mathias Bourgoin, and Julien Tesson. Introduction to the Tezos Blockchain. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 1–10. IEEE, 2019.

[11] 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. ISBN 978-1-4503-5586-5. doi: 10.1145/3167084.

[12] Merlinda Andoni, Valentin Robu, David Flynn, Simone Abram, Dale Geach, David Jenkins, Peter McCallum, and Andrew Peacock. Blockchain technology in the energy sector: A systematic review of challenges and opportunities. *Renewable and Sustainable Energy Reviews*, 100:143–174, 2019. `https://doi.org/10.1016/j.rser.2018.10.014`.

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

[14] Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An Analysis of Uniswap markets. *Cryptoeconomic Systems*, 0(1), April 2021. ISSN 2767-4207,. doi: 10.21428/58320208.c9738e64.

[15] Guillermo Angeris, Tarun Chitra, and Alex Evans. When Does The Tail Wag The Dog? Curvature and Market Making. *Cryptoeconomic Systems*, 2(1), June 2022. ISSN 2767-4207,. doi: 10.21428/58320208.e9e6b7ce.

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

[17] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. Verifying, testing and running smart contracts in concert. 2020. URL `https://cs.au.dk/fileadmin/site_files/cs/AA_pdf/COBRA_Paper_-_Verifying__testing_and_running_smart_contracts_in_ConCert.pdf`.

[18] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: a smart contract certification framework in coq. *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, Jan 2020. doi: 10.1145/3372885.3373829. URL `http://dx.doi.org/10.1145/3372885.3373829`.

[19] 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. ISBN 978-1-4503-7097-4. doi: 10.1145/3372885.3373829.

[20] 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. ISBN 978-1-4503-8299-1. doi: 10.1145/3437992.3439934.

[21] 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. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796822000077.

[22] 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.

[23] Michael J. Ashley and Mark S. Johnson. Establishing a secure, transparent, and autonomous blockchain of custody for renewable energy credits and carbon credits. *IEEE Engineering Management Review*, 46(4):100–102, 2018.

[24] No Author. Return protocol partners with flowcarbon to offer automated token-based carbon offsetting to web3 users. (accessed August 2023) `https://www.flowcarbon.com/knowcarbon/return-protocol-partners-with-flowcarbon`.

[25] Avraham Eisenberg (@avi_eisen). Mango Markets Exploit. https://twitter.com/avi_eisen/status/1581326199682265088, October 2022. Accessed July 2023.

[26] Gbadebo Ayoade, Erick Bauman, Latifur Khan, and Kevin Hamlen. Smart contract defense through bytecode rewriting. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 384–389. IEEE, 2019.

[27] b. Dexter2 Specification. https://gitlab.com/dexter2tz/dexter2tz/-/blob/master/docs/informal-spec/dexter2-cpmm.md. Accessed July 2023.

[28] Julian Barreiro-Gomez and Hamidou Tembine. Blockchain token economics: A mean-field-type game perspective. *IEEE Access*, 7:64603–64613, 2019.

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

[30] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A minimal core calculus for solidity contracts. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2019 International Workshops, DPM 2019 and CBT 2019, Luxembourg, September 26–27, 2019, Proceedings 14*, pages 233–243. Springer, 2019.

[31] 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. ISBN 978-3-662-63958-0. doi: 10.1007/978-3-662-63958-0_20.

[32] 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. ISBN 978-3-030-78142-2. doi: 10.1007/978-3-030-78142-2_11.

[33] Massimo Bartoletti, Fabio Fioravanti, Giulia Matricardi, Roberto Pettinau, and Franco Sainas. Towards Benchmarking of Solidity Verification Tools. In Bruno Bernardo and Diego Marmsoler, editors,

5th International Workshop on Formal Methods for Blockchains (FMBC 2024), volume 118 of *Open Access Series in Informatics (OASIcs)*, pages 6:1–6:15, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-317-1. doi: 10.4230/OASIcs.FMBC.2024.6. URL `https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.FMBC.2024.6`.

[34] Seth Baruch, Jake Leraul, and Slobodan Sudaric. Scaling voluntary carbon markets through open blockchain platforms. (accessed August 2023) `https://dx.doi.org/10.2139/ssrn.4606815`.

[35] Carl Beekhuizen. Ethereum's energy usage will soon decrease by 99.95%. (accessed August 2023) `https://blog.ethereum.org/2021/05/18/country-power-no-more`.

[36] Rob Behnke. Explained: The NowSwap Protocol Hack. https://halborn.com/explained-the-nowswap-protocol-hack-september-2021/, September 2021. Accessed July 2023.

[37] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. A Survey on Blockchain Interoperability: Past, Present, and Future Trends. *ACM Comput. Surv.*, 54(8):168:1–168:41, October 2021. ISSN 0360-0300. doi: 10.1145/3471140.

[38] Chaïmaa Benabbou and Önder Gürcan. A survey of verification, validation and testing solutions for smart contracts. In *2021 Third International Conference on Blockchain Computing and Applications (BCCA)*, pages 57–64. IEEE, 2021.

[39] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(2):1–45, 2011.

[40] Beosin. Beosin — Global Web3 Security Report 2022. https://medium.com/Beosin_com/beosin-global-web3-security-report-2022-7aa2e4bb13. Accessed July 2023.

[41] 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. ISBN 978-3-030-61467-6. doi: 10.1007/978-3-030-61467-6_5.

[42] 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. ISBN 978-3-030-54994-7. doi: 10.1007/978-3-030-54994-7_28.

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

[44] Daniel M Berry. Formal methods: the very idea: Some thoughts about why they work when they work. *Science of computer Programming*, 42(1):11–27, 2002.

[45] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, 1995. doi: 10.1109/2.375178.

[46] Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J Summers. Rich specifications for ethereum smart contract verification. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[47] BscScan.com. Pancake Bunny Exploiter.
Address 0x158c244b62058330f2c328c720b072d8db2c612f, 2021.

[48] BscScan.com. Spartan Protocol Exploit.
Transaction 0xb64ae25b0d836c25d115a9368319902c972a0215bd108ae17b1b9617dfb93af8, 2021.

[49] Vitalik Buterin. 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. Accessed July 2023.

[50] Raphael Cauderlier. Dexter2 Specification (Mi-Cho-Coq). https://gitlab.com/nomadic-labs/mi-cho-coq/-/blob/dexter-verification/src/contracts_coq/dexter_spec.v. Accessed July 2023.

[51] COBRA Research Center. ConCert. https://github.com/AU-COBRA/ConCert. Accessed July 2023.

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

[53] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners? In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[54] 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.

[55] J Craig Cleaveland. Mathematical specifications. *ACM SIGPLAN Notices*, 15(12):31–42, 1980.

[56] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *International Conference on Certified Programs and Proofs*, pages 147–162. Springer, 2013.

[57] Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: proof transfer for free, with or without univalence. In *European Symposium on Programming*, pages 239–268. Springer, 2024.

[58] Consensys. COP27: Leading Technology Companies Launch 'Ethereum Climate Platform' Initiative to Address Ethereum's Former Proof of Work Carbon Emissions. (accessed August 2023) `https://consensys.io/blog/cop27-leading-technology-companies-launch-ethereum-climate-platform-initiative-to-a`

[59] Tim Copeland. Dex protocol kyberswap appears to lose $47 million in possible exploit. https://www.theblock.co/post/264432/dex-protocol-kyberswap-appears-to-lose-47-million-in-possible-exploit. Accessed December 2023.

[60] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

[61] 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.

[62] 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.

[63] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[64] 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.

[65] Andres Diaz-Valdivia and Marta Poblet. Governance of ReFi Ecosystem and the Integrity of Voluntary Carbon Markets as a Common Resource. doi: 10.2139/ssrn.4286167, November 2022.

[66] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, Vikram Saraph, and Eric Koskinen. Proof-Carrying Smart Contracts. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 325–338, Berlin, Heidelberg, 2019. Springer. ISBN 978-3-662-58820-8. doi: 10.1007/978-3-662-58820-8_22.

[67] David Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Emma Zhong. Fast and reliable formal verification of smart contracts with the move prover. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 183–200, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99524-9.

[68] 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.

[69] Gregor Dorfleitner and Diana Braun. Fintech, digitalization and blockchain: Possible applications for green finance. In *The Rise of Green Finance in Europe*, Palgrave Studies in Impact Finance, pages 207–237. Palgrave Macmillan, 2019. `https://doi.org/10.1007/978-3-030-22510-0_9`.

[70] 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. ISSN 1364-0321. doi: 10.1016/j.rser.2021.111378.

[71] Gregor Dorfleitner, Franziska Muck, and Isabel Scheckenbach. Blockchain applications for climate protection: A global empirical investigation. *Renewable and Sustainable Energy Reviews*, 149: 111378, 2021. `https://doi.org/10.1016/j.rser.2021.111378`.

[72] etherscan.io. Beanstalk Exploit.
Transaction 0xcd314668aaa9bbfebaf1a0bd2b6553d01dd58899c508d4729fa7311dc5d33ad7.

[73] etherscan.io. Beanstalk Exploit.
Transaction 0xcd314668aaa9bbfebaf1a0bd2b6553d01dd58899c508d4729fa7311dc5d33ad7, 2022.

[74] etherscan.io. Nomad Bridge Exploit.
Transaction 0xa5fe9d044e4f3e5aa5bc4c0709333cd2190cba0f4e7f16bcf73f49f83e4a5460, 2022.

[75] 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. ISBN 978-3-662-63958-0. doi: 10.1007/978-3-662-63958-0_6.

[76] Alex Evans, Guillermo Angeris, and Tarun Chitra. Optimal Fees for Geometric Mean Market Makers. In *Financial Cryptography and Data Security. FC 2021 International Workshops*, Lecture Notes in Computer Science, pages 65–79, Berlin, Heidelberg, 2021. Springer. ISBN 978-3-662-63958-0. doi: 10.1007/978-3-662-63958-0_6.

[77] Alyssa Exposito. What is regenerative finance (refi) and how can it impact nfts and web3? *Cointelegraph*. `https://cointelegraph.com/news/what-is-regenerative-finance-refi-and-how-can-it-impact-nfts-and-web3`.

[78] Beanstalk Farms. Beanstalk Governance Exploit. https://bean.money/blog/beanstalk-governance-exploit. Accessed July 2023.

[79] James H Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, 1988.

[80] Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*, pages 125–128. Springer, 2013.

[81] Uranium Finance. Uranium Finance Exploit. https://uraniumfinance.medium.com/exploit-d3a88921531c, April 2021. Accessed July 2023.

[82] Luciano Floridi. *The Blackwell guide to the philosophy of computing and information.* John Wiley & Sons, 2008.

[83] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, 1991.

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

[85] Robin Frtisch, Samuel Käser, and Roger Wattenhofer. The economics of automated market makers. In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, pages 102–110, 2022.

[86] John Fullerton. *Regenerative Capitalism: How Universal Principles And Patterns Will Shape Our New Economy.* Greenwich, CT: Capital Institute, 2015. (Available at `https://capitalinstitute.org/wp-content/uploads/2015/04/2015-Regenerative-Capitalism-4-20-15-final.pdf`).

[87] Doug Galen, Nikki Brand, Lyndsey Boucherle, Rose Davis, Natalie Do, Ben El-Baz, Isadora Kimura, Kate Wharton, and Jay Lee. Blockchain for social impact: Moving beyond the hype. *Center for Social Innovation, RippleWorks*, 2018.

[88] Jingxing Gan, Gerry Tsoukalas, and Serguei Netessine. Decentralized platforms: Governance, tokenomics, and ico design. *Management Science*, 2023.

[89] Hubert Garavel and Frédéric Lang. Equivalence checking 40 years after: A review of bisimulation tools. *A Journey from Process Algebra via Timed Automata to Model Learning: Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, pages 213–265, 2022.

[90] 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. ISBN 978-1-4503-8956-3. doi: 10.1145/3437378.3437879.

[91] Neil Ghani, Jules Hedges, Viktor Winschel, and Philipp Zahn. Compositional game theory. In *Proceedings of the 33rd annual ACM/IEEE symposium on logic in computer science*, pages 472–481, 2018.

[92] Leah V Gibbons. Regenerative—The New Sustainable? *Sustainability*, 12(13):5483, 2020. `https://doi.org/10.3390/su12135483`.

[93] Joseph A Goguen. More thoughts on specification and verification. *ACM SIGSOFT Software Engineering Notes*, 6(3):38–41, 1981.

[94] Florian Gronde. Flash loans and decentralized lending protocols: An in-depth analysis. Master's thesis, Center for Innovative Finance, University of Basel Basel, Switzerland, 2020.

[95] World Bank Group. *Blockchain and Emerging Digital Technologies for Enhancing Post-2020 Climate Markets.* (Washington, DC: World Bank), 2018. `http://hdl.handle.net/10986/29499`.

[96] 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.

[97] Samuel Haig. PancakeBunny tanks 96% following $200M flash loan exploit. https://cointelegraph.com/news/pancakebunny-tanks-96-following-200m-flash-loan-exploit, May 2021.

[98] Anthony Hall. Seven myths of formal methods. *IEEE software*, 7(5):11–19, 1990.

[99] Osman Hasan and Sofiene Tahar. Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 7162–7170. IGI global, 2015.

[100] Lioba Heimbach, Ye Wang, and Roger Wattenhofer. Behavior of Liquidity Providers in Decentralized Exchanges. arXiv:2105.13822, October 2021.

[101] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, January 1985. ISSN 0004-5411. doi: 10.1145/2455.2460.

[102] Maurice Herlihy. Atomic Cross-Chain Swaps. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 245–254, New York, NY, USA, July 2018. Association for Computing Machinery. ISBN 978-1-4503-5795-1. doi: 10.1145/3212734.3212736.

[103] Celine Herweijer, Dominic Waughray, and Sheila Warren. Building block(chain)s for a better planet. (accessed August 2023) `https://www3.weforum.org/docs/WEF_Building-Blockchains.pdf`, 2018.

[104] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications

to support testing. *ACM Comput. Surv.*, 41(2), feb 2009. ISSN 0360-0300. doi: 10.1145/1459352.1459354. URL `https://doi.org/10.1145/1459352.1459354`.

[105] 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.

[106] 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. ISBN 978-3-319-70277-3 978-3-319-70278-0. doi: 10.1007/978-3-319-70278-0_33.

[107] Igor Igamberdiev (@FrankResearcher). BUNNY Exploit Report. https://twitter.com/FrankResearcher/status/1395196961108774915, May 2021. Accessed July 2023.

[108] Immunefi. Dfx finance rounding error bugfix review. https://medium.com/immunefi/dfx-finance-rounding-error-bugfix-review-17ba5ffb4114. Accessed December 2023.

[109] Immunefi. Hack Analysis: Nomad Bridge, August 2022. https://medium.com/immunefi/hack-analysis-nomad-bridge-august-2022-5aa63d53814a, January 2023.

[110] PeckShield Inc. The Spartan Incident: Root Cause Analysis. https://peckshield-94632.medium.com/the-spartan-incident-root-cause-analysis-b14135d3415f, May 2021. Accessed July 2023.

[111] Runtime Verification Inc. Dexter2 Specification (K Framework). https://github.com/runtime verification/michelson-semantics/blob/a46be4a542e01b17a93134395c889df1468a067b/tests/proofs/dexter/dexter-spec.md. Accessed July 2023.

[112] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 756–772, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39799-8.

[113] Oussama Jebbar, Ferhat Khendek, and Maria Toeroe. Upgrade of highly available systems: Formal methods at the rescue. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 270–274. IEEE, 2017.

[114] John X. Response to verra's may 25th announcement. (accessed August 2023) `https://blog.toucan.earth/response-to-verras-announcement/`.

[115] Ela Khodai. Toucan's carbon ecosystem is coming to celo! (accessed August 2023) `https://blog.toucan.earth/toucans-carbon-ecosystem-celo/`, .

[116] Ela Khodai. Toucan regen network: Expanding liquidity for tokenized carbon credits. (accessed August 2023) `https://blog.toucan.earth/toucan-regen-network-bridging-carbon-credits/`, .

[117] Daniel Kirste, Niclas Kannengießer, Ricky Lamberty, and Ali Sunyaev. How automated market makers approach the thin market problem in cryptoeocnomic systems. *arXiv preprint arXiv:2309.12818*, 2023.

[118] Samela Kivilo. *Designing a Token Economy: Incentives, Governance and Tokenomics*. PhD thesis, 06 2023.

[119] Ralf Kneuper. Limits of formal methods. *Formal Aspects of Computing*, 9:379–394, 1997.

[120] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access*, 10:6605–6621, 2022.

[121] Kim G Larsen. Proof systems for satisfiability in Hennessy-Milner logic with recursion. *Theoretical Computer Science*, 72(2-3):265–288, 1990.

[122] 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. ISBN 978-3-030-32409-4. doi: 10.1007/978-3-030-32409-4_8.

[123] Barbara Liskov and Stephen Zilles. Specification techniques for data abstractions. In *Proceedings of the international conference on Reliable software*, pages 72–87, 1975.

[124] Kristof Lommers, Jack Kim, and Mohamed Baioumy. Market Making in NFTs. *Available at SSRN 4226987*, 2022.

[125] Anil Madhavapeddy. 4c in the gold standard working groups on digital solutions for carbon markets. (accessed August 2023) `https://4c.cst.cam.ac.uk/news/4c-gold-standard-working-groups-digital-solutions-carbon-markets`.

[126] Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In *International Workshop on Types for Proofs and Programs*, pages 181–196. Springer, 2000.

[127] 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.

[128] 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. *Natural Resources Journal*, 62(2):159–182, 2022. `https://digitalrepository.unm.edu/nrj/vol62/iss2/2`.

[129] Diego Marmsoler and Achim D Brucker. A denotational semantics of solidity in isabelle/hol. In *Software Engineering and Formal Methods: 19th International Conference, SEFM 2021, Virtual Event, December 6–10, 2021, Proceedings 19*, pages 403–422. Springer, 2021.

[130] Diego Marmsoler and Achim D Brucker. Isabelle/solidity: A deep embedding of solidity in isabelle/hol. *Archive of Formal Proofs*, 2022.

[131] Diego Marmsoler and Billy Thornton. Sscalc: A calculus for solidity smart contracts. In *International Conference on Software Engineering and Formal Methods*, pages 184–204. Springer, 2023.

[132] Julie Maupin. The g20 countries should engage with blockchain technologies to build an inclusive, transparent, and accountable digital economy for all, 2017. `https://ideas.repec.org/p/zbw/ifwedp/201748.html`.

[133] Stephen McCamant and Michael D Ernst. Predicting problems caused by component upgrades. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 287–296, 2003.

[134] Paul R. McMullin and John D. Gannon. Combining testing with formal specifications: A case study. *IEEE Transactions on Software Engineering*, (3):328–335, 1983.

[135] Micorriza Association. Nftree – offset co2 with nft certificates. (accessed August 2023) `https://nftree.org/index.php/proposal/`.

[136] Robin Milner. Communication and concurrency. *Prentice Hall International*, 13, 1989.

[137] Nick Mudge. EIP-2535: Diamonds, Multi-Facet Proxy. https://eips.ethereum.org/EIPS/eip-2535. Accessed July 2023.

[138] Dominic P Mulligan, Scott Owens, Kathryn E Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. *ACM SIGPLAN Notices*, 49(9):175–188, 2014.

[139] 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.

[140] Niki Naderi and Yifeng Tian. Leveraging blockchain technology and tokenizing green assets to fill the green finance gap. *Energy Research Letters*, 3(3):33907, 2022. `https://doi.org/10.46557/001c.33907`.

[141] Matthieu Nadini, Laura Alessandretti, Flavio Di Giacinto, Mauro Martino, Luca Maria Aiello, and Andrea Baronchelli. Mapping the NFT revolution: Market trends, trade networks, and visual features. *Sci Rep*, 11(1):20902, October 2021. ISSN 2045-2322. doi: 10.1038/s41598-021-00053-8.

[142] Nihar Neelakanti. Ecosapiens whitepaper. (accessed August 2023) https://mirror.xyz/0x22fbdE4fBB8FF152638cf8e6bB051FF0967c02D2/cyYJOgWQybssWx1NDRZwq3YqSK_ZVgdxayGTcUMG8g8.

[143] Zeinab Nehai and François Bobot. Deductive Proof of Ethereum Smart Contracts Using Why3, August 2019.

[144] PR Newswire. Major win for the climate: Voluntary market closes door to hfc-23 projects. (accessed August 2023) https://verra.org/press/major-win-climate-voluntary-market-closes-door-hfc-23-projects/.

[145] Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Formalising Decentralised Exchanges in Coq, March 2022.

[146] Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Formalising Decentralised Exchanges in Coq. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 290–302, 2023.

[147] Jakob Botsch Nielsen and Bas Spitters. Smart contract interactions in coq. In *International Symposium on Formal Methods*, pages 380–391. Springer, 2019.

[148] Jakob Botsch Nielsen and Bas Spitters. Smart contract interactions in Coq. In *International Symposium on Formal Methods*, pages 380–391. Springer, 2019.

[149] Markus Nissl, Emanuel Sallinger, Stefan Schulte, and Michael Borkowski. Towards Cross-Blockchain Smart Contracts. In *2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pages 85–94, August 2021. doi: 10.1109/DAPPS52256.2021.00015.

[150] No Author. Arbol. (accessed August 2023) https://www.arbol.io/, .

[151] No Author. Cambridge Bitcoin Electricity Consumption Index (CBECI). (accessed August 2023) https://ccaf.io/cbeci/index, .

[152] No Author. Carbovalent Documentation. (accessed August 2023) https://docs.carbovalent.com/, .

[153] No Author. Cascadia carbon. (accessed August 2023) https://cascadiacarbon.com/, .

[154] No Author. Celostrials. (accessed August 2023) https://celostrials.com/, .

[155] No Author. Climatecoin. (accessed August 2023) https://www.climatecoin.io/, .

[156] No Author. Climate collective. (accessed August 2023) https://climatecollective.org, .

[157] No Author. Crypto climate accord. (accessed August 2023) https://cryptoclimate.org/.

[158] No Author. dClimate. (accessed August 2023) https://www.dclimate.net/, .

[159] No Author. Filecoin green. (accessed August 2023) https://green.filecoin.io, .

[160] No Author. Flow3rs. (accessed August 2023) https://www.flow3rs.io/, .

[161] No Author. Flowcarbon docs. (accessed August 2023) https://docs.flowcarbon.com/, .

[162] No Author. Gold standard announces proposals to allow creation of digital tokens for carbon credits. (accessed August 2023) https://www.goldstandard.org/blog-item/gold-standard-announces-proposals-allow-creation-digital-tokens-carbon-credits, .

[163] No Author. Klimadao documentation. (accessed August 2023) https://docs.klimadao.finance/, .

[164] No Author. Unlock the value of carbon credits to secure climate finance. (accessed August 2023) https://kumo.earth/, .

[165] No Author. Official likvidity documentation. (accessed August 2023) https://docs.liquity.org/, .

[166] No Author. Mco2 token documentation. (accessed August 2023) https://mco2token.moss.earth/, .

[167] No Author. The merge. (accessed August 2023) https://ethereum.org/en/upgrades/merge/, .

[168] No Author. Metamazonia. (accessed August 2023) https://www.metamazonia.io, .

[169] No Author. Nftreehaus. (accessed August 2023) https://www.nftreehaus.com/, .

[170] No Author. Nftrees. (accessed August 2023) https://nftrees.cc/, .

[171] No Author. Nori whitepaper. (accessed August 2023) https://nori.com/whitepaper, .

[172] No Author. Open forest protocol documentation. (accessed August 2023) https://www.openforestprotocol.org/documentation, .

[173] No Author. Blockchain for a better planet. (accessed August 2023) https://saveplanetearth.io/, .

[174] No Author. Regen ledger documentation. (accessed August 2023) https://docs.regen.network/, .

[175] No Author. Rewilder foundation docs. (accessed August 2023) https://docs.rewilder.xyz/, .

[176] No Author. Toucan. (accessed August 2023) https://docs.toucan.earth/, .

[177] No Author. Verra addresses crypto instruments and tokens. (accessed August 2023) https://verra.org/verra-addresses-crypto-instruments-and-tokens/.

[178] Nori. Pilot croplands methodology, version 1.3. (accessed August 2023) `https://nori.com/resources/croplands-methodology`.

[179] Eric Nowak. Voluntary Carbon Markets. https://tinyurl.com/k8sbhemf, March 2022.

[180] Eric Nowak. Voluntary Carbon Markets. https://tinyurl.com/k8sbhemf, March 2022.

[181] 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, 2019. `https://doi.org/10.1016/j.egypro.2019.01.509`.

[182] pancakebunny.finance (@PancakeBunnyFin). BUNNY Exploit Report. https://twitter.com/PancakeBunnyFin/status/1395173389208334342, May 2021. Accessed July 2023.

[183] Arim Park and Huan Li. The effect of blockchain technology on supply chain sustainability performances. *Sustainability*, 13(4):1726, 2021.

[184] 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, 2018.

[185] Adele Parmentola, Antonella Petrillo, Ilaria Tutore, and Fabio De Felice. Is blockchain able to enhance environmental sustainability? A systematic review and research agenda from the perspective of Sustainable Development Goals (SDGs). *Business Strategy and the Environment*, 31(1):194–217, 2022.

[186] Charles Parsons. Informal axiomatization, formalization and the concept of truth. *Synthese*, pages 27–47, 1974.

[187] Macauley Peterson. Latest DeFi exploits show audits are no guarantee. https://blockworks.co/news/audits-cannot-guarantee-defi-exploits. Accessed November 2023.

[188] Siraphob Phipathananunth. Using Mutations to Analyze Formal Specifications. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2022, pages 81–83, New York, NY, USA, December 2022. Association for Computing Machinery. ISBN 978-1-4503-9901-2. doi: 10.1145/3563768.3563960.

[189] Likvidi Carbon Platform. Likvidity Origins Collection. (accessed August 2023) `https://www.likvidi.com/nfts/`.

[190] Andrei-Dragoş Popescu. Decentralized finance (defi)—the lego of finance. *Social Sciences and Education Research Review*, 7(1):321–349, 2020.

[191] Daniele Pusceddu and Massimo Bartoletti. Formalizing automated market makers in the lean 4 theorem prover. *arXiv preprint arXiv:2402.06064*, 2024.

[192] 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. ISBN 978-3-662-64322-8. doi: 10.1007/978-3-662-64322-8_1.

[193] Maria Ribeiro, Pedro Adão, and Paulo Mateus. Formal Verification of Ethereum Smart Contracts Using Isabelle/HOL. In *Logic, Language, and Security: Essays Dedicated to Andre Scedrov on the Occasion of His 65th Birthday*, pages 71–97. Springer, 2020.

[194] Talia Ringer. *Proof Repair*. University of Washington, 2021.

[195] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 115–129, 2018.

[196] Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. Proof repair across type equivalences. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 112–127, 2021.

[197] Grigore Roșu and Traian Florin Șerbănută. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[198] John Rushby. Theorem proving for verification. In *Summer School on Modeling and Verification of Parallel Processes*, pages 39–57. Springer, 2000.

[199] Sara Saberi, Mahtab Kouhizadeh, Joseph Sarkis, and Lejia Shen. Blockchain technology and its relationships to sustainable supply chain management. *International Journal of Production Research*, 57(7):2117–2135, 2019.

[200] Kanis Saengchote. Where do defi stablecoins go? a closer look at what defi composability really means. (accessed August 2023) `https://doi.org/10.2139/ssrn.3893487`.

[201] @samczsun. Nomad Tweet Thread. https://twitter.com/samczsun/status/1554252024723546112, August 2022. Accessed July 2023.

[202] Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, October 1998. ISSN 1469-8072, 0960-1295. doi: 10.1017/S0960129598002527.

[203] Davide Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.

[204] Soheil Saraji and Mike Borowczak. A blockchain-based carbon credit ecosystem, 2021. `https://doi.org/10.48550/arXiv.2107.00185`.

[205] Christophe Schinckus. The good, the bad and the ugly: An overview of the sustainability of blockchain technology. *Energy Research & Social Science*, 69:101614, 2020. ISSN 2214-6296. `https://doi.org/10.1016/j.erss.2020.101614`.

[206] 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. ISBN 978-3-030-03427-6. doi: 10.1007/978-3-030-03427-6_25.

[207] 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, 2020.

[208] 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. ISSN 0167-4048. doi: 10.1016/j.cose.2019.101654.

[209] 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. ISSN 2590-3330, 2590-3322. doi: 10.1016/j.oneear.2022.06.004.

[210] Adam Sipthorpe, Sabine Brink, Tyler Van Leeuwen, and Iain Staffell. Blockchain solutions for carbon markets are nearing maturity. *One Earth*, 5(7):779–791, 2022. `https://doi.org/10.1016/j.oneear.2022.06.004`.

[211] Solidity Team. Solidity Documentation, 2023. `https://docs.soliditylang.org/en/v0.5.16/`.

[212] solscan.io. Mango Markets Exploiter. https://solscan.io/account/CQvKSNnYtPTZfQRQ5jkHq8q2swJyRsdQLcFcj3EmKFfX.

[213] solscan.io. Mango Markets Exploiter. Address CQvKSNnYtPTZfQRQ5jkHq8q2swJyRsdQLcFcj3EmKFfX, 2022.

[214] Tianyu Sun and Wensheng Yu. A formal verification framework for security issues of blockchain smart contracts. *Electronics*, 9(2):255, 2020.

[215] John Symons and Jack K Horner. Why there is no general solution to the problem of software verification. *Foundations of Science*, 25:541–557, 2020.

[216] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for free: univalent parametricity for effective transport. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29, 2018.

[217] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)*, 54(7):1–38, 2021.

[218] 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. ISSN 0360-0300. doi: 10.1145/3464421.

[219] Toucan. Toucan Whitepaper. https://docs.toucan.earth/. Accessed March 2023.

[220] Jon Truby, Rafael Dean Brown, Andrew Dahdal, and Imad Ibrahim. Blockchain, climate damage, and death: Policy interventions to reduce the carbon emissions, mortality, and net-zero implications of non-fungible tokens and bitcoin. *Energy Research & Social Science*, 88:102499, 2022. https://doi.org/10.1016/j.erss.2022.102499.

[221] Johan Van Benthem and Jan Bergstra. Logic of transition systems. *Journal of Logic, Language and Information*, 3:247–283, 1994.

[222] Gijs van Leeuwen, Tarek AlSkaif, Madeleine Gibescu, and Wilfried van Sark. An integrated blockchain-based energy management platform with bilateral trading for microgrid communities. *Applied Energy*, 263:114613, 2020. https://doi.org/10.1016/j.apenergy.2020.114613.

[223] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 269–282, 2014.

[224] Fabian Vogelsteller and Vitalik Buterin. Erc-20 token standard. https://github.com/ethereum/ercs/blob/master/ERCS/erc-20.md. Accessed December 2023.

[225] Shermin Voshmgir, Michael Zargham, et al. Foundations of cryptoeconomic systems. *Research Institute for Cryptoeconomics, Vienna, Working Paper Series/Institute for Cryptoeconomics/Interdisciplinary Research*, 1, 2019.

[226] 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. doi: 10.1145/3457977.3460301.

[227] Shuai Wang, Wenwen Ding, Juanjuan Li, Yong Yuan, Liwei Ouyang, and Fei-Yue Wang. Decentralized autonomous organizations: Concept, model, and applications. *IEEE Transactions on Computational Social Systems*, 6(5):870–878, 2019.

[228] Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William Knottenbelt. SoK: Decentralized Finance (DeFi). In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, AFT '22, pages 30–46, New York, NY, USA, July 2023. Association for Computing Machinery. ISBN 978-1-4503-9861-9. doi: 10.1145/3558535.3559780.

[229] Bryan White, Aniket Mahanti, and Kalpdrum Passi. Characterizing the OpenSea NFT Marketplace. In *Companion Proceedings of the Web Conference 2022*, WWW '22, pages 488–496, New York, NY, USA, August 2022. Association for Computing Machinery. ISBN 978-1-4503-9130-6. doi: 10.1145/3487553.3524629.

[230] Jeannette M Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–22, 1990.

[231] Junghoon Woo, Ridah Fatima, Charles J. Kibert, Richard E. Newman, Yifeng Tian, and Ravi S. Srinivasan. Applying blockchain technology for building energy performance measurement, reporting,

and verification (mrv) and the carbon credit market: A review of the literature. *Building and Environment*, 205:108199, 2021. `https://doi.org/10.1016/j.buildenv.2021.108199`.

[232] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):1–36, 2009.

[233] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 154–165, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341271. doi: 10.1145/2854065.2854081. URL `https://doi.org/10.1145/2854065.2854081`.

[234] Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. Sok: Decentralized exchanges (dex) with automated market maker (amm) protocols. *ACM Computing Surveys*, 55(11):1–50, 2023.

[235] Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols. *ACM Comput. Surv.*, 55(11):238:1–238:50, February 2023. ISSN 0360-0300. doi: 10.1145/3570639.

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

[237] 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. ISSN 1024-123X. doi: 10.1155/2020/6191537.

[238] 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. ISSN 2169-3536. doi: 10.1109/ACCESS.2020.2969437.

[239] Joachim Zahnentferner, Dmytro Kaidalov, Jean-Frédéric Etienne, and Javier Díaz. Djed: A formally verified crypto-backed autonomous stablecoin protocol. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, 2023.

[240] Joachim Zahnentferner, Dmytro Kaidalov, Jean-Frédéric Etienne, and Javier Díaz. Djed: A formally verified crypto-backed autonomous stablecoin protocol. *IEEE ICBC 2023*, 2023.

[241] Hans J Zassenhaus. *The theory of groups*. Courier Corporation, 2013.

[242] Xiyue Zhang, Yi Li, and Meng Sun. Towards a Formally Verified EVM in Production Environment. In *Coordination Models and Languages: 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15–19, 2020, Proceedings 22*, pages 341–349. Springer, 2020.

[243] Yi Zhang, Xiaohong Chen, and Daejun Park. Formal specification of constant product (xy= k) market maker model and implementation. *White paper*, 2018.

[244] 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. ISSN 0167-739X. doi: 10.1016/j.future.2019.12.019.

[245] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. SoK: Decentralized Finance (DeFi) Attacks, April 2023.

Appendix A

Structured Pools for Tokenized Carbon Credits

A.1 Introduction

Tokenized carbon credits, which are of growing relevance to voluntary carbon markets [5, 65, 68, 209], have unique metadata and are typically tokenized as non-fungible tokens (NFTs). However, this can lead to low liquidity and high price volatility, so there is a push within the industry to make carbon credits as fungible as possible [70, 180].

As discussed at length in Appendix B, the current, standard solution is to pool carbon credits which have similar features, such as a specific vintage or crediting methodology, and to value each pooled token equally within the pool. From a valuation perspective, this discards the differences in constituent credits. Ideally, we would increase liquidity without ignoring these differences.

To that end we propose a novel pooling mechanism, called a *structured pool*, which pools carbon credits without ignoring their differences by valuing pooled tokens relative to each other and facilitating trades between them.

A.2 Related Work

Tokenized carbon credits can be pooled together in limited ways. For example, Toucan, perhaps the most prominent provider of tokenized carbon credits [209], tokenizes carbon credits from the Verra registry as NFTs on Polygon [219]. Each NFT can then be fractionalized as an ERC20 token using a TCO2 token contract. Distinct TCO2 contracts are not mutually fungible because they carry the metadata of the carbon credit they fractionalize.

To achieve mutual fungibility, Toucan launched the Base Carbon Tonne (BCT) and the Nature Carbon Tonne (NCT) pools. Any TCO2 token which satisfies the acceptance criteria of one of these pools can be pooled, one-for-one, in exchange for BCT or NCT tokens, respectively.

While these pools do increase token fungibility, they do so at the cost of valuing individual metadata, and we can go further by removing the required one-for-one exchange rate. Our strategy is to value pooled tokens relative to each other by enabling trading of the pooled non- or semi-fungible tokens.

Non- and semi-fungible tokens are typically traded via orderbook-style DEXes which implement various types of auctions, because low liquidity makes it difficult to use market-makers [141]. Two examples are OpenSea, the NFT marketplace with the most extensive user base and sales volume [229], where one can list an NFT or sell it via a timed auction; and OBJKT, a Tezos-based NFT marketplace, where one can sell an NFT via English and Dutch auctions [4].

Even so, there is novel research in market-making for non- or semi-fungible tokens. For example, Kim *et al.* propose a model for market-making in NFTs which uses option contracts to achieve fair prices [124]. Eulerbeats, an art and music platform which issues algorithmically-generated NFTs, implements a form of AMM by using mathematical properties of the NFTs to determine mint and burn prices [235]. Both of these examples are able to market-make because they can derive stable prices for the non- or semi-fungible tokens being traded.

We demonstrate a novel mechanism in market-making for non- or semi-fungible tokens which prices trades by leveraging the fact that tokenized carbon credits belong to a single family of tokens, so their granular metadata can help us value carbon credits relative to each other. Unlike previous work, we draw on the mechanics of commonly-used AMMs to price trades.

We proceed as follows: In Section A.3 we specify the structured pool contract. In Section A.4 we draw on previous work on AMMs to derive desirable properties of market-making contracts. We prove mathematically that a contract satisfying the specification of Section A.3 also satisfies these properties. In Section A.5 we discuss limitations to our approach, and then conclude in Section A.6.

A.3 Structured Pools

A structured pool contract has at least three entrypoints: `DEPOSIT`, `WITHDRAW`, and `TRADE`. The first two, `DEPOSIT` and `WITHDRAW`, are for pooling and unpooling constituent tokens, respectively, in exchange for a *pool token*. The exchange rate from a particular tokenized carbon credit to the pool token is called the *pooling exchange rate*, and is set individually for each carbon credit which can be pooled. These are also the entrypoints for, respectively, depositing and withdrawing liquidity used for trades, which are executed via the `TRADE` entrypoint. Each of these entrypoints is governed by equations which price trades and update the pooling exchange rates, which will see shortly.

The contract's storage must keep track of the family of tokens which can be pooled, each of which is called a *constituent token*, along with the pooling exchange rate of each token in the family. Each pooling exchange rate is assumed to be strictly positive when the contract is deployed. It must also keep track of the contract's balance of each constituent token, the address of the pool token contract, and the total number of outstanding pool tokens.

A brief comment on notation. We refer to our family of constituent tokens as T, where a token t_x in the family is described by its *token data*, which is a contract address and token ID pair. We will typically discuss trades from, *e.g.* t_x to t_y, where Δ_x refers to the quantity traded in, Δ_y refers to the quantity traded out, and x and y refer, respectively, to the quantity of each token held by the contract. We also write r_x as the pooling exchange rate in storage for token t_x.

A.3.1 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. It 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.

A.3.2 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.

A.3.3 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. It checks that both t_x and t_y are in the token family, that $k > 0$, and that $\Delta_x > 0$. It calculates Δ_y using formulae we will give below, and checks that it has a sufficient balance y in t_y to execute the trade action. Then in an atomic transaction, the contract updates the exchange rate r_x in response to the trade, transfers Δ_x of tokens t_x from the sender's wallet to itself, and transfers Δ_y of tokens t_y from itself to the sender's wallet. The specification is summarized in Figure A.1.

```
(* 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 *)
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) &&
    (delta_x > 0) &&
    (k > 0) &&
    (self_balance t_y >= delta_y)
    then
        <atomic>
            r_x <- UPDATE_RATE
                x delta_x delta_y r_x r_y;
            transfer (delta_x)
                of (t_x)
                from (sender)
                to (self) ;
            transfer (delta_y)
                of (t_y)
                from (self)
                to (sender) ;
        </atomic>
    else
        fail ;
```

Figure A.1: Pseudocode of the `TRADE` entrypoint function.

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

$$(x + \Delta_x)(y - \Delta_y) = k, \tag{A.1}$$

giving

$$\Delta_y = y - \frac{k}{x + \Delta_x}. \tag{A.2}$$

This is how trades are priced in the wild for liquidity pools of fungible tokens [14]. We call $p_s = \frac{\Delta_y}{\Delta_x}$ the *swap price*.

An important consequence to (A.1) is that the smaller Δ_x is compared to k, the closer the exchange happens at a rate of $p_q = \frac{y}{x}$. This is because the derivative of $xy = k$, or $f(x) = \frac{k}{x}$, is

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

and the smaller Δ_x is relative to k, the more accurately the tangent line at some (x_0, y_0) approximates

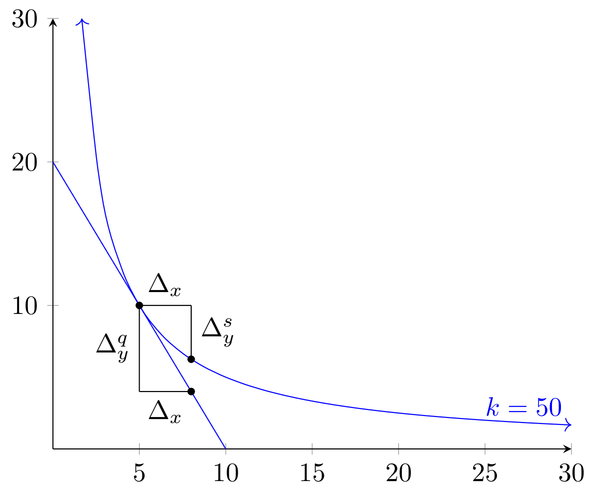

Figure A.2: A trade of $\Delta_x = 3$ for Δ_y^q and Δ_y^s, respectively, at $k = 50$. $\Delta_y^q = p_q \Delta_x$ is the trade priced at the *quoted price* p_q and $\Delta_y^s = p_s \Delta_x$ is the trade priced at the *swap price* p_s.

the convex curve $xy = k$. We call $p_q = \frac{y}{x}$ the *quoted price*.

The difference $p_q - p_s$ is called the *price slippage* [235, §3.2.4]. It is important to note that p_s is always less than p_q because p_q is calculated by moving Δ_x along the tangent line from a starting point (x_0, y_0) representing the current state of the contract's funds available for trading, and p_s is calculated by moving Δ_x along $xy = k$. Since $xy = k$ is convex, moving Δ_x along the tangent line always results in a larger Δ_y than moving along $xy = k$. See Figure A.2 for a graphical illustration, where Δ_y^q is the output of a trade priced at p_q and Δ_y^s is the output of a trade priced at p_s.

In particular, this means that

$$\Delta_y^s < p_q \Delta_x \tag{A.3}$$

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

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

$$r_{x,y} := \frac{r_x}{r_y}. \tag{A.4}$$

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

To price a trade we begin by finding ℓ such that

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

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

$$(\ell r_y + \Delta_x)(\ell r_x - \Delta_y^s) = k. \tag{A.5}$$

This formula yields the quoted price of this trade as

$$p_q = \frac{\ell r_x}{\ell r_y} = \frac{r_x}{r_y} = r_{x,y}, \tag{A.6}$$

as desired. The swap price, then, is

$$p_s = \frac{\Delta_y^s}{\Delta_x} \tag{A.7}$$

where

$$\Delta_y^s = \ell r_x - \frac{k}{\ell r_y + \Delta_x}. \tag{A.8}$$

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

After executing a trade, if we do not adjust pooling exchange rates, the pool token is now overcollateralized. We can see this because by (A.3),

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

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

To avoid this, we adjust the values of the constituent tokens so that their sum at the new valuation is equal to the total number of outstanding pool tokens. In a trade t_x to t_y, because it is possible to deplete t_y from the pool, we cannot reliably regain pooled consistency by adjusting the value of the token being traded for in the pool. We know, however, that we have a supply of t_x because that was the deposited token. Thus to regain pooled consistency, we have to slightly devalue t_x in relation to the rest of the pool tokens. To do so, we divide the quantity of pool tokens by its collateral in t_x to get an updated exchange rate r'_x as follows:

$$r'_x := \frac{r_x x + r_y \Delta_y}{x + \Delta_x}. \tag{A.9}$$

Equation (A.9) updates the pooling exchange rate of t_x so that the pool token is neither under- nor over-collateralized.

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

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

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

pool tokens with our unadjusted pooling exchange rates. To rectify this, bringing the pool back down to the value of 50 pool tokens, we slightly devalue t_x relative to the other tokens in the pool. We use the formula (A.9)

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

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

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

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

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

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

We end this section with a note that in these calculations, we implicitly assumed exchange rates r_x to be rational numbers by which we can multiply and divide freely so long as $r_x > 0$. Of course, implementations will include rounding error, and so we add to the specification that the `UPDATE_RATE` function return a positive number if the numerator and denominator of the quotient are positive. We also specify that the `CALCULATE_TRADE` function return a positive number if k, r_x, r_y, and Δ_x are positive, and that $\Delta_y < \frac{r_x}{r_y}\Delta_x$ always be true for successful trades.

A.4 Properties of Structured Pools

The structured pool contract is designed to imitate AMMs in how it prices trades and updates the pooling exchange rates. While AMMs such as Uniswap have been shown to exhibit desirable economic behaviors [14], it is not obvious that the structured pool contract will do the same. To that end, we draw on work by Angeris *et al.* [13, 14], Bartoletti *et al.* [31, 32], and Xu *et al.* [235] on AMMs and DeFi, from which we derive six properties indicative of desirable market behavior from game-theoretic and economic perspectives.

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

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

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

which holds if $r_y \Delta_y < r_x \Delta_x$. By (A.3) and (A.6):

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

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

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

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

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

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

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

rendering the numerator of r'_x,

$$r_x x + r_y \Delta_y,$$

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

$$x + \Delta_x,$$

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

Property 3 (Swap Rate Consistency)**.** *Let t_x be a token in our family with nonzero pooled liquidity and*

$r_x > 0$. *Then for any* $\Delta_x > 0$ *there is no sequence of trades, beginning and ending with* t_x*, such that* $\Delta'_x > \Delta_x$*, where* Δ'_x *is the output quantity of the sequence of trades.*

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

$$r_y \Delta_y \leq r'_x \Delta_x. \tag{A.10}$$

Since

$$r'_x = \frac{r_x x + r_y \Delta_y}{x + \Delta_x}, \tag{A.9}$$

(A.10) simplifies to

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

which in turn simplifies to

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

Since we know that $r_y \Delta_y \leq r_x \Delta_x$ from (A.3), we can see that our inequality holds for all $x \geq 0$, as desired.

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

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

by (A.3). Now consider a trading loop from t_x to t_y, and back to t_x, for $t_y \neq t_x$. We have our result if we can show

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

is satisfied, because $\frac{r_y}{r'_x} \Delta_y$ is an upper bound on the quantity that Δ_y can be traded for as $p_s < p_q$. This, of course, is given by (A.10) and the fact that $r'_x > 0$ from Property 2.

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

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

is satisfied. But we have from (A.10) 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 (Zero-Impact Liquidity Change)**.** *The quoted price of trades is unaffected by calling* `DEPOSIT` *and* `WITHDRAW`*.*

Proof. We have this result because the quoted price depends only on the pooling exchange rates, as we saw in (A.4), and as per the specification, only the TRADE entrypoint alters pooling exchange rates. □

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

Proof. Suppose the structured pool prices a constituent token t_x higher than an external market. Then an arbitrageur can buy t_x elsewhere and sell them into the structured pool. Doing so devalues t_x relative to the other tokens, as we have shown. Recall that $0 < r'_x < r_x$, so to prove our result we just need to show that 0 is the greatest lower bound of r'_x. Note that by definition, $\Delta_y = \Delta_y^s$, so substituting (A.8)

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

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

Then

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

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

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

$$\Delta'_x < \Delta_x$$

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

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

$$\sum_{t_x} r_x x = k \tag{A.11}$$

Proof. As a base case, by the specification, at the time of contract deployment $k = 0$ and we have no pooled liquidity, so (A.11) holds trivially because $x = 0$ for all t_x. For our inductive step, consider a contract state for which (A.11) holds. If we call DEPOSIT, (A.11) holds by definition because for a deposit of d_x of t_x, we mint $r_x d_x$ pool tokens. The same is true if we call WITHDRAW. Finally, if we call TRADE from tokens t_x to t_y, then there is an excess number of tokens in t_x, violating (A.11). This excess is quantified in (A.9) and remedied by adjusting r_x to r'_x as we saw before. □

A.5 Limitations

While we have proved that some properties of AMMs, shown to be indicative of desirable market behavior by the literature, hold for structured pools, our work has limitations.

Firstly, the properties in the literature which we drew on were derived for AMMs using custom, formal models of the blockchain to justify the results, but we adapted them to our use case somewhat informally and without using a formal model of the blockchain. It is not obvious that they imply the same notions of desirable market behavior for structured pools in the same way that they apply to AMMs.

In particular, Property 5, arbitrage sensitivity, is weaker than the similar properties in the literature because the pool can deplete in some constituent tokens (though not entirely through trading). This may be problematic if the value of one constituent token plummets for whatever reason, as this could cause the pool to deplete in every token but the now-devalued one. Whether this is actually an issue may depend on the specific use case and family of tokens.

Finally, while these proofs are mathematical and rely on the specification, there is always a possibility of error due to incorrect assumptions or because we do not base this in an explicit and formal model of a blockchain. These results would be more reliable if this contract specification and proofs were embedded in a formal system.

A.6 Conclusion

Our goal was to enable greater fungibility for tokenized carbon credits, improving on current methods which pool tokens at the expense of individual token metadata.

We presented structured pools, which are able to pool tokens without imposing a one-for-one valuation. Structured pools value constituent tokens relative to each other in a dynamic way by facilitating trades between constituent tokens and updating relative prices in response to trading activity.

To show that structured pools satisfy desirable properties of AMMs, we drew on previous research on AMMs and DeFi, including [13, 14, 31, 32, 235], and we derived six key properties and proved that a contract satisfying our specification also satisfies these key properties. As we pointed out in Section A.5, for any actual implementation, these properties would ideally be formalized and proved within a formal system.

We demonstrated that it is possible to pool tokens in a wide-ranging token family without imposing a one-for-one valuation on the constituent, pooled tokens. We hope that this will be useful to pool and trade tokenized carbon credits with deeper on-chain liquidity. While this targets tokenized caron credits on the blockchain, we conjecture that this pooling mechanism could be useful for other cases, including tokenized commodities more broadly.

Appendix B

Tokenized Carbon Credits

B.1 Introduction

Blockchains are well-suited for tokenizing, trading, and retiring voluntary carbon credits, as recognized by the UN [6], the World Bank [95], the World Economic Forum [103], and others [69, 71, 128, 181, 204, 210]. They offer several advantages to legacy systems, including transparency and censorship resistance, which lend themselves to robust accounting practices and which can help prevent double counting carbon credits [71, 95]. However, tokenized carbon credits are defined by a varied set of standards, both of token contracts as well as the kind of carbon credits that are acceptable [179]. This hampers interoperability and trading with high liquidity [65, 71, 179], which can lead to fragmented, inefficient, and volatile markets [65, 95, 179].

To promote interoperability between tokenized carbon credits, and to support current efforts in creating a unified token standard [58, 162], we survey the current state of tokenized carbon credits. Our goal is to clarify the technical hurdles to interoperability, so we evaluate tokenized carbon credits from a technical, rather than economic or climate, perspective. We do not make value judgments on the various methodologies which quantify carbon capture, and which back tokenized carbon credits, nor on the reputation or reliability of any particular producer of tokenized carbon credits.

We note that central to the climate debate as it relates to blockchains is the energy expenditure of large proof-of-work chains such as Bitcoin [151, 220]. This included Ethereum up until the so-called *Merge* in September 2022 [167]. Since the Merge, all the projects discussed here operate on proof-of-stake chains, which consume negligible amounts of electricity to secure the blockchain [35]. Thus here we will not treat the energy expenditure of blockchains.

B.1.1 Outline

We proceed as follows. In Section B.2 we survey related work. In Section B.3, we survey tokenized carbon credits and consider how they are tokenized, what substantiates their value, and their individual tokenomics. In Section B.4, we discuss how tokenized carbon credits are most frequently traded on blockchains. In Section B.5, we consider various on-chain applications built on tokenized carbon credits. In Section B.6, we discuss how climate data is made available on-chain, both for tokenization as well as related applications. In Section B.7, we mention related organizations and collectives which aim to promote the usage of blockchains for carbon trading, some of whom are attempting to make standards for tokenized carbon credits. In Section B.8 we conclude.

B.2 Related Work

Tokenized carbon credits are central to *Regenerative Finance* (ReFi), a subset of Decentralized Finance (DeFi), named after a broader movement in regenerative capitalism [86, 92], which is primarily concerned with climate change mitigation through digital assets of various kinds [65, 77]. ReFi consists of a wide range of applications that can be studied from many disciplines, including business [87], law [65, 128] economics [132, 140], and computer science. Tokenized carbon credits have been studied in relation to the energy industry [12], the transition into sustainable energy [140], and how to maximize the efficiency of current grids by allowing for peer-to-peer energy trading [222]. There is also research regarding solutions for measurement, reporting, and verification (MRV) [231], and sustainable supply chains [183, 199].

Further work on tokenized carbon credits includes climate market design [23, 95, 204], surveys of ReFi applications and of related organizations and initiatives [71, 87], studies of the impact and efficacy of blockchain in sustainability efforts [185, 205, 210], possible blockchain-based green fintech applications [69], and studies from the perspective of legacy voluntary carbon markets [179]. There has also been work in the grey literature on bridging voluntary carbon markets onto blockchains [34], but this approaches the issue from a less technical and more market-driven approach than what we give here.

A salient theme from previous work is that governments, agencies, and researchers share an explicit goal of interoperable tokenized carbon credits that can be traded with high liquidity. This is also an explicit, central goal for producers of tokenized carbon credits. Part of the issue is that tokenized carbon credits are expected to behave like commodities [95, 204], and therefore be fully fungible on-chain as an asset class. However, a cursory examination of the industry reveals a heterogeneity which makes it difficult to do so in practice; we will see this in detail in Section B.3.

Our contribution is an exposition of the technical details of tokenized carbon credits with the aim of supporting interoperability, mutual fungibility, and unified token standards. This work could be useful in unifying a fractured marketplace and achieving the goals of the industry. More broadly, our goal is to support efforts to build open, decentralized, and composable financial infrastructure for tokenized carbon credits which leverages advantages of decentralized finance (DeFi) [190, 200].

B.3 Tokenized Carbon Credits

Tokenized carbon credits vary in at least two ways: in what justifies their value and in the implementation details of the token contract. The first relates to interoperability from a qualitative perspective. Tokens that represent equivalent or comparable things are easier to interoperate than tokens which represent distinct or incomparable things. The second relates to interoperability from a more technical and engineering perspective. The fungibility status, metadata, entrypoints, and level of adherence to established standards of various token contracts may impact the ease or difficulty of developing applications that build on and facilitate interoperability among multiple tokenized carbon credits. Because we study tokenized carbon credits through the lens of interoperability, we examine tokenized carbon credits within these broad categories: first, in what justifies their value as a legitimate carbon credit, and second, in the implementation details of their token contracts. We elaborate on each of these categories.

First, tokenized carbon credits must be substantiated in some way as atmospheric carbon dioxide rigorously captured and stored, or something similar to it (*e.g.* avoided emissions). This requires a methodology for data collection and processing that accurately quantifies the amount of carbon stored. As with any discipline, such methodologies can vary in nature, rigor, and reliability. Some tokenized carbon credits draw on established methodologies such as those of Verra or Gold Standard [179], as they are backed by carbon credits on legacy ledgers. Others are minted natively on a blockchain using new methodologies, based on machine learning and satellite data. In either case we encounter the problem of tokenization, which is how one accurately represents some off-chain asset or data on a blockchain in such a way that transactions on the blockchain correctly govern any corresponding real-world asset or event.

More subtly, every entity that tokenizes carbon credits makes a choice on criteria for acceptable carbon credits. For those tokenized from a legacy ledger, the tokenizing party must set criteria for which carbon credits from the legacy ledger are allowed to be tokenized. The most common guards relate to vintage (*e.g.* that credits must be less than ten years old), or restrict to a certain class of carbon credit (*e.g.* restricting to nature-based credits or requiring that credits be from carbon sequestration rather than prevented deforestation). The reason for these choices can vary greatly, from a value judgment on the quality of some credits, to the specific ethos and goals of a particular team. For tokenized carbon credits which employ their own methodology to verify captured carbon and mint credits, the methodology itself makes a judgment on what constitutes rigorous carbon capture, and thus a carbon credit, and what does not. We will look at these details as they relate to what substantiates a tokenized carbon credit, though we reiterate that we are not in a position to make value judgments on one methodology over another.

Secondly, tokenized carbon credits differ in their implementation details, in particular in their token

Company	**Token**	**Fungible**	**Methodology**	**Fungibility Layer**	**Chain(s)**
Toucan	TCO2	N	Verra	Base Carbon Tonne (BCT) Nature Carbon Tonne (NCT)	Polygon/Celo
Flowcarbon	GCO2	N	Gold Standard	Goddess Nature Token (GNT)	Celo
MOSS	MCO2	Y	REDD+	—	Ethereum
Carbovalent	SCT	N	Verra/Gold Standard	Blue Carbon Credit (BCC) Forest Carbon Credit (FCC)	Solana
Nori	NRT	N	US Croplands (Custom)	NORI (Utility Token)	Ethereum
Likvidity	LCO2	Y	REDD+	LIKK (Utility Token)	Ethereum

Table B.1: Tokenized carbon credits vary in fungibility status, methodology, associated fungibility layers and utility tokens, and the hosting blockchain.

contracts and their characteristics. To our knowledge, tokenized carbon credits all conform to token standards typical for the blockchain on which they are deployed. For Ethereum, these are the ERC20 and ERC721 standards, and for Binance, these are the BEP20 and BEP721 standards for, respectively, fungible and non-fungible tokens. Standards for fungible and non-fungible tokens tend to be highly compatible, though not exactly the same. For example, tokens on separate blockchains will require bridging, and are secured by distinct consensus algorithms [37]. Even so, these technical differences are not insurmountable.

Tokenized carbon credits differ technically in other ways, including the degree to which they are fungible. Some are natively fungible, while others are tokenized as NFTs, containing granular data on the carbon credit or carbon project backing it in their metadata, which can then be fractionalized or traded for a fungible token. Tokens can also differ in how much carbon one token represents (*e.g.* 1 tonne per token) or whether they represent captured carbon or avoided emissions. Most importantly, tokens differ in the tokenomics, or in the various incentive mechanisms that underpin the token, its trade, and distribution.

With this broad framework in mind, we consider several tokenized carbon credits which are hosted by various blockchains, highlighting their features as it relates to these two categories of variability. For each of these tokens, we follow their technical structure starting with the ledger or methodologies that back the tokens, how they are bridged or minted onto the blockchain, and then moving to the token type and how they address issues of fungibility and liquidity. We note that these projects are at varied technological readiness levels (TRL) [210]. Of the carbon credits we survey, at the time of writing Toucan is fully competitive, Nori is in deployment, and some aspects of Flowcarbon are still at the proof of concept stage. Because our goal is to understand differences in design and implementation, the details of these projects are relevant to our discussion irrespective of their TRL.

B.3.1 Toucan

Toucan tokenizes Verra credits onto Polygon via their *Carbon Bridge* [176]. Anyone who owns a Verra credit can use the Carbon Bridge, subject to two criteria: the backing methodology of the Verra credit must not be on their blocklist, and the credit's *vintage* (the difference between the date of verification and the date of issuance) must not exceed ten years. At the time of writing, the blocklist only has one methodology, AM0001, which relates to refrigerant manufacturing, and which Verra stopped producing in 2014 [144]. The bridging process is elaborate, one-way, and non-custodial. To bridge, users create a BatchNFT token contract into which the credits can be bridged. They then retire the credits on Verra with specific information about their NFT contract, and then update their contract with the Verra serial number. Users then await approval from Toucan, where Toucan checks that the token contract and retirement on Verra align. If approved, Toucan mints the carbon credits into the NFT contract.

Once bridged, the NFT can be fractionalized using a token contract from Toucan's TCO2 class of ERC20 (fungible) token contracts. Each TCO2 token contract faithfully preserves the metadata of the NFT it fractionalizes, so distinct TCO2 contracts are not mutually fungible. However, the Toucan team has the explicit goal of enabling on-chain trading with high liquidity (as we have mentioned), so they have a pooling mechanism which acts as a fungibility layer on top of the TCO2 contracts. Each pool allows users to deposit TCO2 tokens in exchange for a fungible pool token which is backed by other TCO2 tokens that meet the pool's acceptance criteria. At the time of writing there are two pools, BCT (Base Carbon Tonne) and NCT (Nature Carbon Tonne), each of which has a list of approved methodologies of Verra credits which are allowed in the pool. BCT and NCT tokens can also be bridged from Polygon onto Celo.

B.3.2 Flowcarbon

Flowcarbon tokenizes carbon credits from recognized, non-profit registries onto Celo [161]. They handle the entire tokenization process themselves instead of a public portal or API. Someone wishing to tokenize a carbon credit submits a request, after which the credits are transferred to and held custodially with a bankruptcy remote special purpose vehicle (SPV). Flowcarbon then mints a token for the user. The credits remain on the registry *unretired*, which means that, in contrast to Toucan, tokenization is a two-way bridge: tokenized credits can be redeemed for their underlying credits.

Tokens are minted using Flowcarbon's GCO2 class of ERC20 tokens, including in the metadata the relevant details to the underlying credits that were tokenized. These are similar to Toucan's TCO2 family of tokens, except that GCO2 tokens do not fractionalize an NFT. Flowcarbon has a fungibility layer similar to Toucan's, where their pooling token is a *bundle token*, each bundle has its own acceptance criteria, and bundle tokens are backed one-for-one by GCO2 tokens. At the time of writing, there are no active bundles. The Goddess Nature Token (GNT) is planned to be the first, whose acceptance criteria are that a carbon credit have a nature-based methodology and a five-year vintage period.

B.3.3 MOSS

MOSS tokenizes legacy-ledger carbon credits into its fungible MCO2 token on Ethereum [166]. The MCO2 token is backed one-for-one by carbon credits which are chosen at the discretion of the MOSS team from globally recognized registries. The tokenization process, then, is very simple and done by the MOSS team, where they issue tokens for credits that they have in custody. As the user does not participate in the tokenization process, any criteria or guards on which credits are acceptable for MCO2 are made and enforced by the MOSS team. MCO2 tokens represent ownership of an unretired carbon credit, held custodially by MOSS. The MCO2 token is also a fungible ERC20 token, where one MCO2 token represents a carbon credit for one tonne of prevented CO2 emissions. Since it is backed by a variety of credits, the MCO2 token is more similar to Toucan's BCT and NCT tokens (resp. Flowcarbon's GNT token), which are backed by a pool of tokens, than to the more granular TCO2 tokens (resp. the GCO2 tokens) which represent a specific tokenized carbon credit.

B.3.4 Carbovalent

Carbovalent, built on Solana, uses the *Morpheus Carbon Bridge*, a public bridge with a similar tokenization process to Toucan's, to allow users to tokenize legacy-ledger credits from Verra and Gold Standard [152]. The only guard is that all bridged carbon credits must have been issued within ten years of their claimed vintage end date. One creates an empty NFT contract on Solana, retires their credits on Verra or Gold Standard, and then updates their NFT with the serial number generated by the offset. Carbovalent then verifies and approves the bridging, activating the NFT.

Once bridged, NFTs can be fractionalized into Solana Carbon Tonne (SCT) tokens, where one SCT token corresponds to one tonne of sequestered or prevented emissions. SCT tokens can either be retired to offset emissions or deposited into the *Carbon Vault* in return for so-called *index tokens* Blue Carbon Credit (BCC) or Forest Carbon Credit (FCC). These index tokens function much like aforementioned pool tokens, except BCC targets coastal wetlands and FCC targets forests. One can then trade BCC and FCC on Carbovalent DEX, a decentralized exchange for carbon credits, or trade carbon credits on an orderbook DEX.

B.3.5 Nori

Nori substantiates their own carbon credits, issuing carbon credits to projects which can prove that they have captured carbon and have agreed contractually to store it for at least ten years [171]. An independent third party verifies the project, and the credits issued by Nori are called Nori Carbon Regenerative Tonnes (NRTs). At the time of writing, the only methodology successfully used to substantiate carbon credits has been a version of a custom methodology called *US Croplands* [178].

Each NRT represents a verified claim that one tonne of carbon dioxide has been removed from the atmosphere, along with a contractual commitment that the removed carbon be sequestered for at least ten years. NRTs are retired immediately on the point of sale, so they cannot be traded on a secondary market. Even so, Nori has a fungible token, the NORI token, which acts as a fungibility layer over the NRTs. Each NORI token is redeemable one-for-one for an NRT, where the act of redemption retires the credit immediatly. In contrast to the NRTs, the NORI token can be traded on a secondary market.

The NORI token differs from previously mentioned fungibiliy-layer tokens. It is not backed by pooled credits, as NORI tokens are not minted in exchange for pooled NRTs. Instead, the NORI token is an independent cryptocurrency with complex tokenomics, including a token launch and distribution, a treasury, a supply cap, and an insurance mechanism. Nori holds unretired NRTs, and the tokenomics of the NORI token guarantee it to be redeemable one-for-one to retire carbon credits. The NORI token is a cryptocurrency, partially deriving its value from the value of NRT tokens, as well as a utility token, as it can be used to retire carbon credits. In the event of a breach of contract, where sequestered carbon backing an NRT is released before ten years, NORI tokens are automatically taken from the insurance pool to purchase and retire new credits.

B.3.6 Likvidity

Likvidity tokenizes carbon credits onto Ethereum, using its fungible LCO2 token [165]. Each LCO2 token represents one tonne of carbon dioxide sequestered from the atmosphere, but the token itself is backed by a portfolio of twenty different carbon projects in order to diversify risk associated with any particular project. Credits are currently tokenized from legacy ledgers, including Verra.

Likvidity also has a utility token, LIKK, which can be staked in return for rewards in the form of LCO2 tokens and *escrowed LIKK*, or esLIKK. LIKK and esLIKK are governed by their tokenomics, including a vesting schedule for stakers, a distribution rate, rewards for liquidity provision, a release schedule, and a supply cap at 1 billion tokens. Users can offset their own emissions with their staking rewards, and similar to the NORI token, LIKK is a cryptocurrency, meant to be used as a medium of exchange.

B.3.7 Others

There are many others not covered here, but we finish by mentioning a few that deal with carbon capture in some adjacent ways to the above. Regen Network is a custom blockchain made to host climate data and carbon credits [174]. It was built with the Cosmos SDK, whose native token is the REGEN token. They mint NCT tokens (of the same standard as the Toucan NCT tokens) as fungible tokens, and have partnered with Toucan to bridge between the Regen Network and Polygon [116].

Other groups use NFTs to tokenize, conserve, and reforest land. Cascadia Carbon allows users to tokenize trees into a so-called *NFTree*, and gives rewards in their native token, CODEX, which is meant to be a carbon-backed stablecoin [153]. Other groups have the concept of an NFTree, including NFTreeHaus [169], and some groups simply called NFTrees [135, 170]. Further groups, like Rewilder, raise money with NFTs to purchase and conserve land, effectively tokenizing the land similar to the concept of an NFTree [175]. Finally, Save Planet Earth has various projects combatting climate change, which are facilitated by their cryptocurrency SPE, which is traded on BNB [173]. As they grow, they will sell carbon offsets which can be bought with SPE, advocating their cryptocurrency as a medium of exchange more broadly.

B.3.8 Summary

While there is some reasonable amount of similarity between tokenized carbon credits—most tokens represent one tonne of captured carbon, captured by a reputable source—tokenized carbon credits vary in fungibility status, methodology, associated fungibility layers and utility tokens, and the hosting blockchain. Some represent retired tokens, some unretired; some allow for secondary market trading, while others do not; for some the bridge is two-way, while for others tokenized carbon credits cannot be redeemed for underlying carbon credits on a legacy ledger; and some are explicitly concerned with the permanence of carbon storage, implementing insurance protocols or diversifying risk, while others do not explicitly take permanence into account.

However, it is in the fungibility layer that we get particular disunion. Of the examples we gave, Nori and Likvidity achieve fungibility through a utility token (NORI and LIKK, respectively), each of which is governed by a custom tokenomic structure. Toucan's Nature Carbon Tonne (NCT) token and Flowcarbon's planned Goddess Nature Token (GNT) both target nature-based solutions, but are backed by distinct methodologies. Carbovalent's Blue Carbon Credit (BCC), which targets carbon sequestration in coastal wetlands, and Forest Carbon Credit (FCC), which targets carbon sequestration in forests, both target nature-based solutions, but at a more granular level. Toucan's Base Carbon Tonne (BCT), MOSS's MCO2 token, and Likvidity's LCO2 token represent generic carbon credits, though MCO2 credits are chosen and tokenized at the discretion of MOSS, BCT are Verra credits, and LCO2 tokens represent part of a diversified portfolio of credits. Each of these fungibility layers is an attempt to make carbon credits tradeable with high liquidity, but we can see that in practice these trading pools do not intersect.

B.4 Trading Carbon Credits

Fungibility layer tokens tend to be traded on automated market makers (AMMs) rather than orderbook-style exchanges. At the time of writing, Toucan's BCT can be traded on QuickSwap and SushiSwap (Polygon), and NCT can be traded on Osmosis via the Regen Network. Flowcarbon's GNT can be traded on SushiSwap as well. MCO2 can be traded on Uniswap and QuickSwap. The only exception is Carbovalent, which has an orderbook-style decentralized exchange, called Carbon DEX, which is a central limit order book (CLOB), though there is nothing technical preventing trading on an AMM. While some of these carbon credits trade on the same AMMs, to our knowledge there are no trading pools directly between tokenized carbon credits from distinct projects. There are definitely no trading pools that combine carbon credits from various projects for higher liquidity.

Carbon credits are also hosted on a variety of distinct chains. Of the tokenized carbon credits we surveyed, these include Solana, Ethereum, Polygon, Celo, and the Regen Network. Toucan is by far the most prolific of these, as their tokens are hosted on Polygon, Celo [115], and Regen Network [116]. Much work has gone into blockchain interoperability, [37] including to develop cross-chain swaps and cross-chain smart contracts, [102, 149] which may play a role in facilitating cross-chain carbon markets.

Despite fungibility layers, pooling mechanisms, and any cross-chain bridging, the market liquidity is fractured. Even so, each of the aforementioned projects hopes to trade on-chain with high liquidity. Toucan argues that pooling allows "for some level of commoditization by pooling similar carbon tokens," and claims that "this is necessary to produce a transparent price signal to the market for different categories of carbon credits." [176] Flowcarbon argues that "liquidity is at the heart of any efficient market," and that it "reduces market volatility and overall risk for the market participants." [161] Finally, Open Forest Protocol (OFP), whom we will see in Section B.6, argues that carbon markets as-is are "fragmented, illiquid, and prone to problems of ... price volatility." [172] The simple advantage of the pool token as a solution for market liquidity is that it is fungible and higher in total quantity than its individual constituents.

The solution to this fractured market may include more complex and diverse pools. However, we note a tension between pooling for high liquidity and preserving the key characteristics of each carbon credit. Because existing token pools value all the constituent tokens equally, any time tokenized carbon credits are pooled together, their individual differences beyond the pool's acceptance criteria are discarded for the sake of fungibility. Any such pools will likely have to be able to value constituent tokens relative to each other, not just at a rate of one-for-one.

B.5 Programmable Carbon

In addition to trading, we consider interoperability in terms of the applications that are built on carbon credits. Because interoperability manifests itself in part in how easy or difficult it is to build applications on top of these credits, the kinds of applications which build on tokenized carbon credits are a key component that could inform token interoperability standards. There are already a variety of applications that build directly on tokenized carbon credits or the data which backs credits. Let us review a number of them, and then discuss some key takeaways.

B.5.1 Offsetting Services

A primary purpose of tokenized carbon credits is that they can be retired to offset emissions. Aside from the retirement functionality that all tokenized carbon credits given here offer, there are already some services built around offsetting emissions. For example, from the projects we have seen, Flowcarbon also offers automated offsetting for Web3 users and Nori offers automated offsetting for businesses [24].

B.5.2 Carbon-Backed Digital Assets

There is an emerging landscape of tokenized digital assets which are backed by carbon credits or related climate data. KlimaDAO, who works with Toucan, issues a carbon-backed currency called KLIMA [163]. KLIMA tokens are backed by at least one retired carbon credit from various sources, including BCT and MCO2 tokens. KLIMA can also be minted by the rules governing the treasury, which themselves are governed by the DAO. The goal of this project is to drive up demand for carbon credits, creating what they call a *carbon economy* in which the currency is carbon-backed, and the true cost of carbon is internalized into every transaction.

There are various other projects with similar goals to KlimaDAO. Climatecoin has a coin, ClimateCoin, backed by carbon credits, a governance token, CLIMAT, and a DAO ecosystem that attempts to facilitate and fund sustainable development and make a transparent marketplace [155]. (Note that Climatecoin has a low TRL [210].) KumoDAO is a small project which attempts to back a stablecoin with carbon offsets [164]. And, as previously mentioned, other examples include CODEX and the NORI token. Finally, Arbol sells agriculture and energy derivatives based on climate data collected and monetized on dClimate [150].

B.5.3 Climate Insurance

Arbol also offers parametric insurance to guard against issues related to climate, *e.g.* unexpected weather. The payout is based on a predetermined trigger event, governed by a smart contract, which can be verified using data on dClimate (see Section B.6).

B.5.4 Art and Gaming

Finally, there is a thriving art and gaming ecosystem built around tokenized carbon credits, ranging from NFTs to metaverse projects. Celostrials is an algorithmically-generated collection of NFTs on the Celo blockchain which have partnered with Toucan, and will allow holders of Celostrials to “carbonize” their NFT with NCT tokens [154]. Celostrials holders will also be able to earn so-called climate activity rewards. Flowcarbon is also launching a collection of NFTs, named Flow3rs, which were auctioned off to support various climate-positive projects, including projects which tropical forest conservation, biodiversity, and carbon sequestration [160]. Flowcarbon also calculates and offsets on-chain emissions of NFT projects. Moving on, Likvidity has the Origins Collection, which is a collection of one thousand twenty carbon-backed NFTs [7, 189]. Holders of Origins NFTs can stake their NFTs and earn carbon credits, and have other benefits as part of the originators club. And finally, Ecosapiens is an NFT project where minted NFTs are backed by fifteen tonnes of captured carbon, and which are meant to be used as profile pictures (PFPs) [142].

Taking a slightly different direction, Nori has an API that allows any artist who mints NFTs to offset their minting, and then choose a percentage of their sales that are automatically diverted to purchase and retire carbon offsets. KlimaDAO also interacts with the offsetting process with an initiative called "love letters," where someone retiring carbon can include a message, or a "love letter to the planet," which accompanies the act of retirement. These love letters are encoded on the blockchain, and KlimaDAO has a dashboard where they can be seen.

Finally, in the metaverse space, Metamazonia is building a 3D, photorealistic metaverse to make a digital twin of certain parts of the Amazon rainforest [168]. The metaverse can be explored with an avatar. They use NFTs as a funding mechanism to prevent deforestation, promote R&D in the Amazon, and to fund other projects. NFTs correspond to pieces of land in the metaverse, which themselves correspond to coordinates in the physical reserve in the Amazon rainforest.

B.5.5 Relating to DeFi

Some of the aforementioned efforts relate to trends in decentralized finance (DeFi). In particular, these are carbon-backed digital assets like KLIMA and Climatecoin that attempt to back a more typical digital asset, such as a stablecoin, with carbon credits. Others include derivatives, both on climate data (*e.g.* Arbol) [150] and on carbon credits themselves (*e.g.* Carbovalent's index tokens) [152]. While still in early stages, as tokenized carbon credits mature as a digital asset class, we may see more in the way of derivatives, yield farming, synthetics, and other DeFi-related applications [190, 200].

B.5.6 Key Takeaways

There is already a vibrant ecosystem of applications that build on tokenized carbon credits which align with the general goals of climate action, from art and NFTs to services and derivatives. As carbon credits grow in prominence on blockchains, this is likely to continue developing. From the lens of token interoperability, note that there is little to no technical hurdle to interoperation between various carbon credits, so long as they are on the same blockchain. Because these tokens conform to established token standards, swapping out *e.g.* which carbon credits are retired, or which carbon credits back an NFT of some kind (for two examples), would likely be little more than changing a contract address or a line of code.

The real hurdle, however, to interoperation between tokenized carbon credits in the above examples comes in the semantic meaning of each of the applications: retiring one carbon credit instead of another is not necessarily the same thing semantically, even though technically it is likely a nearly identical process; likewise, the carbon credits that back a particular NFT have to match the ethos and goals of the NFT project itself, so it is unclear *a priori* if one carbon credit can be substituted for another in these instances.

B.6 Climate Data Availability

Readily available climate data is essential for the process of verifying and tokenizing carbon credits which are substantiated by novel methodologies, and for building applications built on carbon credits which rely on up-to-date climate data. Despite the fact that at the time of writing most carbon credits are tokenized from legacy ledgers, for which data is collected and analyzed off-chain, there are good reasons for the data to be publicly stored, and for the methodology to be transparently applied to the data. These include verifiability, reproducibility, and reducing the need to trust intermediaries, and are generally in line with the advantages blockchains offer to voluntary carbon markets. Furthermore, if the process can be largely automated, and climate data can be collected and made available *en masse*, then the process of verification may become more scalable. Finally, if blockchain-based carbon credit projects rely only on legacy ledgers, then the project may be dependent on decisions made by verification agencies out of their control (see Verra's recent statement on tokenized carbon credits and Toucan's response) [114, 177]. We will take a brief look into three organizations which are looking to collect and store climate data in a decentralized fashion.

B.6.1 Filecoin Green

Filecoin is a decentralized ledger for storing data using the IPFS protocol. With Filecoin, a user can pay for storage to be hosted on the Filecoin network over a specified period of time. Filecoin Green is an initiative on Filecoin to store climate data and make it broadly available [159]. Their stated goal is to build infrastructure so that anyone can make transparent and substantive environmental claims. Their first initiative on climate data is to measuring the electricity consumption of their validators to be transparent about the emissions of the blockchain itself. Their goals are to host more extensive climate data and make it available to various applications, which could include tokenized carbon credits.

B.6.2 dClimate

dClimate is a decentralized network for climate data, consisting of four layers: the governance layer, through which dClimate operates like a DAO via its native WTHR token; the oracle layer, which makes climate data available to applications, operating like Chainlink; the blockchain and data storage layer, where data is stored via IPFS; and the marketplace layer, where users can access (and pay for) data [158]. Climate data is published confidentially until accessed and paid for by a user, though contributors can choose to make their data free. As we mentioned previously, Arbol offers parametric insurance against climate-related events, using dClimate as its data source. More broadly, dClimate is attempting to lower the barrier to entry for climate data capture and monetization, and could host data which is relevant to carbon credits and the corresponding ecosystem of applications.

B.6.3 OFP

The Open Forest Protocol (OFP)is a protocol built to manage forest data, with the goal of improving forests [172]. While the protocol hopes to eventually mint carbon credits backed by data managed by the protocol, it is also meant to be a more general forest data management tool. The protocol itself has a native token, the OPN token, which grants access to the OFP, allows a holder to verify or challenge the accuracy of a specific MRV data upload, and governs the protocol's parameters as a DAO. When the time comes, the process of creating carbon credits with OFP is meant to be open and transparent. Users will be able to create a new project and upload data, which will generate an NFT contract for them. To mint tokens, on-the-ground forest data must be collected by the OFP field mobile app, after which validators check the legitimacy of the ground data. Acceptable methodologies to mint carbon credits, as well as any guards on what types of credits or methodologies are acceptable, will be governed by the DAO.

B.6.4 Key Takeaways

Making climate data readily available to blockchain-based applications, and incentivizing people and entities to collect and publish that data, will no doubt play an important role in how tokenized carbon credits are minted and in other financial derivatives built on-chain using the data. If the data is high quality, this could support a wide range of methodologies which compute over the data to substantiate tokenized carbon credits on the blockchain. If done in a transparent and verifiable way, this could be a highly effective marketplace of ideas that works to mitigate climate change and monetize carbon capture from the atmosphere, among other things.

B.7 Related Organizations

Finally, we mention a few groups that are interested in decarbonizing blockchains, in using blockchains for climate action, and in making standards for tokenized carbon credits. From the work of these groups, we are most interested in the work of standards for tokenized carbon credits, though each contributes more to the ecosystem.

The Crypto Climate Accord (CCA), an initiative to decarbonize blockchains and cryptocurrencies supported by various companies and nonprofits, has as its goal to “develop standards, tools, and technologies with CCA Supporters to accelarate the adoption of and verify progress toward renewably-powered blockchains by the 2025 UNFCCC COP30 conference.” [157] It puts forward various solutions, including guidance for accounting and reporting electricity use and carbon emissions from cryptocurrencies.

The Climate Collective is a coalition of entrepreneurs, investors, non-profit organizations, and scientists, whose aim is to promote blockchains and cryptocurrencies (Web3 infrastructure) as a tool for climate action at scale [156].

Finally, Gold Standard has launched some working groups with the aim to develop “an open, global collaboration on digital solutions for carbon market standards and monitoring, reporting, and verification (MRV).” [125] They consist of a digital assets working group, which “looks at the role of blockchain to track carbon credits in decentralised environments;” an open APIs and digital infrastructure working group, which “looks into how new digital methodologies can increase the robustness of carbon credit calculations;” and a digital MRV working group, which looks into the “details of how to turn earth observations into meaningful carbon metrics.”

While we expect the token contracts of these carbon credits to all conform to established token standards, we do not yet have a standardization framework for methodologies substantiating carbon credits. Because methodologies can vary widely, as we saw in Section B.3, we do not expect standards to be strictly prescriptive in the way that token standards are. However, standards on token metadata, data collection, or measurement practices could be useful as key guidelines to ensure that tokenized carbon credits are rigorously substantiated and can be effectively compared along various metrics.

B.8 Conclusion

Tokenized carbon credits vary in their implementation details and in what substantiates them (Section B.3). As the goal of this research is to help maximize interoperability between tokens, we face the technical hurdle that carbon credits exist on various blockchains and with various token standards. Luckily, in practice most (non-fungible) tokenized carbon credits are pooled into a fungible token that conforms to established token standards. Thus by using inter-chain bridges, the technical hurdle itself for token interoperability is not particularly high.

Instead, as we saw, interoperability can be hampered by variations in what substantiates a token. In contrast to token standards, where it is productive to have all tokens conform to established standards, it is likely unproductive to fully standardize the data and methodologies that substantiate carbon tokens. Instead, we can drive innovation with an open and transparent marketplace for methodologies of data capture and processing that rewards accuracy and efficacy. In time, such an open marketplace could be supported by the various decentralized carbon data collection and availability schemes (Section B.6), and reproducible or verifiable computation over them. Even so, organizations seeking to form some consensus around standardization could prove productive in providing a common framework for these methodologies (Section B.7), which could include standard ways of structuring token metadata.

In some sense, however, this desire for a rich variability in the methodologies substantiating tokenized carbon credits stands in conflict with the nearly ubiquitous goal of having as much liquidity as possible on-chain by pooling carbon credits together (Section B.4). This is because, as they exist, fungibility layers pool tokens by valuing them one-for-one, and so tokens can be pooled together insofar as they are valued equally to each other. This achieves fungibility by *discarding* differences between the constituent tokens in methodology, project type, vintage, *etc.* Because we *want* carbon credits to be able to vary in price depending on their characteristics, ideally where higher quality credits are valued more highly, we are inherently restricted in our ability to pool tokens and achieve higher levels of liquidity on-chain.

Ideally, we would be able to pool tokens in such a way that values distinct tokens differently according to some market price (*e.g.* a carbon credit worth twice as much as another would trade for twice as many pool tokens as the other) and allows for dynamic price discovery between the constituent tokens over time. This would achieve the goals stated above of high liquidity and fungibility, while still valuing carbon credits individually with their granular data, encouraging a rich landscape of tokenized carbon credits. It could also prove useful for interoperability as it relates to applications building on tokenized carbon credits (Section B.5), achieving our goals of interoperability more broadly.

Glossary

address An address, or wallet address, is a public key used to represent a destination for transactions on a blockchain network. Addresses are used to send and receive digital assets, and they are generated using cryptographic algorithms to ensure security. See also wallet. 19

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. This in contrast to traditional centralized exchanges, which match buyers and sellers and take a cut of the transaction as a fee. 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. 15, 145

Binance Smart Chain Binance Smart Chain (BSC) is a blockchain developed by the Binance exchange. It is built on top of the Ethereum Virtual Machine (EVM) and uses a similar smart contract language to Ethereum. 18, 145

blockchain A blockchain is a distributed and decentralized digital ledger that records a secure and immutable ledger of transactions across a network of computers. Each transaction, bundled into a block, is cryptographically linked to the previous one, forming a chain of blocks, hence the name "blockchain." Blockchains are commonly used for cryptocurrencies like Bitcoin or Ethereum, but have applications beyond digital currencies including supply chain management, voting systems, and smart contracts. 15, 17

cross-chain bridge A cross-chain bridge is an application, not necessarily decentralized, that allows for the transfer of assets or information between different blockchains. Possible assets that can be exchanged or transferred include tokens, coins, and other digital assets, even if the blockchains being bridged have different consensus algorithms, security models, and underlying infrastructure. 15, 19

crypto insurance protocols Crypto insurance protocols are DeFi 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. Claims are processed and approved by a variety of different mechanisms, depending on the nature of the insured event and whether or not human intervention is required to assess claims. 15, 142

crypto lending Crypto lending, or decentralized lending, refers to the practice of borrowing and lending digital assets within the context of blockchain-based financial systems, such as DeFi platforms. Users can lend their cryptocurrency holdings to others in exchange for earning interest on their loans, while borrowers can access funds by providing collateral in the form of other cryptocurrencies. These transactions are facilitated through smart contracts on blockchain networks, instead of traditional intermediaries like banks. 15, 142

decentralized application A decentralized application (dApp) is a software application that runs on a decentralized network like a blockchain, and is not controlled by any central authority. Decentralized applications can be used for a variety of purposes, ranging from financial applications like exchanges, to gaming platforms, and social media sites. 142, 145

decentralized exchange A decentralized exchange (DEX) is a type of decentralized marketplace that operates on a blockchain, allowing users to trade cryptocurrencies and digital assets directly with each other without the need for intermediaries. Examples of DEXs include, but are not limited to, automated market makers AMMs and decentralized auctions. 15, 141, 145

decentralized finance Decentralized finance (DeFi) refers to a financial ecosystem built on blockchain technology that largely operates without traditional intermediaries. It enables peer-to-peer transactions and interactions with digital assets through smart contracts and decentralized protocols. DeFi includes various services like lending, borrowing and trading. DeFi applications include, but are not limited to, DEXs, AMMs, crypto lending, and crypto insurance protocols. 15, 145

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. 18

entrypoint function A contract entrypoint function is a public-facing function 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, or 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. 16

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. Its native token, ETH, is used to pay for gas fees. It also supports Solidity, 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. 16, 141, 143–145

Ethereum Virtual Machine The Ethereum Virtual Machine (EVM) is a decentralized, Turing-complete virtual machine that serves as the runtime environment for executing smart contracts on the Ethereum blockchain. Developers can write and deploy smart contract code in high-level programming languages like Solidity, which is then compiled into bytecode that can be understood and executed by the EVM. 141, 144, 145

flash loan In DeFi, a flash loan is a type of decentralized loan that allows a user to borrow funds for a single, atomic transaction, without collateral and with a very low interest rate. The must loan is repaid automatically at the end of the transaction. Any transaction that takes out a flash loan which does not end in repayment is invalid. Flash loans are used for a variety of purposes in DeFi, including exploiting market inefficiencies and executing arbitrage strategies. 18, 143

flash loan attack A flash loan attack is a type of exploit in DeFi in which an attacker uses funds borrowed with a flash loan to execute a series of manipulative trades or arbitrage opportunities. The attacker profits from these trades at the expense of other users, such as liquidity providers, taking advantage of the borrowed funds without bearing any risk or requiring collateral. 18

front-running attack In a front-running attack, an actor inserts certain transactions in a block ahead of others which, by their order, are profitable by gaining an unfair advantage and causing financial losses to victims. This manipulation involves monitoring pending transactions, identifying lucrative opportunities, and quickly submitting their own transaction with higher fees to exploit market inefficiencies. 17

gas Gas is a term used to describe the fee required to process a transaction on Ethereum and other blockchains, paid in the blockchain's native token (*e.g.* ETH). The amount of gas required for a transaction depends on the computational complexity of the operation being performed and the demand for block space. 143

liquidity provider In DeFi, a liquidity provider is an entity who supplies their digital assets to liquidity pools, enabling trading and financial activities on the platform and earning rewards in return. 16, 143

peg A peg in the context of digital assets refers to a fixed exchange rate between a cryptocurrency and a real-world asset, such as the US dollar or gold. A pegged asset or cryptocurrency is one whose value maintains a stable value at some peg. Some pegged assets are backed by a reserve of the asset they are pegged to, which can help ensure the stability of the peg even during market fluctuations. Others maintain their peg algorithmically through various debt mechanisms. 144

smart contract A smart contract is a computer program, stored on a blockchain, that automatically executes when certain conditions are met. Smart contracts do not require intermediaries to enforce their terms. They can facilitate exchange of assets, such as cryptocurrencies, and have a wide variety of use cases. 15, 18, 141

Solidity Solidity is a high-level programming language for writing smart contracts that can be compiled into bytecode be executed on the Ethereum Virtual Machine (EVM), and stored on the Ethereum blockchain. 143

stablecoin A stablecoin is a type of cryptocurrency designed to maintain a stable value, typically pegged to a stable asset, such as a fiat currency (e.g., USD, EUR) or a commodity (e.g., gold). The main goal of stablecoins is to reduce price volatility commonly associated with traditional cryptocurrencies like Bitcoin or Ethereum, providing a more reliable medium of exchange and store of value. Stablecoins achieve stability through various mechanisms, such as collateralization, algorithmic control, or a combination of both, which are designed to ensure that the value of the stablecoin remains relatively constant over time. 15, 18, 144

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. They include, but are not limited to, stablecoins, and like stablecoins maintain their peg through various mechanisms such as collateralization, algorithmic control, or a combination of both. 15

Tezos Tezos is a third-generation, proof-of-stake blockchain which supports smart contracts written in Michelson. Its native token is XTZ. 15, 145

transaction A transaction is an action or operation that alters the state of the blockchain network. It typically involves the transfer of digital assets, such as cryptocurrencies or tokens, from one user to another, but can also encompass various other types of interactions, such as smart contract executions, data storage, or authentication processes. Transactions are recorded in blocks and cryptographically linked together. Each transaction must be verified and validated by network participants, known as miners or validators, to ensure its authenticity and compliance with the network's consensus rules before being added to the blockchain. 18, 144

wallet In the context of a blockchain, a wallet refers to a public, private key pair which can be used to execute transactions on a blockchain and hold digital assets. Transactions originating from a wallet must be signed by the private key. The term wallet also refers to software or a device that stores and manages blockchain-based digital assets. 19, 141

yield farming Yield farming is the practice of providing liquidity to DeFi protocols in exchange for rewards in the form of interest or tokens. The rewards can come from the fees generated by the protocols, or through inflation, and are typically distributed to liquidity providers as a way of incentivizing them to provide liquidity and ensure the stability of the platform. 15

Acronyms

AMM Automated market maker. 15–17, 19, 141, 142

BSC Binance Smart Chain. 18

dApp decentralized application. 142

DeFi decentralized finance. 15, 18, 141–144

DEX Decentralized exchange. 15, 141, 142

ETH The native token of the Ethereum blockchain. 143

EVM Ethereum Virtual Machine. 141

XTZ The native token of the Tezos blockchain. 144