UNIVERSITY OF
CAMBRIDGE

Meta Properties of Financial Smart Contracts

Derek Sorensen

Clare College

September 2023

This dissertation is submitted for the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or am concurrently submitting, for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or is being concurrently submitted, for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This dissertation does not exceed the prescribed limit of 60 000 words.

Derek Sorensen
September, 2023

Abstract

Meta Properties of Financial Smart Contracts

Derek Sorensen

Financial smart contracts routinely manage billions of US dollars worth of digital assets, making bugs in smart contracts 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 captures all of its intended high-level behaviors. To mitigate this challenge, we develop formal tools to target *meta properties* of smart contracts, which are properties of a contract which its specification intends to capture, but which are out of scope of said specification. The targeted properties include the economic behaviors of the contract, properties relating to its upgradeability features, and the intended behaviors of systems of contracts. The formal tools presented are written in Coq.

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. I could not have asked for a better College Dean or better chapel camaraderie.

To my good friend Anthony Davies, who kept me sane through lockdown by giving me an extensive and strict dietary and exercise routine, and who supported me to follow it diligently despite his own suffering through lockdown. You were the pin that held me together during those dark times. Thanks to you I am off anti-depressants. Thank you for giving me my life back.

And finally, 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.

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 [135]. 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 [144].

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

1.1 Formal Verification of Smart Contracts

Much work has been done in formal verification of smart contracts [17, 38, 45, 63, 97, 121, 131]. Broadly speaking, the goal of formal verification is to formally prove that a contract is correct with regards to a specification. However, financial smart contracts have complicated specifications, and it is not at all straightforward to write one which has all of its intended behaviors. One reason for this is that specifications of financial contracts inevitably include behaviors which are expressed at a level of abstraction higher than a contract specification.

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

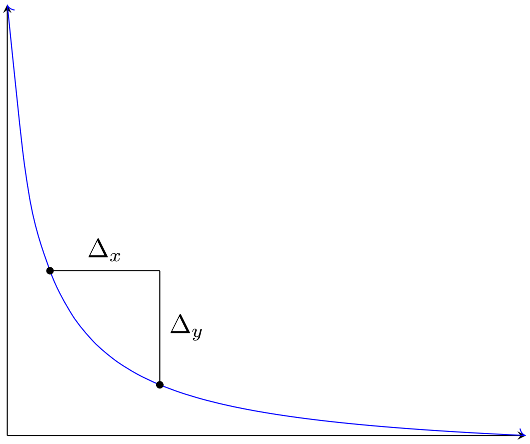

Figure 1.1: A trade of Δ_x for Δ_y along the indifference curve $xy = k$.

1.1.1 Dexter2 Verification

Dexter2, an automated market maker on the Tezos blockchain, has been formally verified by three different groups using three different formal verification tools [34, 80, 102]. Each was based on the same informal specification [16].

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. Note, however, that while the specification is detailed on the contract design and interface, it doesn't include anything about expected or desired economic behavior.

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 [33]. 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 [16] 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 [16] or its formal counterparts [34, 80, 102] actually imply any of these economic behaviors is not a given fact. AMM fees and liquidity provision alone are highly complexs topics, and are the subject of several economic studies, including: choosing optimal transaction fees [56, 62, 61], how liquidity providers react to market changes [69], and how all of that relates to the *curvature* of $xy = k$ [8]. 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 [43].

To complicate matters, a brief study of the three formalizations [34, 80, 102] of the Dexter2 specification [16] 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.

We can see a general problem in specifying financial smart contracts. Financial smart contract specifications are designed to articulate properties, in this example of an economic nature, which are essential to the contract's correct functionality but out of scope of said specification. Unfortunately, failing to correctly capture these intended properties in the specification routinely leads to extraordinarily expensive attacks. Furthermore, because these are vulnerabilities of the *specification*, as it stands formal verification is useless to mitigate them.

1.2 Contract Vulnerabilities and (In)Correct Specifications

Vulnerabilities of financial smart contracts are a serious cause of substantial financial loss. Blockchain-based applications lost 2.44 billion USD in 2021 and 3.6 billion USD in 2022 due to attacks [23, 146]. In 2022, financial smart contracts were the most attacked type of blockchain-based application, making up about two-thirds of all attacks [60, 23]. Importantly, attacks which exploit improper business logic or function design—those which concern us here—are in the top three causes of loss [23, p.10].

We can isolate three classes of vulnerabilities due to poor contract design and specification. These are: economic vulnerabilities, vulnerabilities introduced through poorly-specified upgrades or contract upgradeability, and vulnerabilities due to difficult-to-specify behaviors of a systems of interacting contracts. For each of these classes of vulnerabilities, we give examples of recent, successful attacks and discuss what is needed to target such vulnerabilities with formal verification.

1.2.1 Economic Attacks

Poorly specified financial smart contracts can be vulnerable to costly *economic attacks*, or attacks on the contract's economic design.

Take for example Beanstalk, an Ethereum-based stablecoin protocol which uses a decentralized governance protocol. The 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. 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 [54].

The tool used in the attack, flash loans, are loans mediated by a smart contract, issued for the duration of a single transaction. Due to their atomicity, flash loans remove the creditor's risk of debt default, and thus enable enormous, uncollateralized loans. For example, the Aave flash loan pool has had in excess of 1 billion USD which can be loaned out [112]. Flash loans can introduce unintuitive contract behavior, deviating from that of traditional markets, and have been extensively studied [65, 66, 112, 134]. 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 [57].

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 specification—contract behavior. In May 2021, an attacker drained the Spartan Protocol contract for a profit of roughly the equivalent of 30 million USD [31, 79]. Another attacker drained Pancake Bunny of 114k WBNB and 697k BUNNY tokens, amounting to about 45 million USD at the time in lost funds [30, 67, 76, 108].

Finally, Mango Markets, a Solana-based DEX, was attacked in October 2022 for approximately 116 million USD in contract assets [123]. The attack consisted of a complicated and subtle trading strategy which only a sophisticated trader would be able to see and exploit [23]. CoinDesk reported that the attacker did everything within the parameters of the platform's design [87]. 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.* [15]

We would recognize each of these exploits as attacks, despite the fact that the contracts were functioning as specified. This tells us that the specifications did not accurately capture the intended economic behaviors, and were thus incorrect. To mitigate such attacks by ensuring a specification *does* accurately capture the intended economic behaviors, what is needed is a rigorous notion of a specification's correctness as it relates to its economic properties, as well as formal tools to reason about a said correctness.

1.2.2 Unsafe Contract Upgrades

Poorly specified contract upgrades can introduce costly vulnerabilities.

Consider first 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 [116]. 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 [55, 78].

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 [59]. NowSwap, a nearly identical application, upgraded with the same error and incurred a loss of 1 million USD [22].

It is clear that none of these contract upgrades captured the actual intent of the upgrade. Each introduced vulnerabilities, as small technical changes of the contract, which compromised the contract's fidelity to its intended design. Indeed, each upgrade was meant to preserve properties of the previous contract version, regarding pricing or permissions, while changing others. As each failed to do so, they were incorrect.

In order to safely specify contract upgrades and upgradeable contracts, what is needed is a rigorous notion of a correct specification of individual contract upgrades as well as contract upgradeability, as well as formal tools to reason about said correctness.

1.2.3 Complex System Behavior

Finally, poorly specified systems of contracts can be vulnerable to extremely costly attacks.

Consider Harvest Finance, a yield aggregator on Ethereum. On October 26, 2020, an attacker used a flash loan to trade about 17.2 million USDT for USDC on Curve, which temporarily increased the price of USDC in the Curve Y pool. Due to the fact that Harvest Finance uses the Curve Y pool as a price oracle in real-time to calculate the vault shares for a deposit, the attacker got into a Harvest vault at an advantageous rate. In the same transaction, the attacker reversed the trade on Curve, after which the price of USDC returned to normal in the Curve Y pool, which increased the value of the attacker's shares in terms of the now less expensive USDC. The hacker then exited the vault at this new exchange rate for a profit of 33 million USD in lost user funds [107].

Smart Contract	Vulnerability	Loss (USD)	dApp Type	Exploit	Year
Mango Markets	Contract Design [87]	115M	DEX	[123]	2022
Beanstalk	Contract Design [57]	77M	Stablecoin	[54]	2022
Pancake Bunny	Contract Design [67]	45M	Yield aggregator	[30]	2021
Spartan Protocol	Contract Design [79]	30M	AMM	[31]	2021
Nomad	Unsafe Upgrade [78]	190M	Cross-chain bridge	[55]	2022
Uranium	Unsafe Upgrade [59]	50M	AMM	[32]	2021
Cream Finance	Complex System [77]	130M	DeFi protocol	[53]	2021
Harvest Finance	Complex System [58]	34M	Yield aggregator	[52]	2020

Figure 1.2: A small sample of recent attacks, totalling to about 776M USD in lost funds.

Similarly, consider CREAM finance (short for *Crypto Rules Everything Around Me*), a multi-purpose DeFi protocol that brands itself as a one-stop shop for decentralized finance. It offers crypto lending, borrowing, yield farming, and trading services, and has several connected implementations across multiple chains. An October 2021 attack drained the pool of roughly 260 million USD in assets [53]. The attack was extremely complex, involving 68 assets and over 9 ETH in gas, roughly 36k USD at the time [53]. Immunefi, a bug bounty platform for smart contracts, diagnoses that the exploit was due to an easily manipulable price oracle, and uncapped supply of the token yUSD [77]. Even so, the attack is complex enough that only experts such as Immunefi can give a comprehensive diagnosis.

Aside from complex economic properties, the difficulty in specifying the above examples comes in the utter complexity of interacting systems of contracts. The intended behavior of a system of contracts, typically expressed and written as if it were a monolithic entity, can be difficult to preserve when modularizing the contract design into component pieces without introducing vulnerabilities. In order to safely specify such systems, what is needed is a rigorous notion of a specification's correctness as it relates to how the system of contracts behaves when taken as a whole, and formal tools to reason about said correctness.

1.3 Meta Properties of Financial Smart Contracts

We present tools to formally specify and verify properties of financial smart contracts which a specification intends to capture, but which are out of scope of said specification. Because these properties fall outside the scope of a contract specification, and are sometimes properties of the specification itself, we call these *meta properties* of financial smart contracts.

We target three classes of meta properties: a contract's economic meta properties (Chapter 4), a contract's meta properties relating to upgrades and upgradeability (Chapter 5), and meta properties relating to the behavior of systems of contracts when taken as a whole (Chapter 6).

Our thesis is that we can mitigate costly vulnerabilities of financial smart contracts due to incorrect specifications by formally specifying and verifying a contract's meta properties.

Chapter 2

Related Work

Here we survey previous work on formal verification of smart contracts with the goal of specifying and verifying meta properties. We find that, generally speaking, those verification tools which in principle are able to specify and verify meta properties are not able to do so on code which can be compiled and deployed, and those verification tools which reason about code that can be compiled and deployed are too low-level in scope to specify and verify the meta properties of interest here.

We are able to address this issue because we do our work in ConCert [10], a Coq-based verification tool with verified extraction [11, 13]. Because ConCert has a full embedding of the execution semantics of a blockchain in Coq, an interactive theorem prover [27], we are able to specify and verify arbitrary properties of smart contracts. Because it features verified extraction, we are able to do so on code which can be compiled and deployed.

2.1 Smart Contract Verification

We consider the landscape of formal verification with the goal of expressing and reasoning about meta properties of smart contracts. We will draw on several surveys of formal verification of smart contracts, including [17, 38, 45, 63, 97, 121, 131].

From among the plethora of tools available to formally verify smart contracts, we will mostly focus on those which use interactive theorem provers, or proof assistants. Verification tools based on proof assistants tend to use the proof assistant itself as a specification language, and so smart contracts are treated as mathematical objects, about which any mathematical statement can be made. In principle, this means that proof assistants can be used to verify any correct design [115]. As we will see, we use diverse mathematical tools to formalize meta properties, so the flexibility and robustness of proof assistants come particularly in handy. We also focus on proof assistants because they are recognized in the literature to be of the highest quality for formal verification [97].

2.1.1 Smart Contract Language Embeddings

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

There are several examples of low-level language embeddings into proof assistants. For EVM bytecode alone we have embeddings in Isabelle [5, 74]; in Coq [139]; in the K Framework [73, 109]; in Why3 [143, 100]; and in Z3 [85]. There are similar embeddings for other smart contract languages, including for the Tezos smart contract language Michelson, which is a low-level, stack-based language. These include Mi-Cho-Coq, an embedding of Michelson into Coq [25]; K Michelson [81], an embedding of Michelson into the K Framework; and WhyISon, which transpiles Michelson into WhyML, the programming and specification language of the Why3 framework [42]. There are examples on other chains as well, including an embedding of the low-level, Bitcoin-based contract language Simplicity in Coq [105].

Due to their proximity in language and semantics of the lowest-level executing environment to each of these blockchains, low-level language embeddings are more likely to be faithful to the actual execution environment [84]. Equally, this low-level proximity can make it difficult to correctly specify high-level behavioral or economic properties of smart contracts without some form of abstraction.

On the other hand, higher-level languages tend to be more human-readable, and can be more straightforward to reason about [84], especially if they are statically typed functional programming languages [10]. Naively, this higher level of abstraction seems like it would make it more straightforward to reason about the correctness of a specification. However, the abstraction can come at the cost of rigor. As they sit at a higher level of abstraction, they require a rigorous language embedding as well as a correct compiler down to the low-level, executable code which preserves the proven properties [84].

We have many examples of intermediate- and high-level languages which have been embedded into proof assistants. For Ethereum smart contracts, we have Lolisa [138] and FEther [137], embeddings of Solidity into Coq; as well as embeddings of Solidity into Event-B [147]; into Isabelle [84]; into the K Framework [82]; and into F* [28]. We also have custom implementations of finite state machines used by FSolidM [90, 91] and VeriSolid [92] that verify Solidity code. For Tezos, we have an embedding of Albert [26], an intermediate-level language which compiles down to Michelson, and which targets Mi-Cho-Coq; we also have ConCert [10], which has a certified extraction mechanism from Coq code into multiple smart contract languages [11], including into CameLIGO, an intermediate-level language which compiles down to Michelson [9]. On other chains we have Plutus in Agda [37], and verification for BNB in Coq [127].

At higher levels of abstraction, we also have some DSLs written in proof assistants which target various smart contract languages. Scilla is intermediate-level, and can be used to reason about temporal properties of smart contracts, which targets Solidity [118]. Archetype is a Tezos-based DSL which targets business logic and uses Why3 [24]. Multi is a framework, written in Coq, which targets reasoning about smart contract interactions [36]. At an even higher level of abstraction, we have TLA+, a tool for reasoning about concurrent and distributed systems, which was used to verify the a cross-chain swap protocol [101].

For our purposes the key advantage to verification with intermediate- and high-level languages is that the abstraction makes it easier to write a correct specification, and reasoning about code also tends to be more straightforward [84]. However, there are disadvantages. Many of these are DSLs that specialize to target specific kinds of higher-level properties, such as a particular kind of business logic, and are thus limited in their scope. Others are specialized to reason about specifications or protocols, unattached to code which can be compiled and deployed, which then must be translated by hand down to a low-level specification. Furthermore, language embeddings tend to either omit a formalization of the semantics of the execution environment, or they make various assumptions which if inaccurate could systematically introduce unverifiable vulnerabilities.

In order to be fully rigorous, our work requires an intermediate- or high-level language embedding, which is unrestricted in the contracts it can write and properties it can specify, which includes an explicit model of the execution semantics of a blockchain, and which has a verified mechanism to extract, compile, and deploy code which has been reasoned about.

2.1.2 ConCert

We will use ConCert [10] to formally specify and verify meta properties for three reasons. Firstly, while ConCert has not yet been used to formalize meta properties of smart contracts, it is extremely well-suited to do so. The specification language is in Coq and so it is unrestricted, except by limitations of the blockchain model itself, in the kinds of mathematical statements that can be made and proved about smart contracts. It also formalizes blockchain execution semantics underlying the execution of a smart contract, which means that there is a well-defined smart contract type, `Contract`, in the context of the model, which has a specific semantics within the blockchain and which can thus be reasoned about abstractly. This, to our knowledge, is unique to ConCert. As we will see throughout this thesis, the combination of these characteristics allows us to reason unrestrictedly about smart contracts abstractly, as mathematical objects, which is key to the goal of this thesis.

Finally, ConCert's extraction mechanism from Coq into its target smart contract languages is certified, so despite it not being a low-level language embedding like Mi-Cho-Coq, we have a high-fidelity translation into code which can be compiled and deployed. This is made possible by MetaCoq [11, 12, 126], which is a tool for reasoning about the extraction mechanism. It should be mentioned that we rely on the compilers of the target languages to be correct, though compilers can be certified and so there is no theoretical barrier which prevents ConCert's certified extraction to extend to *e.g.* bytecode.

Thus ConCert gives us the advantages of verification at a higher level of abstraction without compromising the integrity of the verification results.

2.2 Verifying Meta Properties

Our work is set in the context of various types of contract vulnerabilities. While we argue for formally specifying and verifying meta properties to address these issues, there are alternative methods that one can take to address each of these classes of vulnerabilities. Rather than including a discussion of each of these here, in each of Chapters 4, 5, and 6, we discuss related work in formal methods which is more tailored to the particular class of vulnerability at hand.

Chapter 3

Background

Here we introduce the types and tactics of ConCert which are most relevant to the forthcoming work. 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 [35].

3.1 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 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.

3.2.1 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.

3.3 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.

3.4 Contract Induction

An invariant like `counter_correct` is typically proved by a custom ConCert tactic `contract_induction`. As its name suggests, 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.

Chapter 4

A Contract's Economic Properties

The first class of meta properties that we target are the economic properties of financial smart contracts. As we saw in §1.2.1, poorly specified financial smart contracts can be vulnerable to costly economic attacks. In this chapter we rigorously develop the notion of correctness of the specification of a financial smart contract from the perspective of its economic properties.

To this end, we introduce a theoretical tool called a *metaspecification*, which is a specification of a specification, and which we will use throughout this thesis to specify and verify meta properties of financial smart contracts. In this chapter, we use it to verify that a contract specification adequately captures the economic properties which were the intent of its design. We illustrate with an example financial smart contract, called a *structured pool*, but the framework we introduce can be applied generally.

We proceed as follows. In §4.1, we discuss the problem of specifying financial smart contracts with economic intent. In §4.2, we specify the structured pool contract, introducing the economic problem it seeks to solve and mathematically proving that it addresses the problem. In §4.3, we formally introduce the notion of a metaspecification and discuss its relationship to the specification. In §4.4, we formalize the structured pool specification, treating it as an axiomatization of a contract, and separating the specification from the properties of the metaspecification. In §4.5, we formalize the structured pool metaspecification, formally proving that any contract conforming to the formalized specification exhibits the desirable properties given in §4.2. In §4.6 we conclude.

4.1 Contract Specification With Economic Intent

By definition, financial smart contracts are always specified with some economic intent, and that is manifest with varying degrees of rigor in the specification process.

The least rigorous of these communicate the contract's intended economic design through rhetorical means such as diagrams, analogies, and imagery. Let us revisit Beanstalk, an Ethereum-based stablecoin protocol which was the victim of a 77 million USD attack in April 2022 [54]. Consider in particular the Beanstalk white paper [111]. Beanstalk is a complex protocol with many moving parts. Reading the white paper, one gets an intuitive notion for what each component of the protocol contributes to the stablecoin and its tokenomics through analogies to farming. The components of the protocol are referred to by a farm, the sun, silos, fields, barns, fertilizer, temperature, and humidity. There are stalks and seeds, which can be revitalized and fertilized. Finally, assets are referred to as ripe or unripe, and can be chopped. All of this imagery combines into that of a functioning farm, which gives an intuitive sense that the application design is correct in some sense. However, while each of the protocol components have a precise definition and technical notation, there is no codified, precise goal, and certainly no proofs that the definitions work together efficaciously.

Financial smart contracts which are specified with more rigor tend to define their goal in economic terms, and reason about how well their design satisfies those goals. Consider, for example, FairMM [40], an AMM designed to mitigate front-running attacks. It is specified with precise, mathematical definitions, and features proofs that the specification satisfies certain economic properties. Theorem 3, the main theorem which justifies the specification to be correct, shows that no rational, profit-seeking market maker has incentive to manipulate the price in advance of a trade. In principle, this addresses the problem of front-running; the reader might convince himself of this fact by reading the definition of a front-running attack in the paper's abstract.

This leads us to a critical question. Proving theorems about a specification is undoubtedly an important step towards rigor, and indeed to ensuring that the specification captures certain economic meta properties. But *how do we choose which theorems to prove*?

In practice, there are various ways to answer this question. We can think of FairMM's strategy as having defined a threat model, which is a common practice to evaluate software design [132]. Threat models can be designed systematically [89], and if they characterize the problem accurately then they can be useful in preventing vulnerabilities. The properties one proves in response to a threat model are those which, in the context of the model, are sufficient to neutralize or mitigate risk.

Another strategy is to explore the implications of a specification using tools like mutation and unit testing, and from there formulate propositions to be proved true of the specification. For example, Phipathananunth's recent work uses mutation testing to identify any pathological, yet correct, behavior of a specification, and from these tests derive and verify various safety properties [110]. Similarly, the developers of Djed [140], a formally verified stablecoin, used mutation and unit testing to identify potentially pathological behavior of the specification and then targeted these behaviors with formal verification to justify the robustness of the contract specification. The method of using tools like unit and mutation testing is agnostic to any theory or model, but rather helps discover properties which, on their face, are evidently undesirable. They can be used in conjunction with a threat model or some other theory to discover where the model or theory breaks down.

In any case, the theorems one proves about a specification point to some fundamental notion of correct design, and the properties one chooses to prove should be oriented toward that notion of correctness. Thus the more one is able to couch theorems of correctness in a systematic understanding of the surrounding execution environment, corresponding threats, and economic goals of the contract, the more accurately one is able to approximate and prove correctness of a specification's design.

In this spirit, substantial work has been done by Angeris *et al.* [6, 7], Bartoletti *et al.* [19, 20], and Xu *et al.* [136] to systematically understand market behavior of AMMs like Uniswap, and to create formal theories of DeFi and AMMs from which the properties constituting desirable behavior can be thoroughly reasoned through and derived. These studies range from highly theoretical to data-driven, and are a good start at a comprehensive understanding of the economic environment in which financial smart contracts operate.

In the coming sections, we will specify a novel smart contract designed to pool and trade tokenized carbon credits. We do so in the context of the cited studies and theories, and from them derive properties of well-behaved AMMs and other DeFi applications. While the context from which these properties are derived—the studies and formal models on which we draw—can always be improved, we argue that this is an example of rigorous contract design and specification, as there is a clear notion of correctness which is couched in substantial theoretical and practical work on the behavior of financial smart contracts.

4.2 Case Study: Structured Pools

We now move on to specify a financial smart contract, called a *structured pool*, which is designed to address issues of fungibility in blockchain-based carbon markets [125]. We will proceed by defining the economic problem that structured pools aim to solve (§4.2.1), specifying the contract (§4.2.2), and then justifying that the specification meets our economic goals (§4.2.3). Importantly, the theorems that we prove about structured pools are derived from the previously-mentioned work on theories of AMMs and DeFi [6, 7, 19, 20, 136].

Our goal will then be to formally verify this contract design in §4.4 and §4.5. We will show that the process of rigorous contract specification splits naturally into two components: the contract specification, which is a formal definition and axiomatization of the contract, and the metaspecification, which is a formal tool for reasoning about the implications and properties of the specification. Both are essential to ensure a contract's correctness in the formal setting.

4.2.1 The Issue of Fungibility in Tokenized Carbon Credits

Tokenized carbon credits, which are of growing relevance to voluntary carbon markets [46, 48, 122], 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 [49, 104, 124].

The current solution is to pool carbon credits which have similar features, such as a specific vintage or crediting methodology, and to value each pooled token equally within the pool [124]. Unfortunately, from a valuation perspective this discards the differences in constituent credits.

For example, Toucan, perhaps the most prominent provider of tokenized carbon credits [122], tokenizes carbon credits from the Verra registry as NFTs on Polygon [133]. 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.

Our goal then is to increase liquidity without ignoring individual token metadata, and we do so by removing the required one-for-one exchange rate. To that end we define 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. Thus our contract should act both as a pooling contract as well as an AMM, and any relevant properties of correctness should reflect that fact.

We now proceed to specify a structured pool contract.

4.2.2 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.

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.

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.

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 4.1.

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:

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

```
(* 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 4.1: Pseudocode of the `TRADE` entrypoint function.

giving

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

This is how trades are priced in the wild for liquidity pools of fungible tokens [7]. We call $p_s = \frac{\Delta_y}{\Delta_x}$ the *swap price*.

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

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

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

The difference $p_q - p_s$ is called the *price slippage* [136, §3.2.4]. It is important to note that p_s is always less than p_q because p_q is calculated by moving Δ_x along the tangent line from a starting point (x_0, y_0) representing the current state of the contract's funds available for trading, and p_s is calculated by moving Δ_x along $xy = k$. Since $xy = k$ is convex, moving Δ_x along the tangent line always results in a larger Δ_y than moving along $xy = k$. See Figure 4.2 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.

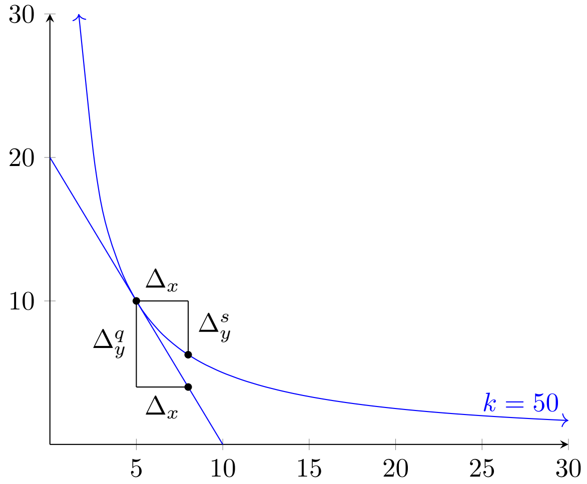

Figure 4.2: A trade of $\Delta_x = 3$ for Δ_y^q and Δ_y^s, respectively, at $k = 50$. $\Delta_y^q = p_q \Delta_x$ is the trade priced at the *quoted price* p_q and $\Delta_y^s = p_s \Delta_x$ is the trade priced at the *swap price* p_s.

In particular, this means that

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

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

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

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

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

To price a trade we begin by finding ℓ such that

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

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

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

This formula yields the quoted price of this trade as

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

as desired. The swap price, then, is

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

where

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

4.2.3 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 [7], it is not immediately obvious that the structured pool contract will do the same. To that end, we draw on work by Angeris *et al.* [6, 7], Bartoletti *et al.* [19, 20], and Xu *et al.* [136] on AMMs and DeFi, from which we derive six properties indicative of desirable market behavior from game-theoretic and economic perspectives. For each of the six properties, we give an informal definition, followed by a formal proposition and proof.

Demand Sensitivity

A trade for a given token increases its price relative to other constituent tokens, so that higher relative demand corresponds to a higher relative price. Likewise, trading one token in for another decreases the first's relative price in the pool, corresponding to slackened demand. This enforces the classical notion of supply and demand and is important to the proper functioning of an AMM, as we see in [20, §4.2].

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 (4.3) and (4.6):

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

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

Nonpathological prices

As relative prices shift through trades, a price that starts out nonzero never goes to zero or to a negative value. This is to avoid pathological behavior of zero or negative prices, and is true for standard AMMs like Uniswap [7, §2]. However, like most AMMs, prices can still get arbitrarily close to zero, so a constituent token which loses its value due to external factors can still become arbitrarily devalued within the pool. This is an important property so that the formulae which price trades never divide by zero.

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. □

Swap Rate Consistency

For a token t_x in T and for any $\Delta_x > 0$, there is no sequence of trades, beginning and ending with t_x, such that $\Delta'_x > \Delta_x$, where Δ'_x is the output quantity of the sequence of trades. Swap rate consistency means that it is never profitable to trade in a loop, *e.g.* t_x to t_y, and back to t_x, which is important so that there are never any opportunities for arbitrage internal to the pool. This is similar to the assertion that trading cost be positive [7, §2], that trading from t_x to t_y, and back to t_x not be profitable in [6, §3, §4.1].

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

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

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

Since

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

(4.10) simplifies to

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

which in turn simplifies to

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

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

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

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

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

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

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

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

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

is satisfied. But we have from (4.10) that

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

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

Zero-Impact Liquidity Change

The quoted price of trades is unaffected by depositing or withdrawing liquidity [136, §3.3.1]. Typically, for AMMs such as Uniswap [3] or Curve [2] this is implemented by requiring that liquidity providers provide liquidity in pairs such that the quoted price of the AMM does not change by depositing or withdrawing liquidity. Liquidity provision works differently for structured pools, but depositing or withdrawing liquidity still does not impact quoted prices.

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 (4.4), and as per the specification, only the `TRADE` entrypoint alters pooling exchange rates. □

Arbitrage sensitivity

If an external, demand-sensitive market prices a constituent token differently from the structured pool, a sufficiently large arbitrage transaction will equalize the prices of the external market and the structured pool, or deplete the pool. In our case, this happens because prices adapt through trades due to demand sensitivity or the pool depletes in that particular token. This is generally considered to be an important property so that prices adjust in line with supply and demand, see [20, §4.3] and [7].

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

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

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

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

Then

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

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. □

Pooled Consistency

The number of outstanding pool tokens is equal to the value, in pool tokens, of all constituent tokens held by the contract. Mathematically, the sum of all the constituent, pooled tokens, multiplied by their value in terms of pooled tokens, always equals the total number of outstanding pool tokens. This means that the pool token is never under- or over-collateralized, and is similar to standard AMMs, where the LP token is always fully backed, representing a percentage of the liquidity pool, and is encoded in the literature as *preservation of net worth* [20, §3].

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

$$\sum_{t_x} r_x x = k \tag{4.11}$$

Proof. As a base case, by the specification, at the time of contract deployment $k = 0$ and we have no pooled liquidity, so (4.11) holds trivially because $x = 0$ for all t_x. For our inductive step, consider a contract state for which (4.11) holds. If we call `DEPOSIT`, (4.11) 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 (4.11). This excess is quantified in (4.9) and remedied by adjusting r_x to r'_x as we saw before. □

4.3 Specification and Metaspecification

The contents of the previous section show us that a correct specification has two components. The first is the contract specification, an axiomatization of the contract which defines the minimal structure needed for a contract implementation to be considered, in this case, a structured pool contract. The second is the *metaspecification*, which consists of properties that justify the specification to be correct. It proves theorems about an arbitrary contract satisfying that specification, which themselves are derived from some broader theory or understanding of the execution context.

These two components are complementary, but separate. The specification should be minimal, since in practice we wish to impose as few constraints as possible on the implementation of any given smart contract and minimize any verification work needed to prove the contract correct. Conversely, the metaspecification should be as comprehensive as possible in order to understand all the properties and behaviors associated with the specification that we possibly can.

In what follows, we formalize the contract specification in ConCert as a predicate on contracts (§4.4), and then formalize the metaspecification as a list of properties which can be proved by assuming a contract that conforms to the specification into the context (§4.5). As we will see, the metaspecification informs the specification, and vice versa, not only in the design of the specification but also in its formalization.

4.4 Formal Specification: A Contract Axiomatization

The formulation of a formal specification is not generally treated systematically in formal verification. Most formal verification is done on specific implementations of contracts, proving an instance of an informal specification. The specification is typically formalized ad hoc and not abstracted as a standalone, formal object. In particular, this means that there is no obvious way to reason abstractly and formally about one or various contracts which conform to a given specification.

We wish to be more systematic. We propose a generic reasoning technique which formalizes a specification as a predicate on smart contracts, axiomatizing that contract in a formal, mathematical way. Using ConCert, we formalize the specification of a structured pool contract [125], introducing a predicate on the contract type.

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

The predicate is a conjunction of each formal property of the specification. It delineates formally what it means to be a structured pool contract, and allows us to reason about the specification's consequences by assuming some contract `C` and a proof `(is_sp : is_structured_pool C)`. We will describe the formal properties of the specification here, and summarize them at the end of the section in Table 4.1.

4.4.1 The Structured Pool Formal Specification

We now outline the formal specification of structured pools. Recall from Chapter 3 that in ConCert, a smart contract is a record type of two functions: `init`, which describes how the contract initializes, and `receive`, which gives the semantics of a call to a contract entrypoint. Contracts are also parameterized by four types: `Setup`, the type to initialize, `Msg`, the entrypoint type, `State`, the storage type, and `Error`, the error type.

To formalize a contract specification, we first introduce typeclasses which characterize the contract types. We then specify how the contract must initialize. Finally, we specify contract calls by individually specifying each entrypoint.

4.4.2 Typeclasses to Characterize Contract Types

Of the four contract types, here we will look at the specification of `State`, the storage type, and `Msg`, the entrypoint type, of the structured pool contract. The formalized typeclasses of the remaining two contract types can be found in Appendix A.1.1.

First, the storage type. The informal specification states that the contract storage must contain:

- the exchange rates for each constituent token,
- the quantity of constituent tokens held in the pool,
- the address of the pool token, and
- the number of outstanding pool tokens.

We can specify this by using a typeclass, which simply requires that the storage type `T` of a structured pool contract have functions which reveal each of these data points.

- `stor_rates : T -> FMap token exchange_rate`
- `stor_tokens_held : T -> FMap token N`

- `stor_pool_token : T -> token`

- `stor_outstanding_tokens : T -> N`

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

Listing 4.1: The typeclass characterizing the storage type of a structured pool contract.

Moving on, consider `Msg`, the entrypoint type. There must be at least three entrypoints: `POOL`, `UNPOOL`, and `TRADE`. Our specification first identifies the minimal amount of information needed to call each of these entrypoints by defining three types:

- `pool_data`, the payload type for the `POOL` entrypoint,

- `unpool_data`, the payload type for the `UNPOOL` entrypoint, and

- `trade_data`, the payload type for the `TRADE` entrypoint.

We then specify that the entrypoint type have at least three (but possibly more) entrypoints by a typeclass which requires the following functions into the entrypoint type `T`:

- `pool : pool_data -> T`

- `unpool : unpool_data -> T`

- `trade : trade_data -> T`

- `other : other_entrypoint -> option T`

The function `other`, going from some generic type `other_entrypoint` to `option T`, allows an implementation of the structured pool contract to optionally contain more entrypoints than the required three.

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

Listing 4.2: The typeclass characterizing the entrypoint type of a structured pool contract.

When reasoning about a contract in ConCert, one must be able to reason about all entrypoints. Since we allow optionally for more than three entrypoints via the `Msg_Spec` typeclass, we need to include in the specification a stipulation that for any type `T` satisfying `Msg_Spec`, all inhabitants of `T` must be able to be written in terms of these four functions. That is, for any `m:T`, `m` is either:

- `pool p`, for some `p`,
- `unpool u`, for some `u`,
- `trade t`, for some `t`, or
- `Some m` equals `other o`, for some `o`.

In particular, this allows us to mimic induction over an arbitrary inhabitant of the message type, despite that type not being explicitly defined as an inductive type. We do so by formally encoding the following proposition into the contract specification:

Proposition 1. *For all* `m:Msg`*, one of the following holds:*

1. `m = pool p`*, for some* `p`
2. `m = unpool u`*, for some* `u`
3. `m = trade t`*, for some* `t`
4. `Some m = other o`*, for some* `o`*.*

```
Definition msg_destruct (contract:Contract Setup Msg State Error) :=
    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 formalization of Proposition 1, which is used to destruct inhabitants of the entrypoint type.

See Appendix A.1.1 for the formalized typeclasses characterizing `Setup`, the setup type, and `Error`, the error type.

4.4.3 Specifying Contract Initialization

We now specify the structured pool contract's `init` function, which governs how it initializes.

A structured pool contract must initialize with positive exchange rates, with no pooled tokens, and with no outstanding pool tokens. We encode each of these notions in three properties of the specification, formalized virtually identically to the informal statements made in the informal specification. They are as follows.

1. `initialized_with_positive_rates`,
2. `initialized_with_zero_balance`, and
3. `initialized_with_zero_outstanding`

Since rates are encoded in a map from tokens to exchange rates as we saw in 4.4.2, the first of these, `initialized_with_positive_rates`, stipulates that for all initialized contract states `cstate`, a rate `r` corresponding to a token `t` must satisfy `r > 0`. The second, `initialized_with_zero_balance`, gives us that the balance of any token `t` in storage initializes to `0`. The third, `initialized_with_zero_outstanding`, stipulates that at the time of initialization, there be no outstanding pool tokens.

This covers what is explicitly mentioned in the informal specification, but also specifies that the data given in the `Setup` type is the same data used to initialize the rates and set the data for the pool token.

4.4.4 Specifying Each Contract Entrypoint

There are twenty-four properties of the full entrypoint specification, encoded as propositions. We will look at a few key properties here. For the full list of propositions, see Table 4.1.

Firstly, let us look at the specification of the `POOL` and `UNPOOL` entrypoints. One proposition in the specification of the `POOL` entrypoint is `pool_increases_tokens_held`, which specifies that when a token is pooled via a successful call to the `POOL` entrypoint, the balance of tokens held in the pool goes up in that token and the balance of all other tokens stays constant. This is given by the following proposition, formalized.

Proposition 2. *Consider a contract* `contract`*. Suppose that for some contract state* `cstate`*, chain, and contract call context, the* `POOL` *entrypoint of* `contract` *is called successfully with some payload* `msg_payload`*. Then in the updated contract state* `cstate'`*, the balance of the token pooled, given in the* *map* (stor_tokens_held cstate’)*, is greater than its balance in the previous state, given in the map* (stor_tokens_held cstate)*. The difference between the two is precisely the quantity pooled, and the balance for all other tokens stays constant between the two states.*

```
Definition pool_increases_tokens_held
  (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate’ acts,
    (* If the call to POOL was successful, *)
    receive contract chain ctx cstate (Some (pool msg_payload)) =
    Ok(cstate’, acts) ->
    (* then in the new state cstate’, tokens_held has
       increased at the token pooled *)
    let token := msg_payload.(token_pooled) in
    let qty := msg_payload.(qty_pooled) in
    let old_bal := get_bal token (stor_tokens_held cstate) in
    let new_bal := get_bal token (stor_tokens_held cstate’) in
    new_bal = old_bal + qty /\
    (* and tokens_held stays the same for all other tokens. *)
    forall t,
    t <> token ->
    get_bal t (stor_tokens_held cstate) =
    get_bal t (stor_tokens_held cstate’).
```

Listing 4.4: The formalization of Proposition 2, which describes pre- and post-conditions of calling the POOL entrypoint.

The POOL and UNPOOL entrypoints are fully characterized as entrypoints by a set of propositions of a similar form to this one. These propositions describe what happens when tokens are (un)pooled, what transactions are emitted from those entrypoint calls, and how the storage gets updated.

Moving on to the specification of the TRADE entrypoint, we introduce two auxiliary functions: calc_delta_y, which calculates the output of a trade, typically written Δ_y in the literature, and calc_rx’, which calculates how exchange rates (and by implication, prices) update in response to trading activity. Structured pool contracts simulate trading along a convex curve, the most common of these being the Uniswap V1 curve $xy = k$ (see Figure 4.2).

Rather than require one implementation or curve, we simply specify that whatever functions are present in the implementation conform to a few requirements true of trades along any convex curve. Since in structured pools, the quoted price of any trade from a token t_x to a token t_y is given by the quotient of their rates

$$\frac{r_x}{r_y},$$

we require that the pricing and slippage (the difference between the quoted price of a trade and the price at which the trade is executed) of any implementation of calc_delta_y mimic the pricing and slippage we get by trading along $xy = k$. For structured pools, this translates to a technical requirement that

$$r_y \Delta_y \leq r_x \Delta_x$$

for all trades from t_x to t_y of quantity Δ_x, where r_x is the exchange rate of t_x and r_y is the exchange rate of t_y. To express this in the specification, we formalize the following proposition.

Proposition 3. *Consider tokens t_x and t_y, with exchange rates r_x and r_y (resp.) and pooled balances x and y (resp.). For all trades from t_x to t_y of quantity Δ_x, the following inequality holds, where Δ_y is the quantity traded out which is calculated by* `calc_delta_y`*:*

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

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

Listing 4.5: The formalization of Proposition 3, which characterizes trade slippage.

We don't require any particular implementation of `calc_delta_y`, so long as it conforms to Proposition 3 (as well as the other relevant propositions of the specification).

Also included in the specification is a proposition that trades are calculated using the function `calc_delta_y`, formalized as follows.

Proposition 4. *Consider a contract* `contract`*. Suppose that for some contract state* `cstate`*, chain, and contract call context, the* `TRADE` *entrypoint is successfully called with some payload* `msg_payload`*, where* `delta_x` *is the quantity traded in. Then for the updated state* `cstate'`*:*

1. *The balance of the token traded in increases by* `delta_x` *from the previous state* `cstate` *to the updated state* `cstate'`*, and*

2. *The balance of the token traded out decreases by* `delta_y` *from the previous state* `cstate` *to the updated state* `cstate'`*, where* `delta_y` *is the quantity calculated by* `calc_delta_y`*.*

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

Listing 4.6: The formalization of Proposition 4, which requires that trades be priced with the function `calc_delta_y`.

The `TRADE` entrypoint is fully characterized as an entrypoint by a set of propositions similar to this one. These describe what happens when tokens are traded, what transactions are emitted from that entrypoint call, and how the storage updates.

Finally, to finish specifying the entrypoints we need to specify required behavior of any other entrypoint aside from `POOL`, `UNPOOL`, or `TRADE`. We simply specify that calling any other entrypoints does not affect exchange rates, token balances, or the outstanding tokens when called.

Proposition 5. *Consider a contract* `contract`*. Suppose that for some contract state* `cstate`*, chain, and contract call context, any entrypoint other than* `POOL`*,* `UNPOOL`*, or* `TRADE` *is successfully called with payload* `o`*. Then for the updated state* `cstate'` *the following hold:*

1. *the exchange rates remain constant, meaning*

   `(stor_rates cstate) = (stor_rates cstate')`

2. *the token balances remain constant, meaning*

   `(stor_tokens_held cstate) = (stor_tokens_held cstate')`

3. *the number of outstanding tokens remains constant, meaning*

   `(stor_outstanding_tokens cstate) = (stor_outstanding_tokens cstate').`

	Propositions of the Specification	**Summary**
Preconditions	none_fails, msg_destruct	A contract call fails if the message payload is empty, and the message type must have at least the three entrypoints specified.
Specification of POOL	pool_entrypoint_check pool_emits_txns pool_increases_tokens_held pool_rates_unchanged pool_outstanding	Calling POOL emits the correct transactions and alters storage correctly, and fails if someone attempts to pool a nonexistent token.
Specification of UNPOOL	unpool_entrypoint_check unpool_entrypoint_check_2 unpool_emits_txns unpool_decreases_tokens_held unpool_rates_unchanged unpool_outstanding	Calling UNPOOL emits the correct transactions and alters storage correctly, and fails if someone attempts to unpool a nonexistent token.
Specification of TRADE	trade_entrypoint_check trade_entrypoint_check_2 trade_pricing_formula trade_update_rates trade_update_rates_formula trade_emits_transfers trade_tokens_held_update trade_outstanding_update trade_pricing trade_amounts_nonnegative	Calling TRADE emits the correct transactions and updates the storage correctly; it prices trades and updates rates correctly; and it fails if someone attempts to trade a nonexistent token or a token with insufficient contract balance.
Specification of OTHER	other_rates_unchanged other_balances_unchanged other_outstanding_unchanged	Any other entrypoint must not alter exchange rates, token balances, or outstanding pool tokens.
calc_delta_y and calc_rx'	rate_decrease, rates_balance, rates_balance_2, trade_slippage, trade_slippage_2, arbitrage_lt, arbitrage_gt	Axiomatizes trading along a convex curve.
Contract Initialization	initialized_with_positive_rates initialized_with_zero_balance initialized_with_zero_outstanding initialized_with_init_rates initialized_with_pool_token	The contract initializes with positive rates, zero pooled balance, zero pool tokens outstanding, and using the initialization data.

Table 4.1: The propositions which constitute the formal specification of a structured pool contract.

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

Listing 4.7: The formalization of Item 1 of Proposition 5, which requires that the number of outstanding tokens remain constant through a successful call to any entrypoint other than `POOL`, `UNPOOL`, or `TRADE`.

4.4.5 The Formal Specification as a Predicate on Contracts

Taken together, these propositions and typeclasses are the axiomatized definition of a structured pool contract, and we can reason about the specification if we assume the existence of some contract

```
contract : Contract Setup Msg State Error
```

such that each of `Setup`, `Msg`, `State`, and `Error` conform to their respective typeclasses `Setup_Spec`, `Msg_Spec`, `State_Spec`, and `Error_Spec`, and such that `contract` conforms to each of the properties defined in the specification.

To do so, we amalgamate all the properties into a predicate on contracts, which is a function

```
is_structured_pool : forall (C : Contract Seetup Msg State Error), Prop.
```

See Table 4.1 for a summary. The full, formal statement of the `is_structured_pool` predicate can be found in Appendix A.1.3.

4.5 Formal Metaspecification

We now turn to give a formal treatment of the structured pool metaspecification. As we saw in §4.2.3, a metaspecification is critical to be sure that our specification is correct in the context of some underlying theory. Thus if we wish to formally verify a contract to be correct, we must verify its formal specification to also be correct. The formalization here has the same economic implications of correctness as we saw in the unformalized metaspecification of §4.2.3, but as we will see in the formal setting it has additional advantages that help us prove the *formalization* of the specification to be correct.

In particular, the reader may recall that the unformalized specification of §4.2.2 uses rational-number arithmetic in the properties of the specification and in the statements and proofs of these six properties of the metaspecification. Using rationals or reals is common, especially with financial contracts like AMMs or lending pools, *e.g.* [18, 20]. However, we have the issue that smart contracts typically use only natural-number arithmetic, and so we need some way of ensuring that the transition from rational to natural-number arithmetic does not compromise the economic properties the contract is meant to satisfy.

4.5.1 Formalizing the Metaspecification

We begin by introducing an arbitrary structured pool contract into our Coq context, formalized as follows.

```
Context { contract : Contract Setup Msg State Error }
        { is_sp : is_structured_pool contract }.
```

Listing 4.8: Introduce into the context an arbitrary contract `contract` which conforms to the structured pools specification.

We have formalized and proved all six of these properties in ConCert (see Appendix A.2), but will focus our discussion on those properties which illustrate how the formalization of the metaspecification helps to derive properties of the specification necessary for a safe transition from rational to natural-number arithmetic.

First, consider Demand Sensitivity (Property 1), which is the property that an individual token's exchange rates decrease relative to other tokens with slackened demand, and increase with rising demand.

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.*

We write the formalized theorem first in prose, and then give the formalized Coq code.

Theorem 1 (Demand Sensitivity, Formalized)**.** *Consider a structured pool contract* `contract` *with state* `cstate`*. Furthermore, consider tokens* `t_x` *and* `t_y`*, rates* `r_x` *and* `r_y`*, and quantities* `x` *and* `y`*, where* `t_x` *is a token with nonzero pooled liquidity* `x` *and with rate* `r_x > 0`*, and* `t_y` *is a token with nonzero pooled liquidity* `y` *and with rate* `r_y > 0`*. In a trade* `t_x` *to* `t_y` *(a successful call to the contract's* `TRADE` *entrypoint from* `t_x` *to* `t_y` *with* `t_x <> t_y`*), as* `r_x` *is updated to* `r_x'`*:*

1. `r_x` *decreases relative to all rates* `r_z`*, for* `t_z <> t_x`*, and*

2. `r_y` *strictly increases relative to* `r_x`*.*

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

Listing 4.9: Theorem 1, the formalized statement of Demand Sensitivity (Property 1).

To see the first issue in the transition from rational to natural-number arithmetic, note first that the unformalized version of Demand Sensitivity (Property 1) uses a notion of “relative increase” and “relative decrease,” which can be understood easily in mathematical terms, but which needs to be encoded somehow formally. These two notions of natural numbers are defined by the following functions, used in the formalization in Listing 4.9. They state that as variables x, y, and z change to x', y', and z' respectively, x increases relative to y if $y - x \leq y' - x'$, and x decreases relative to z if $z - x \leq z' - x'$ (see Listing 4.10).

```
Definition rel_incr (y x y' x' : N) :=
  ((Z.of_N y) - (Z.of_N x) <= (Z.of_N y') - (Z.of_N x'))%Z.

Definition rel_decr (x z x' z' : N) :=
  ((Z.of_N z) - (Z.of_N x) <= (Z.of_N z') - (Z.of_N x'))%Z.
```

Listing 4.10: The formal notion of relative increase and decrease betwen natural numbers.

The careful reader will notice that Property 1 states that the relative decrease is strict: For an exchange rate r_x updated to r'_x, $r'_x < r_x$. In contrast, the inequality in Theorem 1 is not strict: `r_x' <= r_x`. This is because the informal specification uses rational arithmetic in all calculations, but in smart contracts do arithmetic with natural numbers. Thus if `r_x' < r_x`, meaning the rate update is strict for all trades, then after a finite number of trades the rate `r_x` could update to `0`, contradicting Nonpathological Prices (Property 2).

Let us look at the formalization of Pooled Consistency (Property 6), which states that the total value of pooled tokens, calculated in terms of pool tokens, is always the same as the total number of outstanding pool tokens.

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

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

In what follows, the function `tokens_to_values` helps to formalize the sum in Property 6 by taking all tokens with a nontrivial exchange rate and multiplies the contract’s pooled balance in that token by its exchange rate. We then fold over this list to take the sum in the formal statement.

Theorem 6 (Pooled Consistency, Formalized)**.** *Consider a structured pool contract* `contract` *with state* `cstate`*. Then the 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.*

```
Theorem pooled_consistency bstate caddr :
    (* Consider a reachable bstate, with contract deployed at caddr ... *)
    reachable bstate ->
    env_contracts bstate caddr = Some (contract : WeakContract) ->
    exists (cstate : State),
    (* ... with state cstate. *)
    contract_state bstate caddr = Some cstate /\
    (* Then the sum over all tokens of rates * (qty held) equals
        the total number of outstanding tokens. *)
    suml (tokens_to_values
      (stor_rates cstate)
      (stor_tokens_held cstate)) =
    (stor_outstanding_tokens cstate).
```

Listing 4.11: The formalization of Property 6, which requires that 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.

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

Listing 4.12: The tokens_to_values function which produces a list of token balances multiplied by their exchange rates.

From the formal proof of Pooled Consistency (Theorem 6) we derived the property rates_balance, which stipulates that the calculations for pooling and unpooling have to be inverses. That is, if one pools tokens and then unpools the output of that transaction, they should end up with the same amount of tokens as they started with. This is an implicit property of the informal specification because it uses rational exchange rates and their inverses to pool and unpool tokens.

Aside from Pooled Consistency, two other formal proofs helped characterize the change from rational to natural-number arithmetic. The formal proof of Swap Rate Consistency (Property 3) showed that another strict inequality had to be relaxed from the informal specification. The formal proof of Arbitrage Sensitivity (Property 5) showed that prices must be able to range in the open interval $(0, \infty)$. This is a property of trading along a convex curve, and is encoded as arbitrage_lt and arbitrage_gt in the formal specification.

Formalizing the metaspecification also forced us to define the invariants on all entrypoints other than POOL, UNPOOL, or TRADE that we saw in §4.4.4. Specifically, these are Nonpathological Prices (Property 2) and Pooled Consistency (Property 6), the results which required reasoning about arbitrary contract calls. Like other financial contract specifications we saw earlier [145, 40], the structured pool specification is designed to be minimal and explicitly leaves out other entrypoints. Now, because of the metaspecification, we have a specification that includes arbitrary entrypoints aside from the three explicitly specified.

4.5.2 Discussion

The complexity and nuance of proving these six results is an indaction of how nontrivial the problem of correct economic specification is.

We can observe a few notable benefits of having formalized the metaspecification.

1. The formalization made it clear which properties of rational arithmetic must continue to hold for an implementation using natural-number arithmetic so that key economic behaviors of the specified contract remain intact. Specifically, these are the conditions such that Demand Sensitivity (Property 1), Swap Rate Consistency (Property 3), Arbitrage Sensitivity (Property 5), and Pooled Consistency (Property 6) hold, which are the properties of `calc_delta_y` and `calc_rx'` we see in Table 4.1. Because every instance of rational or natural-number arithmetic in the specification is designed to satisfy the economic properties of the metaspecification, these are only discoverable through the formal metaspecification.

2. Similarly, by formalizing the metaspecification we discovered assumptions about the specified and unspecified entrypoints that were implicit to the informal specification but not obvious. Now, the specification is fully precise and unambiguous, and justified to be correct by the metaspecification.

Think back to the costly attacks on BeanStalk and Mango Markets from §1.2.1. The formalization presented here shows that reasoning about a specification's economic properties, including the pathological contract behavior exploited by the attackers, is highly nontrivial. The fact that virtually all contracts are specified and deployed, only reasoning informally at best about a specification's economic properties, points to the potential source of many of these vulnerabilities.

Furthermore, economic bugs can be subtle. We just saw that three of the properties of the metaspecification—designed to prevent pathological economic behavior—would not hold if the transition from rational to natural-number arithmetic wasn't done correctly. Furthermore, the conditions for a correct transition were nontrivial and discovered only by verifying the specification correct with regards to the metaspecification.

4.6 Conclusion

The goal of this chapter was to formally develop the notion of correctness of a specification of a financial smart contract from the perspective of its economic properties. We argued that rigorous specification of any financial smart contract, because it must have economic intent, must be defined using precise definitions and accompanied by theorems which prove it to be correct. Furthermore, these theorems should be couched in substantial theoretical and practical work on the behavior of financial smart contracts.

This gave rise naturally to the separation of a contract specification from its metaspecification. A metaspecification is a specification of a specification, and consists of properties which prove the specification to be correct within the context of a theory. It does so by proving that any contract conforming to the specification exhibits properties which characterize correct economic behavior.

Introducing the specification and metaspecification into the formal setting required defining a predicate on smart contracts, which is the contract specification, and then formally reasoning about an arbitrary contract which has a proof of the predicate corresponding to the specification. This allowed us to keep the specification as concise as possible so as to minimize the work required to formally verify a contract to be correct with regards to the specification, while still inheriting all the desirable economic behavior proved about in the metaspecification.

Finally, we showed that the metaspecification not only improves the rigor of the specification itself in terms of its ability to successfully capture economic meta properties, but also to prove the formalization itself to be correct. As there are always choices that have to be made when both designing and formalizing a contract specification, the metaspecification can help ensure that pathological contract behavior is not introduced because of a poorly formed or formalized specification.

We conclude with a note that the theories from which we derived the properties of our metaspecification [6, 7, 19, 20, 136] were not themselves formalized, which meant that the properties of the metaspecification were derived from the theories by hand and not formally. Future work on this topic could include formalizing said theories and using them to rigorously and systematically derive properties of financial contracts metaspecifications.

Chapter 5

Contract Upgradeability

The meta properties that we target in this chapter are upgradeability properties of financial smart contracts. As we saw in §1.2.2, poorly specified contract upgrades can introduce costly vulnerabilities. In this chapter we rigorously develop the notion of correctness of a specification of both individual contract upgrades, as well as contract upgradeability, articulating meta properties as properties of one contract in relation to those of another.

To this end, we introduce a theoretical tool called a *contract morphism*, which is a formal mechanism to structurally relate smart contracts in ConCert. We show how morphisms can be used in proof and specification, targeting upgradeable contracts in particular, and how they can be used to mathematically characterize the bounds of a contract's upgradeability.

This chapter is organized as follows. In §5.1, we motivate contract morphisms by discussing the problem of formally specifying and verifying contract upgrades. In §5.2, we introduce contract morphisms. In §5.3, we show how contract morphisms can be used with ConCert's contract induction tactic. In §5.4, we discuss strategies in proof and specification using contract morphisms. In §5.5 we mathematically characterize contract upgrades using contract morphisms. In §5.6 we conclude. Each section contains various code snippets; Appendix B mirrors section headings and gives a more complete version of the code from which the snippets are taken.

5.1 Contract Upgrades

Like the economic properties of a contract specification, contract upgradeability involves complex contract behavior which can be difficult to specify correctly. Blockchains do not generally feature built-in methods for upgrading a smart contract once it has been deployed. Instead, if one wishes to upgrade a smart contract, one has to encode an upgradeability framework into the contract before deployment. As we saw in §1.2.2, this is hard to do well and can result in costly bugs.

The alternative to encoding upgradeability into a smart contract is to upgrade via a hard fork. This involves deploying a new contract and convincing users to migrate to the new one, for example with each of the Uniswap upgrades [4, 70]. Especially for financial smart contracts such as AMMs which rely on liquidity providers, this transition can be expensive and difficult.

It is perhaps an indication of the complexity of deploying safe, upgradeable contracts that many financial smart contracts use hard forks. Best practices [142, 106] and established upgrade frameworks [95, 141] are not enough to prevent vulnerabilities. Like the economic properties of Chapter 4, contract upgradeability features nontrivial meta properties, so writing a correct specification is also nontrivial.

5.1.1 Specifying Upgradeability: The Diamond Framework

For example, consider the EIP-2535 Diamond upgrade framework [95], which is a specification of a popular, generic, and flexible upgrade protocol for Ethereum smart contracts. The specification describes contract storage and entrypoints, defining a system of contracts that all interact with each other. Reading the specification, one can convince one's self that it does in fact specify an upgradeable contract. However, there is no property of the specification which precisely characterizes what it means to be upgradeable, much less *how* upgradeable or mutable this defined structure is.

Rather than giving a precise notion of upgradeability, the intuition behind what it means to be upgradeable, including the bounds of a contract's upgradeability, are communicated in the Diamond framework through pictures, diagrams, the motivation section of the specification, and the analogy of a diamond in naming conventions for different parts of the specification. Each of these are rhetorical tools to communicate what it means to be "upgradeable" but they are not mathematically or technically rigorous. In particular, if we were to formally verify this standard, we would have no independent, mathematical notion to verify the claim that this is, indeed, upgradeable in some precise sense.

Thus, just as with the economic properties implied by a specification that we saw in Chapter 4, we see that the specification of an upgradeable contract tries to target some notion of upgradeability which we can understand inuitively; however, whether or not the specification succeeds in doing so is a matter of intuition at best. If we can articulate a precise notion of upgradeability, then we can verify a contract specification correct with regards to a metaspecification that includes these upgradeability properties.

How then do we mathematically capture the notion of upgradeability? We will approach the problem as follows. An upgradeable contract is made up of two constituent parts: the part governing contract upgrades, which is a skeleton that remains constant no matter the state of the contract, and the part corresponding to a specific contract version, which can change through upgrades. We might describe the logic governing upgrades as a contract in its own right, a metacontract of sorts which governs the data and versioning of the upgradeable contract. The contract version corresponding to any given state is one of many possibilites, which we might describe with a family of contracts which parameterizes all possible forms the upgradeable contract can take.

We will make this fully precise using contract morphisms, giving a formal decomposition of an upgradeable smart contract into a *base contract*—its immutable part—and a family of *version contracts*—its mutable part. The base contract contains the upgradeability framework, and the family of *version contracts* contains the contract functionality which can be upgraded.

5.1.2 Evolving Specifications Through Upgrades

Another aspect which merits our attention is that upgrading a contract, whether by hard fork or through built-in upgradeability, involves specifying the new contract version. How do we know the specification of the new contract accurately reflects what we wish to happen in the upgrade?

Consider a contract upgrade from the perspective of a formal specification. Generally speaking, we upgrade with a goal which relates to the previous contract version, whether it be to patch a bug, add functionality, or improve contract features. Indeed, the new specification likely relates to the old: the new should eliminate a vulnerability of the old, be backwards compatible while adding functionality to the old, or make improvements on the old, for example to be more gas-efficient.

The actual intention of the upgrade, and therefore its specification, might then be best written in relation to the old contract. Informally, if the upgrade is to patch a bug, we might specify that the new contract behave identically to the old, except that it patches the buggy functionality. If the upgrade adds functionality, we might specify that new functionality as normal, and then specify that all the functionality of the old contract still hold for the new—or specify exactly how it changes in relation to the old. Finally, if we are optimizing, we might specify that the new contract behave exactly like the old, except that it improve in some metric like gas efficiency or precision of some calculation.

Of course, in practice upgrades are not 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. However, we run into the same issue that we saw with the economic properties implied by a specification in Chapter 4: such a specification tries to target what is meant by an upgrade, which is typically described informally by the difference between old and new contract versions; whether or not the specification succeeds is at best a matter of intuition. In the case of a financial contract, how can we be sure that small changes to the specification do not corrupt its correctness with regards to a metaspecification? As we saw in §1.2.2, it is not straightforward to get these right and can lead to costly exploits.

If we are able to formally compare the old and new contracts and their specifications, we can verify the specification of a contract upgrade correct via a metaspecification that articulates how the new contract should relate to the old, just as we have done informally above. We will show in §5.4 that a contract upgrade can be specified by formally relating the new contract version to the previous version by using contract morphisms.

5.1.3 Related Work

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

The first is work by Antonino *et al.* [14], which seeks to address vulnerabilities in contract upgrades by proposing a novel *systematic deployment framework* that requires contracts to be formally verified before they are deployed or upgraded. The framework relies on a *trusted deployer*, which is an off-chain service that vets contract creations and updates. Under this framework, at deployment of an upgradeable contract one can specify the bounds of upgradeability by requiring from the trusted deployer that certain invariants hold through all upgrades.

The second is work by Dickerson *et al.* [47], which proposes a paradigm shift for blockchains and smart contracts so that smart contracts can carry with them proofs of correctness. These are called *proof-carrying smart contracts*. These contracts can be upgradeable, so long as the upgraded contract carries a proof that it conforms to a specification by the original deployed contract. This work relates to previous work outside the context of blockchains on dynamic software updating [72] and proof-carrying code [99].

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

Our approach is distinct. It requires no trusted third party, nor a fundamental paradigm shift into smart contracts that carry proofs. Rather, using contract morphisms we can mathematically characterize the bounds of a contract's upgradeability, and rigorously specify upgraded contracts in relation to older versions. If one wishes to impose invariants that cannot be changed through upgrades, one can do so by reasoning about an upgradeable contract's decomposition into its base contract and version contracts.

5.2 Contract Morphisms

We now move on to introduce contract morphisms, which are a formal mechanism to structurally compare smart contracts in ConCert. The goal of contract morphisms is to be able to reason about a contract and its specification in relation to another contract and its specification. As we will see, contract morphisms can be used to prove and to specify properties of one contract in terms of another (§5.4), as well as to decompose an upgradeable contract into its mutable and immutable parts (§5.5).

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

```
init : Chain -> ContractCallContext -> Setup -> result State Error.
```

Listing 5.1: The `init` function of a smart contract.

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

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

Listing 5.2: The `receive` function of a smart contract.

Now consider contracts

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

with respective init and receive functions `init1`, `init2` and `receive1`, `receive2`. We define a morphism of contracts, which we write `f : ContractMorphism C1 C2`, as a natural transformation of init and receive functions,

`f_init : init1 -> init2` and `f_recv : receive1 -> receive2`.

In other words, we define a function from inputs to `init1` to inputs to `init2`, and from outputs of `init1` to outputs of `init2`, such that if we transform inputs to `init1` and then take their image under `init2`, we get the same result as if we had first taken the image of the inputs under `init1` and then transformed the outputs to those of `init2`. We do the same for `receive1` and `receive2`. See Figure 5.1 for a graphical representation.

$$\begin{tikzcd} I_i^1 \arrow[r, "\texttt{f\_init}"] \arrow[d, "\texttt{init}"'] & I_i^2 \arrow[d, "\texttt{init}'"] \\ I_o^1 \arrow[r, "\texttt{f\_init}"] & I_o^2 \end{tikzcd} \qquad \begin{tikzcd} R_i^1 \arrow[r, "\texttt{f\_recv}"] \arrow[d, "\texttt{receive}"'] & R_i^2 \arrow[d, "\texttt{receive}'"] \\ R_o^1 \arrow[r, "\texttt{f\_recv}"] & R_o^2 \end{tikzcd}$$

Figure 5.1: The init and receive components of a contract morphism, where I_i^1 and I_o^1 (resp. I_i^2 and I_o^2) are the input and output types of `C1.(init)` (resp. `C2.(init)`), and R_i^1 and R_o^1 (resp. R_i^2 and R_o^2) are the input and output types of `C1.(receive)` (resp. `C2.(receive)`). Both diagrams *commute*, meaning that starting from the upper-left corner, moving horizontally first and then vertically yields the same result as moving verticaly first and then horizontally.

We will derive the natural transformations between init and receive functions from functions on the contract's parameterizing types, requiring four *component functions*:

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

such that the following two coherence conditions hold:

- For all `c`, `ctx`, and `s`, transforming `(C1.(init) c ctx s)` using `state_morph` and `error_morph` gives us `(C2.(init) c ctx (setup_morph s))`; and
- For all `c`, `ctx`, `st`, and `op_msg`, transforming `(C1.(receive) c ctx st op_msg)` with `state_morph` and `error_morph` gives us

  `C2.(receive) c ctx (state_morph st) (option_map msg_morph op_msg).`

See the definition of the type `ContractMorphism C1 C2` in Coq below.

```
Record ContractMorphism
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :=
    build_contract_morphism {
        (* the components of a morphism f *)
        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 5.3: The definition of contract morphisms in ConCert.

Here, `result_functor` is the following function, which for types `T` and `E`, simply takes functions of type `T -> T'` and `E -> E'` and returns a function of type `result T E -> result T' E'`.

```
Definition result_functor {T T' E E' : Type} :
    (T -> T') -> (E -> E') -> result T E -> result T' E' :=
    fun (f_t : T -> T') (f_e : E -> E') (res : result T E) =>
    match res with | Ok t => Ok (f_t t) | Err e => Err (f_e e) end.
```

Listing 5.4: The result functor to transform results of init and receive.

Example 5.2.1 (Hello, world!)**.** Consider a contract C1 which keeps a nat in storage, which can be incremented by calling the contract's unique entrypoint incr. Consider another contract C2 which has the same storage type as C1 and two entrypoints: the first, incr' which behaves identically to incr, and the second, reset, which can be called to reset the nat in storage to 0.

We can construct a contract morphism f : ContractMorphism C1 C2 by simply sending messages to the incr entrypoint of C1 to the incr' entrypoint of C2, and using the identity function for all other component functions.

```
Definition msg_morph (e : entrypoint) : entrypoint' :=
    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.
```

Listing 5.5: The component functions of a morphism from C1 to C2 which sends the incr entrypoint of C1 to the incr' entrypoint of C2.

After proving the coherence results, init_coherence and recv_coherence, we can construct a morphism.

```
Definition f : ContractMorphism C1 C2 :=
    build_contract_morphism C1 C2 setup_morph msg_morph state_morph error_morph
        init_coherence recv_coherence.
```

Listing 5.6: A morphism from C1 to C2 which sends messages to the incr entrypoint of C1 to messages to the incr' entrypoint of C2.

Example 5.2.2 (Identity Morphism)**.** One important contract morphism is the identity morphism id_cm, which is defined for any contract C and inhabits ContractMorphism C C.

```
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 ;
    |}.
```

Listing 5.7: The identity contract morphism id_cm C defined for any contract C.

The associated coherence results are proved trivially by reflexivity.

```
Lemma init_coherence_id (C : Contract Setup Msg State Error) :
    forall c ctx s,
    result_functor id id (init C c ctx s) =
    init C c ctx s.
```

Listing 5.8: The init coherence lemma for the identity morphism which is proved by reflexivity.

```
Lemma recv_coherence_id (C : Contract Setup Msg State Error) :
    forall c ctx st op_msg,
    result_functor
        (fun ’(st, l) => (id st, l)) id
        (receive C c ctx st op_msg) =
    receive C c ctx (id st) (option_map id op_msg).
```

Listing 5.9: The receive coherence lemma for the identity morphism which is proved by reflexivity.

Example 5.2.3 (Injective, surjective morphisms)**.** Injective and surjective functions are ubiquitous in mathematics. Injective functions, also called embeddings, are those which are fully structure-preserving. We can give a formal definition for injectivity here:

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

Listing 5.10: A formal definition of injectivity.

In other words, injective functions send distinct terms to distinct terms.

Surjective functions, also called quotients, are those for which every term in the function's codomain has a preimage. Rather than being structure-preserving, surjective functions are often (but not always) structure-*compressing*. We can give a formal definition for surjectivity here:

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

Listing 5.11: A formal definition of surjectivity.

We wish to define the analogues for contract morphisms. Contract embeddings, or injections, will be those which are fully structure-preserving: if `f : ContractMorphism C1 C2` is an embedding, then we can think of `C2` as having a copy of `C1` in it. This will become relevant later, *e.g.* in Example 5.4.2.

A contract morphism `f`, then, is an *embedding* if all of its component functions are injective. Formalized in ConCert, we have a predicate `is_inj_cm` which is defined as follows in Listing 5.12.

```
Definition is_inj_cm (f : ContractMorphism C1 C2) : Prop :=
    is_inj (setup_morph C1 C2 f) /\
    is_inj (msg_morph   C1 C2 f) /\
    is_inj (state_morph C1 C2 f) /\
    is_inj (error_morph C1 C2 f).
```

Listing 5.12: An embedding of contracts is a contract morphism whose component morphisms are injective.

Likewise, we can define contract quotients, or surjections. These will be contract morphisms that compress structure, and will become relevant *e.g.* in Example 5.4.1 as a tool to categorize contract behavior.

A contract morphism `f`, then, is a *quotient of contracts* if all of its component functions are surjective. Formalized in ConCert, we have a predicate `is_surj_cm` which is defined as follows in Listing 5.13.

```
Definition is_surj_cm (f : ContractMorphism C1 C2) : Prop :=
    is_surj (setup_morph C1 C2 f) /\
    is_surj (msg_morph   C1 C2 f) /\
    is_surj (state_morph C1 C2 f) /\
    is_surj (error_morph C1 C2 f).
```

Listing 5.13: A quotient of contracts is a contract morphism whose component morphisms are surjective.

Example 5.2.4 (Equality of Morphisms)**.** Given two morphisms

`f g :  ContractMorphism C1 C2,`

we might ask ourselves whether or not they are equal. This will be relevant *e.g.* in Example 5.2.7 and Chapter 6 when we introduce contract isomorphisms and equivalences.

By assuming proof irrelevance, we get `f = g` if and only if each of the component functions are equal via function extensionality. This is because the proofs of coherence for `f` have the same type as those for `g` if their component functions are equal.

```
Lemma eq_cm_iff :
    forall (f g : ContractMorphism C1 C2),
    (* the components are equal ... *)
    (setup_morph C1 C2 f) = (setup_morph C1 C2 g) /\
    (msg_morph   C1 C2 f) = (msg_morph   C1 C2 g) /\
    (state_morph C1 C2 f) = (state_morph C1 C2 g) /\
    (error_morph C1 C2 f) = (error_morph C1 C2 g) <->
    (* ... iff the morphisms are equal *)
    f = g.
```

Listing 5.14: Equality of contract morphisms.

5.2.1 Composition of Morphisms

Contract morphisms can be composed. We define composition via a function `compose_cm`, which takes morphisms

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

and returns a morphism

`compose_cm g f : ContractMorphism C1 C3`.

To compose contract morphisms, we simply compose their component functions.

```
Definition compose_cm (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) :
    ContractMorphism C1 C3 := {|
    (* the components *)
    setup_morph := compose (setup_morph C2 C3 g) (setup_morph C1 C2 f) ;
    msg_morph   := compose (msg_morph   C2 C3 g) (msg_morph   C1 C2 f) ;
    state_morph := compose (state_morph C2 C3 g) (state_morph C1 C2 f) ;
    error_morph := compose (error_morph C2 C3 g) (error_morph C1 C2 f) ;
    (* the coherence results *)
    init_coherence := compose_init_coh g f ;
    recv_coherence := compose_recv_coh g f ;
    |}.
```

Listing 5.15: Composition of contract morphisms in ConCert.

The function `compose_cm` relies on two lemmas, `compose_init_coh g f` and `compose_recv_coh g f`, which prove the coherence results for the composed natural transformations of Figure 5.1. These lemmas simply show that commuting diagrams compose. That is, if we have the diagram below such that each of the left and right squares commute, then the outer rectangle also commutes.

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[d, "f"] & C \arrow[r, "j_1"] \arrow[d, "g"] & E \arrow[d, "h"] \\
B \arrow[r, "i_2"] & D \arrow[r, "j_2"] & F
\end{tikzcd}$$

```
Lemma compose_init_coh (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) :
    let setup_morph' := (compose (setup_morph C2 C3 g) (setup_morph C1 C2 f)) in
    let state_morph' := (compose (state_morph C2 C3 g) (state_morph C1 C2 f)) in
    let error_morph' := (compose (error_morph C2 C3 g) (error_morph C1 C2 f)) in
    forall c ctx s,
        result_functor state_morph' error_morph'
            (init C1 c ctx s) =
        init C3 c ctx (setup_morph' s).
```

Listing 5.16: The init coherence lemma for morphism composition.

```
Lemma compose_recv_coh (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) :
    let msg_morph'   := (compose (msg_morph   C2 C3 g) (msg_morph   C1 C2 f)) in
    let state_morph' := (compose (state_morph C2 C3 g) (state_morph C1 C2 f)) in
    let error_morph' := (compose (error_morph C2 C3 g) (error_morph C1 C2 f)) in
    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 C3 c ctx (state_morph' st) (option_map msg_morph' op_msg).
```

Listing 5.17: The receive coherence lemma for morphism composition.

Example 5.2.5 (Composition with the identity morphism)**.** Composing any morphism with the identity morphism does nothing, and the proof is trivial using the equality lemma `eq_cm_iff` from Example 5.2.4.

```
Lemma compose_id_cm_left (f : ContractMorphism C1 C2) :
    compose_cm (id_cm C2) f = f.

Lemma compose_id_cm_right (f : ContractMorphism C1 C2) :
    compose_cm f (id_cm C1) = f.
```

Listing 5.18: Left and right composition of the identity morphism is trivial.

Example 5.2.6 (Morphism Composition is Associative)**.** Composition is associative. This is a trivial proof using the equality lemma `eq_cm_ff` from Example 5.2.4, since composition of each component function of a contract morphism is also associative.

```
Lemma compose_cm_assoc
    (f : ContractMorphism C1 C2)
    (g : ContractMorphism C2 C3)
    (h : ContractMorphism C3 C4) :
    compose_cm h (compose_cm g f) =
    compose_cm (compose_cm h g) f.
```

Listing 5.19: Composition of contract morphisms is associative

Example 5.2.7 (Contract Isomorphism)**.** Finally, using notions of composition, identity, and equality, we can define contract *isomorphisms*. As we will see in Chapter 6, these are morphisms for which the formal, structural relationship expressed between the contracts is an equivalence. We define these via the following predicate.

```
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 5.20: The formal definition of an isomorphism of contracts.

That is, morphisms `f : ContractMorphism C1 C2` and `g : ContractMorphism C2 C1` are together an isomorphism of contracts if, when composed in either order, their composition is equal to the identity morphism. As we will see in Chapter 6, a contract isomorphism results in a bisiumlation of contracts.

5.3 Morphism Induction

In sections 5.4 and 5.5 we will explore specification and proof using contract morphisms in depth. Before doing so, we must introduce a proof technique which incorporates contract morphisms with ConCert's contract induction tactic in order to prove contract invariants.

Consider contracts `C1` and `C2` and a contract morphism

`f : ContractMorphism C1 C2.`

We prove two theorems, addressing two cases. The first, which we call left morphism induction, can be used to prove an invariant of `C1` using known properties of `C2`. The second, which we call right morphism induction, can be used to prove an invariant of `C2` using known properties of `C1`.

5.3.1 Contract Trace and Reachability

Before introducing left and right morphism induction, we need the notions of contract trace and contract reachability. These are analogous to ConCert's notions of chain trace and reachable chain states. The two are related in that the state of a contract deployed in a reachable chain state is always a contract-reachable state, but contract-reachable states could in principle include some states which do not correspond to any reachable chain state.

First, we define contract steps as successful calls to the `receive` function.

```
Record ContractStep (C : Contract Setup Msg State Error)
    (prev_cstate : State) (next_cstate : State) :=
    build_contract_step {
    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 5.21: Contract steps are successful calls to the `receive` function.

Then a contract's trace is a chained list of contract states, linked together by contract steps.

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

Listing 5.22: A contract's trace is a chained list of contract states, linked together by contract steps.

This gives us a natural notion of a contract-genesis state, which is defined as a state into which it is possible for the contract to initialize.

```
Definition is_genesis_state (C : Contract Setup Msg State Error) (init_cstate : State) :=
    exists init_chain init_ctx init_setup,
    init C init_chain init_ctx init_setup = Ok init_cstate.
```

Listing 5.23: A contract genesis state is one into which a contract can initialize.

It also gives us a natural notion of a reachable contract state, which is a state with a trace from some contract genesis state.

```
Definition cstate_reachable (C : Contract Setup Msg State Error) (cstate : State) :=
    exists init_cstate,
    (* init_cstate is a valid initial cstate *)
    is_genesis_state C init_cstate /\
    (* with a trace to cstate *)
    inhabited (ContractTrace C init_cstate cstate).
```

Listing 5.24: A contract-reachable state is one for which there exists a trace from a contract genesis state.

We will use these definitions in the remainder of this chapter, as well as in Chapter 6.

5.3.2 Left Morphism Induction

Recall from §3.4 that to prove a contract invariant with contract induction, one proves the invariant on the base case (contract deployment), and the inductive step consists of all the ways that the blockchain can make progress. A contract morphism `f : ContractMorphism C1 C2` indicates that there is a structural relationship between `C1` and `C2` at each of the relevant steps of induction: first at contract deployment, the base case, via the transformation of the init function; and then at each inductive step via the transformation of the receive function.

This means that when performing contract induction on `C1`, we should have access to the corresponding relational structure of `C2`—and the same going the other way if we're inducting on `C2`. As we will see, we might wish to prove an invariant of `C1` in terms of an invariant on `C2`.

Left morphism induction facilitates such a goal, stating that for any blockchain state `bstate` where `C1` is deployed at some contract address `caddr` with state `cstate1`, there exists a corresponding state `cstate2` of `C2`, which is a contract-reachable state of `C2`, and which is the image of `cstate1` under the `state_morph` component of `f`.

```
(* f : C1 -> C2, inducting on C1 *)
Theorem left_cm_induction :
    (* forall simple morphism and reachable bstate, *)
    forall (f : ContractMorphism C1 C2) bstate caddr
        (trace : ChainTrace empty_state bstate),
    (* where C is at caddr with state cstate, *)
    env_contracts bstate caddr = Some (C1 : WeakContract) ->
    exists (cstate1 : State1),
    contract_state bstate caddr = Some cstate1 /\
    (* every reachable cstate1 of C1 corresponds to a contract-reachable cstate2 of C2: *)
    exists (cstate2 : State2),
    (* 1. init_cstate2 is a valid initial cstate of C'  *)
    cstate_reachable C2 cstate2 /\
    (* 2. cstate and cstate' are related by state_morph. *)
    cstate2 = state_morph C1 C2 f cstate1.
```

Listing 5.25: Left contract morphism induction.

In particular, if we can deduce an invariant of `cstate1` from a known invariant of `cstate2` by their relationship defined by `state_morph`, then we can prove our invariant of `C1` via an invariant of `C2`.

5.3.3 Right Morphism Induction

Right morphism induction takes the opposite case: we prove an invariant of `C2` via known properties of `C1` using a morphism `f : ContractMorphism C1 C2`. Because of the direction of `f`, we do not know that every reachable state of `C2` has a corresponding reachable state of `C1` related to that of `C2` via `state_morph`. Rather, we prove that if we can find a contract trace of `C1`, it can be transformed into one of `C2`.

```
(* f : C1 -> C2, inducting on C2 *)
Theorem right_cm_induction:
    forall (from to : State1) (f : ContractMorphism C1 C2),
    (* every contract trace, and thus reachable state, of C1 ... *)
    ContractTrace C1 from to ->
    (* has a corresponding contract trace of C2 *)
    ContractTrace C2 (state_morph C1 C2 f from) (state_morph C1 C2 f to).
```

Listing 5.26: Right contract morphism induction.

Right morphism induction, then, can be used to prove the existence of some contract-reachable state of `C2` via one of `C1`. For example, if `C2` is an upgraded version of `C1`, one way to specify backwards compatibility is that every contract-reachable state of `C1` has a corresponding contract-reachable state of `C2` which preserves essential data. We prove this by right morphism induction in Example 5.4.2.

5.4 Reasoning with Morphisms: Specification and Proof

Morphisms can be used in a variety of ways in proof and specification of smart contracts. We will demonstrate with three examples: specifying a contract upgrade in relation to the old (§5.4.1), proving backwards compatibility (§5.4.2), and proving Hoare-like properties of partial correctness (§5.4.3).

5.4.1 Specifying a Contract Upgrade With Morphisms

Let us revisit the Urainum Finance exploit from §1.2.2.

Example 5.4.1 (Uranium Finance Upgrade Specification)**.** Recall from §1.2.2 Uranium Finance, a contract which was exploited because developers replaced a constant `k` set at `1,000` with `10,000` in all but one of its instances during an upgrade. The result was wildly incorrect pricing, which rapidly drained their liquidity pools.

Suppose `C1` is the Uranium Finance contract pre-upgrade, and `C2` is the contract post-upgrade, and suppose the upgrade was only to adjust how the contract prices trades. Suppose further that the upgrade was to increase the decimal precision of this calculation 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.

The original contract `C1` will have a storage type, then, which keeps track of internal token balances.

```
Context { storage : Type } { get_bal : storage -> N }.
```

Listing 5.27: We assume a storage type and a function which calculates balances.

It will also have a `TRADE` entrypoint which accepts a message whose payload includes a desired trade quantity, which we formalize with a typeclass as we did in Chapter 4.

```
(* A typeclass which characterizes the entrypoint type *)
Class Msg_Spec (T : Type) := {
    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 5.28: We assume an entrypoit type, characterized by `Msg_Spec`, which includes a trade function, and introduce into the context an `entrypoint` type and an instance of `Msg_Spec entrypoint`.

We now assume that `C1` has some function `calculate_trade` which calculates how many tokens will be traded in for a given contract call to the `TRADE` entrypoint. The trade quantity, internal token balances, and the calculate_trade function will all be accurate up to some decimal place, commonly 6 or 9 in the wild. This is formalized in the following property, which we assume into our context.

```
(* get_bal changes according to calculate_trade, meaning that: *)
Definition spec_trade : Prop :=
    forall cstate chain ctx trade_data cstate' acts,
    (* for any successful call to the trade entrypoint of C1, *)
    receive C1 chain ctx cstate (Some (trade trade_data)) = Ok(cstate', acts) ->
    (* the balance in storage updates as follows. *)
    get_bal cstate' =
    get_bal cstate + calculate_trade (trade_qty trade_data).
```

Listing 5.29: The formalized proposition that C1 uses calculate_trade to price trades.

Now when we upgrade C1 to C2 with the exclusive purpose of increasing the accuracy by one decimal place for internal token balances and trades, we update calculate_trade to calculate_trade_precise which calculates at one higher decimal place of accuracy. This is formalized as follows.

```
(* an auxiliary function which rounds down by one decimal place *)
Definition round_down (n : N) := n / 10.

(* we assume that calculate_trade_precise is related to calculate_trade by round_down *)
Context { calculate_trade_precise : N -> N }
        (* (i.e. calculate_trade_precise rounds down to calculate_trade) *)
        { calc_trade_coherence : forall n,
            round_down (calculate_trade_precise n) =
            calculate_trade (round_down n) }.
```

Listing 5.30: We assume a function calculate_trade_precise, which always rounds down to the calculate_trade function.

Then C2 is assumed to use calculate_trade_precise to calculate its trades, formalized in this assumed proposition which is the analogue of spec_trade but for C2.

```
(* Now trades are calculated in line with calculate_trade_precise. *)
Definition spec_trade_precise : Prop :=
    forall cstate chain ctx trade_data cstate' acts,
    (* ... meaning that for a successful call to the trade entrypoint of C2, *)
    receive C2 chain ctx cstate (Some (trade trade_data)) = Ok(cstate', acts) ->
    (* the balance held in storage goes up by calculate_trade_precise. *)
    get_bal cstate' =
    get_bal cstate + calculate_trade_precise (trade_qty trade_data).
```

Listing 5.31: The formalized proposition that C2 uses calculate_trade_precise to price trades.

Assuming that we have already formally verified C1 correct with respect to some formal specification (and metaspecification), and wished to do the same for C2, we might do so by altering the specification of C1 until it correctly specifies the contract C2, and then verify C2 from the ground up. As nearly all of the contract functionality remained unchanged in the upgrade, this would require re-proving many results already proved about C1, where the slight changes we made to C1 would likely make it impossible to simply copy/paste the proofs. Furthermore, as we saw in the previous chapter, specifications are difficult to write correctly, and small changes to the specification may have unintended consequences. And in altering the specification we might make the same error that the Uranium Finance engineers did and unintentionally create a vulnerability.

To ensure safety, our metaspecification specifies a relationship between C2 and C1: C2 should "do everything that C1 does," except that internal balances and trades should be calculated at one decimal place higher in precision. We can articulate this rigorously through a contract morphism f from the contract upgrade C2 to the original contract C1 that has the following properties.

First, that when f sends inputs of C2.(receive) to C1.(receive), it rounds down the precision of the requested trades using our rounding function round_down.

```
(* 1. 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 = round_down (trade_qty t').
```

Listing 5.32: f rounds down the inputs of the receive function.

Second, that f is the identity morphism on all entrypoints aside from the trade entrypoint.

```
(* 2. aside from trade, f doesn't touch the other entrypoints *)
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.
```

Listing 5.33: f is the identity morphism on all but the trade entrypoint.

Third, that f rounds down on the balances kept in storage exposed by get_bal, but is the identity on all other aspects of the storage.

```
(* 3. f rounds down on the storage, but doesn't touch anything else. *)
Context {st_morph : storage -> storage}
        {state_rounds_down : forall st, get_bal (st_morph st) = round_down (get_bal st)}.

Definition f_state_morph (f : ContractMorphism C2 C1) : Prop :=
    (state_morph C2 C1 f) = st_morph.
```

Listing 5.34: f rounds down balances kept in storage.

Fourth, that f is the identity function on error values.

```
(* 4. f is the identity on error values *)
Definition f_recv_output_err (f : ContractMorphism C2 C1) : Prop :=
    (error_morph C2 C1 f) = id.
```

Listing 5.35: f is the identity on error values.

Fifth and finally, that f is the identity on setup values.

```
(* 5. f is the identity on contract initialization *)
Definition f_init_id (f : ContractMorphism C2 C1) : Prop :=
    (setup_morph C2 C1 f) = id.
```

Listing 5.36: f is the identity on setup values.

The two coherence results of a contract morphism f satisfying these properties show that the upgrade from C1 to C2 was done as intended: we increased precision successfully without changing how trades were priced. We know this because we have that, after rounding, a trade on C2 is the same as the analogous trade on C1. We also know that all other functionality remains the same. In particular, this is where the Uranium Finance engineers would have realized the pricing mechanism of the upgraded contract did not conform to that of the old, avoiding the catastrophic mistake and subsequent exploitation.

We show this via the following formalized contract invariant on C2, which is proved by left morphism induction. It states that for any reachable state of C2, the analogue state of C1 given by the contract morphism f : ContractMorphism C2 C1 simply rounds the balances down.

```
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 = round_down (get_bal cstate').
```

Listing 5.37: All reachable states of C2 round down to their corresponding states in C1.

In particular, any contract-reachable state of C2 has an analogous contract-reachable state of C1, so any invariants of C1 which are independent of the precision of balances kept in storage and of trades still hold for C2. Those which depend on the precision may have an analogous form which holds for C2; with the contract morphism in place, this would be the only verification work required to formally verify C2 to be correct.

5.4.2 Adding Features and Backwards Compatibility

In a similar spirit to Example 5.4.1, let us consider how we might formally specify backwards compatibility using a contract morphism.

Example 5.4.2 (Backwards Compatibility)**.** Consider contracts `C1` and `C2`, where `C2` is an upgrade of `C1`, and suppose that we wish to show that `C2` is backwards compatible with `C1`.

This can be expressed via a contract morphism in a very precise way: one can not only prove that `C2` is backwards compatible with `C1`, but also indicate exactly *how*—we can indicate which entrypoints and actions in the new contract correspond to which functionality of the old through the component functions of the contract morphism.

We illustrate with an example of a counter contract `C1` which keeps `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 5.38: The entrypoint types of `C1` and `C2`, respectively.

We will 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.
```

Listing 5.39: The component functions of a morphism defining backwards compatibility.

Note first that `f` is an embedding since each of its component functions are injections.

```
Lemma embedding : is_inj_cm f.
```

Listing 5.40: The resulting morphism `f` is an embedding of contracts.

In particular, this means that any reachable state of `C1` 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.

We prove this result using morphism induction as follows. The theorem states that for all reachable chain states with `C1` deployed, there is a corresponding contract-reachable state of `C2` whose state is equal to that of `C1`.

```
Theorem injection_invariant bstate caddr (trace : ChainTrace empty_state bstate):
    (* Forall reachable states with contract C1 at caddr, *)
    env_contracts bstate caddr = Some (C1 : WeakContract) ->
    (* forall reachable states of C1 cstate, 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 5.41: `C2` is backwards compatible with `C1` via the contract embedding `f`.

That the corresponding, contract-reachable state of `C2` is equal to the state of `C1` is precisely what is meant by backwards-compatibility: were we to use only the entrypoints of `C2` inherited from `C1`, we would get identical contract behavior.

Like in Example 5.4.1, which specified meta properties of an upgrade, backwards compatibility can be articulated in a metaspecification for an arbitrary pair of contracts and their specifications, that requires a contract embedding of one contract into another.

5.4.3 Transporting Hoare-Like Properties Over a Morphism

Departing slightly from the previous two examples, we introduce a generic proof technique which uses the coherence proofs of a contract morphism to transport Hoare-like properties over a contract morphism.

Consider the property `unpool_emits_txns` from the structured pools specification of §4.4, which is formalized as follows (see line 219 of Listing A.5 in Appendix A.1.2 for the full context).

```
(* When the UNPOOL entrypoint is successfully called, it emits a BURN call to the
    pool_token, with q in the payload *)
Definition unpool_emits_txns (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate' acts,
    (* the call to UNPOOL was successful *)
    receive contract chain ctx cstate (Some (unpool msg_payload)) = Ok(cstate', acts) ->
    (* in the acts list there are burn and transfer transactions *)
    exists burn_data burn_payload transfer_to transfer_data transfer_payload,
    (* there is a burn call in acts *)
    let burn_call := (act_call
        (* calls the pool token address *)
        (stor_pool_token cstate).(token_address)
13          (* with amount 0 *)
14          0
15          (* with payload burn_payload *)
16          (serialize (FA2Spec.Retire burn_payload))) in
17      (* with has burn_data in the payload *)
18      In burn_data burn_payload /\
19      (* and burn_data has these properties: *)
20      burn_data.(FA2Spec.retire_amount) = msg_payload.(qty_unpooled) /\
21      (* the burned tokens go from the unpooler *)
22      burn_data.(FA2Spec.retiring_party) = ctx.(ctx_from) /\
23      (* there is a transfer call *)
24      let transfer_call := (act_call
25          (* call to the token address *)
26          (msg_payload.(token_unpooled).(token_address))
27          (* with amount = 0 *)
28          0
29          (* with payload transfer_payload *)
30          (serialize (FA2Spec.Transfer transfer_payload))) in
31      (* with a transfer in it *)
32      In transfer_data transfer_payload /\
33      (* which itself has transfer data *)
34      In transfer_to transfer_data.(FA2Spec.txs) /\
35      (* whose quantity is the quantity pooled *)
36      let r_x := get_rate msg_payload.(token_unpooled) (stor_rates cstate) in
37      transfer_to.(FA2Spec.amount) = msg_payload.(qty_unpooled) / r_x /\
38      (* and these are the only emitted transactions *)
39      (acts = [ burn_call ; transfer_call ] \/
40       acts = [ transfer_call ; burn_call ]).
```

Listing 5.42: The `unpool_emits_txns` property from the formal specification of §4.4, which is a Hoare-like property of partial correctness.

Note that on line 6 of Listing 5.42, we assume that the contract executes without error, assuming

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

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

More fundamentally, `unpool_emits_txns` is a Hoare-like assertion of partial correctness, reasoning about a successful call to the structured pool contract, where the preconditions are that the state of the chain be reachable with contract address `caddr` and state `cstate`, and the postconditions are statements about `acts` and `cstate′` including that `acts` contain an appropriate `TRANSFER` transaction. Indeed, a reflection on the formal specification of §4.4 reveals that most of the properties of the specification are of this form.

Consider contracts `C1` and `C2` and morphism `f : ContractMorphism C1 C2`, and suppose that `C2` satisfies `unpool_emits_txns`. Suppose further that `C1` also has an `UNPOOL` entrypoint. If we wish to prove `unpool_emits_txns` for `C1`, we must reason about successful calls to the `UNPOOL` entrypoint. As we will see, we can take advantage of the coherence results which come with `f`, namely that a successful call to `C1`

```
receive C1 chain ctx state (Some msg) = Ok(cstate', acts)
```

results in a successful call to `C2`

```
receive C2 chain ctx (f.(state_morph) state) (Some (f.(msg_morph) msg)) =
                    (Ok (f.(state_morph) cstate', acts)).
```

Using this we can reason about successful calls to the `UNPOOL` entrypoint of `C1` via the corresponding call under `f` to the `UNPOOL` entrypoint of `C2`.

Example 5.4.3 (Transporting a Property From the Specification)**.** Consider contracts

```
C1 : Contract setup entrypoint  storage error
C2 : Contract setup entrypoint' storage error
```

and assume `(is_sp : is_structured_pool C2)`, meaning that `C2` is a structured pool, satisfying the specification of §4.4. Suppose further that the types `entrypoint` and `entrypoint'` are defined as follows:

```
(* entrypoint type *)
Inductive entrypoint :=
| Pool : pool_data -> entrypoint
| Unpool : unpool_data -> entrypoint.

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

Listing 5.43: The entrypoint types of `C1` and `C2`, respectively.

This gives us an embedding between the entrypoint types of `C1` and `C2`.

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

Listing 5.44: An embedding of `entrypoint` into `entrypoint'`.

Finally, assume that the functionality of `C1` regarding its `POOL` and `UNPOOL` entrypoints is identical to that of `C2` regarding its `POOL'` and `UNPOOL'` entrypoints via the embedding.

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

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

Listing 5.45: The `init` and `receive` functions behave identically under the embedding.

Then we can construct a morphism `f : ContractMorphism C1 C2` using the coherence proofs and the `embed_entrypoint` function that we've just seen:

```
(* construct a contract morphism *)
Definition f : ContractMorphism C1 C2 := {|
    setup_morph := id ;
    msg_morph   := embed_entrypoint ;
    state_morph := id ;
    error_morph := id ;
    (* coherence *)
    init_coherence := init_coherence_pf ;
    recv_coherence := recv_coherence_pf ;
|}.
```

Listing 5.46: A contract morphism `f : ContractMorphism C1 C2`.

Now suppose that we wish to prove `unpool_emits_txns C` from the structured pools specification.

```
Theorem pullback_unpool_emits_txns : unpool_emits_txns C1.
```

We already have a proof of `unpool_emits_txns C2`, and we know that for all messages `msg` into `C1`,

`receive C1 c ctx st (Some msg) = receive C2 c ctx st (Some (embed_entrypoint msg)).`

Since in proving `unpool_emits_txns C1` we assume a successful execution,

`receive C1 c ctx st (Some msg) = Ok(cstate', acts),`

via `f` we get a successful execution of `C2`,

`receive C2 c ctx st (Some embed_entrypoint msg) = Ok(cstate', acts).`

Our assumed proof of `unpool_emits_txns C2` gives us a proof of the postconditions of `unpool_emits_txns` for `C2`, so we can derive the fact that the relevant postconditions of `unpool_emits_txns` also hold for this call of `C1`. This proves `unpool_emits_txns C1`.

We note that this proof relies on `C1` and `C2` being very similar to each other. The more `C1` and `C2` are similar to each other—for example, in a contract fork or upgrade—the easier it is to use the coherence results of the contract morphism to prove results about `C1` in terms of `C2`. In particular, if a contract is only altered slightly to be upgraded, we could save tedious work of re-proving specified properties or re-specifying by simply using what is already known about the old contract to prove things about the new via a contract morphism.

5.4.4 Summary

Let us reflect on our goal from §5.1.2 of developing formal language to specify contract upgrades. Our main observation was that we need language to formally compare old and new contract versions and their specifications in order to be able to verify that the specification of a new contract version is correct with regards to the intent of the upgrade.

The examples of this section have shown us that contract morphisms provide such a formal language. In Example 5.4.1, we articulated the goal of an upgrade which isolated one piece of the contract functionality, altered it, and left everything else untouched. The intent of the upgrade was expressed through a morphism which asserted that calculations in the upgrade, when rounded down, be equivalent to calculations in the previous version. In Example 5.4.2, we precisely specified our goal that a contract upgrade be backward compatible with a previous version. Again the intent of this upgrade, that we add functionality while preserving the old, was expressed by a contract embedding which identified precisely which pieces of the new contract version corresponded to the old. Finally, Example 5.4.3 showed us that properties of one contract can be used to prove properties of another by way of a contract morphism, which can help us prove facts about an upgrade in terms of its previous versions.

This gives us our desired formal language for individual upgrades, and it is now our task to treat generic upgradeable contracts.

5.5 A Mathematical Characterization of Contract Upgrades

In what follows we will mathematically characterize contract upgradeability using contract morphisms. Before doing so, we go into greater detail on upgradeable contracts.

5.5.1 The Varieties of Upgradeable Contracts

Upgradeable contracts vary, ranging from limited upgradeability to highly flexible frameworks.

Starting from the limited end, take for example MakerDAO, the contract that governs the stablecoin DAI. Certain contract parameters can be updated through a vote by MKR token holders, such as the so-called *stability rate*, which is an interest rate that affects the price of DAI [83]. While MakerDAO cannot radically change the contract features and functionality, we still consider it upgradeable as its functionality can be updated by governance token holders in these limited ways.

Contracts like MakerDAO are an important class of upgradeable contracts called decentralized autonomous organizations (DAOs). These are smart contracts that are governed collectively by holders of a governance token [51]. In some DAOs, governance token holders can also vote to alter, and in some cases fully upgrade, contract functionality. These contracts are governed through a complex web of tokens and incentives which is a fully-fledged research topic in its own right [29].

For example, take Murmuration, a generic DAO template built on Tezos [128]. Murmuration includes a governance token contract and a DAO, the latter of which is governed by those who hold governance tokens. A user submits a proposal which consists of an anonymous lambda function and various metadata. If the proposal passes and executes, which is determined through a voting procedure by governance token holders, the lambda of the proposal becomes the new contract functionality, replacing an entrypoint function. By definition, a lambda has few restrictions, if any, so the upgradeability afforded by Murmuration is substantially greater than that of MakerDAO.

Moving beyond DAOs, the Diamond upgrade framework [95] from §5.1.1 is an extremely flexible system of proxy contracts, which essentially allows for fully altering contract functionality. In addition to changing entrypoint functions like Murmuration, contracts conforming to the Diamond standard can also modify entrypoints to the contract and add to the contract storage.

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

5.5.2 Isolating Mutable and Immutable Parts

We wish to formally understand contract upgradeability in order to adequately form metaspecifications for upgradeable contracts. To do so, we need language which can characterize its upgradeability properties—by precisely identifying its immutable and mutable parts—as well as language for the governance process and how the upgrade process relates to the actual contract functionality. To that end, we now consider an upgradeable contract in relation to its mutable and immutable parts.

First, the immutable part, which we call the *base contract*. This part of an upgradeable contract is what governs the upgradeability framework and the corresponding incentive and game-theoretic structure of the upgrade process. It indicates what about the contract can change, and through what process it can change, similar to constitutional rules which govern legislative rule-making.

To express this mathematically, we isolate the immutable parts of a contract `C` into a base contract `C_b`, and construct a morphism

```
f_p : ContractMorphism C C_b
```

which forgets anything other than the contract structure that governs upgrades and contract versions. This will be a quotient of contracts.

Example 5.5.1 (Quotient Onto the Base Contract)**.** Consider a contract `C` whose storage contains some natural number `n : N` and a function `s : N -> N`, and which has two entrypoints: `next`, which applies `s` to `n`, and updates the number in storage with `(s n)`; and `upgrade_fun`, which accepts a parameter `s' : N -> N`, and replaces `s` in storage with `s'`. This contract is upgradeable in the sense that the functionality of `next`, the primary way in which a user can act on the number in contract storage, can be changed by calling `upgrade_fun`.

```
Inductive entrypoint :=
| next (u : unit)
| upgrade_fun (s' : N -> N).
Record storage := { n : N ; s : N -> N ; }.
```

Listing 5.47: The entrypoint and storage types of our contract `C`.

Now let us consider what the immutable part of `C` is. Because the upgradeability refers to the functionality of the `upgrade_fun` entrypoint rather than `next`, the base contract of upgradeable `C` forgets any natural number in the storage of `C` and remembers only the function `s : N -> N`, which indicates the version of `C`. It only has one entrypoint relevant to the structure of `C`, `upgrade_fun`, which is the upgrade functionality.

To express this formally, we define the base contract `C_b`, whose entrypoint type, `entrypoint_b`, and storage type, `storage_b`, are defined as follows in Listing 5.48.

```
Inductive entrypoint_b' := | upgrade_fun_b (s' : N -> N).
Definition entrypoint_b := (entrypoint_b' + unit)%type.
Record storage_b := { s_b : N -> N ; }.
```

Listing 5.48: The entrypoint and storage types of the base contract C_b.

The entrypoint type of C_b, entrypoint_b, is defined as a sum type of entrypoint_b', a type which isolates the upgradeability entrypoint of C, and the unit type. This is so that we can define a function

msg_morph : entrypoint -> entrypoint_b.

In particular, we need to be able to send messages of type entrypoint which do not correspond to upgradeability somewhere other than an entrypoint—namely, into the summed unit.

We now define a morphism f : ContractMorphism C C_b with the following component functions.

```
Definition msg_morph_p (e : entrypoint) : entrypoint_b :=
    match e with
    | next _ => inr tt (* not upgrade functionality *)
    | upgrade_fun s' => inl (upgrade_fun_b s') (* corresponds to an upgrade *)
    end.
Definition state_morph_p : storage -> storage_b :=
    (fun (x : storage) => {| s_b := x.(s) ; |}).
```

Listing 5.49: Two component functions of a morphism f : ContractMorphism C C_b.

The component function state_morph isolates the function s in storage, which is the data corresponding to a contract's version; msg_morph sends a message to next to the unit, as it does not correspond to an upgrade, and forwards an message to upgrade_fun. The coherence results come easily by design—C_b is desined to simply be a compressed version of C. We will see later on that C_b does in fact characterize the immutable parts of C.

Let us now move to the mutable part of C, which we call the *version contracts*. This is the contract's functionality which stands apart from the upgradeability framework, which changes through upgrades. Were it not implemented as an upgradeable contract, it could in principle be implemented as a standalone, non-upgradeable contract. Indeed, any specific version of the upgradeable contract emulates a version contract from within the upgradeability framework.

Example 5.5.2 (Family of Contract Embeddings)**.** We continue with C from Example 5.5.1 which always has some n : N and s_next : N -> N in storage. Now we wish to define a family of contracts which describes the functionality that is specific to any particular version of the contract.

Opposite to distilling the upgradeability skeleton as we did in Example 5.5.1, we isolate and embed the functionality of a specific version of C. That is, we remove upgrade_fun from the entrypoint type and s from storage, using a (fixed) s_next:N -> N when the next entrypoint is called.

```
Inductive entrypoint_version := | next_f (u : unit).
Record storage_version := { n_f : N }.
```

Listing 5.50: The entrypoint and storage types of a version contract `C_f`.

Any contract `C_f'` with these entrypoint and storage types can be called to update its internal natural number in storage, but it is unable to upgrade in the sense that `s_next` is fixed. Importantly, were we to initialize `C` with `s_next` and never call `upgrade_fun`, then `C` and `C_f'` would be identical behaviorally.

Note further that the particular version that `C_f'` corresponds to is precisely the information in the storage of our base contract `C_b`. In fact, we can define a family of contracts which is parameterized by inhabitants of the storage type of `C_b`, consisting of contracts whose types are defined as follows.

```
Definition entrypoint_f : storage_b -> Type := fun v => entrypoint_version.
Definition storage_f    : storage_b -> Type := fun v => storage_version.
Definition setup_f      : storage_b -> Type := fun v => N.
Definition error_f      : storage_b -> Type := fun v => N.
Definition result_f     : storage_b -> Type :=
    fun v => ResultMonad.result ((storage_f v) * list ActionBody) (error_f v).
```

Listing 5.51: A family of contract types parameterized by `storage_b`.

In particular, for a version `v : storage_b`, the `receive` function of each version contract uses the function `v.(s_b)` to execute the `next` entrypoint. The `init` function is simimlarly parameterized.

```
Definition receive_f (v : storage_b)
                (_ : Chain)
                (_ : ContractCallContext)
                (storage_f : storage_f v)
                (msg : option (entrypoint_f v))
                : result_f v :=
    match msg with
    | Some (next_f _) =>
        let st := {| n_f := v.(s_b) storage_f.(n_f) ; |} in
        Ok (st, [])
    | None => Err 0
    end.
```

Listing 5.52: The `receive` function of `C_f version`, parameterized by `version : sotrage_b`

This gives us our family of version contracts as follows.

```
Definition C_f (v : storage_b) :
    Contract (setup_f v) (entrypoint_f v) (storage_f v) (error_f v) :=
    build_contract (init_f v) (receive_f v).
```

Listing 5.53: A family of version contracts, parameterized by `storage_b`.

We now construct a family of embeddings

```
fi_param (v : storage_b) : ContractMorphism (C_f v)
```

whose component functions are defined as follows.

```
Definition msg_morph_i (v : storage_b) (e : entrypoint_f v) : entrypoint :=
    match e with
    | next_f _ => next tt
    end.
Definition setup_morph_i (v : storage_b) (st_f : setup_f v) : setup := {|
    n := st_f ;
    s := s_b v ; |}.
Definition state_morph_i (v : storage_b) (st_f : storage_f v) : storage :=
    {| n := st_f.(n_f) ; s := s_b v ; |}.
Definition error_morph_i (v : storage_b) : error_f v -> error := id.
```

Listing 5.54: The component functions of the family of morphisms fi_param.

Intuitively, our family of morphisms shows us that any contract-reachable state of C can be characterized as having a copy of some contract in the family C_f, which stays constant until an upgrade is called.

By studying the structure of C, C_b, and C_f from Examples 5.5.1 and 5.5.2, one might be able to convince one's self that there is some sort of decomposition of C, characterized by C_b and C_f. Indeed, this is the case. In the following section we present this first as an abstract theory, and then prove the decomposition results for our contract C.

5.5.3 Decomposing Upgradeability

Generalizing from Examples 5.5.1 and 5.5.2, consider a contract C,

```
C : Contract Setup Msg State Error.
```

types Setup_b, Msg_b, State_b, and Error_b and a contract C_b,

```
C_b : Contract Setup_b (Msg_b + unit) State_b Error_b.
```

and a family of contracts and types parameterized by State_b.

```
setup_f : State_b -> Type.
msg_f   : State_b -> Type.
state_f : State_b -> Type.
error_f : State_b -> Type.
C_f : forall (v : State_b), Contract (setup_f v) (msg_f v) (state_f v) (error_f v).
```

Listing 5.55: A family of contracts parameterized by State_b.

As before, we call `C_b` the base contract and `C_f` the family of version contracts. Furthermore, suppose the following conditions hold.

First, that if `C` receives an empty message, it fails.

```
Definition msg_required := forall chain ctx prev_state,
    exists e, receive C chain ctx prev_state None = Err e.
```

Listing 5.56: Condition 1: `C` returns an error if called with no message.

Second, that the `init` function gives versioned states according to the following predicate `is_versioned`.

```
Definition is_versioned
    (i_param : forall c_version, ContractMorphism (C_f c_version) C)
    cstate :=
    exists c_version cstate_f,
    cstate = state_morph (C_f c_version) C (i_param c_version) cstate_f.
```

Listing 5.57: The predicate `is_versioned`.

```
Definition init_versioned
    (i_param : forall c_version, ContractMorphism (C_f c_version) C) :=
    forall init_state chain ctx setup,
    init C chain ctx setup = Ok init_state ->
    is_versioned i_param init_state.
```

Listing 5.58: Condition 2: All initial states are versioned.

Third, that messages into `C` can be categorized as either a message to the base contract portion of `C`, or to the current version contract portion of `C`. This is defined by the condition that a message is sent to the unit under `p` if and only if it has a preimage under `i`.

```
Definition msg_decomposable
    c_version
    (i : ContractMorphism (C_f c_version) C)
    (p : ContractMorphism C C_b) :=
    forall m,
    msg_morph C C_b p m = inr tt <->
    (exists m', m = msg_morph (C_f c_version) C i m').
```

Listing 5.59: Condition 3: A message is sent to the unit under `p` if and only if it has a preimage under `i`.

Fourth, that all possible states of C can be categorized by the version that they belong to. This is defined by the condition that a contract state has a preimage under `i` if and only if the version contract `C_f` has the version corresponding to the image under `p`.

```
Definition states_categorized
    c_version
    (i : ContractMorphism (C_f c_version) C)
    (p : ContractMorphism C C_b) :=
    forall st,
    (exists st_f, st = state_morph (C_f c_version) C i st_f) <->
    state_morph C C_b p st = c_version.
```

Listing 5.60: Condition 4: A contract state has a preimage under i if and only if the version contract C_f has the version corresponding to the image under p.

Fifth and finally, that there are functions extract_version and new_version_state wich describe how upgrades move between contract versions. The extract_version function takes a message to upgrade C (in other words, an incoming message to C_b under p) and extracts the version into which the message will upgrade C. The new_version_state function takes the previous version old_v and the upgrading message msg, and sends an inhabitant of (state_f old_v), the storage type of the previous version contract, to an an inhabitant of (state_f (extract_version msg)), the storage type of the next version contract. As we will see, in the contracts from Examples 5.5.1 and 5.5.2, extract_version simply uses the payload of upgrade_fun, a function, to get the version of the new contract, and since the storage type of all version contracts is the same, the new_version_state function is simply the identity function.

```
Definition version_transition
    old_v
    (i_param : forall c_version, ContractMorphism (C_f c_version) C)
    (p : ContractMorphism C C_b)
    (extract_version : Msg_b -> State_b)
    (new_version_state : forall old_v msg,
        state_f old_v -> state_f (extract_version msg)) :=
    forall cstate cstate_f,
    (* forall states of version old_v *)
    cstate = state_morph (C_f old_v) C (i_param old_v) cstate_f ->
    (* and forall successful calls ... *)
    forall chain ctx msg new_state new_acts msg',
    receive C chain ctx cstate (Some msg) = Ok (new_state, new_acts) ->
    (* to upgrade the contract C ... *)
    msg_morph C C_b p msg = inl msg' ->
    (* then the new state is the state given by new_version_state *)
    let new_v := extract_version msg' in
    new_state =
        state_morph (C_f new_v) C (i_param new_v) (new_version_state old_v msg' cstate_f).
```

Listing 5.61: Condition 5: The functions extract_version and new_version_state wich describe how upgrades move between contract versions.

Then given contracts C and C_b, a family of contracts C_f, a family of embeddings i_param of contracts from C_f into C, and a quotient p of C onto C_b, C has an upgradeability decomposition into C_b and C_f if there are functions extract_version and new_version_state such that the above-listed conditions hold.

```
Definition upgradeability_decomposition
    (i_param : forall c_version, ContractMorphism (C_f c_version) C)
    (p : ContractMorphism C C_b)
    (extract_version : Msg_b -> State_b)
    (new_version_state : forall old_v msg,
        state_f old_v -> state_f (extract_version msg)) :=
    (* Forall versions of a contract C, *)
    msg_required /\
    init_versioned i_param /\
    forall c_version,
    let i := i_param c_version in
    msg_decomposable c_version i p /\
    states_categorized c_version i p /\
    version_transition c_version i_param p extract_version new_version_state.
```

Listing 5.62: The definition of an upgradeability decomposition, which is a conjunction of all five conditions listed above.

A proof of upgradeability_decomposition C gives us two results which characterize the contract trace of C in terms of its base and version contracts.

First, that all reachable states have a version, defined by the is_versioned predicate from Listing 5.58.

```
Theorem versioned_invariant
    (* Consider family of embeddings, and *)
    (i_param : forall c_version, ContractMorphism (C_f c_version) C)
    (* a projection onto the skeleton C_b. *)
    (p : ContractMorphism C C_b)
    (extract_version : Msg_b -> State_b)
    (new_version_state : forall old_v msg,
        state_f old_v -> state_f (extract_version msg)) :
    (* Then forall reachable states ... *)
    forall bstate caddr (trace : ChainTrace empty_state bstate),
    (* where C is at caddr with state cstate, *)
    env_contracts bstate caddr = Some (C : WeakContract) ->
    exists (cstate : State),
    contract_state bstate caddr = Some cstate /\
    (* if the contract's upgradeability can be decomposed *)
    (upgradeability_decomposition i_param p extract_version new_version_state ->
    (* then every contract state cstate is versioned *)
    is_versioned i_param cstate).
```

Listing 5.63: All reachable states of an upgradeable contract are versioned.

Second, that if we have a contract C with upgradeability decomposition upgradeability_decomposition, then all incoming calls are either upgrades—a call to contract upgradeability, resulting in a new version—or correspond are calls to the current version contract, the version staying constant. Transitions from one version to another result in a new state that is the image of the state of a new version contract.

```
Theorem upgradeability_decomposed
    (* Consider family of embeddings, and *)
    (i_param : forall c_version, ContractMorphism (C_f c_version) C)
    (* a projection onto the base contract C_b. *)
    (p : ContractMorphism C C_b)
    (extract_version : Msg_b -> State_b)
    (new_version_state : forall old_v msg,
        state_f old_v -> state_f (extract_version msg)) :
    (* forall contract states and corresponding contract versions, *)
    forall cstate c_version cstate_f,
    (* with i, the embedding for version c_version, ... *)
    let i := i_param c_version in
    (* if C_f -> C ->> C_b is the decomposition of a contract’s upgradeability ... *)
    upgradeability_decomposition i_param p extract_version new_version_state ->
    (* and cstate is in the image of cstate_f under the embedding i
        (meaning that cstate has version c_version) ... *)
    cstate = state_morph (C_f c_version) C i cstate_f ->
    (* Then forall calls to the versioned contract *)
    forall chain ctx m new_state new_acts,
    receive C chain ctx cstate (Some m) = Ok (new_state, new_acts) ->
    (* it either stays within this version *)
    (exists cstate_f’, new_state = state_morph (C_f c_version) C i cstate_f’) \/
    (* it moves onto a new version *)
    (exists c_version’ cstate_f’,
    new_state = state_morph (C_f c_version’) C (i_param c_version’) cstate_f’).
```

Listing 5.64: All contract calls to an upgradeable contract are either upgrades (to a new version) or stay in the same version; transitions between versions behave as expected.

These two results show that all contract states are versioned by C_f and C_b, and then give precise semantics of how the contract moves between version contracts in C_f by calling the base contract C_b.

Let us reflect on what these mean for our goal of formally characterizing contract upgradeability. We’ve shown that our decomposition isolates the functionality relating to contract upgrades from the functionality relating to each particular contract version, and that it accurately describes all contract states and transitions between them. The definition of the family of version contracts is a precise definition of the bounds of a contract’s upgradeability. And by isolating the governance features of contract upgrades into the base contract, we are able to rigorously reason about any incentive, economic, or game-theoretic aspects of the contract’s upgradeability.

Of course, upgradeable contracts are not typically specified in such a modular way, but as with the Diamond framework [95], upgradeable contracts are typically specified as one, unified contract. We can thus formulate the metaspecification of an upgradeable contract in terms of its base contract, with its associated governance features, and its version contracts, defining the structure common to every possible contract version. The specification of an upgradeable contract is correct if it specifies a contract which can be decomposed into the base and version contracts of the metaspecification.

Example 5.5.3 (Decomposing `C`)**.** We conclude this section by continuing with Examples 5.5.1 and 5.5.2, showing that `C` can be decomposed into the base contract `C_b` and the family of version contracts `C_f`. To do so, we need functions `extract_version` and `new_version_state` which describe how upgrades move between version contracts.

Since versions of `C` are given by the function `s` in storage, the `extract_version` function simply takes the function from the message into the `upgrade_fun` entrypoint.

```
Definition extract_version (m : entrypoint_b') : storage_b :=
    match m with | upgrade_fun_b s_b => {| s_b := s_b |} end.
```

Listing 5.65: The `extract_version` function, which isolates the contract version resulting from a contract call by taking the new function from the upgrade message payload.

The `new_version_state` function takes the state of a previous version contract to the state of a new version contract. Since the sotrage types of all version contracts are the same for this particular family, the function is constant.

```
Definition new_version_state old_v msg (st : storage_f old_v) :
    storage_f (extract_version msg) := st.
```

Listing 5.66: The `new_version_state` function which takes the state of a previous version contract to the state of a new version contract.

We have the following theorem, which says that `C` is decomposable with regards to `C_b` and the quotient `f_p`, and the family of version contracts `C_f` and embeddings `fi_param`.

```
Theorem upgradeability_decomposition :
    upgradeability_decomposition fi_param f_p extract_version new_version_state.
```

Listing 5.67: `C` is decomposable with regards to `C_b` and the family of version contracts `C_f`.

What does this tell us about `C`? We know that all reachable states of `C` are versioned, *i.e.* have an embedding of a contract in the family `C_f` which indicates its version. We also have precise semantics of how the upgrades move between version contracts, via the function `new_version_state`, and we can characterize all incoming messages as either upgrades, or calls to a particular version contract.

5.5.4 Upgradeable Contracts are Fiber Bundles: A Digression

The decomposition of an upgradeable contract resembles a topological phenomenon called a *fiber bundle*. Just as an upgradeable contract knits together its base contract with a family of version contracts, a fiber bundle knits a topological space in with a family of topological spaces into one so-called *total space* [75].

Fiber bundles are hugely influential within mathematics, particularly in geometry and topology, and have an extremely rich theory surrounding them. They are a tool used to understand topological spaces, which are famously complex and mysterious mathematical objects. We point out an analogous structure between topological spaces and smart contracts because smart contracts (and software more broadly) are also complex mathematical objects, and understanding them is precisely the goal of formal verification.

For what follows, we assume basic knowledge about topological spaces and continuous maps; for an introduction to topology, see *e.g.* Munkres' introductory course [96].

First, recall that for a map of topological spaces $f : X \to Y$, the *fiber* of f over $y \in Y$ is the preimage $f^{-1}(y)$, where $f^{-1}(y) := \{x \in X | f(x) = y\}$.

Definition 5.5.1 (Fiber bundle)**.** Consider topological spaces E and B. A continuous surjection $\pi : E \to B$ is a fiber bundle if the following conditions hold:

1. For all $b \in B$, $\pi^{-1}(b)$ is homeomorphic to a fixed topological space F.
2. There is an open cover $\{U_\alpha\}_{\alpha \in I}$ of B with isomorphisms
$$\phi_\alpha : \pi^{-1}(U_\alpha) \xrightarrow{\sim} U_\alpha \times F$$
which restricts on fibers to a homeomorphism.
3. For $\alpha, \beta \in I$, the composition $\phi_\beta^{-1} \circ \phi_\alpha : (U_\alpha \cap U_\beta) \times F \to (U_\alpha \cap U_\beta) \times F$ is well-defined and satisfies
$$\phi_\beta^{-1} \circ \phi_\alpha(x, v) = (x, g_{\alpha\beta}(x)v)$$
for some $\mathrm{Aut}(F)$-valued function $g_{\alpha\beta} : U \cup V \to \mathrm{Aut}(F)$.
4. These maps satisfy
$$g_{\alpha\alpha} = \mathrm{Id} \quad \text{and} \quad g_{\alpha\beta}(x) g_{\beta\gamma}(x) g_{\gamma\alpha}(x) = \mathrm{Id}.$$

The functions $\phi_\beta^{-1} \circ \phi_\alpha$ are called transition functions, E is called the total space, B is called the base space, and F is called the fiber. Diagrammatically, a fiber bundle is often drawn as follows.

$$\begin{tikzcd}
F \arrow[r, hook] & E \arrow[d, two heads] \\
& B
\end{tikzcd}$$

One can think of fiber bundles intuitively as a quotient, similar to a group quotient, where F takes on the analogous role played by a normal subgroup in a group quotient.

Fiber bundles form a category, where maps of fiber bundles, which we simply call *bundle maps*, are commuting squares

$$\begin{tikzcd}
E \arrow[r] \arrow[d] & E' \arrow[d] \\
B \arrow[r] & B',
\end{tikzcd} \tag{5.1}$$

where $E \to B$ and $E' \to B'$ are fiber bundles and all the maps are continuous. If we fix a base space B, we can define a category of fiber bundles over B by defining morphisms to be commuting squares such as (5.1) with the condition that the map on the bottom row be the identity map.

One important feature of fiber bundles is that they have the homotopy lifting property for all CW-complexes. The function $\pi : E \to B$ has the homotopy lifting property with respect to a space X if, for all homotopies

$$h : [0,1] \times X \to B,$$

if there exists a map $f_0 : \{0\} \times X \to E$ such that $\pi \circ f_0 = h|_{\{0\}\times X}$, then there exists a homotopy $f : [0,1] \times X \to E$ such that $\pi \circ f = h$ and $f|_{\{0\}\times X} = f_0$. A *fibration* is a surjection $\pi : E \to B$ which satisfies the homotopy lifting property with respect to any topological space.

If the lift is unique, one important corollary is that for points $b, b' \in B$, a path from b to b' induces a function from the fiber of b to the fiber of b': given $x \in \pi^{-1}(b)$, use the homotopy lifting property to lift the path; the image of our function is the endpoint of that path in $\pi^{-1}(b')$.

Example 5.5.4 (Trivial Bundle)**.** For all topological spaces B and F, $E := B \times F$ is a fiber bundle over B given by the obvious inclusion and projection functions.

Example 5.5.5 (Covering spaces)**.** Let X be a topological space. A covering of X is a fiber bundle, with X as the base space, such that the fibers of X are discrete topological spaces. Paths in a covering space lift uniquely to paths in the covering space, yielding a function on fibers.

Example 5.5.6 (Möbius Bundle)**.** Consider a Möbius band M defined by taking the space $[0,1] \times [-1,1]$ and identifying $(0, x)$ with $(1, -x)$ for all $x \in [-1, 1]$. Similarly, construct a copy of S^1 by taking $[0,1]$ and identifying 0 with 1. We can define a function $\pi : M \to S^1$ given by $(x, y) \mapsto x$. Then π is a fiber bundle with fibers $[-1, 1]$.

Some fiber bundles admit a *section*, which is a function $s : B \to E$ such that $\pi \circ s = \text{id}$. Sections are useful tools in analysis, topology, and differential geometry for a variety of reasons. Topologically, one way to conceptualize the existence of a section is as an indicator of how "twisted" (or rather, untwisted) a fiber bundle is. The trivial bundle, for example, has a section for every $x \in F$ given by $b \mapsto (b, x)$, and is considered to be the "least twisted" type of fiber bundle.

Another example is the Möbius bundle M from Example 5.5.6. M has a section $s : S^1 \to M$ given by $x \mapsto (x, 0)$. However if we remove $(x, 0)$ from every fiber, this gives us a fiber bundle M' with disconnected fibers $[-1, 0) \cup (0, 1]$, which now admits no sections. This is because the Möbius band has a twist: if we try to construct a section of S^1 into M', *e.g.* by drawing a loop in M', no matter what we choose, after one revolution around M' we've swapped sides in the fiber.

The analogy between fiber bundles and upgradeable contracts is as follows. We consider contracts as analogous to topological spaces, where the points of the space are inhabitants of a contract's storage. A successful contract call alters the storage of the contract, and is thus a continuous path—which indicates movement—within the contract.

Our decomposition of a contract into its base and version contracts

$$\begin{tikzcd}
\texttt{C\_fv} \arrow[r, hook] & \texttt{C} \arrow[d, two heads] \\
 & \texttt{C\_b}
\end{tikzcd}$$

follows the analogy of a fiber bundle, where the version contracts are the fibers and the base contract is the base space of the fiber bundle. That `C_f` parameterizes all the fibers over the storage type of `C_b` is the analogue of a continuous parameterization of the fibers by the base space in E.

Some of the conditions of §5.5.3 also have an interpretation in the context of fiber bundles. Condition 2, that all initial states are versioned, indicates that every initialization corresponds to the point in a fiber of `p`. Condition 3, that messages are decomposable, means that a path in `C` is either a path that stays within a fiber, meaning it is a call to a version contract `C_f v`, or a path moves *between* fibers, meaning it is a call to the base contract `C_b`. Condition 4 is that `C_f v` is precisely the fiber over `v` under `p`.

Condition 5 gives us the analogy with the path lifting property: a call to upgrade a contract, which is a path within the base contract `C_b`, lifts to a path in `C` which starts in the fiber `C_f prev_v` of the previous version `prev_v` and ends in the new fiber `C_f new_v` of the new contract version `new_v`. The path of an upgrade in `C_b` is given by the `extract_version` function, which isolates the new version from an incoming upgrade call. Then `new_version_state` is our function between fibers, taking an inhabitant of the state of the previous fiber contract to an inhabitant of the state of the new fiber contract.

Considering this analogy, we might ask if `f_p : ContractMorphism C C_b` admits a section. Recall that in defining `C_b`, we had to sum the entrypoint type `entrypoint_b'` with the unit in order to be able to define the morphism from `C` to `C_b`. This construction, which we call the `pointed_contract` construction, is a general construction on contracts: given a contract

```
C : Contract Setup Msg State Error,
```

we have a so-called *pointed contract*

```
pointed_contract C : Contract Setup (Msg + unit) State Error
```

which admits an extra entrypoint which succeeds when called, but does nothing to the state and emits no actions. In particular, `C_b` from our example is `pointed_contract C_b'` for a contract `C_b'`.

There is also a canonical embedding of `C` into `pointed_contract C` for all `C`,

```
pointed_include :  ContractMorphism C (pointed_contract C),
```

whose component functions are the identity apart from on the message type, where the component function is `(fun m => inl m)`.

We will then define a section of our fiber bundle analogue to be a morphism

```
fp_rinv :  ContractMorphism C_b' C
```

such that

```
compose_cm f_p (fp_rinv n) = pointed_include C_b'.
```

As it turns out, we have such a section for the upgradeable contract of Example 5.5.3, defined by the following component moprhisms.

```
Definition setup_morph_s n : setup_b -> setup :=
    (fun (x : setup_b) => {| n := n ; s := x.(s_b) |}).
Definition msg_morph_s (e : entrypoint_b') : entrypoint :=
    match e with | upgrade_fun_b s' => upgrade_fun s' end.
Definition state_morph_s n : storage_b -> storage :=
    (fun (x : storage_b) => {| n := n ; s := x.(s_b) ; |}).
Definition error_morph_s : error_b -> error := (fun (x : error_b) => x).
```

Listing 5.68: A right inverse of `p` from Example 5.5.3

Note in particular that `setup_morph_s` and `state_morph_s` are parameterized by a natural number `n`. This is precisely what parameterizes `p'`: every inhabitant of the storage of `C_b` can be canonically lifted to an inhabitant of its fiber by simply inserting `n` as the natural number in storage. Thus we have a family of sections parameterized by `n : N`.

```
Definition fp_rinv (n : N) : ContractMorphism C_b' C.
```

And we can prove that for all `n`, `fp_rinv n` is a section of our fiber bundle.

```
Theorem p_rinv_section (n : N) : compose_cm f_p (fp_rinv n) = pointed_include C_b'.
```

This family of sections is remeniscent of the sections of the trivial bundle we saw before, which were given by $b \mapsto (b, x)$ for $x \in F$. Because our fibers here are not isomorphic contracts, we do not have a decomposition of `C` as a product of `C_b` and some fixed `C_f'`. Even so, the family of sections indicates that `C` is a simple analogue of a product of contracts, which seems appropriate on reflection as `C` clearly separates the upgrade functionality from that corresponding to each contract version.

5.5.5 Summary

Let us reflect on the motivating example for this section, the Diamond upgrade framework of §5.1.1. Our main observation was that the specification uses diagrams and analogies to a diamond in order to communicate the different components of the upgradeable contract's functionality. The parts of the specification which deal with the process of adding or removing a facet are a description of the Diamond standard's base contract—the functionality governing how the contract can be changed. The parts of the specification which describe the structure the contract as a diamond, having many facets, describes the version contracts—the structure common to all possible versions of a Diamond contract.

Of course, producing an actual decomposition of the Diamond standard would likely be highly nontrivial. Instead, we would take the approach of writing the specification of an upgradeable contract explicitly in terms of its base and version contracts. Any implementation conscious of the required decomposition can then be crafted in such a way that the decomposition, using contract morphisms, yields itself easily.

5.6 Conclusion

Our goal in this chapter was to develop formal tools to evaluate the correctness of a specification of an individual contract upgrade or an upgradeable contract. This is because contract upgrades have nontrivial meta properties which can make correct specification of individual upgrades, as well as upgradeable contracts, a nontrivial endeavor. We argued that correct specification of contract upgradeability has at least two components, both of which we addressed with contract morphisms.

The first component is that in an upgrade, whether by a hard fork or within an upgradeable contract, the specification of the upgraded contract is typically written with the intention to relate to the specification of a previous version in some way. One can thus evaluate the correctness of the specification of an upgrade if one is able to formally express that intended relationship.

To address this component, we illustrated how contract morphisms can be used to clearly specify the intention of upgrades, by specifying properties of upgrades such as backwards compatibility. In particular, we revisited a major attack on a faulty contract upgrade, giving a simple metaspecification of the intended upgrade which clearly exposes the vulnerability. We also showed how one can prove Hoare-like properties of one contract using those of another.

The second component is that upgradeable contracts are difficult to specify with a prose specification because an upgradeable contract encodes the logic of two separate contracts—one that governs upgrades, and one that represents the contract's version at any given state. The relationship between these two contracts is complex, so to evaluate the correctness of a specification of an upgradeable contract we need some formal way to decompose it into its mutable and immutable parts.

Using contract morphisms we gave a formal and general decomposition of an upgradeable contract into a base contract and a family of version contracts. This rigorously and precisely articulates what it means for the contract to be upgradeable, including what the semantics of upgrades are, and it completely characterizes the forms that contract upgrades can take. Furthermore, there is a strong mathematical analogue to fiber bundles, a tool used in topology and geometry to understand the structure of a topological space in terms of a decomposition into a *base space* and a family of *fibers*.

By addressing both of these components with contract morphisms, we have a rigorous, mathematical treatment of upgradeability. This can help address vulnerabilities in contract upgrades by evaluating the correctness of specifications of individual contract upgrades, as well as upgradeable contracts generally.

Chapter 6

Contract Systems

The meta properties that we target in this chapter are those relating to the behavior of a system of contracts, taken as a whole. As we saw in §1.2.3, poorly specified systems of contracts can be vulnerable to extremely costly attacks. In this chapter we rigorously develop the notion of a specification's correctness as it relates to how the system of contracts behaves when taken as a whole.

To this end, we develop various notions of equivalences of contracts so that we can understand the specification of a system of contracts in terms of a monolithic contract, bisimilar to the system. This will allow us to to distinguish the specification of a system's infrastructure—a description of when and with what payload contracts in the system call each other—from the specification of its core, intended functionality when considering the contract system as a whole.

We proceed as follows. In §6.1, we motivate contract and system bisimulations by discussing the problem of specifying contract systems. In §6.2, we introduce the notion of a contract bisimulation as a precursor to a bisimulation of contract systems, and show that isomorphic contracts are also bisimilar. In §6.3, we formally define the notion of contract systems, using Milner's *bigraphs* [94] as our data type, to isolate the specification of the system infrastructure from its core functionality. In §6.4, we introduce bisimulations of contract systems. In §6.5, we conclude. Each section contains various code snippets; Appendix C mirrors section headings and gives a more complete version of the code from which the snippets are taken.

6.1 Contracts Systems

Financial smart contracts are ubiquitously implemented as modular systems of contracts rather than as monolithic contracts. One reason for this is purely practical. A blockchain is a highly resource-constrained platform on which to write programs due to gas fees, which are paid by the user [148]. Depending on the contract, it can be more gas-efficient to implement modularly, for example to prevent the storage of a contract from getting too large and thus expensive to access. In this case, implementing a contract modularly is an optimization technique.

Other reasons are more endemic. Systems of contracts are ubiquitous in financial smart contracts because new applications frequently build off of existing ones [130]. Indeed, one of the strenghts of smart contracts is that they are *composable*—so much so that decentralized finance is colloquially referred to as “money legos” [130], which refers to building new DeFi protocols by composing existing ones as if with lego bricks.

Recall for example that the implementation of Dexter2—indeed, that of any AMM—features a main, trading contract, and a secondary LP token contract [102]. The functionality of the main contract depends on that of the token contract, and conversely the token contract depends on the main trading contract. Furthermore, Dexter2 facilitates trading between a diversity of tokens, each of which has its own tokenomics (rules governing minting, burning, and distribution) and are themselves contracts independent from the AMM. One previously mentioned example of the ensuing complexity is that *governance tokens*—tokens that give the holder governance rights over an upgradeable contract or a DAO—can be traded on AMMs. In the exploit mentioned in §1.2.1, the Beanstalk attacker first traded for governance tokens on an AMM before executing flash loan attack (via yet another contract), winning a majority governance vote and draining the Beanstalk contract of its funds.

The examples continue. Yield aggregators like Harvest or Yearn [41] optimize over a set of existing DeFi protocols so that users can maximize yield farming. DEX aggregators like 1inch or Matcha minimize trading fees over various DEXs. Synthetics, including stablecoins like DAI, rely on price oracles to manage their tokenomics and maintain their peg [129]. CREAM, the multi-purpose DeFi protocol of §1.2.3, relies intimately on a whole web of contracts across multiple chains to offer its financial services.

In each of these examples, contract security and even correctness depends intimately on the accumulated and interconnected behavior of many interacting contracts [130]. Because of the overwhelming prevalence of contract systems, the very endeavor of smart contract verification can be rendered inept if we don’t have robust language to specify and reason about contract systems. Famously, however, contract composability is a source of extraordinary complexity in formal reasoning [103, 130].

6.1.1 Specifying Contract Systems

We focus on the complexity introduced to formal specification by contract composability. Consider what is needed to specify of a system of contracts. As well as specifying each individual contract in the system, we must inevitably specify when and with what payload contracts in the system call each other. From these specifications one must deduce, either by intuition or by other means, the intended behavior of the system of contracts when taken as a whole, or considering it as a single process.

We argue that specifying a system of contracts in this way is difficult to do correctly because the details of inter-contract communication can obfuscate the actual intended behavior of the system specification.

Recall the specification of the Diamond standard [95] from Chapter 5. The core, intended functionality is illustrated by pictures and naming conventions (*e.g.* facets, diamonds, loupes), precisely because the specification alone has many moving parts and is difficult to understand. If it were possible to somehow separate the specification of system infrastructure, including inter-contract communication, from that of the contract's core, desired functionality, the specification would likely be much easier to understand.

There is more at work here. One can see conceptually that the intended behavior of a contract system is generally articulated in terms of one, coherent process, which is agnostic to whether and how the system is modularized, or whether it is monolithic. Indeed one can imagine various implementations of a given specification, where some are modular and some are monolithic, but one has an intuitive notion that as long as the contracts of a given implementation behave together, coherently, according to the specification, that implementation is correct.

We make this rigorous by introducing formal tools to reduce the modular system of contracts to a monolithic contract, by way of a process algebra, such that the modular and monolithic contracts are bisimilar. We can then reason about our system in terms of that single contract. By using Milner's bigraphs [94] as our data type for contract systems, we can isolate the system infrastructure, allowing us to separate the specification of the interacting system from the specification of the core, desired contract behavior. From there, we have tools already at our disposal from previous chapters to rigorously specify and reason about the desired core contract behavior.

There are other reasons for which we may wish to consider contract bisimilarity. Consider the problem of specifying and verifying a contract which is highly optimized for performance. Performant code is frequently difficult to read, much less reason about—and conversely, contracts optimized for formal reasoning are seldom optimized for performance. Thus a contract on which it is feasible to articulate and verify meta properties might be too inefficient (and thus expensive) to feasibly deploy. In this scenario, we may wish to verify a contract optimized for formal reasoning, and then perform bisimulation-preserving optimizations on the verified contract before deploying it.

6.1.2 Related Work

While applying process-algebraic techniques to formal reasoning about smart contracts is still a relatively new practice, there is related work that reasons about systems of contracts and that draws in various ways on process-algebraic techniques.

One example is Tolmach *et al.*'s work on the formal analysis of composable DeFi protocols [130]. Tolmach *et al.* formulate a process-algebraic model of DEXs and tokens, abstracting each as primitives. These are used to analyze the behavior of contract systems by formally abstracting contract interactions and then using a model-checker to verify correctness properties. All of the correctness properties are properties of one or more components of the system.

Another example is 20squares, a smart contract auditing firm that uses compositional game theory [64] to analyze the economic properties of the specification of a composable set of Ethereum-based smart contracts. They are able to ask certain game-theoretic questions about contract specifications, such as how much certain agents have to be bribed to deviate from intended behavior [1]. In particular, they address system complexity by composing games which represent individual contracts, and studying the resulting incentives.

Finally, Madl *et al.* [86] reason about contract systems using interface automata, which is a formal way of specifying interactions and synchronization of input and output actions within a system.

Each of these examples takes a specification of a modular system and uses tools to reason in a composable manner over that system. Our contrasting approach is to separate the specification of the system infrastructure from that of the core functionality, which removes the complexity of the contract system before we formally reason about it. We do so by formally developing the notion of contract and system bisimulations, allowing us to consider contract specifications that are agnostic to whether or not, or how, the implementation is modularized.

6.2 Bisimulations of Contracts

The essential goal of this chapter is to formally define bisimulations of contract systems, but as a precursor our first task is to formally introduce bisimulations of individual contracts. As we will see, contract bisimulations extend naturally into bisimulations of contract systems. They are also a generalization of contract isomorphisms, which we saw briefly in Chapter 5.

6.2.1 Contract Trace Morphism

Recall from §5.3.1 that a contract's trace is a chained list of contract steps, where contract steps are the data for a successful call to the `receive` function. We restate the formal definitions of each here.

```
Record ContractStep (C : Contract Setup Msg State Error)
    (prev_cstate : State) (next_cstate : State) :=
    build_contract_step {
    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 6.1: Contract steps are successful calls to the `receive` function.

```
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.

A *contract trace morphism*, similar to a contract morphism, encodes a formal, structural relationship between the traces of two contracts. Like contract morphisms, a contract trace morphism features a function between contract storage types. However, unlike contract morphisms, which also features functions between message, error, and setup types, a contract trace morphism simply requires that the function between state types send intial states to inital states, and that for any two states connected by a step of the first contract, the corresponding states be also connected by a step in the second contract.

More concretely, for contracts

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

a morphism of contract traces includes the following data:

- A function `ct_state_morph : State1 -> State2`
- A proof that if there are inputs to the `init` function of `C1` that result in an initialized state `init_state`, then there are inputs to the `init` function of `C2` that result in a corresponding initialized state `(ct_state_morph init_state)`
- For states `state1` and `state2`, and any step forward of `C1`,

  `step1 : ContractStep C1 state1 state2`,

  we have a corresponding step forward of `C2` between the analogous states

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

```
Record ContractTraceMorphism
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :=
    build_ct_morph {
        (* a function of state types *)
        ct_state_morph : State1 -> State2 ;
        (* init state C1 -> init state C2 *)
        genesis_fixpoint : forall init_cstate,
            is_genesis_cstate C1 init_cstate ->
            is_genesis_cstate C2 (ct_state_morph init_cstate) ;
        (* coherence *)
        cstep_morph : forall state1 state2,
            ContractStep C1 state1 state2 ->
            ContractStep C2 (ct_state_morph state1) (ct_state_morph state2) ;
    }.
```

Listing 6.3: The formal definition of a morphism of contract traces.

Inductively, this gives us a relationship between reachable states: initial states of each contract are related via the function between state types, and from there, for any step of the first contract there is a related step of the second contract that respects the function on states.

Example 6.2.1 (The Identity Contract Trace Morphism)**.** One important contract trace morphism is the identity morphism `id_ctm`, which is defined for every contract `C` and inhabits the type

`ContractTraceMorphism C C.`

This is analogous to the identity morphism of Example 5.2.2. We give the formal definition here.

```
Definition id_ctm (C : Contract Setup1 Msg1 State1 Error1) :
    ContractTraceMorphism C C := {|
    ct_state_morph := id ;
    genesis_fixpoint := id_genesis_fixpoint C ;
    cstep_morph := id_cstep_morph C ;
|}.
```

Listing 6.4: The identity contract trace morphism `ic_ctm C` defined for any contract `C`.

The definition relies on a lemma, `id_genesis_fixpoint C`, and a function, `id_cstep_morph C`, which are both defined trivially.

```
Definition id_genesis_fixpoint (C : Contract Setup1 Msg1 State1 Error1)
    init_cstate (gen_C : is_genesis_cstate C init_cstate) :
    is_genesis_cstate C (id init_cstate) :=
    gen_C.
```

Listing 6.5: The genesis fixpoint result for the identity contract trace morphism.

```
Definition id_cstep_morph (C : Contract Setup1 Msg1 State1 Error1)
    state1 state2 (step : ContractStep C state1 state2) :
    ContractStep C (id state1) (id state2) :=
    step.
```

Listing 6.6: The step result for the identity contract trace morphism.

Example 6.2.2 (Equality of Contract Trace Morphisms)**.** Given two contract trace morphisms

`f g : ContractTraceMorphism C1 C2,`

like with contract morphisms we may ask ourselves whether or not they are equal.

Because of the dependent nature of the definition of contract trace morphisms, this definition is not quite as straightforward as equality of contract morphisms (Example 5.2.4). We must first assume equality of each state morphisms, and then we can state the (weaker) equality result, which states that we have equality of contract trace morphisms if the two functions between contract steps are equal.

```
Lemma eq_ctm_dep
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2)
    (* one single state morphism *)
    (ct_st_m : State1 -> State2)
    (gen_fix1 gen_fix2 : forall init_cstate,
        is_genesis_cstate C1 init_cstate ->
        is_genesis_cstate C2 (ct_st_m init_cstate))
    (cstep_m1 cstep_m2 : forall state1 state2,
        ContractStep C1 state1 state2 ->
        ContractStep C2 (ct_st_m state1) (ct_st_m state2)) :
    (* if the step morphisms are equal ... *)
    cstep_m1 = cstep_m2 ->
    (* then the contract trace morphisms are equal *)
    {| ct_state_morph := ct_st_m ;
       genesis_fixpoint := gen_fix1 ;
       cstep_morph := cstep_m1 ; |}
    =
    {| ct_state_morph := ct_st_m ;
       genesis_fixpoint := gen_fix2 ;
       cstep_morph := cstep_m2 ; |}.
```

Listing 6.7: Equality of contract trace morphisms.

Of course, a more sophisticated formulation of equality would transport over an equality of state morphisms, though we leave that for future work.

Example 6.2.3 (Contract Trace Morphism Composition)**.** Like contract morphisms, contract trace morphisms can be composed. Similar to the composition of contract morphisms (§5.2.1), we define composition 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.

```
Definition compose_ctm
    (m2 : ContractTraceMorphism C2 C3)
    (m1 : ContractTraceMorphism C1 C2) :
    ContractTraceMorphism C1 C3 :=
{|
    ct_state_morph := compose (ct_state_morph C2 C3 m2) (ct_state_morph C1 C2 m1) ;
    genesis_fixpoint := genesis_compose m2 m1 ;
    cstep_morph := cstep_compose m2 m1 ;
|}.
```

Listing 6.8: Composition of CT Morphisms.

This relies on two functions: `genesis_compose` and `cstep_compose`. However, since `genesis_fixpoint` is a simple implication and `cstep_morph` is a function, these simply compose each component.

```
Definition genesis_compose
    (m2 : ContractTraceMorphism C2 C3) (m1 : ContractTraceMorphism C1 C2) :
    forall init_cstate,
    is_genesis_state C1 init_cstate ->
    is_genesis_state C3
        (compose (ct_state_morph C2 C3 m2) (ct_state_morph C1 C2 m1) init_cstate).
```

Listing 6.9: The function `genesis_compose`

```
Definition cstep_compose
    (m2 : ContractTraceMorphism C2 C3) (m1 : ContractTraceMorphism C1 C2) :
    forall state1 state2,
    ContractStep C1 state1 state2 ->
    ContractStep C3
        (compose (ct_state_morph C2 C3 m2) (ct_state_morph C1 C2 m1) state1)
        (compose (ct_state_morph C2 C3 m2) (ct_state_morph C1 C2 m1) state2).
```

Listing 6.10: The function `cstep_compose`.

That composition is associative comes trivially.

```
Lemma compose_ctm_assoc
    (f : ContractTraceMorphism C1 C2)
    (g : ContractTraceMorphism C2 C3)
    (h : ContractTraceMorphism C3 C4) :
    compose_ctm h (compose_ctm g f) =
    compose_ctm (compose_ctm h g) f.
```

Listing 6.11: Composition of CT morphisms is associative.

Similarly, it comes immediately that composition on either side with the identity is a trivial operation.

```
Lemma compose_id_ctm_left (f : ContractTraceMorphism C1 C2) :
    compose_ctm (id_ctm C2) f = f.

Lemma compose_id_ctm_right (f : ContractTraceMorphism C1 C2) :
    compose_ctm f (id_ctm C1) = f.
```

Listing 6.12: Composition with the identity does nothing.

6.2.2 The Lifting Theorem

Our first result about contract trace morphisms is that contract morphisms carry all the information needed to define a contract trace morphism. Indeed, the coherence components of a contract morphism are simply stronger versions of the components of a contract trace morphism. We say that a contract morphism *lifts* to a contract trace morphism, and we prove this via a *lifting theorem*.

We prove this result by defining a function `cm_lift_ctm`, which takes a contract morphism

```
f : ContractMorphism C1 C2
```

and returns a contract trace morphism

```
cm_lift_ctm f : ContractTraceMorphism C1 C2,
```

which we define formally here.

```
Definition cm_lift_ctm (f : ContractMorphism C1 C2) : ContractTraceMorphism C1 C2 :=
    {|
        ct_state_morph := state_morph _ _ f ; (* use the state morph of f *)
        genesis_fixpoint := lift_genesis f ;  (* from f's init coherence  *)
        cstep_morph := lift_cstep_morph f ;   (* from f's recv coherence  *)
    |}.
```

Listing 6.13: Contract morphisms lift to contract trace morphisms.

As we can see in the definition, we use the function of contract states

```
state_morph _ _ f
```

from the contract morphism `f` to define the contract trace morphism.

We then use the `init` and `receive` coherence results of `f` to, respectively, prove the genesis fixpoint result and the contract step result. These are each proved and stated, respectively, as functions `lift_genesis` and `lift_cstep_morph`.

```
Definition lift_genesis (f : ContractMorphism C1 C2) :
    forall init_cstate,
        is_genesis_state C1 init_cstate ->
        is_genesis_state C2 (state_morph C1 C2 f init_cstate).
```

Listing 6.14: Using `init` coherence from `f`, we prove the genesis fixpoint result.

```
Definition lift_cstep_morph (f : ContractMorphism C1 C2) :
    forall state1 state2,
        ContractStep C1 state1 state2 ->
        ContractStep C2
            (state_morph C1 C2 f state1)
            (state_morph C1 C2 f state2).
```

Listing 6.15: Using `receive` coherence from `f`, we prove the contract step result.

Importantly, the identity contract morphism lifts to the identity contract trace morphism, and compositions of contract morphisms lift to compositions of contract trace morphisms. This will give us that isomorphic contracts are also bisimilar.

```
Theorem cm_lift_ctm_id :
    cm_lift_ctm (id_cm C1) = id_ctm C1.
```

Listing 6.16: Identity lifts to identity.

```
Theorem cm_lift_ctm_compose
    (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) :
    cm_lift_ctm (compose_cm g f) =                (* lifting the composition = ... *)
    compose_ctm (cm_lift_ctm g) (cm_lift_ctm f). (* composing the lifts *)
```

Listing 6.17: Compositions lift to compositions.

6.2.3 Contract Bisimulations

Our analogue to a contract isomorphism (Example 5.2.7) is now a contract *bisimulation*, or a contract trace isomorphism. A pair of contract trace morphisms form an isomorphism if they compose each way to the identity.

```
(** A bisimulation of contracts, or an isomorphism of contract traces *)
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 6.18: Formal definition of a bisimulation of contracts.

Let us briefly reflect on this as a definition of a bisimulation. Bisimilarity is a stable and mathematically natural concept formulated to capture the notion of equivalence between processes [71, 117].

Consider a labelled transition system $(S, \Lambda, \to)$, where S is a set of states, Λ is a set of labels, and $\to$ is a set of labelled transitions (a subset of $S \times \Lambda \times S$). Recall that 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 $(p', q') \in R$.

We can see that our formal definition of a bisimulation in Listing 6.18 achieves precisely this correspondence, as it gives a one-to-one correspondence on contract states and transitions such that corresponding transitions move between corresponding states.

With this definition in hand we can show an important corollary of the lifting theorem. Contract isomorphisms lift to contract bisimulations, due simply to the fact that the identity lifts to the identity, and compositions lift to compositions.

```
Corollary ciso_to_ctiso (f : ContractMorphism C1 C2) (g : ContractMorphism C2 C1) :
    is_iso_cm f g -> is_iso_ctm (cm_lift_ctm f) (cm_lift_ctm g).
```

Listing 6.19: Contract isomorphisms induce bisimulations of contracts.

We summarize with the bisimilarity predicate on contract pairs.

```
Definition contracts_bisimilar
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :=
    exists (f : ContractTraceMorphism C1 C2) (g : ContractTraceMorphism C2 C1),
    is_iso_ctm f g.
```

Listing 6.20: The bisimilarity predicate on contract pairs.

Finally, we point out that bisimilarity is an equivalence relation. We prove this with three results:

Reflexivity,

```
Lemma bisim_refl C : contracts_bisimilar C C.
```

symmetry,

```
Lemma bisim_sym C1 C2 :
    contracts_bisimilar C1 C2 ->
    contracts_bisimilar C2 C1.
```

and transitivity.

```
Lemma bisim_trans C1 C2 C3 :
    contracts_bisimilar C1 C2 /\ contracts_bisimilar C2 C3 ->
    contracts_bisimilar C1 C3.
```

6.2.4 Discussion: Propositional Indistinguishability

Contract bisimulations indicate a strong structural correspondence between bisimilar contracts. In particular, they require an isomorphism of contract states which is preserved under contract steps. This tells us that state invariants which are also invariant under the state isomorphism of the bisimulation carry over that bisimulation.

Example 6.2.4 (Propositional Indistinguishability)**.** Consider contracts `C1` and `C2`, and suppose that we have morphisms

`m1 : ContractTraceMorphism C1 C2` and `m2 : ContractTraceMorphism C2 C1`

which form a bisimulation of contracts, *i.e.* we have a witness

`C1_C2_bisim : is_iso_ctm m1 m2.`

Assume that both state types, `State1` and `State2`, have a constant in storage, which we have access to via functions

`const_in_stor_C1 : State1 -> nat` and `const_in_stor_C2 : State2 -> nat.`

Furthermore, suppose that constant is invariant over the state morphism components of `m1`, meaning that for all `st : State1`,

`const_in_stor_C1 st = const_in_stor_C2 (ct_state_morph C1 C2 m1 st).`

Finally, suppose that `C1` has an invariant relating to this constant, *e.g.* that for all contract-reachable states,

`const_in_stor_C1 st > 0.`

Using the bisimulation of contracts, we can prove that the same invariant holds with the following theorem:

Theorem 6.2.1. *For all reachable chain states* `bstate` *with* `C2` *deployed at an address* `caddr` *with state* `cstate`*, the inequality* `const_in_stor_C2 cstate > 0` *holds.*

```
Theorem invariant_C2 bstate caddr (trace : ChainTrace empty_state bstate):
    (* Forall reachable states with contract at caddr, *)
    env_contracts bstate caddr = Some (C2 : WeakContract) ->
    (* such that cstate is the state of the contract, *)
    exists (cstate : State2),
    contract_state bstate caddr = Some cstate /\
    (* the constant in storage is > 0 *)
    (const_in_stor_C2 cstate > 0)%nat.
```

Listing 6.21: caption text

We prove this by contract induction—since every initial state of `C2` corresponds to an initial state of `C1` under the state isomorphism of the bisimulation, and all steps of `C1` correspond to isomorphic steps of `C2`, at each step of contract induction we can take advantage of the bisimulation and the state isomorphism to prove our invariant.

Of course, contracts can be bisimilar but not identical, so it will not likely be true that *any* proposition holding for one holds for the other. Even so, this example is an illustration toward understanding the degree to which bisimilar contracts are propositionally indistinguishable. We leave any further, formal analysis of the degree to which bisimilar contracts are also propositionally indistinguishabile to future work.

6.3 Contract Systems as Bigraphs

With bisimulations of contracts formally defined, we now move on to consider contract systems. Our goal will be to represent a system of interacting contracts as a single process in ConCert, which we can then specify and reason about as a single entity. The corresponding notions of bisimilarity will build off of what we have just seen.

To do so, we require a data type in which to represent systems of contracts, as well as a formal way to separate the specification of a contract system's infrastructure—a specification of which contracts pass messages to which other contracts, and when—from the specification of its core functionality when considering the contracts as a whole. The data type that we will use is a *bigraph*, which we define in the following section.

6.3.1 Bigraphs

We give a very brief introduction to bigraphs here and direct the reader to some of Milner's earlier writings [93, 94] for a full exposition. A bigraph is a universal mathematical model for representing the spatial configuration of physical or virtual objects, their interaction capabilities, and temporal evolution [120]. A bigraph consists of a set of nodes, denoting processes, and defines two independent structures on those nodes, which are:

- The *place graph*, a directed forest representing a spacial or nesting relationship between nodes
- The *link graph*, a hypergraph representing interactions between nodes.

Bigraphs have been the subject of both theoretical and practical work to reason about systems of processes [119, 120, 88]. Because they facilitate generic, process-algebraic approaches to reasoning about smart contracts, we will use bigraphs as a data type here in which to encode contract systems, treating individual contracts as nodes. We proceed to formalize the place graph and discuss the link graph.

6.3.2 The Place Graph

The place graph of a contract system indicates a nesting or hierarchical structure within the contract system. We argue that this is a natural way to conceptualize contract systems, and thus a natural data type for contract systems as opposed to *e.g.* lists.

Consider an AMM, one of our most basic and important financial smart contracts, which facilitates trades between token contracts. Conceptually, we might think of token contracts being nested inside of the process of the AMM. From the AMM's perspective, the interface to interacting with the tokens is the trading contract; the trading contract then propogates transactions into its nested processes when it

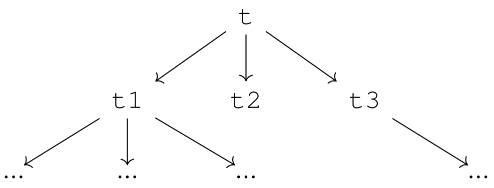

Figure 6.1: A visualization of the place graph `Node t [t1 ; t2 ; t3]`. Nodes `t1` and `t3` have further child n-trees, while `t2` is a leaf.

executes trades. This lends intuition to understanding how the system works, which is a good thing, but it may have practical applications as well. If one models multiple AMMs on the same blockchain, one can see visually how these AMMs interact with each other by where their place graphs intersect—along token or other contracts.

Other contract systems that we might consider include yield aggregators, like Harvest Finance or other one-stop-shop DeFi protocols like CREAM, both from the exploits of §1.2.3. Yield and DEX aggregators have an interface contract, and then try to optimize whatever operation they do—whether that is looking for highest yield pools or best price for trades. In both cases, the interface contract can be thought of as the root node of the place graph, and the processes across which it tries to optimize can be thought of as nested within that node. A similar argument goes for one-stop-shop DeFi protocols such as CREAM, which provide a single, united interface that builds off of other DeFi protocols by forwarding messages. The interface contract has nested within it the contracts that form its backend.

Finally, thinking of DeFi more broadly as a set of highly interconnected financial contracts, dubbed "money legos," a new DeFi contract that builds off of previous contracts can be visualized as having those contracts on which it builds nested within the new contract.

With all this in mind, we proceed to define an inductive data structure for contract systems, where contracts are nodes. We first give a formal definition of an n-ary tree, or n-tree, over an arbitrary type `T`.

```
Inductive ntree (T : Type) : Type :=
| Node : T -> list (ntree T) -> ntree T.
```

Listing 6.22: The formal definition of an n-tree.

For example, an n-tree of the form

```
Node t [t1 ; t2 ; t3]
```

is a node inhabited by `t : T`, and has three child trees `t1`, `t2`, and `t3`, each of which can have any number of their own child trees. We call the node `t` the *root*. See Figure 6.1.

```
Definition ContractPlaceGraph (Setup Msg State Error : Type) :=
    ntree (Contract Setup Msg State Error).
```

Listing 6.23: The formal definition of a contract system via its place graph.

A contract system's place graph is then simply defined to be an n-tree whose nodes are contracts. Of course, this formal definition is of a homogeneous tree, only allowing for nodes to be contracts parameterized by the same four types. In practice, systems of contracts are not homogeneous, but we will show presently how to construct a homogeneous system from a heterogeneous one.

Once defined, a contract system

```
sys : ContractPlaceGraph Setup Msg State Error
```

can be initialized and called, like a contract, but with custom `init` and `receive` functions `sys_init` and `sys_receive`. To initialize a system, we initialize the root contract with some data `s : Setup`. If successful, this returns a state `st : State`, which we call the state of the system.

```
Definition sys_init
    (sys : ContractPlaceGraph Setup Msg State Error)
    (c : Chain)
    (ctx : ContractCallContext)
    (s : Setup) : result State Error :=
    match sys with
    | Node _ root_contract _ =>
        init root_contract c ctx s
    end.
```

Listing 6.24: System initialization.

Then to call a system, we give it a state `st : State` and an optional message `op_msg : option Msg` and, if successful, it returns an updated state and a list of emitted actions.

```
Definition sys_receive
    (sys : ContractPlaceGraph Setup Msg State Error)
    (c : Chain)
    (ctx : ContractCallContext)
    (st : State)
    (op_msg : option Msg) : result (State * list ActionBody) Error :=
    ntree_fold_left
    (fun (recv_propogate : result (State * list ActionBody) Error)
         (contr : Contract Setup Msg State Error) =>
         match recv_propogate with
            | Ok (st0, lacts0) =>
                match receive contr c ctx st0 op_msg with
                | Ok (st1, lacts1) => Ok (st1, lacts0 ++ lacts1)
                | Err e => Err e
                end
            | Err e => Err e
         end)
    sys
    (Ok (st, nil)).
```

Listing 6.25: Calls to a contract system are governed by the `sys_receive` function.

The sys_receive function iteratively calls *every contract* in the system with the given message. Each call either returns an updated state or an error, and we either fold over all of those updates to the state, or propagate the error. We also accumulate all emitted actions. This is done with the n-tree fold function.

```
Fixpoint ntree_fold_left {A T}
    (f : A -> T -> A)
    (sys : ntree T)
    (a0 : A) : A :=
    match sys with
    | Node _ t list_child_trees =>
        List.fold_left
            (fun (a0' : A) (sys' : ntree T) =>
                ntree_fold_left f sys' a0')
            list_child_trees
            (f a0 t)
    end.
```

Listing 6.26: Folding over an n-tree.

These semantics for a contract system may not seem obviously intuitive; that they accurately describe the behavior of an actual system is made more clear when we define the iterative process of building an n-tree of contracts.

6.3.2.1 Iteratively Building a Contract System

As we mentioned previously, a contract system is parameterized by the same four types that parameterize a contract, and the definition requires that all the contracts in a system be parameterized by the same four types. Of course, this does not reflect actual systems of contracts, so we define an iterative building technique to take heterogeneous contracts and make them homogeneous.

We define two functions, c_sum_l and c_sum_r. The aim of each is to produce a contract which preserves the essential functionality of, respectively, the left (C1) and right (C2) contracts of the pair of contracts (C1 C2) given it as an input.

Each takes a pair of contracts C1 and C2 of distinct types, and returns a contract whose type is the *product* of the setup and state types, and the *sum* of the message and error types

```
    Contract (Setup1 * Setup2) (Msg1 + Msg2) (State1 * State2) (Error1 + Error2).
```

The intuition behind this is that the state of a system requires state values for each constituent contract; likewise, to initialize a system we need setup data for each constituent contract. But to call the contract, we only need a message for the specific contract we wish to call. Likewise if a call results in an error it only need be an error the constituent contract that was called.

We give the formal definition of each function below.

```
Definition c_sum_l
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :
    Contract (Setup1 * Setup2) (Msg1 + Msg2) (State1 * State2) (Error1 + Error2).
```

Listing 6.27: `c_sum_l` produces a contract with the essential functionality of the left contract, `C1`.

```
Definition c_sum_r
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :
    Contract (Setup1 * Setup2) (Msg1 + Msg2) (State1 * State2) (Error1 + Error2).
```

Listing 6.28: `c_sum_r` produces a contract with the essential functionality of the right contract, `C2`.

A call to `c_sum_l C1 C2` takes a state `(st1, st2) : State1 * Stat2`. Given a message `inl msg`—that is, a message `msg : Msg1` to `C1`—it calls the `receive` function of `C1` on `st1` and `msg`. If the call to `(receive C1)` is successful, returning an updated state `st1'`, we return `(st1', st2)` and any emitted actions. If given a message `inr msg`—that is, a message `msg : Msg2` to `C2`—the call does nothing, returning `(st1, st2)` and no emitted actions.

By executing along the semantics of `C1` when called with a message to `C1`, and doing nothing otherwise, the function `c_sum_l` is meant to give us a contract that represents `C1` within the context of `C1` and `C2` together. A call to `c_sum_r C1 C2` executes in an alogous fashion, but calling `C2` with a message `inr msg` and doing nothing otherwise.

Using `c_sum_l` and `c_sum_r`, we can iteratively build systems. First, we define the simple example of nesting a contract `C2` within another contract `C1`. We might do this, for example, if `C1` is an interface contract and `C2` is a backend contract.

```
Definition nest
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :
    ContractPlaceGraph (Setup1 * Setup2) (Msg1 + Msg2) (State1 * State2) (Error1 + Error2)
    := let T :=
    Contract (Setup1 * Setup2) (Msg1 + Msg2) (State1 * State2) (Error1 + Error2) in
    Node T (c_sum_l C1 C2) [Node T (c_sum_r C1 C2) nil].
```

Listing 6.29: A function to nest a contract `C2` within another contract `C1`.

The function `nest` takes two contracts `C1` and `C2`, and takes their image under `c_sum_l` and `c_sum_r`, producing a place graph

```
nest C1 C2 := Node T (c_sum_l C1 C2) [ Node T (c_sum_r C1 C2) nil ].
```

The root contract, `c_sum_l C1 C2`, represents `C1`, and the child contract, `c_sum_r C1 C2`, represents `C2`. The place graph `nest C1 C2` can receive messages of the form `inl msg`, targeting `C1`, or `inr msg`,

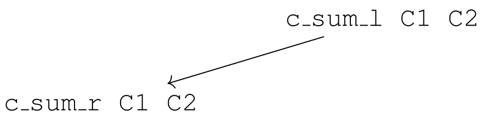

Figure 6.2: A contract `C2` nested within another contract `C1`.

targeting `C2`, and the state of `nest C1 C2` is a witness of the state of `C1` as well as the state of `C2`. The system `nest C1 C2` makes progress as its constituents, `C1` and `C2`, are called successfully and update the system's state. See Figure 6.2.

Moving to a slightly more complicated example, we can iteratively add children to an existing contract system. We start with the singleton system, which is simply a singleton n-tree, or an n-tree with only one node, of the form `Node _ C nil`.

```
Definition singleton_place_graph
    (C : Contract Setup Msg State Error)
    : ContractPlaceGraph Setup Msg State Error :=
    Node _ C nil.
```

From here, we can add children to the singleton system via a function `sys_add_child_r`.

```
Definition sys_add_child_r
    (sys : ContractPlaceGraph Setup1 Msg1 State1 Error1)
    (C : Contract Setup2 Msg2 State2 Error2) :
    ContractPlaceGraph (Setup1 * Setup2) (Msg1 + Msg2) (State1 * State2) (Error1 + Error2)
    :=
    let T := Contract (Setup1 * Setup2) (Msg1 + Msg2) (State1 * State2) (Error1 + Error2)
    in
    match sys with
    | Node _ root_contract _ =>
        match (ntree_map (fun C1 => c_sum_l C1 C) sys) with
        | Node _ root_contract' children =>
            Node T root_contract' (children ++ [Node T (c_sum_r root_contract C) nil])
        end
    end.
```

Listing 6.30: Iterataively add a child `C` to a contract system `sys`.

Given a contract system `sys`, we iteratively add a child `C` to the system by first mapping over `sys` with `c_sum_l _ C`, or rather

`(fun C1 => c_sum_l C1 C),`

and then appending the right sum of the root contract `root_contract` and `C`,

`c_sum_r root_contract C,`

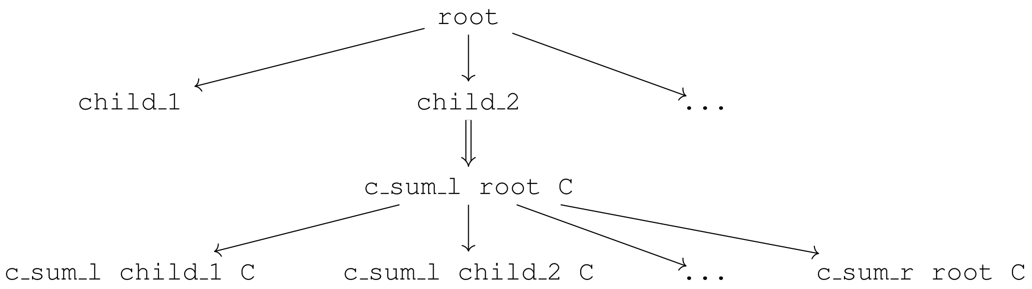

Figure 6.3: Appending a child `C` to a system `Node root [child_1 ; child_2 ; ... ]`.

to the list of children of the root contract. This maintains the same functionality of each node inherited from `sys`, while adding `C` to the system, by iteratively applying `c_sum_l` and `c_sum_r`. See Figure 6.3.

6.3.2.2 System Contracts, Morphisms, and Isomorphisms

System place graphs are inductive structures built on contracts, so naturally we might consider morphisms of contract systems, building off of contract morphisms. In this context, our main goal is to establish notions of equivalences of system place graphs analogous to what we saw in §6.2.3.

Like contract morphisms, a morphism of system place graphs consists of four component morphisms, along with coherence conditions for `sys_init` and `sys_receive`.

```
(** A morphism of system place graphs *)
Record SystemMorphism
    (sys1 : ContractPlaceGraph Setup1 Msg1 State1 Error1)
    (sys2 : ContractPlaceGraph Setup2 Msg2 State2 Error2) :=
    build_system_morphism {
        (* the components of a morphism *)
        sys_setup_morph : Setup1 -> Setup2 ;
        sys_msg_morph   : Msg1   -> Msg2   ;
        sys_state_morph : State1 -> State2 ;
        sys_error_morph : Error1 -> Error2 ;
        (* coherence conditions *)
        sys_init_coherence : forall c ctx s,
            result_functor sys_state_morph sys_error_morph
                (sys_init sys1 c ctx s) =
            sys_init sys2 c ctx (sys_setup_morph s) ;
        sys_recv_coherence : forall c ctx st op_msg,
            result_functor (fun '(st, l) => (sys_state_morph st, l)) sys_error_morph
                (sys_receive sys1 c ctx st op_msg) =
            sys_receive sys2 c ctx (sys_state_morph st) (option_map sys_msg_morph op_msg)
    ;
}.
```

Listing 6.31: The formal definition of a morphism of system place graphs.

As before, composition is given by composition of component morphisms, which we formalize as follows, and composition is associative.

```
Definition compose_sm (g : SystemMorphism sys2 sys3) (f : SystemMorphism sys1 sys2) :
  SystemMorphism sys1 sys3 := {|
  (* the components *)
  sys_setup_morph := compose (sys_setup_morph sys2 sys3 g) (sys_setup_morph sys1 sys2 f) ;
  sys_msg_morph   := compose (sys_msg_morph   sys2 sys3 g) (sys_msg_morph   sys1 sys2 f) ;
  sys_state_morph := compose (sys_state_morph sys2 sys3 g) (sys_state_morph sys1 sys2 f) ;
  sys_error_morph := compose (sys_error_morph sys2 sys3 g) (sys_error_morph sys1 sys2 f) ;
  (* the coherence results *)
  sys_init_coherence := sys_compose_init_coh g f ;
  sys_recv_coherence := sys_compose_recv_coh g f ;
|}.
```

Listing 6.32: Composition of morphisms of system place graphs.

The identity system morphism is given by identity component functions, given as follows.

```
Definition id_sm (sys : ContractPlaceGraph Setup Msg State Error) :
    SystemMorphism sys sys := {|
    (* components *)
    sys_setup_morph := id ;
    sys_msg_morph   := id ;
    sys_state_morph := id ;
    sys_error_morph := id ;
    (* coherence conditions *)
    sys_init_coherence := sys_init_coherence_id sys ;
    sys_recv_coherence := sys_recv_coherence_id sys ;
|}.
```

Listing 6.33: The identity morphism of system place graphs.

Similar to before we define an isomorphism of systems as a pair of morphisms that compose each way to the identity.

```
Definition is_iso_sm (m1 : SystemMorphism sys1 sys2) (m2 : SystemMorphism sys2 sys1) :=
    compose_sm m2 m1 = id_sm sys1 /\
    compose_sm m1 m2 = id_sm sys2.
```

Listing 6.34: An isomorphism of contract systems.

The reader may notice that system morphisms look almost identical to contract morphisms. Because system place graphs have the `sys_init` and `sys_receive` functions, which mimic the `init` and `receive` of contracts, we might ask whether systems are themselves contracts.

Indeed, this is the case. We can define the *system contract*, which is the contract in ConCert that represents a contract system.

```
Definition sys_contract (sys : ContractPlaceGraph Setup Msg State Error) :=
    build_contract (sys_init sys) (sys_receive sys).
```

Listing 6.35: The system contract.

Furthermore, system morphisms are in one-to-one correspondence with the morphisms of the corresponding contracts. We have two functions—one that takes system morphisms to contract morphisms,

```
Definition sysm_to_cm
    (sys1 : ContractPlaceGraph Setup1 Msg1 State1 Error1)
    (sys2 : ContractPlaceGraph Setup2 Msg2 State2 Error2)
    (f : SystemMorphism sys1 sys2) : ContractMorphism (sys_contract sys1) (sys_contract
    sys2).
```

and one that takes contract morphisms to system morphisms—

```
Definition cm_to_sysm
    (sys1 : ContractPlaceGraph Setup1 Msg1 State1 Error1)
    (sys2 : ContractPlaceGraph Setup2 Msg2 State2 Error2)
    (f : ContractMorphism (sys_contract sys1) (sys_contract sys2)) : SystemMorphism sys1
    sys2.
```

which compose each way to the identity. In this correspondence, the identity corresponds to the identity, and compositions to compositions.

6.3.3 The Link Graph

Now that we have the data type for contract systems as its place graph, let us turn our attention to a contract system's link graph. In a bigraph, the link graph represents the interactions of processes in a system, as opposed to the place graph which only represents a spacial, or hierarchical relationship between processes.

For contract systems, a link graph has an obvious definition, at least intuitively. That is, that for contracts `C1` and `C2` in a contract system, there is an edge

$$\mathtt{C1} \longrightarrow \mathtt{C2}$$

when there is a contract step of `C1` which emits a call to `C2`. Even more, because of the semantics of a blockchain, that call to `C1` succeeds only if the emitted call to `C2` succeeds. Indeed, a contract call succeeds if and only if it doesn't throw an error and all the calls which it emits succeed.

Furthermore, the place graph is essential to the semantics of a system of contracts. Going back to the example of an interface and backend contract forming a system, that the link graph connect calls to the interface to (permissioned) calls to the backend is essential to the system functioning as intended. Indeed, the link graph dictates how systems move forward, grouping and ordering contract calls that constitute a genuine step forward for the system. Depending on the link graph structure, which itself depends on contract addresses and message-passing, a contract system can behave in very different ways.

A genuine step forward for a system, then, is a full traversal of the link graph from the point of entry of an initiating call. Such a traversal is a sequence of recursive calls to `sys_receive`, each of which is a step forward for the system, such that in the end the accumulated, emitted transactions do not include any system-recursive calls.

6.3.3.1 System Steps and System Trace

To encode the semantics of a link graph into a contract system, we need first to formalize the semantics of a system stepping forward, and its trace.

Consider a system of interacting contracts that has a frontend, or interface, contract, and a backend contract. Messages come in through the frontend, whose storage has no meaningful data to the system, and are simply forwarded to the backend, where the meaningful part of the contract system's storage is. A transaction coming in through the interface only really has full meaning for the system when the ensuing chain of transactions is completed.

Now compare such a system to its monolithic counterpart—where the interface and backend are housed in the same contract and there is no need for any message-passing—then the contract call corresponding to the call to the interface and message to the backend would be a single call and execution to the monolithic contract which updates the storage.

Because of this, we distinguish between incremental, or single, steps, and system steps. The distinction is that a system step can be one, many, or no incremental steps. An incremental step is defined to be a single, successful call to the contract system.

```
Record SingleSystemStep (sys : ContractPlaceGraph Setup Msg State Error)
    (prev_sys_state next_sys_state : State) :=
    build_sys_single_step {
    sys_step_chain : Chain ;
    sys_step_ctx : ContractCallContext ;
    sys_step_msg : option Msg ;
    sys_step_nacts : list ActionBody ;
    (* we can call receive successfully *)
    sys_recv_ok_step :
        sys_receive sys sys_step_chain sys_step_ctx prev_sys_state sys_step_msg =
        Ok (next_sys_state, sys_step_nacts) ;
}.
```

Listing 6.36: An incremental, or single, step of a contract system.

Irrespective of the contract system in question, any meaningful step forward for a system must be expressible as a chained list of single system steps, and thus has semantics in the following type.

```
Definition ChainedSingleSteps (sys : ContractPlaceGraph Setup Msg State Error) :=
    ChainedList State (SingleSystemStep sys).
```

Listing 6.37: A system step can be one or several incremental steps.

On the other hand, it is not true that *any* chained list of system steps constitutes a meaningful step forward. We must instead define what it means for a contract system to meaningfully step forward by defining a function,

```
sys_link :  State -> State -> Type,
```

which links two system states by a system step forward. Those steps must have semantics as chained single steps via a function of the form

```
sys_link_semantics st1 st2 :  sys_link st1 st2 -> ChainedSingleSteps _ st1 st2.
```

Such a definition of system steps is in fact a *specification* because whether or not a system actually satisfies these semantics for stepping forward depends on conditions of a given state of the chain, including contract addresses and contract storage. This is our definition of a system's link graph.

From here, we can give a full definition of a contract system as a bigraph: a set of contracts, encoded in a place graph, with link graph semantics.

```
Record ContractSystem (Setup Msg State Error : Type) :=
    build_contract_system {
        (* the place and link graphs *)
        sys_place : ContractPlaceGraph Setup Msg State Error ;
        sys_link  : State -> State -> Type ;
        (* the link graph has semantics in ChanedSingleSteps *)
        sys_link_semantics : forall st1 st2,
            sys_link st1 st2 ->
            ChainedSingleSteps sys_place st1 st2 ;
    }.
```

Given this definition of a contract system, we can define a type of system steps for any system `sys`, which are steps given by the link graph semantics.

```
Definition SystemStep (sys : ContractSystem Setup Msg State Error) :=
    sys_link _ _ _ _ sys.
```

Listing 6.38: The type of system steps.

Finally, a system trace is a chained list of system steps.

```
Definition SystemTrace (sys : ContractSystem Setup Msg State) :=
    ChainedList (SystemState State) (SystemStep sys).
```

Listing 6.39: A contract system's trace is a chained list of system steps.

As we did for contracts in §6.2.3, we use the definitions of system steps and system trace to reason about bisimulations of systems.

6.4 Bisimulations of Contract Systems

With the data type for contract systems in place, we are able to now define bisimulations of contract systems, building off of §6.2.1. Like bisimulations of contracts, a bisimulation of contract systems is a correspondence between system states, and steps forward in a system state. Because system steps are defined by the link graph, a bisimulation of systems indicates a correspondence between the link graph structure of two contract systems. As before, we will consider a system's trace, and system trace morphisms, to establish bisimulations.

6.4.1 System Trace Morphisms and System Bisimulations

A system trace morphism between systems sys1 and sys2 consists of a function from the state type of sys1 to that of sys2 which sends initial states to initial states, and steps defined by the link graph of sys1 to corresponding steps defined by the link graph of sys2.

```
Record SystemTraceMorphism
    (sys1 : ContractSystem Setup1 Msg1 State1 Error1)
    (sys2 : ContractSystem Setup2 Msg2 State2 Error2) :=
    build_st_morph {
        (* a function *)
        st_state_morph : State1 -> State2 ;
        (* init state sys1 -> init state sys2 *)
        sys_genesis_fixpoint : forall init_sys_state,
            is_genesis_sys_state sys1 init_sys_state ->
            is_genesis_sys_state sys2 (st_state_morph init_sys_state) ;
        (* step morphism *)
        sys_step_morph : forall sys_state1 sys_state2,
            SystemStep sys1 sys_state1 sys_state2 ->
            SystemStep sys2 (st_state_morph sys_state1) (st_state_morph sys_state2) ;
    }.
```

Listing 6.40: A morphism of system traces.

We can define composition analogously to contract trace morphisms, and as before, composition is associative. Composition relies on two functions, `sys_genesis_compose` and `sys_step_compose`, which are also defined analogously to their counterparts in contract trace morphisms.

```
Definition compose_stm
    (m2 : SystemTraceMorphism sys2 sys3)
    (m1 : SystemTraceMorphism sys1 sys2) : SystemTraceMorphism sys1 sys3 := {|
    st_state_morph := compose (st_state_morph _ _ m2) (st_state_morph _ _ m1) ;
    sys_genesis_fixpoint := sys_genesis_compose m2 m1 ;
    sys_step_morph := sys_step_compose m2 m1 ;
|}.
```

Listing 6.41: Composition of system trace morphisms.

The dependent notion of equality of system trace morphisms also mirrors that of contract trace morphisms.

```
Lemma eq_stm_dep
    (sys1 : ContractSystem Setup1 Msg1 State1 Error1)
    (sys2 : ContractSystem Setup2 Msg2 State2 Error2)
    (st_st_m : State1 -> State2)
    sys_gen_fix1 sys_gen_fix2
    (sys_step_m1 sys_step_m2 : forall sys_state1 sys_state2,
        SystemStep sys1 sys_state1 sys_state2 ->
        SystemStep sys2 (st_st_m sys_state1) (st_st_m sys_state2)) :
    sys_step_m1 = sys_step_m2 ->
    {| st_state_morph := st_st_m ;
       sys_genesis_fixpoint := sys_gen_fix1 ;
       sys_step_morph := sys_step_m1 ; |}
    =
    {| st_state_morph := st_st_m ;
       sys_genesis_fixpoint := sys_gen_fix2 ;
       sys_step_morph := sys_step_m2 ; |}.
```

Listing 6.42: Equality of system trace morphisms.

Finally, the identity morphism is defined similarly to that of contract trace morphisms.

```
Definition id_stm
    (sys : ContractSystem Setup Msg State Error) : SystemTraceMorphism sys sys :=
{|
    st_state_morph := id ;
    sys_genesis_fixpoint := id_sys_genesis_fixpoint sys ;
    sys_step_morph := id_sys_step_morph sys ;
|}.
```

Listing 6.43: The identity system trace morphism.

It relies on two functions, which are both defined trivially.

```
Definition id_sys_genesis_fixpoint (sys : ContractSystem Setup Msg State Error)
    init_sys_state
    (gen_sys : is_genesis_sys_state sys init_sys_state) :
    is_genesis_sys_state sys (id init_sys_state) :=
    gen_sys.
```

Listing 6.44: The genesis fixpoint result for the identity system trace morphism.

```
Definition id_sys_step_morph (sys : ContractSystem Setup Msg State Error)
    sys_state1 sys_state2 (step : SystemStep sys sys_state1 sys_state2) :
    SystemStep sys (id sys_state1) (id sys_state2) :=
    step.
```

Listing 6.45: The step result for the identity system trace morphism.

We then define system trace isomorphisms as morphisms which compose each way to the identity.

```
Definition is_iso_stm
    (m1 : SystemTraceMorphism sys1 sys2) (m2 : SystemTraceMorphism sys2 sys1) :=
    compose_stm m2 m1 = id_stm sys1 /\
    compose_stm m1 m2 = id_stm sys2.
```

Listing 6.46: An isomorphism of system trace morphisms is a pair of system trace morphisms which compose each way to the identity.

And finally, a bisimulation of systems is an isomorphism of system traces.

```
Definition systems_bisimilar
    (sys1 : ContractSystem Setup1 Msg1 State1 Error1)
    (sys2 : ContractSystem Setup2 Msg2 State2 Error2) :=
    exists (f : SystemTraceMorphism sys1 sys2) (g : SystemTraceMorphism sys2 sys1),
    is_iso_stm f g.
```

Listing 6.47: The formal definition of a bisimulation of contract systems.

A note on system bisimulations. Because system traces are defined by a system's link graph, in order for a bisimulation of contract systems to have legitimate semantic meaning, the system must satisfy the specification given by its link graph. This will typically mean that constituent contracts of the system are deployed at certain addresses, and that contracts are able to call each other, *e.g.* by having the appropriate addresses in storage so that they can emit transactions which call other contracts in the system. Crucially, a bisimulation of systems only reflects an equivalence for the behavior of a system which is deployed in such a way that it satisfies the place graph.

6.4.2 Lifting Theorems for Contract Systems

We have various lifting theorems for system morphisms to system trace morphisms which generalize how contract morphisms lift to contract trace morphisms.

System morphisms indicate a relationship between single system steps of two systems, and so from a system morphism we can derive a correspondence between all single system steps. Thus system morphisms lift to system trace morphisms if the link graph structure on each system is compatible with regards to individual system steps. In particular, if both system steps have the *discrete* link graph, for which all system steps are single system steps, then system morphisms lift to system trace morphisms.

More formally, given a system place graph, we can construct the discrete link graph, which is a link graph where all system steps are given by single system steps.

```
Definition discrete_link (sys : ContractPlaceGraph Setup Msg State Error) st1 st2 :=
    SingleSystemStep sys st1 st2.

Definition discrete_link_semantics (sys : ContractPlaceGraph Setup Msg State Error)
    st1 st2 (step : discrete_link sys st1 st2) :
    ChainedSingleSteps sys st1 st2 :=
    (snoc clnil step).

Definition discrete_sys (sys : ContractPlaceGraph Setup Msg State Error) := {|
    sys_place := sys ;
    sys_link := discrete_link sys ;
    sys_link_semantics := discrete_link_semantics sys ;
|}.
```

Listing 6.48: The discrete link graph construction on any contract place graph.

The lifting theorem for system trace morphisms is that a system morphism lifts to a system trace morphism of discrete systems.

```
Definition sm_lift_stm (f : SystemMorphism sys1 sys2) :
    SystemTraceMorphism (discrete_sys sys1) (discrete_sys sys2).
```

Listing 6.49: A function that lifts system morphisms to system trace morphisms.

We have that the identity lifts to the identity,

```
Theorem sm_lift_stm_id :
    sm_lift_stm (id_sm sys1) = id_stm (discrete_sys sys1).
```

Listing 6.50: The identity system morphism lifts to the identity system trace morphism under the discrete link graph.

and compositions to compositions.

```
Lemma sm_lift_stm_compose (g : SystemMorphism sys2 sys3) (f : SystemMorphism sys1 sys2) :
    sm_lift_stm (compose_sm g f) =
    compose_stm (sm_lift_stm g) (sm_lift_stm f).
```

Listing 6.51: Compositions of system morphisms lift to compositions of system trace morphisms.

Thus isomorphic systems are also bisimilar, under the discrete link graph.

```
Corollary sys_iso_to_bisim
    (sys1 : ContractPlaceGraph Setup1 Msg1 State1 Error1)
    (sys2 : ContractPlaceGraph Setup2 Msg2 State2 Error2) :
    systems_isomorphic sys1 sys2 ->
    systems_bisimilar (discrete_sys sys1) (discrete_sys sys2).
```

Listing 6.52: Isomorphism contract systems are bisimilar under the discrete link graph.

Furthermore, because system morphisms correspond to contract morphisms of the system contract, a contract morphism lifts to a system morphism of singleton systems (systems with only one constituent contract). We define this with a function `lift_cm_to_sm`.

```
Definition lift_cm_to_sm (f : ContractMorphism C1 C2) :
    SystemMorphism (singleton_place_graph C1) (singleton_place_graph C2).
```

Listing 6.53: A contract morphism lifts to a system morphism of singleton systems.

Similarly, contract morphisms lift to system trace morphism on the singleton system.

```
Definition lift_ctm_to_stm (f : ContractTraceMorphism C1 C2) :
    SystemTraceMorphism
        (discrete_sys (singleton_place_graph C1))
        (discrete_sys (singleton_place_graph C2)).
```

The identity contract morphism lifts to the identity system morphism on the singleton place graph,

```
Lemma lift_id_cm_to_id_sm :
    lift_cm_to_sm (id_cm C) = id_sm (singleton_place_graph C).
```

Listing 6.54: The identity contract moprhism lifts to the identity system morphism of singleton systems.

and compositions lift to compositions.

```
Lemma lift_cm_to_sm_comp
    (f : ContractMorphism C1 C2) (g : ContractMorphism C2 C3) :
    lift_cm_to_sm (compose_cm g f) = compose_sm (lift_cm_to_sm g) (lift_cm_to_sm f).
```

Thus isomorphic contracts are isomorphic (and thus bisimilar) as singleton systems with the discrete link graph.

```
Theorem c_iso_csys_iso :
    contracts_isomorphic C1 C2 ->
    systems_isomorphic (singleton_place_graph C1) (singleton_place_graph C2).
```

Listing 6.55: The singleton systems of isomorphic contracts are isomorphic.

These lifting theorems show the details of the generalization from contracts, contract trace morphisms, and contract bisimulations to contract systems, system trace morphisms, and system bisimulations.

6.5 Conclusion

Contract system specifications obfuscate the intent of the core functionality of the system with details about how different processes pass messages between each other and how their executions relate to each other. Our core thesis of this chapter was that this can be mitigated by separating the specification into that of system infrastructure and that of the core, intended behavior, the latter being agnostic to a contract's modular structure. In contrast with other efforts to address system complexity, which use tools like model checkers to reason in a composable manner *over* the system, we develop process-algebraic formalisms with the intent of separating the specification of the system's infrastructure from its core functionality.

In particular, we formalized contract systems as bigraphs, which describe the spacial hierarchy of a contract system as well as how the constituent contracts are linked through contract calls. A contract system's link graph is a specification of system infrastructure. Furthermore, contract systems defined with a place and link graph take a computational form which acts coherently, as one single process. Our formalisms are designed so that a specified contract can be implemented as a single contract, or proved to be bisimilar to a contract whose specification isolates that of the desired, core functionality.

Though we only used bigraphs in this chapter as a data type, the process algebraic properties of bigraphs have applications further than what we have mentioned. By embedding the process-algebraic semantics of bigraphs into ConCert, it may be possible to reason about contract systems in more sophisticated ways that leverage the process algebra, building on previous work relating to bigraphs [120] and formally reasoning about composable DeFi protocols [130].

Future work includes formalizing theories to support such work, as well as using the formalisms of this chapter to attempt verification of a practical, deployed system of contracts in terms of a single process bisimilar to the system.

Chapter 7

Conclusion

Financial smart contracts have complicated specifications. Meta properties, by definition, are difficult to accurately capture with a contract specification, and the failure to do so routinely exposes fatal smart contract vulnerabilities. However, formal verification tools based in interactive theorem provers, like ConCert, offer us a chance to introduce mathematical techniques to reason about contracts and their specifications with greater precision and mathematical maturity.

We targeted three classes of meta properties common to financial smart contracts: the intended economic properties (§4), the intended upgradeability properties (§5), and the properties intended of a system of interacting contracts, taken as a whole (§6). For each we introduced formal and theoretical tools into ConCert designed to rigorously specify each class of meta properties, including contract metaspecifications, contract morphisms, and bisimulations of both individual contracts and contract systems. In each of these cases, we showed examples to illustrate how these tools can be used in specification and proof.

The goal of this work has been to add mathematical precision and maturity to the art of contract specification and verification, in the hopes that critical financial infrastructure can be more carefully designed, rigorously specified, formally verified, and safely deployed to the blockchain.

7.1 Limitations and Future Work

There are limitations to our work which we hope to address in future work.

In Chapter 4, we stated and verified economic properties derived from a theoretical formulation of AMMs and DeFi which modeled a blockchain as a state machine [19, 20]. Crucially, these formulations are not formalized. We could more rigorously state and prove economic properties of financial smart contracts if we formalized a theory of AMMs and DeFi in ConCert, for which the bigraph embedding of Chapter 6 lays a foundation.

To do so, we must first formally define DeFi *primitives*, including swaps, swap rate, exchange rate, liquidity provider, and liquidity. From these we can give a formalized derivation of key properties of financial smart contracts, including those that we stated in the structured pools metaspecification of §4.5: demand sensitivity, incentive consistency, positive trading cost, and zero-impact liquidity change. Such an embedding makes *all* of the formal reasoning relevant to the contract specification's economic properties fully and clearly encoded in ConCert. The result improves on the rigor of previously cited work on theories of DeFi and AMMs [19, 20] because the DeFi primitives and subsequently derived properties have explicit, formalized semantics in a process algebra, and can be stated of contracts with verified extraction.

The ultimate goal of embedding a theory of DeFi and AMMs into ConCert is to produce a comprehensive, usable, Coq-based toolkit for precisely stating and verifying economic properties of a contract specification. From there it could be made adaptable and automated with specialized tactics and a broad library of verified economic properties, usable at least in principle in contract specifications by engineers in the wild. While the foundations will be process-algebraic (using bigraphs), carefully formalized DeFi primitives could make this scalable by largely abstracting complex process-algebraic reasoning away. Ideally, future researchers could build off of these foundations to reason, in an ever more sophisticated way, about behaviors and vulnerabilities of financial smart contracts.

Another direction of future work builds off the observation of §5.5.4 that general upgradeability frameworks exist in analogy to *fiber bundles*, a geometric construction which is used in mathematics and phyisics to separate the interacting structure of various components of complicated mathematical objects. These results suggest that programs formalized within a proof assistant exhibit properties commonly articulated by mathematicians—in this case, of geometric objects and topological spaces.

There is good reason to suspect that these results are indicative of something deeper, that techniques from topology have real application to formal verification. Topology, by way of homotopy theory, is known to have deep connections to computation by way of *computational trinitarianism* [68, 98]. In this theoretical context, computation, propositions, and proofs all have a geometric, or topological, interpretation. Geometries also emerge in various type theories under active study, including Homotopy Type Theory (HoTT) [21] and others [39, 50]. Each of these type theories are logics as well as computable foundations of geometry, so propositions and proofs have an associated geometry. From this it is natural to hypothesize that the formal structure and behavior of programs may have geometric properties, and so might the propositions and proofs of a formal specification—and that these geometries may interact.

An explicit study of the geometry of programs, propositions, and proofs may be of interest to computer scientists and mathematicians alike. To computer scientists, in formal methods. We have already shown that programs themselves can have geometric properties which can be leveraged in formal specification and verification. It may also be worthwhile to understand the relationship of a proposition's formal proof to its geometry to support work in *proof repair* [113, 114], by understanding geometrically how alterations to a proposition correspond to alterations to its proof. To mathematicians, an understanding of the geometry of propositions and their proofs gives a formal and geometric way to study the (non-)equivalence of proofs and proof spaces of theorems, adding theoretical richness to the common practice of searching for multiple proofs of a single theorem [44].

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. This is done by treating formalized programs as well-defined mathematical objects, and introducing formal, mathematical techniques to state and prove propositions about programs which are difficult to state correctly in prose. 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 2023.

[2] Curve finance. https://curve.fi/. Accessed July 2023.

[3] Uniswap Protocol. https://uniswap.org/. Accessed July 2023.

[4] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core. https://rholang.io/community-site/whitepaper-v3.pdf. Accessed July 2023.

[5] 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.

[6] Guillermo Angeris, Akshay Agrawal, Alex Evans, Tarun Chitra, and Stephen Boyd. Constant Function Market Makers: Multi-Asset Trades via Convex Optimization. arXiv:2107.12484, July 2021.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] Danil Annenkov, Mikkel Milo, and Bas Spitters. Code Extraction from Coq to ML-like languages. 2021. URL `https://icfp21.sigplan.org/details/mlfamilyworkshop-2021-papers/8/Code-Extraction-from-Coq-to-ML-like-languages`.

[13] 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.

[14] 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.

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

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

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

[18] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch Lafuente. SoK: Lending Pools in Decentralized Finance. In Matthew Bernhard, Andrea Bracciali, Lewis Gudgeon, Thomas Haines, Ariah Klages-Mundt, Shin'ichiro Matsuo, Daniel Perez, Massimiliano Sala, and Sam Werner, editors, *Financial Cryptography and Data Security. FC 2021 International Workshops*, Lecture Notes in Computer Science, pages 553–578, Berlin, Heidelberg, 2021. Springer. ISBN 978-3-662-63958-0. doi: 10.1007/978-3-662-63958-0_40.

[19] 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.

[20] 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.

[21] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT library: A formalization of homotopy type theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 164–172, Paris France, January 2017. ACM. ISBN 978-1-4503-4705-1. doi: 10.1145/3018610.3018615. URL `https://dl.acm.org/doi/10.1145/3018610.3018615`.

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

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

[24] 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.

[25] 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.

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

[27] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Science & Business Media, 2013.

[28] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, October 2016. Association for Computing Machinery. ISBN 978-1-4503-4574-3. doi: 10.1145/2993600.2993611.

[29] Giancarlo Bigi, Andrea Bracciali, Giovanni Meacci, and Emilio Tuosto. Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods. In Chiara Bodei, Gianluigi Ferrari, and Corrado Priami, editors, *Programming Languages with Applications to Biology and Security: Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*, Lecture Notes in

Computer Science, pages 142–161. Springer International Publishing, Cham, 2015. ISBN 978-3-319-25527-9. doi: 10.1007/978-3-319-25527-9_11.

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

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

[32] BscScan.com. Uranium Finance Exploiter.
Address 0x2b528a28451e9853f51616f3b0f6d82af8bea6ae, 2021.

[33] 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.

[34] 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.

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

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

[37] James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. System F in Agda, for Fun and Profit. In Graham Hutton, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 255–297, Cham, 2019. Springer International Publishing. ISBN 978-3-030-33636-3. doi: 10.1007/978-3-030-33636-3_10.

[38] 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.

[39] Felix Cherubini, Thierry Coquand, and Matthias Hutzler. A Foundation for Synthetic Algebraic Geometry.

[40] Michele Ciampi, Muhammad Ishaq, Malik Magdon-Ismail, Rafail Ostrovsky, and Vassilis Zikas. FairMM: A Fast and Frontrunning-Resistant Crypto Market-Maker. In Shlomi Dolev, Jonathan Katz, and Amnon Meisels, editors, *Cyber Security, Cryptology, and Machine Learning*, Lecture Notes in Computer Science, pages 428–446, Cham, 2022. Springer International Publishing. ISBN 978-3-031-07689-3. doi: 10.1007/978-3-031-07689-3_31.

[41] Simon Cousaert, Jiahua Xu, and Toshiko Matsui. SoK: Yield Aggregators in DeFi. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–14, May 2022. doi: 10.1109/ICBC54727.2022.9805523.

[42] 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.

[43] 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.

[44] John W. Dawson. Why do mathematicians re-prove theorems? *Philosophia Mathematica*, 14(3): 269–286, 2006. doi: 10.1093/philmat/nkl009.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] Christoph Dorn and Christopher L. Douglas. Framed combinatorial topology, December 2021. URL `http://arxiv.org/abs/2112.14700`.

[51] Youssef El Faqir, Javier Arroyo, and Samer Hassan. An overview of decentralized autonomous organizations on the blockchain. In *Proceedings of the 16th International Symposium on Open Collaboration*, OpenSym 2020, pages 1–8, New York, NY, USA, August 2020. Association for Computing Machinery. ISBN 978-1-4503-8779-8. doi: 10.1145/3412569.3412579.

[52] etherscan.io. Harvest Finance Hacker.
Address 0xf224ab004461540778a914ea397c589b677e27bb, 2020.

[53] etherscan.io. CREAM Finance Exploit. Transaction 0x0fe2542079644e107cbf13690eb9c2c65963ccb79089ff96bfaf8dced2331c92, 2021.

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

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

[56] 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.

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

[58] Harvest Finance. Harvest Flashloan Economic Attack Post-Mortem. https://medium.com/harvest-finance/harvest-flashloan-economic-attack-post-mortem-3cf900d65217, October 2020.

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

[60] Savannah Fortis. DeFi-type projects received the highest number of attacks in 2022: Report. https://cointelegraph.com/news/defi-type-projects-received-the-highest-number-of-attacks-in-2022-report. Accessed July 2023.

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

[62] 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.

[63] 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.

[64] Neil Ghani, Jules Hedges, Viktor Winschel, and Philipp Zahn. Compositional game theory. arXiv:1603.04641, February 2018.

[65] 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.

[66] 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.

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

[68] Robert Harper. The Holy Trinity (2011). https://existentialtype.wordpress.com/2011/03/27/the-holy-trinity/. Accessed August 2023.

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

[70] Lioba Heimbach, Eric Schertenleib, and Roger Wattenhofer. Risks and Returns of Uniswap V3 Liquidity Providers. In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, pages 89–101, September 2022. doi: 10.1145/3558535.3559772.

[71] 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.

[72] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, November 2005. ISSN 0164-0925. doi: 10.1145/1108970.1108971.

[73] 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.

[74] 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.

[75] Dale Husemoller. *Fibre Bundles*, volume 20 of *Graduate Texts in Mathematics*. Springer, New York, NY, 1994. ISBN 978-1-4757-2263-5 978-1-4757-2261-1. doi: 10.1007/978-1-4757-2261-1.

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

[77] Immunefi. Hack Analysis: Cream Finance Oct 2021. https://medium.com/immunefi/hack-analysis-cream-finance-oct-2021-fc222d913fc5, November 2022.

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

[79] 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.

[80] 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.

[81] Runtime Verification Inc. K-Michelson: A Michelson Semantics. https://github.com/runtime verification/michelson-semantics, August 2022. Accessed July 2023.

[82] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1695–1712. IEEE, 2020.

[83] Roman Kozhan and Ganesh Viswanath-Natraj. Fundamentals of the MakerDAO Governance Token. In *3rd International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[84] 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.

[85] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, October 2016. Association for Computing Machinery. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978309.

[86] Gabor Madl, Luis Bathen, German Flores, and Divyesh Jadav. Formal Verification of Smart Contracts Using Interface Automata. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 556–563, July 2019. doi: 10.1109/Blockchain.2019.00081.

[87] 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.

[88] Alessio Mansutti, Marino Miculan, and Marco Peressotti. Multi-agent Systems Design and Prototyping with Bigraphical Reactive Systems. In Kostas Magoutis and Peter Pietzuch, editors, *Distributed Applications and Interoperable Systems*, volume 8460, pages 201–208. Springer

Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-662-43351-5 978-3-662-43352-2. doi: 10.1007/978-3-662-43352-2_16.

[89] Aaron Marback, Hyunsook Do, Ke He, Samuel Kondamarri, and Dianxiang Xu. A threat model-based approach to security testing. *Software: Practice and Experience*, 43(2):241–258, 2013. ISSN 1097-024X. doi: 10.1002/spe.2111.

[90] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers 22*, pages 523–540. Springer, 2018.

[91] Anastasia Mavridou and Aron Laszka. Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust*, Lecture Notes in Computer Science, pages 270–277, Cham, 2018. Springer International Publishing. ISBN 978-3-319-89722-6. doi: 10.1007/978-3-319-89722-6_11.

[92] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 446–465, Cham, 2019. Springer International Publishing. ISBN 978-3-030-32101-7. doi: 10.1007/978-3-030-32101-7_27.

[93] Robin Milner. The Bigraphical Model. https://www.cl.cam.ac.uk/archive/rm135/uam-theme.html. Accessed July 2023.

[94] Robin Milner. Bigraphs and Their Algebra. *Electronic Notes in Theoretical Computer Science*, 209: 5–19, April 2008. ISSN 1571-0661. doi: 10.1016/j.entcs.2008.04.002.

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

[96] James R. Munkres. *Topology.* Prentice Hall, Incorporated, 2000. ISBN 978-0-13-181629-9.

[97] 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.

[98] ncatlab.org. Computational trinity. https://ncatlab.org/nlab/show/computational+trilogy. Accessed August 2023.

[99] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, January 1997. Association for Computing Machinery. ISBN 978-0-89791-853-4. doi: 10.1145/263699.263712.

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

[101] Zeinab Nehaï, François Bobot, Sara Tucci-Piergiovanni, Carole Delporte-Gallet, and Hugues Fauconnier. A TLA+ Formal Proof of a Cross-Chain Swap. In *23rd International Conference on Distributed Computing and Networking*, ICDCN 2022, pages 148–159, New York, NY, USA, January 2022. Association for Computing Machinery. ISBN 978-1-4503-9560-1. doi: 10.1145/3491003.3491006.

[102] 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.

[103] Jakob Botsch Nielsen and Bas Spitters. Smart Contract Interactions in Coq. In *FM Workshops (1)*, volume 12232 of *Lecture Notes in Computer Science*, pages 380–391. Springer, 2019.

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

[105] Russell O'Connor. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pages 107–120, 2017.

[106] Trail of Bits. Contract Upgrade Anti-Patterns. https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns, September 2018. Accessed July 2023.

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

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

[109] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. A formal verification tool for Ethereum VM bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 912–915, Lake Buena Vista FL USA, October 2018. ACM. ISBN 978-1-4503-5573-5. doi: 10.1145/3236024.3264591.

[110] 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.

[111] Publius. Beanstalk: A Permissionless Fiat Stablecoin Protocol. https://bean.money/beanstalk.pdf. Accessed July 2023.

[112] 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.

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

[114] 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.

[115] John Rushby. Theorem Proving for Verification. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, *Modeling and Verification of Parallel Processes: 4th Summer School, MOVEP 2000 Nantes, France, June 19–23, 2000 Revised Tutorial Lectures*, Lecture Notes in Computer Science, pages 39–57. Springer, Berlin, Heidelberg, 2001. ISBN 978-3-540-45510-3. doi: 10.1007/3-540-45510-8_2.

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

[117] 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.

[118] 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.

[119] Michele Sevegnani and Muffy Calder. Bigraphs with sharing. *Theoretical Computer Science*, 577: 43–73, April 2015. ISSN 0304-3975. doi: 10.1016/j.tcs.2015.02.011.

[120] Michele Sevegnani and Muffy Calder. BigraphER: Rewriting and analysis engine for bigraphs. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, pages 494–501. Springer, 2016.

[121] 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.

[122] 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.

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

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

[125] Derek Sorensen. Structured Pools for Tokenized Carbon Credits. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–6, May 2023. doi: 10.1109/ICBC56567.2023.10174866.

[126] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *J Autom Reasoning*, 64(5):947–999, June 2020. ISSN 1573-0670. doi: 10.1007/s10817-019-09540-0.

[127] Tianyu Sun and Wensheng Yu. A Formal Verification Framework for Security Issues of Blockchain Smart Contracts. *Electronics*, 9(2):255, February 2020. ISSN 2079-9292. doi: 10.3390/electronics9020255.

[128] Keefer Taylor. Murmuration: A Generalizable DAO for Tezos. https://github.com/Hover-Labs/murmuration. Accessed March 2023.

[129] Huang Teng, Wayneyuan Tian, Haocheng Wang, and Zhiyuan Yang. Applications of the Decentralized Finance (DeFi) on the Ethereum. In *2022 IEEE Asia-Pacific Conference on Image Processing, Electronics and Computers (IPEC)*, pages 573–578. IEEE, 2022.

[130] Palina Tolmach, Yi Li, Shang-Wei Lin, and Yang Liu. Formal Analysis of Composable DeFi Protocols. In *Financial Cryptography and Data Security. FC 2021 International Workshops*, Lecture Notes in Computer Science, pages 149–161, Berlin, Heidelberg, 2021. Springer. ISBN 978-3-662-63958-0. doi: 10.1007/978-3-662-63958-0_13.

[131] 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.

[132] P. Torr. Demystifying the threat modeling process. *IEEE Security & Privacy*, 3(5):66–70, September 2005. ISSN 1558-4046. doi: 10.1109/MSP.2005.119.

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

[134] 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.

[135] 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.

[136] 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.

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

[138] 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.

[139] 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.

[140] 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, May 2023. doi: 10.1109/ICBC56567.2023.10174901.

[141] Open Zeppelin. Proxy Upgrade Pattern. https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies, Accessed July 2023.

[142] Open Zeppelin. Writing Upgradeable Contracts. https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable, Accessed July 2023.

[143] Xiyue Zhang, Yi Li, and Meng Sun. Towards a Formally Verified EVM in Production Environment. *Coordination Models and Languages*, 12134:341–349, May 2020. doi: 10.1007/978-3-030-50029-0_21.

[144] 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.

[145] Liyi Zhou, Kaihua Qin, and Arthur Gervais. A2mm: Mitigating frontrunning, transaction reordering and consensus instability in decentralized exchanges. *arXiv preprint arXiv:2106.07371*, 2021.

[146] 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.

[147] Jian Zhu, Kai Hu, Mamoun Filali, Jean-Paul Bodeveix, and Jean-Pierre Talpin. Formal Verification of Solidity contracts in Event-B, May 2020.

[148] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering*, 47(10):2084–2106, October 2021. ISSN 1939-3520. doi: 10.1109/TSE.2019.2942301.

Appendix A

Proofs and Definitions of Chapter 4

A.1 Formal Specification

We include the typeclass specifications for each contract type and the specification predicate.

A.1.1 Typeclass Specifications

Listing A.1: The formalization of the typeclass `Setup_Spec`, which characterizes the storage type `Setup` of a structured pool contract.

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

Listing A.2: The formalization of the typeclass `Msg_Spec` and its associated types, which characterize the storage type `Msg` of a structured pool contract.

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

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

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

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

Listing A.3: The formalization of the typeclass `State_Spec` and its associated types, which characterize the storage type `State` of a structured pool contract.

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

Listing A.4: The formalization of the typeclass `Error_Spec`, which characterizes the storage type `Error` of a structured pool contract.

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

A.1.2 The Formal Specification of a Structured Pool Contract

Listing A.5: The full, formal sepcification of a structured pool contract.

```
(* ============================================================================
 * The Contract Specification `is_structured_pool` :
      We detail a list of propositions of a contract’s behavior which must be
      proven true of a given contract for it to be a correct structured pool contract.
 * ============================================================================ *)

    { Setup Msg State Error : Type }
    { other_entrypoint : Type }
    `{Serializable Setup} `{Serializable Msg} `{Serializable State} `{Serializable Error}.

(** Specification of the Msg type:
  - A Pool entrypoint, whose interface is defined by the pool_data type
  - An Unpool entrypoint, whose interface is defined by the unpool_data type
  - A Trade entrypoint, whose interface is defined by the trade_data type
  - Possibly other types
*)

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

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

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

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

```
(** Specification of the State type:
  The contract state keeps track of:
    - the exchange rates
    - tokens held
    - pool token address
    - number of outstanding pool tokens
*)
Class State_Spec (T : Type) :=
    build_state_spec {
      (* the exchange rates *)
      stor_rates : T -> FMap token exchange_rate ;
      (* token balances *)
      stor_tokens_held : T -> FMap token N ;
      (* pool token data *)
      stor_pool_token : T -> token ;
      (* number of outstanding pool tokens *)
      stor_outstanding_tokens : T -> N ;
  }.

(** Specification of the Setup type:
  To initialize the contract, we need:
    - the initial rates
    - the pool token
*)
Class Setup_Spec (T : Type) :=
  build_setup_spec {
    init_rates : T -> FMap token exchange_rate ;
    init_pool_token : T -> token ;
}.

(* specification of the Error type *)
Class Error_Spec (T : Type) :=
  build_error_type {
    error_to_Error : error -> T ;
}.

(* we assume that our contract types satisfy the type specifications *)
Context `{Msg_Spec Msg}  `{Setup_Spec Setup}  `{State_Spec State} `{Error_Spec Error}.

(* First, we assume that all successful calls require a message *)
Definition none_fails (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx,
    (* the receive function returns an error if the token to be pooled is not in the
       rates map held in the storage (=> is not in the semi-fungible family) *)
    exists err : Error,
    receive contract chain ctx cstate None = Err err.
```

```
(* We also specify that the Msg type is fully characterized by its typeclass *)
Definition msg_destruct (contract : Contract Setup Msg State Error) : Prop :=
    forall (m : Msg),
    (exists p, m = pool p) \/
    (exists u, m = unpool u) \/
    (exists t, m = trade t) \/
    (exists o, Some m = other o).


(** Specification of the POOL entrypoint *)

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

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

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

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

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


```

```
(** Specification of the UNPOOL entrypoint *)

(* We assume an inverse rate function *)
Context { calc_rx_inv : forall (r_x : N) (q : N), N }.

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

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

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


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

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

(* Defines how the UNPOOL entrypoint updates outstanding tokens *)
Definition unpool_outstanding (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate cstate’ chain ctx msg_payload acts,
    (* the call to POOL was successful *)
    receive contract chain ctx cstate (Some (unpool msg_payload)) = Ok(cstate’, acts) ->
    (* rates all stay the same *)
    let rate_in := get_rate msg_payload.(token_unpooled) (stor_rates cstate) in
    let qty := msg_payload.(qty_unpooled) in
    stor_outstanding_tokens cstate’ =
    stor_outstanding_tokens cstate - qty.

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

(* A successful call to TRADE means that the inputs to the trade calculation are
    all positive *)
Definition trade_entrypoint_check_2 (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate’ acts,
    (* a successful call *)
    receive contract chain ctx cstate (Some (trade (msg_payload))) = Ok(cstate’, acts) ->
    (* exchange rates exist *)
    exists x r_x r_y k,
    ((FMap.find msg_payload.(token_in_trade) (stor_tokens_held cstate) = Some x) /\
    (FMap.find msg_payload.(token_in_trade) (stor_rates cstate) = Some r_x) /\
    (FMap.find msg_payload.(token_out_trade) (stor_rates cstate) = Some r_y) /\
    (stor_outstanding_tokens cstate = k) /\
    r_x > 0 /\ r_y > 0 /\ x > 0 /\ k > 0).

(* Specification of the TRADE entrypoint *)
(* We assume the existence of two functions *)
Context
    { calc_delta_y : forall (rate_in : N) (rate_out : N) (qty_trade : N) (k : N) (x : N),
        N }
    { calc_rx’ : forall (rate_in : N) (rate_out : N) (qty_trade : N) (k : N) (x : N), N }.

(* When TRADE is successfully called, the trade is priced using the correct formula
    given by calculate_trade. The updated rate is also priced using the formula from
    calculate_trade. *)
Definition trade_pricing_formula (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload t_x t_y q cstate’ acts,
    (* the TRADE entrypoint was called succesfully *)
    t_x = msg_payload.(token_in_trade) ->
    t_y = msg_payload.(token_out_trade) ->
    t_x <> t_y ->
    q = msg_payload.(qty_trade) ->
    receive contract chain ctx cstate (Some (trade msg_payload)) =
      Ok(cstate’, acts) ->
    (* calculate the diffs delta_x and delta_y *)
    let delta_x :=
        (get_bal t_x (stor_tokens_held cstate’)) -
        (get_bal t_x (stor_tokens_held cstate)) in
    let delta_y :=
        (get_bal t_y (stor_tokens_held cstate)) -
        (get_bal t_y (stor_tokens_held cstate’)) in
    let rate_in := (get_rate t_x (stor_rates cstate)) in
    let rate_out := (get_rate t_y (stor_rates cstate)) in
    let k := (stor_outstanding_tokens cstate) in
    let x := get_bal t_x (stor_tokens_held cstate) in
    (* the diff delta_x and delta_y are correct *)
    delta_x = q /\
    delta_y = calc_delta_y rate_in rate_out q k x.


Definition trade_update_rates (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate’ acts,
    (* the TRADE entrypoint was called succesfully *)
    receive contract chain ctx cstate (Some (trade msg_payload)) =
      Ok(cstate’, acts) ->
    let t_x := msg_payload.(token_in_trade) in
    let t_y := msg_payload.(token_out_trade) in
    t_x <> t_y /\
    (* calculate the diffs delta_x and delta_y *)
    let rate_in := (get_rate t_x (stor_rates cstate)) in
    let rate_out := (get_rate t_y (stor_rates cstate)) in
    let q := msg_payload.(qty_trade) in
    let k := (stor_outstanding_tokens cstate) in
    let x := get_bal t_x (stor_tokens_held cstate) in
    (* the new rate of t_x is correct *)
    let r_x’ := calc_rx’ rate_in rate_out q k x in
    (stor_rates cstate’) =
    (FMap.add (token_in_trade msg_payload)
        (calc_rx’
            (get_rate (token_in_trade msg_payload) (stor_rates cstate))
            (get_rate (token_out_trade msg_payload) (stor_rates cstate))
            (qty_trade msg_payload)
            (stor_outstanding_tokens cstate)
            (get_bal (token_in_trade msg_payload) (stor_tokens_held cstate)))
        (stor_rates cstate)).


Definition trade_update_rates_formula
    (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate’ acts,
    (* the TRADE entrypoint was called succesfully *)
    receive contract chain ctx cstate (Some (trade msg_payload)) =
      Ok(cstate', acts) ->
    let t_x := msg_payload.(token_in_trade) in
    let t_y := msg_payload.(token_out_trade) in
    t_x <> t_y /\
    (* calculate the diffs delta_x and delta_y *)
    let rate_in := (get_rate t_x (stor_rates cstate)) in
    let rate_out := (get_rate t_y (stor_rates cstate)) in
    let q := msg_payload.(qty_trade) in
    let k := (stor_outstanding_tokens cstate) in
    let x := get_bal t_x (stor_tokens_held cstate) in
    (* the new rate of t_x is correct *)
    let r_x' := calc_rx' rate_in rate_out q k x in
    FMap.find t_x (stor_rates cstate') = Some r_x' /\
    (forall t, t <> t_x ->
        FMap.find t (stor_rates cstate') =
        FMap.find t (stor_rates cstate)).


(* When TRADE is successfully called, it emits two TRANSFER actions *)
Definition trade_emits_transfers (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate cstate' chain ctx msg_payload acts,
    (* the call to TRADE was successful *)
    receive contract chain ctx cstate (Some (trade (msg_payload))) = Ok(cstate', acts) ->
    (* the acts list consists of two transfer actions, specified as follows: *)
    exists transfer_to_x transfer_data_x transfer_payload_x
           transfer_to_y transfer_data_y transfer_payload_y,
    (* there is a transfer call for t_x *)
    let transfer_tx := (act_call
        (* call to the correct token address *)
        (msg_payload.(token_in_trade).(token_address))
        (* with amount = 0 *)
        0
        (* and payload transfer_payload_x *)
        (serialize (FA2Spec.Transfer transfer_payload_x))) in
    (* with a transfer in it *)
    In transfer_data_x transfer_payload_x /\
    (* which itself has transfer data *)
    In transfer_to_x transfer_data_x.(FA2Spec.txs) /\
    (* whose quantity is the quantity traded, transferred to the contract *)
    transfer_to_x.(FA2Spec.amount) = msg_payload.(qty_trade) /\
    transfer_to_x.(FA2Spec.to_) = ctx.(ctx_contract_address) /\
    (* there is a transfer call for t_y *)
    let transfer_ty := (act_call
        (* call to the correct token address *)
        (msg_payload.(token_out_trade).(token_address))
        (* with amount = 0 *)
        0
        (* and payload transfer_payload_y *)
        (serialize (FA2Spec.Transfer transfer_payload_y))) in
    (* with a transfer in it *)
    In transfer_data_y transfer_payload_y /\
    (* which itself has transfer data *)
    In transfer_to_y transfer_data_y.(FA2Spec.txs) /\
    (* whose quantity is the quantity traded, transferred to the contract *)
    let t_x := msg_payload.(token_in_trade) in
    let t_y := msg_payload.(token_out_trade) in
    let rate_in := (get_rate t_x (stor_rates cstate)) in
    let rate_out := (get_rate t_y (stor_rates cstate)) in
    let k := (stor_outstanding_tokens cstate) in
    let x := get_bal t_x (stor_tokens_held cstate) in
    let q := msg_payload.(qty_trade) in
    transfer_to_y.(FA2Spec.amount) = calc_delta_y rate_in rate_out q k x  /\
    transfer_to_y.(FA2Spec.to_) = ctx.(ctx_from) /\
    (* acts is only these two transfers *)
    (acts = [ transfer_tx ; transfer_ty ] \/
     acts = [ transfer_ty ; transfer_tx ]).


Definition trade_tokens_held_update (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate’ acts,
    (* the call to TRADE was successful *)
    receive contract chain ctx cstate (Some (trade (msg_payload))) = Ok(cstate’, acts) ->
    (* in the new state *)
    let t_x := msg_payload.(token_in_trade) in
    let t_y := msg_payload.(token_out_trade) in
    let rate_in := (get_rate t_x (stor_rates cstate)) in
    let rate_out := (get_rate t_y (stor_rates cstate)) in
    let k := (stor_outstanding_tokens cstate) in
    let x := get_bal t_x (stor_tokens_held cstate) in
    let delta_x := msg_payload.(qty_trade) in
    let delta_y := calc_delta_y rate_in rate_out delta_x k x in
    let prev_bal_y := get_bal t_y (stor_tokens_held cstate) in
    let prev_bal_x := get_bal t_x (stor_tokens_held cstate) in
    (* balances update appropriately *)
    get_bal t_y (stor_tokens_held cstate’) = (prev_bal_y - delta_y) /\
    get_bal t_x (stor_tokens_held cstate’) = (prev_bal_x + delta_x) /\
    forall t_z,
        t_z <> t_x ->
        t_z <> t_y ->
        get_bal t_z (stor_tokens_held cstate’) =
        get_bal t_z (stor_tokens_held cstate).

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

```

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

Definition trade_amounts_nonnegative (contract : Contract Setup Msg State Error) : Prop :=
    forall cstate chain ctx msg_payload cstate’ acts,
    (* the call to TRADE was successful *)
    receive contract chain ctx cstate (Some (trade (msg_payload))) = Ok(cstate’, acts) ->
    (* delta_x and delta_y are nonnegative *)
    let t_x := msg_payload.(token_in_trade) in
    let t_y := msg_payload.(token_out_trade) in
    let rate_in := (get_rate t_x (stor_rates cstate)) in
    let rate_out := (get_rate t_y (stor_rates cstate)) in
    let k := (stor_outstanding_tokens cstate) in
    let x := get_bal t_x (stor_tokens_held cstate) in
    let delta_x := msg_payload.(qty_trade) in
    let delta_y := calc_delta_y rate_in rate_out delta_x k x in
    0 <= delta_x /\
    0 <= delta_y.

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

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

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

(** Specification of the functions calc_rx' and calc_delta_y
    - This specification distils the features of trading along a convex curve which are
      relevant to the correct functioning of structured pools.
    - While these are supposed to be nontrivial functions that simulate trading along
      a convex curve, `calc_rx'`, `calc_delta_y`, and `calc_rx_inv` can be made trivial
      by setting all rates to 1, setting the first function to the identity function,
      the second to multiplication by r_x = 1, and the third to division by r_x = 1
*)

(* update_rate function returns positive number if the num, denom are positive *)
Definition update_rate_stays_positive :=
    forall r_x r_y delta_x k x,
    let r_x' := calc_rx' r_x r_y delta_x k x in
    r_x > 0 ->
    r_x' > 0.

(* r_x' <= r_x *)
Definition rate_decrease :=
    forall r_x r_y delta_x k x,
    let r_x' := calc_rx' r_x r_y delta_x k x in
    r_x' <= r_x.

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

Definition rates_balance_2 :=
    forall t prev_state,
    let r_x' := calc_rx' (get_rate (token_in_trade t) (stor_rates prev_state))
          (get_rate (token_out_trade t) (stor_rates prev_state)) (qty_trade t) (stor_outstanding_tokens prev_state)
          (get_bal (token_in_trade t) (stor_tokens_held prev_state)) in
    let delta_y := calc_delta_y (get_rate (token_in_trade t) (stor_rates prev_state))
             (get_rate (token_out_trade t) (stor_rates prev_state)) (qty_trade t) (
    stor_outstanding_tokens prev_state)
             (get_bal (token_in_trade t) (stor_tokens_held prev_state)) in
    let r_x := get_rate (token_in_trade t) (stor_rates prev_state) in
    let x := get_bal (token_in_trade t) (stor_tokens_held prev_state) in
    let y := get_bal (token_out_trade t) (stor_tokens_held prev_state) in
    let r_y:= get_rate (token_out_trade t) (stor_rates prev_state) in
    r_x' * (x + qty_trade t) + r_y * (y - delta_y) =
    r_x * x + r_y * y.

(* the trade always does slightly worse than predicted *)
Definition trade_slippage :=
    forall r_x r_y delta_x k x,
    let delta_y := calc_delta_y r_x r_y delta_x k x in
    r_y * delta_y <= r_x * delta_x.
Definition trade_slippage_2 :=
    forall r_x r_y delta_x k x,
    let delta_y := calc_delta_y r_x r_y delta_x k x in
    let r_x' := calc_rx' r_x r_y delta_x k x in
    r_y * delta_y <= r_x' * delta_x.

(* rates have no positive lower bound *)
Definition arbitrage_lt :=
    forall rate_x rate_y ext k x,
    0 < ext ->
    ext < rate_x ->
    exists delta_x,
    calc_rx' rate_x rate_y delta_x k x <= ext.

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

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

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

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

Definition initialized_with_init_rates (contract : Contract Setup Msg State Error) :=
    forall chain ctx setup cstate,
    (* If the contract initializes successfully *)
    init contract chain ctx setup = Ok cstate ->
    (* then the init rates map is the same as given in the setup *)
    (stor_rates cstate) = (init_rates setup).

Definition initialized_with_pool_token (contract : Contract Setup Msg State Error) :=
    forall chain ctx setup cstate,
    (* If the contract initializes successfully *)
    init contract chain ctx setup = Ok cstate ->
    (* then the pool token is the same as given in the setup *)
    (stor_pool_token cstate) = (init_pool_token setup).
```

A.1.3 The Formal Specification Predicate

Listing A.6: The amalgamation of the propositions of the specification of Listing ?? into a single predicate on smart contracts, which is the formal specification of a structured pool contract.

```
Definition is_structured_pool
    (C : Contract Setup Msg State Error) : Prop :=
    none_fails C /\
    msg_destruct C /\
    (* pool entrypoint specification *)
    pool_entrypoint_check C /\
    pool_emits_txns C /\
    pool_increases_tokens_held C /\
    pool_rates_unchanged C /\
    pool_outstanding C /\
    (* unpool entrypoint specification *)
    unpool_entrypoint_check C /\
    unpool_entrypoint_check_2 C /\
    unpool_emits_txns C /\
    unpool_decreases_tokens_held C /\
    unpool_rates_unchanged C /\
    unpool_outstanding C /\
    (* trade entrypoint specification *)
    trade_entrypoint_check C /\
    trade_entrypoint_check_2 C /\
    trade_pricing_formula C /\
    trade_update_rates C /\
    trade_update_rates_formula C /\
    trade_emits_transfers C /\
    trade_tokens_held_update C /\
    trade_outstanding_update C /\
    trade_pricing C /\
    trade_amounts_nonnegative C /\
    (* specification of all other entrypoints *)
    other_rates_unchanged C /\
    other_balances_unchanged C /\
    other_outstanding_unchanged C /\
    (* specification of calc_rx' and calc_delta_y *)
    update_rate_stays_positive /\
    rate_decrease /\
    rates_balance /\
    rates_balance_2 /\
    trade_slippage /\
    trade_slippage_2 /\
    arbitrage_lt /\
    arbitrage_gt /\
    (* initialization specification *)
    initialized_with_positive_rates C /\
    initialized_with_zero_balance C /\
    initialized_with_zero_outstanding C /\
    initialized_with_init_rates C /\
    initialized_with_pool_token C.
```

A.2 Formal Metaspecification

Here we include the formalization and proofs of each property of the metaspecification in Chapter 4.

A.2.1 Demand Sensitivity

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.*

Listing A.7: The formalization and proof of Demand Sensitivity, Property 1.

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

Listing A.8: Definitions and lemmas supporting the formal definition and proof of Demand Sensitivity.

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

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

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

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

A.2.2 Nonpathological Prices

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.*

Listing A.9: The formalization and proof of Nonpathological Prices, Property 2.

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

A.2.3 Swap Rate Consistency

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.*

Listing A.10: The formalization and proof of Swap Rate Consistency, Property 3.

```
Theorem swap_rate_consistency bstate cstate :
    (* Let t_x be a token with nonzero pooled liquidity and rate r_x > 0 *)
    forall t_x r_x x,
    FMap.find t_x (stor_rates cstate) = Some r_x /\ r_x > 0 ->
    FMap.find t_x (stor_tokens_held cstate) = Some x /\ x > 0 ->
    (*then for any delta_x > 0 and any sequence of trades, beginning and ending with t_x*)
    forall delta_x (trade_sequence : list trade_sequence_type) t_fst t_last,
    delta_x > 0 ->
    (* trade_sequence is a list of successive trades *)
    are_successive_trades trade_sequence ->
    (* with a first and last trade, t_fst and t_last respectively, *)
    (hd_error trade_sequence) = Some t_fst ->
    (hd_error (rev trade_sequence)) = Some t_last ->
    (* starting from our current bstate and cstate *)
    seq_chain t_fst = bstate ->
    seq_cstate t_fst = cstate ->
    (* the first trade is from t_x *)
    token_in_trade (seq_trade_data t_fst) = t_x ->
    qty_trade (seq_trade_data t_fst) = delta_x ->
    (* the last trade is to t_x *)
    token_out_trade (seq_trade_data t_last) = t_x ->
    FMap.find t_x (stor_rates cstate) = FMap.find t_x (stor_rates (seq_cstate t_last)) ->
    (* delta_x', the output of the last trade, is never larger than delta_x. *)
    let delta_x' := trade_to_delta_y t_last in
    delta_x' <= delta_x.
Proof.
    intros * H_rate H_held * dx_geq_0 trades_successive fst_txn lst_txn start_bstate
    start_cstate from_tx trade_delta_x to_tx one_tx_txn *.
    unfold delta_x'.
    rewrite <- trade_delta_x.
    apply (geq_list_is_sufficient trade_sequence t_x t_fst t_last cstate r_x); auto.
    now apply swap_rate_lemma.
Qed.
```

Listing A.11: Definitions and lemmas relevant to the formal definition of Swap Rate Consistency

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

(* a function to calculate the the trade output of the final trade, delta_x' *)
Definition trade_to_delta_y (t : trade_sequence_type) :=
    let cstate := seq_cstate t in
    let token_in := token_in_trade (seq_trade_data t) in
    let token_out := token_out_trade (seq_trade_data t) in
    let rate_in := get_rate token_in (stor_rates cstate) in
    let rate_out := get_rate token_out (stor_rates cstate) in
    let delta_x := qty_trade (seq_trade_data t) in
    let k := stor_outstanding_tokens cstate in
    let x := get_bal token_in (stor_tokens_held cstate) in
    (* the calculation *)
    calc_delta_y rate_in rate_out delta_x k x.

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

Fixpoint geq_list (l : list N) : Prop :=
    match l with
    | [] => True
    | h :: tl =>
        match tl with
        | [] => True
        | h’ :: tl’ => (h >= h’) /\ geq_list tl
        end
    end.
```

Listing A.12: The two lemmas that constitue the essence of the formal proof of Swap Rate Consistency.

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

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

Listing A.13: Definitions relevant to the formal definition and proof of the two auxiliary lemmas of the proof of Swap Rate Consistency.

```
Definition trade_to_ry_delta_y (t : trade_sequence_type) :=
    let delta_y := trade_to_delta_y t in
    let rate_y := get_rate (token_out_trade (seq_trade_data t)) (stor_rates (seq_cstate t)
    ) in
    rate_y * delta_y.
```

A.2.4 Zero-Impact Liquidity Change

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

Listing A.14: The formalization and proof of Zero-Impact Liquidity Change, Property 4.

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

A.2.5 Arbitrage Sensitivity

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.*

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

Listing A.15: The formalization and proof of Arbitrage Sensitivity, Property 5.

A.2.6 Pooled Consistency

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

$$\sum_{t_x} r_x x = k \tag{A.1}$$

Listing A.16: The formalization and proof of Pooled Consistency, Property 6.

```
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 A.17: Definitions and lemmas supporting the formal definition and proof of Pooled Consistency.

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

(* take the sum of a list *)
Definition suml l := fold_right N.add 0 l.
```

Appendix B

Proofs and Definitions of Chapter 5

B.1 Contract Morphisms

Listing B.1: The formal definition of a contract morphism.

```
Section MorphismDefinition.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}.

(** The definition *)
Record ContractMorphism
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :=
    build_contract_morphism {
        (* the components of a morphism f *)
        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) ;
}.

End MorphismDefinition.
```

Listing B.2: The definition of `result_functor`.

```
Definition result_functor {T T’ E E’ : Type} : (T -> T’) -> (E -> E’) -> result T E -> result T’ E’ :=
    fun (f_t : T -> T’) (f_e : E -> E’) (res : result T E) =>
    match res with | Ok t => Ok (f_t t) | Err e => Err (f_e e) end.
```

Listing B.3: The identity contract morphism.

```
Section IdentityMorphism.
Context `{Serializable Msg} `{Serializable Setup} `{Serializable State} `{Serializable Error}.

Lemma init_coherence_id (C : Contract Setup Msg State Error) :
    forall c ctx s,
    result_functor id id (init C c ctx s) =
    init C c ctx s.
Proof.
    intros.
    unfold result_functor.
    now destruct (init C c ctx s).
Qed.

Lemma recv_coherence_id (C : Contract Setup Msg State Error) :
    forall c ctx st op_msg,
    result_functor
        (fun '(st, l) => (id st, l)) id
        (receive C c ctx st op_msg) =
    receive C c ctx (id st) (option_map id op_msg).
Proof.
    intros.
    unfold result_functor, option_map. cbn.
    destruct op_msg.
    -   now destruct (receive C c ctx st (Some m)); try destruct t.
    -   now destruct (receive C c ctx st None); try destruct t.
Qed.

(** The identity morphism *)
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 ;
    |}.
```

```

End IdentityMorphism.
```

Listing B.4: The formal definition of a contract injection and surjection.

```
Section Injections.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
        {C1 : Contract Setup1 Msg1 State1 Error1}
        {C2 : Contract Setup2 Msg2 State2 Error2}.

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

Lemma is_inj_compose {A B C : Type} :
    forall (f1 : A -> B) (f2 : B -> C),
    is_inj f1 ->
    is_inj f2 ->
    is_inj (compose f2 f1).
Proof.
    intros * f1_inj f2_inj.
    unfold is_inj in *.
    intros * H_compose.
    simpl in H_compose.
    apply f2_inj in H_compose.
    now apply f1_inj in H_compose.
Qed.

(* A morphism is a *weak embedding* if it embeds the message and storage types *)
Definition is_weak_inj_cm (f : ContractMorphism C1 C2) : Prop :=
    is_inj (msg_morph   C1 C2 f) /\
    is_inj (state_morph C1 C2 f).

Definition contract_weakly_embeds : Prop :=
    exists (f : ContractMorphism C1 C2), is_weak_inj_cm f.

(* A morphism is an embedding if it embeds on all contract types *)
Definition is_inj_cm (f : ContractMorphism C1 C2) : Prop :=
    is_inj (setup_morph C1 C2 f) /\
    is_inj (msg_morph   C1 C2 f) /\
    is_inj (state_morph C1 C2 f) /\
    is_inj (error_morph C1 C2 f).

Definition contract_embeds : Prop :=
    exists (f : ContractMorphism C1 C2), is_inj_cm f.

End Injections.
```

```


(** Surjective contract morphisms *)
Section Surjections.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable
    Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable
    Error2}
        {C1 : Contract Setup1 Msg1 State1 Error1}
        {C2 : Contract Setup2 Msg2 State2 Error2}.

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

(* A morphism is a *weak quotient* if it embeds on all contract types *)
Definition is_weak_surj_cm (f : ContractMorphism C1 C2) : Prop :=
    is_surj (msg_morph   C1 C2 f) /\
    is_surj (state_morph C1 C2 f).

Definition contract_weakly_surjects : Prop :=
    exists (f : ContractMorphism C1 C2), is_weak_surj_cm f.

(* A morphism is a surjection if it surjects on all contract types *)
Definition is_surj_cm (f : ContractMorphism C1 C2) : Prop :=
    is_surj (setup_morph C1 C2 f) /\
    is_surj (msg_morph   C1 C2 f) /\
    is_surj (state_morph C1 C2 f) /\
    is_surj (error_morph C1 C2 f).

Definition contract_surjects : Prop :=
    exists (f : ContractMorphism C1 C2), is_surj_cm f.

End Surjections.
```

Listing B.5: Equality of contract morphisms.

```
Section EqualityOfMorphisms.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable
    Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable
    Error2}
        {C1 : Contract Setup1 Msg1 State1 Error1}
        {C2 : Contract Setup2 Msg2 State2 Error2}.

Lemma eq_cm :
    forall (f g : ContractMorphism C1 C2),
    (* if the components are equal ... *)
    (setup_morph C1 C2 f) = (setup_morph C1 C2 g) ->
    (msg_morph C1 C2 f) = (msg_morph C1 C2 g) ->
    (state_morph C1 C2 f) = (state_morph C1 C2 g) ->
    (error_morph C1 C2 f) = (error_morph C1 C2 g) ->
    (* ... then the morphisms are equal *)
    f = g.
Proof.
    intros f g.
    destruct f, g.
    cbn.
    intros msg_eq set_eq st_eq err_eq.
    subst.
    f_equal;
    apply proof_irrelevance.
Qed.

Lemma eq_cm_rev :
    forall (f g : ContractMorphism C1 C2),
    (* if the morphisms are equal ... *)
    f = g ->
    (* ... then the components are equal *)
    (setup_morph C1 C2 f) = (setup_morph C1 C2 g) /\
    (msg_morph C1 C2 f) = (msg_morph C1 C2 g) /\
    (state_morph C1 C2 f) = (state_morph C1 C2 g) /\
    (error_morph C1 C2 f) = (error_morph C1 C2 g).
Proof.
    intros f g fg_eq.
    now inversion fg_eq.
Qed.

Lemma eq_cm_iff :
    forall (f g : ContractMorphism C1 C2),
    (* the components are equal ... *)
    (setup_morph C1 C2 f) = (setup_morph C1 C2 g) /\
    (msg_morph   C1 C2 f) = (msg_morph   C1 C2 g) /\
    (state_morph C1 C2 f) = (state_morph C1 C2 g) /\
    (error_morph C1 C2 f) = (error_morph C1 C2 g) <->
    (* ... iff the morphisms are equal *)
    f = g.
Proof.
    intros.
    split.
    -   intro H_components.
        destruct H_components as [H_set [H_msg [H_st H_err]]].
        now apply eq_cm.
    -   now apply eq_cm_rev.
Qed.

End EqualityOfMorphisms.
```

B.1.1 Composition of Morphisms

Listing B.6: Composition of contract morphisms.

```
Section MorphismComposition.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
        `{Serializable Setup3} `{Serializable Msg3} `{Serializable State3} `{Serializable Error3}
        { C1 : Contract Setup1 Msg1 State1 Error1 }
        { C2 : Contract Setup2 Msg2 State2 Error2 }
        { C3 : Contract Setup3 Msg3 State3 Error3 }.

Lemma compose_init_coh (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) :
    let setup_morph’ := (compose (setup_morph C2 C3 g) (setup_morph C1 C2 f)) in
    let state_morph’ := (compose (state_morph C2 C3 g) (state_morph C1 C2 f)) in
    let error_morph’ := (compose (error_morph C2 C3 g) (error_morph C1 C2 f)) in
    forall c ctx s,
        result_functor state_morph’ error_morph’
            (init C1 c ctx s) =
        init C3 c ctx (setup_morph’ s).
Proof.
    intros.
    unfold setup_morph’.
    cbn.
    rewrite <- (init_coherence C2 C3 g).
    rewrite <- (init_coherence C1 C2 f).
    unfold result_functor.
    now destruct (init C1 c ctx s).
Qed.

Lemma compose_recv_coh (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) :
    let msg_morph’   := (compose (msg_morph   C2 C3 g) (msg_morph   C1 C2 f)) in
    let state_morph’ := (compose (state_morph C2 C3 g) (state_morph C1 C2 f)) in
    let error_morph’ := (compose (error_morph C2 C3 g) (error_morph C1 C2 f)) in
    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 C3 c ctx (state_morph’ st) (option_map msg_morph’ op_msg).
Proof.
    intros.
    pose proof (recv_coherence C2 C3 g).
    pose proof (recv_coherence C1 C2 f).
    unfold state_morph’, msg_morph’.
    cbn.
    replace (option_map (compose (msg_morph C2 C3 g) (msg_morph C1 C2 f)) op_msg)
    with (option_map (msg_morph C2 C3 g) (option_map (msg_morph C1 C2 f) op_msg)).
    2:{ now destruct op_msg. }
    rewrite <- H11.
    rewrite <- H12.
    unfold result_functor.
    now destruct (receive C1 c ctx st op_msg).
Qed.

(** Composition *)
Definition compose_cm (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) :
    ContractMorphism C1 C3 := {|
    (* the components *)
    setup_morph := compose (setup_morph C2 C3 g) (setup_morph C1 C2 f) ;
    msg_morph   := compose (msg_morph   C2 C3 g) (msg_morph   C1 C2 f) ;
    state_morph := compose (state_morph C2 C3 g) (state_morph C1 C2 f) ;
    error_morph := compose (error_morph C2 C3 g) (error_morph C1 C2 f) ;
    (* the coherence results *)
    init_coherence := compose_init_coh g f ;
    recv_coherence := compose_recv_coh g f ;
    |}.

End MorphismComposition.
```

Listing B.7: Some results on contract morphisms.

```
Section MorphismCompositionResults.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
        `{Serializable Setup3} `{Serializable Msg3} `{Serializable State3} `{Serializable Error3}
        `{Serializable Setup4} `{Serializable Msg4} `{Serializable State4} `{Serializable Error4}
        { C1 : Contract Setup1 Msg1 State1 Error1 }
        { C2 : Contract Setup2 Msg2 State2 Error2 }
        { C3 : Contract Setup3 Msg3 State3 Error3 }
        { C4 : Contract Setup4 Msg4 State4 Error4 }.

(** Composition with the Identity morphism is trivial *)
Lemma compose_id_cm_left (f : ContractMorphism C1 C2) :
    compose_cm (id_cm C2) f = f.
Proof.
    now apply eq_cm.
Qed.

Lemma compose_id_cm_right (f : ContractMorphism C1 C2) :
    compose_cm f (id_cm C1) = f.
Proof.
    now apply eq_cm.
```

```
Qed.

(** Composition is associative *)
Lemma compose_cm_assoc
    (f : ContractMorphism C1 C2)
    (g : ContractMorphism C2 C3)
    (h : ContractMorphism C3 C4) :
    compose_cm h (compose_cm g f) =
    compose_cm (compose_cm h g) f.
Proof.
    now apply eq_cm.
Qed.

End MorphismCompositionResults.
```

Listing B.8: Definition of a contract isomorphism.

```
Section IsomorphismDefinition.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
        {C1 : Contract Setup1 Msg1 State1 Error1}
        {C2 : Contract Setup2 Msg2 State2 Error2}.

Definition is_iso {A B : Type} (f : A -> B) (g : B -> A) : Prop :=
    compose g f = @id A /\ compose f g = @id B.

Lemma is_iso_reflexive {A B : Type} (f : A -> B) (g : B -> A) :
    is_iso f g -> is_iso g f.
Proof.
    unfold is_iso.
    intro H_is_iso.
    now destruct H_is_iso.
Qed.

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.

Lemma iso_cm_components :
    forall (f : ContractMorphism C1 C2) (g : ContractMorphism C2 C1),
    is_iso (msg_morph   C1 C2 f) (msg_morph   C2 C1 g) /\
    is_iso (setup_morph C1 C2 f) (setup_morph C2 C1 g) /\
    is_iso (state_morph C1 C2 f) (state_morph C2 C1 g) /\
    is_iso (error_morph C1 C2 f) (error_morph C2 C1 g)
    <->
    is_iso_cm f g.
Proof.
```

```
    intros f g.
    unfold is_iso.
    unfold iff.
    split.
    (* -> *)
    -   intro H_iso.
        unfold is_iso_cm.
        split; now apply eq_cm.
    (* <- *)
    -   unfold is_iso_cm, compose_cm, id_cm.
        now intro H_iso.
Qed.

End IsomorphismDefinition.
```

Listing B.9: Some results on contract isomorphisms.

```
Section IsomorphismsResults.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}.

(** An equality predicate on contracts *)
Definition contracts_isomorphic
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) : Prop :=
    exists (f : ContractMorphism C1 C2) (g : ContractMorphism C2 C1),
    is_iso_cm f g.

Context `{Serializable Setup3} `{Serializable Msg3} `{Serializable State3} `{Serializable Error3}
        `{Serializable Setup4} `{Serializable Msg4} `{Serializable State4} `{Serializable Error4}
        { C1 : Contract Setup1 Msg1 State1 Error1 }
        { C2 : Contract Setup2 Msg2 State2 Error2 }
        { C3 : Contract Setup3 Msg3 State3 Error3 }
        { C4 : Contract Setup4 Msg4 State4 Error4 }.

Lemma iso_cm_reflexive (f : ContractMorphism C1 C2) (g : ContractMorphism C2 C1) :
    is_iso_cm f g ->
    is_iso_cm g f.
Proof.
    intro H_is_iso.
    apply iso_cm_components in H_is_iso.
    apply iso_cm_components.
    destruct H_is_iso as [H_ind_iso H_is_iso].
    do 2 (apply is_iso_reflexive in H_ind_iso;
    split;
    try exact H_ind_iso;
    clear H_ind_iso;
    destruct H_is_iso as [H_ind_iso H_is_iso]).
    split; now apply is_iso_reflexive.
Qed.

Lemma composition_iso_cm
    (f1 : ContractMorphism C1 C2)
    (g1 : ContractMorphism C2 C3)
    (f2 : ContractMorphism C2 C1)
    (g2 : ContractMorphism C3 C2) :
    is_iso_cm f1 f2 ->
    is_iso_cm g1 g2 ->
    is_iso_cm (compose_cm g1 f1) (compose_cm f2 g2).
Proof.
    unfold is_iso_cm.
    intros iso_f iso_g.
    destruct iso_f as [iso_f1 iso_f2].
    destruct iso_g as [iso_g1 iso_g2].
    split; rewrite compose_cm_assoc.
    -   rewrite <- (compose_cm_assoc g1 g2 f2).
        rewrite iso_g1.
        now rewrite (compose_id_cm_right f2).
    -   rewrite <- (compose_cm_assoc f2 f1 g1).
        rewrite iso_f2.
        now rewrite (compose_id_cm_right g1).
Qed.

End IsomorphismsResults.
```

Listing B.10: An isomorphism of contracts is both injective and surjective.

```
Theorem inj_surj_iso_cm (f : ContractMorphism C1 C2) :
    (exists (g : ContractMorphism C2 C1), is_iso_cm f g) ->
    is_inj_cm f /\ is_surj_cm f.
```

B.2 Morphism Induction

B.2.1 Contract Trace and Reachability

Listing B.11: The definition of contract trace and contract reachability.

```
Section ContractTrace.
Context { Setup Msg State Error : Type }
        `{Serializable Msg} `{Serializable Setup} `{Serializable State} `{Serializable Error}.
```

```

(* Notions of contract state stepping forward *)
Record ContractStep (C : Contract Setup Msg State Error)
    (prev_cstate : State) (next_cstate : State) :=
    build_contract_step {
    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) ;
}.

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

Definition is_genesis_state (C : Contract Setup Msg State Error) (init_cstate : State) :
    Prop :=
    exists init_chain init_ctx init_setup,
    init C init_chain init_ctx init_setup = Ok init_cstate.

Definition cstate_reachable (C : Contract Setup Msg State Error) (cstate : State) : Prop
    :=
    exists init_cstate,
    (* init_cstate is a valid initial cstate *)
    is_genesis_state C init_cstate /\
    (* with a trace to cstate *)
    inhabited (ContractTrace C init_cstate cstate).

Lemma inhab_trace_trans (C : Contract Setup Msg State Error) :
forall from mid to,
    (ContractStep C mid to) ->
    inhabited (ContractTrace C from mid) ->
    inhabited (ContractTrace C from to).
Proof.
    intros from mid to step.
    apply inhabited_covariant.
    intro mid_t.
    apply (snoc mid_t step).
Qed.

End ContractTrace.
```

B.2.2 Right Morphism Induction

Listing B.12: The theorem of left morphism induction.

```
(* f : C1 -> C2, inducting on C1 *)
Theorem left_cm_induction :
    (* forall simple morphism and reachable bstate, *)
    forall (f : ContractMorphism C1 C2) bstate caddr (trace : ChainTrace empty_state
    bstate),
    (* where C is at caddr with state cstate, *)
    env_contracts bstate caddr = Some (C1 : WeakContract) ->
    exists (cstate1 : State1),
    contract_state bstate caddr = Some cstate1 /\
    (* every reachable cstate1 of C1 corresponds to a contract-reachable cstate2 of C2 ...
     *)
    exists (cstate2 : State2),
    (* 1. init_cstate2 is a valid initial cstate of C’  *)
    cstate_reachable C2 cstate2 /\
    (* 2. cstate and cstate’ are related by state_morph. *)
    cstate2 = state_morph C1 C2 f cstate1.
```

B.2.3 Right Morphism Induction

Listing B.13: The theorem of right morphism induction.

```
(* f : C1 -> C2, inducting on C2 *)
Theorem right_cm_induction:
    forall (from to : State1) (f : ContractMorphism C1 C2),
    ContractTrace C1 from to ->
    ContractTrace C2 (state_morph C1 C2 f from) (state_morph C1 C2 f to).
```

B.3 Reasoning with Morphisms: Specification and Proof

B.3.1 Specifying a Contract Upgrade with Morphisms

Listing B.14: The Uranium Finance example.

```
(** Example 5.3.1:
    This example recalls the Uranium Finance hack of 2021 due to an incorrect upgrade:
    A constant ‘k‘ was changed from 1_000 to 10_000 in all but one of its instances
    in the contract.

    This example illustrates how a contract upgrade can be *specified* using contract
    morphisms, and uses that example.
    We have formulated this example to be as general as possible.
*)

```

```
Section UpgradeSpec.
Context { Base : ChainBase }.

(** Assume we have a calculate_trade function which is used to calculate trades in
    a smart contract. It takes some input in N and returns the output of the trade, also in N. *)
Context  { calculate_trade : N -> N }.

(** Assume that we have some storage type which keeps track of balances via
    a function `get_bal` *)
Context { storage : Type } `{ storage_ser : Serializable storage }
        { get_bal : storage -> N }
        (* Also assume some other relevant types of the contract. *)
        { setup : Type } `{ setup_ser : Serializable setup }
        { other_entrypoint : Type }.

(** We assume that entrypoint type includes a trade function, among other entrypoints. *)
Context { trade_data : Type } { trade_qty : trade_data -> N }.

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

Context { entrypoint : Type } `{ e_ser : Serializable entrypoint } `{ e_msg : Msg_Spec entrypoint }.

(** And we assume anything in the entrypoint type is of the form `trade n` or (roughly) `other o`. *)
Definition msg_destruct : Prop :=
    forall e,
    (exists n, e = trade n) \/
    (exists o, Some e = other o).
Context { e_msg_destruct : msg_destruct }.

(** Thus, the entrypoint type has this form:

    Inductive entrypoint :=
        | trade (qty : N)
        | ... .
*)

(* final definitions of contract types *)
Definition error := N.
Definition result : Type := ResultMonad.result (storage * list ActionBody) error.

(*** Now suppose that we have a contract with those types ... *)
Context { C1 : Contract setup entrypoint storage error }.
```

```

(** such that get_bal changes according to calculate_trade, meaning that: *)
Definition spec_trade : Prop :=
    forall cstate chain ctx trade_data cstate’ acts,
    (** for any successful call to the trade entrypoint of C1, *)
    receive C1 chain ctx cstate (Some (trade trade_data)) = Ok(cstate’, acts) ->
    (** the balance in storage updates as follows. *)
    get_bal cstate’ =
    get_bal cstate + calculate_trade (trade_qty trade_data).

Context { spec_trade : Prop }.

(** Now suppose that we have another calculate_trade function, this time which calculates
    trades at one more digit of precision. *)
Definition round_down (n : N) := n / 10.

Context { calculate_trade_precise : N -> N }
        (** (i.e. calculate_trade_precise rounds down to calculate_trade) *)
        { calc_trade_coherence : forall n,
            round_down (calculate_trade_precise n) =
            calculate_trade (round_down n) }.

(** Suppose also that we have a round-down function on the storage type. *)

(** And that we have another contract, C2, ... *)
Context { C2 : Contract setup entrypoint storage error }.

(** but now trades are calculated in line with calculate_trade_precise. *)
Definition spec_trade_precise : Prop :=
    forall cstate chain ctx trade_data cstate’ acts,
    (** ... meaning that for a successful call to the trade entrypoint of C2, *)
    receive C2 chain ctx cstate (Some (trade trade_data)) = Ok(cstate’, acts) ->
    (** the balance held in storage goes up by calculate_trade_precise. *)
    get_bal cstate’ =
    get_bal cstate + calculate_trade_precise (trade_qty trade_data).

Context { spec_trade_precise : Prop }.


(** Now, to specify the *upgrade* from C1 to C2, we specify that there exist some morphism
    f : C1 -> C2 which satisfies the following conditions: *)
Context { st_morph : storage -> storage }
        { state_rounds_down : forall st, get_bal (st_morph st) = round_down (get_bal st)
    }.

(** 1. f rounds trades down when it maps inputs *)
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 = round_down (trade_qty t').

(** 2. aside from trade, f doesn't touch the other entrypoints *)
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. f rounds down on the storage, but doesn't touch anything else. *)
Definition f_state_morph (f : ContractMorphism C2 C1) : Prop :=
    (state_morph C2 C1 f) = st_morph.

(** 4. f is the identity on error values *)
Definition f_recv_output_err (f : ContractMorphism C2 C1) : Prop :=
    (error_morph C2 C1 f) = id.

(** 5. f is the identity on the setup *)
Definition f_init_id (f : ContractMorphism C2 C1) : Prop :=
    (setup_morph C2 C1 f) = id.

(** Now we have a specification of the correct upgrade in terms of the existence of
    a contract morphism. *)
Definition upgrade_spec (f : ContractMorphism C2 C1) : Prop :=
    f_recv_input_rounds_down f /\
    f_recv_input_other_equal f /\
    f_state_morph f /\
    f_recv_output_err f /\
    f_init_id f.


(** The Upgrade Metaspecification.
    To justify that upgrade_spec actually specifies a correct upgrade, we prove
    the following result(s). *)

(*** Suppose there exists some f : C2 -> C1 satisfying upgrade_spec. *)
Context { f : ContractMorphism C2 C1 }
        { is_upgrade_morph : upgrade_spec f }.

(* All states of C2 relate to equivalent states of C1 by rounding down *)
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 = round_down (get_bal cstate').
Proof.
    intros c_at_caddr.
    pose proof (left_cm_induction f bstate caddr trace c_at_caddr)
    as H_cm_ind.
    destruct H_cm_ind as [cstate' [contr_cstate' [cstate [reach H_cm_ind]]]].
    exists cstate', cstate.
    repeat split; auto.
    cbn in H_cm_ind.
    rewrite H_cm_ind.
    destruct is_upgrade_morph as [_ [_ [f_state [_ _]]]].
    now rewrite f_state.
Qed.

End UpgradeSpec.
```

B.3.2 Adding Features and Backwards Compatibility

Listing B.15: An example of backwards compatibility.

```
Section BackwardsCompatible.
Context { Base : ChainBase }.
Set Primitive Projections.
Set Nonrecursive Elimination Schemes.

(** The initial contract C *)
(* contract types definition *)
Inductive entrypoint1 := | incr (u : unit).
Definition storage := N.
Definition setup := N.
Definition error := N.
Definition result : Type := ResultMonad.result (storage * list ActionBody) error.

Section Serialization.
    Global Instance entrypoint1_serializable : Serializable entrypoint1 :=
        Derive Serializable entrypoint1_rect<incr>.
End Serialization.

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

(* receive function definition *)
Definition receive (_ : Chain) (_ : ContractCallContext) (n : storage)
    (msg : option entrypoint1) : result :=
    match msg with
    | Some (incr _) => Ok (n + 1, [])
    | None => Err 0
    end.

(* construct the contract *)
Definition C1 : Contract setup entrypoint1 storage error :=
    build_contract init receive.


(** The updated contract C’ *)
(* contract types definition *)
Inductive entrypoint2 := | incr’ (u : unit) | decr (u : unit).

Section Serialization.
    Global Instance entrypoint2_serializable : Serializable entrypoint2 :=
        Derive Serializable entrypoint2_rect<incr’,decr>.
End Serialization.

(* receive function definition *)
Definition receive’ (_ : Chain) (_ : ContractCallContext) (n : storage)
    (msg : option entrypoint2) : result :=
    match msg with
    | Some (incr’ _) => Ok (n + 1, [])
    | Some (decr _) => Ok (n - 1, [])
    | None => Err 0
    end.

(* construct the contract *)
Definition C2 : Contract setup entrypoint2 storage error :=
    build_contract init receive’.


(** The contract morphism confirming backwards compatibility *)
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.

(* the coherence results *)
Lemma init_coherence : forall c ctx s,
    result_functor state_morph error_morph (init c ctx s) =
    init c ctx (setup_morph s).
Proof. auto. Qed.

Lemma recv_coherence : forall c ctx st op_msg,
    result_functor (fun ’(st, l) => (state_morph st, l)) error_morph (receive c ctx st
    op_msg) =
    receive’ c ctx (state_morph st) (option_map msg_morph op_msg).
Proof.
    intros.
    unfold result_functor, msg_morph, state_morph, error_morph.
    induction op_msg; auto.
    now destruct a.
Qed.

(* construct the morphism *)
Definition f : ContractMorphism C1 C2 :=
    build_contract_morphism C1 C2 setup_morph msg_morph state_morph error_morph
        init_coherence recv_coherence.


(* This theorem shows a strong notion of backwards compatibility because there is
    an embedding of the old contract into the new *)
Lemma embedding : is_inj_cm f.
Proof.
    unfold is_inj_cm; unfold is_inj.
    repeat split; intros.
    -   cbn in H.
        now unfold setup_morph in H.
    -   now destruct a, a’, u, u0.
    -   cbn in H.
        now unfold state_morph in H.
    -   cbn in H.
        now unfold error_morph in H.
Qed.

(** Theorem:
    All reachable states have a corresponding reachable state, related by the
    *embedding* f. *)
Theorem injection_invariant bstate caddr (trace : ChainTrace empty_state bstate):
    (* Forall reachable states with contract C1 at caddr, *)
    env_contracts bstate caddr = Some (C1 : WeakContract) ->
    (* forall reachable states of C1 cstate, 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.
Proof.
    intros c_at_caddr.
    pose proof (left_cm_induction f bstate caddr trace c_at_caddr)
    as H_cm_ind.
    destruct H_cm_ind as [cstate [cstate_c [cstate’ [reach H_cm_ind]]]].
    now exists cstate’, cstate.
```

```
Qed.

End BackwardsCompatible.
```

B.3.3 Transporting Hoare-Like Properties Over a Morphism

Listing B.16: Transporting a Hoare-like property of partial correctness over a contract morphism.

```
Section TransportHoare.
Context { Base : ChainBase }.

Context  {setup storage error : Type}
        `{setup_ser : Serializable setup}  `{stor_ser : Serializable storage}
        `{err_ser : Serializable error} `{storage_state : @State_Spec Base storage}
        `{err_err : Error_Spec error}.

Set Primitive Projections.
Set Nonrecursive Elimination Schemes.

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

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

Context { other_entrypoint : Type }.

Definition e_pool (p : pool_data) : entrypoint := Pool p.
Definition e_unpool (u : unpool_data) : entrypoint := Unpool u.
Definition e_trade (t : trade_data) : entrypoint := Null t.
Definition e_other (o : other_entrypoint) : option entrypoint := None.
Global Instance entrypoint_msg_spec : Msg_Spec entrypoint := {
    pool := e_pool ;
    unpool := e_unpool ;
    trade := e_trade ;
    other := e_other ;
}.

Definition e’_pool (p : pool_data) : entrypoint’ := Pool’ p.
Definition e’_unpool (u : unpool_data) : entrypoint’ := Unpool’ u.
Definition e’_trade (t : trade_data) : entrypoint’ := Trade’ t.
Definition e’_other (o : other_entrypoint) : option entrypoint’ := None.
Global Instance entrypoint’_msg_spec : @Msg_Spec Base other_entrypoint entrypoint’ := {
    pool := e’_pool ;
    unpool := e’_unpool ;
    trade := e’_trade ;
    other := e’_other ;
}.

Section Serialization.
    Global Instance token_serializable : Serializable token :=
        Derive Serializable token_rect<Build_token>.
    Global Instance pool_data_serializable : Serializable pool_data :=
        Derive Serializable pool_data_rect<Build_pool_data>.
    Global Instance unpool_data_serializable : Serializable unpool_data :=
        Derive Serializable unpool_data_rect<Build_unpool_data>.
    Global Instance trade_data_serializable : Serializable trade_data :=
        Derive Serializable trade_data_rect<Build_trade_data>.
    Global Instance entrypoint_serializable : Serializable entrypoint :=
        Derive Serializable entrypoint_rect<Pool,Unpool,Null>.
    Global Instance entrypoint’_serializable : Serializable entrypoint’ :=
        Derive Serializable entrypoint’_rect<Pool’,Unpool’,Trade’>.
End Serialization.

Context `{ set_setup : @Setup_Spec Base setup }
        `{ stor_state : @State_Spec Base storage }.

Context
    {C1 : Contract setup entrypoint  storage error}
    {C2 : Contract setup entrypoint’ storage error}.

Context { calc_rx_inv : forall (r_x : N) (q : N), N }
        { calc_delta_y : forall (rate_in : N) (rate_out : N) (qty_trade : N) (k : N) (x : N), N }
        { calc_rx’ : forall (rate_in : N) (rate_out : N) (qty_trade : N) (k : N) (x : N), N }.

Axiom is_sp : @is_structured_pool _ _ _ _ _ _ _ _ _ _ _ _ _ calc_rx_inv calc_delta_y calc_rx’ C2.

Definition embed_entrypoint (e : entrypoint) : entrypoint’ :=
match e with
| Pool p => Pool’ p
| Unpool p => Unpool’ p
(* redirect the null entrypoint *)
| Null t => Trade’ t
end.

(* some assumptions about C and C’ *)
Definition init_coherence_prop : Prop := forall c ctx s,
    result_functor id id
        (init C1 c ctx s) =
    init C2 c ctx (id s).
```

```
Axiom init_coherence_pf : init_coherence_prop.

Definition recv_coherence_prop : Prop := forall c ctx st op_msg,
    result_functor (fun '(st, l) => (id st, l)) id
        (receive C1 c ctx st op_msg) =
    receive C2 c ctx (id st) (option_map embed_entrypoint op_msg).
Axiom recv_coherence_pf : recv_coherence_prop.


(* construct a contract morphism *)
Definition f : ContractMorphism C1 C2 := {|
    setup_morph := id ;
    msg_morph := embed_entrypoint ;
    state_morph := id ;
    error_morph := id ;
    (* coherence *)
    init_coherence := init_coherence_pf ;
    recv_coherence := recv_coherence_pf ;
|}.


(* TODO get rid of this *)
Tactic Notation "is_sp_destruct" :=
    match goal with
    | is_sp : is_structured_pool _ |- _ =>
        unfold is_structured_pool in is_sp;
        destruct is_sp  as [none_fails_pf is_sp'];
        destruct is_sp' as [msg_destruct_pf is_sp'];
        (* isolate the pool entrypoint specification *)
        destruct is_sp' as [pool_entrypoint_check_pf is_sp'];
        destruct is_sp' as [pool_emits_txns_pf is_sp'];
        destruct is_sp' as [pool_increases_tokens_held_pf is_sp'];
        destruct is_sp' as [pool_rates_unchanged_pf is_sp'];
        destruct is_sp' as [pool_outstanding_pf is_sp'];
        (* isolate the unpool entrypoint specification *)
        destruct is_sp' as [unpool_entrypoint_check_pf is_sp'];
        destruct is_sp' as [unpool_entrypoint_check_2_pf is_sp'];
        destruct is_sp' as [unpool_emits_txns_pf is_sp'];
        destruct is_sp' as [unpool_decreases_tokens_held_pf is_sp'];
        destruct is_sp' as [unpool_rates_unchanged_pf is_sp'];
        destruct is_sp' as [unpool_outstanding_pf is_sp'];
        (* isolate the trade entrypoint specification *)
        destruct is_sp' as [trade_entrypoint_check_pf is_sp'];
        destruct is_sp' as [trade_entrypoint_check_2_pf is_sp'];
        destruct is_sp' as [trade_pricing_formula_pf is_sp'];
        destruct is_sp' as [trade_update_rates_pf is_sp'];
        destruct is_sp' as [trade_update_rates_formula_pf is_sp'];
        destruct is_sp' as [trade_emits_transfers_pf is_sp'];
        destruct is_sp' as [trade_tokens_held_update_pf is_sp'];
        destruct is_sp’ as [trade_outstanding_update_pf is_sp’];
        destruct is_sp’ as [trade_pricing_pf is_sp’];
        destruct is_sp’ as [trade_amounts_nonnegative_pf is_sp’];
        (* isolate the specification of all other entrypoints *)
        destruct is_sp’ as [other_rates_unchanged_pf is_sp’];
        destruct is_sp’ as [other_balances_unchanged_pf is_sp’];
        destruct is_sp’ as [other_outstanding_unchanged_pf is_sp’];
        (* isolate the specification of calc_rx’ and calc_delta_y *)
        destruct is_sp’ as [update_rate_stays_positive_pf is_sp’];
        destruct is_sp’ as [rate_decrease_pf is_sp’];
        destruct is_sp’ as [rates_balance_pf is_sp’];
        destruct is_sp’ as [rates_balance_2_pf is_sp’];
        destruct is_sp’ as [trade_slippage_pf is_sp’];
        destruct is_sp’ as [trade_slippage_2_pf is_sp’];
        destruct is_sp’ as [arbitrage_lt_pf is_sp’];
        destruct is_sp’ as [arbitrage_gt_pf is_sp’];
        (* isolate the initialization specification *)
        destruct is_sp’ as [initialized_with_positive_rates_pf is_sp’];
        destruct is_sp’ as [initialized_with_zero_balance_pf is_sp’];
        destruct is_sp’ as [initialized_with_zero_outstanding_pf is_sp’];
        destruct is_sp’ as [initialized_with_init_rates_pf initialized_with_pool_token_pf]
end.


(* Prove the following about C1 using C2 and f *)
Theorem pullback_unpool_emits_txns : unpool_emits_txns C1.
Proof.
    pose proof is_sp as is_sp.
    is_sp_destruct.
    pose proof recv_coherence_pf as recv_eq.
    unfold recv_coherence_prop in recv_eq.
    unfold unpool_emits_txns.
    intros * recv_some.
    pose proof (unpool_emits_txns_pf cstate chain ctx msg_payload cstate’ acts )
    as sp_spec_result.
    pose proof (recv_eq chain ctx cstate (Some (unpool msg_payload))) as recv_eq.
    unfold result_functor in recv_eq.
    rewrite recv_some in recv_eq.
    unfold embed_entrypoint in recv_eq.
    cbn in recv_eq.
    pose proof (eq_sym recv_eq) as recv_eq’.
    now apply sp_spec_result in recv_eq’.
Qed.

End TransportHoare.
```

B.4 A Mathematical Characterization of Contract Upgrades

Listing B.17: The definitions of the various contracts in the upgradeability decomposition.

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

(** Base’ contract C_b’ (to be C_b) *)
Inductive entrypoint_b’ :=
| upgrade_fun_b (s’ : N -> N).
Record storage_b := { s_b : N -> N ; }.
Definition setup_b := storage_b.
Definition error_b := N.
Definition result_b : Type := ResultMonad.result (storage_b * list ActionBody) error_b.

(** Version contracts C_f v, for v : storage_b *)
Inductive entrypoint_version := | next_f (u : unit).
Record storage_version := { n_f : N }.
Definition entrypoint_f : storage_b -> Type := fun v => entrypoint_version.
Definition storage_f    : storage_b -> Type := fun v => storage_version.
Definition setup_f      : storage_b -> Type := fun v => N.
Definition error_f      : storage_b -> Type := fun v => N.
Definition result_f     : storage_b -> Type :=
    fun v => ResultMonad.result ((storage_f v) * list ActionBody) (error_f v).

Section Serialization.
    Section SerializeFunctionType.
        Context `{Serializable A} `{Serializable B}.

        (* for simplicity, we assume that function types are serializable *)
        Definition serialize_nn (f : A -> B) : SerializedValue. Admitted.
        Definition deserialize_nn (val : SerializedValue) : option (A -> B). Admitted.
        Lemma deserialize_serialize_nn (f : A -> B) :
            deserialize_nn (serialize_nn f) = Some f.
        Admitted.

        Global Instance nn_serializable : Serializable (A -> B) :=
        {| serialize := serialize_nn ;
            deserialize := deserialize_nn ;
            deserialize_serialize := deserialize_serialize_nn ; |}.
    End SerializeFunctionType.

    (* assuming function types are serializable, ... *)
    Global Instance storage_serializable : Serializable storage :=
        Derive Serializable storage_rect<Build_storage>.
    Global Instance entrypoint_serializable : Serializable entrypoint :=
        Derive Serializable entrypoint_rect<next,upgrade_fun>.
    Global Instance storage_b_serializable : Serializable storage_b :=
        Derive Serializable storage_b_rect<Build_storage_b>.
    Global Instance entrypoint_s_serializable : Serializable entrypoint_b' :=
        Derive Serializable entrypoint_b'_rect<upgrade_fun_b>.
    Global Instance storage_f_serializable v : Serializable (storage_f v) :=
        Derive Serializable storage_version_rect<Build_storage_version>.
    Global Instance entrypoint_f_serializable v : Serializable (entrypoint_f v) :=
        Derive Serializable entrypoint_version_rect<next_f>.
End Serialization.

(** Contract, init, and receive definitions *)

(** Main contract C *)
(* init *)
Definition init (_ : Chain)
                (_ : ContractCallContext)
                (init_state : setup)
                : ResultMonad.result storage N :=
    Ok (init_state).

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

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

(** Base' Contract (to be the base contract) *)
(* init *)
Definition init_b' (_ : Chain)
```

```
                (_ : ContractCallContext)
                (init_state_b : setup_b)
                : ResultMonad.result storage_b N :=
    Ok (init_state_b).

(* receive *)
Definition receive_b’ (_ : Chain)
                (_ : ContractCallContext)
                (_ : storage_b)
                (msg : option entrypoint_b’)
                : result_b :=
    match msg with
    | Some (upgrade_fun_b s_new) =>
        let st := {| s_b := s_new ; |} in
        Ok (st, [])
    | None => Err 0
    end.

(* the contract C_b’ *)
Definition C_b’ : Contract setup_b entrypoint_b’ storage_b error_b :=
    build_contract init_b’ receive_b’.

(** For the morphisms, we define the base contract *)
Definition C_b := pointed_contract C_b’.
Definition entrypoint_b := (entrypoint_b’ + unit)%type.

(** The family of version contracts C_f *)
(* init *)
Definition init_f (version : storage_b)
                (_ : Chain)
                (_ : ContractCallContext)
                (n : setup_f version)
                : ResultMonad.result (storage_f version) (error_f version) :=
    let storage_f := {| n_f := n ; |} in
    Ok (storage_f).

(* receive *)
Definition receive_f (version : storage_b)
                (_ : Chain)
                (_ : ContractCallContext)
                (storage_f : storage_f version)
                (msg : option (entrypoint_f version))
                : result_f version :=
    match msg with
    | Some (next_f _) =>
        let st := {| n_f := version.(s_b) storage_f.(n_f) ; |} in
        Ok (st, [])
    | None => Err 0
    end.
```

```

(* the contract C_f *)
Definition C_f (version : storage_b) : Contract (setup_f version) (entrypoint_f version) (storage_f version) (error_f version) :=
    build_contract (init_f version) (receive_f version).

End ContractDefinitions.
```

B.4.1 Isolating Mutable and Immutable Parts

Listing B.18: The quotient onto the base contract.

```
(** f_p : C ->> C_b *)
Section Quotient.
Definition zero_fn : N -> N := (fun (x : N) => 0).
Definition null_storage_b : storage_b := {| s_b := zero_fn |}.
Definition null_setup_b : setup_b := {| s_b := zero_fn |}.

(* component morphisms *)
Definition msg_morph_p (e : entrypoint) : entrypoint_b :=
    match e with
    | next _ => inr tt (* not upgrade functionality *)
    | upgrade_fun s' => inl (upgrade_fun_b s') (* corresponds to an upgrade *)
    end.
Definition state_morph_p : storage -> storage_b := (fun (x : storage) => {| s_b := x.(s) ; |}).
Definition setup_morph_p : setup -> setup_b := (fun (x : setup) => {| s_b := x.(s) |}).
Definition error_morph_p : error -> error_b := (fun (x : error) => x).

(* the coherence results *)
Lemma init_coherence_p :
    init_coherence_prop C C_b
        setup_morph_p state_morph_p error_morph_p.
Proof. unfold init_coherence_prop. auto. Qed.

Lemma recv_coherence_p :
    recv_coherence_prop C C_b
        msg_morph_p state_morph_p error_morph_p.
Proof.
    unfold recv_coherence_prop.
    intros.
    unfold result_functor.
    cbn.
    destruct op_msg; cbn.
    -   unfold msg_morph_p.
        destruct e eqn:H_e.
        +   now destruct st.
        +   now unfold state_morph_p.
    -   now unfold error_morph_p.
Qed.

(* construct the morphism *)
Definition f_p : ContractMorphism C C_b :=
    build_contract_morphism C C_b
        (* the morphisms *)
        setup_morph_p msg_morph_p state_morph_p error_morph_p
        (* coherence *)
        init_coherence_p recv_coherence_p.

End Quotient.
```

Listing B.19: The family of contract embeddings.

```
(** f_i : forall v, C_f v -> C *)
Section Embedding.

Definition msg_morph_i (v : storage_b) (e : entrypoint_f v) : entrypoint :=
    match e with
    | next_f _ => next tt
    end.
Definition setup_morph_i (v : storage_b) (st_f : setup_f v) : setup := {|
    n := st_f ;
    s := s_b v ; |}.
Definition state_morph_i (v : storage_b) (st_f : storage_f v) : storage :=
    {| n := st_f.(n_f) ; s := s_b v ; |}.
Definition error_morph_i (v : storage_b) : error_f v -> error := id.

(* the coherence results *)
Lemma init_coherence_i (v : storage_b) :
    init_coherence_prop (C_f v) C (setup_morph_i v) (state_morph_i v) (error_morph_i v).
Proof. unfold init_coherence_prop. auto. Qed.

Lemma recv_coherence_i (v : storage_b) :
    recv_coherence_prop (C_f v) C (msg_morph_i v) (state_morph_i v) (error_morph_i v).
Proof.
    unfold recv_coherence_prop.
    intros.
    destruct op_msg; auto.
    now destruct e.
Qed.

(* construct the morphism *)
Definition fi_param (v : storage_b) : ContractMorphism (C_f v) C :=
    build_contract_morphism (C_f v) C
        (* the morphisms *)
        (setup_morph_i v) (msg_morph_i v) (state_morph_i v) (error_morph_i v)
        (* coherence results *)
        (init_coherence_i v) (recv_coherence_i v).

End Embedding.
```

B.4.2 Decomposing Upgradeability

Listing B.20: A proof of the upgradeability decomposition.

```
(** Here we prove the core result that the family of versioned contracts and the
    base contract given above provide an upgradeability decomposition of our contract C. *)
Section UpgradeabilityDecomposition.
Definition extract_version (m : entrypoint_b’) : storage_b :=
    match m with | upgrade_fun_b s_b => {| s_b := s_b |} end.
Definition new_version_state old_v msg (st : storage_f old_v) : storage_f (extract_version msg) := st.

Theorem upgradeability_decomposition :
    upgradeability_decomposition fi_param f_p extract_version new_version_state.
Proof.
    unfold upgradeability_decomposition.
    repeat split.
    (* msg_required *)
    -   unfold ContractMorphisms.msg_required.
        intros.
        now exists 0.
    (* init_versioned *)
    -   unfold init_versioned.
        intros ? ? ? s init_ok.
        destruct init_state as [n_i s_i].
        unfold is_versioned.
        now exists {| s_b := s_i |}, {| n_f := n_i |}.
    (* msg_decomposable *)
    (* -> *)
    -   simpl.
        intro H_null.
        unfold msg_morph_p in H_null.
        destruct m; try inversion H_null.
        destruct u.
        now exists (next_f tt).
    (* <- *)
    -   simpl.
        intro H_preim.
        unfold msg_morph_i in H_preim.
        destruct H_preim as [m’ H_preim].
        destruct m’.
        unfold msg_morph_p.
        now destruct m; try inversion H_preim.
    (* states categorized *)
    (* -> *)
    -   simpl.
        intro H_preim.
        destruct H_preim as [st_f H_preim].
        destruct st_f as [nf].
        unfold state_morph_i in H_preim.
        simpl in H_preim.
        destruct st as [n s].
        now inversion H_preim.
    (* <- *)
    -   simpl.
        unfold state_morph_p, state_morph_i.
        destruct c_version as [sb], st as [n s].
        cbn.
        intro H.
        inversion H.
        now exists {| n_f := n |}.
    (* version transition *)
    -   unfold version_transition.
        simpl.
        unfold msg_morph_p, state_morph_i.
        intros * H_cstate_preim ? ? ? ? ? ? recv_some is_upgrade_msg.
        destruct msg; try inversion is_upgrade_msg.
        inversion recv_some.
        f_equal.
        unfold new_version_state.
        now rewrite H_cstate_preim.
Qed.

End UpgradeabilityDecomposition.
```

Listing B.21: Some results provable due to the decomposition.

```
Section Decomposition.

(** Theorem: all reachable contract states are versioned according to this indexing *)
Theorem all_states_versioned :
    forall bstate caddr (trace : ChainTrace empty_state bstate),
    (* where C is at caddr with state cstate, *)
    env_contracts bstate caddr = Some (C : WeakContract) ->
    exists (cstate : storage),
    contract_state bstate caddr = Some cstate /\
    (* then every contract state cstate is versioned *)
    is_versioned fi_param cstate.
Proof.
    intros * ? c_at_caddr.
    pose proof (versioned_invariant fi_param f_p extract_version new_version_state bstate
    caddr trace c_at_caddr).
    destruct H as [cstate [cstate_st versioned]].
    exists cstate.
    split; auto.
    apply (versioned upgradeability_decomposition).
Qed.

(** Theorem: Versioning moves along as we've described it. *)
Theorem upgrade_decomposed :
    (* Forall versioned contract states (incl. all reachable states), *)
    forall cstate c_version cstate_f,
    cstate = state_morph (C_f c_version) C (fi_param c_version) cstate_f ->
    (* And forall calls to the versioned contract *)
    forall chain ctx m new_state new_acts,
    receive chain ctx cstate (Some m) = Ok (new_state, new_acts) ->
    (* the contract state either stays within this version *)
    (exists cstate_f', new_state = state_morph (C_f c_version) C (fi_param c_version)
    cstate_f') \/
    (* or it moves onto a new version as we've described it. *)
    (exists c_version' cstate_f',
    new_state = state_morph (C_f c_version') C (fi_param c_version') cstate_f').
Proof.
    intros * cstate_preim * recv_some.
    apply (upgradeability_decomposed fi_param f_p extract_version new_version_state cstate
     c_version
    cstate_f upgradeability_decomposition cstate_preim chain ctx m new_state new_acts
    recv_some).
Qed.

End Decomposition.
```

B.4.3 Upgradeable Contracts are Fiber Bundles: A Digression

Listing B.22: The fiber bundle of our upgradeability decomposition admits a section.

```
Section FiberBundle.

(* A section of p is a morphism C_b' -> C such that C_b' -> C -> C_b = C_b' -> C_b *)
Definition setup_morph_s n : setup_b -> setup := (fun (x : setup_b) => {| n := n ; s := x
    .(s_b) |}).
Definition msg_morph_s (e : entrypoint_b') : entrypoint :=
    match e with | upgrade_fun_b s' => upgrade_fun s' end.
Definition state_morph_s n : storage_b -> storage :=
    (fun (x : storage_b) => {| n := n ; s := x.(s_b) ; |}).
Definition error_morph_s : error_b -> error := (fun (x : error_b) => x).

Definition fp_rinv (n : N) : ContractMorphism C_b' C.
Proof.
    apply (build_contract_morphism C_b' C
        (setup_morph_s n) msg_morph_s (state_morph_s n) error_morph_s).
    -   intros.
        simpl.
        now unfold state_morph_s, setup_morph_s, init.
    -   intros.
        simpl.
        unfold result_functor, state_morph_s, error_morph_s, receive.
        simpl.
        destruct op_msg; cbn; auto.
        now destruct e.
Defined.

Theorem p_rinv_section (n : N) :
    compose_cm f_p (fp_rinv n) = pointed_include C_b'.
Proof.
    unfold compose_cm, pointed_include.
    apply eq_cm; cbn.
    -   now unfold setup_morph_p, setup_morph_s.
    -   unfold msg_morph_p, msg_morph_s.
        apply functional_extensionality.
        intro e.
        now destruct e.
    -   now unfold state_morph_p, state_morph_s.
    -   now unfold error_morph_p, error_morph_s.
Qed.

End FiberBundle.
```

Appendix C

Proofs and Definitions of Chapter 6

C.1 Bisimulations of Contracts

C.1.1 Contract Trace Morphisms

Listing C.1: The formal definition of a contract trace morphism.

```
Section ContractTraceMorphism.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}.

Record ContractTraceMorphism
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :=
    build_ct_morph {
        (* a function of state types *)
        ct_state_morph : State1 -> State2 ;
        (* init state C1 -> init state C2 *)
        genesis_fixpoint : forall init_cstate,
            is_genesis_cstate C1 init_cstate ->
            is_genesis_cstate C2 (ct_state_morph init_cstate) ;
        (* coherence *)
        cstep_morph : forall state1 state2,
            ContractStep C1 state1 state2 ->
            ContractStep C2 (ct_state_morph state1) (ct_state_morph state2) ;
    }.

End ContractTraceMorphism.
```

Listing C.2: The identity contract trace morphism.

```
Section IdentityCTMorphism.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}.

Definition id_genesis_fixpoint (C : Contract Setup1 Msg1 State1 Error1)
    init_cstate
    (gen_C : is_genesis_cstate C init_cstate) :
    is_genesis_cstate C (id init_cstate) :=
    gen_C.

Definition id_cstep_morph (C : Contract Setup1 Msg1 State1 Error1)
    state1 state2
    (step : ContractStep C state1 state2) :
    ContractStep C (id state1) (id state2) :=
    step.

Definition id_ctm (C : Contract Setup1 Msg1 State1 Error1) : ContractTraceMorphism C C := {|
    ct_state_morph := id ;
    genesis_fixpoint := id_genesis_fixpoint C ;
    cstep_morph := id_cstep_morph C ;
|}.

End IdentityCTMorphism.
```

Listing C.3: Equality of contract trace morphisms.

```
Section EqualityOfCTMorphisms.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}.

Lemma eq_ctm_dep
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2)
    (ct_st_m : State1 -> State2)
    (gen_fix1 gen_fix2 : forall init_cstate,
        is_genesis_cstate C1 init_cstate ->
        is_genesis_cstate C2 (ct_st_m init_cstate))
    (cstep_m1 cstep_m2 : forall state1 state2,
        ContractStep C1 state1 state2 ->
        ContractStep C2 (ct_st_m state1) (ct_st_m state2)) :
    cstep_m1 = cstep_m2 ->
    {| ct_state_morph := ct_st_m ;
      genesis_fixpoint := gen_fix1 ;
      cstep_morph := cstep_m1 ; |}
    =
    {| ct_state_morph := ct_st_m ;
       genesis_fixpoint := gen_fix2 ;
       cstep_morph := cstep_m2 ; |}.
Proof.
    intro cstep_equiv.
    subst.
    f_equal.
    apply proof_irrelevance.
Qed.

End EqualityOfCTMorphisms.
```

Listing C.4: Composition of contract trace morphisms.

```
Section CTMorphismComposition.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
        `{Serializable Setup3} `{Serializable Msg3} `{Serializable State3} `{Serializable Error3}
        {C1 : Contract Setup1 Msg1 State1 Error1}
        {C2 : Contract Setup2 Msg2 State2 Error2}
        {C3 : Contract Setup3 Msg3 State3 Error3}.

Definition genesis_compose (m2 : ContractTraceMorphism C2 C3) (m1 : ContractTraceMorphism C1 C2)
    init_cstate (gen_C1 : is_genesis_cstate C1 init_cstate) :
    is_genesis_cstate C3 (compose (ct_state_morph C2 C3 m2) (ct_state_morph C1 C2 m1) init_cstate) :=
  match m2 with
  | build_ct_morph _ _ _ gen_fix2 step2 =>
      match m1 with
      | build_ct_morph _ _ _ gen_fix1 step1 =>
          gen_fix2 _ (gen_fix1 _ gen_C1)
      end
  end.

Definition cstep_compose (m2 : ContractTraceMorphism C2 C3) (m1 : ContractTraceMorphism C1 C2)
    state1 state2 (step : ContractStep C1 state1 state2) :
    ContractStep C3
        (compose (ct_state_morph C2 C3 m2) (ct_state_morph C1 C2 m1) state1)
        (compose (ct_state_morph C2 C3 m2) (ct_state_morph C1 C2 m1) state2) :=
  match m2, m1 with
  | build_ct_morph _ _ _ _ step2, build_ct_morph _ _ _ _ step1 =>
          step2 _ _ (step1 _ _ step)
  end.

```

```
Definition compose_ctm
    (m2 : ContractTraceMorphism C2 C3)
    (m1 : ContractTraceMorphism C1 C2) : ContractTraceMorphism C1 C3 :=
{|
    ct_state_morph := compose (ct_state_morph _ _ m2) (ct_state_morph _ _ m1) ;
    genesis_fixpoint := genesis_compose m2 m1 ;
    cstep_morph := cstep_compose m2 m1 ;
|}.

End CTMorphismComposition.
```

Listing C.5: Some results about contract trace morphism composition, including that composition is associative, and that left and right composition with the identity is trivial.

```
Section CTMorphismCompositionResults.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable
    Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable
    Error2}
        `{Serializable Setup3} `{Serializable Msg3} `{Serializable State3} `{Serializable
    Error3}
        `{Serializable Setup4} `{Serializable Msg4} `{Serializable State4} `{Serializable
    Error4}
        { C1 : Contract Setup1 Msg1 State1 Error1 }
        { C2 : Contract Setup2 Msg2 State2 Error2 }
        { C3 : Contract Setup3 Msg3 State3 Error3 }
        { C4 : Contract Setup4 Msg4 State4 Error4 }.

(* Composition is associative *)
Lemma compose_ctm_assoc
    (f : ContractTraceMorphism C1 C2)
    (g : ContractTraceMorphism C2 C3)
    (h : ContractTraceMorphism C3 C4) :
    compose_ctm h (compose_ctm g f) =
    compose_ctm (compose_ctm h g) f.
Proof. now destruct f, g, h. Qed.

(* Composition with the identity is trivial *)
Lemma compose_id_ctm_left (f : ContractTraceMorphism C1 C2) :
    compose_ctm (id_ctm C2) f = f.
Proof. now destruct f. Qed.

Lemma compose_id_ctm_right (f : ContractTraceMorphism C1 C2) :
    compose_ctm f (id_ctm C1) = f.
Proof. now destruct f. Qed.

End CTMorphismCompositionResults.
```

C.1.2 The Lifting Theorem

Listing C.6: The lifting theorem.

```
Section LiftingTheorem.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
        {C1 : Contract Setup1 Msg1 State1 Error1}
        {C2 : Contract Setup2 Msg2 State2 Error2}.

Definition lift_genesis (f : ContractMorphism C1 C2) :
    forall init_cstate,
        is_genesis_cstate C1 init_cstate ->
        is_genesis_cstate C2 (state_morph C1 C2 f init_cstate).
Proof.
    destruct f as [setup_morph msg_morph state_morph error_morph i_coh r_coh].
    cbn.
    intros * genesis.
    unfold is_genesis_cstate in *.
    destruct genesis as [c [ctx [s init_coh]]].
    exists c, ctx, (setup_morph s).
    rewrite <- i_coh.
    unfold result_functor.
    now destruct (init C1 c ctx s).
Defined.

Definition lift_cstep_morph (f : ContractMorphism C1 C2) :
    forall state1 state2,
        ContractStep C1 state1 state2 ->
        ContractStep C2
            (state_morph C1 C2 f state1)
            (state_morph C1 C2 f state2).
Proof.
    destruct f as [setup_morph msg_morph state_morph error_morph i_coh r_coh].
    cbn.
    intros * step.
    destruct step as [seq_chain seq_ctx seq_msg seq_new_acts recv_step].
    apply (build_contract_step C2 (state_morph state1) (state_morph state2) seq_chain seq_ctx
        (option_map msg_morph seq_msg) seq_new_acts).
    rewrite <- r_coh.
    unfold result_functor.
    destruct (receive C1 seq_chain seq_ctx state1 seq_msg);
    try destruct t;
    now inversion recv_step.
Defined.

(** Lifting Theorem *)
```

```
Definition cm_lift_ctm (f : ContractMorphism C1 C2) : ContractTraceMorphism C1 C2 :=
    build_ct_morph _ _ (state_morph _ _ f) (lift_genesis f) (lift_cstep_morph f).

End LiftingTheorem.
```

Listing C.7: Some results on the lifting theorem, including that the identity lifts to the identity, and compositions to compositions.

```
Section LiftingTheoremResults.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
        `{Serializable Setup3} `{Serializable Msg3} `{Serializable State3} `{Serializable Error3}
        {C1 : Contract Setup1 Msg1 State1 Error1}
        {C2 : Contract Setup2 Msg2 State2 Error2}
        {C3 : Contract Setup3 Msg3 State3 Error3}.

(* id lifts to id *)
Theorem cm_lift_ctm_id :
    cm_lift_ctm (id_cm C1) = id_ctm C1.
Proof.
    unfold cm_lift_ctm, id_ctm.
    simpl.
    apply (eq_ctm_dep C1 C1 (@id State1)).
    apply functional_extensionality_dep.
    intro st1.
    apply functional_extensionality_dep.
    intro st1'.
    apply functional_extensionality_dep.
    intro cstep.
    destruct cstep as [s_chn s_ctx s_msg s_nacts s_recv].
    unfold id_cstep_morph.
    cbn.
    unfold option_map.
    destruct s_msg;
    cbn;
    f_equal;
    apply proof_irrelevance.
Qed.

(* compositions lift to compositions *)
Theorem cm_lift_ctm_compose
    (g : ContractMorphism C2 C3) (f : ContractMorphism C1 C2) :
    cm_lift_ctm (compose_cm g f) =
    compose_ctm (cm_lift_ctm g) (cm_lift_ctm f).
Proof.
    unfold cm_lift_ctm, compose_ctm.
    cbn.
    apply (eq_ctm_dep C1 C3 (compose (state_morph C2 C3 g) (state_morph C1 C2 f))).
    apply functional_extensionality_dep.
    intro st1.
    apply functional_extensionality_dep.
    intro st1'.
    apply functional_extensionality_dep.
    intro cstep.
    destruct cstep as [s_chn s_ctx s_msg s_nacts s_recv].
    unfold lift_cstep_morph.
    destruct g as [smorph_g msgmorph_g stmorph_g errmorph_g initcoh_g recvcoh_g].
    destruct f as [smorph_f msgmorph_f stmorph_f errmorph_f initcoh_f recvcoh_f].
    cbn.
    destruct s_msg;
    cbn;
    f_equal;
    apply proof_irrelevance.
Qed.

End LiftingTheoremResults.
```

C.1.3 Contract Bisimulations

Listing C.8: The definition and results about contract bisimulations.

```
Section ContractBisimulation.

Section ContractTraceIsomorphism.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable
    Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable
    Error2}
        {C1 : Contract Setup1 Msg1 State1 Error1}
        {C2 : Contract Setup2 Msg2 State2 Error2}.

(* a bisimulation of contracts, or an isomorphism of contract traces *)
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.

(* contract isomorphism -> contract trace isomorphism *)
Corollary ciso_to_ctiso (f : ContractMorphism C1 C2) (g : ContractMorphism C2 C1) :
    is_iso_cm f g -> is_iso_ctm (cm_lift_ctm f) (cm_lift_ctm g).
Proof.
    unfold is_iso_cm, is_iso_ctm.
    intro iso_cm.
    destruct iso_cm as [iso_cm_l iso_cm_r].
    rewrite <- (cm_lift_ctm_compose g f).
    rewrite <- (cm_lift_ctm_compose f g).
    rewrite iso_cm_l.
    rewrite iso_cm_r.
    now repeat rewrite cm_lift_ctm_id.
Qed.

End ContractTraceIsomorphism.

(* the definition of bisimilar contracts *)
Definition contracts_bisimilar
    `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
    `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :=
    exists (f : ContractTraceMorphism C1 C2) (g : ContractTraceMorphism C2 C1),
    is_iso_ctm f g.

(* bisimilarity is an equivalence relation *)
Lemma bisim_refl
    `{Serializable Setup} `{Serializable Msg} `{Serializable State} `{Serializable Error}
    (C : Contract Setup Msg State Error) :
    contracts_bisimilar C C.
Proof.
    exists (id_ctm C), (id_ctm C).
    unfold is_iso_ctm.
    split; apply compose_id_ctm_left.
Qed.

Lemma bisim_sym
    `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
    `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :
    contracts_bisimilar C1 C2 ->
    contracts_bisimilar C2 C1.
Proof.
    intro c_bisim.
    unfold contracts_bisimilar in *.
    destruct c_bisim as [f [f' iso_f_g]].
    exists f', f.
    unfold is_iso_ctm in *.
    destruct iso_f_g as [f_id1 f_id2].
    split.
    -   apply f_id2.
    -   apply f_id1.
Qed.

Lemma bisim_trans
    `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
    `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
    `{Serializable Setup3} `{Serializable Msg3} `{Serializable State3} `{Serializable Error3}
    {C1 : Contract Setup1 Msg1 State1 Error1}
    {C2 : Contract Setup2 Msg2 State2 Error2}
    {C3 : Contract Setup3 Msg3 State3 Error3} :
    contracts_bisimilar C1 C2 /\ contracts_bisimilar C2 C3 ->
    contracts_bisimilar C1 C3.
Proof.
    intros c_bisims.
    destruct c_bisims as [[f [f' iso_f]] [g [g' iso_g]]].
    unfold contracts_bisimilar in *.
    exists (compose_ctm g f), (compose_ctm f' g').
    destruct iso_g as [g_id1 g_id2].
    destruct iso_f as [f_id1 f_id2].
    unfold is_iso_ctm.
    split.
    -   rewrite <- compose_ctm_assoc.
        replace (compose_ctm g' (compose_ctm g f)) with (compose_ctm (compose_ctm g' g) f).
        2:{ now rewrite <- compose_ctm_assoc. }
        rewrite g_id1.
        rewrite compose_id_ctm_left.
        apply f_id1.
    -   rewrite <- compose_ctm_assoc.
        replace (compose_ctm f (compose_ctm f' g')) with (compose_ctm (compose_ctm f f') g').
        2:{ now rewrite <- compose_ctm_assoc. }
        rewrite f_id2.
        rewrite compose_id_ctm_left.
        apply g_id2.
Qed.

(* an isomorphism of contracts lifts to a bisimulation *)
Theorem c_iso_to_bisim
    `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
    `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
    {C1 : Contract Setup1 Msg1 State1 Error1}
    {C2 : Contract Setup2 Msg2 State2 Error2} :
    contracts_isomorphic C1 C2 -> contracts_bisimilar C1 C2.
```

```
Proof.
    intro c_iso.
    destruct c_iso as [f [g [is_iso_1 is_iso_2]]].
    unfold contracts_bisimilar.
    exists (cm_lift_ctm f), (cm_lift_ctm g).
    unfold is_iso_ctm.
    split;
    rewrite <- cm_lift_ctm_compose;
    try rewrite is_iso_1;
    try rewrite is_iso_2;
    now rewrite cm_lift_ctm_id.
Qed.

End ContractBisimulation.
```

C.1.4 Discussion: Propositional Indistinguishability

Listing C.9: An example illustrating the degree to which bisimilar contracts are propositionally indistinguishable.

```
(*  Bisimilar contracts have very similar invariants, depending on the state
    isomorphism.

    In this example, we show that, for bisimilar contracts C1 and C2, if there
    is a state invariant of C1 which is preserved by the state isomorphism of
    the bisimulation, then that same state invariant holds for C2.
*)

Section PropositionalIndistinguishability.
Context {Base : ChainBase}.

Context
    `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
    `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
(* Consider contracts C1 and C2 ... *)
    {C1 : Contract Setup1 Msg1 State1 Error1}
    {C2 : Contract Setup2 Msg2 State2 Error2}.

(* such that C1 and C2 are bisimilar. *)
Context {m1 : ContractTraceMorphism C1 C2} {m2 : ContractTraceMorphism C2 C1}
        {C1_C2_bisim : is_iso_ctm m1 m2}.

(* Assume that both State1 and State2 have a constant in storage, given by a function
    const_in_stor *)
Context { const_in_stor_C1 : State1 -> nat } { const_in_stor_C2 : State2 -> nat }
```

```
(* and assume that this constant is invariant under the ct_state isomorphism *)
        { const_pres : forall st,
            const_in_stor_C2 (ct_state_morph C1 C2 m1 st) = const_in_stor_C1 st }.

(* Assume that an invariant holds for C1 ... *)
Axiom invariant_C1_init : forall c ctx s cstate,
    init C1 c ctx s = Ok cstate ->
    (const_in_stor_C1 cstate > 0)%nat.

Axiom invariant_C1_recv : forall c ctx cstate op_msg new_st nacts,
    receive C1 c ctx cstate op_msg = Ok (new_st, nacts) ->
    (const_in_stor_C1 new_st > 0)%nat.

(* Because the invariant is preserved under the state isomorphism of the bisimulation,
    we can prove that the same invariant of C2 *)
Theorem invariant_C2 bstate caddr (trace : ChainTrace empty_state bstate):
    (* Forall reachable states with contract at caddr, *)
    env_contracts bstate caddr = Some (C2 : WeakContract) ->
    (* such that cstate is the state of the contract, *)
    exists (cstate : State2),
    contract_state bstate caddr = Some cstate /\
    (* the constant in storage is > 0 *)
    (const_in_stor_C2 cstate > 0)%nat.
Proof.
    intros.
    contract_induction; auto; intros.
    (* deployment *)
    -   assert (is_genesis_cstate C2 result)
        as is_gen2.
        {   unfold is_genesis_cstate.
            now exists chain, ctx, setup. }
        pose proof (genesis_fixpoint C2 C1 m2 result is_gen2)
        as is_gen1.
        destruct is_gen1 as [c2 [ctx2 [s2 init_ok2]]].
        pose proof (invariant_C1_init _ _ _ _ init_ok2)
        as invar_C1.
        rewrite <- const_pres in invar_C1.
        assert ((ct_state_morph C1 C2 m1 (ct_state_morph C2 C1 m2 result)) = result)
        as state_id.
        {   unfold is_iso_ctm in C1_C2_bisim.
            destruct C1_C2_bisim as [iso_l iso_r].
            unfold compose_ctm, id_ctm in *.
            replace (ct_state_morph C1 C2 m1 (ct_state_morph C2 C1 m2 result))
            with (Basics.compose (ct_state_morph C1 C2 m1) (ct_state_morph C2 C1 m2)
    result).
            2:{ auto. }
            inversion iso_r.
            now rewrite H8. }
        now rewrite state_id in invar_C1.
```

```
(* nonrecursive call *)
-   assert (ContractStep C2 prev_state new_state)
    as step_C2.
    {   apply (build_contract_step C2 prev_state new_state chain ctx msg
            new_acts receive_some). }
    pose proof (cstep_morph C2 C1 m2 prev_state new_state step_C2)
    as step_morph.
    destruct step_morph.
    pose proof (invariant_C1_recv seq_chain seq_ctx
        (ct_state_morph C2 C1 m2 prev_state) seq_msg
        (ct_state_morph C2 C1 m2 new_state) seq_new_acts recv_ok_step)
    as invar_C2.
    rewrite <- const_pres in invar_C2.
    replace (ct_state_morph C1 C2 m1 (ct_state_morph C2 C1 m2 new_state))
    with (Basics.compose (ct_state_morph C1 C2 m1) (ct_state_morph C2 C1 m2) new_state
)
    in invar_C2.
    2:{ auto. }
    assert (Basics.compose (ct_state_morph C1 C2 m1) (ct_state_morph C2 C1 m2) = id)
    as state_id.
    {   unfold is_iso_ctm in C1_C2_bisim.
        destruct C1_C2_bisim as [iso_l iso_r].
        unfold compose_ctm, id_ctm in *.
        now inversion iso_r. }
    now rewrite state_id in invar_C2.
(* recursive call *)
-   assert (ContractStep C2 prev_state new_state)
    as step_C2.
    {   apply (build_contract_step C2 prev_state new_state chain ctx msg
            new_acts receive_some). }
    pose proof (cstep_morph C2 C1 m2 prev_state new_state step_C2)
    as step_morph.
    destruct step_morph.
    pose proof (invariant_C1_recv seq_chain seq_ctx
        (ct_state_morph C2 C1 m2 prev_state) seq_msg
        (ct_state_morph C2 C1 m2 new_state) seq_new_acts recv_ok_step)
    as invar_C2.
    rewrite <- const_pres in invar_C2.
    replace (ct_state_morph C1 C2 m1 (ct_state_morph C2 C1 m2 new_state))
    with (Basics.compose (ct_state_morph C1 C2 m1) (ct_state_morph C2 C1 m2) new_state
)
    in invar_C2.
    2:{ auto. }
    assert (Basics.compose (ct_state_morph C1 C2 m1) (ct_state_morph C2 C1 m2) = id)
    as state_id.
    {   unfold is_iso_ctm in C1_C2_bisim.
        destruct C1_C2_bisim as [iso_l iso_r].
        unfold compose_ctm, id_ctm in *.
        now inversion iso_r. }
        now rewrite state_id in invar_C2.
    (* prove facts *)
    -   solve_facts.
Qed.

End PropositionalIndistinguishability.
```

C.2 Contract Systems as Bigraphs

C.2.1 Bigraphs

C.2.2 The Place Graph

Listing C.10: The formalization of an n-ary tree, or n-tree.

```
Section ntree.

Inductive ntree (T : Type) : Type :=
| Node : T -> list (ntree T) -> ntree T.

Definition singleton_ntree {T} (t : T) := Node T t nil.

(* fold/traversal for ntrees *)
Fixpoint ntree_fold_left {A T}
    (f : A -> T -> A)
    (sys : ntree T)
    (a0 : A) : A :=
    match sys with
    | Node _ t list_child_trees =>
        List.fold_left
            (fun (a0' : A) (sys' : ntree T) =>
                ntree_fold_left f sys' a0')
            list_child_trees
            (f a0 t)
    end.

(* ntree map : the functoriality of ntrees *)
Fixpoint ntree_map {T T'} (f : T -> T') (tree : ntree T) : ntree T' :=
    match tree with
    | Node _ v children =>
        Node T' (f v) (List.map (fun child => ntree_map f child) children)
    end.

Fixpoint replace_at_index {T : Type} (n : nat) (new_elem : T) (l : list T) : list T :=
  match l, n with
  | nil, _ => nil
  | _ :: tl, 0 => new_elem :: tl
  | hd :: tl, S n' => hd :: replace_at_index n' new_elem tl
  end.

Fixpoint add_tree_at_leaf {T} (orig append : ntree T) (leaf_index : list nat) : ntree T :=
    match orig, leaf_index with
    | Node _ v children, nil => Node T v (append :: children)
    | Node _ v children, i :: is =>
        match List.nth_error children i with
        | Some child => Node T v (replace_at_index i (add_tree_at_leaf child append is)
    children)
        | None => orig
        end
    end.

End ntree.
```

Listing C.11: The formal definition of a contract's place graph.

```
Definition ContractPlaceGraph
    (Setup Msg State Error : Type)
    `{sys_set : Serializable Setup}
    `{sys_msg : Serializable Msg}
    `{sys_st  : Serializable State}
    `{sys_err : Serializable Error} :=
    ntree (Contract Setup Msg State Error).
```

Listing C.12: The formal definition of a contract place graph's interface.

```
Section SystemInterface.
Context `{Serializable Setup} `{Serializable Msg} `{Serializable State} `{Serializable
    Error}.

(* system init : just initialize the root, since all contract init behaves identically *)
Definition sys_init
    (sys : ContractPlaceGraph Setup Msg State Error)
    (c : Chain)
    (ctx : ContractCallContext)
    (s : Setup) : result State Error :=
    match sys with
    | Node _ root_contract _ =>
        init root_contract c ctx s
    end.

(* system receive: take the message and state and run through the entire system.
    since systems are iteratively built so that a message not intended for a given
    contract
    returns the identity, this targets the contract in question and leaves the rest
    untouched. *)
```

```
Definition sys_receive
    (sys : ContractPlaceGraph Setup Msg State Error)
    (c : Chain)
    (ctx : ContractCallContext)
    (st : State)
    (op_msg : option Msg) : result (State * list ActionBody) Error :=
    ntree_fold_left
    (fun (recv_propogate : result (State * list ActionBody) Error)
         (contr : Contract Setup Msg State Error) =>
         match recv_propogate with
            | Ok (st0, lacts0) =>
                match receive contr c ctx st0 op_msg with
                | Ok (st1, lacts1) => Ok (st1, lacts0 ++ lacts1)
                | Err e => Err e
                end
            | Err e => Err e
         end)
    sys
    (Ok (st, nil)).

(* thes two functions give us a contract *)
Definition sys_contract (sys : ContractPlaceGraph Setup Msg State Error) :=
    build_contract (sys_init sys) (sys_receive sys).

End SystemInterface.
```

C.2.2.1 Iteratively Building a Contract System

Listing C.13: Some definitions and functions for iteratively constructing the place graph of a contract system.

```
Section IterativePlaceGraphBuild.
(* the definition of a singleton system *)
Definition singleton_place_graph
    `{Serializable Setup} `{Serializable Msg} `{Serializable State} `{Serializable Error}
    (C : Contract Setup Msg State Error)
    : ContractPlaceGraph Setup Msg State Error := singleton_ntree C.

Section IterativeSum.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable
    Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable
    Error2}.

(* an iterative add to contract systems s.t. type goals are satisfied *)

(* accepts messages on the left *)
Definition c_sum_l
```

```
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :
    Contract (Setup1 * Setup2) (Msg1 + Msg2) (State1 * State2) (Error1 + Error2).
Proof.
    destruct C1 as [init1 recv1].
    destruct C2 as [init2 recv2].
    apply build_contract.
    (* each setup must succeed, providing the new system state *)
    -   apply (fun c ctx s’ =>
            let ’(s1, s2) := s’ in
            match init1 c ctx s1, init2 c ctx s2 with
            | Ok st1, Ok st2 => Ok (st1, st2)
            | Err e, _ => Err (inl e) (* the left error is first *)
            | _, Err e => Err (inr e) (* followed by the right *)
            end).
    -   apply (fun c ctx st’ op_msg =>
            let ’(st1, st2) := st’ in
            match op_msg with
            | Some msg =>
                match msg with
                (* the message was intended for this contract,
                    so we attempt to udpate the state *)
                | inl msg =>
                    match recv1 c ctx st1 (Some msg) with
                    | Ok (new_st1, nacts) => Ok ((new_st1, st2), nacts)
                    | Err e => Err (inl e)
                    end
                (* the message was not intended for this contract, so we do nothing *)
                | inr msg => Ok (st’, nil)
                end
            | None => (* if there is no message, we call the contract with None *)
                match recv1 c ctx st1 None with
                | Ok (new_st1, nacts) => Ok ((new_st1, st2), nacts)
                | Err e => Err (inl e)
                end
            end).
Defined.

(* same as before, but accepts messages on the right now *)
Definition c_sum_r
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :
    Contract (Setup1 * Setup2) (Msg1 + Msg2) (State1 * State2) (Error1 + Error2).
Proof.
    destruct C1 as [init1 recv1].
    destruct C2 as [init2 recv2].
    apply build_contract.
    (* each setup must succeed, providing the new system state *)
    -   apply (fun c ctx s’ =>
            let '(s1, s2) := s' in
            match init1 c ctx s1, init2 c ctx s2 with
            | Ok st1, Ok st2 => Ok (st1, st2)
            | Err e, _ => Err (inl e) (* the left error is first *)
            | _, Err e => Err (inr e) (* followed by the right *)
            end).
    -   apply (fun c ctx st' op_msg =>
            let '(st1, st2) := st' in
            match op_msg with
            | Some msg =>
                match msg with
                (* the message was not intended for this contract, so we do nothing *)
                | inl msg => Ok (st', nil)
                (* the message was intended for this contract,
                    so we attempt to udpate the state *)
                | inr msg =>
                    match recv2 c ctx st2 (Some msg) with
                    | Ok (new_st2, nacts) => Ok ((st1, new_st2), nacts)
                    | Err e => Err (inr e)
                    end
                end
            | None => (* if there is no message, we call the contract with None *)
                match recv2 c ctx st2 None with
                | Ok (new_st2, nacts) => Ok ((st1, new_st2), nacts)
                | Err e => Err (inr e)
                end
            end).
Defined.

End IterativeSum.


Section IterativeChild.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable
    Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable
    Error2}.

(* add a contract as a child to a system(/nest contracts) *)
Definition sys_add_child_r
    (sys : ContractPlaceGraph Setup1 Msg1 State1 Error1)
    (C : Contract Setup2 Msg2 State2 Error2) :
    ContractPlaceGraph (Setup1 * Setup2) (Msg1 + Msg2) (State1 * State2) (Error1 + Error2)
     :=
    let T := (Contract (Setup1 * Setup2) (Msg1 + Msg2) (State1 * State2) (Error1 + Error2)
    ) in
    match sys with
    | Node _ root_contract _ =>
        match (ntree_map (fun C1 => c_sum_l C1 C) sys) with
```

```
        | Node _ root_contract’ children =>
            Node T root_contract’ (children ++ [Node T (c_sum_r root_contract C) nil])
        end
    end.

(* nest C1 C2 indicates that C2 is nested within C1 *)
Definition nest
    (C1 : Contract Setup1 Msg1 State1 Error1)
    (C2 : Contract Setup2 Msg2 State2 Error2) :
    ContractPlaceGraph (Setup1 * Setup2) (Msg1 + Msg2) (State1 * State2) (Error1 + Error2)
     :=
    let T := (Contract (Setup1 * Setup2) (Msg1 + Msg2) (State1 * State2) (Error1 + Error2)
    ) in
    Node T (c_sum_l C1 C2) [Node T (c_sum_r C1 C2) nil].

End IterativeChild.

End IterativePlaceGraphBuild.
```

C.2.2.2 System Contracts, Morphisms, and Isomorphisms

Listing C.14: The formal definition of a system morphism, or a morphism of system place graphs.

```
Record SystemMorphism
    (sys1 : ContractPlaceGraph Setup1 Msg1 State1 Error1)
    (sys2 : ContractPlaceGraph Setup2 Msg2 State2 Error2) :=
    build_system_morphism {
        (* the components of a morphism *)
        sys_setup_morph : Setup1 -> Setup2 ;
        sys_msg_morph   : Msg1   -> Msg2   ;
        sys_state_morph : State1 -> State2 ;
        sys_error_morph : Error1 -> Error2 ;
        (* coherence conditions *)
        sys_init_coherence : forall c ctx s,
            result_functor sys_state_morph sys_error_morph
                (sys_init sys1 c ctx s) =
            sys_init sys2 c ctx (sys_setup_morph s) ;
        sys_recv_coherence : forall c ctx st op_msg,
            result_functor (fun ’(st, l) => (sys_state_morph st, l)) sys_error_morph
                (sys_receive sys1 c ctx st op_msg) =
            sys_receive sys2 c ctx (sys_state_morph st) (option_map sys_msg_morph op_msg)
    ;
}.
```

Listing C.15: A system place graph can be defined as a contract, and system morphisms are in one-to-one correspondence with contract morphisms of that contract.

```
Definition sys_contract (sys : ContractPlaceGraph Setup Msg State Error) :=
    build_contract (sys_init sys) (sys_receive sys).

(* a system morphism is in one-to-one correspondence with a morphism of contracts,
    when we consider a system as its own contract *)
Definition cm_to_sysm
    (sys1 : ContractPlaceGraph Setup1 Msg1 State1 Error1)
    (sys2 : ContractPlaceGraph Setup2 Msg2 State2 Error2)
    (f : ContractMorphism (sys_contract sys1) (sys_contract sys2)) : SystemMorphism sys1
    sys2.
Proof.
    destruct f.
    apply (build_system_morphism sys1 sys2 setup_morph msg_morph state_morph error_morph
        init_coherence recv_coherence).
Defined.

Definition sysm_to_cm
    (sys1 : ContractPlaceGraph Setup1 Msg1 State1 Error1)
    (sys2 : ContractPlaceGraph Setup2 Msg2 State2 Error2)
    (f : SystemMorphism sys1 sys2) : ContractMorphism (sys_contract sys1) (sys_contract
    sys2).
Proof.
    destruct f as [sys_setup_morph sys_msg_morph sys_state_morph sys_error_morph
    sys_init_coh sys_recv_coh].
    apply (build_contract_morphism (sys_contract sys1) (sys_contract sys2)
        sys_setup_morph sys_msg_morph sys_state_morph sys_error_morph
        sys_init_coh sys_recv_coh).
Defined.

Lemma cm_sysm_one_to_one
    (sys1 : ContractPlaceGraph Setup1 Msg1 State1 Error1)
    (sys2 : ContractPlaceGraph Setup2 Msg2 State2 Error2) :
    compose (cm_to_sysm sys1 sys2) (sysm_to_cm sys1 sys2) = id /\
    compose (sysm_to_cm sys1 sys2) (cm_to_sysm sys1 sys2) = id.
Proof.
    split;
    unfold sysm_to_cm, cm_to_sysm;
    apply functional_extensionality;
    intro;
    now destruct x.
Qed.
```

C.3 The Link Graph

C.3.1 System Steps and System Trace

Listing C.16: The formal definition of a single system step, and chained single steps.

```
Section LinkGraph.
Context `{Serializable Setup} `{Serializable Msg} `{Serializable State} `{Serializable Error}.

(* system state stepping forward *)
Record SingleSystemStep (sys : ContractPlaceGraph Setup Msg State Error)
    (prev_sys_state next_sys_state : State) :=
    build_sys_single_step {
    sys_step_chain : Chain ;
    sys_step_ctx : ContractCallContext ;
    sys_step_msg : option Msg ;
    sys_step_nacts : list ActionBody ;
    (* we can call receive successfully *)
    sys_recv_ok_step :
        sys_receive sys sys_step_chain sys_step_ctx prev_sys_state sys_step_msg =
        Ok (next_sys_state, sys_step_nacts) ;
}.

Definition ChainedSingleSteps (sys : ContractPlaceGraph Setup Msg State Error) :=
    ChainedList State (SingleSystemStep sys).

End LinkGraph.
```

Listing C.17: The formal definition of a contract system consists of a definition of its place and link graphs. The link graph must have semantics in chained single steps.

```
Record ContractSystem
    (Setup Msg State Error : Type)
    `{sys_set : Serializable Setup}
    `{sys_msg : Serializable Msg}
    `{sys_st  : Serializable State}
    `{sys_err : Serializable Error} :=
    build_contract_system {
        (* the place and link graphs *)
        sys_place : ContractPlaceGraph Setup Msg State Error ;
        sys_link  : State -> State -> Type ;
        (* the link graph has semantics in ChanedSingleSteps *)
        sys_link_semantics : forall st1 st2,
            sys_link st1 st2 ->
            ChainedSingleSteps sys_place st1 st2 ;
    }.
```

Listing C.18: The formal definition of the steps of a contract system.

```
Definition SystemStep (sys : ContractSystem Setup Msg State Error) :=
    sys_link' sys.
```

Listing C.19: The formal definition of the a contract system's trace.

```
Definition SystemTrace (sys : ContractSystem Setup Msg State Error) :=
    ChainedList State (SystemStep sys).
```

C.4 Bisimulations of Contract Systems

C.4.1 System Trace Morphisms and System Bisimulations

Listing C.20: The formal definition of a system trace morphism.

```
Record SystemTraceMorphism
    (sys1 : ContractSystem Setup1 Msg1 State1 Error1)
    (sys2 : ContractSystem Setup2 Msg2 State2 Error2) :=
    build_st_morph {
        (* a function *)
        st_state_morph : State1 -> State2 ;
        (* init state sys1 -> init state sys2 *)
        sys_genesis_fixpoint : forall init_sys_state,
            is_genesis_sys_state sys1 init_sys_state ->
            is_genesis_sys_state sys2 (st_state_morph init_sys_state) ;
        (* step morphism *)
        sys_step_morph : forall sys_state1 sys_state2,
            SystemStep sys1 sys_state1 sys_state2 ->
            SystemStep sys2 (st_state_morph sys_state1) (st_state_morph sys_state2) ;
    }.
```

Listing C.21: The identity system trace morphism.

```
Section IdentitySTMorphism.
Context `{Serializable Setup} `{Serializable Msg} `{Serializable State} `{Serializable Error}.

Definition id_sys_genesis_fixpoint (sys : ContractSystem Setup Msg State Error)
    init_sys_state
    (gen_sys : is_genesis_sys_state sys init_sys_state) :
    is_genesis_sys_state sys (id init_sys_state) :=
    gen_sys.

Definition id_sys_step_morph (sys : ContractSystem Setup Msg State Error)
    sys_state1 sys_state2 (step : SystemStep sys sys_state1 sys_state2) :
    SystemStep sys (id sys_state1) (id sys_state2) :=
    step.

Definition id_stm (sys : ContractSystem Setup Msg State Error) : SystemTraceMorphism sys sys :=
{|
    st_state_morph := id ;
    sys_genesis_fixpoint := id_sys_genesis_fixpoint sys ;
    sys_step_morph := id_sys_step_morph sys ;
|}.

End IdentitySTMorphism.
```

Listing C.22: A lemma to assert equality of system trace morphisms.

```
Section EqualityOfSTMorphisms.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}.

Lemma eq_stm_dep
    (sys1 : ContractSystem Setup1 Msg1 State1 Error1)
    (sys2 : ContractSystem Setup2 Msg2 State2 Error2)
    (st_st_m : State1 -> State2)
    sys_gen_fix1 sys_gen_fix2
    (sys_step_m1 sys_step_m2 : forall sys_state1 sys_state2,
        SystemStep sys1 sys_state1 sys_state2 ->
        SystemStep sys2 (st_st_m sys_state1) (st_st_m sys_state2)) :
    sys_step_m1 = sys_step_m2 ->
    {| st_state_morph := st_st_m ;
       sys_genesis_fixpoint := sys_gen_fix1 ;
       sys_step_morph := sys_step_m1 ; |}
    =
    {| st_state_morph := st_st_m ;
       sys_genesis_fixpoint := sys_gen_fix2 ;
       sys_step_morph := sys_step_m2 ; |}.
Proof.
    intro cstep_equiv.
    subst.
    f_equal.
    apply proof_irrelevance.
Qed.

End EqualityOfSTMorphisms.
```

Listing C.23: The formal definition of system trace morphism composition.

```
Section STMorphismComposition.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
        `{Serializable Setup3} `{Serializable Msg3} `{Serializable State3} `{Serializable Error3}
        {sys1 : ContractSystem Setup1 Msg1 State1 Error1}
        {sys2 : ContractSystem Setup2 Msg2 State2 Error2}
        {sys3 : ContractSystem Setup3 Msg3 State3 Error3}.

Definition sys_genesis_compose
    (m2 : SystemTraceMorphism sys2 sys3) (m1 : SystemTraceMorphism sys1 sys2)
    init_sys_state (gen_s1 : is_genesis_sys_state sys1 init_sys_state) :
    is_genesis_sys_state sys3
        (compose (st_state_morph sys2 sys3 m2) (st_state_morph sys1 sys2 m1)
    init_sys_state) :=
    match m2, m1 with
    | build_st_morph _ _ _ gen_fix2 step2, build_st_morph _ _ _ gen_fix1 step1 =>
        gen_fix2 _ (gen_fix1 _ gen_s1)
    end.

Definition sys_step_compose
    (m2 : SystemTraceMorphism sys2 sys3) (m1 : SystemTraceMorphism sys1 sys2)
    sys_state1 sys_state2
    (step : SystemStep sys1 sys_state1 sys_state2) :
    SystemStep sys3
        (compose (st_state_morph sys2 sys3 m2) (st_state_morph sys1 sys2 m1) sys_state1)
        (compose (st_state_morph sys2 sys3 m2) (st_state_morph sys1 sys2 m1) sys_state2)
    :=
    match m2, m1 with
    | build_st_morph _ _ _ _ step2, build_st_morph _ _ _ _ step1 =>
        step2 _ _ (step1 _ _ step)
    end.

Definition compose_stm
    (m2 : SystemTraceMorphism sys2 sys3)
    (m1 : SystemTraceMorphism sys1 sys2) : SystemTraceMorphism sys1 sys3 :=
{|
    st_state_morph := compose (st_state_morph _ _ m2) (st_state_morph _ _ m1) ;
    sys_genesis_fixpoint := sys_genesis_compose m2 m1 ;
    sys_step_morph := sys_step_compose m2 m1 ;
|}.

End STMorphismComposition.
```

Listing C.24: Some results about system trace morphism composition.

```
Section STMorphismComposition.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable
    Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable
    Error2}
        `{Serializable Setup3} `{Serializable Msg3} `{Serializable State3} `{Serializable
    Error3}
        {sys1 : ContractSystem Setup1 Msg1 State1 Error1}
        {sys2 : ContractSystem Setup2 Msg2 State2 Error2}
        {sys3 : ContractSystem Setup3 Msg3 State3 Error3}.

Definition sys_genesis_compose
    (m2 : SystemTraceMorphism sys2 sys3) (m1 : SystemTraceMorphism sys1 sys2)
    init_sys_state (gen_s1 : is_genesis_sys_state sys1 init_sys_state) :
    is_genesis_sys_state sys3
        (compose (st_state_morph sys2 sys3 m2) (st_state_morph sys1 sys2 m1) init_sys_state) :=
    match m2, m1 with
    | build_st_morph _ _ _ gen_fix2 step2, build_st_morph _ _ _ gen_fix1 step1 =>
        gen_fix2 _ (gen_fix1 _ gen_s1)
    end.

Definition sys_step_compose
    (m2 : SystemTraceMorphism sys2 sys3) (m1 : SystemTraceMorphism sys1 sys2)
    sys_state1 sys_state2
    (step : SystemStep sys1 sys_state1 sys_state2) :
    SystemStep sys3
        (compose (st_state_morph sys2 sys3 m2) (st_state_morph sys1 sys2 m1) sys_state1)
        (compose (st_state_morph sys2 sys3 m2) (st_state_morph sys1 sys2 m1) sys_state2) :=
    match m2, m1 with
    | build_st_morph _ _ _ _ step2, build_st_morph _ _ _ _ step1 =>
        step2 _ _ (step1 _ _ step)
    end.

Definition compose_stm
    (m2 : SystemTraceMorphism sys2 sys3)
    (m1 : SystemTraceMorphism sys1 sys2) : SystemTraceMorphism sys1 sys3 :=
{|
    st_state_morph := compose (st_state_morph _ _ m2) (st_state_morph _ _ m1) ;
    sys_genesis_fixpoint := sys_genesis_compose m2 m1 ;
    sys_step_morph := sys_step_compose m2 m1 ;
|}.

End STMorphismComposition.
```

Listing C.25: A system trace isomorphism.

```
Definition is_iso_stm
    (m1 : SystemTraceMorphism sys1 sys2) (m2 : SystemTraceMorphism sys2 sys1) :=
    compose_stm m2 m1 = id_stm sys1 /\
    compose_stm m1 m2 = id_stm sys2.
```

Listing C.26: The formal definition of a bisimulation of contract systems.

```
Definition systems_bisimilar
    `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
    `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable
    Error2}
    (sys1 : ContractSystem Setup1 Msg1 State1 Error1)
    (sys2 : ContractSystem Setup2 Msg2 State2 Error2) :=
    exists (f : SystemTraceMorphism sys1 sys2) (g : SystemTraceMorphism sys2 sys1),
    is_iso_stm f g.
```

C.4.2 Lifting Theorems for Contract Systems

Listing C.27: The discrete link graph construction on a place graph.

```
Section DiscreteLinkSys.
Context `{Serializable Setup} `{Serializable Msg} `{Serializable State} `{Serializable
    Error}.

Definition discrete_link (sys : ContractPlaceGraph Setup Msg State Error) st1 st2 :=
    SingleSystemStep sys st1 st2.

Definition discrete_link_semantics (sys : ContractPlaceGraph Setup Msg State Error)
    st1 st2 (step : discrete_link sys st1 st2) :
    ChainedSingleSteps sys st1 st2 :=
    (snoc clnil step).

Definition discrete_sys (sys : ContractPlaceGraph Setup Msg State Error) := {|
    sys_place := sys ;
    sys_link := discrete_link sys ;
    sys_link_semantics := discrete_link_semantics sys ;
|}.

End DiscreteLinkSys.
```

Listing C.28: The lifting theorem for system trace morphisms.

```
Definition sm_lift_stm (f : SystemMorphism sys1 sys2) :
    SystemTraceMorphism (discrete_sys sys1) (discrete_sys sys2) :=
    build_st_morph _ _ (sys_state_morph _ _ f) (lift_sys_genesis f) (lift_sys_step_morph f
    ).
```

Listing C.29: Some results on the lifting theorem for contract trace morphisms.

```
Section LiftingTheoremResults.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable
    Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable
    Error2}
        `{Serializable Setup3} `{Serializable Msg3} `{Serializable State3} `{Serializable
    Error3}
```

```
        {sys1 : ContractPlaceGraph Setup1 Msg1 State1 Error1}
        {sys2 : ContractPlaceGraph Setup2 Msg2 State2 Error2}
        {sys3 : ContractPlaceGraph Setup3 Msg3 State3 Error3}.

(* id lifts to id *)
Lemma sm_lift_stm_id :
    sm_lift_stm (id_sm sys1) = id_stm (discrete_sys sys1).
Proof.
    apply (eq_stm_dep (discrete_sys sys1) (discrete_sys sys1) (@id State1)).
    apply functional_extensionality_dep.
    intro st1.
    apply functional_extensionality_dep.
    intro st1'.
    apply functional_extensionality_dep.
    intro sys_step.
    destruct sys_step.
    unfold lift_sys_step_morph, id_sm, discrete_sys, option_map, id_sys_step_morph.
    cbn.
    do 2 f_equal; auto.
    destruct sys_step_msg;
    apply f_equal;
    apply proof_irrelevance.
Qed.

(* compositions lift to compositions *)
Lemma sm_lift_stm_compose
    (g : SystemMorphism sys2 sys3) (f : SystemMorphism sys1 sys2) :
    sm_lift_stm (compose_sm g f) =
    compose_stm (sm_lift_stm g) (sm_lift_stm f).
Proof.
    apply (eq_stm_dep (discrete_sys sys1) (discrete_sys sys3)
        (compose (sys_state_morph sys2 sys3 g) (sys_state_morph sys1 sys2 f))).
    apply functional_extensionality_dep.
    intro st1.
    apply functional_extensionality_dep.
    intro st1'.
    apply functional_extensionality_dep.
    intro sys_step.
    induction sys_step.
    destruct g as [smorph_g msgmorph_g stmorph_g errmorph_g initcoh_g recvcoh_g].
    destruct f as [smorph_f msgmorph_f stmorph_f errmorph_f initcoh_f recvcoh_f].
    unfold lift_sys_step_morph, sys_step_compose, compose_sm.
    destruct sys_step_msg;
    cbn;
    f_equal;
    apply proof_irrelevance.
Qed.

End LiftingTheoremResults.
```

Listing C.30: Isomorphic contract systems are bisimilar under the discrete link graph.

```
Corollary sys_iso_to_bisim
    `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
    `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
    (sys1 : ContractPlaceGraph Setup1 Msg1 State1 Error1)
    (sys2 : ContractPlaceGraph Setup2 Msg2 State2 Error2) :
    systems_isomorphic sys1 sys2 -> systems_bisimilar (discrete_sys sys1) (discrete_sys sys2).
Proof.
    intro sys_iso.
    destruct sys_iso as [f [g [is_iso_1 is_iso_2]]].
    unfold systems_bisimilar.
    exists (sm_lift_stm f), (sm_lift_stm g).
    unfold is_iso_stm.
    split;
    rewrite <- sm_lift_stm_compose;
    try rewrite is_iso_1;
    try rewrite is_iso_2;
    now rewrite sm_lift_stm_id.
Qed.
```

Listing C.31: Contract morphisms lift to system morphisms, and system morphisms lift to system trace morphisms.

```
Section LiftCMtoSM.
Context `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
        `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
        {C1 : Contract Setup1 Msg1 State1 Error1}
        {C2 : Contract Setup2 Msg2 State2 Error2}.

Definition lift_cm_to_sm (f : ContractMorphism C1 C2) :
    SystemMorphism (singleton_place_graph C1) (singleton_place_graph C2).
Proof.
    destruct f as [setup_morph msg_morph state_morph error_morph init_coherence recv_coherence].
    apply (build_system_morphism (singleton_place_graph C1) (singleton_place_graph C2)
        setup_morph msg_morph state_morph error_morph);
    unfold singleton_place_graph, singleton_ntree, sys_init, sys_receive, ntree_fold_left in *.
    -   apply init_coherence.
    -   intros.
        rewrite <- recv_coherence.
        cbn.
        now destruct (receive C1 c ctx st op_msg).
Defined.

Definition lift_ctm_to_stm (f : ContractTraceMorphism C1 C2) :
    SystemTraceMorphism
        (discrete_sys (singleton_place_graph C1))
        (discrete_sys (singleton_place_graph C2)).
Proof.
    destruct f as [ct_st_morph gen_fixp cstep_morph].
    apply (build_st_morph
        (discrete_sys (singleton_place_graph C1)) (discrete_sys (singleton_place_graph C2)) ct_st_morph);
    unfold singleton_place_graph, singleton_ntree, sys_init, sys_receive, ntree_fold_left in *.
    -   apply gen_fixp.
    -   intros * step.
        assert (ContractStep C2 (ct_st_morph sys_state1) (ct_st_morph sys_state2)
        -> SingleSystemStep (Node (Contract Setup2 Msg2 State2 Error2) C2 [])
        (ct_st_morph sys_state1) (ct_st_morph sys_state2))
        as H_step.
        {   intro cstep.
            destruct cstep as [c ctx msg nacts recv_ok].
            apply (build_sys_single_step _ _ _ c ctx msg nacts).
            unfold sys_receive.
            cbn.
            destruct (receive C2 c ctx (ct_st_morph sys_state1) msg); auto.
            now destruct t. }
        apply H_step, cstep_morph.
        clear H_step.
        destruct step as [c ctx msg nacts recv_ok].
        apply (build_contract_step C1 sys_state1 sys_state2 c ctx msg nacts).
        unfold sys_receive in recv_ok.
        cbn in *.
        destruct (receive C1 c ctx sys_state1 msg); auto.
        destruct t.
        now inversion recv_ok.
Defined.

End LiftCMtoSM.
```

Listing C.32: The identity contract morphism lifts to the identity system morphism, and compositions lift to compositions. Thus isomorphic contracts lift to isomorphic singleton contract systems under the discrete link graph.

```
(* id lifts to id *)
Lemma lift_id_cm_to_id_sm
    `{Serializable Setup} `{Serializable Msg} `{Serializable State} `{Serializable Error}
    {C : Contract Setup Msg State Error} :
    lift_cm_to_sm (id_cm C) = id_sm (singleton_place_graph C).
```

```
Proof.
    unfold lift_cm_to_sm, id_cm, id_sm, singleton_place_graph.
    cbn.
    f_equal;
    apply proof_irrelevance.
Qed.

(* compositions lift to compositions *)
Lemma lift_cm_to_sm_comp
    `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
    `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
    `{Serializable Setup3} `{Serializable Msg3} `{Serializable State3} `{Serializable Error3}
    {C1 : Contract Setup1 Msg1 State1 Error1}
    {C2 : Contract Setup2 Msg2 State2 Error2}
    {C3 : Contract Setup3 Msg3 State3 Error3}
    (f : ContractMorphism C1 C2) (g : ContractMorphism C2 C3) :
    lift_cm_to_sm (compose_cm g f) = compose_sm (lift_cm_to_sm g) (lift_cm_to_sm f).
Proof.
    destruct g as [smorph_g msgmorph_g stmorph_g errmorph_g initcoh_g recvcoh_g].
    destruct f as [smorph_f msgmorph_f stmorph_f errmorph_f initcoh_f recvcoh_f].
    unfold compose_cm, compose_sm, lift_cm_to_sm.
    cbn.
    f_equal;
    apply proof_irrelevance.
Qed.

(* isomorphic contracts => isomorphic singleton systems *)
Theorem c_iso_csys_iso
    `{Serializable Setup1} `{Serializable Msg1} `{Serializable State1} `{Serializable Error1}
    `{Serializable Setup2} `{Serializable Msg2} `{Serializable State2} `{Serializable Error2}
    {C1 : Contract Setup1 Msg1 State1 Error1}
    {C2 : Contract Setup2 Msg2 State2 Error2} :
    contracts_isomorphic C1 C2 ->
    systems_isomorphic (singleton_place_graph C1) (singleton_place_graph C2).
Proof.
    intro c_iso.
    destruct c_iso as [f [g [is_iso_1 is_iso_2]]].
    unfold systems_isomorphic.
    exists (lift_cm_to_sm f), (lift_cm_to_sm g).
    unfold is_iso_sm.
    split;
    rewrite <- lift_cm_to_sm_comp;
    try rewrite is_iso_1;
    try rewrite is_iso_2;
    now rewrite lift_id_cm_to_id_sm.
Qed.
```

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

arbitrage In the context of cryptocurrency, arbitrage refers to the practice of taking advantage of price differences between different decentralized exchanges (DEXs) or different trading pairs within the same DEX to make a profit. If a token is priced differently on different DEXs, an arbitrageur can buy the token on the DEX where it is underpriced, and then sell it on the DEX where it is overpriced, making a profit. 45, 46

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, 16, 243, 245, 247

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, 247

collateralized In DeFi, an asset is collateralized if it is backed by collateral greater than or equal to its value, often as part of a crypto lending scheme in DeFi applications. The borrower puts up the collateral as a guarantee that they will be able to repay the loan. If the borrower defaults, the lender can seize the collateral to recover their funds, often by auctioning the seized collateral. This arrangement is typically mediated by a smart contract and executed automatically. An asset can be over- or under-collateralized, indicating that the value of collateral exceeds or subceeds, respectively, the value of the collateralized asset. 47

constituent token A constituent token of a pool is a (typically non-fungible) token which can be pooled in exchange for a (typically fungible) token, called a pool token. Whether and under what conditions the pool token can be redeemed for underlying constituent tokens will vary contract by contract, depending on the tokenomics. 44, 47, 243

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, 20

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, 241

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, 20, 239, 241

Curve Curve is a decentralized exchange (DEX) on the Ethereum blockchain that specializes in stablecoins such as USDC, DAI, and USDT. Its key value proposition is low slippage on trades. 19, 46

DAI DAI is a stablecoin on the Ethereum blockchain, pegged to the value of the US dollar. It is minted as a receipt of debt, where the ETH is held in a smart contract as collateral to a loan. Interest rates on the loan fluctuate in response to market prices in order to stabilize the price of DAI. 89, 106, 240, 242

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. 241, 247

decentralized autonomous organization A decentralized autonomous organization (DAO) is a type of organization that is run using rules encoded in a smart contract, on a blockchain. DAOs are designed to operate without the need for intermediaries or a central authority. Members of a DAO participate by holding and voting with governance tokens, or tokens that represent rights or ownership in the organization, and decisions are made through a consensus mechanism such as voting or staking. DAOs are often used in DeFi as a way to create and manage decentralized investment funds, decentralized exchanges, or other types of decentralized organizations. 89, 247

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, 239, 240, 243, 245, 247

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, 20, 106, 247

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

DEX aggregator A DEX aggregator is a platform that enables users to trade cryptocurrencies across multiple decentralized exchanges (DEXs) at once, using complex algorithms to find the best prices across all supported exchanges. 106

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, 19, 22, 108, 239, 240, 242–245, 247

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. 239, 244, 247

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, 19, 242

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, 106

fractionalize In the context of non-fungible tokens (NFTs), "fractionalizing" refers to the process of dividing ownership of a single NFT into smaller, tradeable fractions or shares, typically as fungible tokens. 38

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. 36

fungible See fungible token. 38, 239, 243

fungible token A fungible token is a type of digital asset that represents a unit of value that is interchangeable with other units of the same value. This means that each unit of the token is identical and interchangeable with any other unit. Examples of fungible tokens include the cryptocurrencies BTC or ETH, stablecoins USDC or DAI, or other tokens conforming to the ERC20 token standard. 242, 247

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. 20, 67, 105, 241

governance token A governance token is a type of token used to facilitate decision-making in a decentralized organization, such as a DAO. Holders of governance tokens can be given voting rights that allow them to participate in decisions relevant to the organization, such as changes to the organization's protocol or the allocation of funds. Governance tokens can also be used to propose and vote on changes to the organization's governance structure, as well as to elect or recall members of the organization's leadership. 89, 106, 240

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, 66, 242, 243

LP token An LP token, short for liquidity provider token, is a digital asset issued to liquidity providers in DeFi platforms, representing their share in a liquidity pool and enabling them to manage and withdraw their deposited assets and rewards. 47, 106, 243

non-fungible See non-fungible token. 239, 243

non-fungible token A non-fungible token (NFT) is a unique and indivisible digital asset built on blockchain, often representing ownership of distinct items like digital art, collectibles, or virtual real estate. NFTs are associated with the ERC721 token standard on Ethereum. 37, 242, 243, 247

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. 106, 240, 244

pool In DeFi, a pool is a shared aggregation of assets that provides liquidity for a specific asset or market. Users can add their own assets to the pool and receive a proportional share of the pool's tokens, called LP tokens or pool tokens. These tokens give users a share of the fees generated by the trades that occur in the pool. 239, 243

pool token A pool token is a (usually fungible) token which can be minted in exchange for pooling a (usually non-fungible) constituent token into a pool. The terms governing when and under what conditions the pool token can be redeemed for underlying constituent tokens are set by the contract governing the pool. 38, 47, 239, 243

price oracle In DeFi, a price oracle is a source of truth for the current price of a cryptocurrency or other asset. A price oracle is typically a smart contract to which information about the price of an asset from an external data source, such as an exchange, is pushed, and provides it to other smart contracts in a standardized format. In order for a price oracle to be effective, it must be reliable and resistant to tampering and manipulation. 19, 106

slippage In the context of decentralized exchanges (DEXs) and automated market makers (AMMs), slippage refers to the difference between the expected price of an asset and the actual price at which it is traded, which can differ due to the changing market conditions such as an large trades. This is particularly relevant in DeFi, where price slippage can have a significant impact on the profitability of trading strategies such as arbitrage. 240

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

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. 22, 241

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, 20, 36, 89, 106, 240, 244, 245

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, 106

Tezos Tezos is a third-generation, proof-of-stake blockchain which supports smart contracts written in Michelson. Its native token is XTZ. 16, 22, 244, 247

token A token is a digital asset that represents a unit of value and can be traded on a blockchain platform. Tokens can serve a variety of purposes, such as representing ownership in a company, access to a particular product or service, or as a medium of exchange. Tokens can be created using smart contracts on blockchain platforms, such as Ethereum, and they are typically stored and traded using a digital wallet. Tokens are standardized on most blockchains, for example the ERC20 and ERC721 standards on the Ethereum blockchain, and the FA2 standard on the Tezos blockchain. 89, 106, 107, 242, 244, 247

tokenization Tokenization is the process of converting ownership rights or assets, external to the blockchain, into a blockchain-based digital representation such as a token. 244

tokenize See tokenization. 244

tokenized carbon credit Tokenized carbon credits are tokens, stored on a blockchain, that represent a carbon credit, typically a unit of carbon emissions reduction. Most are tokenized from legacy organizations that verify carbon emissions reductions such as Verra. 37, 38

tokenomics Tokenomics, a portmanteau of “token” and “economics,” refers to the study and design of the economic system and incentives behind a cryptocurrency or digital token. It encompasses the rules, policies, and mechanisms that determine the creation, distribution, usage, and value of the tokens within a blockchain or decentralized ecosystem. Tokenomics plays a crucial role in determining the success and sustainability of a cryptocurrency or digital token project. 36, 106, 239

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, 245

Uniswap Uniswap is a decentralized exchange (DEX) built on the Ethereum blockchain. It was the first, and is arguably the most popular, automated market maker (AMM). 43, 44, 46, 66

USDC USDC (USD Coin) is a stablecoin pegged to the value of the US dollar. 19, 240, 242

USDT Tether, or USDT, is a stablecoin pegged to the value of the US dollar. 19, 240

vault A vault is a secure digital container or smart contract function designed to hold and manage digital assets, such as cryptocurrencies or tokens, in a decentralized and tamper-resistant manner. Vaults are often utilized in DeFi protocols to facilitate various financial operations like lending, borrowing, and yield farming. They are programmed with specific rules and conditions governing the access, withdrawal, and management of the assets they hold, and they aim to provide a high level of security and transparency. Vaults can have different strategies and mechanisms to optimize asset utilization, protect against risks, and maximize returns for users within the blockchain ecosystem. 19

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, 239

yield aggregator A yield aggregator is a type of smart contract that allows users to automatically aggregate their crypto assets into various yield-generating DeFi protocols. The purpose of a yield aggregator is to maximize the yield received by the user on their cryptocurrency investments. Yield aggregators typically work by automatically re-allocating funds to the highest yielding protocols, taking into account various factors such as fees, liquidity, and performance. 19, 20, 106

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, 20, 106, 245

Acronyms

AMM Automated market maker. 15, 16, 19, 20, 36–38, 43, 44, 46, 47, 66, 106, 117, 118, 239, 241

BSC Binance Smart Chain. 18

DAO decentralized autonomous organization. 89, 106, 242

dApp decentralized application. 241

DeFi decentralized finance. 18, 20, 37, 43, 106, 107, 118, 133, 239, 240, 242, 243, 245

DEX Decentralized exchange. 20, 106, 107, 118, 239–241

ERC20 A token standard for fungible tokens on the Ethereum blockchain. 38, 242, 244

ERC721 A token standard for non-fungible tokens on the Ethereum blockchain. 243, 244

ETH The native token of the Ethereum blockchain. 20, 240–242

EVM Ethereum Virtual Machine. 22, 239

FA2 A generic standard on the Tezos blockchain for fungible and non-fungible tokens. 244

NFT See non-fungible token. 37, 38, 242

XTZ The native token of the Tezos blockchain. 244