NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

JOÃO CARLOS RAPOSO DOS REIS

Bachelor in Computer Science

TOWARDS A SOLIDER SOLIDITY

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon

December, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

TOWARDS A SOLIDER SOLIDITY

JOÃO CARLOS RAPOSO DOS REIS

Bachelor in Computer Science

Adviser: António Ravara
Associate Professor, NOVA University Lisbon

Co-adviser: Mário Pereira
Assistant Professor, NOVA University Lisbon

Examination Committee

Chair: Nuno Preguiça
Full Professor, NOVA University Lisbon

Rapporteur: Simão Melo de Sousa
Full Professor, University of Algarve

Adviser: António Ravara
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
December, 2023

Towards a Solider Solidity

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.0.3) [**novathesis-manual**].

Abstract

Blockchains are being widely adopted since its inception. Since Ethereum popularized smart contracts and introduced its platform for developing and deploying these programs on the blockchain, there has been an unprecedented boom in decentralized applications (dApps). Smart contracts bring a new flexibility which we didn't see in the beginning with the first generation blockchains. These programs rule important agreements between two parties involving transacting valuable assets between them. Recently we have seen a record of stealing assets from the blockchain , as they exploded and became more popular in 2020. Many known vulnerabilities include: Re-entrancy, Gasless Send, Phishing with tx.origin and Type Casts.

This reinforces the importance to ensure that these programs are correct. However this is still a hard task, as the programming languages used to developed them often do not really help and smart contracts are still being exploited. Static analysis tools can prevent these bugs, if they are robust enough to detect most of the vulnerabilities which are known and not known, and ensure that these contracts are developed with the best design practices. Many of those existing tools and programming languages are still in a very early stage and not yet prepared to be used in production environments.

The development of smart contracts can be done in two types of languages: domain-specific and general-purpose languages. Domain-specific languages, specifically designed for writing smart contracts, are generally considered safer than their general-purpose counterparts. Solidity, created by the Ethereum Foundation, is a DSL and stands out as the most popular language to date. However, it is not without its flaws, primarily stemming from its complexity and Turing completeness. While recent academic efforts have introduced languages aimed at addressing known vulnerabilities in Solidity, the language's widespread adoption needs continuous improvements to improve its safety.

Our contribution targets a vulnerability in Solidity related to Type Casts. We introduce a proof-of-concept language inspired by Featherweight Solidity, a formalization of Solidity as a subset with an improved type system. Specifically, our language features an extended typing address to mitigate this vulnerability.

Resumo

As blockchains ganharam ampla adoção desde sua criação. Desde que a Ethereum popularizou os smart contracts e introduziu sua plataforma para desenvolver e implantar esses programas na blockchain, houve um crescimento sem precedentes nas aplicações descentralizadas (dApps). Smart contracts trazem uma nova flexibilidade que não vimos no início com as blockchains de primeira geração. Esses programas regem acordos importantes entre duas partes envolvendo a transação de ativos valiosos entre elas. Recentemente, observamos um aumento no roubo de ativos da blockchain. Muitas vulnerabilidades conhecidas incluem Re-entrancy, Gasless Send, Phishing com tx.origin e Type Casts.

Isso reforça a importância de garantir que esses programas estejam corretos. No entanto, ainda é uma tarefa difícil, pois as linguagens de programação usadas para desenvolvê-los muitas vezes não ajudam realmente, e smart contracts continuam sendo explorados. Ferramentas de análise estática podem prevenir esses bugs se forem robustas o suficiente para detectar a maioria das vulnerabilidades conhecidas e desconhecidas, e garantir que esses contratos sejam desenvolvidos com as melhores práticas de design. Muitas dessas ferramentas e linguagens de programação existentes ainda estão numa fase inicial e ainda não estão preparadas para serem usadas em ambientes de produção.

O desenvolvimento de smart contracts pode ser feito em dois tipos de linguagens: linguagens de domínio específico e linguagens de propósito geral. As primeiras são desenvolvidas especificamente para escrever smart contracts, sendo geralmente consideradas mais seguras do as segundas. O Solidity, criado pela Ethereum Foundation, destaca-se como a linguagem mais popular até o momento. No entanto, não está isento de falhas, principalmente devido à sua complexidade e ser Turing complete. Recentemente têm surgido linguagens académicas a abordar vulnerabilidades conhecidas no Solidity. Contudo a já ampla adoção da linguagem aprimorar sua segurança nas áreas em que atualmente apresenta deficiências.

A nossa contribuição visa uma vulnerabilidade no Solidity: Type Casts. Introduzimos uma linguagem de prova de conceito inspirada no Featherweight Solidity, uma formalização do Solidity como um subconjunto com um sistema de tipos estendido. Especificamente, a nossa linguagem apresenta um tipo address estendido mitigando essa vulnerabilidade.

Contents

1 Introduction

1.1 Motivation

Blockchains are being widely adopted since its inception. Since Ethereum popularized smart contracts and introduced its platform for developing and deploying these programs on the blockchain, there has been an unprecedented boom in decentralized applications (dApps). Smart contracts bring a new flexibility which we did not see in the beginning with the first generation blockchains. These programs rule important agreements between two parties involving transacting valuable assets between them. It is therefore important to ensure that these programs are correct. However this is still a hard task, as the programming languages used to developed them often do not really help and smart contracts are still being exploited. Static analysis tools can prevent these bugs, if they are robust enough to detect most of the vulnerabilities which are known and not known, and ensure that these contracts are developed with the best design practices. Many of those existing tools and programming languages are still in a very early stage and not yet prepared to be used in production environments.

1.2 The Problem

Since smart contracts once deployed cannot be modified, unless they use proxy contracts, which are contracts that can be upgraded, there is an urge that these programs are auditable. Being auditable, means that they are certified to do whichever features and only these features they are intended to. The main job of an auditor is to find any errors or vulnerabilities that could cause a loss of funds and/or control of the project. Static analysis can then help them to do this important task.

Recently we have seen a record of stealing assets from the blockchain , as they exploded and became more popular in 2020. One example occurred in March 2022, when Ronin said that hackers stole cryptocurrency worth almost 615 milion of dollars from its systems. At the time, this was a record-making in smart contracts history. There is a good article explaining what was the purposes of this project and how it got exploited by the hackers.

Understanding the underlying language we are studying is crucial when building tools to deal with vulnerabilities. To improve the safety of a programming language, there are two possible approaches: designing a language to be proof against vulnerabilities or developing static analysis tools to detect them. We have encountered both approaches in either academic or industrial contexts and we will delve deeper into them in Chapter 3.

1.3 Objectives

The goal of the dissertation is, in a nutshell, to develop solutions to deal with presently undetected vulnerabilities in Solidity. To achieve this, our initial step involved an in-depth analysis of the state-of-the-art landscape, encompassing programming languages designed for smart contract development, existing vulnerabilities within Solidity, and the corresponding static analysis tools employed for their detection. One of our objectives is to discern the efficacy of domain-specific languages versus general-purpose languages for smart contract development. Additionally, we seek to identify vulnerabilities within Solidity that may not have received adequate attention. Our investigation revealed that Type Casts is a vulnerability that has been relatively overlooked. This vulnerability is related to the weakness in the typing of contract addresses, which is the most common way to refer to them. When casting an address to a contract instance and calling a function of it, Solidity's compiler does not verify whether such a function is defined in the target contract. When we try to execute it and it does not exist, an exception is thrown, resulting in users losing funds without the possibility of reimbursement. However, with a more refined address type, we could prevent the function from being called, thus safeguarding the funds being sent. Consequently, we have undertaken the development of an environment where this particular vulnerability is mitigated.

1.4 Contributions

There have been some formalizations on Solidity semantics and type system [29, 14]. However, being a relatively new language, formalizations are currently dispersed and focus on different aspects. In this master thesis we gathered all together, introducing many concepts on the one we are extending: Featherweight Solidity [21, 6].

We present herein a proof-of-concept: a refined implementation and formalization of Solidity. We introduce Featherweight Solidity 2.0 as a Solidity subset with a new type system. This new type system addresses what it was called as Type Casts vulnerability, where there is currently no way to have subtyping in address type. Therefore our language supports an extended typing address, aiming to fix this problem. Beyond formalizing it, we also leveraged the expressiveness power of OCaml, a functional programming language known for its strong type inference and formal verification capabilities, to implement Featherweight Solidity as a proof-of-concept compiler.[1]

[1] https://github.com/jcrreis/featherweight-solidity

1.5 Organization of this document

The rest of this document is structured as follows:

- Chapter 2 - Foundational concepts of blockchain and smart contracts.
- Chapter 3 - A study of the state of the art in smart contracts, covering academic languages, vulnerabilities and static analysis tools in Solidity.
- Chapter 4 - Introduction to Featherweight Solidity, a calculus language that models the core features of Solidity.
- Chapter 5 - Presentation of our revised operational semantics for FS 2.0.
- Chapter 6 - Demonstration of how we use property-based testing to prove properties of our operational semantics.
- Chapter 7 - Presentation of our revised type system rules for FS 2.0.
- Chapter 8 - Illustration of our work using a complex case study.
- Chapter 9 - Summary of our work and suggestions for future development.

2

Background

2.1 General Context

Blockchain has been one of the hottest and promising technologies of the decade. In 2008, Satoshi Nakamoto (an alias used by the creator/creators) released a paper in which he explained a new form of digital payment without relying in a trusted third party, using this technology [19]. This was a big event as this was the first time someone did conceptualize a **decentralized blockchain** effectively, a new kind of distributed system. Later on, in 2014 a new decentralized blockchain surged: Ethereum [2]. This new architecture popularized an old concept in computer science: **smart contracts**. They were first proposed by Nick Szabo in 1996 who coined the term, using it to refer to "a set of promises, specified in digital form, including protocols within which the parties perform on these promises" [24]. A smart contract is simply a computer program that run on a blockchain and allow to automatically execute, control or document legally relevant events and actions between two parties.

2.2 Distributed System

2.2.1 General Context

A distributed system is a set of hardware and software components interconnected through a communications infrastructure, which cooperate and coordinate with each other only by exchanging messages, for the execution of distributed applications [25]. In a distributed system, data is replicated through many equivalent nodes that can be replaced by another if one fails, improving performance and reliability of the system. Nevertheless, having multiple copies of the same data in many nodes can lead to consistency problems and this systems need to guarantee that its users always view a single coherent system, requiring that all nodes always have consistent replica of the system.

Consistency in distributed systems means that every node has the same view of data at a given point in time regardless of whichever client has updated the data [25]. When you make a request to any node, you receive the exact same response so that it looks

like there is only a single node performing all the operations. There are many types of consistency models in distributed systems, where the extremes are **strong consistency** and **weak consistency** [4]. Blockchains by their nature, being an immutable distributed ledger that often stores and transacts valuable assets, need to have a strong consistency model. This consistency model requires that all accesses are seen by all nodes in the same order and there are one and only one consistent state. On the other hand, weak consistency does not requires that all accesses are seen in the same order and there are no guarantees that all nodes have the same data at any times. Strong consistency requires a much stronger synchronization between nodes and for this reason it is usually slower to accomplish than weak consistency.

2.2.2 Consensus Protocols

To achieve consistency, distributed systems need a **consensus protocols**, which are the basis for the state machine replication approach to distributed computing [17]. However, reaching consensus on distributed networks, in a safe and efficient way, is far from being an easy task. If some nodes fail or act maliciously the whole distributed system is compromised. This is known as the Byzantine Generals problem (or Byzantine fault). The Byzantine Generals problem[1] was conceived in 1982 and is a condition of a distributed computer system, where components may fail and there is imperfect information on whether a component has failed [16]. A system that is able to resist the class of failures derived from the Byzantine Generals' Problem, has a property called **Byzantine Fault Tolerant** (BFT). A BFT system is able to continue operating even if some of the nodes fail or act maliciously.

One of the first algorithms proved to be correct to achieve consensus in a distributed system was Paxos [15]. This algorithm is used to achieve consensus among a distributed set of computers that communicate via an asynchronous network. One or more clients proposes a value to Paxos and we have consensus when a majority of systems running Paxos agrees on one of the proposed values. Even today Paxos is widely used by many big cloud computing companies such as Amazon, Google and Microsft.

2.3 Blockchain

Blockchain is a technology that aims to allow digital information to be recorded and distributed, but not edited. As a distributed ledger technology, the blockchain is intentionally designed to be highly resistant to modification and frauds (such as double-spending) [27].

[1]This analogy behind the Byzantine Generals Problem is that several generals are besieging Byzantium and they have surrounded the city, but they must collectively decide when to attack. If all generals attack at the same time, they will win, but if they attack at different times, they will lose. The generals have no secure communication channels with one another because any messages they send or receive may have been intercepted or deceptively sent by Byzantium's defenders [28].

A distributed ledger is a database that is consensually and synchronized shared between multiple nodes, any changes made to the ledger are reflected and copied to all participants, which check the validity of the operation. Every node in a distributed ledger processes and validates every record.

To understand better how does blockchains work we need to know what is a block, a chain and a consensus protocol. Next, I will present these concepts and explain what role each one have in this architecture.

2.3.1 Block

Blocks are data structures that store immutable data. Each block must store a pointer to the **hash of the previous block**, a **timestamp** representing the time when a block has been validated by the network and more data that varies depending on the blockchain [5]. This blocks are permanently stored in a distributed ledger, meaning that a blockchain only grows and there are no delete operations. Although this is important to secure blockchains, this also is a problem because blockchain grow unboundly over time. To overcome this problem, blockchains in general use Merkle trees [18]. A Merkle tree is a data structure, that is a generalisation of hash lists and hash chains, where every leaf node is labelled with the hash of a data block and every non-leaf node is labelled with the cryptographic hashes of the labels of its child nodes. This data structures allow efficient and secure verification of the contents of large data structures, without relying in the actual data.

2.3.2 Chain

A chain is a set of interconnected ordered blocks, formed through the pointer to the hash of the previous block that each block has. This mechanism turns out to be similar to linked lists but instead of pointing to the next block, each block points to the previous one [5]. Adding a new block to a chain is a transactional operation: either a block is committed, if they are validated by nodes participating in the consensus protocol or the transaction is aborted if a block is malicious. As hashes are cryptographically derived from the block data, collisions are very unlikely and one change in any block will invalidate all the subsequent blocks in the chain, because all their hashes will also change.

2.3.3 Ledger

A ledger is a database with the purpose of recording contracts and transactions. Centralized ledgers are used by companies to contain all the accounts for recording transactions relating to a company's assets, liabilities, owners' equity, revenue, and expenses. A distributed ledger is a database that is synchronized and accessible across different sites and geographies by multiple participants. This type of ledgers are used by blockchains and it is a decentralized form of a ledger. Distributed ledgers are more secure and reduce cyber

attacks and financial fraud because it does not have a central point of failure and are better able to withstand malicious attacks.

2.3.4 Consensus Protocols

Blockchains are intentionally designed to be decentralized, working as a digital ledger that is maintained by a distributed network of computer nodes. For this reason, blockchain technology allowed the creation of trustless economic systems, where transparent and reliable financial transactions could be executed without the need for intermediaries. Just as most distributed systems, the participants of a cryptocurrency network need to regularly agree on the current state of the blockchain, known as **consensus achievement**.

As blockchains are decentralized distributed systems used to transfer valuable digital assets, they need to be **BFT**. To be a BFT, a system needs to embed a **consensus protocol**. However if a node or group of nodes control over 50% of a blockchain hashing power they can change the order of blocks, reverse past approved blocks (which would then lead to a double-spending problem) and prevent other nodes from mining new blocks. This is known as the 51% attack [9]. Blockchain's consensus protocol discourage attackers from doing this attack by making it cost inefficient.

Although there are many consensus protocols currently used in blockchains, the two most popular ones are **Proof-of-Work** (PoW) and **Proof-of-Stake** (PoS), which I will explain next.

2.3.4.1 Proof-of-Work

This concept was invented in 1993 by computer scientists Cynthia Dwork and Moni Naor to combat e-mail spamming [8]. Eventually in 2009 when Bitcoin was released it was adopted as its consensus protocol. Proof-of-Work is a system where many miners compete with each other to solve a very complex puzzle, but easy to verify. The first miner solving this puzzle will then broadcast this block to the network and other miners will verify that the solution is correct. If the solution is approved by other nodes, the miner which solved the puzzle will earn a reward. Although this ensures that it is extremely costly to control the majority of the network, preventing a 51% attack, as it would require a lot of computational power to do it and therefore making the attack inefficient, this could also have an impact on environment, if the energy used by miners are from fossil fuels. This protocol is currently used in blockchains like Bitcoin and Ethereum.

2.3.4.2 Proof-of-Stake

This protocol surged in 2012, when Peercoin was launched and used both Proof-of-Work and Proof-of-Stake for its consensus protocol [23]. Therefore it was adopted by many other blockchains as their consensus protocol. Cardano, Solana and Polkadot are examples of 3rd generation blockchains that adopted a fully PoS consensus algorithm and Ethereum

is planning switching from PoW to PoS in the first quarter of 2022. This algorithm uses validators instead of miners. Validators are coin owners that simply deposit (or stake) an amount of coins into the network. They are selected randomly by the algorithm to verify new block. This action is called minting, which is equivalent to mining in PoW. This is one of the differences from PoW, instead of being a competition-based algorithm, it chooses validators randomly, with validators with higher amount of coins staked having an higher probability of being choose. In order to perform a 51% attack a validator needs to own more than 50% of the token supply, meaning that if Bitcoin used PoS an attacker would need 383 billion euros to perform the attack. Also, Proof-of-Stake requires much less computational power and consequently have a much lesser environment footprint than Proof-of-Work.

2.3.5 Blockchain Types

Blockchain types can be characterized as permissionless, permissioned or both. A permissionless blockchain allow any user to join the network and don't restrict the rights of the nodes. Conversely, a permissioned blockchain can restrict access to the network to certain nodes and can also restrict the rights of those nodes on the network. There are 4 types of blockchain: **Public Blockchains**, **Private Blockchains**, **Consortium Blockchains** and **Hybrid blockchains** [20].

2.3.5.1 Public Blockchains

Public blockchains are permissionless, allowing anyone to join and are completely decentralized. They allow all nodes of the blockchain to have equal rights to access blockchain, create and validate new blocks. Some examples of public Blockchains are Bitcoin and Ethereum.

2.3.5.2 Private Blockchains

Private blockchains are permissioned, controlled by a single organization. The central authority controls the rights of each node to perform function. This type of blockchains are only partially decentralized, because public access to these networks are restricted. The most prominent private blockchain is HyperLedger Fabric.

Both private and public blockchains have drawbacks, with public blockchains tending to have longer validation times and private blockchains being more vulnerable to fraud and bad actors. The next two types of blockchain, address these drawbacks.

2.3.5.3 Consortium Blockchains

Consortium blockchains are permissioned blockchains governed by a group of organizations rather than a single one. Therefore this blockchains are more decentralized than private ones, resulting in higher levels of security. Global Shipping Business Network

(GSBN), a blockchain built by CargoSmart, is an example of a consortium blockchain, aiming to digitalize the shipping industry and allow maritime industry operators to work more collaboratively.

2.3.5.4 Hybrid blockchains

An Hybrid blockchain is a blockchain that is controlled by a single organization, but with a level of supervision performed by the public blockchain, which is requires to perform certain transaction validations. An example of a hybrid blockchain is IBM Food Trust, which aims to improve efficiency throughout the whole food supply chain

2.3.6 Examples of Blockchain

2.3.6.1 Bitcoin

Bitcoin, as I already mentioned in this article, was the first decentralized blockchain released. Launched in early 2009, Bitcoin bring to the internet a new revolutionary payment process, fully decentralized and without a central authority, meaning that it does not rely in a trusted third party, like Visa, MasterCard or PayPal. Everyone can create a bitcoin address (equivalent to a bank account in traditional financial system) without needing an approve, send transactions without approve (only miners need to confirm validity of transactions) and become a miner. However bitcoin transactions still take 10 or more minutes to be processed, making them much slower than transactions made by other centralized protocols. Because of this many question if it is a viable alternative to them and many believe that Bitcoin could just become a digital store of value, a digital alternative to gold.

2.3.6.2 Ethereum

Ethereum was launched in 2015 and brought with it the ability to create applications that run on the blockchain. The main difference from Bitcoin is that Ethereum is more than just a cryptocurrency, it is a platform. It allows developers to deploy smart contracts and decentralized apps (DApps), these run on a fully decentralized blockchain, without any downtime or interference from a third party and is powered by its own native cryptocurrency: Ether (ETH). Nevertheless, as each block in a blockchain is immutable, unlike traditional computer programs, these applications once deployed cannot be modified and redeployed, meaning that if a security or safety breach is detected it cannot be corrected, unless owners deploy it to a new address.

2.3.6.3 HyperLedger Fabric

HyperLedger Fabric is an open-source platform managed by Linux Foundation designed to create business-to-business (B2B) businesses and cross-industry applications. As we already have seen, it is a private blockchain with a modular architecture that delivers high

degrees of confidentiality, flexibility, resiliency, and scalability. Unlike Ethereum, where all transactions are public, in HyperLedger Fabric access is limited. This platform also supports smart contracts, they are packaged in a chaincode and then can be deployed into blockchain. It also has a flexible consensus protocol, that can be configurable by developers using well-established toolkits for Crash Fault-Tolerant (CFT) or Byzantine Fault-Tolerant (BFT) ordering.

2.3.6.4 Ripple

Ripple is a blockchain created by three former bitcoin developers. Their goal was to create a more sustainable and faster consensus algorithm than PoW, to make transactions faster and more efficient than Bitcoin. This protocol uses XRP Ledger (XRPL) as its consensus protocol, in which designated servers (or validators) come to an agreement on the order and outcome of transactions made in Ripple network. All validators process transactions following the same rules and transactions are public, with strong cryptography to guarantee the integrity of the system. At the time of writing there are 156 validators in the network but the consensus protocol ensures more decentralization over time as the validator pool grows. Unlike bitcoin which can only process 7 per second, Ripple can process more than 1500 transactions per second and its goal is to create a SWIFT 2.0.

2.4 Smart Contracts

2.4.1 General Context

Smart contracts aim to automatize the execution of business processes, removing arbitrariness that humans can introduce. Typically every blockchain node have a virtual machine, which is a computation engine which acts like a decentralized computer that runs programs, smart contracts, just like Java Virtual Machine (JVM) does and maintain consensus across the network. This virtual machine is very import to support smart contracts as they interpret these programs on a lower level basis, often an assembly-like language or a bytecode.

There are many examples of these type of virtual machines being Ethereum Virtual Machine (EVM) the most famous one. This virtual machine executes as a stack machine with a depth of 1024 items, where each item is a 256-bit word, which was chosen for the ease of use with 256-bit cryptography, normally Keccak-256. Smart contract's bytecode are then executed as a number of EVM operation codes (opcodes), which perform standard stack operations like XOR, AND, ADD, SUB, etc. or blockchain-specific stack operations, such as ADDRESS, BALANCE, BLOCKHASH, etc. On other hand Algorand Virtual Machine (AVM) interpets an assembly-like language, TEAL. TEAL programs are comprised of a set of opcodes,which are used to implement the logic of smart contracts and smart signatures. Examples of opcodes interpreted by AVM can be found here.

Although bitcoin support smart contracts, through BitML, they are less flexible than in newer architectures like Ethereum, Tezos or Algorand. In Bitcoin transactions can specify simple conditions on how to redeem them, using a limited set of logic, arithmetic and cryptographic operators. Despite the limited expressiveness of these conditions, it is possible to encode a variety of smart contracts (e.g. gambling games, escrow services, crowdfunding systems), by suitably chaining transactions.

The most popular programming languages are general-purpose programming languages, because they are used for many areas of research in computer science. If we look into Tiobe, an index where ratings are based on the number of skilled engineers world-wide, courses and third party vendors, that uses popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu to calculate the ratings, we can see that Python, C and Java are the most famous programming languages by these metrics. This suggests that these languages boast a thriving developer community with abundant online resources, facilitating newer developers with a lot of free and paid content to learn more about them. This can lead to an increase in popularity of a blockchain that chooses to use one of these languages for smart contract programming.

When you think of languages to write smart contracts you can either choose general-purpose programming languages or domain-specific programming languages. The former are designed to be used for building software in a wide variety of application domains. The latter are only designed to be used for building software in a specific application domain, in this case they are designed to write smart contracts. However digital contracts need to protect valuable assets and general-purpose programming languages don't provide any means to help developers making them more secure and safer. In 2018 Vitalik Buterin, co-founder of Ethereum blockchain, published a tweet where he shares an opinion that unlike regular coding smart contracts need to focus on 4 properties [1]:

- Very small code size - This makes smart contracts less prone to bugs and more intelligible.
- Much higher focus on safety - Smart contracts receive, store and transfers valuable user assets and they need to have safe operations.
- Much higher focus on auditability (misleading code very bad) - In smart contracts code dictates what a participant can do or cannot do ("CODE IS LAW").
- Perfect determinism - Code in smart contracts cannot have an unpredictable behaviour, like we have in C/C++ when for instance you try to access a position out of an array length, called undefined behaviour.

For these reasons, many organisations managing blockchains have developed domain-specific programming languages. However these languages are still very new, many times suffering from several vulnerabilities and need to be developed and improved. To summarize and answer our initial question, **Domain-specific programming languages**

are better to write smart contracts but, as they are new programming languages, they are still susceptible to many vulnerabilities and need to be improved.

Even though Vitalik Buterin stated that general-purpose programming languages are worse than domain-specific languages to write smart contracts, there are a lot of blockchains which allow developers to write smart contracts in one of these languages.

Next I will present the most popular languages in the industry to write smart contracts, including some general-purpose ones.

2.4.2 Brief Presentation of Industry Languages

2.4.2.1 Solidity

Ethereum's official programming language to build and deploy smart contracts is Solidity. This programming language is compatible with every blockchain protocol that runs on Ethereum Virtual Machine (EVM). Polygon and Binance Smart Chain are two examples of protocols that support EVM and consequently Solidity. Solidity is an object-oriented, high-level language for implementing smart contracts, created and desgined by the Ethereum Foundation to target the EVM. It is a strongly typed and Turing-complete language, inspired by C++, Python and JavaScript, supporting inheritance, libraries and complex user-defined types (structs). It also is the most popular programming language to write smart contracts. This popularity combined with its expressiveness made the language a target for exploits, with several vulnerabilities and bugs discovered over the last years.

2.4.2.2 C++

C++ is a low level language, that delegates into developers memory management. it is supported and the native programming language in EOSIO blockchain to write smart contracts. Solana also supports C++ for smart contract development, but they recommend developers using Rust instead to have a better development experience. As we have already seen C++ is not a perfect deterministic programming language, many operations are prescribed to be unpredictable and as one of the lowest level languages, contracts written in this language are much more prone to have errors, bugs or vulnerabilities.

If you have some curiosity to see how smart contracts are written in C++, here is a repository with some examples of smart contracts for EOSIO blockchain.

```
1  void token::transfer( const name&    from,
2                        const name&    to,
3                        const asset&   quantity,
4                        const string&  memo )
5  {
6      check( from != to, "cannot transfer to self" );
7      require_auth( from );
8      check( is_account( to ), "to account does not exist");
9      auto sym = quantity.symbol.code();
10     stats statstable( get_self(), sym.raw() );
```

```
11      const auto& st = statstable.get( sym.raw() );
12
13      require_recipient( from );
14      require_recipient( to );
15
16      check( quantity.is_valid(), "invalid quantity" );
17      check( quantity.amount > 0, "must transfer positive quantity" );
18      check( quantity.symbol == st.supply.symbol, "symbol precision mismatch" );
19      check( memo.size() <= 256, "memo has more than 256 bytes" );
20
21      auto payer = has_auth( to ) ? to : from;
22
23      sub_balance( from, quantity );
24      add_balance( to, quantity, payer );
25  }
```

This is a function implementing a token transfer in C++ for EOSIO blockchain. As we can see, this approach brings a more confusing and harder to interpret code, making it less auditable. The **name** data type, represents an address and this data type wraps an uint64_t to ensure it is only passed to methods that expect a name.

2.4.2.3 Python

Algorand is the most popular blockchain project that uses Python as the primary language to write smart contracts. To write smart contracts for Algorand you will need to use PyTeal library that binds python code into TEAL code, an assembly like language used to express contracts in Algorand. This blockchain uses pure Proof-of-Stake (PPoS) consensus protocol. The main edge over Ethereum is the incredible transaction processing, it can process 1000 transactions per second. Another way to write smart contracts in Python is using SmartPY, an intuitive platform to write smart contracts for Tezos. However, Tezos uses Michelson as their native supported language for smart contract development.

When you look at PyTeal documentation, they note that **PyTeal hasn't been security audited.**

In the following url you can see some examples of smart contracts written using PyTeal for Algorand Virtual Machine.

```
1  alice = Addr("6ZHGHH5Z5CTPCF5WCESXMGRSVK7QJETR63M3NY5FJCUYDHO57VTCMJOBGY")
2  bob = Addr("7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXTY3WTNP5NU2MHBW27M")
3  secret = Bytes("base32", "2323232323232323")
4  timeout = 3000
5
6  def htlc(
7      tmpl_seller=alice,
8      tmpl_buyer=bob,
9      tmpl_fee=1000,
10     tmpl_secret=secret,
11     tmpl_hash_fn=Sha256,
```

```
    tmpl_timeout=timeout,
):

    fee_cond = Txn.fee() < Int(tmpl_fee)
    safety_cond = And(
        Txn.type_enum() == TxnType.Payment,
        Txn.close_remainder_to() == Global.zero_address(),
        Txn.rekey_to() == Global.zero_address(),
    )

    recv_cond = And(Txn.receiver() == tmpl_seller, tmpl_hash_fn(Arg(0)) ==
        tmpl_secret)

    esc_cond = And(Txn.receiver() == tmpl_buyer, Txn.first_valid() >
        Int(tmpl_timeout))

    return And(fee_cond, safety_cond, Or(recv_cond, esc_cond))
```

Python brings a more readable approach, as its syntax is more concise than C++. However, it is a dynamic data type language, which makes it prone to errors and not type safe for this purpose. Addresses are treated with a special class **Addr** offered by this library.

2.4.2.4 Plutus

Cardano uses Plutus as the native language for smart contract development. Plutus is a Turing-complete language and smart contracts written in Plutus are effectively Haskell programs. Haskell is a statically-typed, purely functional programming language with type inference and lazy evaluation, very popular in academic world.

You can find examples of smart contracts written with Plutus for the Cardano blockchain in the following repository.

```
(tid1, tid2)  <- startExampleCampaign 700 700
tick 20
void $ addTx $ Tx
    { _txInputs    = [Input (optr tid1 0) unit]
    , _txOutputs   = [Output "Bob" (fromAda 700) unit]
    , _txSignees   = ["Bob"]
    , _txSlotRange = SlotRange 20 Forever
    , _txForge     = mempty
    }
void $ addTx $ Tx
    { _txInputs    = [Input (optr tid2 0) unit]
    , _txOutputs   = [Output "Charlie" (fromAda 700) unit]
    , _txSignees   = ["Charlie"]
    , _txSlotRange = SlotRange 20 Forever
    , _txForge     = mempty
    }
```

In the snippet above, these two procedures transfer 700 ADA from a certain address to another target address, using Plutus a subset of Haskell. This target address receives a total amount of 1400 ADA. As we can see Haskell brings simplicity and an easy to read code. However developers coming from Orient-Object programming languages or non-developers might struggle to understand what each line of code does. It is required that auditors have some previous Haskell knowledge, to have a more accurate audit.

2.4.2.5 Rust

Rust is a low-level statically-typed programming language that is fast and memory-efficient. Solana's smart contracts are fully written in Rust, while Polkadot has a framework built on top of Rust called Substrate. Near protocol also uses Rust for the purpose of smart contract development.

Here is a tutorial teaching how to handle the creation and ownership of non-fungible tokens (NFTs) in the Substrate framework.

```
fn transfer_from_to(&mut self, from: AccountId, to: AccountId, value:
    Balance) -> bool {
    let from_balance = self.balance_of_or_zero(&from);
    if from_balance < value {
        return false
    }

    // Update the sender's balance.
    self.balances.insert(from, from_balance - value);

    // Update the receiver's balance.
    let to_balance = self.balance_of_or_zero(&to);
    self.balances.insert(to, to_balance + value);

    self.env().emit_event(Transfer {
        from: Some(from),
        to: Some(to),
        value,
    });

    true
}
```

The snippet above is a function that transfers, in Polkadot network, some value of a specific ERC-20 token, from an account to another, using Substrate framework. As we can see Rust syntax is clear, easy to audit and supports either static or dynamic typing. Although it also has support for dynamic typing, Rust's compiler infers types by determining the type of a value based on the context in which it is used, making it a type safe language. Addresses in Substrate are handled with a specific data type created by Polkadot, named **AccountId**.

2.4.2.6 Go

Go is a statically typed, compiled programming language designed by Google very similar to C. This language offers many features like memory safety, garbage collection and structural typing. Hyperledger fabric's chaincodes can be written in Go.

If you want to have know how chaincodes written in Go look like, you can see some examples here.

```
1  func (s *SmartContract) TransferAsset(ctx
       contractapi.TransactionContextInterface, id string, newOwner string) error
       {
2
3    asset, err := s.ReadAsset(ctx, id)
4    if err != nil {
5      return err
6    }
7
8    clientID, err := s.GetSubmittingClientIdentity(ctx)
9    if err != nil {
10     return err
11   }
12
13   if clientID != asset.Owner {
14     return fmt.Errorf("submitting client not authorized to update asset, does
           not own asset")
15   }
16
17   asset.Owner = newOwner
18   assetJSON, err := json.Marshal(asset)
19   if err != nil {
20     return err
21   }
22
23   return ctx.GetStub().PutState(id, assetJSON)
24 }
```

2.4.2.7 C#

NEO is a blockchain protocol fully built in C# and consequently it is also the platform native language to write smart contracts. However, this protocol supports many high level languages, providing compilers and plug-ins for these languages, which are used to compile high-level languages into instruction sets supported by NEO virtual machines. Apart from C# you can write smart contracts for NEO virtual machines in: Python, Java/Kotlin, Go and JavaScript.

Here are some examples of smart contracts written in C# for NEO blockchain.

```
1  private static bool Transfer(byte[] from, byte[] to, BigInteger amount,
       byte[] callscript)
2  {
3        //Check parameters
4        if (from.Length != 20 || to.Length != 20)
5            throw new InvalidOperationException("The parameters from and to
                 SHOULD be 20-byte addresses.");
6        if (amount <= 0)
7            throw new InvalidOperationException("The parameter amount MUST be
                 greater than 0.");
8        if (!Runtime.CheckWitness(from))
9            return false;
10       StorageMap asset = Storage.CurrentContext.CreateMap(nameof(asset));
11       var fromAmount = asset.Get(from).AsBigInteger();
12       if (fromAmount < amount)
13           return false;
14       if (from == to)
15           return true;
16
17       //Reduce payer balances
18       if (fromAmount == amount)
19           asset.Delete(from);
20       else
21           asset.Put(from, fromAmount - amount);
22
23       //Increase the payee balance
24       var toAmount = asset.Get(to).AsBigInteger();
25       asset.Put(to, toAmount + amount);
26
27       Transferred(from, to, amount);
28       return true;
29  }
```

2.4.2.8 Move

Move is a safe and flexible programming language for the Diem Blockchain, originally developed by Facebook and tailored as a secure and verified programming language. It allows developers to define types to represent and control assets, but has no native support for contract protocols or gas usage. Move is designed to be a platform-agnostic language to enable common libraries, tooling, and developer communities across diverse blockchains with vastly different data and execution models. This language aims to be the "JavaScript of web3" in terms of ubiquity, when developers want to quickly write safe code involving assets, it should be written in Move.

In the following repository, you can find an example of a contract implementing ERC-20 standard, that models the standard for fungible tokens, in Move.

2.4.2.9 Michelson

Michelson is an open-source function programming language designed for Tezos smart contracts development, introduced in 2014, when Tezos whitepaper [10] was released. The language is stack-based, with high level data types and primitives, strict static type checking and was designed to facilitate formal verification, allowing users to prove the properties of their contracts. It is inspired in not so popular languages like Forth, Scheme, ML and Cat.

2.4.2.10 TEAL

TEAL is an assembly-like language, processed by the AVM. The language is Turing-complete that supports looping and subroutines, but limits the amount of time the contract has to execute using a dynamic opcode cost evaluation algorithm. TEAL programs are processed one line at a time pushing and popping values on and off the stack. These stack values are either unsigned 64 bit integers or byte strings. TEAL provides a set of operators that operate on the values within the stack.

2.4.2.11 Marlowe

Marlowe is a programming language designed by Cardano Foundation for the eponymous blockchain. It is written as a Haskell data type, enabling users to create smart contracts in a low code platform called Blockly. This platform makes it easier for non-developers to build and deploy smart contracts. You can also write smart contracts in Haskell or JavaScript, which are more familiar to programmers and then convert into Marlowe code.

3

State of the Art

In this chapter, we offer an overview of smart contract languages developed through academic research. These languages are carefully crafted to address specific functionalities while emphasizing safety and security—a crucial consideration given the foundational role of smart contracts in decentralized applications. While we will not delve deeply into technical details, we will explore the key features and advantages of each language, shedding light on their contributions to secure and reliable smart contract execution. Familiarizing ourselves with these academic developments provides valuable insights into the broader landscape of smart contract development and the ongoing pursuit of safer blockchain technologies. Additionally, we will discuss well-known vulnerabilities in Solidity and the static analysis tools—both academic and industrial—that can be employed to detect them. Finally, we introduce the most important features of Featherweight Solidity.

3.1 Academic Languages

In the following table there is a summary of the different domain-specific languages to develop smart contracts.

	Solidity	Vyper	Scilla	Move	Michelson	Bamboo	Flint	Obsidian	TEAL
Turing Completeness	X				X				X
Type safe		X	X	X	X	X	X	X	X
Recursive calls	X				X				X
Function Modifiers	X								
Production ready	X				X				X
Assembly Language					X				X
Platform Agnostic			X	X				X	

3.1.1 Vyper

Vyper is a pythonic programming language for EVM which addresses resource usage and overflow checking. This programming language dropped some features that Solidity have in order to make contracts more secure and easier to audit. However it still suffers from

re-entrancy vulnerabilities and it is not intended to be a full replacement for everything that can be done in Solidity. It will deliberately forbid things or make things harder if it deems fit to do so for the goal of increasing security.

3.1.2 Scilla

Scilla is a language being developed for the Zilliqa blockchain. It imposes a structure on smart contracts that makes applications less vulnerable to attacks by eliminating known vulnerabilities directly at the language-level. Ziliqa has been designed to be scalable, employing the idea of sharding[1] to validate transactions in parallel thus making transaction processing faster [22]. Although Scilla was designed in the context of Zilliqa, it is platform-agnostic and hence can be used with any other blockchain.

3.1.3 Bamboo

Bamboo is a programming language designed to write smart contracts for EVM. Bamboo makes state transition explicit and avoids reentrancy problems by default. However this language does not support features the following features:

- Loop constructs (for, while, ...) - Due to the constant block gas limit, loops should be avoided and each iteration should be done in separate transactions.
- Assignments into storage variables, except array elements - Instead of assigning a new value to a storage variable, the new value can be given as an argument in the continuation

Here are a lot of smart contract examples written in Bamboo.

3.1.4 Flint

Flint is a new type-safe contract-oriented programming language to write smart contracts for EVM, that has a variety of novel contract-oriented features, such as caller capabilities, immutability by default, and asset types. This programming language also provides safer operations, meaning that arithmetic operations on integers are safe by default: an overflow/underflow causes the Ethereum transaction to be reverted. However, Flint is still in alpha development, is an academic language developed in Imperial College London thus is still not ready to be used in production yet, lacking several important aspects like asset protection or gas consumption control.

3.1.5 Obsidian

Obsidian is another academic programming language that enables developing smart contracts for EVM. The authors of this project claim that Obsidian includes two innovations

[1]Sharding is a process by which processing of the network can be made less intensive and allow for greater transaction volume[13].

relative to current languages for these platforms: state-oriented programming which lets developers declare and transition among states explicitly and linear types which ensure that important resources managed by your programs are managed correctly. Like Flint, this programming language is an academic language currently under development and is not production ready yet.

3.2 Known Vulnerabilities in Solidity

For the purpose of this dissertation, we have chosen to focus on studying Solidity, as it is currently the most popular language for writing smart contracts and has the most extensive resources available on the internet. Next, we will address known vulnerabilities in Solidity and provide examples of contracts that have the vulnerability and are not safe.

3.2.1 Type Casts

The Solidity compiler can detect some standard type errors (e.g., assigning an integer value to a variable of type string), as any strongly typed programming language. However, as a contract written in the Solidity language can call another contract by directly referencing the callee contract's instance, when a contracts calls another contracts function, it only checks if the interface matches; it does not check in particular if the contract passed as an argument is from the same type as it is declared in a function. Therefore, a developer should be careful whenever a public function in a contract calls another contract interface.

```
1  contract CounterLibrary {
2      uint256 public counter;
3
4      function add() public returns (uint256) {
5          counter++;
6          return counter;
7      }
8  }
9
10 contract FakeCounter {
11     uint256 public counter;
12
13     function add() public returns (uint256) {
14         return counter;
15     }
16 }
17
18 contract Game {
19     uint256 public counter;
20
21     function play(CounterLibrary c1) public returns (uint256) {
22         uint256 res = c1.add();
23         return res;
24     }
25 }
```

Figure 3.1: An example of Type Casts vulnerability

Figure 3.1 shows an example of this vulnerability. Assuming these three contracts, if you deploy them into the blockchain, you can then call the play function from Game contract, which receives a CounterLibrary type contract. Nevertheless, you can either call this function with CounterLibrary contract address or FakeCounter contract address, even if only the first contract match the type declared, both match the interface required by play function. Obviously they will have different execution and output (CounterLibrary will return a true counter, but FakeCounter will always return 0).

There is still no way to prevent this vulnerability.

3.2.2 Reentrancy

This vulnerability was first seen in the DAO attack [7]. The DAO was a digital decentralized autonomous organization and a form of investor-directed venture capital fund. This idea was inspired by crowdfunding and was meant to work as a big decentralized hedge fund. At the time, the DAO was the largest crowdfunding project ever and raised 150 milion dolars, approximately 15% of Ethereum market cap. It was simply too big to fail and Ethereum Foundation decided controversially to hard fork [3] their own blockchain. However, some miners refused to fork because the DAO incident was not a defect in the protocol, arguing that "Code is Law" and formed Ethereum Classic, the original unaltered Ethereum blockchain. The vulnerability occurs when an external callee contract calls back to a function in the caller contract before the caller contract finishes, allowing an attack to

bypass the due validity check until the caller contract is drained of Ether or the transaction runs out of gas. Reentrancy can be prevented using one of the following methods:

- Assuring that a contract's state variables are updated before calling another contract.
- Introducing a mutex lock on the contract state to assure that only the lock owner can change the state(see ReentrancyGuard created by OpenZeppelin).
- Using the transfer or send (instead of call) methods to send money to other contracts, because this methods only forwards 2,300 gas to the callee contract.

```
function withdraw() public {
    uint bal = balances[msg.sender];
    require(bal > 0);

    (bool sent, ) = msg.sender.call{value: bal}("");
    require(sent, "Failed to send ether");
    balances[msg.sender] = 0;
}

function nonReentrantWithdraw() public {
    uint bal = balances[msg.sender];
    require(bal > 0);

    balances[msg.sender] = 0;
    (bool sent, ) = msg.sender.call{value: bal}("");
    require(sent, "Failed to send ether");
}
```

Figure 3.2: An example of a function vulnerable to reentrant attacks

3.2.3 Phishing waith tx.origin

In Solidity, tx.origin is a global variable that points to the initial externally owned account (EOA) responsible for initiating the current transaction. This vulnerability occurs when a contract uses tx.origin for authorization, which can be compromised by a phishing attack. A malicious contract can deceive the owner of a contract into calling a function that only the owner should be able to call.

This vulnerability can be prevented by using msg.sender, instead of tx.origin, for authentication, because if a contract A calls B, and B calls C, in C msg.sender will return the address of contract B and tx.origin will return the address of contract A.

```
contract  Wallet {
    address public owner;

    constructor() payable {
        owner = msg.sender;
    }

    function transfer(address payable _to, uint _amount) public payable {
        require(tx.origin == owner, "Not owner");
        // require(msg.sender == owner, "Not owner"); correct way to
            authenticate!

        (bool sent, ) = _to.call{value: amount}("");
        require(sent, "Failed to send ether");
    }
}

contract Attack {
    address payable public owner;
    Wallet wallet;

    constructor(Wallet _wallet) {
        wallet = Wallet(_wallet);
        owner = payable(msg.sender);
    }

    function attack() public {
        wallet.transfer(owner, address(wallet).balance);
    }
}
```

Figure 3.3: An example of a vulnerable contract to phishing

3.2.4 Delegatecall Injection

Delegatecall is a solidity function to facilitate code-reuse inserting a callee contract's bytecode into the bytecode of the caller contract: it simply let's a contract call another contract's external function. However using delegatecall is tricky to use and wrong usage or incorrect understanding can lead to devastating results. As a consequence, a malicious callee contract can directly modify (or manipulate) the state variables of the caller contract.

```
function delegateCallToAnotherContract(address _contract) public {
    (bool success, bytes memory data) =
        _contract.delegatecall(abi.encodeWithSignature("setVars(uint256)")
    );
}
```

Figure 3.4: An example of a dangerous delegatecall usage

3.2.5 Unprotected Selfdestruct

```
contract EtherGame {
    uint public targetAmount = 7 ether;
    address public winner;

    fuction deposit() public payable {
        require(msg.value == 1 ether, "You can only send 1 ether");
        uint balance = address(this).balance;
        require(balance <= targetAmount, "Game is over");

        if(balance == targetAmount) {
            winner = msg.sender;
        }
    }
    function claimReward public {
        require(msg.sender == winer, "Not winner");
        (bool sent, ) = msg.sender.call{value: address(this).balance}("");
        require(sent, "Failed to send ether");
    }
}

contract Attack {
    EtherGame etherGame;

    constructor(EtherGame _etherGame) {
        etherGame = EtherGame(_etherGame);
    }

    public attack() public payable {
        //You can simply break the game forcing sending ether so that
        // the game balance >= 7
        address payable addr = payable(address(etherGame));
        selfdestruct(etherGame)
    }
}
```

Figure 3.5: An example of a malicous use of selfdestruct

Selfdestruct is a built-in Solidity function that destroys the current contract, sending its funds to the given address and end execution. This function has some peculiarities inherited from the EVM:

- The receiving contract's receive function is not executed.
- The contract is only really destroyed at the end of the transaction and revert s might "undo" the destruction.

This vulnerability was first seen in Parity wallet [26]. The Parity wallet was designed to integrate seamlessly with all standard tokens as well as manage Ether transfers. However, there was a vulnerability with this contract reported on 6th November 2017 by an anonymous user. The user decided to exploit this vulnerability and made himself the "owner" of

the library contract. Subsequently, the user could access a special function which only the owner of the Parity wallet should have been able to access. This function allowed a user to destroy this component affecting all subsequent contracts that relied on this contract. This action blocked funds in 587 wallets holding a total amount of 513,774.16 Ether as well as additional tokens. To prevent this vulnerability a developer must be careful when implementing a function that call the built-in selfdestruct function, making sure that an arbitrary user can never get access to it.

An attacker can also create a contract with a selfdestruct() function, send ether to it, call selfdestruct(target) and force ether to be sent to a target. This vulnerability arises from the misuse of the globally available variable address(this).balance. For this reason your contract should never rely on this variable, as it can be artificially manipulated.

3.2.6 Gasless Send/Transfer

```
1  contract C {
2
3      constructor() payable {
4
5      }
6
7      function pay(uint n, D1 d) public payable {
8          address d1 = address(d);
9          payable(d1).transfer(n);
10     }
11     function balance() public view returns (uint) {
12         return address(this).balance;
13     }
14 }
15
16 contract D1 {
17     uint public counter = 0;
18
19     //this fallback function costs more than 2300 gas
20     //so the transfer/send method will fail, use call instead
21     fallback() external payable {
22         counter++;
23     }
24
25     function balance() public view returns (uint) {
26         return address(this).balance;
27     }
28 }
```

Figure 3.6: An example of a transfer that will fail due to not having enough gas

When using the functions send() or transfer() to transfer ether to another contract, it is possible to incur in an out-of-gas exception, as these functions only forward 2300 gas to the callee contract. This may be quite unexpected by programmers, because transferring ether is not generally associated to executing code. The reason behind this exception is

subtle. Note that c.send(amount) or c.transfer(amount) are compiled in the same way of a call with empty signature, but remember that the actual number of gas units available to the callee is always bound by 2300, to prevent reentrancy attacks. Now, since the call has no signature, it will invoke the callee's receive or fallback function. However, 2300 units of gas only allow to execute a limited set of bytecode instructions, e.g. those which do not alter the state of the contract. In any other case, the call will end up in an out-of-gas exception.

Summing up, sending ether via send or transfer functions succeed in two cases: when the recipient is a contract with an inexpensive fallback, or when the recipient is a user. If a developer wants to execute complex computations and change a contract state, after a contract receives ether, he should use the call function to transfer ether since this function does not impose a forward gas limit when sending funds to the callee contract.

3.2.7 Unprotected ether withdrawal

```
1
2  function owner() public view virtual returns (address) {
3      // returns owner
4      ....
5  }
6
7  modifier onlyOwner() {
8      require(owner() == msg.sender, "Not owner");
9  }
10
11 fuction protectedWithdraw(uint amount) public onlyOwner {
12     require(amount > 0);
13     require(amount <= balance, "No funds to withdraw the amount requested.");
14
15     balance -= amount;
16     payable(msg.sender).transfer(amount);
17 }
18
19 fuction unProtectedWithdraw(uint amount) public {
20     require(amount > 0);
21     require(amount <= balance, "No funds to withdraw the amount requested.");
22
23     balance -= amount;
24     payable(msg.sender).transfer(amount);
25 }
```

Figure 3.7: An example of a protected and an unprotected ether withdraw function

This vulnerability occurs when bad actors without adequate access controls, are be able to withdraw some or all ether from a contract. To avoid this vulnerability, only allow withdrawals to be triggered by those authorized, or as intended, normally only the owner of the contract should be able to withdraw ether. This vulnerability can be avoided extending Ownable class offered by OpenZeppelin.

3.3 Tools To Verify Solidity Smart Contracts

In our pursuit of enhancing the security of Solidity smart contracts, it is essential to understand how different static analysis tools can help us mitigate vulnerabilities. Before we delve into the specifics of each tool, let's take a moment to explore a comprehensive summary table. This table will provide a quick reference, showcasing which vulnerabilities each tool excels at identifying. Understanding these strengths will help us make informed choices when selecting the most suitable tool for our particular needs.

	Slither	Mythril	SMTChecker	Solc-verify	Solidifier
Type Casts					
Reentrancy	X	X		X	
Phishing with tx.origin	X	X	X		X
Delegatecall Injection	X	X			
Unprotected Selfdestruct	X	X			
Gasless send/transfer					
Unprotected Ether Withdrawal	X	X*	X	X*	X*

3.3.1 Slither

Slither is a Solidity static analysis framework that checks for specific vulnerabilities. Moreover it also lets developers extend this tool through an API that enables one to write custom analyses. At the time of its launch, it was the first open-source static analysis framework for Solidity. The main features provided by this tool are:

- Very fast and precise, averaging less than 1 second of execution time per contract and detects vulnerable Solidity code with low false positives.
- Identifies where the error condition occurs in the source code.
- Easily integrates into continuous integration and Truffle builds.
- Built-in 'printers' quickly report crucial contract information.
- Detector API to write custom analyses in Python.
- Correctly parses 99.9% of all public Solidity code
- Ability to analyze contracts written with Solidity >= 0.4 and so compatible with the most recent Solidity versions.
- Intermediate representation (SlithIR) enables simple, high-precision analyses.

This tool was the best performer in both execution speed time and accuracy in all these 7 vulnerabilities. It worked very well to detect 5 of those, only missing Type Casts and Gasless send/transfer vulnerabilities.

3.3.2 Mythril

Mythril is a security analysis tool for EVM bytecode. It offers a command that checks a contract or all the contracts inside a directory, similarly to slither. However I could note that this tool is less precise and slower than slither, as this tool checks Solidity compiled code (binaries), giving more false-positives and more complex to analyze. It is also not possible to customize this tool, because it does not offer any API to do so, like slither does. It is also worth noting that this tool could not be accurate when checking a contract protected against Unprotected Ether Withdrawal when using modifiers, like "onlyOwner" implemented by OpenZeppelin. It only stopped giving this vulnerability if we explicit put a pre-condition "**require(owner == msg.sender)**" before executing a contract code. This tool was not as good as Slither when treating reentrancy vulnerability as it warned reentrancy when we transferred ether using call() method, even if the function was protected against reentrancy with ReentrancyGuard or using good development practices.

3.3.3 SMTChecker

SMTChecker is a Solidity implementation of a formal verification approach based on Satisfiability Modulo Theories (SMT) and Horn solving. This module automatically tries to prove that the code satisfies the specification given by require (pre-conditions) and assert (post-conditions) statements. That is, it considers require statements as assumptions and tries to prove that the conditions in assert statements are always true. If an assertion fails, a counterexample may be given to the user showing how the assertion can be violated. If no warning is given by the SMTChecker for a property, it means that the property is safe. This module also targets other verifications that are checked at compile time:

- Arithmetic underflow and overflow.
- Division by zero.
- Trivial conditions and unreachable code.
- Popping an empty array.
- Out of bounds index access.
- Insufficient funds for a transfer.

All the targets above are automatically checked by default if all engines are enabled, except underflow and overflow for Solidity versions equal or greater than 0.8.7.

I have to note that I talked with one of the authors and did not get any response whether it is possible or not, to detect Reentrancy vulnerability with this tool. The way I tried was not working, as the tool was stuck in a infinite loop, producing no output.

3.3.4 Solidifier

Solidifier is a bounded model checker for Solidity. This tool translates Solidity code into the Boogie language which is later verified by Corral. Unlike many of the tools developed to analyse smart contracts that check for known possibly problematic/vulnerable behaviours, Solidifier looks for violation to assertions. The semantic properties described in this way should better describe the intent of developers and so they more finely capture mistakes and possibly vulnerabilities of smart contracts. This tool works very similar like VeriSol and Solc-verify. Both try to falsify/prove semantic properties of smart contracts.

This tool requires a contract to declare the following library in order to get access to verification primitives:

```
1 library Verification {
2     function Assume (bool b);
3     function Assert (bool b);
4     function CexPrintui (uint ui);
5 }
```

Verification.Assume(condition) primitive fails silently (stopping the contract's execution) when the condition is not met and **Verification.Assert(condition)** fails with an error that is reported by Solidifier. Both this functions are identical to SMTChecker primitives **require** and **assert**. The Verification library can also declare functions to print the value of expressions of basic types to the counterexample trace. These are prefixed by **CexPrint**. In this example, CexPrintui (uint ui) declares a function that prints a unsigned integer. On other hand we could have declared a function CexPrintad (address ad) that print addresses. It is important noting that printing only happens if the counterexample reaches a given printing statement.

Using this tool we were only able to detect Phishing with tx.origin vulnerability. This tool does not support function modifiers, delegatecall or external contract calls, thus making impossible to detect other vulnerabilities. However it was possible to kinda detect and prevent Unprotected Ether Withdrawal using Assume function as a pre-condition to ensure that only the owner of the contract is able to withdraw ether from a contract and using Assert to prove that pre-condition.

3.3.5 Solc-verify

Solc-verify [11] is an extended version of the Solidity compiler that is able to perform automated formal verification on Solidity smart contracts using specification annotations and modular program verification. They must be side-effect free Solidity expressions and can refer to variables within the scope of the annotated element. This tool allows access to loop and contract invariants, function pre and post conditions, functional modifiers, specific functions referring to a sum of a collection or old variable values and some more that you can find here.

However this tool does not support built-in Solidity functions or state variables like tx.origin, delegatecall and selfdestruct making it impossible to detect vulnerabilities involving these. It also has false positives for reentrancy when using send() and transfer() functions which are indeed reentrant proof.

3.3.6 Other Tools

In this section I will talk about tools that I was not able to try because they either are discontinued or I could not install and the authors did not help me with my installation errors.

3.3.6.1 VeriSmart

VeriSmart is a safety analyzer for Solidity smart contracts written in OCaml. This tool apparently is still under development, since the last commit on GitHub was in January 2023. It supports two types of analysis modes:

- **Safety verification** - In this mode, VeriSmart can be used to prove the absence of bugs or detect bugs. The key feature in this mode is, VeriSmart automatically infers transaction invariants, which are conditions that hold under arbitrary interleaving of transactions, of smart contracts. With this ability, VeriSmart can precisely analyze safety properties in smart contracts.

- **Vulnerable transaction sequence generation** - In this mode, VeriSmart can be used to generate vulnerable transaction sequences (with concrete arguments for each transaction) that demonstrate the flaws. They claim that the key feature in this mode is, they guide a symbolic execution procedure with language models to find vulnerable sequences effectively.

Unfortunately I was not able to experiment this tool because when running a specific contract I got an error and even though I opened an issue on GitHub, the owners did not reply me in time.

3.3.6.2 SmartAce

SmartACE is a tool which extends the Solidity compiler to support bounded and parameterized smart contract verification. This tool is built upon a novel model for smart contracts, in which users are processes and communication is explicit. In this model, communication is over-approximated by static analysis, and the results are sufficient to find all local neighbourhoods. This is achieved by using SeaHorn, an automated analysis framework for LLVM-based languages. It takes as input a smart contract annotated with assertions, then it checks that all assertions hold.

I did not have time to explore more this tool and had some problems running it. It is worth noting that this tool is only compatible with contracts written in Solidity version 0.5.17 and we need to port them back if we want to test contracts from version above.

3.3.6.3 VeriSol

VeriSol is a Microsoft Research project for prototyping a formal verification and analysis system for smart contracts developed in the popular Solidity programming language. It is based on translating programs in Solidity language to programs in Boogie intermediate verification language, and then leveraging and extending the verification toolchain for Boogie programs. It is is not compatible with Solidity versions 0.8 authors are not actively maintaining the tool since 2021.

3.3.6.4 SmartCheck

SmartCheck is an extensible static analysis tool for discovering vulnerabilities and other code issues in Ethereum smart contracts that are written in the Solidity programming language. However this tool is deprecated since 2020 and I could not run it properly on my machine.

4

Featherweight Solidity

4.1 General Context

As previously stated, our main focus during this master thesis was studying, and redefining Featherweight Solidity and then implementing it in OCaml. Featherweight Solidity is a minimal core calculus for Solidity, inspired by Featherweight Java [12], focused on the core of Solidity and dropping complex features, to study its type system. Therefore, it enables the study of various aspects of smart contract programming, such as new contracts deployment, interaction among deployed contracts, and money transfers.

It includes both the semantics and the type system of Solidity, providing a comprehensive understanding of the language's behavior and its potential vulnerabilities. It also allows a new subtype for the type address, making this new redefining system able to detect which contract an address refers to.

The formalization of Solidity in FS was a significant step towards understanding the language's features and vulnerabilities, marking the first formalization of Solidity. While FS has its limitations and some aspects of Solidity have not been fully modeled yet, it provides a foundation for further research into new language features and contract vulnerabilities. Our implementation keeps all of essential feature embedded in the original formalization, but we extended it to meet some of the most important feature from Solidity yet to be modeled: multiple inheritance, flexibility on constructor bodies, default values assigned to variables and another adjustments to be in-line with the most recent Solidity version to date (0.8.*).

Next, we will present the original formalization.

4.2 Meta Variables

Let's begin by introducing some meta variables that will be used in the subsequent rules:

- C: The set of contract names, and C, D identifies each element in the set.
- $\mathcal{S}$: The set of state variable identifiers, and s identifies each element.

- $\mathcal{F}$: The set of function names, and f identifies each element.
- **X**: The set of argument names, and x, y identifies each element.
- **V**: The set of values, and v identifies each element.
- **T**: The set containing all possible types of our language, and T identifies each element.

it is important to note that we consider null as a concrete value for a contract name. Therefore, every reference to c represents a value within an infinite set that also includes null.

Additionally, the variable f represents all possible values for a function identifier, including the unknown. However, this special value cannot be used in a function defined by the user.

4.3 Grammar

(*Contract decl.*)	SC	::=	$\texttt{contract}\ C\ is\ D\{(\widetilde{T\ s})\ K\ \widetilde{F}\}$
(*Constructor decl.*)	K	::=	$constructor\ ((\widetilde{T_1\ y}, \widetilde{T_2\ x}))\{(\texttt{super}(\widetilde{y}); \texttt{this}.\widetilde{s} = \widetilde{x}\}$
(*Function decl.*)	F	::=	$T\ f\ \langle C\rangle\ ((\widetilde{T\ x}))\{\texttt{return}\ e\}$ |
			$\texttt{unit}\ f b\ \langle C\rangle\ ()\ \{\texttt{return}\ e\}$
(*Expression*)	e	::=	$v \mid x \mid b \mid \texttt{this.s} \mid \texttt{this}.f(\widetilde{e}) \mid$
			$\texttt{msg.sender} \mid \texttt{msg.value} \mid$
			$\texttt{address}(e) \mid e.s \mid e.\texttt{transfer}(e)$
			$\texttt{new}\ C.\texttt{value}(e)((\widetilde{e})) \mid$
			$e; e \mid T\ x = e \mid x = e \mid e.s = e \mid$
			$e[e] \mid e[e \rightarrow e] \mid e.\texttt{value}(e)(\widetilde{e}) \mid$
			$e.f.\texttt{value}(e)(\widetilde{e}) \mid \texttt{revert} \mid$
			$e.f.\texttt{value}(e).\texttt{sender}(e)(\widetilde{e}) \mid$
			$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
(*Values*)	v	::=	$\texttt{true} \mid \texttt{false} \mid n \mid a \mid \texttt{u} \mid M \mid$
			c
(*Types*)	T	::=	$\widetilde{T} \rightarrow T \mid \texttt{bool} \mid \texttt{uint} \mid \texttt{address}\langle C\rangle \mid$
			$\texttt{unit} \mid \texttt{mapping}(T \Rightarrow T) \mid C$

Figure 4.1: Syntax of Featherweight Solidity+

This is the grammar for the original extended Featherweight Solidity formalization. Please note that it only allows for simple inheritance. Although mutual recursion is permitted according to this grammar definition, we assume non-mutually recursive contract definitions. it is important to highlight that their grammar enforces the requirement for every contract to inherit from another. **However, we disagree with this choice because we**

believe it should be an axiom that every contract always inherits from a top-level class or contract and thus there should be the possibility for a contract to not inherit from any class but the top class.

Additionally, there are four important properties of subtyping that we'd like to emphasize:

$$\frac{}{C <: \texttt{Top}}\ (\text{TOP}) \qquad \frac{}{C <: C}\ (\text{REFLEXIVITY})$$

$$\frac{\texttt{contract}\ C\ is\ D\{(\widetilde{T\ s})\ K\ \widetilde{F}\}}{C <: D}\ (\text{CONTRACT}) \qquad \frac{C <: D \qquad D <: E}{C <: E}\ (\text{TRANSITIVITY})$$

These rules also apply to subtyping in address subtyping. In short, for $\texttt{address}\langle C\rangle <: \texttt{address}\langle D\rangle$ to be true, C must be a subtype of D, that is $C <: D$. This is known as co-variance and its rule is listed below:

$$\frac{C <: D}{\texttt{address}\langle C\rangle <: \texttt{address}\langle D\rangle}\ (\text{ADDRESS-COVARIANCE})$$

We can also see that constructor declaractions are more strict than Solidity, enforcing one to initialize all state variables.

Functions are identical to Solidity, except in FS+ we can restrict what type msg.sender can be. This states the required maximum supertype allowed for the implicit variable msg.sender. It is validated at compile-time, supposing msg.sender as an expression of type $\texttt{address}\langle C\rangle$ and checking the subtyping for each call of f. **We also disagree with making it a mandatory annotation because this approach is not retro-compatible with Solidity. By default, it should expect all types of addresses.**

4.3.1 Contract Definition

We write $\texttt{contract}\ C\ is\ D\{(\widetilde{T\ s})\ K\ \widetilde{F}\}$ to denote a smart contract declaration. It indicates that contract C depedends on a contract D, a sequence possible empty of typed variables, denoted by $\widetilde{Ts}$, a mandatory constructor declaration and a set of functions ($\widetilde{F}$), also can be empty. However if a contract does not have at least a fallback function ($\texttt{unit}\ fb()\{\texttt{return}\ e\}$) it will not be able to receive ether.

4.3.2 Constructor Declaration

In this version, it is imperative that all state variables of the contract be meticulously initialized. The set of typed parameters, denoted as $\widetilde{T_1 y}$, must align in length with the

state variables of the super contract. Similarly, the set $\widetilde{T_2 x}$ must conform to the contract's own variables, ensuring a precise match.

4.3.3 Expression

This represents our set of possible expressions. Our keyword **this** allow to referring to the current contract state variables and functions, unlike Solidity. In Solidity, it is only used to reference function, we do not need to use the keyword to access contract's state variables.

4.3.4 Values

As values we accept boolean values (true or false) and an unsigned integer, denoted by n. There is also a possibility to write a specific address, if one uses the constructor $address(0)$ it will refer to the special empty address (default value for all addresses). The rest of the values are only allowed as a return type of a function or being the value of specific keywords, for example the keyword **this**.

4.3.5 Types

We have all types matching each possible value for our expressions, plus a type for typing functions $\widetilde{T} \to T$.

4.4 Configuration

$$
\begin{array}{llll}
(\textit{Blockchain}) & \beta & ::= & \emptyset \mid \beta \cdot [x \mapsto v] \mid \\
 & & & \beta \cdot [(c, a) \mapsto (C, \widetilde{s : v}, n)] \mid \\
(\textit{Call Stack}) & \sigma & ::= & \beta \mid \sigma \cdot a \\
(\textit{Contract Table}) & CT & ::= & \emptyset \mid CT \cdot [C \mapsto D]
\end{array}
$$

Figure 4.2: Environments of Featherweight Solidity+

The call stack σ is responsible for tracking the nesting of function calls, specifically the addresses of their enclosing contracts, during the execution of a transaction. To simplify the operational semantics during transaction execution, they mandate that the initial element of the call stack must represent a copy of the blockchain as it existed at the commencement of the top-level transaction.

This initial blockchain copy, situated at the base of the call stack, serves a critical purpose in the event of an abort, where the blockchain state must be reverted to the beginning of the transaction. Conversely, if the transaction successfully commits, the modified blockchain will be duplicated to replace the copy held within σ.

Their configuration differs slightly from ours. As previously mentioned, they store a copy of the blockchain within the call stack, denoting their configuration as $\langle \beta, \sigma, e \rangle$, with its initial value set as $\langle \beta_0, \beta_0, e \rangle$.

On the other hand, our approach utilizes a quadruple configuration, and we maintain a separate entry for the copy of the blockchain, represented as $\langle \beta, \beta_0, \sigma, e \rangle$.

In summary, a successfully evaluated transaction concludes with a configuration in the form of $\langle \beta, \beta_0, v \rangle$, leading to the commitment of the updated blockchain β. Conversely, when an error occurs, the configuration is represented as $\langle \beta, \beta_0, \texttt{revert} \rangle$. In this scenario, the modified blockchain β is discarded, returning to the initial state β_0 and effectively simulating the rollback of the aborted transaction.

4.5 Lookup Functions

These lookup functions are used in operational semantics and type system rules and are formally defined in this section.

The first function accepts a contract as input and retrieves the contract's state variables. The second function takes a contract, a function name, and a set of values, returning a tuple containing the function's parameters and its body. The third function, given a contract and a function name, provides the type associated with the function's parameters and body

$$\texttt{sv}(\texttt{Top}) = \emptyset$$

(STATE VARIABLE LOOKUP)

$$\frac{\texttt{contract}\ C\ \textit{is}\ D\{(\widetilde{T\ s})\ K\ \widetilde{F}\} \qquad \texttt{sv}(D) = \widetilde{T_1\ r}}{sv(C) = \widetilde{T\ s}; \widetilde{T_1\ r}}$$

$$\texttt{fbody}(\texttt{Top}, fb, \widetilde{v}) = (\{\}, \texttt{revert})$$

(FUNCTION BODY LOOKUP - 1)

$$\frac{CT(C) = \texttt{contract}\ C\ \textit{is}\ D\{(\widetilde{T\ s})\ K\ \widetilde{F}\} \qquad T\ f\ \langle C' \rangle\ ((\widetilde{T\ x}))\{\texttt{return}\ e\} \in \widetilde{F} \qquad \widetilde{x} = \widetilde{v}}{\texttt{fbody}(C, f, \widetilde{v}) = (\widetilde{x}, \texttt{return}\ e)}$$

(FUNCTION BODY LOOKUP - 2)

$$\frac{CT(C) = \texttt{contract}\ C\ \textit{is}\ D\{(\widetilde{T\ s})\ K\ \widetilde{F}\} \qquad T\ f\ \langle C' \rangle\ ((\widetilde{T\ x}))\{\texttt{return}\ e\} \notin \widetilde{F}\ \vee\ \widetilde{x} \neq \widetilde{v}}{\texttt{fbody}(C, f, \widetilde{v}) = \texttt{fbody}(SC, f, \widetilde{v})}$$

(FUNCTION SIGNATURE LOOKUP - 1)

$$\frac{CT(C) = \texttt{contract}\ C\ \textit{is}\ D\{(\widetilde{T\ s})\ K\ \widetilde{F}\} \qquad T\ f\ \langle C'\rangle\ ((\widetilde{T\ x}))\{\texttt{return}\ e\} \in \widetilde{F}}{\texttt{ftype}(C, f) = \widetilde{T} \to T}$$

(FUNCTION SIGNATURE LOOKUP - 2)

$$\frac{CT(C) = \texttt{contract}\ C\ \textit{is}\ D\{(\widetilde{T\ s})\ K\ \widetilde{F}\} \qquad T\ f\ \langle C'\rangle\ ((\widetilde{T\ x}))\{\texttt{return}\ e\} \notin \widetilde{F}}{\texttt{ftype}(C, f) = \texttt{ftype}(SC, f)}$$

Lastly, we introduce `fsender` function. It retrives the type of `msg.sender` inside a function.

(SENDER MAXIMUM TYPE LOOKUP - 1)

$$\frac{CT(C) = \texttt{contract}\ C\ \textit{is}\ D\{(\widetilde{T\ s})\ K\ \widetilde{F}\} \qquad T\ f\ \langle C'\rangle\ ((\widetilde{T\ x}))\{\texttt{return}\ e\} \in \widetilde{F}}{\texttt{fsender}(C, f) = C'}$$

(SENDER MAXIMUM TYPE LOOKUP - 2)

$$\frac{CT(C) = \texttt{contract}\ C\ \textit{is}\ D\{(\widetilde{T\ s})\ K\ \widetilde{F}\} \qquad T\ f\ \langle C'\rangle\ ((\widetilde{T\ x}))\{\texttt{return}\ e\} \notin \widetilde{F}}{\texttt{fsender}(C, f) = \texttt{fsender}(SC, f)}$$

Note that all these functions are recursive. If the contract does not provide either the function of state variable we go up on the hierarchy and search for it again, until we reach the Top contract and return the base case value for each one.

4.6 Auxiliary Predicates

$$\texttt{uptbal} : BLOCKCHAIN \times ADDRESS \times BALANCE \to BLOCKCHAIN$$

$$\texttt{uptbal}(\beta, a, n) = \begin{cases} \beta[(c, a) \mapsto (C, \widetilde{s : v}, n' + n)] & \beta((c, a)) = (C, \widetilde{s : v}, n') \ \wedge \ n' + n \geq 0 \\ \bot & \texttt{otherwise} \end{cases}$$

We have simplified the original definition, making it equivalent. This predicate succeeds if and only if there exists a valid tuple (c, a) in β, and the sum of $n' + n$ is greater than or equal to 0. it is crucial to emphasize that n can be positive, negative, or zero.

$$\texttt{Top}(\sigma) = \begin{cases} a & \texttt{if}\ \sigma = \sigma' \cdot a \\ \emptyset & \texttt{if}\ \sigma = \beta \end{cases}$$

This predicate simply returns the top of the call stack, if there is one or $\emptyset$, if the call stack is empty.

4.7 Operational Semantics

4.7.1 Conditional and Sequential Statements

$$\frac{}{\langle \beta, \beta_i, \sigma, E[\texttt{revert}] \rangle \rightarrow \langle \beta, \beta_i, \sigma, \texttt{revert} \rangle} \text{(Revert)}$$

Every expression can eventually be evaluated to a revert. This rule is for cases that our expression violates a pre-condition and can't be evaluated by our rules, it should fallback to this rule. In the end, if we do not have explicitly made a rule for a failure, by omission, that rule should always be treated as a revert.

$$\frac{}{\langle \beta, \sigma, \texttt{if true then } e1 \texttt{ else } e2 \rangle \rightarrow \langle \beta, \sigma, e1 \rangle} \text{(If-True)}$$

$$\frac{}{\langle \beta, \sigma, \texttt{if false then } e1 \texttt{ else } e2 \rangle \rightarrow \langle \beta, \sigma, e2 \rangle} \text{(If-False)}$$

These rules have the standard meaning. Important to note that if we want to omit either branch the omitted branch should take the value unit internally.

$$\frac{\sigma = \beta_i}{\langle \beta, \beta_i, \sigma, v; e \rangle \rightarrow \langle \beta, \beta, e \rangle} \text{(Seq-C)}$$

$$\frac{\sigma = \beta_i}{\langle \beta, \sigma, \texttt{revert}; e \rangle \rightarrow \langle \beta_i, \sigma, \texttt{revert} \rangle} \text{(SEQ-R)}$$

$$\frac{\texttt{Top}(\sigma) = a}{\langle \beta, \sigma, v; e \rangle \rightarrow \langle \beta, \sigma, e \rangle} \text{(SEQ)}$$

These rules have to take into account different scenarios. First, when an expression like $v; e$ is found at the top level ($\texttt{Top}(\sigma) = \emptyset$)) then the last transaction was successful. In fact, it evaluated to a value v, which can be discarded to proceed with the next transaction e. Furthermore, since evaluating to a value is equivalent to a commit, we apply the changes and proceed with β. This scenario is modeled by SEQ-C. Secondly, v; e might be at an inner

level ($\texttt{Top}(\sigma) = a$). In this case (SEQ) we cannot apply any changes, since the transaction is not over yet. Hence, we just proceed with the evaluation of e, discarding v. Lastly, a revert may have been thrown during the evaluation, so instead of v; e the expression involving sequential composition is $\texttt{revert}; e$ (SEQ-R). In this case we do not further proceed in evaluating the other transactions and we replace β with β_i.

4.7.2 Variable Operations

$$\frac{}{\langle \beta, \sigma, x \rangle \rightarrow \langle \beta, \sigma, \beta(x) \rangle} \text{(VAR)}$$

Accesses a variable and returns its value.

$$\frac{x \in \texttt{dom}(\beta)}{\langle \beta, \sigma, \sigma, x = v \rangle \rightarrow \langle \beta \cdot [x \mapsto v], \sigma, v \rangle} \text{(ASSIGN)}$$

Assigns a value to an existing variable.

$$\frac{x \notin dom(\beta)}{\langle \beta, \sigma, T\ x = v; e \rangle \rightarrow \langle \beta \cdot [x \mapsto v], \sigma, v; e \rangle} \text{(LET)}$$

When a variable x is declared, it is added to the β. This behavior is represented and modeled by the LET function. VAR and ASSIGN functions, on the other hand, solely access β to either read or modify the value associated with a variable x.

The rules governing the reading and mutation of state variables are the following:

$$\frac{\exists\ a.\beta((c,a)) = (C, \widetilde{sv}, n) \qquad s \in \widetilde{s}}{\langle \beta, \sigma, c.s \rangle \rightarrow \langle \beta, \sigma, v_s \rangle} \text{(STATEREAD)}$$

$$\frac{\exists\ a.\beta((c,a)) = (C, \widetilde{sv}, n) \qquad s \in \widetilde{s}}{\langle \beta, \sigma, c.s = v' \rangle \rightarrow \langle \beta \cdot [c.s \mapsto v'], \sigma, v' \rangle} \text{(STATEASSIGN)}$$

When provided with a contract reference and a state variable identifier, STATEREAD returns the value of the specified variable. Similarly, STATEASSIGN facilitates the modification of this value.

4.7.3 Mappings

$$\text{(MAPREAD)}\quad \frac{}{\langle \beta, \sigma, M[v1] \rangle \rightarrow \langle \beta, \sigma, M(v1) \rangle}$$

$$\text{(MAPASSIGN)}\quad \frac{M' = M \setminus \{(v1, M(v1))\} \cup \{(v1, v2)\}}{\langle \beta, \sigma, M[v1 \mapsto v2] \rangle \rightarrow \langle \beta, \sigma, M' \rangle}$$

Although Di Pirro claims that their MAPREAD rule is a total function, we disagree because if $v1$ has not been previously assigned a value, there is no insight on how to handle the situation.

4.7.4 Contract Instantiation

$$\text{(NEW-1)}\quad \frac{(c, a) \notin \texttt{Dom}(\beta) \qquad \texttt{sv}(C) = \widetilde{Ts} \qquad |\widetilde{v}| = |\widetilde{s}| \qquad \texttt{Top}(\sigma) \neq \emptyset}{\langle \beta, \sigma, \texttt{new}\ C.\texttt{value}(n)(\widetilde{v}) \rangle \rightarrow \langle \texttt{uptbal}(\beta, \texttt{Top}(\sigma), -n).[(c, a) \mapsto (C, \widetilde{s : v}, n)], \sigma, c \rangle}$$

$$\text{(NEW-2)}\quad \frac{(c, a) \notin \texttt{Dom}(\beta) \qquad \texttt{sv}(C) = \widetilde{Ts} \qquad |\widetilde{v}| = |\widetilde{s}| \qquad \texttt{Top}(\sigma) = \emptyset}{\langle \beta, \sigma, \texttt{new}\ C.\texttt{value}(n)(\widetilde{v}) \rangle \rightarrow \langle \beta.[(c, a) \mapsto (C, \widetilde{s : v}, n)], \sigma, c \rangle}$$

$$\text{(NEW-R)}\quad \frac{\texttt{uptbal}(\beta, \texttt{Top}(\sigma), -n) = \bot \qquad Top(\sigma) \neq \emptyset}{\langle \beta, \sigma, \texttt{new}\ C.\texttt{value}(n)(\widetilde{v}) \rangle \rightarrow \langle \beta, \sigma, \texttt{revert} \rangle}$$

NEW rules deploy a new contract on the blockchain. To ensure consistency with the definition of K, where the body consists solely of a sequence of assignments, the number of typed values provided as input must match the number of state variables in contract C. This differs slightly from Solidity's capabilities, as it restricts explicit references to msg.sender or msg.value and initializing state variables with different types. In FS+, a strict one-to-one correspondence between parameters and state variables is required.

The NEW-2 rule increases liquidity by allowing external sources to provide Wei without checking balances. In Ethereum, the balance of the contract posing as a sender is decreased instead. This rule also enables the creation of externally-owned accounts on their implementation.

4.7.5 Balance and Address

$$\text{(BALANCE)}\quad \frac{\beta(a) = (C, \widetilde{sv}, n)}{\langle \beta, \sigma, \texttt{balance}(a)\rangle \rightarrow \langle \beta, \sigma, n\rangle}$$

$$\text{(ADDRESS)}\quad \frac{\beta(c) = a}{\langle \beta, \sigma, \texttt{address}(c)\rangle \rightarrow \langle \beta, \sigma, a\rangle}$$

In Solidity, balance is a property of address values that returns the balance of the account referred to by each address. Here they claim that to avoid confusion with contract state variables, they chose to define balance as an expression of the language. Furthermore, in Solidity, an implicit cast is applied whenever a contract reference is used instead of an address value. To model such behavior in FS, which does not have any implicit casts, they defined an explicit global function called address that extracts the address corresponding to a given reference.

4.7.6 Cast

$$\text{(CONTRRETR)}\quad \frac{\exists\, c.\beta(c, a) = C \qquad C <: D}{\langle \beta, \sigma, D(a)\rangle \rightarrow \langle \beta, \sigma, c\rangle}$$

$$\text{(CONTRRETR-R)}\quad \frac{\exists\, c.\beta(c, a) = C \qquad C \not<: D}{\langle \beta, \sigma, D(a)\rangle \rightarrow \langle \beta, \sigma, \texttt{revert}\rangle}$$

These rules apply to the operation of casting an address to a contract. If there is a contract c with the associated address a on the blockchain, we check if the contract type C is equal to or inherits from D at some point.

These rules are applied to the operation which we can cast an address to a contract. If there is in the blockchain a contract c with the associated address a we simply check if the contract type C is equal or inherits from D at some point.

4.7.7 Money Transfer

$$\text{(TRANSFER)}\quad \frac{\beta(a) = (C, sv, n) \qquad \texttt{fbody}(C, fb, \{\}) = (\{\}, e) \qquad \beta' = \texttt{uptbal}(\texttt{uptbal}(\beta, \texttt{Top}(\sigma), -n), a, n)}{\langle \beta, \sigma, a.\texttt{transfer}(n)\rangle \rightarrow \langle \beta', \sigma * a, e\{\texttt{this} := c, \texttt{msg.sender} := \texttt{Top}(\sigma), \texttt{msg.value} := n\}\rangle}$$

$$\text{(TRANSFER-R)}\quad \frac{\texttt{uptbal}(\beta, \texttt{Top}(\sigma), -n) = \bot}{\langle \beta, \sigma, a.\texttt{transfer}(n)\rangle \rightarrow \langle \beta, \sigma, \texttt{revert}\rangle}$$

These rules model the transfer function in Solidity as it is. If contract executing this function has sufficient balance to transfer n to a, then it is executed successfully and the contract with associated with address a executes its fallback function. Otherwise the transaction is reverted.

4.7.8 Function Calls

$$\text{(CALL)}\quad \frac{\begin{array}{c}\exists\, a.\beta((c,a)) = (C, \widetilde{sv}, n) \qquad \texttt{fbody}(C, f, \widetilde{v}) = (\widetilde{x}, e) \\ \widetilde{x} \notin Dom(\beta) \qquad \beta' = \texttt{uptbal}(\texttt{uptbal}(\beta, \texttt{Top}(\sigma), -n), a, n) \cdot [\widetilde{x} \mapsto \widetilde{v}] \\ e_s = e\{\texttt{this} := c, \texttt{msg.sender} := \texttt{Top}(\sigma), \texttt{msg.value} := n\}\end{array}}{\langle \beta, \sigma, c.f.\texttt{value}(n)(\widetilde{v})\rangle \rightarrow \langle \beta', \sigma \cdot a, e_s\rangle}$$

$$\text{(CALL-R)}\quad \frac{\texttt{uptbal}(\beta, \texttt{Top}(\sigma), -n) = \bot}{\langle \beta, \sigma, c.f.\texttt{value}(n)(\widetilde{v})\rangle \rightarrow \langle \beta, \sigma, \texttt{revert}\rangle}$$

$$\text{(CALLTOPLEVEL)}\quad \frac{\begin{array}{c}\exists\, a.\beta((c,a)) = (C, \widetilde{sv}, n) \qquad \texttt{fbody}(C, f, \widetilde{v}) = (\widetilde{x}, e) \\ \widetilde{x} \notin Dom(\beta) \qquad \beta' = \texttt{uptbal}(\texttt{uptbal}(\beta_w, a', -n), a, n) \cdot [\widetilde{x} \mapsto \widetilde{v}] \\ e_s = e\{\texttt{this} := c, \texttt{msg.sender} := a', \texttt{msg.value} := n\} \qquad \texttt{Top}(\sigma) = \emptyset\end{array}}{\langle \beta, \sigma, c.f.\texttt{value}(n).\texttt{sender}(a')(\widetilde{v}); e'\rangle \rightarrow \langle \beta', \sigma \cdot a, e_s; e'\rangle}$$

$$\text{(CALLTOPLEVEL-R)}\quad \frac{\texttt{uptbal}(\beta, a', -n) = \bot \qquad \texttt{Top}(\sigma) = \emptyset}{\langle \beta, \sigma, c.f.\texttt{value}(n).\texttt{sender}(a')(\widetilde{v}); e'\rangle \rightarrow \langle \beta', \sigma \cdot a, \texttt{revert}; e'\rangle}$$

In our interpretation and comparison with Solidity's state of the art, CALLTOPLEVEL is used to initiate a transaction that is always triggered by an externally-owned account, while CALL is used when a contract wants to invoke a function of another contract. Both are quite similar, and you can pass a number n posing as the value of Wei that the other party will receive. However, if the initiator does not have enough funds to cover the amount, the transaction will fail.

$$\text{(RETURN)}\quad \frac{}{\langle \beta, \sigma \cdot a, \texttt{return}\, v \rangle \to \langle \beta, \sigma, v \rangle}$$

$$\text{(RETURN-R)}\quad \frac{}{\langle \beta, \sigma \cdot a, \texttt{return revert} \rangle \to \langle \beta, \sigma, \texttt{revert} \rangle}$$

4.8 Type System

4.8.1 Axioms

$$\text{(REF)}\quad \frac{\Gamma, c : C, \Gamma' \vdash \langle\,\rangle}{\Gamma, c : C, \Gamma' \vdash c : C}$$

$$\text{(VAR)}\quad \frac{\Gamma, x : T, \Gamma' \vdash \langle\,\rangle}{\Gamma, x : T, \Gamma' \vdash x : T}$$

$$\text{(TRUE)}\quad \frac{\Gamma \vdash \langle\,\rangle}{\Gamma \vdash \texttt{true} : \texttt{bool}}$$

$$\text{(FALSE)}\quad \frac{\Gamma \vdash \langle\,\rangle}{\Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$\text{(UNIT)}\quad \frac{\Gamma \vdash \langle\,\rangle}{\Gamma \vdash u : \texttt{unit}}$$

$$\text{(REVERT)}\quad \frac{\Gamma \vdash \langle\,\rangle}{\Gamma \vdash \texttt{revert} : T}$$

$$\frac{n \in \mathbb{N}_0 \qquad \Gamma \vdash \langle\,\rangle}{\Gamma \vdash n : \texttt{uint}} \text{(UINT)}$$

$$\frac{\Gamma, a : \texttt{address}\langle C\rangle, \Gamma' \vdash \langle\,\rangle}{\Gamma, a : \texttt{address}\langle C\rangle, \Gamma' \vdash a : \texttt{address}\langle C\rangle} \text{(ADDRESS)}$$

The axioms have all a standard meaning. Here it is important to note that $\Gamma \vdash \texttt{revert} : T$ is well-typed regardless of T. We do not give a unique type to this expression, since it may be used in any context whatsoever. For instance, $\Gamma \vdash \texttt{if true then } 10 \texttt{ else revert} : \texttt{uint}$ and $\Gamma \vdash \texttt{if true then } u \texttt{ else revert} : \texttt{unit}$ are both valid expression in FS, but the former has type uint and the latter has type unit. If we want to omit the else branch then its value should always be u.

4.8.2 Standard Rules

These rules have a standard meaning are listed below. They are compose by the if expression, variable assignment and declaration as well as sequential composition and special operations balance and address.

$$\frac{\Gamma \vdash e : \texttt{address}}{\Gamma \vdash \texttt{balance}(e) : \texttt{uint}} \text{(BALANCE)}$$

$$\frac{\Gamma \vdash e : C}{\Gamma \vdash \texttt{address}(e) : \texttt{address}} \text{(ADDR)}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \texttt{return}\, e : T} \text{(RETURN)}$$

$$\frac{\Gamma \vdash e_1 : T_1' \qquad T_1' <: T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash T_1\ x = e_1; e_2 : T_2} \text{(LET)}$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T_e \qquad T_e <: T}{\Gamma \vdash x = e : T_e} \text{(ASSIGN)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : T_1 \qquad \Gamma \vdash e_3 : T_2 \qquad T_1 <: T \qquad T_2 <: T}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T} \text{(IF)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1; e_2 : T_2} \text{(SEQ)}$$

4.8.3 Mappings

The following rules provide a type to mapping values, as well as to read and write accesses on them. To read these rules it is important to mention that they consider $\Gamma \vdash \widetilde{v} : T$, represents $n \in \mathbb{N}$ distinct judgments where the type T is fixed and does not vary: $\Gamma \vdash v_i : T, 1 \leq i \leq n$

$$\frac{M = \{\widetilde{(k, v)}\} \qquad \Gamma \vdash \widetilde{k} : T_k \qquad \Gamma \vdash \widetilde{v} : T_v \qquad T_k <: T_1 \qquad T_v <: T_2}{\Gamma \vdash M : \texttt{mapping}(T_1 \Rightarrow T_2)} \text{(MAPPING)}$$

$$\frac{\Gamma \vdash e1 : \texttt{mapping}(T_1 \rightarrow T_2) \qquad \Gamma \vdash e2 : T_1' \qquad \Gamma \vdash e3 : T_2' \qquad T_1' <: T_1 \qquad T_2' <: T_2}{\Gamma \vdash e1[e2 \rightarrow e3] : \texttt{mapping}(T_1 \Rightarrow T_2)} \text{(MAPASSIGN)}$$

$$\frac{\Gamma \vdash \mathbf{e1} : \texttt{mapping}(\mathbf{T_1} \rightarrow \mathbf{T_2}) \qquad \Gamma \vdash e2 : T_1' \qquad T_1' <: T_1}{\Gamma \vdash e1[e2] : T_2} \text{(MAPREAD)}$$

4.8.4 Contract Instantiation and Access

The following rules are about the deploy of a new contract as well as the access (read/write) to its state variables.

(NEW)
$$\frac{\mathtt{sv}(C) = \widetilde{Ts} \qquad \Gamma \vdash \widetilde{e} : \widetilde{T'} \qquad \widetilde{T'} <: \widetilde{T} \qquad |\widetilde{e}| = |\widetilde{s}| \qquad \Gamma \vdash e' : \mathtt{uint}}{\Gamma \vdash \mathtt{new}\ C.\mathtt{value}(e')(\widetilde{e}) : C}$$

(STATEASSIGN)
$$\frac{\Gamma \vdash e_1.s : T \qquad \Gamma \vdash e_2 : T' \qquad T' <: T}{\Gamma \vdash e_1.s = e_2 : T}$$

(STATEREAD)
$$\frac{\Gamma \vdash e : C \qquad \mathtt{sv}(C) = \widetilde{Ts} \qquad s_i \in s}{\Gamma \vdash e.s_i : T_i}$$

4.8.5 Casts and Money Transfers

(ContrRetr)
$$\frac{\Gamma \vdash e : \mathtt{address}\langle C\rangle \qquad C <: D}{\Gamma \vdash D(e) : D}$$

(Transfer)
$$\frac{\mathtt{ftype}(C, \mathtt{fb}) = \{\} \to \mathtt{unit} \qquad \begin{array}{c}\Gamma \vdash e_1 : \mathtt{address}\langle C\rangle \\ \Gamma \vdash e_2 : \mathtt{uint} \end{array} \qquad \Gamma \vdash \mathtt{this} : C' \qquad C' <: \mathtt{fsender}(C, \mathtt{fb})}{\Gamma \vdash e_1.\mathtt{transfer}(e_2) : \mathtt{unit}}$$

Starting by the first rule, they claim we now we have precise information about address e and the operation $D(e)$ is well-typed if and only if C is either equals D or a subclass of it. In other words, we can cast an address pointing to a subcontract of D to a reference of type D

In the second rule, we can notice that it is mandatory for the contract invoking the transfer function to inherit from a contract defining a fb function that has to be annotated with a contract C that is a supertype of our current contract. We disagree with this implementation because it is mandatory on our language to have a fb function without any annotation, i.e it can be called by every address in our environment including externally-owned accounts.

4.8.6 Functions

(CALLTOPLEVEL)
$$\frac{\mathtt{ftype}(C, f) = \widetilde{T_1} \to T_2 \qquad |\widetilde{e}| = |\widetilde{T_1}| \qquad \begin{array}{c}\boldsymbol{\Gamma \vdash \mathbf{e_1} : \mathbf{C}} \\ \Gamma \vdash e_3 : \mathtt{address}\langle C'\rangle \end{array} \qquad \begin{array}{c}\Gamma \vdash e_2 : \mathtt{uint} \\ C' <: \mathtt{fsender}(C, \mathtt{f}) \end{array}}{\Gamma \vdash e_1.f.\mathtt{value}(e_2).\mathtt{sender}(e_3)(\widetilde{e}) : T_2}$$

(CALL)

$$\frac{\texttt{ftype}(C, f) = \widetilde{T_1} \to T_2 \qquad |\widetilde{e}| = |\widetilde{T_1}| \qquad \Gamma \vdash \mathbf{e_1} : \mathbf{C} \qquad \Gamma \vdash e_2 : \texttt{uint} \qquad \Gamma \vdash \texttt{this} : C' \qquad C' <: \texttt{fsender}(C, \texttt{f})}{\Gamma \vdash e_1.f.\texttt{value}(e_2)(\widetilde{e}) : T_2}$$

(FUN)

$$\frac{\mathbf{\Gamma \vdash e_1 : C} \qquad \texttt{ftype}(C, f) = \widetilde{T_1} \to T_2 \qquad \Gamma \vdash \texttt{this} : C' \qquad C' <: \texttt{fsender}(C, \texttt{f})}{\Gamma \vdash c.f : \widetilde{T_1} \to T_2}$$

(CALLVALUE)

$$\frac{\Gamma \vdash \widetilde{T_1} \to T_2 \qquad \Gamma \vdash e_2 : \texttt{uint} \qquad \Gamma \vdash \widetilde{e_1} : \widetilde{T_1'} \qquad \widetilde{T_1'} <: T_1 \qquad |\widetilde{e}| = \widetilde{T_1}}{\Gamma \vdash e_1.\texttt{value}(e_2)(\widetilde{e}) : T_2}$$

In these rules, it is worth mentioning that since every function has a restricted msg.sender type value, we need to ensure that the contract calling the function f is equal to or a subclass of the value we obtain from the annotation. However, we disagree with restricting it on CALLTOPLEVEL. Our argument is that this constructor is only intended to be called by externally-owned accounts (EOAs), i.e., users initiating a transaction and not by contracts. Therefore, restricting it is pointless.

5

FS 2.0: Revised Operational Semantics

In this chapter, we introduce the revised operational semantics for the programming language, building upon the original formalization by Di Pirro and Silvia Crafa. Key modifications include: integration of multiple inheritance, clarification of ambiguous definitions identified during our study and align more closely with the actual behavior of the current Solidity version (0.8.*).

In our notation going forward, we will employ $\widetilde{X}$ to denote a sequence of values, and $\widetilde{\widetilde{X}}$ to signify a set comprising sequences of values, encapsulating the concept of a sequence of sequences.

5.1 Grammar

(*Contract decl.*)	SC	::=	$\texttt{contract}\ C\ is\ \widetilde{D}\ \{(\widetilde{T\ s})\ K\ \widetilde{F}\}$
(*Constructor decl.*)	K	::=	$constructor\ \widetilde{C(\widetilde{e})}\ ((\widetilde{T\ x}))(e))$
(*Function decl.*)	F	::=	$T\ f\ ((\widetilde{T\ x}))\{\texttt{return}\ e\}$ | $\texttt{unit}\ fb()\ \{\texttt{return}\ e\}$
(*Expression*)	e	::=	$v \mid x \mid b \mid \texttt{this.s} \mid \texttt{this.}f(\widetilde{e}) \mid$ $\texttt{msg.sender} \mid \texttt{msg.value} \mid$ $\texttt{address}(e) \mid e.s \mid e.\texttt{transfer}(e)$ $\texttt{new}\ C.\texttt{value}(e)((\widetilde{e})) \mid$ $e; e \mid T\ x = e \mid x = e \mid e.s = e \mid$ $e[e] \mid e[e \rightarrow e] \mid e.\texttt{value}(e)(\widetilde{e}) \mid$ $e.f.\texttt{value}(e)(\widetilde{e}) \mid \texttt{revert} \mid$ $e.f.\texttt{value}(e).\texttt{sender}(e)(\widetilde{e}) \mid$ $\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
(*Values*)	v	::=	$\texttt{true} \mid \texttt{false} \mid n \mid a \mid \texttt{u} \mid M \mid$ c
(*Types*)	T	::=	$\widetilde{T} \rightarrow T \mid \texttt{bool} \mid \texttt{uint} \mid \texttt{address} \mid$ $\texttt{unit} \mid \texttt{mapping}(T \Rightarrow T) \mid C$

Figure 5.1: Syntax of Featherweight Solidity 2.0

Like the original formalization, the grammar of Featherweight Solidity 2.0 only contains a subset of Solidity. Looping primitives, arrays, structs, modifiers and events are removed from our syntax to simplify the formal system and associated proofs and remove extra complexity from a proof-of-concept. Nevertheless, conditional expressions are included, we allow multiple inheritance and one can also simulate loops using recursion. We also do not take into account the different types of visibility (represented in Solidity with external, internal, public, and private) and we removed pure and view from function declaration. Lastly, we consider every function as payable, and we do not model gas. Next we explain in detail the user syntax.

5.1.1 Contract Definition

We write `contract` C *is* $\widetilde{D}$ $\{(\widetilde{T\ s})\ K\ \widetilde{F}\}$ to denote a smart contract declaration. It indicates that contract C contains a sequence, possibly empty, $D_1 ... D_n$, representing a possible contract extension, another sequence possible empty of typed variables, denoted by $\widetilde{Ts}$, a mandatory constructor declaration and a set of functions ($\widetilde{F}$), also can be empty. However if a contract does not have at least a fallback function (`unit` $fb()\{$`return` $e\}$) it will not be able to receive ether.

5.1.2 Constructor Declaration

Our implementation doesn't enforce to initialize all state variables here, bringing more flexibility when declaring constructors. Thereafter we are much more liberal here and treat constructor essentially as a special function, just like it currently works in Solidity. Note that if a contract has super contracts inheriting from, their constructors should be also called here in place of $\widetilde{C(\widetilde{e})}$.

5.2 Configuration

$$
\begin{array}{llll}
(\textit{Blockchain}) & \beta & ::= & \emptyset \mid \beta \cdot [x \mapsto v] \mid \\
 & & & \beta \cdot [(c, a) \mapsto (C, \widetilde{s : v}, n)] \mid \\
 & & & \beta \cdot [a \mapsto n] \\
(\textit{Call Stack}) & \sigma & ::= & \varepsilon \mid \sigma \cdot a \\
(\textit{Contract Table}) & CT & ::= & \emptyset \mid CT \cdot [C \mapsto SC]
\end{array}
$$

Figure 5.2: Environments of Featherweight Solidity 2.0

Here we make two changes to Di Pirro's original formalization. One of those being, instead of having a call stack that contains an initial copy of the blockchain and a stack of addresses, we separate these two concepts and our call stack becomes just the stack of addresses. In our definition whenever we refer to β_i in the rules, we are referring to a copy

of the blockchain before the first call. Another change we make is having a blockchain composed by three productions. The first two are identical, a map between variables and their values and a mapping between a key comprised with contract reference and an address to a tuple with contract information (name, state variables and balance). The third formation is a mapping between addresses and balances, representing externally-owned accounts.

5.3 Contract Example in FS 2.0

Before introducing our rules, we would like to introduce a smart contract example that we will use to some of the rules we will be presenting. This contract will also be used to exemplify type system rules and in chapter 8.

```
1  import "./NFTStorage.sol";
2  contract Game is Context{
3      mapping(address => NFTStorage) stores;
4      constructor() Context(){}
5      function fb() {}
6      function createStore() returns (address) {
7          NFTStorage store = new NFTStorage{value: 0}();
8          this.stores = (this.stores[address(store)] = store);
9          return address(store);
10     }
11     function addExternalStore(address<NFTStorage> store) {
12         this.stores = (this.stores[store] = NFTStorage(store));
13     }
14     function setNFTPrice(address<NFTStorage> store, uint price) {
15         NFTStorage storeInstance = this.stores[store];
16         storeInstance.setNFTPrice{value: 0}(price);
17     }
18     function createNFT(address<NFTStorage> store, address to) {
19         NFTStorage storeInstance = this.stores[store];
20         storeInstance.createNFT{value: 0}(to);
21     }
22     function buyNFT(address<NFTStorage> store) {
23         NFTStorage storeInstance = this.stores[store];
24         storeInstance.buyNFT{value: msgvalue}();
25     }
26     function transferNFT(address<NFTStorage> store, uint tokenId, address
           from, address to) {
27         address snder = this.msgSender();
28         NFTStorage storeInstance = this.stores[store];
29         storeInstance.transferNFT{value: 0}(tokenId, snder, to);
30     }
31     function destroyNFT(address<NFTStorage> store, uint tokenId) {
32         address snder = this.msgSender();
33         NFTStorage storeInstance = this.stores[store];
34         storeInstance.destroyNFT{value: 0}(snder, tokenId);
35     }
36 }
```

Figure 5.3: A smart contract written in our Solidity refined grammar.

This is a simple contract that serves as a proxy to interact with a set of NFT stores, being a high-level API to interact them. Each externally-owned account can invoke this transactions, to create a new store, add an existing external store, set NFT prices for a store, create NFTs and so one.

We can see that our syntax is very similar with the one Solidity has, except we allow declaring a subset of type address, as we can see in function signatures. With this refined syntax, we can detect before executing each function, if the address passed is indeed

pointing to the `NFTStorage` contract specification. Thus passing an address belonging to an externally-owned account, will result in an error, before executing the function.

5.4 Auxiliary Predicates

During this chapter, we will be referencing some auxiliary methods and predicates, to simplify our rules presentation. In this section we will be providing and explaining each one.

5.4.1 Default Values

$$\texttt{defaultvalue} : \mathbf{T} \rightarrow \mathbf{V}$$

$$\texttt{defaultvalue}(T) = \begin{cases} 0 & T = \texttt{uint} \\ \texttt{false} & T = \texttt{bool} \\ \texttt{0x0} & T = \texttt{address} \\ \texttt{Null} & T = \texttt{C} \\ [\,] & T = \texttt{mapping}(T \Rightarrow T) \\ \texttt{unknown} & T = \widetilde{T} \rightarrow T \end{cases}$$

$$\texttt{init} : \mathbf{T} \times \mathbf{S} \rightarrow \mathbf{S} \times \mathbf{V}$$

$$\texttt{init}(Ts) = [s \mapsto \texttt{defaultvalue}(T)]$$

Our formalization does not enforce one to initialize state variables. Like Solidity if the user does not initialize it through constructor, we will be assuming a default value for each one. This auxiliary function is intended to assign default values to the state variables of a contract when it is initialized, ensuring that they always have values even when none are assigned in the constructor's body.

The function $init(\widetilde{Ts})$ is a sequence of functions init(Ts), where $\widetilde{Ts}$ represents a list of tuples with the type and the name of a certain state variable of a contract and Ts represents each tuple. Moreover $\widetilde{SC \mapsto e}$ denotes a mapping like $[C1 \mapsto \widetilde{e1}; \ldots; Cn \mapsto \widetilde{en}]$

For the concrete contract example, this is applied to the state variable **stores**. This variable is not initialized in the constructor with a value and so will fallback to the value defined by this function, which is simply the empty map.

5.4.2 Multiple Inheritance

The implementation of multiple inheritance in our system adheres to a set of fundamental rules. As previously mentioned, we have adopted the C3 linearization algorithm as the cornerstone of our approach to determine the hierarchy order for each contract and effectively resolve the diamond problem.

The diamond problem is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C.

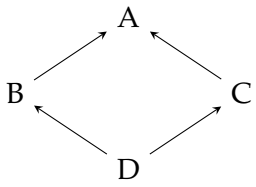

Figure 5.4: The diamond problem

As illustrated in the figure above, the graph have two distinct topological sort orders: [D;C;B;A] and [D;B;C;A]. In the context of programming languages and software design, it is imperative to establish consistency in choosing a specific order to resolve such ambiguities.

To incorporate C3 Linearization, as an algorithm to solve contract's dependencies we adapted an existing generic implementation to suit our project [0].

$$A : \mathbf{C} \times \widetilde{A}$$

$$\texttt{remove} : \widetilde{A} \times \mathbf{C} \to \widetilde{A}$$

(Remove)

$$\frac{\widetilde{L} = [L_1; \ldots; L_n] \qquad SC = \texttt{contract}\ C\ is\ \widetilde{D}\ \{(\widetilde{T\ s});\ K\ \widetilde{F}\} \qquad \widetilde{L_1} = [L_1 \setminus \{SC\}; \ldots; L_n \setminus \{SC\}]}{\texttt{remove}(\widetilde{L}, SC) = \widetilde{L_1}}$$

$$\texttt{merge} : \widetilde{\widetilde{A}} \to \widetilde{\mathbf{C}}$$

$$Let\ SC = \texttt{contract}\ C\ is\ \widetilde{D}\ \{(\widetilde{T\ s});\ K\ \widetilde{F}\}$$

$$\texttt{merge}(\widetilde{L}) = \begin{cases} SC \cdot (\texttt{merge}(\texttt{remove}(\widetilde{L}, SC))) & \text{if } SC \in L_k, L_k \in \widetilde{L}, SC = \texttt{head}(L_k), SC \notin \texttt{tail}(\widetilde{L}) \\ \texttt{fail} & \text{otherwise} \end{cases}$$

$$\texttt{c3linearization} : \mathcal{SC} \to \widetilde{C}$$

$$Let\ SC = \texttt{contract}\ C\ is\ \widetilde{D}\ \{(\widetilde{T\ s});\ K\ \widetilde{F}\}$$

$$\texttt{c3linearization}(SC) = \begin{cases} [C] & \widetilde{D} = \emptyset \\ [C] \cdot \texttt{merge}(\texttt{c3linearization}(\widetilde{D})) & \texttt{otherwise} \end{cases}$$

The c3linearization(SC) function computes a contract's hierarchy based on its superclass relationships. The functions `head` and `tail` are elementary operations employed

to extract information from a set. Specifically, `head` retrieves the first element of the set, while `tail` gathers all elements except the first one.

In Chapter 8, we provide an illustrative example of its execution with a set of contracts.

5.4.3 Functions

This section presents the auxiliary methods used to decide which functions our contract will execute, based on its hierarchy.

$$\texttt{name} : \mathcal{F} \rightarrow f$$

(Get Function name)

$$\frac{F = T\ f\ ((\widetilde{T\ x}))\{\texttt{e}\}}{\texttt{name}(F) = f}$$

This is a trivial function to retrieve a function name. For our illustrative contract possible return values are: fb, createStore, addExternalStore, setNFTPrice, createNFT, buyNFT, transferNFT and destroyNFT.

$$\texttt{functions} : C \rightarrow \widetilde{\mathcal{F}}$$

(Retrieve Contract Functions Map)

$$\frac{CT(C) = \texttt{contract}\ C\ is\ \widetilde{D}\ [D_1 \mapsto \widetilde{e_1}; \ldots; D_n \mapsto \widetilde{e_n}]\{(\widetilde{T\ s});\ K\ F_1; \ldots; F_n\} \qquad n \geq 0}{\texttt{functions}(C) = [\texttt{name}(F_1) \mapsto F_1; \ldots; \texttt{name}(F_n) \mapsto F_n]}$$

This function computes a map for each contract, where the key is the name of the function and the value is its declaration. In the illustrative example we would get the following map:

$$[\texttt{fb} \mapsto \texttt{unit}; \texttt{createStore} \mapsto e_1; \ldots; \texttt{destroyNFT} \mapsto e_n]$$

where e_1 and e_n represent the expressions of each function body.

Now for this example suppose that the Game contract inherits from a contract Context that follows this specification:

```
contract Context{
    uint balance;
    constructor() Context(){}
    function fb() {}
    function createStore() returns (address) {
       ....
    }
    function newFunction() {
      ....
    }
}
```

Figure 5.5: A dummy example for Context contract.

$$\texttt{cmap} : (\widetilde{\mathcal{F}} \times \widetilde{\mathcal{F}})^2 \to \widetilde{\mathcal{F}} \times \widetilde{\mathcal{F}}$$

(Concat Functions Map)

$$\frac{M = [fname_1 \mapsto f_1; \ldots; fname_n \mapsto f_n] \qquad fm = [fn_1 \mapsto fm_1; \ldots; fn_{n_1} \mapsto fm_{n_1}] \qquad n \geq 0 \qquad n_1 \geq 0 \qquad r = M \cup (fm \setminus \{\forall n_1 \in fm : fn_{n_1} \in M\}}{\texttt{cmap}(M, fm) = r}$$

This method concatenates a function to a map M if and only if there is not a function with the same name (key). This is useful when calculating our table lookup of each contract, that we will represent next.

$$\texttt{tlookup}(\widetilde{Cs}, \texttt{tbl}) : (\widetilde{SC}, [f \mapsto F]) \to [f \mapsto F] = \begin{cases} \texttt{tbl} & \widetilde{Cs} = [\,] \\ \texttt{tlookup}(\widetilde{Cs}', \texttt{cmap}(\texttt{tbl}, \texttt{functions}(C_i))) & \widetilde{Cs} = C_i \cdot \widetilde{Cs}' \end{cases}$$

This function computes the table lookup for a contract. It is simply a recursive function using the previous cmap function and it starts from the head of the computed linearization of classes. If we assume Context as illustrated in 5.5 and we try to apply this function to our Game contract we get the following result:

$$\texttt{tlookup}([\texttt{Game}; \texttt{Context}], []) =$$

$$[\texttt{fb} \mapsto \texttt{unit}; \ldots; \texttt{destroyNFT} \mapsto e_n; \texttt{newFunction} \mapsto e_{n2}]$$

Note that the only function we concatenate from Context contract is **newFunction**, because it does not exist in the previous iterated contracts.

(Retrieve Function Body)

$$\frac{\texttt{tlookup}(\widetilde{Cs}, [\,])[f] = T\ f\ ((\widetilde{T\ x}))\{\texttt{e}\} \qquad \widetilde{Cs} = \texttt{c3linearization}(CT(C)) \qquad |\widetilde{v}| = |\widetilde{x}|}{\texttt{fbody}(C, \widetilde{v}, f) = (\widetilde{x}, \texttt{e})}$$

(Retrieve Function Body)

$$\frac{\widetilde{Cs} = \texttt{c3linearization}(CT(C)) \qquad \texttt{tlookup}(\widetilde{Cs}, [\,])[f] = \bot \ \vee\ (\texttt{tlookup}(\widetilde{Cs}, [\,])[f] = T\ f\ ((\widetilde{T\ x}))\{\texttt{e}\} \ \wedge\ |\widetilde{v}| \neq |\widetilde{x}|)}{\texttt{fbody}(C, \widetilde{v}, f) = (\{\}, \texttt{return revert})}$$

This function retrieves a function body and its arguments from a function table. It receives a contract, a set of values and a function name. Imagine that we apply this function to Game contract, with $f = \texttt{addExternalStore}, \widetilde{v} = [\texttt{address}(0)]$ we get the following tuple as return: $([\texttt{store}], e_b)$ where e_b represents the body of our function.

(Retrieve function signature)
$$\frac{CT(C) = \texttt{contract}\ C\ is\ \widetilde{D}\ [\widetilde{SC \mapsto e}]\ \{(\widetilde{T\ s});\ K\ \widetilde{F}\} \qquad B\ f\ ((\widetilde{A\ x}))\{\texttt{e}\} \in \widetilde{F}}{\texttt{ftype}(C, f) = \widetilde{A} \to B}$$

(Sender maximum type lookup - 1)
$$\frac{CT(C) = \texttt{contract}\ C\ is\ \widetilde{D}\{(\widetilde{T\ s})\ K\ \widetilde{F}\} \qquad T\ f\ \langle C' \rangle\ ((\widetilde{T\ x}))\{\texttt{return}\ e\} \in \widetilde{F}}{\texttt{fsender}(C, f) = C'}$$

(Sender maximum type lookup - 2)
$$\frac{CT(C) = \texttt{contract}\ C\ is\ \widetilde{D}\{(\widetilde{T\ s})\ K\ \widetilde{F}\} \qquad T\ f((\widetilde{T\ x}))\{\texttt{return}\ e\} \in \widetilde{F}}{\texttt{fsender}(C, f) = \emptyset}$$

(Sender maximum type lookup - 3)
$$\frac{CT(C) = \texttt{contract}\ C\ is\ \widetilde{D}\{(\widetilde{T\ s})\ K\ \widetilde{F}\} \qquad T\ f\ \langle C' \rangle\ ((\widetilde{T\ x}))\{\texttt{return}\ e\} \notin \widetilde{F}}{\texttt{fsender}(C, f) = \bot}$$

These two functions will be used later on when we present our typing rules for expressions. The first one retrieves a tuple with a set of types for arguments and a type for a function body, while the latter retrieves for a function f, its required type for the implicit variable `msg.sender`. When the annotation is omitted from the function declaration, we return an empty set, as there will be no restrictions on the type.

5.4.4 State Variables

(state variable lookup)
$$\frac{CT(C) = \texttt{contract}\ C\ is\ \widetilde{D}\ [\widetilde{SC \mapsto e}]\ \{(\widetilde{T\ s});\ K\ \widetilde{F}\}}{sv(C) = \widetilde{T\ s}}$$

This is a state variable lookup, retrieving a contract's state variables.

$$\texttt{mkstate}(\widetilde{Cs}, \texttt{state}) = \begin{cases} \texttt{state} & \widetilde{Cs} = [\ ] \\ \texttt{mkstate}(\widetilde{Cs}', \texttt{state} \cdot \texttt{sv}(C_i)) & \widetilde{Cs} = C_i \cdot \widetilde{Cs}' \end{cases}$$

This function computes the state variables of a contract based on its linearization. It receives a list of contracts representing the contract's linearization and a state which will be an empty map in the beginning. Then it recursively pop the head of the list and concatenates each contract's state to our state. When the list of contracts is empty it returns our final state. Supposing that Context has the implementation we seen in 5.5, computing the state of contract Game we will get:

$$\texttt{mkstate}([\texttt{Game};\texttt{Context}],[]) =$$

$$[(\texttt{mapping}(\texttt{address} \Rightarrow \texttt{NFTStorage}),\texttt{stores});(\texttt{uint},\texttt{balance})]$$

5.4.5 Update Contract Balance

$$\texttt{uptbal} : BLOCKCHAIN \times ADDRESS \times BALANCE \rightarrow BLOCKCHAIN$$

$$\texttt{uptbal}(\beta, a, n) = \begin{cases} \beta[(c,a) \mapsto (C, \widetilde{s : v}, n' + n)] & \beta((c,a)) = (C, \widetilde{s : v}, n') \ \wedge \ n' + n \geq 0 \\ \beta[a \mapsto (n' + n)] & \texttt{if}\ \beta(a) = n' \ \wedge \ n' + n \geq 0 \\ \bot & \texttt{otherwise} \end{cases}$$

This function updates the balance of both EOAs or contracts. It receives a blockchain β, an address a and a number n. Note that n can either negative or positive. Also this is a function because in β both c and a are unique and so is the par formed by both, c representing a reference to a contract. Obviously an EOA does not have c, so it is only referenced by an address.

$$\texttt{Top} : \widetilde{ADDRESS} \rightarrow ADDRESS \cup \{\emptyset\}$$

$$\texttt{Top}(\sigma) = \begin{cases} a & \texttt{if}\ \sigma = \sigma' \cdot a \\ \emptyset & \texttt{if}\ \sigma = \epsilon \end{cases}$$

Returns the last element added to our stack of addresses, identifying the last address which initiated a transaction.

5.4.6 Constructors

(Retrieve Constructor Parameters)

$$\frac{CT(C) = \texttt{contract}\ C\ is\ \widetilde{D}\ [D_1 \mapsto \widetilde{e_1}; \ldots; D_1 \mapsto \widetilde{e_n}]\{(\widetilde{T\ s});\ K\ F_1; \ldots; F_n\} \qquad K = \texttt{constructor}\ ((\widetilde{T\ x}))(e))}{\texttt{cparams}(C) = \widetilde{x}}$$

Auxiliary function to that receives a contract and returns its constructor parameters.

(Retrieve Constructor Body)

$$\frac{CT(C) = \texttt{contract}\ C\ is\ \widetilde{D}\ [D_1 \mapsto \widetilde{e_1}; \ldots; D_n \mapsto \widetilde{e_n}]\{(\widetilde{T\ s});\ K\ F_1; \ldots; F_n\} \qquad K = \texttt{constructor}\ ((\widetilde{T\ x}))(e))}{\texttt{cbody}(C) = e}$$

Same as the previous one, however this function returns the body of the constructor.

(Retrieve Super Contracts Arguments Map)

$$\frac{CT(C) = \texttt{contract}\ C\ is\ \widetilde{D}\ [D_1 \mapsto \widetilde{e_1}; \ldots; D_n \mapsto \widetilde{e_n}]\{(\widetilde{T\ s});\ K\ F_1; \ldots; F_n\} \qquad n \geq 1}{\texttt{cargs}(C) = [D_1 \mapsto [\texttt{cparams}(D_1) \mapsto \widetilde{e_1}]; \ldots; D_n \mapsto [\texttt{cparams}(D_n) \mapsto \widetilde{e_n}]]}$$

This function computes a map where it maps each super contract to a map between its constructor parameters and values each one is going to get. It receives a contract definition as input.

$$\texttt{mkcvars}(\widetilde{Cs}, \widetilde{vs}, \widetilde{e}) = \begin{cases} \widetilde{vs} & \widetilde{Cs} = [\ ] \\ \texttt{mkcvars}(\widetilde{Cs}', (\widetilde{vs} \cdot [C_i \mapsto [\texttt{cparams}(C_i) \rightarrow \widetilde{e}]]) \cdot \texttt{cargs}(C_i), \widetilde{e}) & \widetilde{Cs} = C_i \cdot \widetilde{Cs}' \\ & \wedge i = 1 \\ \texttt{mkcvars}(\widetilde{Cs}', \widetilde{vs} \cdot \texttt{cargs}(C_i), \widetilde{e}) & \widetilde{Cs} = C_i \cdot \widetilde{Cs}' \\ & \wedge i \neq 1 \end{cases}$$

This auxiliary function takes as input a set of contract definitions, a mapping between constructor parameters and their corresponding values, and a set of values. It then computes a mapping between constructor parameters and values required for the initialization of each contract and its supercontracts.

(Execute Constructor)

$$\frac{\texttt{cparams}(C) = \widetilde{x} \qquad \texttt{cbody}(C) = e \qquad e_s = \langle \beta', e\{\widetilde{vs}[C]\}\rangle}{\texttt{exec}(C, \beta, \sigma, \widetilde{vs}) = e_s}$$

Simple function where constructor is executed and we get a tuple with a modified blockchain and the expression executed with the arguments passed as parameters. It receives a contract name, a blockchain, a sigma and a set of values (arguments).

5.5 Operational Semantics Rules

Now we will present our operational semantics rules. These rules are used by our PoC interpreter and its goal is to be as close as possible to Solidity.

5.5.1 Arithmetic and Boolean Operations

(ARIT/BOOL)

$$\frac{v = a \diamond b}{\langle \beta, \beta_i, \sigma, a \diamond b\rangle \rightarrow \langle \beta, \beta_i, \sigma, v\rangle}$$

All arithmetic and boolean operators follow the consensual trivial rule. it is important to note that we support the following arithmetic operations: plus, minus, division, multiplication, exponential and modulo. For boolean operations we support: negation, conjunction, disjunction, lesser (or equal), greater (or equal), equals and inequals.

Next, one can see how these rules should be applied. In this case, we demonstrate a simple addition operation, where we sum two values to obtain a new value, which represents the result.

$$\langle \beta, \beta, \emptyset, 1 + 1 \rangle \rightarrow \langle \beta, \beta, \emptyset, 2 \rangle$$

5.5.2 Variable Operations

$$\text{(VAR)} \quad \frac{}{\langle \beta, \beta_i, \sigma, x \rangle \rightarrow \langle \beta, \beta_i, \sigma, \beta(x) \rangle}$$

Accesses a variable and returns its value.

$$\text{(ASSIGN)} \quad \frac{}{\langle \beta, \beta_i, \sigma, x = v \rangle \rightarrow \langle \beta \cdot [x \mapsto v], \beta_i, \sigma, v \rangle}$$

Assigns a value to an existing variable.

$$\text{(LET)} \quad \frac{x \notin dom(\beta)}{\langle \beta, \beta_i, \sigma, T\ x = v; e \rangle \rightarrow \langle \beta \cdot [x \mapsto v], \beta_i, \sigma, v; e \rangle}$$

This rule allows developers to declare a new variable. it is important to highlight that, unlike Solidity, which allows for collision names between state variables and local variables, our approach is designed with a distinct advantage. We do allow for collisions, but the state variable is never shadowed. To access its value, users must employ the keyword **this**.

Next, we illustrate these rules with an example. Initially, we declare a variable x as a `uint`with a value of 3, using LET rule. Subsequently, we modify this value using the ASSIGN rule, and finally, we read the variable x from the blockchain, using VAR rule.

$$\langle \beta, \beta, \emptyset, \texttt{uint}\ x = 3; x = 2 \rangle \rightarrow \langle \beta \cdot [x \mapsto 3], \beta, \emptyset, x = 2 \rangle \rightarrow$$

$$\langle \beta \cdot [x \mapsto 2], \beta, \emptyset, 2 \rangle$$

$$\langle \beta \cdot [x \mapsto 2], \beta, \emptyset, x \rangle \rightarrow \langle \beta \cdot [x \mapsto 2], \beta, \emptyset, 2 \rangle$$

Now, we present the rules for reading and writing state variables.

$$\text{(STATEREAD)} \quad \frac{\exists\, a.\beta((c, a)) = (C, \widetilde{sv}, n) \qquad s \in \widetilde{s}}{\langle \beta, \beta_i, \sigma, c.s \rangle \rightarrow \langle \beta, \beta_i, \sigma, v_s \rangle}$$

(STATEASSIGN)

$$\frac{\exists\, a.\beta((c,a)) = (C, \widetilde{sv}, n) \qquad s \in \widetilde{s}}{\langle \beta, \beta_i, \sigma, c.s = v' \rangle \rightarrow \langle \beta \cdot [c.s \mapsto v'], \beta_i, \sigma, v' \rangle}$$

These rules lets us access a state variable or write a new value for it. Obviously if there is no such state variable s this should fail.

These rules allow us to access a state variable or write a new value to it. However, if there is no such state variable s, this operation should fail.

To illustrate these rules, we assume that we already have a contract in the blockchain. In this case, we use generic values to identify the contract, denoted by the pair containing values c and a, and C for the contract name. The examples are as follows:

$$\langle [(c,a) \mapsto (C, [x \mapsto 0], 0] \cdot [\texttt{this} \mapsto c], [(c,a) \mapsto (C, [x \mapsto 0], 0], \emptyset, \texttt{this}.x = 30 \rangle \rightarrow$$

$$\langle [(c,a) \mapsto (C, [x \mapsto 30], 0] \cdot [\texttt{this} \mapsto c], [(c,a) \mapsto (C, [x \mapsto 0], 0], \emptyset, 30 \rangle$$

$$\langle [(c,a) \mapsto (C, [x \mapsto 30], 0] \cdot [\texttt{this} \mapsto c], [(c,a) \mapsto (C, [x \mapsto 0], 0], \emptyset, \texttt{this}.x \rangle \rightarrow$$

$$\langle [(c,a) \mapsto (C, [x \mapsto 30], 0] \cdot [\texttt{this} \mapsto c], [(c,a) \mapsto (C, [x \mapsto 0], 0], \emptyset, 30 \rangle$$

Note how we initially have the state variable x with a value of 0. Using STATEASSIGN, we change the value to 30, and after that, we use STATEREAD to retrieve its value from the blockchain. In both operations, we make use of the variable **this** to access the contract's reference.

5.5.3 Constructor Rule

(NEW)

$$\frac{\begin{array}{c}(c,a) \notin Dom(\beta) \qquad \texttt{mkstate}(\widetilde{SC}, [\,]) = \widetilde{Ts} \qquad \texttt{c3linearization}(CT(C)) = \widetilde{SC} \\ \texttt{mkcvars}(\widetilde{SC}, [\,], \widetilde{v}) = \widetilde{vs} \qquad \texttt{exec}(\widetilde{SC}, \beta, \sigma, \widetilde{vs}) = \langle \beta', e_s \rangle \qquad Top(\sigma) \neq \emptyset\end{array}}{\langle \beta, \beta_i, \sigma, \texttt{new}\ C.\texttt{value}(n)(\widetilde{v}) \rangle \rightarrow \langle \texttt{uptbal}(\beta', \texttt{Top}(\sigma), -n).[(c,a) \mapsto (C, init(\widetilde{Ts}), n)].[\widetilde{x} \mapsto \widetilde{v}], \beta_i, \sigma, e_s; c \rangle}$$

(NEW-R)

$$\frac{uptbal(\beta, \texttt{Top}(\sigma), -n) = \bot \qquad Top(\sigma) \neq \emptyset}{\langle \beta, \beta_i, \sigma, new\ C.value(n)(\widetilde{v}) \rangle \rightarrow \langle \beta, \beta_i, \sigma, revert \rangle}$$

NEW rule deploys a new contract on the blockchain. In these rules, we made a modification to Di Pirro's formalization. Instead of requiring a one-to-one relation between constructor arguments and state variables, there is a freedom in these aspect. So constructor body and its arguments are evaluated as a function and the contract's state variables are

initially set to a default value. Our specification is identical to what Solidity does. NEW-R is the case if there is no balance on the EOA or contract deploying.

We should also note that in the original formalization there was a NEW-2 rule, which increased the blockchain liquidity and was used to model a creation of external-owned accounts (EOAs). However we dropped this rule since we have in mind that contracts and EOA are distinct and accounts in blockchain is just a mapping between their addresses and balances.

In Section 5.5.8, we provide a detailed example illustrating how this rule applies in our environment.

5.5.4 Conditional and Sequential Statements

(If-True)

$$\frac{}{\langle \beta, \beta_i, \sigma, \texttt{if true then } e1 \texttt{ else } e2 \rangle \rightarrow \langle \beta, \beta_i, \sigma, e1 \rangle}$$

(If-False)

$$\frac{}{\langle \beta, \beta_i, \sigma, \texttt{if false then } e1 \texttt{ else } e2 \rangle \rightarrow \langle \beta, \beta_i, \sigma, e2 \rangle}$$

These rules have the standard meaning. Important to note that if we want to omit either branch the omitted branch should take the value unit internally.

Here is an example how this rule simplifies:

$$\langle \beta, \beta, \emptyset, \texttt{if } "0x01" == "0x01" \texttt{ then } 2 \texttt{ else } 3 \rangle \rightarrow \langle \beta, \beta, \emptyset, \texttt{if true then } 2 \texttt{ else } 3 \rangle \rightarrow \langle \beta, \beta, \emptyset, 2 \rangle$$

(Seq-C)

$$\frac{\sigma = \varepsilon}{\langle \beta, \beta_i, \sigma, v; e \rangle \rightarrow \langle \beta, \beta_i, \sigma, e \rangle}$$

(SEQ-R)

$$\frac{\sigma = \varepsilon}{\langle \beta, \beta_i, \sigma, revert; e \rangle \rightarrow \langle \beta_i, \beta_i, \sigma, revert \rangle}$$

(SEQ)

$$\frac{\texttt{Top}(\sigma) = a}{\langle \beta, \beta_i, \sigma, v; e \rangle \rightarrow \langle \beta, \beta_i, \sigma, e \rangle}$$

These rules have to take into account different scenarios. First, when an expression like $v; e$ is found at the top level ($\texttt{Top}(\sigma) = \emptyset$)) then the last transaction was successful. In fact,

it evaluated to a value v, which can be discarded to proceed with the next transaction e. Furthermore, since evaluating to a value is equivalent to a commit, we apply the changes and proceed with β. This scenario is modeled by SEQ-C. Secondly, v; e might be at an inner level ($\texttt{Top}(\sigma) = a$). In this case (SEQ) we cannot apply any changes, since the transaction is not over yet. Hence, we just proceed with the evaluation of e, discarding v. Lastly, a revert may have been thrown during the evaluation, so instead of v; e the expression involving sequential composition is $\texttt{revert}; e$ (SEQ-R). In this case we do not further proceed in evaluating the other transactions and we replace β with β_i.

We make extensively use of these rules in our example in section 5.5.8.

5.5.5 Mappings

$$\frac{\exists\, a.\beta((c,a)) = (C, \widetilde{sv}, n)}{\langle \beta, \beta_i, \sigma, M[v1] \rangle \rightarrow \langle \beta, \beta_i, \sigma, M(v1) \rangle} \text{(MAPREAD)}$$

$$\frac{M[v1] = \bot}{\langle \beta, \beta_i, \sigma, M[v1] \rangle \rightarrow \langle \beta, \beta_i, \sigma, \texttt{defaultvalue}(T) \rangle} \text{(MAPREAD)}$$

$$\frac{M' = M \setminus \{(v1, M(v1))\} \cup \{(v1, v2)\}}{\langle \beta, \beta_i, \sigma, M[v1 \mapsto v2] \rangle \rightarrow \langle \beta, \beta_i, \sigma, M' \rangle} \text{(MAPASSIGN)}$$

In our formalization, users have the capability to define mappings, just like in Solidity. The initial two rules ensure that our mappings read function operates as a total function: in the worst-case scenario, attempting to access a mapping position should return a default value corresponding to its return type. The ASSIGN rule, on the other hand, serves to replace the value associated with a specific key. Unlike Di Pirro's original formalization, we align with Solidity's behavior. Consequently, when a key lacks a value, we return the default value corresponding to the value type.

Next, we exemplify how these rules work in our language. In the first example, we demonstrate that attempting to access the value of a key in an empty map always yields a value in return. In this case, we assume a map where keys and values are of type `uint`, and we return 0 as it is the default value for this type. In the second example, we illustrate that if we insert a key with a value into an empty map, the returned map contains this key. Lastly, using the map we obtained, we attempt to access the key and obtain the desired return: the value associated with the key in the map.

$$\langle \beta, \beta, \emptyset, [][1] \rangle \rightarrow \langle \beta, \beta, \emptyset, 0 \rangle$$

$$\langle \beta, \beta, \emptyset, [][1 \mapsto 2] \rangle \rightarrow \langle \beta, \beta, \emptyset, [1 \mapsto 2] \rangle$$

$$\langle \beta, \beta, \emptyset, [1 \mapsto 2][1] \rangle \rightarrow \langle \beta, \beta, \emptyset, 2 \rangle$$

5.5.6 Functions

$$\text{(BALANCE)} \quad \frac{\exists\, c.\beta((c,a)) = (C, \widetilde{sv}, n) \vee \beta(a) = n}{\langle \beta, \beta_i, \sigma, \texttt{balance}(a) \rangle \rightarrow \langle \beta, \beta_i, \sigma, n \rangle}$$

As in Solidity, our system includes an operation to retrieve a balance from an address. Above, we provide an operational semantic rule that models this behavior.

Next we present a pratical example of this rule, assuming a contract game deployed in our blockchain:

$$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)], \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)], \texttt{balance}(a_{\texttt{game}}) \rangle \rightarrow$$

$$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)], \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)], 200 \rangle$$

$$\text{(ADDRESS)} \quad \frac{\exists\, a.\beta((c,a)) = (C, \widetilde{sv}, n) \vee c = a}{\langle \beta, \beta_i, \sigma, \texttt{address}(c) \rangle \rightarrow \langle \beta, \beta_i, \sigma, a \rangle}$$

Similarly to the previous operation, this operation is also present in Solidity and it just casts a contract instance or EOA. We only can point to externally-owned accounts by their address, so in that case we cannot cast an instance of it, but an address, which does essentially nothing. Note that c can be either a contract instance or an address.

Now, let's consider a similar example to the one previously shown, but this time focusing on this operation:

$$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)], \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)], \emptyset, \texttt{address}(c_{\texttt{game}}) \rangle \rightarrow$$

$$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)], \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)], \emptyset, a_{\texttt{game}} \rangle$$

(TRANSFER)

$$\frac{\beta(a) = (C, sv, n) \qquad \texttt{fbody}(C, fb, \{\}) = (\{\}, e) \qquad \beta' = \texttt{uptbal}(\texttt{uptbal}(\beta, \texttt{Top}(\sigma), -n), a, n)}{\langle \beta, \beta_i, \sigma, a.\texttt{transfer}(n) \rangle \rightarrow \langle \beta', \beta_i, \sigma * a, e\{\texttt{this} := c, \texttt{msg.sender} := \texttt{Top}(\sigma), \texttt{msg.value} := n\} \rangle}$$

$$\frac{\text{(TRANSFER-R)} \quad uptbal(\beta, \texttt{Top}(\sigma), -n) = \bot}{\langle \beta, \beta_i, \sigma, v; e \rangle \rightarrow \langle \beta, \beta_i, \sigma, \texttt{revert} \rangle}$$

These rules model the behavior of the transfer function in Solidity. This function is callable through an address pointer and is used to transfer Ether from the contract to the address specified in the function call. The value of the transfer is passed as an argument n. Notably, we have a second rule that causes this function to fail if the contract's balance does not contain the required amount for the transfer

In the example below, we illustrate how this rule works for a transfer between two contracts. Assuming we have two contracts deployed in our blockchain: Game and Dummy. Game has a balance of 200, and Dummy has a balance of 0. If we invoke a transfer function from Game to Dummy with a value of 200, we will have Game as our address at the top of the stack σ. The expression will be reduced to the body of the fallback function of the Dummy contract ($e_{\texttt{fb}}$) with all the variables set: `this`, `msg.sender`, and `msg.value`. We also transfer 200 from Game to Dummy, leaving Game with a balance of 0 and Dummy with a balance of 200.

$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)] \cdot [(c_{\texttt{dummy}}, a_{\texttt{dummy}}) \mapsto (\texttt{Dummy}, s_2, 0)],$

$\beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)] \cdot [(c_{\texttt{dummy}}, a_{\texttt{dummy}}) \mapsto (\texttt{Dummy}, s_2, 0)], [a_{\texttt{game}}], a_{\texttt{dummy}}.\texttt{transfer}(200)\rangle \rightarrow$

$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 0)] \cdot [(c_{\texttt{dummy}}, a_{\texttt{dummy}}) \mapsto (\texttt{Dummy}, s_2, 200)] \cdot$

$[\texttt{this} \mapsto c_{\texttt{dummy}}; \texttt{msg.sender} \mapsto a_{\texttt{game}}; \texttt{msg.value} \mapsto 200],$

$\beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)] \cdot [(c_{\texttt{dummy}}, a_{\texttt{dummy}}) \mapsto (\texttt{Dummy}, s_2, 0)], [a_{\texttt{game}}], e_{\texttt{fb}}\rangle$

$$\frac{\text{(CONTRRETR)} \quad \exists\, c.\beta(c, a) = C \qquad \texttt{subtyping}(C, D, CT)}{\langle \beta, \beta_i, \sigma, D(a) \rangle \rightarrow \langle \beta, \beta_i, \sigma, c \rangle}$$

$$\frac{\text{(CONTRRETR-R)} \quad \exists\, c.\beta(c, a) = C \qquad \neg\texttt{subtyping}(C, D, CT)}{\langle \beta, \beta_i, \sigma, D(a) \rangle \rightarrow \langle \beta, \beta_i, \sigma, \texttt{revert} \rangle}$$

Since there may not be possible to know what type of address we are passing before hand (i.e in compile time), we need to have a premise to ensure that the address we are casting

to a contract is pointing to a contract C that is a subtype of D. We also do this verification in the typechecker and it saves us this computation before hand.

This `subtyping` will be introduce in section 7.3.2 and so we do not develop more here.

Considering the next example:

$$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)], \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)], \emptyset, \texttt{Game}(a_{\texttt{game}}) \rangle \rightarrow$$

$$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)], \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 200)], \emptyset, c_{\texttt{game}} \rangle$$

As we can see, we use the cast operation applied to $a_{\texttt{game}}$, which is the address reference to our deployed contract, to obtain the value $c_{\texttt{game}}$ - a reference to the contract associated with the address. Since our contract Game also inherits from the Context contract, we could also use Context to cast. However, using any other contract for casting, aside from these two, would have resulted in a revert.

(CALL)

$$\frac{\exists\, a.\beta((c,a)) = (C, \widetilde{sv}, n) \qquad \texttt{fbody}(C, f, \widetilde{v}) = (\widetilde{x}, e) \qquad \widetilde{x} \notin Dom(\beta) \qquad \beta' = \texttt{uptbal}(\texttt{uptbal}(\beta, \texttt{Top}(\sigma), -n), a, n) \cdot [\widetilde{x} \mapsto \widetilde{v}] \qquad e_s = e\{\texttt{this} := c, \texttt{msg.sender} := \texttt{Top}(\sigma), \texttt{msg.value} := n\}}{\langle \beta, \beta_i, \sigma, c.f.\texttt{value}(n)(\widetilde{v}) \rangle \rightarrow \langle \beta, \beta_i, \sigma \cdot a, e_s \rangle}$$

(CALL-R)

$$\frac{\texttt{uptbal}(\beta, \texttt{Top}(\sigma), -n) = \bot}{\langle \beta, \beta_i, \sigma, C(a) \rangle \rightarrow \langle \beta, \beta_i, \sigma, \texttt{revert} \rangle}$$

These rules enable one contract to call a function within another contract. To execute this operation, we require a contract reference c, a function name f, a value n to pass as the value (which will represent the special value msg.value inside the called function) and a set of arguments (potential empty). This function call can fail if and only if, the contract where we are executing this, does not have enough balance to pass the value n.

(CALLTOPLEVEL)

$$\frac{\exists\, a.\beta((c,a)) = (C, \widetilde{sv}, n) \qquad \texttt{fbody}(C, f, \widetilde{v}) = (\widetilde{x}, e) \qquad \widetilde{x} \notin Dom(\beta) \qquad \beta' = \texttt{uptbal}(\texttt{uptbal}(\beta_w, a', -n), a, n) \cdot [\widetilde{x} \mapsto \widetilde{v}] \qquad e_s = e\{\texttt{this} := c, \texttt{msg.sender} := a', \texttt{msg.value} := n\} \qquad \texttt{Top}(\sigma) = \emptyset}{\langle \beta, \beta_i, \sigma, c.f.\texttt{value}(n).\texttt{sender}(a')(\widetilde{v}); e' \rangle \rightarrow \langle \beta', \beta_i, \sigma \cdot a, e_s; e' \rangle}$$

(CALLTOPLEVEL-R)

$$\frac{\texttt{uptbal}(\beta, a', -n) = \bot \qquad \texttt{Top}(\sigma) = \emptyset}{\langle \beta, \beta_i, \sigma, c.f.\texttt{value}(n).\texttt{sender}(a')(\widetilde{v}); e' \rangle \rightarrow \langle \beta', \beta_i, \sigma \cdot a, \texttt{revert}; e' \rangle}$$

While this rule does not inherently exist in Solidity, it represents our and Di Pirro's solution for initiating transactions. This rule is designed to be invoked by a specific externally-owned account, enabling them to interact with our deployed contracts.

In our example in Section 5.5.8, we use CALLTOPLEVEL and one can see how it works practically. it is worth noting that the only distinction between CALLTOPLEVEL and CALL lies in the ability of CALLTOPLEVEL to specify the value of `msg.sender` as one of the arguments in the function call. In contrast, CALL does not allow for the setting of `msg.sender` directly, and its value is determined by invoking the Top function.

$$\text{(RETURN)}\quad \frac{}{\langle \beta, \beta_i, \sigma \cdot a, \texttt{return}\ v \rangle \rightarrow \langle \beta', \beta_i, \sigma, v \rangle}$$

$$\text{(RETURN-R)}\quad \frac{}{\langle \beta, \beta_i, \sigma \cdot a, \texttt{return revert} \rangle \rightarrow \langle \beta_i, \beta_i, \sigma, \texttt{revert} \rangle}$$

We have straightforward rules for returning a value: we can either return a value or trigger a revert. The key distinction lies in the fact that when we initiate a revert, we replace β with β_i, which represents the blockchain's original state before the transaction that was reverted was executed.

5.5.7 Computation Rules

$$\text{(REVERT)}\quad \frac{}{\langle \beta, \beta_i, \sigma, \texttt{revert} \rangle \rightarrow \langle \beta, \beta_i, \sigma, \texttt{revert} \rangle}$$

$$\text{(CONG)}\quad \frac{\langle \beta, \beta_i, \sigma, e \rangle \rightarrow \langle \beta', \beta_i, \sigma', e' \rangle}{\langle \beta, \beta_i, \sigma, E[e] \rangle \rightarrow \langle \beta', \beta_i, \sigma', E[e'] \rangle}$$

These two rules above define our computation rules. $E[e]$ represents the expression obtained by replacing the hole in context E by the expression e. Below (Figure 5.6) we formally define all evaluation contexts.

$$
\begin{array}{rcl}
E & ::= & [] \mid \texttt{balance}(E) \mid \texttt{address}(E) \mid E.s \mid \\
& & E.\texttt{transfer}(e) \mid a.\texttt{transfer}(E) \mid E; e \mid \\
& & \texttt{new}\ C.\texttt{value}(E)(\widetilde{e}) \mid \\
& & C(e) \mid E.f.\texttt{value}(e)(\widetilde{e}) \mid c.f.\texttt{value}(E)(\widetilde{e}) \mid \\
& & E.f.\texttt{value}(e).\texttt{sender}(e)(\widetilde{e}) \mid c.f.\texttt{value}(E).\texttt{sender}(e)(\widetilde{e}) \mid \\
& & c.f.\texttt{value}(n).\texttt{sender}(E)(\widetilde{e}) \mid \\
& & T\ x = E; e \mid x = E \mid E.s = e \mid c.s = E \mid E[e] \mid \\
& & M[E] \mid E[e \rightarrow e] \mid M[E \rightarrow e] \mid M[v \rightarrow E] \mid \\
& & \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{return}\ E
\end{array}
$$

Figure 5.6: Evaluation Contexts of Featherweight Solidity 2.0

5.5.8 Operational Semantics by Example

In this section we will show some examples of our operational semantics rules are applied. Let's assume that our initial state for all examples is:

$$\langle [0x01 \rightarrow 2000; 0x02 \rightarrow 2000], [0x01 \rightarrow 2000; 0x02 \rightarrow 2000], \emptyset, e \rangle$$

As one can see, our blockchain already have two EOAs, one with address 0x01 and another with 0x02 and both with 2000 balance.

To simplify and adjust to our needs we will redefine our Game contract

```
contract Game {
    uint totalNfts;
    mapping(uint => address) nfts;
    uint price;
    address owner;
    constructor() {
        this.owner = msgsender;
    }
    function fb() {}
    function setNFTPrice(uint price) {
        if(msgsender == this.owner) {
            this.price = price;
        }
        else {
            revert;
        }
    }
    function buyNFT() returns (uint) {
        if(msgvalue >= price){
            this.nfts = (this.nfts[this.totalNfts] = msgsender);
            this.totalNfts = this.totalNfts + 1;
            return this.totalNfts;
        }
        else {
            revert;
        }
    }
    function getOwner(uint nft) returns (address) {
        return this.nfts[nft];
    }
}
```

Figure 5.7: A simpler NFT Game contract

Initially, the contract is deployed by the address "$0x01$". Subsequently, this address will proceed to set the price for each NFT at 200 by invoking setNFTPrice function. Following this, address "$0x02$" will execute the buyNFT function to acquire an NFT. Finally, it will invoke the getOwner function to confirm its ownership of the recently purchased token.

$e :=$ `Game game = new Game();`

`game.setNFTPrice.value(0).sender("0x01")(200);`

`uint num = game.buyNFT.value(200).sender("0x02")();`

`game.getOwner.value(0).sender("0x02")(num)`

Next we will exemplify the application of the previously mentioned rules and how they reduce our expressions. It is important to note that for the sake of simplicity, we will omit each variable after its initial appearance, assuming that they persist in β after their omission.

We start by applying together both the NEW and LET rule. Applying these rules we deploy the contract to the blockchain and store its reference on a variable game. Then applying the SEQ-C rule we commit our blockchain, i.e replace the blockchain replica by the actual blockchain. In the end we apply the CALL rule, simplifying our expression with its arguments values.

$\langle \beta_a, \beta_a, \emptyset, \texttt{Game game = new Game.value}(0)(); e' \rangle \rightarrow$

$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 0)], \beta_a, \emptyset, \texttt{Game game} = c_{\texttt{game}}; e' \rangle \rightarrow$ `SEQ-C`

$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 0)] \cdot [\texttt{game} \mapsto c_{\texttt{game}}], \beta_a \cdot [(c_{\texttt{game}}, (a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 0)], \emptyset,$

`game.setNFTPrice.value(0).sender("0x01")(200); e'⟩` $\rightarrow$ `CALLTOPLEVEL`

After we have our CALLTOPLEVEL rule applied we add to our variable environment two keys: msgsender and price and its corresponding values. Doing this helps us replacing in the expression wherever these values appear. Eventually we commit the result applying the SEQ-C rule again. After all these rules are applied we have our contract's state transitioned from s_1 to s_2 and we get the third expression to evaluate.

$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 0)] \cdot [\texttt{msgsender} \mapsto \text{"}0x01\text{"}; \texttt{this} \mapsto c_{\texttt{game}}; \texttt{msg.value} \mapsto 0; \texttt{price} \mapsto 200],$

$\beta_a \cdot [(c_{\texttt{game}}, (a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 0)], [a_{\texttt{game}}],$

$\texttt{if}(\texttt{msgsender} == \texttt{this.owner})\ \texttt{then}\ \texttt{this.price} = \texttt{price}\ \texttt{else}\ \texttt{revert}; e' \rangle \rightarrow$

$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 0)], \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 0)], [a_{\texttt{game}}],$

$\texttt{if}(\text{"}0x01\text{"} == \text{"}0x01)\ \texttt{then}\ \texttt{this.price} = 200\ \texttt{else}\ \texttt{revert}; e' \rangle \rightarrow$

$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 0)], \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 0)], [a_{\texttt{game}}]$

$\texttt{if}(\texttt{true})\ \texttt{then}\ \texttt{this.price} = 200\ \texttt{else}\ \texttt{revert}; e' \rangle \rightarrow$

$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_2, 0)], \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_1, 0)], [a_{\texttt{game}}],$

$\texttt{this.price} = 200; e' \rangle \rightarrow \texttt{SEQ-C}$

$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_2, 0)], \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_2, 0)], \emptyset,$

$\texttt{uint num} = \texttt{game.buyNFT.value}(200).\texttt{sender}(\text{"}0x02\text{"})(); e' \rangle \rightarrow \texttt{CALLTOPLEVEL}$

In the next step, we will assess the expression in which address "0x02" purchases an NFT and stores the token ID in a variable num. We will exclude the evaluation of the function body for simplicity, as it closely resembles previous instances. Subsequently, by utilizing this variable as the argument for the getOwner function, we obtain the return value, which is the address "0x02".

$\langle \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_2, 0)], \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_2, 0)] \cdot [\texttt{msgsender} \mapsto \text{"}0x02\text{"};$

$\texttt{this} \mapsto c_{\texttt{game}}; \texttt{msg.value} \mapsto 200], [a_{\texttt{game}}],$

$\texttt{uint num} = \texttt{game.buyNFT.value}(200).\texttt{sender}(\text{"}0x02\text{"})(); e' \rangle \rightarrow^*$

$\langle \beta_b \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_3, 0)], \beta_a \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_2, 0)], [a_{\texttt{game}}], \texttt{uint num} = 0; e' \rangle \rightarrow$

$\langle \beta_b \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_3, 200)], \beta_b \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_3, 200)], \emptyset,$

$\texttt{game.getOwner.value}(0).\texttt{sender}(\text{"}0x02\text{"})(\texttt{num}) \rangle \rightarrow$

$\langle \beta_b \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_3, 200)] \cdot [\texttt{msgsender} \mapsto \text{"}0x02\text{"}; \texttt{this} \mapsto c_{\texttt{game}}; \texttt{msg.value} \mapsto 0; \texttt{nft} \mapsto 0],$

$\beta_b \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_3, 200)], [a_{\texttt{game}}], \texttt{return this.nfts[nft]} \rangle \rightarrow$

$\langle \beta_b \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_3, 200)], \beta_b \cdot [(c_{\texttt{game}}, a_{\texttt{game}}) \mapsto (\texttt{Game}, s_3, 200)], \emptyset, \text{"}0x02\text{"} \rangle$

$\beta_a = [0x01 \rightarrow 2000; 0x02 \rightarrow 2000]$
$\beta_b = [0x01 \rightarrow 2000; 0x02 \rightarrow 1800]$
$s_1 = [\texttt{nfts} \mapsto [\ ]; \texttt{totalNfts} \mapsto 0; \texttt{price} \mapsto 0; \texttt{owner} \mapsto \text{"}0x01\text{"}]$
$s_2 = [\texttt{nfts} \mapsto [\ ]; \texttt{totalNfts} \mapsto 0; \texttt{price} \mapsto 200; \texttt{owner} \mapsto \text{"}0x01\text{"}]$
$s_3 = [\texttt{nfts} \mapsto [0 \mapsto \text{"}0x02\text{"}]; \texttt{totalNfts} \mapsto 1; \texttt{price} \mapsto 200; \texttt{owner} \mapsto \text{"}0x01\text{"}]$

5.5.9 Implementation in OCaml

When choosing to implement operational semantics, two distinct approaches are available: small-step semantics and big-step semantics. The latter differs from the former by hiding even more execution details.

For our project, which aimed to develop a proof-of-concept interpreter, we opted for big step semantics due to several advantages aligned with this objective. First and foremost, it allowed us to create a simpler implementation of the interpreter, focusing on the overall result of program execution rather than the intricate details of each computational step. This simplicity facilitated a swift development process and enabled us to build a functional prototype efficiently.

Furthermore, big step semantics provided a higher level of abstraction, making it easier to capture the essential behavior of the language or system being modeled. This abstraction proved beneficial in conceptualizing and formalizing the core functionality of the interpreter, allowing us to abstract away from low-level operational intricacies.

Lastly, the use of big step semantics eased the transition from formalization to code implementation. The high-level nature of big step semantics made it more straightforward to translate our rules into executable code in a functional programming language, establishing a direct mapping between the theoretical model and the practical interpreter.

In summary, the choice of big step semantics was well-suited to our proof-of-concept interpreter project. It facilitated a simpler implementation, provided a higher level of abstraction, and streamlined the process of moving from formalization to executable code.

In Figure 5.9, a snippet of our implementation is presented. Note how we evaluate all necessary expressions recursively in a single step, rather than dividing them into multiple steps. Additionally, our function takes a contract table (ct) and a variable table (vars) as input, both implemented as hash tables. Our configuration consists of a quadruple: the first two elements are identical, with one representing our blockchain and the other a copy of the blockchain before a transaction. The third element is a stack of addresses (sigma), and the last one is an expression to evaluate.

Since blockchain definition is more robust than others, we provide its definition below:

```
1 type contracts = ((values * values), (string * (expr) StateVars.t * values))
2 type accounts =  (values, values) Hashtbl.t (*  addresses * balance *)
3 type blockchain = (contracts * accounts)
```

Figure 5.8: Blockchain definition in OCaml

As we can see, our blockchain is structured as a tuple, composed by two distinct types: contracts and accounts. The first type is designed to model contracts, and it closely resembles Di Pirro's formalization of the blockchain. However, we have extended our implementation to include accounts, represented as a hash table that maps each account's address to their respective balance.

```
let rec eval_expr
    (ct: (string, contract_def) Hashtbl.t)
    (vars: (string, expr) Hashtbl.t)
    (conf: conf) : conf
  =
  let (blockchain, blockchain', sigma, e) = conf in
  match e with
  | Let (_, x, e1, e2) ->
    begin if Hashtbl.mem vars x then
        (
      (blockchain, blockchain', sigma, Revert))
      else
        let (_, _, _, e1') = eval_expr ct vars (blockchain, blockchain',
            sigma, e1) in
        Hashtbl.add vars x e1'; eval_expr ct vars (blockchain, blockchain',
            sigma, e2)
    end
  | Assign (x, e1) ->
    let (_, _, _, e1') = eval_expr ct vars (blockchain, blockchain', sigma,
        e1) in
    Hashtbl.replace vars x e1';
    Val VUnit
  | If (e1, e2, e3) -> let (_, _, _, e1') = eval_expr ct vars
      (blockchain, blockchain', sigma, e1) in
    begin match e1' with
      | Val (VBool b) -> begin match b with
          | True -> eval_expr ct vars (blockchain, blockchain', sigma, e2)
          | False -> eval_expr ct vars (blockchain, blockchain', sigma, e3)
        end
      | e1' -> assert false
    end
    .....
```

Figure 5.9: Snippet of Op Sem Ocaml implementation

As one can observe, we make extensively use of OCaml's pattern matching feature. The first two cases, LET and ASSIGN, deal with local variables. In the former, we check if the name x already exists in our table: if it exists we just return a *Revert*, otherwise it will be identical to ASSIGN, where we add the value evaluated to our hash table and call recursively the function with e_2 as an expression.

In conditional statements, we evaluate the condition: if we get True we return the evaluation of e_2, if we get False we return the evaluation of e_3.

6

Property-based Testing

6.1 General Context

Property-based testing is a software testing technique that focuses on verifying the correctness of a program by specifying a set of properties or invariants that the program should satisfy. Rather than providing specific input values and expected outcomes, property-based testing tools, such as QuickCheck [0], automatically generate a wide range of test cases based on the defined properties.

While we did not test all of our system properties, the results for the ones we did test were highly encouraging, strongly indicating their correctness and robustness.

In this section, we will provide a brief overview of our accomplishments and outline what should be done in the futures to ensure the correctness of the compiler.

6.2 Arithmetic and Boolean Operations

One notable instance of property-based testing involved the validation of our arithmetic operations by employing the concept of an abelian group. By applying this property, we aimed to ensure that our arithmetic operations adhered to the fundamental principles of associativity, commutativity, and identity elements. In doing so, we sought to confirm that our system's arithmetic operations not only produced accurate results but also exhibited the expected algebraic properties that underpin their behavior.

In figure 6.1 one can see the definition on how our test will build the random expressions, in this case always arithmetic expressions.

Then in figure 6.5 we use these previously pre-built generators and test against three properties: commutative, associative and neutral element.

```
let leafgen_int = Gen.oneof[ Gen.map (fun i -> if i > 0 then Val(VUInt i)
    else Val(VUInt 0)) Gen.int]
let rec gen_arit_op_ast n = match n with
  | 0 -> leafgen_int
  | n -> Gen.oneof [
      leafgen_int;
      Gen.map2 (fun l r -> AritOp(Plus(l,r))) (gen_arit_op_ast (n/2))
          (gen_arit_op_ast (n/2));
      Gen.map2 (fun l r -> AritOp(Div(l,r))) (gen_arit_op_ast (n/2))
          (gen_arit_op_ast (n/2));
      Gen.map2 (fun l r -> AritOp(Times(l,r))) (gen_arit_op_ast (n/2))
          (gen_arit_op_ast (n/2));
      Gen.map2 (fun l r -> AritOp(Minus(l,r))) (gen_arit_op_ast (n/2))
          (gen_arit_op_ast (n/2));
      Gen.map2 (fun l r -> AritOp(Exp(l,r))) (gen_arit_op_ast (n/2))
          (gen_arit_op_ast (n/2));
      Gen.map2 (fun l r -> AritOp(Mod(l,r))) (gen_arit_op_ast (n/2))
          (gen_arit_op_ast (n/2));
    ]
let arb_tree_arit = make ~print:expr_to_string (gen_arit_op_ast 8)
```

Figure 6.1: Generating random arithmetic expressions

6.3 Conditional Expressions

Additionally, we conducted property-based testing on our system's handling of conditional expressions, specifically exploring the commutative property of condition reversal. This test suite was designed to verify whether reversing the condition and changing the order of statements within an if expression would yield equal outcomes.

To implement this, we use the following generator:

```
let rec gen_if_expr n =
  match n with
  | 0 -> Gen.map3 (fun e1 e2 e3 -> If(e1, e2, e3)) (gen_bool_revert_safe_ast
      8) (gen_arit_op_ast 8) (gen_arit_op_ast 8)
  | n -> Gen.map3 (fun e1 e2 e3 -> If(e1, e2, e3)) (gen_bool_revert_safe_ast
      8) (gen_if_expr (n/2)) (gen_if_expr (n/2))
let arb_if_expr = make ~print:expr_to_string (gen_if_expr 8)
```

Figure 6.3: Generating random conditional expressions

The function `gen_bool_revert_safe_ast` returns a random boolean expression that is safe from `revert`, as including `revert`would render our tests unable to verify any property. Additionally, we generate e_2 and e_3 as random arithmetic expressions using the previously presented generator.

As shown in Figure 6.5, when testing conditional expressions, we used the following properties:

```
let test_arit_plus_op = Test.make ~name:"test plus arithmetic operator"
    (set_shrink tshrink arb_tree_arit)
    (fun (e) ->
       begin
         let ct = Hashtbl.create 64 in
         let vars = Hashtbl.create 64 in
         let blockchain = (Hashtbl.create 64, Hashtbl.create 64) in
         let sigma = Stack.create() in
         let e' = AritOp(Plus(e,e)) in
         (
           eval_expr ct vars (blockchain, blockchain, sigma,
               (AritOp(Plus(e',e))))
           =
           eval_expr ct vars (blockchain, blockchain, sigma,
               (AritOp(Plus(e,e'))))
         )
         &&
         (
           eval_expr ct vars (blockchain, blockchain, sigma,
               (AritOp(Plus((AritOp(Plus(e,e'))), e))))
           =
           eval_expr ct vars (blockchain, blockchain, sigma,
               (AritOp(Plus((AritOp(Plus(e,e))), e'))))
         )
         &&
         (
           eval_expr ct vars (blockchain, blockchain, sigma,
               (AritOp(Plus(Val(VUInt 0),e))))
           =
           eval_expr ct vars (blockchain, blockchain, sigma, e)
         )
       end
    )
```

Figure 6.2: Plus operation property-based testing

$$\texttt{if false then } e_1 \texttt{ else } e_2 = \texttt{if true then } e_2 \texttt{ else } e_1$$

$$\texttt{if } b \texttt{ then } e_1 \texttt{ else } e_2 = \texttt{if } \neg\, b \texttt{ then } e_2 \texttt{ else } e_1$$

Essentially, reversing the condition and the order of the statements in a conditional expression results in an equivalent expression.

6.4 Future Work

As we have seen, leveraging the power of property-based testing is highly effective in ensuring the correctness of a given system. This approach enables comprehensive validation of the system's behavior across a wide range of scenarios, with a particular emphasis on critical properties essential for correctness and reliability.

The outcomes of our tests proved to be not only highly encouraging but also compellingly indicative of the correctness and robustness of select operational semantics rules.

```
let test_if = Test.make ~name:"test if operator"
    (set_shrink tshrink arb_if_expr)
    (fun (e) ->
       begin
         let ct = Hashtbl.create 64 in
         let vars = Hashtbl.create 64 in
         let blockchain = (Hashtbl.create 64, Hashtbl.create 64) in
         let sigma = Stack.create() in
         let (e1, e2, e3) = match e with
           | If (e1, e2, e3) ->
             let (_, _, _, e1') = eval_expr ct vars (blockchain, blockchain,
                 sigma, e1) in
             if e1' <> Revert then (e1, e2, e3) else (Val(VBool False), e2,
                 e3)
           | _ -> assert false
         in
         (
           eval_expr ct vars (blockchain, blockchain, sigma, (If(Val(VBool
               True), e2, e3)))
           =
           eval_expr ct vars (blockchain, blockchain, sigma, e2)
         )
         &&
         (
           eval_expr ct vars (blockchain, blockchain, sigma, (If(Val(VBool
               False), e2, e3)))
           =
           eval_expr ct vars (blockchain, blockchain, sigma, e3)
         )
         &&
         (
           eval_expr ct vars (blockchain, blockchain, sigma, (If(e1, e2, e3)))
           =
           eval_expr ct vars (blockchain, blockchain, sigma,
               (If(BoolOp(Neg(e1)), e3, e2)))
         )
         &&
         (
           let (_, _, _, e') = eval_expr ct vars (blockchain, blockchain,
               sigma, (If(e1, e2, e3))) in
           let (_, _, _, e2') = eval_expr ct vars (blockchain, blockchain,
               sigma, e2) in
           let (_, _, _, e3') = eval_expr ct vars (blockchain, blockchain,
               sigma, e3) in
           (e' = e2') || (e' = e3')
         )
       end
    )
```

Figure 6.4: Conditional expressions property-based testing

it is important to note that while we successfully applied these tests to some rules, for an industrial tool we advise extending property-based testing to encompass all expressions within a system. By doing so, one can systematically evaluate the behavior and consistency of every aspect of the system, thus reinforcing its overall reliability and adherence to established properties.

Figure 6.5: Property-based testing results

In the figure above, we can see the output we get after running *dune runtest* in the project root.

7

FS 2.0: Type System

7.1 General Context

In this section, we delve into the core of our study: the type system rules in Featherweight Solidity 2.0. A well-defined type system is the backbone of any programming language, governing how values and expressions are classified, checked, and enforced during program execution. These rules form the foundation upon which developers build robust, error-free software.

Our goal here is to elucidate the intricacies of the type system in FS 2.0, providing a comprehensive understanding of how types are assigned, inferred, and verified within the language. In particular, we explain in detail, how do we deal with multiple inheritance and subtyping in the type address. This exploration will lay the groundwork for subsequent discussions on type checking, type inference, and ultimately, the assurance of program correctness.

Next, we present the fundamental type system rules that shape the programming experience in FS 2.0.

7.2 Type Environment

Our environment differs slightly from Di Pirro's in that we incorporate an additional component to store the types of each state variable. In our model, we assume that a contract's state variables are only accessible internally, through the **this** keyword. In Solidity, state variables are also private by default, but when declared as **public**, the language automatically generates a getter function for external access. In our language, you need to explicitly create it.

$$(\textit{Type Environemnt}) \quad \Gamma \quad ::= \quad \emptyset \mid \Gamma, x : T \mid \Gamma, \texttt{this}.s : T \mid \Gamma, a : \texttt{address} \mid \Gamma, c : C$$

Figure 7.1: Type Environment of Featherweight Solidity 2.0

7.3 Rules

We will start by presenting the type system rules without inheritance in section 7.3.1. Since inheritance is only possible in two types: contracts and addresses, some rules will not change between both formalization and we chose to omit to avoid duplicating. Most of the rules, only need a less strict type matching: if we are expecting an expression with type T, but we instead get T', if T' is a subtype of T, it is still valid. The rule to type check a cast operation is the one that will change the most.

7.3.1 Rules Without Inheritance

Our presentation begins with the foundational axioms that underpin our type system. These axioms serve as the cornerstone, shaping the system's core principles.

Next, we systematically categorize rules based on their similarity:

- Arithmetic and Boolean Operations - Emphasizing mathematical and logical operations.
- Variable Operations - Covering state and local variables.
- Conditional and Sequential Statements - If statements and sequence of expressions.
- Function-like Operations - Including castings, function calls, top-level calls, and balance calculations.

This classification framework provides a structured approach to explore our type system comprehensively.

7.3.1.1 Axioms

$$\text{(REF)}\quad \frac{\Gamma, c : C, \Gamma' \vdash \langle\,\rangle}{\Gamma, c : C, \Gamma' \vdash c : C}$$

$$\text{(VAR)}\quad \frac{\Gamma, x : T, \Gamma' \vdash \langle\,\rangle}{\Gamma, x : T, \Gamma' \vdash x : T}$$

$$\text{(TRUE)}\quad \frac{\Gamma \vdash \langle\,\rangle}{\Gamma \vdash \texttt{true} : \texttt{bool}}$$

$$\text{(FALSE)}\ \frac{\Gamma \vdash \langle\,\rangle}{\Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$\text{(UNIT)}\ \frac{\Gamma \vdash \langle\,\rangle}{\Gamma \vdash u : \texttt{unit}}$$

$$\text{(REVERT)}\ \frac{\Gamma \vdash \langle\,\rangle}{\Gamma \vdash \texttt{revert} : T}$$

$$\text{(UINT)}\ \frac{n \in \mathbb{N}_0 \qquad \Gamma \vdash \langle\,\rangle}{\Gamma \vdash n : \texttt{uint}}$$

$$\text{(ADDRESS)}\ \frac{\Gamma, a : \texttt{address}, \Gamma' \vdash \langle\,\rangle}{\Gamma, a : \texttt{address}, \Gamma' \vdash a : \texttt{address}}$$

These rules are equivalent to Di Pirro, formalization, and so there is no much to say.

7.3.1.2 Arithmetic and Boolean

$$\text{(PLUS/MINUS/DIV/MULT/EXP/MOD)}\ \frac{\Gamma \vdash e_1 : \texttt{uint} \qquad \Gamma \vdash e_2 : \texttt{uint}}{\Gamma \vdash e_1 \diamond e_2 : \texttt{uint}}$$

$$\text{(EQUALS/INEQUALS/GREATER/LESSER)}\ \frac{\Gamma \vdash e_1 : \texttt{uint} \qquad \Gamma \vdash e_2 : \texttt{uint}}{\Gamma \vdash e_1 \diamond e_2 : \texttt{bool}}$$

$$\text{(Equals/Inequals)}\ \frac{\Gamma \vdash e_1 : \texttt{address} \qquad \Gamma \vdash e_2 : \texttt{address}}{\Gamma \vdash e_1 \diamond e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \diamond e_2 : \texttt{bool}} \text{(CONJ/DISJ)}$$

These rules establish criteria for well-typed expressions involving arithmetic and Boolean operators supported by our programming language. It is essential to note that our equality and inequality operations support comparisons between address and uint data types.

7.3.1.3 Variables

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash T_1\ x = e_1; e_2 : T_2} \text{(LET)}$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T}{\Gamma \vdash x = e : T} \text{(ASSIGN)}$$

$$\frac{\Gamma \vdash e_1.s : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1.s = e_2 : T} \text{(STATEASSIGN)}$$

$$\frac{}{\Gamma \vdash e_1.s_i : T_i} \text{(STATEREAD)}$$

These rules handle either local variables (ASSIGN and LET) or state variables (STATEASSIGN and STATEREAD) and closely resemble the corresponding rules we previously introduced in Chapter 4. The only difference here is that, since we store our state variables types in our type environment, the STATEREAD rule is straightforward, and we can directly retrieve its value from our Γ.

7.3.1.4 Constructor

$$\frac{SC = \texttt{contract}\ C\ \textit{is}\ \widetilde{D}\ \{(\widetilde{T\ s})\ K\ \widetilde{F}\} \qquad K = \texttt{constructor}\ ((\widetilde{T\ x}))(e))}{\texttt{ctypes}(SC) = \widetilde{T}} \text{(Retrieve Constructor Parameters types)}$$

$$\frac{\text{(NEW)}}{\Gamma \vdash e' : \texttt{uint} \qquad \Gamma \vdash \widetilde{e} : \widetilde{T} \qquad \texttt{ctypes}(C) = \widetilde{T}}{\Gamma \vdash \texttt{new}\ C.\texttt{value}(e')(\widetilde{e}) : C}$$

To check if a statement with NEW is well-typed we need an auxiliary predicate, which appears first, where we retrieve for a contract its constructor parameters type ($\widetilde{T}$). Then in the rule, we check that e' is an unsigned integer and check that argument in $\widetilde{e}$ respect the expected type in $\widetilde{T}$.

7.3.1.5 Conditional and Sequential Statements

$$\frac{\text{(SEQ)}}{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1; e_2 : T_2}$$

This rule ensures that e_1 is well-typed and has type T_1 and e_2 typechecks sucessfully with T_2.

$$\frac{\text{(IF)}}{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T}$$

This follows the traditional expect type check rule for conditional statements, e_1 has the condition must have a boolean type and e_2 and e_3 should have the same type.

7.3.1.6 Mappings

$$\frac{\text{(MAPPING)}}{M = \{\widetilde{(k, v)}\} \qquad \Gamma \vdash \widetilde{k} : T_1 \qquad \Gamma \vdash \widetilde{v} : T_2}{\Gamma \vdash M : \texttt{mapping}(T_1 \Rightarrow T_2)}$$

$$\frac{\text{(MAPASSIGN)}}{\Gamma \vdash e1 : \texttt{mapping}(T_1 \rightarrow T_2) \qquad \Gamma \vdash e2 : T_1 \qquad \Gamma \vdash e3 : T_2}{\Gamma \vdash e1[e2 \rightarrow e3] : \texttt{mapping}(T_1 \Rightarrow T_2)}$$

$$\frac{\text{(MAPREAD)}}{\mathbf{\Gamma} \vdash \mathbf{e1} : \texttt{mapping}(\mathbf{T_1} \rightarrow \mathbf{T_2}) \qquad \Gamma \vdash e2 : T_1}{\Gamma \vdash e1[e2] : T_2}$$

For mappings we have three rules. The first one, checks if a mapping is well-typed, by checking that each key k and each value v has type T_1 and T_2 respectively. When we are

assigning a value to a key in a certain expression e_1, we check that e_1 is a mapping, e_2 has type T_1 and e_3 has type T_2. When we are reading a value, we need to infer that e_1 can be typed to a mapping and check that e_2 has type T_1, which is the type of keys of e_1.

7.3.1.7 Functions

$$\text{(BALANCE)}\quad \frac{\Gamma \vdash e : \texttt{address}}{\Gamma \vdash \texttt{balance}(e) : \texttt{uint}}$$

$$\text{(ADDR)}\quad \frac{\Gamma \vdash e : C}{\Gamma \vdash \texttt{address}(e) : \texttt{address}}$$

$$\text{(RETURN)}\quad \frac{\Gamma \vdash e : T}{\Gamma \vdash \texttt{return}\ e : T}$$

$$\text{(CONTRRETR)}\quad \frac{\Gamma \vdash e : \texttt{address}}{\Gamma \vdash C(e) : C}$$

$$\text{(TRANSFER)}\quad \frac{\mathbf{\Gamma \vdash e_1 : address\langle C\rangle} \qquad \texttt{ftype}(C, \texttt{fb}) = \{\} \to \texttt{unit} \qquad \Gamma \vdash e_2 : \texttt{uint}}{\Gamma \vdash e_1.\texttt{transfer}(e_2) : \texttt{unit}}$$

$$\text{(CALLTOPLEVEL)}\quad \frac{\texttt{ftype}(C, f) = \widetilde{T_1} \to T_2 \qquad |\widetilde{e}| = |\widetilde{T_1}| \qquad \mathbf{\Gamma \vdash e_1 : C} \qquad \Gamma \vdash e_2 : \texttt{uint} \qquad \Gamma \vdash e_3 : \texttt{address} \qquad \Gamma \vdash \widetilde{e} : \widetilde{T_e} \qquad \widetilde{T_e} = \widetilde{T_1}}{\Gamma \vdash e_1.f.\texttt{value}(e_2).\texttt{sender}(e_3)(\widetilde{e}) : T_2}$$

$$\text{(CALL)}\quad \frac{\Gamma \vdash e_2 : \texttt{uint} \qquad \texttt{ftype}(C, f) = \widetilde{T_1} \to T_2 \qquad |\widetilde{e}| = |\widetilde{T_1}| \qquad \mathbf{\Gamma \vdash e_1 : C} \qquad \Gamma \vdash \widetilde{e} : \widetilde{T_e} \qquad \widetilde{T_e} = \widetilde{T_1}}{\Gamma \vdash e_1.f.\texttt{value}(e_2)(\widetilde{e}) : T_2}$$

$$\text{(FUN)}\quad \frac{\Gamma \vdash \mathbf{e_1} : \mathbf{C} \qquad \texttt{ftype}(C, f) = \widetilde{T_1} \to T_2}{\Gamma \vdash c.f : \widetilde{T_1} \to T_2}$$

$$\text{(CALLVALUE)}\quad \frac{\Gamma \vdash \widetilde{T_1} \to T_2 \qquad \Gamma \vdash e_2 : \texttt{uint} \qquad \Gamma \vdash \widetilde{e_1} : \widetilde{T_1} \qquad |\widetilde{e}| = \widetilde{T_1} \qquad \Gamma \vdash \widetilde{e} : \widetilde{T_e} \qquad \widetilde{T_e} = \widetilde{T_1}}{\Gamma \vdash e_1.\texttt{value}(e_2)(\widetilde{e}) : T_2}$$

Excluding inheritance, these rules are straightforward and identical to Di Pirro's formalization. We will delve deeper into CALLTOPLEVEL, CALL, CONTRRETR, FUN, and CALLVALUE as they are the ones that differ the most from either without inheritance rules or Di Pirro's rules.

7.3.2 Rules With Inheritance

In this section, we will delve into the concept of inheritance within our rule framework. It is crucial to emphasize the existence of a top type for contracts, denoted as CTOP. As such, it operates akin to the Object class in Java. However, users cannot declare this type themselves, as it serves as an internal representation encompassing all contracts.

$$\texttt{issc}(C_1, C_2, CT) = C_2 \in \texttt{c3linearization}(CT(C_1)) \vee C_2 = CTOP$$

$$\texttt{subtyping}(T_1, T_2, CT) = \begin{cases} \texttt{issc}(C_1, C_2, CT) & T_1 = C_1 \wedge T_2 = C_2 \\ \texttt{issc}(C, CTOP, CT) & T_1 = \texttt{address}\langle C\rangle \wedge T_2 = \texttt{address} \\ \texttt{issc}(CTOP, C, CT) & T_1 = \texttt{address} \wedge T_2 = \texttt{address}\langle C\rangle \\ \texttt{subtyping}(C_1, C_2, CT) & T_1 = \texttt{address}\langle C_1\rangle \wedge T_2 = \texttt{address}\langle C_2\rangle \\ T_1 = T_2 & \texttt{otherwise} \end{cases}$$

This algorithm serves as our method to determine if two types, denoted as T_1 and T_2, exhibit a subtype relationship. It is essential to note that the order in which we conduct these comparisons holds significance because our goal is to establish whether T_1 is either identical or a subtype of T_2. Our type hierarchy incorporates inheritance exclusively for contract and address types; therefore, for all other types, strict equality suffices.

To qualify as a subtype of contract C_t, one of two conditions must be met: either C_t is equal to the top-level contract, denoted as CTOP, or it is part of the linearization of the contract type under consideration. In the case of address types, the determination of subtype status hinges on comparing the enclosing annotation contract of the address type with the other type or, equivalently, assessing if the generic address is akin to the CTOP contract.

7.3.2.1 Axioms

$$\frac{\Gamma, a : \texttt{address}\langle C\rangle, \Gamma' \vdash \langle\,\rangle}{\Gamma, a : \texttt{address}\langle C\rangle, \Gamma' \vdash a : \texttt{address}\langle C\rangle}\ (\text{ADDRESS})$$

Just one axiom change to support inheritance type, since we introduce a new subtyping address and so every address should be annotated with it is contract type. For C = TOP we have the generic type address, that already exists in Solidity.

7.3.2.2 Adapted Rules

$$\frac{\Gamma \vdash e_1 : T_1' \qquad \texttt{subtyping}(T_1', T1, CT) \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash T_1\ x = e_1; e_2 : T_2}\ (\text{LET})$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T_e \qquad \texttt{subtyping}(t_e, T, CT)}{\Gamma \vdash x = e : T_e}\ (\text{ASSIGN})$$

$$\frac{\Gamma \vdash e_1.s : T \qquad \Gamma \vdash e_2 : T' \qquad \texttt{subtyping}(T', T, CT)}{\Gamma \vdash e_1.s = e_2 : T'}\ (\text{STATEASSIGN})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : T_1 \qquad \Gamma \vdash e_3 : T_2 \qquad \texttt{subtyping}(T_1, T, CT) \qquad \texttt{subtyping}(T_2, T, CT)}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T}\ (\text{IF})$$

$$\frac{M = \{\widetilde{(k, v)}\} \qquad \Gamma \vdash \widetilde{k} : T_k \qquad \Gamma \vdash \widetilde{v} : T_v \qquad \texttt{subtyping}(T_k, T_1, CT) \qquad \texttt{subtyping}(T_v, T_2, CT)}{\Gamma \vdash M : \texttt{mapping}(T_1 \Rightarrow T_2)}\ (\text{MAPPING})$$

$$\frac{\Gamma \vdash e1 : \texttt{mapping}(T_1 \rightarrow T_2) \qquad \Gamma \vdash e2 : T_1' \qquad \Gamma \vdash e3 : T_2' \qquad \texttt{subtyping}(T_1', T_1, CT) \qquad \texttt{subtyping}(T_2', T_2, CT)}{\Gamma \vdash e1[e2 \rightarrow e3] : \texttt{mapping}(T_1 \Rightarrow T_2)}\ (\text{MAPASSIGN})$$

$$\frac{\text{(MAPREAD)}\quad \mathbf{\Gamma \vdash e1 : \texttt{mapping}(T_1 \rightarrow T_2)} \qquad \Gamma \vdash e2 : T_1' \qquad \texttt{subtyping}(T_1', T_1, CT)}{\Gamma \vdash e1[e2] : T_2}$$

$$\frac{\text{(TRANSFER)}\quad \mathbf{\Gamma \vdash e_1 : \texttt{address}\langle C\rangle} \qquad \texttt{ftype}(C, \texttt{fb}) = \{\} \rightarrow \texttt{unit} \qquad \Gamma \vdash e_2 : \texttt{uint}}{\Gamma \vdash e_1.\texttt{transfer}(e_2) : \texttt{unit}}$$

In all of these rules, we adjusted they, relaxing the condition where we get T', but instead we expected a type T. In these cases, we just need to call our algorithm of subtyping and ensure that T' is either equal or a subtype of T: `subtyping`(T', T, CT)

7.3.2.3 Rules Changing the Most

$$\frac{\text{(CONTRRETR)}\quad \Gamma \vdash e : \texttt{address}\langle C\rangle \qquad \texttt{subtyping}(C, D, CT)}{\Gamma \vdash D(e) : D}$$

With inheritance and subtyping, we are more restrictive in our casts. First our address is either annotated with a contract or it is a generic address. If its a generic address it cannot be casted to a contract, since the result for `subtyping`(CTOP, D, CT) will always be false. This is a good improvement from the current Solidity version, where we can typecast every address, even addresses referencing EOAs. However if we have an expression annotated with an address subtype, we can cast this address to every contract that is in the linearization of the said contract, as a result of the premise using `subtyping` function. We consider our address type safe, making our language type safe, contrary to Solidity.

$$\frac{\begin{array}{l}\text{(CALLTOPLEVEL)}\quad \mathbf{\Gamma \vdash e_1 : C} \qquad \Gamma \vdash e_2 : \texttt{uint} \qquad \texttt{ftype}(C, f) = \widetilde{T_1} \rightarrow T_2 \\ |\widetilde{e}| = |\widetilde{T_1}| \qquad \Gamma \vdash e_3 : \texttt{address} \qquad \Gamma \vdash \widetilde{e} : \widetilde{T_e} \qquad \texttt{subtyping}(\widetilde{T_e}, \widetilde{T_1}, CT)\end{array}}{\Gamma \vdash e_1.f.\texttt{value}(e_2).\texttt{sender}(e_3)(\widetilde{e}) : T_2}$$

As mentioned earlier, CALLTOPLEVEL is exclusively called by externally owned accounts (EOAs), ensuring that the value of e_3 is always an address. Therefore, we can dispense with the premise $\texttt{subtyping}(C', \texttt{fsender}(C, f), CT)$ since we do not categorize external-owned accounts as contracts. Additionally, by default, the functions that can be invoked by this constructor should be those without any annotations, expecting only a generic address. The expression $\texttt{subtyping}(\widetilde{T_e}, \widetilde{T_1}, CT)$ is essentially a syntactic shortcut for applying our subtyping function to each element in $\widetilde{T_e}$ and $\widetilde{T_1}$ individually:

$$\texttt{subtyping}(\widetilde{T_e}, \widetilde{T_1}, CT) \equiv \bigwedge_{i=1}^{n} \texttt{subtyping}(T_{e_i}, T_{1_i}, CT)$$

(CALL)

$$\frac{\mathbf{\Gamma \vdash e_1 : C} \quad \Gamma \vdash e_2 : \texttt{uint} \quad \texttt{ftype}(C, f) = \widetilde{T_1} \to T_2 \quad |\widetilde{e}| = |\widetilde{T_1}| \quad \Gamma \vdash \texttt{this} : C' \quad \texttt{subtyping}(C', \texttt{fsender}(C, f), CT) \quad \Gamma \vdash \widetilde{e} : \widetilde{T_e} \quad \texttt{subtyping}(\widetilde{T_e}, \widetilde{T_1}, CT)}{\Gamma \vdash e_1.f.\texttt{value}(e_2)(\widetilde{e}) : T_2}$$

(FUN)

$$\frac{\mathbf{\Gamma \vdash e_1 : C} \quad \texttt{ftype}(C, f) = \widetilde{T_1} \to T_2 \quad \Gamma \vdash \texttt{this} : C' \quad \texttt{subtyping}(C', \texttt{fsender}(C, \texttt{f}), CT)}{\Gamma \vdash c.f : \widetilde{T_1} \to T_2}$$

(CALLVALUE)

$$\frac{\Gamma \vdash \widetilde{T_1} \to T_2 \quad \Gamma \vdash e_2 : \texttt{uint} \quad \widetilde{T_1'} <: T_1 \quad |\widetilde{e}| = |\widetilde{T_1}| \quad \Gamma \vdash \widetilde{e} : \widetilde{T_e} \quad \texttt{subtyping}(\widetilde{T_e}, \widetilde{T_1}, CT)}{\Gamma \vdash e_1.\texttt{value}(e_2)(\widetilde{e}) : T_2}$$

These rules are designed to ensure well-typed function calls within contracts. They require that function calls are either unannotated or annotated with a contract that is either the contract from which we are invoking the function or a supertype of it.

Below, we present an OCaml snippet that demonstrates how we did implement the premises:

$$|\widetilde{e}| = |\widetilde{T_1}| \wedge \texttt{subtyping}(\widetilde{T_e}, \widetilde{T_1}, CT)$$

In this snippet, we leverage the power of OCaml's iter2 function from the List library. This function allows us to iterate over two lists simultaneously. If they don't have the same length, it throws an exception. Inside this iterator, we check whether each element in $\widetilde{e}$ has the expected type according to the parameters, recursively calling our type-checking function.

```
let fun_check gamma cname fun_name le ct t =
    let ftype = function_type cname fun_name ct in
    let (t_es, rettype) = ftype in
    if subtyping rettype t ct then
      try
        List.iter2 (fun t_e e' -> typecheck gamma e' t_e ct;) t_es le
      with Invalid_argument _ ->
        raise (Failure "Invalid arguments")
    else
      raise (TypeMismatch (rettype, t))
  in
```

Figure 7.2: OCaml implementation type checking function calls

7.3.3 Type System by Example

In this section, we will use the contract example introduced previously, which was also utilized in the examples demonstrating the application of semantic operational rules (Figure 5.7). In the first example, we will present the rules applied when verifying the setNFTPrice function:

$$\text{IF}\ \frac{\overline{\quad}\ \overline{\quad}\ \overline{\quad}}{\Gamma \vdash \texttt{if(msgsender == this.owner) then this.price = price else revert} : \texttt{unit}}$$

We begin by applying the IF rule to decompose our expression into simpler expressions. This process allows us to eventually validate it using an axiom or identify a point where we cannot proceed, resulting in a poorly typed expression. Applying this rule we obtain three sub trees which we show next:

$$\frac{\text{UINT}\ \dfrac{}{\Gamma \vdash \texttt{this.price} : \texttt{uint}} \quad \dfrac{}{\Gamma \vdash \texttt{price} : \texttt{uint}}\ \text{UINT}}{\Gamma \vdash \texttt{this.price = price} : \texttt{unit}}\ \text{STATEASSIGN}$$

$$\text{REVERT}\ \frac{\dfrac{}{\Gamma \vdash \texttt{revert} : \texttt{unit}}}{\Gamma \vdash \texttt{revert} : T'}$$

$$\frac{\text{ADDRESS}\ \dfrac{}{\Gamma \vdash \texttt{msgsender} : \texttt{address}} \quad \dfrac{}{\Gamma \vdash \texttt{this.owner} : \texttt{address}}\ \text{ADDRESS}}{\Gamma \vdash \texttt{msgsender == this.owner} : \texttt{bool}}\ \text{EQUALS}$$

Γ = [this.price $\mapsto$ uint;price $\mapsto$ uint;this.owner $\mapsto$ address;msgsender $\mapsto$ address;this.totalNfts $\mapsto$ uint;this.nfts $\mapsto$ mapping(uint $\rightarrow$ address)]

Note that in all rules, we obtain an axiom that allows us to validate that our expressions are well-typed. The REVERT rule is particularly interesting, as it is well-typed regardless of the type T' we need to achieve. This is crucial since revert is often used to abort a transaction with an error and revert to the blockchain initial state. Consequently, it can be employed in conditional statements if the desired evaluation is not achieved, necessitating the use of REVERT with other supported types.

We also want to note that we omit the premises using the subtyping function. However, assume that after this derivation tree, there are two additional steps to verify:

$$\texttt{subtyping}(\texttt{unit}, \texttt{unit}, CT)$$

$$\texttt{subtyping}(\texttt{unit}, \texttt{unit}, CT)$$

Both of which obviously evaluate to true, since this expression is also well-typed if we did not have inheritance.

Now we see how we can apply our type system rules to successfully verify that getOwner function is correctly typed:

$$\text{RETURN}\ \dfrac{\text{MAPREAD}\ \dfrac{\text{STATEREAD}\ \dfrac{}{\Gamma \vdash \texttt{this.nfts} : \texttt{mapping(uint} \rightarrow \texttt{address)}} \quad \dfrac{}{\Gamma \vdash \texttt{nft} : \texttt{uint}} \quad \dfrac{\text{UINT}}{\texttt{subtyping(uint, uint, } CT\texttt{)}}}{\Gamma \vdash \texttt{this.nfts[nft]} : \texttt{address}}}{\Gamma \vdash \texttt{return this.nfts[nft]} : \texttt{address}}$$

Γ = [this.price $\mapsto$ uint;nft $\mapsto$ uint;this.owner $\mapsto$ address;msgsender $\mapsto$ address;this.totalNfts $\mapsto$ uint;this.nfts $\mapsto$ mapping(uint $\rightarrow$ address)]

7.3.4 Implementation in OCaml

Our approach to implementing the type system rules in OCaml leverage mutually recursive functions: `infer_type` and `typecheck` are mutually recursive (Figure 7.5). The `infer_type` function was necessary because some rules had premises that required inferring types (indicated in bold on the rules). However, to infer a type from an expression, we also needed to ensure that the expression was well-composed and correctly typed. Both functions took as parameters a gamma (*type environment*), an expression e, and a contract table.

```
1 type gamma_vars = (string, t_exp) Hashtbl.t
2 type gamma_state_vars = (string, t_exp) Hashtbl.t
3 type gamma_addresses = (values, t_exp) Hashtbl.t
4 type gamma_contracts = (values, t_exp) Hashtbl.t
5 type gamma = (gamma_vars * gamma_state_vars *gamma_addresses *
    gamma_contracts)
```

Figure 7.3: Gamma definition in OCaml

Since every expression e will most of the times reduce to an axiom, note how we have a function `typecheck_axioms` (Figure 7.4) where we enforce that the type of an axiom, t_e is either *TRevert* (line 7) or it evaluates to true calling the function: `subtyping`(t_e, t, CT) (line 8). This ensures inheritance is applied across most of the rules.

```
1
2 let typecheck_axioms (gamma: gamma) (e: expr) (t: t_exp) (ct:
    contract_table): unit =
3 let t_e = axioms gamma e in
4 begin match t_e with
5   | Ok(t_e) ->
6     begin match t_e with
7       | TRevert -> ()
8       | _-> (if subtyping t_e t ct then () else (raise (TypeMismatch (t_e,
            t))))
9       (* | _ -> if t_e = t then () else raise (TypeMismatch (t_e, t)) *)
10    end
11  | Error(s) -> raise (Failure s)
12 end
```

Figure 7.4: Typecheck axioms in OCaml

In the `typecheck` function, we introduced an additional argument t, representing the expected type for an expression e. While `infer_type` returns either a type or an error, `typecheck` returns either a *Failure* or an *Exception*.

```
1  let rec infer_type (gamma: gamma) (e: expr) (ct: contract_table) : (t_exp,
       string) result =
2    match e with
3    | Val _ | Var _ | Revert | This None | MsgSender | MsgValue -> axioms gamma
         e
4    | Balance e1 -> typecheck gamma (Balance(e1)) UInt ct; Ok(UInt)
5    | Assign (s, e1) -> typecheck gamma (Assign(s, e1)) Unit ct; Ok(Unit)
6    | Address e1 -> typecheck gamma (Address e1) (Address None) ct; Ok(Address
         None)
7    | Return e1 -> let t_e1 = infer_type gamma e1 ct in t_e1
8    | If (e1, e2, e3) ->
9      begin
10       typecheck gamma e1 Bool ct; (* checks if condition is a boolean *)
11       let t_e2 = infer_type gamma e2 ct in (* infers e2 type *)
12       match t_e2 with (*if it can be inferred, checks e3 have the same type*)
13       | Ok(t_e2) -> typecheck gamma e3 t_e2 ct; Ok(t_e2)
14       | Error s -> Error s
15     end
16 ....
17 and typecheck (gamma: gamma) (e: expr) (t: t_exp) (ct: contract_table) : unit
       =
18     match e with
19     | Call (e1, s, e2, le) ->
20     begin
21       let t_e1 = infer_type gamma e1 ct in
22       match t_e1 with
23       | Ok(C name) ->
24         let (gv, _, _, _) = gamma in
25         let t_sender = match fsender name s ct with
26           | Ok t_sender -> t_sender
27           | Error s -> raise (Failure s)
28         in
29         Hashtbl.add gv "msg.sender" t_sender;
30         typecheck gamma e2 UInt ct;
31         fun_check gamma name s le ct t;
32       | Ok(CTop) -> raise (Failure "Can't reference a top class")
33       | Ok(t_e1) -> raise (TypeMismatch (t_e1, CTop))
34       | Error s -> raise (Failure s)
35     end
36 ....
```

Figure 7.5: Snippet of type check implementation

In line 4, we observe that we return the type `uint` if and only if the expression balance(e) is well-typed. The same principle applies to the if expression (lines 8 to 15).Initially, we check whether e_1 is a boolean expression. Subsequently, we infer the type of e_2, and if e_3 has the same type as e_2, we return the type of both as the result. Otherwise, we return an *Error*.

In the `typecheck` function, we can observe the implementation of type checking for the CALL rule. Firstly, in line 21, we infer the type of e_1. Subsequently, we match this type with the possible values we can obtain (lines 22 to 35). If we obtain anything other than an expression typed with a contract, we raise a *Failure*. Otherwise, we proceed by storing the sender type annotation. If the sender type annotation is hidden, the sender type defaults

to just an address (lines 25 to 29). Following this, we ensure that e_2 has the type `uint` and finally call the previously presented function, `fun_check` (Figure 7.2), checking that each argument has a type that is equal or a subtype of the corresponding parameter type.

8

Case Studies

8.1 General Context

In this section we will present our main case study during our implementation. The contract which will be the one interacted by EOAs is in Figure 5.3. The following contracts are all used to build our Game contract:

```
1 contract Context {
2
3     constructor() {}
4
5     function fb() {}
6     function msgSender() returns (address) {
7         return msgsender;
8     }
9 }
```

Figure 8.1: Context Contract

```
1  import "./Context.sol";
2
3  contract Ownable is Context {
4
5      address owner;
6
7      constructor(address initialOwner) Context() {
8          this.owner = initialOwner;
9      }
10
11     function fb() {}
12     function checkOwner() {
13         if (this.owner != this.msgSender()) {
14             revert;
15         }
16     }
17     function onlyOwner() {
18         this.checkOwner();
19     }
20     function renounceOwnership() {
21         this.onlyOwner();
22         this.transferOwnership(address(0));
23     }
24     function transferOwnership(address newOwner) {
25         this.checkOwner();
26         if (newOwner == address(0)) {
27             revert;
28         } else {
29             this.transferOwnership(newOwner);
30         }
31     }
32 }
```

Figure 8.2: Ownable Contract

```
import "./Context.sol";

contract TinyERC721 is Context {

    mapping(uint => address) owners;
    mapping(address => uint) balances;

    constructor() Context(){}

    function fb() {}
    function ownerOf(uint tokenId) returns (address) {
        address ownr = this.owners[tokenId];
        if (ownr == address(0)) {
           return address(0);
        }
        else {
            return ownr;
        }
    }
    function balanceOf(address owner) returns (uint) {
        if (owner == address(0)) {
            revert;
        }
        else{
           return this.balances[owner];
        }
    }
    function exists(uint tokenId) returns (bool) {
        return (this.ownerOf(tokenId) != address(0));
    }
    function transferFrom(address from, address to, uint tokenId) {
        this.balances = (this.balances[from] = this.balances[from] - 1);
        this.balances = ( this.balances[to] =  this.balances[to] + 1);
        this.owners = (this.owners[tokenId] = to);
    }
    function mint(address to, uint tokenId) {
        if ((to == address(0)) || this.exists(tokenId)) {
            revert;
        }
        else {
            this.balances = ( this.balances[to] =  this.balances[to] + 1);
            this.owners = (this.owners[tokenId] = to);
        }
    }
    function burn(address own, uint tokenId) {
        this.owners = (this.owners[tokenId] = address(0));
        uint currentBal = this.balances[own];
        this.balances = (this.balances[own] = currentBal - 1);
    }
}
```

Figure 8.3: ERC721 Contract

```
import "./Ownable.sol";
import "./ERC721.sol";

contract NFTStorage is Ownable, TinyERC721 {

    uint lastUnsoldToken;
    uint lastTokenId;
    uint price;

    constructor() Ownable(msgsender) TinyERC721()  {}

    function fb() {}
    function setNFTPrice(uint val) {
        this.onlyOwner();
        this.price = val;
    }
    function createNFT(address to) {
        this.onlyOwner();
        this.mint(to, this.lastTokenId);
        this.lastTokenId = this.lastTokenId + 1;
    }
    function buyNFT(address buyer) {
        if(msgvalue >= this.price) {
            this.transferFrom(address(0), buyer, this.lastUnsoldToken);
        }
        else {
            revert;
        }
    }
    function transferNFT(uint tokenId, address from, address to) {
        if(from == this.ownerOf(tokenId)) {
            this.transferFrom(from, to, tokenId);
        }
        else {
            revert;
        }
    }
    function destroyNFT(address owner, uint tokenId) {
        if(owner == this.ownerOf(tokenId)) {
            this.burn(owner, tokenId);
        }
        else {
            revert;
        }
    }
}
```

Figure 8.4: NFT Storage Contract

If one prefers, there is also a GitHub repository, where all the contracts are conveniently organized. The repository can be found here.

8.2 Contract Hierarchy and C3 Linearization Results

In the presented class hierarchy (figure 8.5), following the C3 Linearization algorithm we did implement, the linearization sequences for each class are as follows:

$$
\begin{aligned}
\text{Context} &\rightarrow [\text{Context}] \\
\text{Ownable} &\rightarrow [\text{Ownable}, \text{Context}] \\
\text{TinyERC721} &\rightarrow [\text{TinyERC721}, \text{Context}] \\
\text{NFTStorage} &\rightarrow [\text{NFTStorage}, \text{TinyERC721}, \text{Ownable}, \text{Context}] \\
\text{Game} &\rightarrow [\text{Game}, \text{Context}]
\end{aligned}
$$

In the linearization of TinyERC721, the order of appearance for TinyERC721 and Ownable is not critical: both [NFTStorage, Ownable, TinyERC721, Context] and [NFTStorage, TinyERC721, Ownable, Context] are valid linearization sequences. However, in Solidity, the last-written superclass within a contract's definition takes precedence in the linearization order, which accounts for the specific output observed with our algorithm. it is worth noting that our algorithm does not consider the `CTOP` class in its computations. This decision is based on the rationale that `CTOP` will always occupy the last position in the hierarchy. It also does not have any methods or state variables, essentially serving as a top-level class for distinguishing contracts from externally owned accounts (EOAs) in addresses. This allows for the use of addresses of the form `address⟨CTOP⟩`, which exclusively accept contract instances.

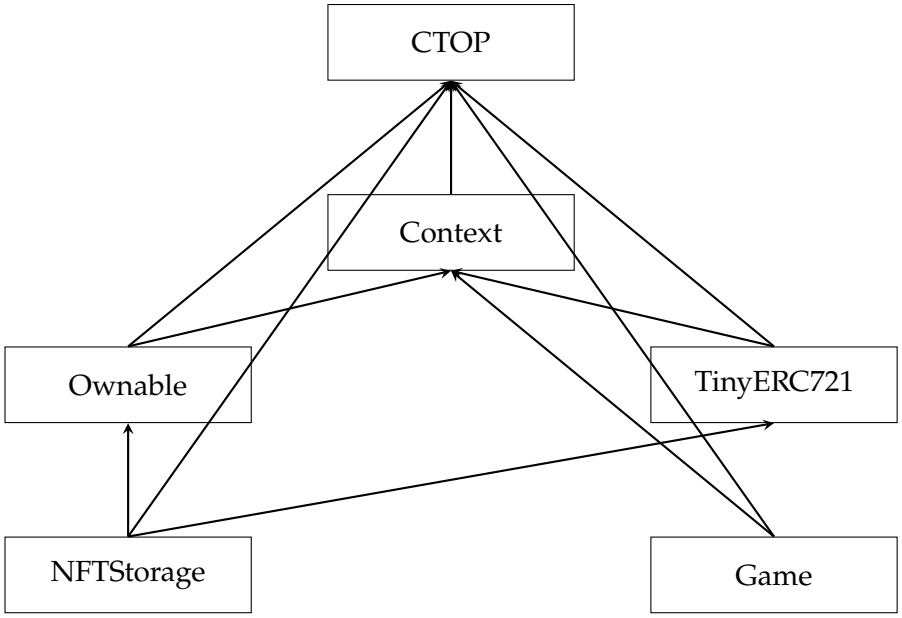

Figure 8.5: Case study class hierarchy

8.3 Execution Example

To illustrate the operational semantics, we show a possible interaction with these contracts. First, we deploy the `NFTStorage` contract into our mock blockchain, where we have two addresses, say **a1** (0xe1abd8...) and **a2** (0x229005...). Subsequently both will interact with the contract deployed and the whole blockchain through function calling (i.e., transactions).

First **a1** deploys the `Game` contract, creating an instance of it on our blockchain and we can referencing this instance either by their unique instance identifier or address. Then the same user creates a new store, invoking function `createStore`. After the transaction we have another contract on our blockchain, an instance of `NFTStorage` and it is stored in the state variable `stores`. We also return the address where this instance is located and store it because it is useful to pass on the next functions. It is important to say that this address is stored in the type environment as a subtype of the address type and therefore its refined type is `address⟨NFTStorage⟩`. Figure 8.6 shows the state of the blockchain after these first two transactions (for each contract shows the content of state variables and its balance). Next **a1** sets each NFT price to 322. In Figure 8.7 you can see that **a2** earned an NFT, by invoking the `buyNFT` function: balances and owners are two state variables of `NFTStorage` contract, both being maps. The former mapping each externally owned account to the amount of tokens this user is owner of and the latter is a mapping between each token and its owner. We repeat this transaction in Figure 8.8, making now **a2** holding two NFTs. Lastly **a2** transfers the first NFT bought to **a1**, using the `transferNFT` function, as you can see in Figure 8.9: key 0 of state variable `owners`, it changed the value and we can also note that **a1** now has a balance of 1 in state variable `balances`.

```
Contract 0, 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d, Contract Name: Game,
    State Variables:
stores ----> [
0x117c8569914f13732494006d1118888e8c4e9751 ---> Contract 1
 ]
Balance: 0

Contract 1, 0x117c8569914f13732494006d1118888e8c4e9751, Contract Name:
    NFTStorage, State Variables:
balances ----> [
 ]
lastTokenId ----> 0
lastUnsoldToken ----> 0
owner ----> 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d
owners ----> [
 ]
price ----> 0
Balance: 0

Account: 0x2290052e72090ecde135bde4c864732239f32c19  ,Balance: 100000

Account: 0xe1abd8e36dfeb46a79436f2d7a1bcfde8d70e097  ,Balance: 100000
```

Figure 8.6: Output of our tool after executing first two transactions

```
Contract 0, 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d, Contract Name: Game,
    State Variables:
stores ----> [
  0x117c8569914f13732494006d1118888e8c4e9751 ---> Contract 1
 ]
Balance: 0

Contract 1, 0x117c8569914f13732494006d1118888e8c4e9751, Contract Name:
    NFTStorage, State Variables:
balances ----> [
  0x2290052e72090ecde135bde4c864732239f32c19 ---> 1
 ]
lastTokenId ----> 1
lastUnsoldToken ----> 0
owner ----> 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d
owners ----> [
  0 ---> 0x2290052e72090ecde135bde4c864732239f32c19
 ]
price ----> 322
Balance: 0

Account: 0x2290052e72090ecde135bde4c864732239f32c19  ,Balance: 100000

Account: 0xe1abd8e36dfeb46a79436f2d7a1bcfde8d70e097  ,Balance: 100000
```

Figure 8.7: Output after the fourth transaction

```
Contract 0, 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d, Contract Name: Game,
    State Variables:
stores ----> [
0x117c8569914f13732494006d1118888e8c4e9751 ---> Contract 1
 ]
Balance: 0

Contract 1, 0x117c8569914f13732494006d1118888e8c4e9751, Contract Name:
    NFTStorage, State Variables:
balances ----> [
  0x2290052e72090ecde135bde4c864732239f32c19 ---> 2
 ]
lastTokenId ----> 2
lastUnsoldToken ----> 0
owner ----> 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d
owners ----> [
  0 ---> 0x2290052e72090ecde135bde4c864732239f32c19
  1 ---> 0x2290052e72090ecde135bde4c864732239f32c19
 ]
price ----> 322
Balance: 0

Account: 0x2290052e72090ecde135bde4c864732239f32c19  ,Balance: 100000

Account: 0xe1abd8e36dfeb46a79436f2d7a1bcfde8d70e097  ,Balance: 100000
```

Figure 8.8: Output after the fifth transaction

```
Contract 0, 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d, Contract Name: Game,
    State Variables:
stores ----> [
0x117c8569914f13732494006d1118888e8c4e9751 ---> Contract 1
 ]
Balance: 0

Contract 1, 0x117c8569914f13732494006d1118888e8c4e9751, Contract Name:
    NFTStorage, State Variables:
balances ----> [
  0xe1abd8e36dfeb46a79436f2d7a1bcfde8d70e097 ---> 1
  0x2290052e72090ecde135bde4c864732239f32c19 ---> 1
 ]
lastTokenId ----> 2
lastUnsoldToken ----> 0
owner ----> 0xa5a12d5c2d4b9af407325dd3b0d3491cf3f5304d
owners ----> [
  0 ---> 0xe1abd8e36dfeb46a79436f2d7a1bcfde8d70e097
  1 ---> 0x2290052e72090ecde135bde4c864732239f32c19
 ]
price ----> 322
Balance: 0

Account: 0x2290052e72090ecde135bde4c864732239f32c19  ,Balance: 100000

Account: 0xe1abd8e36dfeb46a79436f2d7a1bcfde8d70e097  ,Balance: 100000
```

Figure 8.9: Output after all transactions

9

Conclusions and Future Work

In this final chapter, we summarize the key findings and contributions of our research in the domain of safety on smart contracts. We also outline potential enhancements and improvements for our PoC.

9.1 Conclusion

Smart contracts are still in an early stage of development, and their adoption is growing rapidly. Ensuring their correctness through formal verification is paramount due to their immutability and the financial stakes involved, specially because they are immutable and errors can possibly have catastrophic financial consequences. Formal verification emerges as a powerful tool to achieve this goal.

Correctness can be achieved through either making a language robust against vulnerabilities or developing tools to detect and mitigate potential issues. Both approaches are essential for a safer smart contract ecosystem.

We have also shown compelling evidence indicating that domain-specific languages are significantly more suitable for writing smart contracts.

Our work has focused on formalizing an extension to a Solidity's subset, making it type safe. We have taken an existing formalization and modified it to align with the latest version of Solidity, including features like multiple inheritance between contracts. Additionally, we have implemented our formalization in OCaml to demonstrate that our solution is feasible.

9.2 Future Work

One limitation of our formalization in the context of blockchains arises from the fact that smart contracts run on shared virtual machines, increasing the likelihood of contract name collisions. To address this issue, we propose an enhancement: instead of using only the contract name, we include the hash of the contract code as part of the address type, represented as $\texttt{address}\langle C, H\rangle$, where H represents the hash of the contract code.

Furthermore, it would be beneficial for blockchains to collaborate and adopt an universal language that is compatible across all platforms, akin to JavaScript's ubiquity in web development. Having a single language would concentrate the efforts of blockchain projects and developers on ensuring the safety of one language, rather than dealing with multiple languages.

When it comes to programming languages for writing smart contracts, it is also essential to offer tested and audited libraries and frameworks, similar to the established OpenZeppelin, while promoting safe coding practices.

Bibliography

[1] V. Buterin. https://twitter.com/VitalikButerin/status/1024265820849860609. 2018 (cit. on p. 11).

[2] V. Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf. 2014 (cit. on p. 4).

[3] V. Buterin. *Hard Fork Completed*. https://blog.ethereum.org/2016/07/20/hard-fork-completed/. 2016 (cit. on p. 22).

[0] K. Claessen and J. Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*. Ed. by M. Odersky and P. Wadler. ACM, 2000, pp. 268–279. DOI: 10.1145/351240.351266. URL: https://doi.org/10.1145/351240.351266 (cit. on p. 75).

[4] *Consistency model*. https://en.wikipedia.org/wiki/Consistency_model. Accessed: 2022-05-10 (cit. on p. 5).

[5] T. H. Cormen et al. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms (cit. on p. 6).

[6] S. Crafa, M. D. Pirro, and E. Zucca. "Is Solidity Solid Enough?" In: *Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*. Ed. by A. Bracciali et al. Vol. 11599. Lecture Notes in Computer Science. Springer, 2019, pp. 138–153. DOI: 10.1007/978-3-030-43725-1_11 (cit. on p. 2).

[7] P. Daian. *Analysis of the DAO exploit*. https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/. 2016 (cit. on p. 22).

[8] C. Dwork and M. Naor. "Pricing via Processing or Combatting Junk Mail". In: *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*. Ed. by E. F. Brickell.

Vol. 740. Lecture Notes in Computer Science. Springer, 1992, pp. 139–147. DOI: 10.1007/3-540-48071-4_10. URL: https://doi.org/10.1007/3-540-48071-4_10 (cit. on p. 7).

[9] J. FRANKENFIELD. *51% Attack*. https://www.investopedia.com/terms/1/51-attack.asp. 2021 (cit. on p. 7).

[10] L. Goodman. *Tezos — a self-amending crypto-ledger White paper*. https://tezos.com/whitepaper.pdf. 2014 (cit. on p. 18).

[11] Á. Hajdu and D. Jovanović. "solc-verify: A Modular Verifier for Solidity Smart Contracts". In: *Verified Software. Theories, Tools, and Experiments*. Ed. by S. Chakraborty and J. A. Navas. Vol. 12301. Lecture Notes in Computer Science. Springer, 2020, pp. 161–179. ISBN: 978-3-030-41600-3. DOI: 10.1007/978-3-030-41600-3_11 (cit. on p. 30).

[12] A. Igarashi, B. C. Pierce, and P. Wadler. "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3 (2001), pp. 396–450. DOI: 10.1145/503502.503505. URL: https://doi.org/10.1145/503502.503505 (cit. on p. 33).

[0] *Implementing C3 Linearization*. https://xivilization.net/~marek/blog/2014/12/08/implementing-c3-linearization/ (cit. on p. 54).

[13] N. Insights. *Zilliqa: The Sharding Chain*. https://novuminsights.medium.com/zilliqa-the-sharding-chain-d25bb81c3fdc. 2022 (cit. on p. 20).

[14] J. Jiao et al. "Executable Operational Semantics of Solidity". In: *CoRR* abs/1804.01295 (2018). arXiv: 1804.01295. URL: http://arxiv.org/abs/1804.01295 (cit. on p. 2).

[15] L. Lamport. "The Part-Time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169. DOI: 10.1145/279227.279229. URL: https://doi.org/10.1145/279227.279229 (cit. on p. 5).

[16] L. Lamport, R. Shostak, and M. Pease. *The Byzantine Generals Problem*. https://www.microsoft.com/en-us/research/uploads/prod/2016/12/The-Byzantine-Generals-Problem.pdf. 1981 (cit. on p. 5).

[17] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN: 1-55860-348-4 (cit. on p. 5).

[18] R. C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*. Ed. by C. Pomerance. Vol. 293. Lecture Notes in Computer Science. Springer, 1987, pp. 369–378. DOI: 10.1007/3-540-48184-2_32. URL: https://link.springer.com/chapter/10.1007/3-540-48184-2_32 (cit. on p. 6).

[19] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. https://bitcoin.org/bitcoin.pdf. 2009 (cit. on p. 4).

[20] C. Parizo. *What are the 4 different types of blockchain technology?* https://www.techtarget.com/searchcio/feature/What-are-the-4-different-types-of-blockchain-technology. 2021 (cit. on p. 8).

[21] M. D. Pirro. *How Solid is Solidity? An In-dept Study of Solidity's Type Safety*. Msc Thesis. Università degli Studi di Padova. 2018 (cit. on p. 2).

[22] Z. D. Portal. *Sharding Mechanism*. https://dev.zilliqa.com/docs/basics/basics-zil-sharding/. 2022 (cit. on p. 20).

[23] S. N. Sunny King. *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake*. https://decred.org/research/king2012.pdf. 2012 (cit. on p. 7).

[24] N. Szabo. *Smart Contracts: Building Blocks for Digital Markets*. https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html. 1996 (cit. on p. 4).

[25] A. S. Tanenbaum and M. van Steen. *Distributed systems - principles and paradigms, 3rd Edition*. distributed-systems.net, 2017. ISBN: 978-1543057386 (cit. on p. 4).

[26] P. Technologies. *A Postmortem on the Parity Multi-Sig Library Self-Destruct*. https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/. 2017 (cit. on p. 25).

[27] R. M. Townsend. *Distributed Ledgers: Design and Regulation of Financial Infrastructure and Payment Systems*. https://direct.mit.edu/books/book/4932/chapter/625100/Preface. 2020 (cit. on p. 5).

[28] *What Is the Byzantine Generals Problem?* https://river.com/learn/what-is-the-byzantine-generals-problem/. Accessed: 2022-05-10 (cit. on p. 5).

[29] J. Zakrzewski. "Towards Verification of Ethereum Smart Contracts: A Formalization of Core of Solidity". In: *Verified Software. Theories, Tools, and Experiments*. Ed. by R. Piskac and P. Rümmer. Cham: Springer International Publishing, 2018, pp. 229–247. ISBN: 978-3-030-03592-1 (cit. on p. 2).