NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

MARIA BEATRIZ LOURENÇO E SÁ DE FERREIRA MOREIRA

Master in ⟨Computer Science and Engineering⟩

FORMALISATION OF SMART CONTRACT LANGUAGES

⟨COMPUTER SCIENCE AND ENGINEERING⟩

NOVA University Lisbon
⟨September⟩, ⟨2021⟩

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

FORMALISATION OF SMART CONTRACT LANGUAGES

MARIA BEATRIZ LOURENÇO E SÁ DE FERREIRA MOREIRA

Master in ⟨Computer Science and Engineering⟩

Adviser: António Ravara
Associate Professor, NOVA University Lisbon

Co-adviser: Mário Pereira
Researcher, NOVA University Lisbon

⟨COMPUTER SCIENCE AND ENGINEERING⟩

NOVA University Lisbon
⟨September⟩, ⟨2021⟩

Formalisation of Smart Contract Languages

Acknowledgements

A year and a half has passed, a pandemic has passed, and I would like to thank the people without whom this work would not be possible.

I would like to start by thanking Professor António Ravara and Doctor Mário Pereira for the opportunity, for their support, and for the insight they shared with me throughout this work.

I thank my friends: André for always understanding my frustrations; Guilherme for making all those all nighters feel less stressful; and my friends João and Tiago for keeping me in check. And I cannot forget Dr. Tânia Carneiro who has helped me through this journey, and made me feel heard and validated.

I would also like to thank my boyfriend Filipe, who was always there for me in times of need, who has never let me give up and has always celebrated my accomplishments , and who, whenever I was doubtful about my path, has always shined a bright light to help me see my future. I could not have asked for a better person to share my successes with.

To my family, who has always pushed me forward and allowed me to follow my dreams.

I would also like to give special thanks to my sister, Madalena, my best friend and confidant, my work partner on those long days and even longer nights, for always believing in me and for always reminding me of who I am whenever I lost sight of who I was. I cannot wait to see you grow and accomplish all the things you want in life and deserve.

And of course, my parents. To my mother who bestowed upon me her work ethic and has always held my hand through my failures, and to my father who always told me I could do everything. I thank both for working tirelessly to give me a good education and to allow me to have all these opportunities, and for that I could not be more grateful.

"Success is not final, failure is not fatal: it is the courage to continue that counts. " (Winston Churchill)

Abstract

Smart contracts automatically verify and enforce contractual agreements without the need of a trusted intermediary, as potential conflicts are resolved by the network's consensus protocol. Since "code is law", contracts should be correct, but bugs and vulnerabilities, often exploited by attackers, allow erroneous or even fraudulent behaviour. These days smart contracts are still mostly being written in general purpose programming languages, without proper specifications, let alone correctness proofs. Immutability is one of its selling points, but it is also one of its major problems, as once a contract is deployed to the blockchain it cannot be amended. Additionally, many vulnerabilities come from the misimplementation of contracts' intended behaviour, as developers struggle to grasp the behavioural impact that the contract has in the blockchain. It is thus crucial to achieve correct implementations of smart contracts.

In order to aid developers to promote the design of safer contracts that follow the protocols they are supposed to implement, we propose the use of behavioural types in smart languages. We believe that the use of typestates, for dynamic checking, and session types, for static checking, can ensure the intended behaviour of the contract before and during its execution.

To better understand a contract's behaviour throughout its execution, we took advantage of Racket (and PLT Redex), to have a visualisation of a step-by-step execution graph. By formally defining the syntax and reduction rules of a "core" smart contract language, and how each rule affects the statements and the programs configuration, this visualisation tool allows programmers to check and adjust the language's formal semantics. This is a successful proof-of-concept exercise, confirming the suitability of Racket to develop program semantics which can be analysed throughout its execution. In the context of smart contract languages, these are important features (to be combined with formal verification with proof assistants). Furthermore, we also implemented a typechecker in OCaml that provides a type derivation tree of the program, in addition to preventing the occurrence of execution errors.

To illustrate the usefulness of this approach, we took two different smart contract languages, one completely formalised with syntax, operational semantics and type system

(Featherweight solidity, FS), and another only with its natural language semantics (Flint). We formalised FS in Racket and OCaml, where we were able to detect an inaccuracy; and we repeated this process with Flint, formalising its operation semantics and type system. The latter was much more challenging as it incorporated the use of typestates. Throughout this thesis, we present many examples on how the use of visual tools can help in the developing states of contracts and better understand the correct execution of programs, as well as how the use of behavioural types can prevent many execution errors even before running.

The framework we define herein not only finds defects in the contracts, but also, crucially, detects vulnerabilities in a language construction, as we demonstrate with our use-cases. Therefore, this approach is very valuable not only for the programmer as visual debugging, but also for the language designer to test the effects of definitions.

Keywords: Executable Operational Semantics, Error Detection, Smart Contract Languages, Behavioural types, Type Systems, Programming Language Formalisation, Blockchain

Contents

List of Figures

List of Tables

List of Listings

Glossary

liveness The liveness property guarantees that something good will happen. [7] i

safety The safety property guarantees that nothing bad will ever happen. [7] i

type inference The process which consists of reasoning the type of a term within a given typed system is called type inference [41]. Similar to type checking, this process should be done at compile time. i

type checking To ensure that the program executes in accordance with the type system and prevent some execution errors related to improper use of a variable or expression type [41], the program is checked to impede these errors from happening. This process is called type checking. When the type checking is done at compile time, it is said to be a static checking, which allows the errors to be detected earlier [29]. If these checks are done during runtime, then it is a dynamic check [41]. i

type safety Property which declares that programs will not have errors which will cause an unexpected behavior. [41] i

Symbols

$\tilde{x}$ **Sequence Notation**: The symbol $\tilde{}$ represents a sequence of zero or more elements, i.e, $\tilde{e}$ represents $e_0 \cdots e_n$, for some $n \in \mathbb{N}_0$. Similarly, $\widetilde{x : e}$ defines $x_0 : e_0 \cdots x_n : e_n$ for some $n \in \mathbb{N}_0$. i

1

Introduction

With the rise of blockchain's popularity, so did smart contracts'. Blockchain is a distributed infrastructure which comprises of peer nodes. Each node stores information about the state of the network, which in itself contains a structure, ledger, that has the information about the order of the transactions that were made within the network [46]. Every transaction executed in the blockchain has to be validated by the network through a consensus protocol which assures that every node has the same ledger's copy [46]. There are two main consensus protocols, proof of work (PoW) and proof of stake (PoS).

Smart contracts are programs that describe and enforce the execution of a contractual agreement without needing to rely on a third party to enforce it [13]. Without a trusted intermediary, the network relies on a consensus protocol to reach an agreement on the order of the transactions[46]. There are many applications of these contracts in the real world, such as in the financial and banking sector - in the settlement and clearing; or in the healthcare sector - to provide reliable and easy access to information [1].

Even though these contracts seem to be perfect for certain applications, there are still many vulnerabilities that can be exploited. Like all software, smart contracts also have security vulnerabilities and bugs that can be exploited and lead to millions in losses [13]. A well-known example of an attack to the Ethereum blockchain is The DAO [37], which was a result of the exploitation of a vulnerability in the contract that managed to drained millions of dollars in Ether, cryptocurrency of Ethereum. The vulnerability was due to the variables of the contract only updating after calling the fallback function of the other contract, which subsequently lead to a loop, resulting in the continuous extraction of money; hence the importance of preventing such errors.

As a result, contract vulnerabilities are well known and categorized in order to help prevent developers to commit the same mistakes. However, this approach is not enough as many smart contract are written in general purpose programming languages that do not enforce any type of specification of correctness. Furthermore, to make matters worse, once contracts are deployed to the blockchain they cannot be modified.

As Dijkstra stated: "Program testing can be used to show the presence of bugs, but

never to show their absence!", and we believe many of the mistakes done by smart contract developers arise from the difficulty in identifying the disparity between the actual behaviour of the contract and the intended one [6].

1.1 Motivation

BlockKing [1],a smart contract with the intent of gambling money, is an interesting case study as it utilises a server outside of the blockchain, Oraclize[2], which in turn "invites true concurrency" [36].

The behaviour of the BlockKing contract is defined by having, at any moment, a "block king", which in the beginning is assigned to the writer of the contract. A new BlockKing is appointed whenever a client sends money to the contract, and the modulo 10 of its current block number (`singleDigitNumber`), which finds a number between 0 and 9, ends up being the same as the random number *j* generated by a trusted third party, line 341. The now BlockKing is then sent a percentage of the money in the contract.

In Figure 1.1, we present a few snippets of the BlockKing's Solidity code. The main function is `enter`, called whenever a client *s* sends money to the contract. Through lines 299-301, we can see information of *s* being stored in single valued variables. In line 303, the contract queries Oraclize for a random number. The Oraclize's response enters through function `__callback` (line 305), which then triggers the function `process_payment` in line 308. This last function is responsible for checking if the current warrior's `singleDigitBlock` matches the random number sent by Oraclize (line 340), and if so, assigns it the new king (lines 347 and 348).

As Oraclize services are outside the blockchain, its response might not be immediate, which means that by the time the BlockKing contract gets its random number, its state can be significantly different. As stated before, all senders' information are stored in single valued variables, which can be a problem. In between the random number query and the response from Oraclize, another client can enter the gamble, consequently rewriting the `warrior` variables, and by the time of its response, the latter client has two chances of winning, as it takes the previous client random number.

These kind of mistakes are common in smart contracts, but cannot permissible, as these programs deal with great amounts of valuable assets; hence the importance of preventing such errors.

1.2 Contributions

As testing alone is not sufficient, the need for tools to aid programmers in the design of smart contracts and ensure safety before deployment is crucial. For this reason, we advocate for behavioural types in smart languages, which have already been proven

[1] https://etherscan.io/address/0x3ad14db4e5a658d8d20f8836deabe9d5286f79e1
[2] http://www.oraclize.it

```
293 function enter() {
294     // 100 finney = .05 ether minimum payment otherwise refund payment and
            stop contract
295     if (msg.value < 50 finney) {
296       msg.sender.send(msg.value);
297       return;
298     }
299     warrior = msg.sender;
300     warriorGold = msg.value;
301     warriorBlock = block.number;
302     bytes32 myid = oraclize_query(0, "WolframAlpha", "random number between 1
            and 9");
303   }
304
305   function __callback(bytes32 myid, string result) {
306     if (msg.sender != oraclize_cbAddress()) throw;
307     randomNumber = uint(bytes(result)[0]) - 48;
308     process_payment();
309   }
310
311   function process_payment() {...
312     ...
340   singleDigitBlock = singleDigit;
341   if (singleDigitBlock == randomNumber) {
342       rewardPercent = 50;
343       // If the payment was more than .999 ether then increase reward
              percentage
344       if (warriorGold > 999 finney) {
345       rewardPercent = 75;
346     }
347       king = warrior;
348       kingBlock = warriorBlock;
349     }
```

Figure 1.1: BlockKing code fragments

successfully in object oriented programming language, as well as in communication-centered ones, in order to promote safe smart contract design.

We present herein a proof-of-concept: an approach that takes advantage of both typestates and session types. We introduce Flint-2, a smart contract languages with typesstates for dynamic checking, to session types, in order to promote static verification. Furthermore, throughout our work we introduce two visual debugging tools, PLT Redex and OCaml typechecker to display all typing derivations in order to get a better understanding of the actual behaviour of the contracts.´

This document is structured as follows: Chapter 2 presents the state of the art study on Blockchain, smart contracts, Racket and behavioural types. In Chapter 3 we introduce Featherweight Solidity, a calculus that models the core of Solidity. We present our implementation and running examples with Racket, as well as a typechecker in OCaml. This language was completed with syntax, operational semantics and typing system formalisation, which we found to be a great base to our own implementation

of another smart language. In Chapter 4 we present our own formalisation of Flint-2, and our implementation in Racket and OCaml. We finish the chapter with the integration of session types to Flint-2, and another example that proves the efficiency of behavioural types. Chapter 5 summarises the accomplishments of our work and what we want to do form future work. The work we developed is available at https://bitbucket.org/beatrizmoreira/msc/src/master/thesis/.

2

Background

2.1 Distributed Systems: Concepts

In this section we will present some relevant concepts, such as the CAP properties, what is blockchain and some problems that arise from it, what are smart contracts and behavioural types.

2.1.1 CAP Properties

As Gilbert and Lynch [23] state, it is impossible for a web service to provide the following guarantees:

- Consistency
- Availability
- Partition-tolerance

Web services attempt to achieve consistency in their database by relying on ACID properties. ACID stands for Atomic (either the operations are committed or fail in their entirety), Consistent (all transactions must result in consistent data), Isolated (uncommitted transactions are isolated from each other), and Durable (once a transaction is committed it is permanent) [23]. Likewise, web services are also expected to be always available. And finally, on a distributed network, if any component fails, it should still perform as expected.

2.2 Blockchain

A Blockchain is a distributed infrastructure within a network of peer nodes. Each node maintains a copy of the ledger which consists in a "transaction log structure as hash-linked blocks of transaction" [24]. Each block contains an ordered set of transactions, and a hash that links each block to the previous one. A block can only be added to the blockchain if it has been validated by a consensus protocol, and once a block is added to

the chain it cannot be changed, making blockchain's ledgers immutable in comparison with other distributed ledgers. Blockchain provides ledger and smart contract services to applications.

Blockchains can be classified into three main categories [46]:

- Public Blockchain: Anyone can read, send and receive transactions, and any participant can join the consensus procedure.
- Consortium Blockchain: Has some constraints when it comes to writing permissions, only pre-selected participants of the network can influence the consensus procedure, whilst reading permissions are given to any participant.
- Private Blockchain: The writing permissions are restricted to a single participant (or organization), despite the reading permissions being open to anyone.

2.2.1 Consensus Algorithms

A consensus algorithm is used to reach an agreement within the network in order to validate the new blocks added to the chain, and to assure that every peer has the same order of transactions in its ledger's copy [46]. The problem of reaching consensus in a network is that this process may be manipulated by malicious actors and faulty processes. The "Byzantine Generals Problem" 2.2.1.1 is when the network fails to reach consensus due to faulty actors. This problem will be explained with detail in section 2.2.1.1 and in sections 2.2.1.2 and 2.2.1.3, we will discuss two of the most used consensus algorithms.

2.2.1.1 Byzantine Fault Tolerance

Byzantine Fault Tolerance is defined as the failure tolerance of a system against the Byzantine Generals' Problem (BGP) [46].

This problem is an abstraction that conveys the system components as generals, in which their main goal is to reach consensus over the plan of attack. This problem also describes how a reliable system should deal with faulty components that send messages to the system to mislead others. The solution to this problem was proposed by Leslie Lamport, Robert Shostak and Marshall Pease [28] that says that the problem can be solved if more than 2/3 of the generals are honest, and if there are only 3 generals and one of them is a traitor, then there is no possible solution. For it to work, the algorithm must guarantee that the generals satisfy the following properties [28]:

1. All honest generals must decide the same plan, that is, an honest general will always execute the algorithm correctly, whilst the traitors can behave in anyway. The algorithm must guarantee this condition, and the honest generals must reach an agreement and also come up with a reasonable plan. For this condition to be satisfied, the following must be true:

a) Any two honest generals must get the same value as the other honest general.

b) If the general is honest, then the value that he sends must be used by any other honest general that receives it.

2. A small number of traitors will not be able to cause the honest generals to follow the wrong plan. As the final decision is based on the majority of the votes, a small number of traitors can influence the decision of the honest ones but only if they are torn between two possible plans, in which case both plans are reasonable.

With this said, we will analyse two consensus algorithms that provide a probabilistic solution to the Byzantine Generals' Problem [46].

2.2.1.2 Proof of Work (PoW)

This consensus protocol is characterized by the following properties [46]:

1. It should be difficult and time consuming for any participant to produce a proof that meets certain requirements, this is, it is hard to find a correct nonce.

2. It should be easy and fast to verify if the proof is correct, in other words, it is easy to validate the resulting hash.

This protocol is efficient in terms of solving the BGP, but it suffers other limitations like the following [46]:

1. It is inefficient due to the great computational complexity and low probability of generating a successful proof of work.

2. The security of this protocol derives from the reward associated to the creation of blocks, which attracts a large number of participants, which may be a problem when trying to reach consensus.

3. The fact that the participants may have different levels of computational capabilities, which leads to different probabilities of successfully generating proof of work.

2.2.1.3 Proof of Stake (PoS)

Unlike PoW, this protocol doesn't depend on incentives to guarantee security, it promotes penalty-based solutions [46]. It sets the constraint that only participants "who have locked up their capital as deposits" (stake) can be chosen to be miners or validators. Anyone can become a participant as long as they send "a special type of transaction to lock up a certain amount of their coins" [46]. The blockchain maintains a record of the set of validators that have shown proof of stake.

To add a block in the chain, each validator will place a bet on the block in order to qualify as validator for the block. If the block gets added to the chain, then all the

validators that bet on it will be rewarded. So, unlike PoW in which security is achieved by rewarding the "burning of computational energy", in PoS security is ensured by penalizing the ones that cause economic losses.

2.2.1.4 CAP Properties in Blockchain

As it was previously presented, all web services try to achieve the CAP properties, and blockchain is no different. In blockchain achieving them means [46]:

- Consistency: All peer nodes have the same ledger with the most recent update.
- Availability: Any transaction made will be accepted by the ledger.
- Partition Tolerance: If any node fails, the network can still operate.

2.3 Smart Contracts

A smart contract is code used for definition of protocols between different organisations, that intends to automatically verify or enforce contractual agreements just like a traditional contract [40]. When these contracts are invoked, they generate transactions that are recorded on the ledger [24]. Blockchain is a great platform as it is immutable, and also if a "smart contract is properly implemented it leaves little space for corruption, and eliminates the need for third-party authentication" [1].

In the following sections we will present some real world applications of smart contracts and vulnerabilities that can be exploited.

2.3.1 Real World Examples

With smart contracts' popularity growing over the years, many believe that they will revolutionize many industries as they will replace humans. In this section, we will analyse some cases where smart contracts are being utilized.

One of the sectors that most benefits from the blockchain technology is the financial and banking ones, as its immutable properties make the blockchain perfect for storing financial information and records [1]. Additionally, the settlement and clearing process must guarantee that the money was indeed transferred and that all parties involved in the exchange updated their respective accounts [1]. This can be ensured by smart contracts as they enforce and verify if all participants respected the agreements in the contract [40]. Besides, by doing this automatically, it reduces the risk of human error [1].

Another area in which smart contracts are being used is healthcare, as they provide reliable and easy access to patients' data. Some applications, like Ethereum's MedRec [1], try to tackle the issue of data scatter throughout multiple organisations by having a single platform where you can keep and access every patient's record.

[1] https://medrec.media.mit.edu

In the real estate market, smart contracts can also be used to store all types of documents and records from buyers and sellers. With blockchain contracts, intermediates can be removed and payments can be done through it. Propy [2] is an application that intends to integrate the real estate market model in the blockchain to make the whole process easier and more secure [1].

Just like the examples above, the education sector benefits from blockchain for record keeping, and also for organisation and verification of diplomas and certificates in order to eliminate its forgery [1].

2.3.2 Problems

Since these contracts deal with such valuable assets, programmers must be extra wary of attacks that aim to steal or tamper with them [6], as these vulnerabilities cause a lot of money loss. For example, the DAO attack, which stands for Decentralised Autonomous Organisation, was a crowdfunding platform implemented in the Ethereum blockchain, that loss approximately 60M$ by exploiting a vulnerability where the variables of the contract only updated after calling the other contracts fallback function.

To exemplify the attack, we present a simplified version of the protocol taken from [6], where in 2.1 contracts can call function `withdraw` to take from the account a specific amount, and in 2.2 participants can `donate` money to any contract of their choice.

```
function withdraw(uint amount) {
    if (credit[msg.sender]>= amount
        ) {
        msg.sender.call.value(
            amount)();
        credit[msg.sender]-=amount;
    }
}
```

Listing 2.1: Withdraw function

```
contract SimpleDAO {
    mapping (address => uint)
        public credit;
    function donate(address to){
        credit[to] += msg.value;
    }
    function queryCredit(address to
        ) returns (uint){
        return credit[to];
    }
}
```

Listing 2.2: Simplified DAO Contract

To initiate the attack, the adversary needs to publish the contract in 2.3, and then `donate` some money to `Mallory`. Invoking the command `call` when withdrawing, triggers the fallback function of `Mallory` which calls the function `withdraw` that transfers money to Mallory. As the update of the `credit` variable is only done after the `call` function, this will subsequently loop until there is no more money in DAO, or it runs out of gas, resulting in the continuous extraction of money from the account.

```
contract Mallory {
    SimpleDAO public dao = SimpleDAO(0x354...);
    address owner;
```

[2] https://propy.com/browse/about/

```
4       function Mallory(){
5           owner = msg.sender;
6       }
7       function() {
8           dao.withdraw(dao.queryCredit(this));
9       }
10      function getJackpot(){
11          owner.send(this.balance);
12      }
13  1
```

Listing 2.3: Mallory Contract

Other problems from Ethereum's Solidity arise from the difficulty that programmers have of expressing themselves with Solidity's semantics[6]. These weaknesses are known and well described with test cases for each and everyone in [11], as these security breaches have to be kept in mind as they can lead to millions in losses.

2.4 Racket

Racket is a programming language in the Lisp family, which offers a rich language to help create other programming languages, by offering multiple tools for programmers to define their language and to implement it [16].

PLT Redex consists of a domain-specific language that was developed to specify reduction semantics [18]. To model a programming language in Redex, the programmer must formulate the grammar and its reduction rules [15]. As Redex is embedded in Racket, all its features are also available, including the one used to developed our work, DrRacket. This allows the generation of reduction graphs automatically, which then enables programmers to visualise the step-by-step reductions. Pattern matching and judgement-form evaluation, are other key features of Redex, which can be use, respectively, for grammar and type-system testing.

2.4.1 Typed Arithmetic Expressions

In this section we introduce a simple language from Pierce [33] of typed boolean and arithmetic expressions in order to demonstrate how we implement other, more complex, languages. This language consists of the boolean constants `true` and `false`; the if-expression; the constant 0; the arithmetic functions for successor (`succ`) and predecessor (`pred`); and finally the function `iszero` that evaluates a term and returns `true` when it is 0, and `false` when it is not. The language has two types, `Bool` and `Nat`, to distinguish between numeric and boolean values. The syntax is presented in Figure 2.1. Furthermore, we also present the evaluation context of AE in Figure 2.2.

In Figure 2.3, we can see the semantic rules. The first three rules regard the evaluation of the conditional (E-IFTRUE, E-IFFALSE, E-IF); then E-SUCC, E-PRED and E-ISZERO

(*Terms*)	t	::=	true | false | if t then t else t |
			0 | succ t | pred t | iszero t
(*Values*)	v	::=	true | false | nv
(*Numeric Values*)	nv	::=	0 | succ nv
(*Types*)	T	::=	Bool | Nat

Figure 2.1: Syntax of AE

$$E \quad ::= \quad [] \mid \text{if } E \text{ then } t \text{ else } t \mid \text{succ } E \mid \text{pred } E \mid \text{iszero } E$$

Figure 2.2: Evaluation Context of AE

that evaluate their subterm; and finally the computation rules, which determine how `pred` and `iszero` behave when applied to numbers [33].

$$\frac{t \to t'}{\langle \text{if } t \text{ then } t_1 \text{else } t_2 \rangle \to \langle \text{if } t' \text{ then } t_1 \text{else } t_2 \rangle} \text{ E-IF}$$

$$\frac{}{\langle \text{if true then } t_1 \text{else } t_2 \rangle \to \langle t_1 \rangle} \text{ E-IFTRUE} \qquad \frac{}{\langle \text{if false then } t_1 \text{else } t_2 \rangle \to \langle t_2 \rangle} \text{ E-IFFALSE}$$

$$\frac{t_1 \to t_1'}{\langle \text{succ } t_1 \rangle \to \langle \text{succ } t_1' \rangle} \text{ E-SUCC} \qquad \frac{t_1 \to t_1'}{\langle \text{pred } t_1 \rangle \to \langle \text{pred } t_1' \rangle} \text{ E-PRED}$$

$$\frac{t_1 \to t_1'}{\langle \text{iszero } t_1 \rangle \to \langle \text{iszero } t_1' \rangle} \text{ E-ISZERO}$$

$$\frac{}{\langle \text{pred}(0) \rangle \to \langle 0 \rangle} \text{ E-PREDZERO} \qquad \frac{}{\langle \text{pred}(\text{succ}(nv)) \rangle \to \langle nv \rangle} \text{ E-PREDSUCC}$$

$$\frac{}{\langle \text{iszero}(0) \rangle \to \langle true \rangle} \text{ E-ISZEROZERO} \qquad \frac{}{\langle \text{iszero}(\text{succ}(nv)) \rangle \to \langle false \rangle} \text{ E-ISZEROSUCC}$$

Figure 2.3: Operational semantic rules of AE

Lastly, we present the typing rules in Figure 2.4 that assign types to terms. Both `true` and `false` are assigned `Bool` by the inference rules T-TRUE and T-FALSE respectively. Rule T-IF assigns a type to the conditional that is derived from the type of the subterms, t_1 has to evaluate to `Bool`, and both t_2 and t_3 have to evaluate to the same type T.

The last three rules evaluate to `Nat`, T-Succ and T-Pred' sub-expression must evaluate to `Nat`, and T-IsZero gives the type `Nat` as long as its sub-term evaluates to `Bool`.

$$\frac{}{\text{true} : \text{Bool}}\ \text{T-TRUE} \qquad \frac{}{\text{false} : \text{Bool}}\ \text{T-FALSE}$$

$$\frac{t_1 : \text{Bool} \quad t_2 : \text{T} \quad t_3 : \text{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \text{T}}\ \text{T-IF}$$

$$\frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}}\ \text{T-SUCC} \qquad \frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}}\ \text{T-PRED} \qquad \frac{t_1 : \text{Nat}}{\text{iszero } t_1 : \text{Bool}}\ \text{T-ISZERO}$$

Figure 2.4: AE's Typing Rules

2.4.1.1 Grammar

In Racket, we start by defining a language with the `define-language` form to name our grammar, which in this case is `AE`, and then specifying the syntax, values and evaluation context.

Referencing Figure 2.1, we can define our syntax as t, and the terminal terms as v. The evaluation context is defined in Figure 2.2, and will decide where in the term the evaluation will occur.

The grammar of AE is defined in Figure 2.5, and as we can see it is nearly identical to its formalization, being the substitution `hole` term the only difference, but its meaning is the same as [].

2.4.1.2 Relation Rules

Now that we defined our language in Racket, the next step is to write the operational semantic rules. To do so, one must use the `reduction-relation` form, which receives the language name, domain and rules.

Every rule has the form `-> pattern pattern rule-name`. These rules work by *pattern-matching*, which means that every pattern going into the rules has to match its domain. Since the terms t are bound to be evaluated according to our evaluation context `E`, we use the `in-hole` pattern which decomposes the patterns into some E that will match to the succeeding pattern. This is why the operational semantic rules presented in Figure 2.6 are less than the ones presented in 2.3, as rules E-If, E-Succ, E-Pred and E-IsZero are described in the evaluation context of the grammar in Figure 2.5.

As we can see in the rules presented in Figure 2.6, the first pattern we present left side of the rules, and on the second one we write what the first term evaluates to.

2.4.1.3 Reduction Graph

Now that we presented our Racket implementation, we can test it and show the step-by-step execution graph. The expression is the following:

```
(define-language AE
  ;Syntax
                              ;terms
  (t ::= true                 ;constant true
     false                    ;constant false
     (if t then t else t)     ;conditional
     0                        ;constant zero
     (succ t)                 ;successor
     (pred t)                 ;predecessor
     (iszero t)               ;zero test
     )
                              ;values
  (v ::= true                 ;true value
     false                    ;false value
     nv                       ;numeric values
     )

                              ;numeric values
  (nv ::= 0                   ;zero value
      (succ nv)               ;successor value
      )

  ;Context Evaluation
  (E ::= hole
     (if E then t else t)     ;E-IF
     (succ E)                 ;E-SUCC
     (pred E)                 ;E-PRED
     (iszero E)               ;E-ISZERO
     )
 )
```

Figure 2.5: AE's Grammar in Racket

`if (iszero 0) then (pred (succ 0)) else (succ 0)`,
and its evaluation produces the graph in Figure 2.7.

2.4.1.4 Type System

In Racket, we are also able to check if from a type environment we are able to derive a type. To do so, we need to firstly define a type environment by extending the existing language. AE does not have a type environment as it is a really simple language.

Afterwards, we write the required typing rules. We use `define-judgement-form`, which allowed us to implement them, by defining the shape of our judgements through `#:contract`, and specifying how Redex should compute the derivations, either as inputs or outputs in `#:mode`. AE's judgements follow the $\vdash$ `t T` pattern, where `t` is thought as input, as it is the term that is judged, whereas `T` is the output, as in the type of the term.

By looking at Figure 2.8, we can see that the judgements in our Racket code follow the same lines as the typing rules presented in Figure 2.4, since the judgements above the dotted line must check in order for the one below to hold.

If our judgements hold, Racket will print out `t`, and `f` if not.

```
(reduction-relation AE
  #:domain (t)
   ;Arithmetic Expressions
   (--> [(in-hole E (if true then t_1 else t_2))]
        [(in-hole E t_1)]
        "E-IFTRUE")
   (--> [(in-hole E (if false then t_1 else t_2))]
        [(in-hole E t_2)]
        "E-IFFALSE")
   (--> [(in-hole E (pred 0))]
        [(in-hole E 0)]
        "E-PREDZERO")
   (--> [(in-hole E (pred (succ nv)))]
        [(in-hole E nv)]
        "E-PREDSUCC")
   (--> [(in-hole E (iszero 0))]
        [(in-hole E true)]
        "E-ISZEROZERO")
   (--> [(in-hole E (iszero (succ nv)))]
        [(in-hole E false)]
        "E-ISZEROSUCC")
   )
```

Figure 2.6: AE's Operational Semantic Rules in Racket

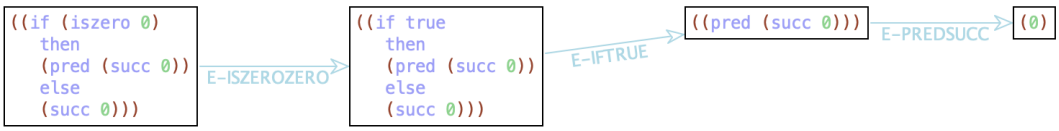

Figure 2.7: AE's Reduction Graph

2.5 Behavioural Types

It is important to ensure that a program correctly executes its defined protocol [43]. A program is considered to be well behaved if it follows its supposed execution flow. Programming languages rely on type systems to prevent execution errors and to determine whether or not a program is well behaved, which inherently ensures its safety [41].

Typed systems analyse the variables and expression types of typed languages to determine if the program follows the specification rules. [41]

Behavioural types, initially proposed by Nielson and Nielson [32], define a program's expected patterns of execution, which are then used to determine if a sequence of interaction is permissible according to its specification [25]. Additionally, behavioural types are known to describe notions such as causality, choice and resource usage.

Two approaches to behavioural types are session types and typestates that are presented in the following sections.

```
(define-judgment-form
  AE
  #:mode(|- I O)
  #:contract(|- t T)
  [-------------
   (|- true Bool)]                          ;T-TRUE
  [-------------
   (|- false Bool)]                         ;T-FALSE
  [(|- t_1 Bool) (|- t_2 T) (|- t_3 T)
   --------------------------------
   (|- (if t_1 then t_2 else t_3) T)]       ;T-IF
  [---------
   (|- 0 Nat)]                              ;T-ZERO
  [(|- t_1 Nat)
   ------------------
   (|- (succ t_1) Nat)]                     ;T-SUCC
  [(|- t_1 Nat)
   ------------------
   (|- (pred t_1) Nat)]                     ;T-PRED
  [(|- t_1 Nat)
   ------------------
   (|- (iszero t_1) Bool)]                  ;T-ISZERP
 )
```

Figure 2.8: AE's Typing Rules in Racket

2.5.1 Typestates

One notion of behavioural types is typestates, which can be preceded to 1986 in Strom and Yemini's [39]. Typestates are similar to finite-state machines [25], as a typestate defines the set of functions that are allowed for a certain entity at a certain state. These states are associated to types, allowing the program to be statically verified at compilation time and check its correctness, i.e., if all resulting sequences of procedures are valid.

Garcia et al [21] established this idea of typestate-oriented programming by integrating the typestate concept, where each object type also includes access permissions and state guarantees. The Plaid language, developed by Aldrich [4], has typestates as its central concept.

Mungo [2, 27] is a type-checking tool for Java that statically checks the sequence of permitted calls for each class. This tool validates implementations by comparing them to their type state definitions, and it works by associating classes with their respective state machines which define their sequence of permitted method calls. To link the class to a typestate definition, the user must add the annotation `@Typestate` to the class with the name of the typestate definition file, `@Typestate("OwnerProtocol")` [27]. This declaration allows Mungo to ensure that all instances of the class with that annotation are handled correctly according to the typestate declaration. Each protocol is defined as a state machine, where for each state there is a set of possible methods that can be executed; and for each method a successor state is specified. In 2.4, we present a possible

implementation of a typestate protocol for a stack from [27].

```
typestate StackProtocol {
    Empty    = {void push(int): NonEmpty,
                void deallocate(): end}
    NonEmpty = {void push(int): NonEmpty,
                int pop(): Unknown}
    Unknown  = {void push(int): NonEmpty,
                Check isEmpty(): <EMPTY: Empty,
                                  NONEMPTY: NonEmpty >}
}
```

Listing 2.4: Stack Typestate Protocol

The definition of the stack implies that the initial state is `Empty`. From the `Empty` state we can either do `push(int)`, where it pushes an integer into the stack and it is no longer empty (state `NonEmpty`); or do `deallocates()` where it frees all the resources and terminates the procedure. This method can only be called from the `Empty` state, requiring the client to empty the stack beforehand [27]. The following declarations of state are just as straight forward, where from the state `NonEmpty`, we can only call `push(int)`, which after execution remains in the same state; or `pop()`, from which the succeeding state is `Unknown` as it depends on the number of elements in the stack. From the `Unknown` state we can then execute `push(int)` and go to the state `NonEmpty`; or call `isEmpty()` which tests the size of the stack and returns either `EMPTY` or `NONEMPTY` which, depending on the result, will proceed to state `Empty` or `NonEmpty`, respectively.

Mungo's typestate inference guarantees that the program will behave accordingly to the typestate protocols [27], by associating classes to state machines which then define the permitted sequences of method calls.

2.5.2 Session Types

Session types provide a specification of the program's behaviour [44], enforcing the sequence of permitted function calls using static verification in a concurrent environment. The use of session types provides a great tool to developers due to it being statically verifiable and ensuring type-safety, as well as forcing the programmer to reason over the intended behaviour of the program.

The integration of session types in the formulation of programming languages has been attempted by many. Neubauer and Thiemann [31] proposed an Haskell based core language with session types. Capecchi et al. [9] present an object-oriented language which focuses on sessions. Gay et al. [22] propose a amalgamation of session-typed communication channels and distributed object-oriented programming, where different states within an object allow for different permitted operations.

MOOL (Mini Object-Oriented Language) [8, 42], is based in the Java programming language, which takes advantage of session types. MOOL was based on the work of Gay

```
1  class Auction {
2      usage lin init;
3                *{bid + getInitialPrice + getMaxBid + getBidder };
4      unit init(string item, int initPrice) {
5          ... // initialize fields
6      }
7      sync unit bid(int pid, int bid) {
8          if(maxBid <= bid) {
9              bidder = pid ;
10             maxBid = bid;
11         }
12     }
13     ... // the getters
14 }
```

Figure 2.9: An auction implementation in MOOL [8]

et al. [22] and their proposal of a object-oriented programming language with session types, also known as modular session types. The modularity aspect derives from the fact that session is implemented over a set of methods instead of individual operations.

This core language has constructs that allow programmers to specify usage protocols, which define how a object should be used and accessed [8] based on its state. An usage type can then be defined as the combination between the state and the status of an object. An object can only be granted access if the aliasing restrictions allow it. Thus, there is a distinction between linear and unrestricted status. A linear status means that only one client can reference a specific object, whereas an unrestricted, or shared, status can be used by many. By virtue of usage types, MOOL's type system allows for the static verification of the correct sequence of operations, as well as aliasing.

In Figure 2.9, we present an auction implementation to illustrate the use of usage types. In line 2, we specify that the constructor of the object `Auction` is `init`, and `lin` denotes that only one user can reference it. Subsequently, the object status changes to unrestricted, here characterised by *, allowing multiple users to repeatedly call any of the following methods: `bid`, `getInitialPrice`, `getMaxBid` and `getBidder`.

3

Featherweight Solidity

Solidity is an imperative language that has a similar syntax to JavaScript which, means that it is simple and developers may be able to learn it more quickly [30]. The Solidity code is executed in the EVM, and developers need to set an amount of gas for the contract, which needs to be enough for the contract to execute completely, as each line needs some amount of gas to be executed [30]. Furthermore, Solidity is statically typed and also supports complex types [38]. In 3.1, we present the function `bid` from an example of auction taken from [38]. The `onlyBefore(biddingEnd)` declaration checks if the auction has not ended, and only if this is verified, can the client bid on the auction.

```
1  contract BlindAuction {
2      function bid(bytes32 _blindedBid)
3          public
4          payable
5          onlyBefore(biddingEnd)
6      {
7          bids[msg.sender].push(Bid({
8              blindedBid: _blindedBid,
9              deposit: msg.value
10         }));
11     }
12 }
```

Listing 3.1: Example of an Auction in Solidity

Featherweight Solidity (FS) [14] is a calculus that models the core of Solidity. Its intent is to allow the study of smart contracts, such as contract deployment, interaction among contracts and money transfers, as it includes formalised operational semantics and a type-system. It is inspired by two other calculi, FJ [26] and DJ [3], and it is aimed to prove type safety.

(*Contract decl.*)	SC	::=	$\texttt{contract}\ C\ \{(\widetilde{T\ s})\ K\ \widetilde{F}\}$
(*Constructor decl.*)	K	::=	$C\ ((\widetilde{T\ x}))(\widetilde{this.s = x}))$
(*Function decl.*)	F	::=	$T\ f\ ((\widetilde{T\ x}))\{\texttt{return}\ e\}$ |
			$\texttt{unit}\ f\,b()\ \{\texttt{return}\ e\}$
(*Expression*)	e	::=	$v \mid x \mid b \mid \texttt{this} \mid \texttt{this}.f \mid$
			$\texttt{msg.sender} \mid \texttt{msg.value} \mid$
			$\texttt{address}(e) \mid e.s \mid e.\texttt{transfer}(e)$
			$\texttt{new}\ C.\texttt{value}(e)((\widetilde{x : e}))(c\ a) \mid$
			$e\ e \mid T\ x = e \mid x = e \mid e.s = e \mid$
			$e[e] \mid e[e \rightarrow e] \mid e.\texttt{value}(e)(\widetilde{x : e}) \mid$
			$e.f.\texttt{value}(e)(\widetilde{x : e}) \mid \texttt{revert} \mid$
			$e.f.\texttt{value}(e).\texttt{sender}(e)(\widetilde{x : e}) \mid$
			$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
(*Values*)	v	::=	$\texttt{true} \mid \texttt{false} \mid n \mid a \mid \texttt{u} \mid M \mid$
			c
	vv	::=	$v \mid c.f$
(*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 3.1: Modified Syntax of Featherweight Solidity [14]

3.1 Racket Implementation

In this section we present our implementation of Featherweight Solidity using PLT Redex, starting with the grammar and the modifications we made due to Racket's syntax in section 3.1.1; its context evaluation in 3.1.2.1; how we model the environments in 3.1.2.2; and how we implemented the operational semantic rules in 3.1.2.3.

3.1.1 Syntax

We implemented FS as presented in [14] using PLT Redex. Due to Racket's syntax, we had to adapt its grammar, such as:

- Every expression must be in between parenthesis.
- ; is reserved by Racket for comments in the code, so expressions cannot be separated using a semicolon.
- . is reserved by Racket, so we replaced it with a →.
- ... denotes the repetition of the previous pattern 0 or more times.
- −− > denotes the representation of the arrow in the statement $e[e \rightarrow e]$.

3.1.1.1 Revised Syntax

In Figure 3.1 we present the modified syntax of the FS language. The following are the changes we made to the original FS' grammar:

- `return` e was added to e since in Racket our semantic rules evaluate expressions of e, therefore all expressions in the evaluation context must be in e.
- To simplify, instead of the program generating random references and addresses to the contracts, we provide in the instantiation of a new contract, its respective contract reference (c) and address (a) to make it easier to understand and to be similar to the examples presented in [14].
- We separate the *values* in v and vv, so we can separate the value `c -> f` from the other terminal values, as it is a special value which identifies a function and can be evaluated. This is because the Racket pattern-matching cannot distinguish function instantiations from state variable calls, as both are represented by the pattern $x \rightarrow y$. Therefore, it can result in multiple evaluation branches, as the syntax allows for variables to not reach their terminal values. This special case, required a special call rule, for when we have functions passed in its arguments.
- Instead of only passing the values in the calling functions, in our implementation the parameters names have to be written explicitly as such: $(x : v)$.

Values. In Racket, the terminal values were declared as such:

- x, s, C as `variable-not-otherwise-mentioned`, which matches any symbol except the ones that are used as literals.
- f as `variable`, matches any symbol
- c, a as `variable-prefix`, every variable has to begin with a c and a, respectively.
- M is defined as set of key-value pairs of types v, which we denote as $(v_k :: v_v)$
- `u` is a literal.
- n is a `number`.

3.1.2 Operational Semantics

In this section we present the operational semantics of the Featherweight Solidity language. It is a binary relation between configurations $\mathscr{C}$, which describes the state of the execution. The configuration of the FS language is defined by a tuple $\langle e, \beta, \sigma \rangle$, which is comprised of an expression that is going to be evaluated over the following environments: the blockchain and the call stack. Both environments store the state of the program, in which the blockchain stores the information of the contracts; the call stack keeps record

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

Figure 3.2: Evaluation Contexts of Featherweight Solidity

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

Figure 3.3: Environments of Featherweight Solidity

of the addresses that call functions during the current transaction, and when it is empty it means that the transaction was successful; and the typestates stack which tracks the type states of each contract.

3.1.2.1 Context Evaluation

The context of evaluation remains almost the same as in [14], with the only changes being on the evaluation of parameters and the representation of the hole pattern, but only because of Racket's syntax.

3.1.2.2 Environments

The environments in our implementation are the same as the ones in [14], as presented in Figure 3.3. Although not explicit in the formalisation, in Racket we need the contract table CT in the domain, in order to match functions' inputs and return function bodies.

We now explain each of the productions of the grammar in the figure:

Blockchain. The β represents the blockchain, and follows the formalisation of the Ethereum one. β maps pairs (c, a) to triples $(C, \widetilde{s : v}, n)$, and variable identifiers x to values v. The unique pair of unique identifiers (c, a) represents the contracts reference and its address respectively, and it is associated to the name of the contract C, the contract's state variables $(\widetilde{s : v})$, and its balance n, and also to its instantiated variables. We made this change in order to be able to call functions multiple times, as when the variable

is already in the scope of the contract, it is pushed out and its value modified. This allows us to keep a record of all variables' values.

A Call Stack σ tracks the addresses of the contracts that performed function calls within the execution of a transaction [14]. At the bottom of the stack, there is a copy of the blockchain at the beginning of the transaction. In case of abort, the state of the blockchain has to be unrolled to the beginning of the transaction, which is the state on the bottom of the call stack. If the transaction is proven successful, β will be a copy of σ. This stack grows, as top-level calls are made, and addresses of those contracts are appended to σ. Upon a function return, an element a is popped from σ until it becomes β again.

The Contract Table maps contract names to their declaration SC.

3.1.2.3 Reduction Rules

In Racket, to define the relation rules, one has to define its domain and co-domain. They are the following: an expression e; a blockchain environment (env-β); a call stack (env-σ); and a contract table CT.

These rules work by *pattern-matching*, which means that every pattern going into the rules has to match each domain. Since the expressions e are bound to be evaluated according to our evaluation context, we use the `in-hole` pattern which allows us to match the pattern of the input expression with the one from Figure 3.3, where the hole appears to match.

Now that we have introduced how the relation rules work in Racket, we now present the implementation of some rules we considered most relevant from [14]. Nevertheless, all the reduction rules are in Annex I.

Variable Declaration. The inference rule presented below was taken from Di Pirro's operational semantic rules [14]. The rule Decl models variable declarations, where the variable x is added to the blockchain β, with value v. Additionally, this rule has a premise that states that the variable x cannot be in β.

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

The Racket implementation of this rule is below.

```
(--> [(in-hole E (T x = v e)) (B ...) env-s CT]
       [(in-hole E (v e)) (B ... (x -> v)) env-s CT]
       "DECL"    (side-condition (not (judgment-holds (xin (B ...) x)))))
```

```
1 (define-judgment-form FS
2   #:mode (xin I I)
3   #:contract (xin env-B any)
4   [------------------
5    (xin (B_1 ... (x -> v) B_2 ...) x)]
6 )
```

Figure 3.4: Judgement `xin`

We can divide this rule in three distinct parts, one for each line: what is on the left of the arrow in the inference rule; what is on the right side, the conclusion of the rule; and then the side-conditions.

The similarities are clear between the semantic rule and the one implemented, apart from the representation of the expression part, where instead of $T\ x = v$ we have it enclose a `in-hole` pattern, which as explained before allows it to decompose the term where the hole appears to match. The conclusion of the rule is also very similar, as we can see the variable declaration being appended to the blockchain environment, as well as its premise which denotes that x cannot be in the blockchain. This is enforced by the side-condition in the last line, which judgement returns true or false depending on if x is in β or not. This judgement is presented in Figure 3.4.

This judgement takes as input the environment $env - \beta$ and a variable x, and returns true if there is a pair $(x \mapsto v)$ in said environment.

Call Function. This next rule is a bit more complex, as it has more premises. The rule of function calling can be seen below:

$$\frac{\begin{array}{c}\hat{\beta}(c) = a \quad \beta^C(c) = C \quad \text{fbody}(C, f, \tilde{v}) = (\tilde{x}, e) \\ \beta' = \text{uptbal}(\text{uptbal}(\text{declcall}(\beta, c, (\widetilde{x : v})), a, n), \text{Top}(\sigma), -n) \\ e_s = e\{\text{this} := c, \text{msg.sender} := \text{Top}(\sigma), \text{msg.value} := n\}\end{array}}{\langle c.f.\text{value}(n)(\tilde{v}), \beta, \sigma\rangle \rightarrow \langle e_s, \beta', \sigma \cdot a\rangle}\ \textsc{Call}$$

This rule defines the call of function f, by returning its body e_s. The premises are that the contract c can only be called if and only if the contract exists in the blockchain and if the function f occurs in said contract.

In Figure 3.5, we present our revised Racket implementation of the Call rule. We remove the premise where x cannot be in β to allow for multiple calls of the same function. As explained prior, the instantiated variables are now within the contract scope, and if the variables $\tilde{x}$ are already in the blockchain, we modify them to its new value, and add the old value to the end of the blockchain.

Again, we can see the similarities in both the rule and its implementation. The premise that checks if the contract is in the blockchain is represented in line 2 in Figure 3.5, where the contract reference c has to match some contract declaration. The expression body e is returned by matching the contract name C and the function name f to the contract table CT. The contract in CT is denoted as:

```
1 (--> [(in-hole E (c -> f -> value (n_0) ((x : v) ...)))
2       (B_1 ... ((c a) -> (C (s : v_s)... n)) B_2 ...)
3       (B ... a_0 ... a_1)
4       CT]
5      [(in-hole E (return (subst-e c a_1 n_0 (subst-x ((x : v) ...) e))))
6       (uptbal (uptbal (declcall env-B c ((x : v) ...) a n_0) a_1 ,(- (term 0)(
            term n_0))))
7       (B ... a_0 ... a_1 a)
8       CT]
9      "CALL")
```

Figure 3.5: Racket implementation of rule CALL - simplified

$$(C \rightarrow (\texttt{contract}\ C\ \{(T_x\ x_x)...K\ F_0...(T_f\ f\ ((T\ x)...)\{\texttt{return}\ e\})\ F_1...\}))$$

The balance update is done in line 6, which is almost equal to its formal representation. The expression substitution and argument values of the function body is done with functions subst-e and subst-x respectively, and represented in line 5. The function `subst-e` substitutes all occurrences of `this`, `msg.sender` and `msg.value` with c, Top(σ) and n, respectively; whilst `subst-x` replaces all the occurrences of the variables x with their respective value v. The function `declcall` appends the instantiated variables to the contract's c scope. If the variables already exist in the scope, we modify their new values, and their old values are appended to the end of the blockchain.

3.1.2.4 Revised Operational Semantics

In this section, we present how we implement the operational semantic rules of Featherweight Solidity in Racket.

Parenthesis Removal. This rule is needed in cases where the evaluation of the first term of an expression of type `e -> value (e) ((x : e)...)` returns a function call and the parenthesis need to me removed so it matches the pattern we define in the syntax.

```
1    (--> [(in-hole E ((c -> f) -> value (n) ((s : vv) ...))) env-B env-s CT]
2         [(in-hole E (c -> f -> value (n) ((s : vv) ...))) env-B env-s CT]
3         "REM-PARENTHESIS")
```

Figure 3.6: Remove parenthesis

Function call 2. Both `CALL` and `CALL2` have the same behaviour, being that the only difference is the values of the input parameters. This is because of the value separation in terminal values and function instantiations, due to the pattern matching issue explained earlier. This problem was brought to our attention by the Applier example in Di Pirro's, as the input argument of a function is another function, and without this separation there

would be multiple branches, as Racket could evaluate the state variable lookups in the functions parameters or not.

```
1    (--> [(in-hole E (c -> f -> value (n_0) ((x : vv) ...)))
2          env-B
3          (B ... a_0 ... a_1)
4          CT]
5         [(in-hole E (return (subst-e c a_1 n_0 (subst-x ((x : vv) ...) e))))
             (uptbal (uptbal env-B a n_0) a_1 ,(- (term 0)(term n_0)))
6          (B ... a_0 ... a_1 a)
7          CT]
8         "CALL-2" (side-condition (not (judgment-holds (xin env-B (x ...))))))
```

Figure 3.7: Function CALL2 - simplified

3.1.3 Implementation Examples in Racket

In this section we present the execution of the examples presented in Di Pirro's thesis [14]. For presentation purposes and to keep our results as close to the original example, in the first three examples we ignore the blockchain changes that we presented of the instantiated variables, where we insert them in the contract scope. Additionally, we present the previously mentioned contract, BlockKing, that aims to prove the need of instantiated variables within each contract's scope to allow multiple calls to the same function, as well as showing that FS does not prevent some ill-behaved contracts.

3.1.3.1 Bank

Figure 3.8 shows a bank contract. It contains a state variable `balances` that represents the amount in each bank account. Function `deposit` increases the balance of the `msg.sender` account by the amount read form `msg.value`. Function `getBalance` returns the balance of the caller. In addition, functions `withdraw` and `transfer` allow the clients to transfer the money to their own account or to another client's account, respectively.

Figure 3.9 shows the FS' bank contract in our Racket implementation. We can see that the contracts in both figures are the same.

Now that we have presented the contract, we proceed to the program's execution which is presented in Figure 3.10. This is the same as the one presented in Di Pirro's [14], where we first initialise the contracts and deploy them to the blockchain, and then the client x_{eoa} deposits 500 and withdraws 100. The following execution steps are the same as the ones presented in FS's thesis, which we can compare and conclude that the execution is the same.

The following state presented in Figure 3.11, is after the deployment of the external client x_{eoa}. We can see that the client has been deployed to the blockchain. Because of our visualisation tool we are also able to tell the sequence of the operational rules applied to reach that state, which is: New-2; Decl and Seq.

```
1  contract Bank{
2      mapping(address => unit) balances;
3
4      Bank(mapping(address => unit) balances) {
5          this.balances = balances;
6      }
7      unit deposit() {
8          return this.balances = this.balances[msg.sender -> this.balances[msg.
               sender] + msg.value]; u
9      }
10     uint getBalance() {
11         return this.balances[msg.sender]
12     }
13     unit transfer(address to, uint amount) {
14         return
15             if this.balances[msg.sender] >= amount
16             then
17                 this.balances = this.balances[msg.sender -> this.balances[msg.
                       sender] - amount];
18                 this.balances = this.balances[to -> this.balances[to] + amount
                       ];
19                 u
20             else
21                 u
22     }
23     unit withdraw(uint amount) {
24         return
25             if this.balances[msg.sender] >= amount
26             then
27                 this.balances = this.balances[msg.sender -> this.balances[msg.
                       sender] - amount];
28                 msg.sender.transfer(amount);
29                 u
30             else
31                 u
32     }
33 }
```

Figure 3.8: Bank contract in Featherweight Solidity

The contract y_{Bank} is then deployed and we reach the state in Figure 3.12. This state is also reached by applying the same rules as before: New-2; Decl and Seq.

By calling the function `deposit`, its body is returned and we get to the state described in Figure 3.13. The sequence of rules applied were: Var; Var; Address; and CallToplevel. It is also important to note that Di Pirro's exercise in this state is different, as the expression presented here is different from the one in the original contract.

After the 500 have been deposited in x_{eoa} account, we can see in Figure 3.14 that the state variable `balances` has been updated and this information added. Again, this state can be reached by applying the following sequence: StateSel; StateSel; MappSel; Add; MappAss; StateAss; Seq; Return, and Seq. We do want to note that we think the `u` in Di Pirro's exercise was added by mistake, since it is not in the original expression.

After the expression has been evaluated and the function `withdraw` has been called,

```
(contract Bank {
    ((mapping (address => unit)) balances)

    (Bank (((mapping (address => unit)) balances)) {
        (this -> balances = balances)
    })
    (unit deposit () {
        return ((this -> balances = ((this -> balances)[(msg -> sender) -->
            (((this -> balances)[(msg -> sender)]) + (msg -> value))])) u)
    })
    (uint getBalance() {
        return ((this -> balances)[(msg -> sender)])
    })
    (unit transfer ((address to) (uint amount)) {
        return
            (if (((this -> balances)[(msg -> sender)]) >= amount)
            then
                (((this -> balances = ((this -> balances)[(msg -> sender) -->
                    (((this -> balances)[(msg -> sender)]) - amount)]))
                (this -> balances = ((this -> balances)[to --> (((this ->
                    balances)[to]) + amount)])))
                u)
            else
                u)
    })
    (unit withdraw ((uint amount)) {
        return
            (if (((this -> balances)[(msg -> sender)]) >= amount)
            then
                (((this -> balances = ((this -> balances)[(msg -> sender) -->
                    (((this -> balances)[(msg -> sender)]) - amount)]))
                ((msg -> sender) -> transfer (amount)))
                u)
            else u)
    })
})
```

Figure 3.9: Racket implementation of Bank contract

```
(EOC xEOA = (new EOC -> value (1000) ()
     (cEOA aEOA))
 (Bank yBank = (new Bank -> value (0)
     ((balances : ())) (cBank aBank))
 ((yBank -> deposit -> value (500) ->
     sender ((address (xEOA))) ())
 (yBank -> withdraw -> value (0) ->
     sender ((address (xEOA))) ((amount
      : 100))))))
()
()
```

$\langle \emptyset, \emptyset, EOC\ x_{eoa} = \mathsf{new}\ C.\mathsf{value}(1000)(); e' \rangle \longrightarrow^*$

Figure 3.10: Contract's initial state

```
1  (Bank yBank = (new Bank -> value (0) ((
       balances : ())) (cBank aBank))
2   ((yBank -> deposit -> value (500) ->
       sender ((address (xEOA))) ())
3   (yBank -> withdraw -> value (0) ->
       sender ((address (xEOA))) ((amount
        : 100)))))
4  (((cEOA aEOA) -> (EOC 1000))
5  (xEOA -> cEOA))
6  (((cEOA aEOA) -> (EOC 1000))
7  (xEOA -> cEOA))
```

$$\langle \emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, 1000)] \cdot [x_{eoa} \mapsto c_{eoa}],$$
$$\emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, 1000)] \cdot [x_{eoa} \mapsto c_{eoa}],$$
$$Bank\ y_{bank} = \mathsf{new}\ Bank(0_{\{\}}); e'' \rangle \longrightarrow^*$$

Figure 3.11: State after x_{eoa} deployment

```
1  ((yBank -> deposit -> value (500) ->
       sender ((address (xEOA))) ())
2    (yBank -> withdraw -> value (0) ->
        sender ((address (xEOA))) ((
        amount : 100))))
3  (((cEOA aEOA) -> (EOC 1000))
4   (xEOA -> cEOA)
5   ((cBank aBank) -> (Bank (balances : ()
       ) 0))
6   (yBank -> cBank))
7  (((cEOA aEOA) -> (EOC 1000))
8   (xEOA -> cEOA)
9   ((cBank aBank) -> (Bank (balances : ()
       ) 0))
10  (yBank -> cBank))
```

$$\langle \emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, 1000)] \cdot [x_{eoa} \mapsto c_{eoa}] \cdot [(c_{bank}, a_{bank}) \mapsto (Bank, 0_{\{\}}, 0)]$$
$$\cdot [y_{bank} \mapsto c_{bank}], \emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, 1000)] \cdot [x_{eoa} \mapsto c_{eoa}]$$
$$\cdot [(c_{bank}, a_{bank}) \mapsto (Bank, 0_{\{\}}, 0)] \cdot [y_{bank} \mapsto c_{bank}],$$
$$y_{bank}.deposit.\mathsf{value}(500).\mathsf{sender}(\mathsf{address}(x_{eoa}))(); e''' \rangle \longrightarrow^*$$

Figure 3.12: State after y_{Bank} deployment

```
1  ((return ((cBank -> balances = ((cBank
       -> balances) (aEOA --> (((cBank ->
       balances) (aEOA)) + 500)))) u))
2   (yBank -> withdraw -> value (0) ->
       sender ((address (xEOA))) ((amount
        : 100))))
3  (((cEOA aEOA) -> (EOC 500))
4   (xEOA -> cEOA)
5   ((cBank aBank) -> (Bank (balances : ()
       ) 500))
6   (yBank -> cBank))
7  (((cEOA aEOA) -> (EOC 1000))
8   (xEOA -> cEOA)
9   ((cBank aBank) -> (Bank (balances : ()
       ) 0))
10  (yBank -> cBank)
11  aBank)
```

$$\langle \emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, 1000)] \cdot [x_{eoa} \mapsto c_{eoa}] \cdot [(c_{bank}, a_{bank}) \mapsto$$
$$(Bank, 0_{\{\}}, 0)] \cdot [y_{bank} \mapsto c_{bank}], \emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, 1000)]$$
$$\cdot [x_{eoa} \mapsto c_{eoa}] \cdot [(c_{bank}, a_{bank}) \mapsto (Bank, 0_{\{\}}, 0)] \cdot [y_{bank} \mapsto c_{bank}] \cdot \underline{a_{bank}},$$
$$(\mathsf{return}\ c_{bank}.balances[a_{eoa}] = c_{bank}.balances[a_{eoa}] \rightarrow c_{bank}.balances[a_{eoa}] +$$
$$500]; \mathsf{u}); e''' \rangle \longrightarrow^*$$

Figure 3.13: Evaluation of function `deposit`

```
1  (yBank -> withdraw -> value (0) ->
       sender ((address (xEOA))) ((amount
       : 100)))
2  (((cEOA aEOA) -> (EOC 500))
3   (xEOA -> cEOA)
4   ((cBank aBank) -> (Bank (balances : ((
        aEOA :: 500))) 500))
5   (yBank -> cBank))
6  (((cEOA aEOA) -> (EOC 500))
7   (xEOA -> cEOA)
8   ((cBank aBank) -> (Bank (balances : ((
        aEOA :: 500))) 500))
9   (yBank -> cBank))
```

$\langle \emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, \underline{500})] \cdot [x_{eoa} \mapsto c_{eoa}] \cdot [(c_{bank}, a_{bank}) \mapsto \underline{(Bank, \{(a_{eoa}, 500)\}, 500)}] \cdot [y_{bank} \mapsto c_{bank}], \emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, \underline{500})] \cdot [x_{eoa} \mapsto c_{eoa}] \cdot [(c_{bank}, a_{bank}) \mapsto \underline{(Bank, \{(a_{eoa}, 500)\}, 500)}] \cdot [y_{bank} \mapsto c_{bank}], y_{bank}.withdraw.\mathsf{sender}(\mathsf{address}(x_{eoa}))(100); \mathsf{u}\rangle \longrightarrow^*$

Figure 3.14: Calling function `withdraw`

```
1  (return (if (((cBank -> balances) (aEOA
       )) >= 100)
2      then
3      (((cBank -> balances = ((cBank ->
           balances) (aEOA --> (((cBank ->
            balances) (aEOA)) - 100))))
4        (aEOA -> transfer (100)) u)
5      else u))
6  (((cEOA aEOA) -> (EOC 500))
7   (xEOA -> cEOA)
8   ((cBank aBank) -> (Bank (balances : ((
        aEOA :: 500))) 500))
9   (yBank -> cBank)
10  (amount -> 100))
11 (((cEOA aEOA) -> (EOC 500))
12  (xEOA -> cEOA)
13  ((cBank aBank) -> (Bank (balances : ((
        aEOA :: 500))) 500))
14  (yBank -> cBank)
15  aBank)
```

$\langle \beta \cdot \underline{[amount \mapsto 100]},$
$\sigma \cdot \underline{a_{bank}},$
$(\mathsf{return\ if}\ c_{bank}.balances[a_{eoa}] >= 100$
$\quad \mathsf{then}$
$\quad\quad c_{bank}.balances[a_{eoa}] = c_{bank}.balances[a_{eoa} \rightarrow c_{bank}.balances[a_{eoa}] - 100];$
$\quad\quad a_{eoa}.\mathsf{transfer}(100); \mathsf{u}$
$\quad \mathsf{else\ u}); \mathsf{u}\rangle \longrightarrow^*$

Figure 3.15: Function `withdraw`'s body

its body is returned so it can be evaluated. Figure 3.15 represents just that, and it was reached by the following rules: Var; Var; Address; and CallTopLevel.

Figure 3.15 represents the state of the program at the end of the evaluation of the if statement's first branch. We can gather that the call stack is yet to be changed to the new version of the blockchain since there is an address on top of the stack. Again, we got to this state by applying this sequence of rules: StateSel; MappSel; Greater-Eq; If-True; StateSel; StateSel; MappSel; Sub; MappAss; StateAss; and Seq.

The next and last state is the one presented in Figure 3.17. The only rule applied was Transfer.

```
1  (return ((aEOA -> transfer (100)) u))
2  (((cEOA aEOA) -> (EOC 500))
3   (xEOA -> cEOA)
4   ((cBank aBank) -> (Bank (balances : ((
        aEOA :: 400))) 500))
5   (yBank -> cBank)
6   (amount -> 100))
7  (((cEOA aEOA) -> (EOC 500))
8   (xEOA -> cEOA)
9   ((cBank aBank) -> (Bank (balances : ((
        aEOA :: 500))) 500))
10  (yBank -> cBank)
11  aBank)
```

$$\langle \emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, 500)] \cdot [x_{eoa} \mapsto c_{eoa}] \cdot [(c_{bank}, a_{bank}) \mapsto (Bank, \underline{\{(a_{eoa}, 400)\}}, 500)] \cdot [y_{bank} \mapsto c_{bank}] \cdot [amount \mapsto 100], \sigma \cdot a_{bank}, (\mathsf{return}\ a_{eoa}.\mathsf{transfer}(100); \mathsf{u}); \mathsf{u}\rangle \longrightarrow^*$$

Figure 3.16: Calling `transfer`

```
1  (return ((return u) u))
2  (((cEOA aEOA) -> (EOC 600))
3   (xEOA -> cEOA)
4   ((cBank aBank) -> (Bank (balances : ((
        aEOA :: 400))) 400))
5   (yBank -> cBank)
6   (amount -> 100))
7  (((cEOA aEOA) -> (EOC 500))
8   (xEOA -> cEOA)
9   ((cBank aBank) -> (Bank (balances : ((
        aEOA :: 500))) 500))
10  (yBank -> cBank)
11  aBank
12  aEOA)
```

$$\langle \emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, \underline{600})] \cdot [x_{eoa} \mapsto c_{eoa}] \cdot [(c_{bank}, a_{bank}) \mapsto (Bank, \{(a_{eoa}, 400)\}, \underline{400})] \cdot [y_{bank} \mapsto c_{bank}] \cdot [amount \mapsto 100], \sigma \cdot a_{bank} \cdot \underline{a_{eoa}}, (\mathsf{return}\ ((\mathsf{return}\ \mathsf{u}); \mathsf{u}))\rangle \longrightarrow^*$$

Figure 3.17: Final state of Di Pirro's evaluation

3.1.3.2 Blood Bank

The contract shown in Figure 3.18 models a blood bank. This contract has a map of all donors and their health state. The only actor that can modify these variables is the `doctor` in function `setHealth`, which is verified by an if statement. Only verified donors in a healthy state and enough blood can `donate`.

The `Donor` contract represents the blood donors which are defined by the amount of blood and the address of their blood bank.

In Figure 3.19, we present our implementation of the BloodBank contract in Racket. We can see that both contracts are really similar apart from the few changes to the FS' syntax.

In the following figures we will present the side by side visualisation of the execution example presented by Di Pirro. In Figure 3.20 we can see that the program deploys three contracts, a doctor x_{doctor}; a blood bank y_{Bank} and a blood donor z_{Donor}, followed by the doctor setting the health of z_{Donor}. Our blockchain and call stack are empty in the beginning of the execution (lines 5 and 6).

The state presented in Figure 3.21 represents the state where the doctor's contract has

```
contract BloodBank{
    mapping(address => bool) healty;
    address doctor;
    uint blood;
    BloodBank(mapping(address => bool) healty, address doctor, uint blood) {
        this.healty = healty;
        this.doctor = doctor;
        this.blood = blood;
    }
    unit setHealth(address donor, bool isHealty) {
        return
            if msg.sender == this.doctor
            then this.healty = this.healty[donor -> isHealty]; u
            else revert
    }
    bool isHealty(address donor) {
        return
            if msg.sender == this.doctor
            then this.healty[donor]
            else revert
    }
    unit donate(uint amount) {
        return
            uint donorBlood = Donor(msg.sender).getBlood();
            if this.healty[msg.sender]&&donorBlood > 3000&&donorBlood - amount
                > 0
            then this.blood = this.blood + amount; true
            else false
    }
    address getDoctor() {
        return this.doctor
    }
    uint getBlood() {
        return this.blood
} }
contract Donor {
    uint blood;
    address bank;
    Donor(uint blood, address bank) {
        this.blood = blood;
        this.bank = bank;
    }
    unit donate(uint amount){
        return
            if BloodBank(this.bank).donate( amount)
            then this.blood=this.blood - amount;u
            else u
    }
    BloodBank getBank() {
        return this.bank
    }
    uint getBlood() {
        return this.blood
} }
```

Figure 3.18: Featherweight Solidity Contract of Blood Bank

```
(contract BloodBank {
    ((mapping (address => bool)) healthy)
    (address doctor)
    (uint blood)
    (BloodBank (((mapping (address => bool)) healthy) (address doctor) (uint
        blood)) {
        (this -> healthy = healthy)
        (this -> doctor = doctor)
        (this -> blood = blood)
    })
    (unit setHealth ((address donor) (bool isHealthy)) {
        return
            (if ((msg -> sender) == (this -> doctor))
            then ((this -> healthy = ((this -> healthy) [donor -> isHealthy]))
                u)
            else revert)
    })
    (bool isHealty ((address donor)) {
        return
            (if ((msg -> sender) == (this -> doctor))
            then ((this -> healthy) [donor])
            else revert)
    })
    (bool donate ((uint amount)) {
        return
            ((uint donorBlood = ((msg -> sender) -> getBlood -> value (0) ()))
            (if (((this -> healthy)[(msg -> sender)]) && ((donorBlood > 3000)
                && ((donorBlood - amount) > 0)))
            then ((this -> blood = ((this -> blood) + amount)) true)
            else false))
    })
    (address getDoctor () {
        return (this -> doctor)
    })
    (uint getBlood () {
        return (this -> blood)
    })
})
(contract Donor {
    (uint blood)
    (address bank)
    (Donor ((uint blood) (address bank)) {
        (this -> blood = blood)
        (this -> bank = bank)
    })
    (unit donate ((uint amount)) {
        return
            (if ((this -> bank) -> donate -> value (0) ((amount : amount)))
            then ((this -> blood = ((this -> blood) - amount)) u)
            else u)
    })
    (BloodBank getBank () {
        return (this -> bank)
    })
    (uint getBlood () {
        return (this -> blood)
    })})
```

Figure 3.19: BloodBank contract in Racket

```
1 (EOC xDoctor = (new EOC -> value (0) ()
      (cDoctor aDoctor))
2  (BloodBank yBank = (new BloodBank ->
      value (0) ((healthy : ()) (doctor
      : (address (xDoctor))) (blood : 0)
      ) (cBank aBank))
3  (Donor zDonor = (new Donor -> value
      (0) ((blood : 5000) (bank : (
      address (yBank)))) (cDonor aDonor)
      )
4  (yBank -> setHealth -> value (0) ->
      sender ((address (xDoctor))) ((
      donor : (address (zDonor))) (
      isHealthy : true))))))
5 ()
6 ()
```

$\langle \emptyset, \emptyset, EOC\ x_{doctor} = \mathsf{new}\ EOC(); e' \rangle \longrightarrow$

Figure 3.20: Initial state

```
1 (EOC xDoctor = cDoctor
2  (BloodBank yBank = (new BloodBank ->
      value (0) ((healthy : ()) (doctor
      : (address (xDoctor))) (blood : 0)
      ) (cBank aBank))
3  (Donor zDonor = (new Donor -> value
      (0) ((blood : 5000) (bank : (
      address (yBank)))) (cDonor aDonor)
      )
4  (yBank -> setHealth -> value (0) ->
      sender ((address (xDoctor))) ((
      donor : (address (zDonor))) (
      isHealthy : true))))))
5 (((cDoctor aDoctor) -> (EOC 0)))
6 ()
```

$\langle \emptyset \cdot [(c_{doctor}, a_{doctor}) \mapsto (EOC, \epsilon, 0)], \emptyset, EOC\ x_{doctor} = c_{doctor}; e' \rangle \longrightarrow$

Figure 3.21: Defining x_{Doctor}

been deployed to the blockchain, and we are now defining the variable x_{Doctor}. This state was reached by applying the New-2 rule.

After applying the Decl rule, we reach the state in Figure 3.22. We can see that the variable x_{Doctor} is now defined in the blockchain.

The next state is presented in Figure 3.23, which is after applying the Seq rule. We can see that both the blockchain and the call stack are the same, which means the operation was successful.

Figure 3.24 represents the state after the deployment of the `Bloodbank` and `Donor` contracts. Both contracts are now in the blockchain and call stack, and we are ready to evaluate the function calling in line 1.

The state presented in Figure 3.25 is the state after the parameters have been reduced to values and we can apply the CallTopLevel function. The rule evaluation sequence is: Var, Var; Address; Var; Address.

After the CallTopLevel, we reach the state in Figure 3.26, we can see that the

```
1 (cDoctor
2  (BloodBank yBank = (new BloodBank ->
       value (0) ((healthy : ()) (doctor
       : (address (xDoctor))) (blood : 0)
       ) (cBank aBank))
3  (Donor zDonor = (new Donor -> value
       (0) ((blood : 5000) (bank : (
       address (yBank)))) (cDonor aDonor)
       )
4  (yBank -> setHealth -> value (0) ->
       sender ((address (xDoctor))) ((
       donor : (address (zDonor))) (
       isHealthy : true))))))
5 (((cDoctor aDoctor) -> (EOC 0))
6  (xDoctor -> cDoctor))
7 ()
```

$$\langle \emptyset \cdot [(c_{doctor}, a_{doctor}) \mapsto (EOC, \epsilon, 0)] \cdot [x_{doctor} \mapsto c_{doctor}], \emptyset, c_{doctor}; e' \rangle \longrightarrow^{\text{SEQ-C}}$$

Figure 3.22: Doctor variables

```
1 (BloodBank yBank = (new BloodBank ->
      value (0) ((healthy : ()) (doctor :
       (address (xDoctor))) (blood : 0))
      (cBank aBank))
2  (Donor zDonor = (new Donor -> value
       (0) ((blood : 5000) (bank : (
       address (yBank)))) (cDonor aDonor)
       )
3  (yBank -> setHealth -> value (0) ->
       sender ((address (xDoctor))) ((
       donor : (address (zDonor))) (
       isHealthy : true)))))
4 (((cDoctor aDoctor) -> (EOC 0))
5  (xDoctor -> cDoctor))
6 (((cDoctor aDoctor) -> (EOC 0))
7  (xDoctor -> cDoctor))
```

$$\begin{aligned}\langle \emptyset \cdot [(c_{doctor}, a_{doctor}) \mapsto (EOC, \epsilon, 0)] \cdot [x_{doctor} \mapsto c_{doctor}],\\ \emptyset \cdot [(c_{doctor}, a_{doctor}) \mapsto (EOC, \epsilon, 0)] \cdot [x_{doctor} \mapsto c_{doctor}],\\ BloodBank\ y_{bank} = \text{new}\ BloodBank(\{\}, \text{address}(x_{doctor}), 0); e'' \rangle \longrightarrow^*\end{aligned}$$

Figure 3.23: BloodBank deployment

`setHealth`'s body has been returned and the address of the caller a_{Bank} is on top of the call stack, and the values of the function arguments appended to the contract.

By applying the sequence: StateSel; Equals; If-True, we reach the state in Figure 3.27.

After returning the value of the map `healthy`, by rule StateSel, we get to state in Figure 3.28. Since the map is empty, it returns ().

The MappAss rule gets us to the state shown in the Figure3.29, where the key-value (`aDonor, true`) has been added to the map variable.

The last state of our execution is reached in Figure 3.30. The `healthy` variable has been updated, and the call stack is different from the blockchain as the statement has not been consumed through the Seq rule.

```
1  (yBank -> setHealth -> value (0) ->
       sender ((address (xDoctor))) ((
       donor : (address (zDonor))) (
       isHealthy : true)))
2  (((cDoctor aDoctor) -> (EOC 0))
3   (xDoctor -> cDoctor)
4   ((cBank aBank) -> (BloodBank (healthy
        : ()) (doctor : aDoctor) (blood :
        0) 0))
5   (yBank -> cBank)
6   ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
7   (zDonor -> cDonor))
8  (((cDoctor aDoctor) -> (EOC 0))
9   (xDoctor -> cDoctor)
10  ((cBank aBank) -> (BloodBank (healthy
        : ()) (doctor : aDoctor) (blood :
        0) 0))
11  (yBank -> cBank)
12  ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
13  (zDonor -> cDonor))
```

$$\langle \beta_1, \sigma_1, y_{bank}.setHealth.\mathsf{sender}(\mathsf{address}(x_{doctor}))(\mathsf{address}(z_{donor}), true)\rangle \longrightarrow^*$$

Figure 3.24: Calling setHealth

```
1  (cBank -> setHealth -> value (0) ->
       sender (aDoctor) ((donor : aDonor)
       (isHealthy : true)))
2  (((cDoctor aDoctor) -> (EOC 0))
3   (xDoctor -> cDoctor)
4   ((cBank aBank) -> (BloodBank (healthy
        : ()) (doctor : aDoctor) (blood :
        0) 0))
5   (yBank -> cBank)
6   ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
7   (zDonor -> cDonor))
8  (((cDoctor aDoctor) -> (EOC 0))
9   (xDoctor -> cDoctor)
10  ((cBank aBank) -> (BloodBank (healthy
        : ()) (doctor : aDoctor) (blood :
        0) 0))
11  (yBank -> cBank)
12  ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
13  (zDonor -> cDonor))
```

$$\langle \beta_1, \sigma_1, c_{bank}.setHealth.\mathsf{sender}(a_{doctor})(a_{donor}, true)\rangle \longrightarrow^{\textsc{Call}}$$

Figure 3.25: Applying CallTopLevel rule

```
1  (return (if (aDoctor == (cBank ->
       doctor)) then ((cBank -> healthy =
       ((cBank -> healthy) (aDonor -->
       true))) u) else revert))
2  (((cDoctor aDoctor) -> (EOC 0))
3   (xDoctor -> cDoctor)
4   ((cBank aBank) -> (BloodBank (healthy
        : ()) (doctor : aDoctor) (blood :
        0) 0))
5   (yBank -> cBank)
6   ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
7   (zDonor -> cDonor)
8   (donor -> aDonor)
9   (isHealthy -> true))
10 (((cDoctor aDoctor) -> (EOC 0))
11  (xDoctor -> cDoctor)
12  ((cBank aBank) -> (BloodBank (healthy
        : ()) (doctor : aDoctor) (blood :
        0) 0))
13  (yBank -> cBank)
14  ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
15  (zDonor -> cDonor)
16  aBank)
```

$$\langle \beta_1 \cdot [\_donor \mapsto a_{donor}] \cdot [\_isHealty \mapsto \mathsf{true}], \sigma_1 \cdot a_{bank},$$
$$\mathsf{return\ if}\ a_{doctor} == c_{bank}.doctor\ \mathsf{then}$$
$$c_{bank}.healty = c_{bank}.healty[\_donor \to \_isHealty]; \mathsf{u}$$
$$\mathsf{else\ revert}\rangle \longrightarrow^*$$

Figure 3.26: `setHeatlh` body

```
1  (return ((cBank -> healthy = ((cBank ->
       healthy) (aDonor --> true))) u))
2  (((cDoctor aDoctor) -> (EOC 0))
3   (xDoctor -> cDoctor)
4   ((cBank aBank) -> (BloodBank (healthy
        : ()) (doctor : aDoctor) (blood :
        0) 0))
5   (yBank -> cBank)
6   ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
7   (zDonor -> cDonor)
8   (donor -> aDonor)
9   (isHealthy -> true))
10 (((cDoctor aDoctor) -> (EOC 0))
11  (xDoctor -> cDoctor)
12  ((cBank aBank) -> (BloodBank (healthy
        : ()) (doctor : aDoctor) (blood :
        0) 0))
13  (yBank -> cBank)
14  ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
15  (zDonor -> cDonor)
16  aBank)
```

$$\langle \underline{\beta_1 \cdot [\_donor \mapsto a_{donor}] \cdot [\_isHealty \mapsto \mathsf{true}]}, \underline{\sigma_1 \cdot a_{bank}},$$
$$\mathsf{return}\ c_{bank}.healty = c_{bank}.healty[\_donor \to \_isHealty]; \mathsf{u}\rangle \longrightarrow$$

Figure 3.27: Modifying `healthy` variable

```
1  (return ((cBank -> healthy = (() (
       aDonor --> true))) u))
2  (((cDoctor aDoctor) -> (EOC 0))
3   (xDoctor -> cDoctor)
4   ((cBank aBank) -> (BloodBank (healthy
        : ()) (doctor : aDoctor) (blood :
        0) 0))
5   (yBank -> cBank)
6   ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
7   (zDonor -> cDonor)
8   (donor -> aDonor)
9   (isHealthy -> true))
10 (((cDoctor aDoctor) -> (EOC 0))
11  (xDoctor -> cDoctor)
12  ((cBank aBank) -> (BloodBank (healthy
        : ()) (doctor : aDoctor) (blood :
        0) 0))
13  (yBank -> cBank)
14  ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
15  (zDonor -> cDonor)
16  aBank)
```

$$\langle \beta_1 \cdot [\_donor \mapsto a_{donor}] \cdot [\_isHealty \mapsto \mathsf{true}], \sigma_1 \cdot a_{bank}, \\ \mathsf{return}\, c_{bank}.healty = 0_{\{\}}[a_{donor} \to \mathsf{true}]; \mathsf{u} \rangle \longrightarrow$$

Figure 3.28: Returning healthy

```
1  (return ((cBank -> healthy = ((aDonor
       :: true))) u))
2  (((cDoctor aDoctor) -> (EOC 0))
3   (xDoctor -> cDoctor)
4   ((cBank aBank) -> (BloodBank (healthy
        : ()) (doctor : aDoctor) (blood :
        0) 0))
5   (yBank -> cBank)
6   ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
7   (zDonor -> cDonor)
8   (donor -> aDonor)
9   (isHealthy -> true))
10 (((cDoctor aDoctor) -> (EOC 0))
11  (xDoctor -> cDoctor)
12  ((cBank aBank) -> (BloodBank (healthy
        : ()) (doctor : aDoctor) (blood :
        0) 0))
13  (yBank -> cBank)
14  ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
15  (zDonor -> cDonor)
16  aBank)
```

$$\langle \beta_1 \cdot [\_donor \mapsto a_{donor}] \cdot [\_isHealty \mapsto \mathsf{true}], \sigma_1 \cdot a_{bank}, \\ \mathsf{return}\, c_{bank}.healty = \{(a_{donor}, \mathsf{true})\}; \mathsf{u} \rangle \longrightarrow$$

Figure 3.29: Adding to healthy

```
1  (return u)
2  (((cDoctor aDoctor) -> (EOC 0))
3   (xDoctor -> cDoctor)
4   ((cBank aBank) -> (BloodBank (healthy
        : ((aDonor :: true))) (doctor :
        aDoctor) (blood : 0) 0))
5   (yBank -> cBank)
6   ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
7   (zDonor -> cDonor)
8   (donor -> aDonor)
9   (isHealthy -> true))
10 (((cDoctor aDoctor) -> (EOC 0))
11  (xDoctor -> cDoctor)
12  ((cBank aBank) -> (BloodBank (healthy
        : ()) (doctor : aDoctor) (blood :
        0) 0))
13  (yBank -> cBank)
14  ((cDonor aDonor) -> (Donor (blood :
        5000) (bank : aBank) 0))
15  (zDonor -> cDonor)
16  aBank)
```

$$\langle \emptyset \cdot [(c_{doctor}, a_{doctor}) \mapsto (EOC, \epsilon, 0)] \cdot [x_{doctor} \mapsto c_{doctor}]$$
$$\cdot \underline{[(c_{bank}, a_{bank}) \mapsto (BloodBank, healty = \{(a_{donor}, \mathsf{true})\};}$$
$$\underline{doctor = a_{doctor}; blood = 0, 0]}$$
$$\cdot [y_{bank} \mapsto c_{bank}] \cdot [(c_{donor}, a_{donor}) \mapsto (Donor, blood : 5000; bank : a_{bank}, 0)]$$
$$\cdot [z_{donor} \mapsto c_{donor}] \cdot [\_donor \mapsto a_{donor}] \cdot [\_isHealty \mapsto \mathsf{true}], \sigma_1 \cdot a_{bank},$$
$$\mathsf{return\ u}\rangle \longrightarrow$$

Figure 3.30: Final state

3.1.3.3 Applier

This example models a contract Applier, that has one state variable `state` and one single method `apply`, that takes as input a function, and then applies it to the variable `state`. The contract Test is there to use the contract Applier. This is an interesting example that depicts functions as first class values.

Figure 3.31 is the Featherweight Solidity version of these two contracts, whereas Figure 3.32 is its implementation in Racket.

The initial state of the program's execution is represented in Figure 3.33. It begins with the deployment of all three contracts until it reaches the state in Figure 3.34, by applying the following sequence: New-2; Decl; Seq-C; New-2; Decl; Seq-C; Var; New-2; Decl; Seq-C.

As depicted in Figure 3.35, the body of the function `f1` has been returned and the caller `aTest` added to the top of the call stack. The state is reached by applying this sequence of rules: Var; Var; Address; and CallTopLevel.

After calling the function `apply`, we reach the state in Figure 3.36 by rules StateSel and Call.

The next step in our evaluation example is the return of function `square`, depicted in Figure3.37 and then the return of the calculated value, in Figure 3.38. The sequence of rules is the following: Rem-Par; Call; StateSel; StateSel; Mult; Return; Return; and Return.

```
contract Applier {
    uint state;
    Applier(uint state) {
        this.state = state;
    }
    unit apply(uint -> uint f) {
        return f (this.state)
    } }
contract Test {
    Applier app;
    Test(Applier app) {
        this.app = app
    }
    unit f1() {
        return this.app.apply(this.square)
    }
    unit f2() {
        return this.app.apply(this.double)
    }
    unit square(uint n) {
        return n*n
    }
    unit double(uint n) {
        return n+n
} }
```

Figure 3.31: Applier and Test contracts in Featherweight Solidity

```
(contract Applier {
    (uint state)
    (Applier ((uint state)) {
        (this -> state = state)
    })
    (unit apply (((uint -> uint) f)) {
        return (f -> value (0) ((n : (this -> state))))
    })  })
(contract Test {
    (Applier app)
    (Test ((Applier app)) {
        (this -> app = app)
    })
    (unit f1 () {
        return ((this -> app) -> apply -> value (0) ((f : (this -> square))))
    })
    (unit f2 () {
        return ((this -> app) -> apply -> value (0) ((f : (this -> double))))
    })
    (unit square ((uint n)) {
        return (n * n)
    })
    (unit double ((uint n)) {
        return (n + n)
    })  })
```

Figure 3.32: Racket implementation of the Applier and Test contracts

```
1  (EOC xEOA = (new EOC -> value (0) () (
       cEOA aEOA))
2   (Applier yApp = (new Applier -> value
       (0) ((state : 10)) (cApp aApp))
3   (Test zTest = (new Test -> value (0)
       ((app : yApp)) (cTest aTest))
4   (zTest -> f1 -> value (0) -> sender ((
       address (xEOA))) ())))))
5   ()
6   ()
```

$$e := \quad EOC\ x_{eoa} = \mathsf{new}\ EOC();\ Applier\ y_{app} = \mathsf{new}\ Applier(10);\ Test\ z_{test} = \mathsf{new}\ Test(y_{app});\ z_{test}.f1.\mathsf{sender}(\mathsf{address}(x_{eoa}))()$$

Figure 3.33: Contract's initial state

```
1  (zTest -> f1 -> value (0) -> sender ((
       address (xEOA))) ())
2  (((cEOA aEOA) -> (EOC 0))
3   (xEOA -> cEOA)
4   ((cApp aApp) -> (Applier (state : 10)
       0))
5   (yApp -> cApp)
6   ((cTest aTest) -> (Test (app : cApp)
       0))
7   (zTest -> cTest))
8  (((cEOA aEOA) -> (EOC 0))
9   (xEOA -> cEOA)
10  ((cApp aApp) -> (Applier (state : 10)
       0))
11  (yApp -> cApp)
12  ((cTest aTest) -> (Test (app : cApp)
       0))
13  (zTest -> cTest))
```

$$\langle \emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, 0)] \cdot [x_{eoa} \mapsto c_{eoa}] \cdot [(c_{app}, a_{app}) \mapsto (Applier, state = 10, 0)] \cdot [y_{app} \mapsto c_{app}] \cdot [(c_{test}, a_{test}) \mapsto (Test, app = c_{app}, 0)] \cdot [x_{test} \mapsto c_{test}], \emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, 0)] \cdot [x_{eoa} \mapsto c_{eoa}] \cdot [(c_{app}, a_{app}) \mapsto (Applier, state = 10, 0)] \cdot [y_{app} \mapsto c_{app}] \cdot [(c_{test}, a_{test}) \mapsto (Test, app = c_{app}, 0)] \cdot [x_{test} \mapsto c_{test}],\ z_{test}.f1.\mathsf{sender}(\mathsf{address}(x_{eoa}))(); \mathsf{u}\rangle \longrightarrow^*$$

Figure 3.34: Calling `f1`

```
1  (return (cApp -> apply -> value (0) ((f
       : (cTest -> square)))))
2  (((cEOA aEOA) -> (EOC 0))
3   (xEOA -> cEOA)
4   ((cApp aApp) -> (Applier (state : 10)
       0))
5   (yApp -> cApp)
6   ((cTest aTest) -> (Test (app : cApp)
       0))
7   (zTest -> cTest))
8  (((cEOA aEOA) -> (EOC 0))
9   (xEOA -> cEOA)
10  ((cApp aApp) -> (Applier (state : 10)
       0))
11  (yApp -> cApp)
12  ((cTest aTest) -> (Test (app : cApp)
       0))
13  (zTest -> cTest)
14  aTest)
```

$$\langle \beta, \sigma \cdot \underline{a_{test}}, (\mathsf{return}\ c_{test}.app.apply(c_{test}.square)); \mathsf{u}\rangle \longrightarrow^*$$

Figure 3.35: Calling `apply`

```
1  (return (return (cTest -> square ->
       value (0) ((n : (cApp -> state)))))
       )
2  (((cEOA aEOA) -> (EOC 0))
3   (xEOA -> cEOA)
4   ((cApp aApp) -> (Applier (state : 10)
       0))
5   (yApp -> cApp)
6   ((cTest aTest) -> (Test (app : cApp)
       0))
7   (zTest -> cTest)
8   (f -> (cTest -> square)))
9  (((cEOA aEOA) -> (EOC 0))
10  (xEOA -> cEOA)
11  ((cApp aApp) -> (Applier (state : 10)
       0))
12  (yApp -> cApp)
13  ((cTest aTest) -> (Test (app : cApp)
       0))
14  (zTest -> cTest)
15  aTest
16  aApp)
```

$$\langle \beta \cdot \underline{[f \mapsto c_{test}.square]}, \sigma \cdot a_{test} \cdot \underline{a_{app}}, \\ (\mathsf{return}\ (\mathsf{return}\ c_{test}.square(c_{app}.state)))\mathsf{;u}\rangle \longrightarrow^*$$

Figure 3.36: Calling square

```
1  (return (return (return (10 * 10))))
2  (((cEOA aEOA) -> (EOC 0))
3   (xEOA -> cEOA)
4   ((cApp aApp) -> (Applier (state : 10)
       0))
5   (yApp -> cApp)
6   ((cTest aTest) -> (Test (app : cApp)
       0))
7   (zTest -> cTest)
8   (f -> (cTest -> square))
9   (n -> 10))
10 (((cEOA aEOA) -> (EOC 0))
11  (xEOA -> cEOA)
12  ((cApp aApp) -> (Applier (state : 10)
       0))
13  (yApp -> cApp)
14  ((cTest aTest) -> (Test (app : cApp)
       0))
15  (zTest -> cTest)
16  aTest
17  aApp
18  aTest)
```

$$\langle \beta \cdot [f \mapsto c_{test}.square] \cdot \underline{[n \mapsto 10]}, \sigma \cdot a_{test} \cdot a_{app} \cdot \underline{a_{test}}, \\ (\mathsf{return}\ (\mathsf{return}\ (\mathsf{return}\ n\ *\ n)))\rangle \longrightarrow^*$$

Figure 3.37: Evaluating square

```
100
(((cEOA aEOA) -> (EOC 0))
 (xEOA -> cEOA)
 ((cApp aApp) -> (Applier (state : 10)
     0))
 (yApp -> cApp)
 ((cTest aTest) -> (Test (app : cApp)
     0))
 (zTest -> cTest)
 (f -> (cTest -> square))
 (n -> 10))
(((cEOA aEOA) -> (EOC 0))
 (xEOA -> cEOA)
 ((cApp aApp) -> (Applier (state : 10)
     0))
 (yApp -> cApp)
 ((cTest aTest) -> (Test (app : cApp)
     0))
 (zTest -> cTest))
```

$$\langle \beta \cdot [f \mapsto c_{test}.square] \cdot [n \mapsto 10], \sigma, 100 \rangle$$

Figure 3.38: Final state

3.1.3.4 BlockKing

In addition to the examples presented in Di Pirro's thesis, we present BlockKing contract, previously mention in Chapter 1. As we recall, this contract allowed for many clients to enter the gamble, which is a problem as it would only store the information of one (the last) user, allowing other users to steal the prizes of others as they were waiting on their result.

As FS' specification does not support concurrency, in Figure 3.39, we present a simplified version of the BlockKing code in Figure 1.1. As the present version of the language specification does not support concurrency, we have disregarded the `__callback` function, and we trigger the response of the Oraclize service by simply calling the `process_payment` function to mimic concurrency. We have also left out the if statements and some variables to have less evaluation steps. However, we guarantee the contract's behaviour, and we still maintain the key elements, such as the variables `king` and `warrior`.

Figure 3.40 introduces the initial state of our BlockKing program. Through the demonstration of the execution of the contract, we are going to omit the call stack as it stores a copy of the blockchain at the beginning of a transaction and the addresses that call functions during each transaction, which is not relevant to the presentation. Because of the changes made to the contracts representation in the blockchain to allow for multiple calls of the same function, they are now represented as mappings of pairs (c, a), contract reference and address, to a triple $(C, \widetilde{s : v}, n)$: contract name, a sequence of state variables and respective values, and its balance, and a set of instantiated variables $(x \mapsto v)$.

In Figure 3.41, we present the state of the blockchain after the deployment of all three contracts, aBK being the BlockKing and both clients aBx and aBy; and the remaining expressions. Both BlockKing's `warrior` and `king` are set with the contract's address.

```
((contract BlockKing(
    (address warrior)
    (uint warriorGold)
    (uint myid)
    (address king)
    (BlockKing ((address warrior) (uint warriorGold)
(uint myid) (address king))
     ((this -> warrior = warrior)
      (this -> warriorGold = warriorGold)
      (this -> myid = myid)
      (this -> king = king)))
    (uint enter () (
     return
      ((this -> warrior = (msg -> sender))
       (this -> warriorGold = (msg -> value)))))
    (uint process_payment () (
     return
       (this -> king = (this -> warrior))))))
 (contract EOC (
    (EOC () ())
    (unit fb () (
     return u))))))
```

Figure 3.39: BlockKing code in Featherweight Solidity

```
((BlockKing bk = (new BlockKing -> value (0) ((warrior : aBK) (warriorGold :
    8) (myid : 9) (king : aBK)) (cBK aBK)))
 ((EOC x = (new EOC -> value (6) () (cBx aBx)))
  ((EOC y = (new EOC -> value (4) () (cBy aBy)))
   ((bk -> enter -> value (2) -> sender (aBx) ())
    ((bk -> enter -> value (3) -> sender (aBy) ())
     ((bk -> process_payment -> value (0) -> sender (aBK) ())
      ((bk -> process_payment -> value (0) -> sender (aBK) ()) ()))))))))
(())
(())
```

Figure 3.40: BlockKing initial state

```
((bk -> enter -> value (2) -> sender (aBx) ())
 ((bk -> enter -> value (3) -> sender (aBy) ())
  ((bk -> process_payment -> value (0) -> sender (aBK) ())
   ((bk -> process_payment -> value (0) -> sender (aBK) ()) ())))))
((((cBK aBK) -> (BlockKing
    (warrior : aBK)
    (warriorGold : 8)
    (myid : 9)
    (king : aBK)
    0) ->)
  (bk -> cBK)
  ((cBx aBx) -> (EOC 6) ->)
  (x -> cBx)
  ((cBy aBy) -> (EOC 4) ->)
  (y -> cBy)))
```

Figure 3.41: Contracts deployment

```
((bk -> enter -> value (3) -> sender (aBy) ())
 ((bk -> process_payment -> value (0) -> sender (aBK) ())
  ((bk -> process_payment -> value (0) -> sender (aBK) ()) ())))
(((((cBK aBK) -> (BlockKing (warrior : aBx) (warriorGold : 2) (myid : 9) (king
     : aBK) 2) ->)
  (bk -> cBK)
  ((cBx aBx) -> (EOC 4) ->)
  (x -> cBx)
  ((cBy aBy) -> (EOC 4) ->)
  (y -> cBy)))
```

Figure 3.42: BlockKing after aBx enters

```
((bk -> process_payment -> value (0) -> sender (aBK) ())
 ((bk -> process_payment -> value (0) -> sender (aBK) ()) ()))
(((((cBK aBK) -> (BlockKing (warrior : aBy) (warriorGold : 3) (myid : 9) (king
     : aBK) 5) ->)
  (bk -> cBK)
  ((cBx aBx) -> (EOC 4) ->)
  (x -> cBx)
  ((cBy aBy) -> (EOC 1) ->)
  (y -> cBy)))
```

Figure 3.43: BlockKing after aBy enters

```
((bk -> process_payment -> value (0) -> sender (aBK) ()) ())
(((((cBK aBK) -> (BlockKing (warrior : aBy) (warriorGold : 3) (myid : 9) (king
     : aBy) 5) ->)
  (bk -> cBK)
  ((cBx aBx) -> (EOC 4) ->)
  (x -> cBx)
  ((cBy aBy) -> (EOC 1) ->)
  (y -> cBy)))
```

Figure 3.44: BlockKing appoints aBy king

The state presented in Figure 3.42 is after the client aBx has entered the gamble. aBx is the new warrior and is now waiting for the Oraclize service to respond. But, as we can see, the next expression to be evaluated is the entrance of client aBy, which simulates the concurrency we have previously discussed.

The evaluation of the statement in which the client aBy enters the gamble takes us to the state shown in Figure 3.43. As we can see, aBy is the new `warrior`, which means that whenever the Oraclize server returns the random number for the aBx call, the chance to win the prize will not be for aBx, but for aBy, as it is the current `warrior`.

Figure 3.44 represents the state after the first random number has been sent by the Oraclize service. As we can clearly see, the new `king` is the current `warrior` and not aBx.

This behaviour can not be prevented with the current FS specification, as it does not guarantee safety when in a concurrent setting. We believe that other smart contract

languages, which are equipped with mechanisms such as typestates, can prevent some ill-behaved programs, in a world where these types of errors can cost millions in losses [6].

3.2 Type System

Our OCaml implementation, besides checking that a program is well-typed, it also produces the type derivation trees of the execution, for illustrative purposes. This tree is useful by at least two reasons: (i) it illustrates the application of the typing rules as concrete examples; and (ii) saves the programmer from doing it by hand. In short, it helps the language designer to check how the rules actually work.

Before proceeding, we recall the types of FS syntax, defined in Figure 3.1.

$$\begin{aligned}(\textit{Types}) \quad T \quad ::= \quad & \widetilde{T} \to T \mid \texttt{bool} \mid \texttt{uint} \mid \texttt{address} \mid \\ & \texttt{unit} \mid \texttt{mapping}(T \Rightarrow T) \mid C\end{aligned}$$

3.2.1 Type System Judgements

The FS type environment is defined as follows:

$$(\textit{Type environment}) \quad \Gamma \quad ::= \quad \emptyset \mid \Gamma, x : t \mid \Gamma, a : \text{Address}$$

Further, Di Pirro defines the type system judgements as the relation between types, terms and environments. Thus, representing this relation as:

$$\Gamma \vdash e : T$$

where Γ is the typing context input, and $e : t$ means that the expression e has type t, given the assumptions in Γ.

As we explain next, we made changes to the judgements representation. The typing relation is now defined as:

$$\Gamma \vdash e : T \triangleright \Gamma'$$

where Γ and Γ' are the input and output typing contexts , respectively., and again $e : t$ means that the expression e has type t in Γ.

This change is due to the typing rule Decl in page 91 of Di Pirro's thesis [14]. This rule makes the original type system not driven by the syntax of expressions. To solve the issue, we introduce the output context Γ in the sequential composition typing rule, as shown below. Furthermore, this alteration removes the need for the particular rule Decl.

$$\frac{\Gamma \vdash e_1 : T_1 \triangleright \Gamma' \quad \Gamma' \vdash e_2 : T_2 \triangleright \Gamma''}{\Gamma \vdash e_1 ; e_2 : T_2 \triangleright \Gamma''} \ \textsc{Seq}$$

In Annex A, we present the revised typing rules with the added output Γ context.

```
let rec typecheck whites gamma ct ty t =
  match ty with
  | None -> for i = 0 to whites - 1 do Format.eprintf " " done ;
    Format.eprintf "%a , %a |- %a : " pp_contract ct pp_gamma gamma pp_term t;
        begin match t with
        | TmSeq(e1, e2) -> Format.eprintf "(SEQ) @." ;
            ignore(typecheck (whites + 2) gamma ct None e1);
            typecheck (whites + 2) gamma ct None e2;
    ...
  | Some ty -> for i = 0 to whites - 1 do Format.eprintf " " done ;
    Format.eprintf "%a , %a |- %a : %a@." pp_contract ct pp_gamma gamma
        pp_term t pp_typ ty ;
        begin match t with
        | TmSeq(e1, e2) -> Format.eprintf "(SEQ) @." ;
            ignore(typecheck (whites + 2) gamma ct None e1);
            typecheck (whites + 2) gamma ct (Some ty) e2;
        ...
```

Figure 3.45: OCaml typechecker - simplified

3.2.2 OCaml Typechecker

In this section, we briefly present our OCaml implementation of FS' typechecker. In Figure 3.45, we show part of the typechecker code, where we present the implementation of the typing rule Seq. The `typecheck` function has as input the amount of white spaces (in order to distinguish which lines are which premises); a context `gamma`; a contract table `ct`; a type `ty`; and a term `t`.

Firstly, we need to match the type of the statement `ty` to some type or none, as in the case of the rule Seq the conclusion of the rule has no information of the type of the e_1 statement. As a result, we then get two evaluations, one for each case. If we have the type of term, we recursively continue the evaluation of the statement. Else, the evaluation continues with no type, until we reach the end of the typification and a type is returned through the judgement of an axiom.

The typechecker allows us to prove if the program is well-typed, as well as it outputs the derivation tree.

3.2.3 Examples

In this example we will prove that the example defined in Section 3.1.3.1 is well-typed, as well as it provides the derivation trees with the typing rules. This example is the same as the one Di Pirro uses to prove that the program is well-typed. The expression we are going to evaluate is the following:

$$\begin{aligned} e = \quad & EOAx = newEOA.value(500)(); \\ & Banky = newBank.value(0)([]); \\ & y.deposit.value(100).sender(address(x))() \end{aligned}$$

The Γ is initially empty, and we use placeholders like (1) to indicate another derivation tree.

$$\dfrac{\dfrac{\dfrac{\checkmark}{\Gamma \vdash 500 : uint \triangleright \Gamma}\ \text{Nat}}{\Gamma \vdash EOA\ x = new\ EOA.value(500)() : EOA \triangleright \Gamma}\ \text{NEW} \qquad \dfrac{(1)}{\Gamma, x : EOA \vdash e' : unit \triangleright \Gamma, x : EOA}}{\Gamma \vdash EOA\ x = new\ EOA.value(500)(); e' : unit \triangleright \Gamma, x : EOA}\ \text{DECL}$$

The derivation tree represented by (1) is as follows, where $\Gamma' = \emptyset, x : EOA$.

$$(1)\ \dfrac{\dfrac{\dfrac{\checkmark}{\Gamma' \vdash 0 : uint \triangleright \Gamma'}\ \text{Nat} \qquad \dfrac{\checkmark}{\Gamma' \vdash [] : mapping(address \Rightarrow uint)}\ \text{MAPPING}}{\Gamma' \vdash Bank\ y = new\ Bank.value(0)([]) : Bank \triangleright \Gamma'}\ \text{NEW} \qquad (2)}{\Gamma' \vdash Bank\ y = new\ Bank.value(0)([]); e'' : unit \triangleright \Gamma', y : Bank}\ \text{DECL}$$

And where (2) is, $\Gamma'' = \emptyset, x : EOA, y : Bank$:

$$(2)\ \dfrac{\dfrac{\dfrac{\checkmark}{\Gamma'' \vdash x : EOA \triangleright \Gamma''}\ \text{REF}}{\Gamma'' \vdash address(x) : Address \triangleright \Gamma''}\ \text{ADDR} \qquad (3)}{\Gamma'' \vdash y.deposit.value(100).sender(address(x))() \triangleright \Gamma''}\ \text{CALLTOPLEVEL}$$

$$(3)\ \dfrac{\dfrac{\checkmark}{\Gamma'' \vdash y : Bank \triangleright \Gamma''}\ \text{REF} \qquad \dfrac{\checkmark}{\Gamma'' \vdash 100 : uint}\ \text{NAT}}{\Gamma'' \vdash y.deposit.value(100)() : unit \triangleright \Gamma''}\ \text{CALL}$$

In Figure 3.46, we present the output of this same example in the OCaml typechecker.

```
1  contract Bank {(mapping(address => uint) : balance)} unit deposit () ; uint
       getBalance () ; unit transfer ((address : to)(uint : amount)) ; unit
       withdraw ((uint : amount))
2  contract EOA {}
3   ,  |- EOA x = new EOA.value(500)() ; Bank y = new Bank.value(0)([]) ; y.
        deposit.value(100).sender(address(x))() : EOA
4  (DECL)
5     ,  |- new EOA.value(500)() : EOA
6  (NEW)
7       ,  |- 500 : uint
8  (NAT)
9    , (x : EOA);  |- Bank y = new Bank.value(0)([]) ; y.deposit.value(100).
         sender(address(x))() : Bank
10 (DECL)
11      , (x : EOA);  |- new Bank.value(0)([]) : Bank
12 (NEW)
13        , (x : EOA);  |- 0 : uint
14 (NAT)
15        , (x : EOA);  |- [] : mapping(address => uint)
16 (MAPPING)
17      , (x : EOA); (y : Bank);  |- y.deposit.value(100).sender(address(x))() :
            unit
18 (CALLTOPLEVEL)
19        , (x : EOA); (y : Bank);  |- address(x) : address
20 (ADDR)
21          , (x : EOA); (y : Bank);  |- x : EOA
22 (REF)
23        , (x : EOA); (y : Bank);  |- y.deposit.value(100)() : unit
24 (CALL)
25          , (x : EOA); (y : Bank);  |- y : Bank
26 (REF)
27          , (x : EOA); (y : Bank);  |- 100 : uint
28 (NAT)
29 Success
```

Figure 3.46: OCaml typechecker output

4

Flint-2

Flint is a statically-typed language created for writing smart contracts [20, 34, 35]. It was developed originally by Franklin Schrans and carried out by other students at Imperial College Department of Computing under Professors Susan Eisenbach and Sophia Drossopoulou orientation. Inspired by the Swift Programming Language [5], Flint was designed to be integrated in the Ethereum blockchain, by supporting Solidity [38] contracts and other applications to interact with contracts written in Flint.

Like many, Schrans [34, 35] believes that the vulnerabilities found in smart contracts were mostly derived from the programmers own mistakes. As we emphasise throughout this paper, these types of errors cannot occur under the smart contracts' execution, as many users rely on the correct behaviour of these programs, which mostly deal with sensitive information, such as great amounts of money and voting systems. Furthermore, these programs cannot be changed once they have been deployed, which adds an even bigger emphasis on safe programming.

With this in mind, Schrans and the Imperial College team took upon themselves to design a language that made it harder on programmers to write unsafe contracts. They employed caller blocks to protect sensitive functions from unauthorised callers, as well as to force programmers to infer about the correct behaviour of a contract.

Additionally, these *protection blocks* can prevent incorrect behaviour of contracts, as it uses typestates. These blocks restrict when the code can be executed, as states are checked statically for internal calls, and at runtime for external calls to the contract.

In Figure 4.1, we present a method from an auction written in Flint, where it checks the state before executing the function.

Flint-2 is the newest iteration of Flint. Rust [1] based, Flint-2 captures the "safety-oriented" characteristics once introduced in Flint, being the protection blocks the main feature we want to focus on.

Flint-2 was formalised by us, as we took inspiration from the configurations and structure of di Piro's work [14]. Flint-2 poses an extra challenge as it had no formalisation,

[1] https://www.rust-lang.org

```
1 // Enumeration of states.
2 contract Auction (AuctionRunning, AuctionEnded) {}
3
4 Auction @(AuctionRunning) :: caller <- (owner) {
5   public endAuction() {
6     // ...
7     become AuctionEnded
8   }
9 }
```

Figure 4.1: Example of an Auction in Flint

like many other smart contract languages, but also because it takes advantage of typestates and caller groups by restricting the function callings. These features give Flint-2 an advantage when it comes to ensuring contract's safety.

4.1 Racket Implementation / Executable Semantics

In this section, we present our definition of the syntax of a core version of Flint-2, based on the language guide [19] and examples available [2]. It is important to note that we left some parts unimplemented and made some modifications of our own, which are addressed in Section 4.1.1.1.

4.1.1 Syntax

We assume the following sets:

1. $\mathcal{CN}$, of contract names, ranged over by C;
2. $\mathcal{TS}$, of typestates names, ranged over by ts (possibly indexed);
3. $\mathbb{CL}$, of caller groups, ranged over by cl (possibly indexed);
4. $\mathcal{FN}$, of function names, ranged over by f (possibly indexed);
5. $\mathcal{X}$, of variable names, ranged over by x (possibly indexed);

Moreover, we assume that a set of integer values ranged over by n, and a set of addresses ranged over by a. In Figure 4.2, we present a simplified version of Flint-2's syntax[3].

Contract declaration. The contracts are made up by the following components, in this order:

- the keyword `contract`;
- the contract name C;

[2] https://github.com/flintlang/flint-2/tree/master/tests
[3] https://github.com/flintlang/flint-2/blob/master/docs/guide.md#language-guide

(*Contract Declaration*)	CD	::=	`contract` $C\ (ts\ \tilde{fs})\ \{\widetilde{vd}\}\ K\ \widetilde{PB}$
(*Constructor Block*)	K	::=	$C :: (cl\ \tilde{cl})\ \{(\texttt{public init}(\widetilde{x:t})\ \{\tilde{e}\})\ \tilde{F}\}$
(*ProtectionBlock*)	PB	::=	$C\ @\ (ts\ \tilde{fs}) :: (cl\ \tilde{cl})\{\widetilde{F}\}$
(*FunctionDeclaration*)	F	::=	`public func` $f\ (\widetilde{x:t}) \rightarrow t\{\tilde{e}$ `return` $e\}$
(*Variable Declaration*)	VD	::=	`let` $x:t=v$ | `var` $x:t=v$ | `let` $x:t$ | `var` $x:t$
(*Expressions*)	e	::=	x | v | `self` | `if` e `then` e `else` e | `var` $x:t=e$ | `var` $x:t$ | `let` $x:t=e$ | `let` $x:t$ | $x=e$ | $e.x=e$ | $e.x$ | C.`init`$(\widetilde{x:e}).a$ | $e.f\ (\widetilde{x:e})$ | $f\ (\widetilde{x:e})$ | $e.f\ (\widetilde{x:e})$.`sender`(a) | `return` e | $e\ \tilde{e}$ | `become` ts | $e[e]$ | $e[e:e]$ | e a-op e | e b-op1 e | e b-op2 e
(*Values*)	v	::=	n | a | true | false | unit | M
(*Types*)	t	::=	Int | Address | Bool | Void | $(t:t)$

Figure 4.2: Syntax of the Racket Implementation of Flint-2

- a list of its typestates ts;
- variable and constant declarations and assignments VD;
- a special protection block K for the function `init`;
- and protection blocks (PB).

Constructor block. The constructor block K is composed by the name of the contract C, a caller group cl, a function `init`, and other functions F .

We added this constructor to the syntax for two main reasons. Firstly, we noticed that as the function init is the *entry point* of the contract, in most of the examples, this particular protection block did not have the typestate protection. Furthermore, by adding a constructor to the contract we ensure that every contract has a function `init`.

Protection block. Unlike in Flint-2, the *typestate* protection is not optional. Every *protection block* must consist of the contract name C; the typestate protection ts; the caller group cl; and finally the functions F.

Additionally, we make sure every protection block has at least one typestate and one caller member, even if it is any.

Function declaration. In our Racket implementation of Flint-2, every function is declared `public`, and has a return type t. With this said, our function declaration is composed by: a function name f; a list of parameters $(\widetilde{x:t})$; a return type t; a list of statements $\tilde{e}$; and a return statement.

Expressions. e denotes the expressions. x denotes a variable, v a value, and self is a variable that refers to the address of the contract which is calling the current function.

The expression if e then e else e defines the if expression. var $x : t = e$ and let $x : t = e$ denote variable and constant declaration respectively. Both $x = e$ and $e_1.x = e$ are variable assignment, where if e_1 evaluates to an address; and $e.x$ denote an access to a field (variable, constant, function). The expression $f(\widetilde{x : e})$ is the function call. We added another call function with a slight nuance, $e_1.f(\widetilde{x : e})$, where the programmer can declare the contract's address of which the function belongs, as e_1 evaluates to a contract's address. The expression C.init$(\widetilde{x : e}).a$ deploys a new contract C to the blockchain with an address a, and invokes the function `init`. $e.f(\widetilde{x : e})$.sender(a) is a special top-level call function where we can declare the address of the contract who is calling the function. return e is the return statement, the expression $e\ \tilde{e}$ is sequential composition, and the become statement is defined as become e. Arithmetic and boolean operations are denoted as e a-op e; e b-op1 e; and e b-op2 e, where a-op represents the following arithmetic operators: + (addition); - (subtraction); * (multiplication); / (division); ** (exponentiation); and % (modulus). b-op1 denotes < (less than); <= (less than or equal to); == (equal to); >= (greater than or equal to); > (greater than); != (not equal to), and b-op2 && (logical and). $e[e']$ denotes the reading of a variable with e' as a key, where e evaluates to a map, whilst $e[e' : e'']$ is for the writing of a pair with e'' as value.

Values. v ranges over values. true, false and unit have the expected meaning. n represents any integer and a indicates an address. M denotes a map with key-value pairs of values.

Types t ranges over types. Bool, Int, Void and String have the expected meaning. Address represents an address, which in out Racket implementation means that it must start with the character a. The type of mapping is $(t : t')$, being t the type of the keys and t' of values.

4.1.1.1 Revised Syntax

Due to Racket's syntax, we had to adapt Flint-2's grammar as such:

- Every expression must be in parenthesis.
- ; is reserved by Racket for comments in the code, so expressions cannot be separated using a semicolon.
- . is reserved by Racket, so we replaced it with a ->.
- any is a reserved word, we use `anycaller` and `anystate` to represent it in the caller group and typestate protection in protection blocks.
- ... denotes the repetition of the previous pattern.
- The expressions $e[e]$ and $e[e : e]$ in Racket have the forms $e[e ::]$ and $e[e :: e]$ respectively, to make the expression evaluation deterministic.

These are the changes we made to the original Flint-2's grammar, mainly because we wanted to simplify it:

- The type `Void` must be explicit in the return type of a function, as every function terminates with a return statement.
- We removed type String, as in Racket, we cannot distinguish between a variable and a String, leaving us with an inconsistent execution
- Dynamic, range and external types were not implemented, except for Dictionary.
- `public` is the only function and variable modifier.
- Structs were not implemented.
- The caller group is either a list of addresses or `anycaller`.
- Contract and external traits were not implemented.
- External call and an attempt to call a function were also not implemented.
- Type cast was not implemented.
- Loops and do catch blocks were not implemented.
- Enumerations were not implemented.

4.1.1.2 Evaluation Contexts

In Figure 4.3, we define the evaluation contexts of our take on the Flint-2 language. The contexts contain a hole, denoted by [], which represents the location of the next evaluation [17]. The evaluation contexts E guarantee that the evaluation of expressions is done from left to right [45], by uniquely identifying the next expression that will be evaluated [17]. We can formalize it as the following:

$$\frac{\langle e \rangle \rightarrow \langle e' \rangle}{\langle E[e] \rangle \rightarrow \langle E[e'] \rangle}$$

$$\begin{aligned} E \quad ::= \quad & [] \mid \text{if } E \text{ then } e \text{ else } e \mid \text{var } x : t = E \mid \text{let } x : t = E \mid \\ & C.\text{init}(\widetilde{x : v}, x : E, \widetilde{x : e}) \mid E.f(\widetilde{x : e}) \mid a.f(\widetilde{x : v}, x : E, \widetilde{x : e}) \mid \\ & f(\widetilde{x : v}, x : E, \widetilde{x : e}) \mid E.f(\widetilde{x : e}).\text{sender}(a) \mid \\ & a.f(\widetilde{x : v}, x : E, \widetilde{x : e}).\text{sender}(a) \mid E.x \mid E.x = e \mid a.x = E \\ & E\,\tilde{e} \mid \text{return } E \mid E \text{ a-op } e \mid v \text{ a-op } E \mid E \text{ b-op1 } e \mid v \text{ b-op1 } E \mid \\ & E \text{ b-op2 } e \mid \text{bool b-op2 } E \mid E[e] \mid M[E] \mid E[e : e] \mid M[E : e] \mid M[v : E] \end{aligned}$$

Figure 4.3: Evaluation Contexts

$$
\begin{array}{lllll}
(Blockchain) & \beta & ::= & \emptyset \mid env - \beta \cdot [a \mapsto ((C, \widetilde{s:v}, n), (\widetilde{c:v}), (\widetilde{x:v}))] \mid \\
 & & & env - \beta \cdot [x \mapsto v] \\
(Call\ Stack) & env - \sigma & ::= & \emptyset \mid env - \sigma \cdot a \\
(Type\ State\ Stack) & CTS & ::= & \emptyset \mid CTS \cdot [a \mapsto \tilde{ts}]
\end{array}
$$

Figure 4.4: Flint-2's Environments

4.1.2 Operational Semantics

In this section we present the operational semantics of the Flint-2 language. It is a binary relation between configurations $\mathscr{C}$, which describes the state of the execution. The configuration of the Flint-2 language is defined by a tuple $\langle e, \beta, \sigma, CTS \rangle$, which is comprised of an expression that is going to be evaluated over the following environments: the blockchain, the call stack, and the typestate stack. All three of these environments store the state of the program, in which the blockchain stores the information of the contracts; the call stack keeps record of the addresses that call functions during the current transaction, and when empty it means that the transaction was successful; and the typestate stack which tracks the typestates of each contract.

We define the evaluation relation as the transitions between states, $\mathscr{C} \to \mathscr{C}$, where $\to$ represents the one step evaluation of an expression [17], and is defined inductively by the inference rules applied to our syntax of Flint-2 in Section 3.1.2.3.

Subsequently, we define the multi-step evaluation $\to^*$ as the reflexive, transitive closure of the relation $\to$ [33].

$$
\mathscr{C} \to^* \mathscr{C} \qquad \frac{\mathscr{C} \to^* \mathscr{C}' \quad \mathscr{C}' \to \mathscr{C}''}{\mathscr{C} \to^* \mathscr{C}''}
$$

4.1.2.1 Environments

We based Flint-2's environments on the presentation of another programming language for smart contracts, Featherweight Solidity [14]. In Figure 4.4, we present the environments we see fit for Flint-2's formalisation.

Blockchain. Flint-2 maps the contracts' address a to its contract name C, state variables s and balance n, its constants c and the arguments of functions x and its respective values. Mappings of variables and its values are also kept in the *blockchain*.

Remark (Contract Representation)**.** *To represent the contracts in the blockchain, we simplified the notation of the state, constant and local variables and their respective values as such:*

$$\widetilde{s:v} \text{ as } s$$

$$\widetilde{c:v} \text{ as } c$$

$$\widetilde{x:v} \text{ as } y$$

Call stack 's only purpose is to keep the addresses of the contracts that make function calls. When a return statement is evaluated, the address on top of the stack is removed.

Typestate stack contains the record of the program's states during execution, being the top of the stack the current state. When a contract calls a function, the current state has to match with the state declared in the protection blocks. When a `become ts` is evaluated, `ts` is pushed to the top of the stack, becoming its current state.

4.1.2.2 Auxiliary functions

In this section, we present some functions that make the operational rules simpler and more readable.

Top(σ) returns the address on the top of the environment σ.

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

Top(CTS) returns the current typestate.

$$\text{Top}(CTS) = \begin{cases} ts & \text{if } CTS = CTS' \cdot ts \\ \emptyset & \text{if } CTS = \emptyset \end{cases}$$

finit returns the variables and constant declaration of the contract $C\ vars_n$, and the body of the function `init` with the statement `return unit` appended so the contract's address can be popped from σ at the end of the evaluation.

$$\begin{aligned} \text{finit}(C, \tilde{x}) = (\widetilde{VD}, e \cdot \text{return unit}) \text{ if } \quad & classes(C) = \texttt{contract}\ \ C(\tilde{ts})\{\widetilde{VD}\}K\ \widetilde{PB} \\ & \wedge K = C :: (\tilde{cl})\{(\texttt{public}\ \ \texttt{init}(\widetilde{x:t})\{e\})\ \tilde{F}\} \end{aligned}$$

fbody checks if function f exists in the contract C, and returns both the body of the function and the set of typestates and callers that guard the protection block pb.

$$\begin{aligned} \text{fbody}(C, f, \tilde{x}) = (e' \text{ return } e, \tilde{ts'}, \tilde{cl}, t) \text{ if } \quad & classes(C) = \texttt{contract}\ \ C(\tilde{ts})\{\widetilde{VD}\}\ K\ \widetilde{PB} \\ & \wedge pb = C\ @\ (\tilde{ts'}) :: (\tilde{cl})\ \{\tilde{F}\} \\ & \wedge pb \in \widetilde{PB} \\ & \wedge fc = \texttt{public func}\ \ f(\widetilde{x':t'}) \rightarrow t\ \{e'\ \texttt{return}\ \ e\} \\ & \wedge fc \in \tilde{F} \\ & \wedge \tilde{x} = \tilde{x'} \end{aligned}$$

fbodyinit is applied when the function f belongs to the protection block of the `init` function. Just like `fbody`, it takes as input the contract C, the function f and the set of the arguments $\{x_v\}$. It then returns only the body of f and the set of the caller group, as this protection block has no typestate restrictions.

$$\begin{aligned}\text{fbodyinit}(C, f, \tilde{x}) = (e' \text{ return e}, \widetilde{cl}, t) \text{ if } \quad & classes(C) = \texttt{contract } C(\widetilde{ts})\{\widetilde{vd}\}\ K\ \widetilde{PB} \\ & \wedge K = C :: (\widetilde{cl})\{(\texttt{public init}(\widetilde{x' : t'})\{e\})\ \widetilde{F}\} \\ & \wedge fc = \texttt{public func } f(\widetilde{x'' : t''}) \to t\ \{e' \texttt{ return } e\} \\ & \wedge fc \in \widetilde{F} \\ & \wedge \tilde{x} = \widetilde{x''}\end{aligned}$$

sv returns the state variables, and their types, of a given contract C.

$$\text{sv}(C) = (\widetilde{x : t}) \text{ if } \quad classes(C) = \texttt{contract } C(\tilde{ts})\{\widetilde{vd}\}\ K\ \widetilde{PB}$$

where the variables in $\widetilde{x : t}$ are those in each vd in $\widetilde{vd}$, which is in turn of one of the following forms: `let` $x : t = v$; `var` $x : t = v$; `let` $x : t$;and `var` $x : t$.

uptbal updates the balance of the contract with address a.

$$\begin{aligned}\text{uptbal}(\beta, a, n) = \beta[a \mapsto ((C, \tilde{s}, n' + n), \tilde{c}, \tilde{y})] \text{ if } \quad & \beta(a) = ((C, \tilde{s}, n'), \tilde{c}, \tilde{y}) \\ & \wedge\ n' + n > 0\end{aligned}$$

4.1.2.3 Rules

The rules presented in this section follow the $\langle e, \beta, \sigma, CTS \rangle$ configuration, where e is the expression, and the succeeding ones are the environments: a blockchain β; a call stack σ; a typestate map CTS. Boolean and arithmetic operation rules are omitted.

If expression. The rules for if expressions have the standard meaning, and both *then* and *else* branches are mandatory.

$$\frac{}{\langle \text{if true then } e_1 \text{else } e_2, \beta, \sigma, \text{CTS} \rangle \to \langle e_1, \beta, \sigma, \text{CTS} \rangle}\ \text{IF-TRUE}$$

$$\frac{}{\langle \text{if false then } e_1 \text{else } e_2, \beta, \sigma, \text{CTS} \rangle \to \langle e_2, \beta, \sigma, \text{CTS} \rangle}\ \text{IF-FALSE}$$

Variable and constant declaration. Rule DeclVar models variable declaration, whilst DeclCons models constants.

Remark (x not in $\tilde{x}$)**.** *In the following rules, the notation $x \notin \tilde{x}$ represents that the x does not belong to the set of identifiers $\tilde{x}$ in $\widetilde{x : v}$.*

$$\frac{\begin{array}{c}\text{Top}(\sigma) = a \quad x \notin \tilde{x} \\ \beta(a) = (s, c, \widetilde{x : v}) \\ \beta' = \beta[a \mapsto (s, c, \widetilde{x : v} \cdot x : v)]\end{array}}{\langle \text{var } x : t = v, \beta, \sigma, \text{CTS}\rangle \rightarrow \langle v, \beta', \sigma, \text{CTS}\rangle} \text{ DECLVAR}$$

$$\frac{\begin{array}{c}\text{Top}(\sigma) = a \quad x \notin \tilde{c} \\ \beta(a) = ((C, \tilde{s}, n), \widetilde{c : v}, \tilde{y}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \widetilde{c : v} \cdot x : v, \tilde{y})]\end{array}}{\langle \text{let } x : t = v, \beta, \sigma, \text{CTS}\rangle \rightarrow \langle v, \beta', \sigma, \text{CTS}\rangle} \text{ DECLCONS}$$

For variables and constants that are not initialised, we made a rule for each type, so when the type t is Int its default value v_t is 0, Bool is false, and Adress is aNULL.

$$\frac{\begin{array}{c}\text{Top}(\sigma) = a \quad x \notin \tilde{x} \\ \beta(a) = ((C, \tilde{s}, n), \tilde{c}, \widetilde{x : v}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \widetilde{x : v} \cdot x : v_t)]\end{array}}{\langle \text{var } x : t, \beta, \sigma, \text{CTS}\rangle \rightarrow \langle v_t, \beta', \sigma, \text{CTS}\rangle} \text{ DECLVAR-T}$$

$$\frac{\begin{array}{c}\text{Top}(\sigma) = a \quad x \notin \tilde{c} \\ \beta(a) = ((C, \tilde{s}, n), \widetilde{c : v}, \tilde{y}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \widetilde{c : v_t} \cdot x : v_t, \tilde{y})]\end{array}}{\langle \text{let } x : t, \beta, \sigma, \text{CTS}\rangle \rightarrow \langle v_t, \beta', \sigma, \text{CTS}\rangle} \text{ DECLCONS-T}$$

Function call. To simplify our approach to Flint-2, we removed some of the call functions, such as external and attempt calls, and kept the function call. If the programmer does not declare the contract instantiated in the calling of the function, we add the contract name on top of the stack to know what function to call.

All these rules have the same behaviour, they return the body of the function f, and append the input arguments to the scope of variables of the contract with address a. Additionally, when a function is called, it is assumed that the caller is the address on top of the call stack. Therefore, we push the same address that is on top of σ, so at the end of the function body evaluation it can be popped.

Rule CALL is applied when function f belongs to a protection block and the typestate guard is different than any, whereas rule CALL-2 is applied whenever function f is in the constructor block of the contract. Lastly, the rule CALL-ANY is for when the typestate guard of the protection block is anystate. Rule TRY is the same as CALL, it just has a different syntax, whereas rule CALL-SENDER-R is called when the function caller does not have enough gas to proceed with the transaction.

$$\frac{\begin{array}{c}C \in \text{classes} \quad \text{fbody}(C, f, \tilde{x}) = (e, \widetilde{ts}, \widetilde{cl}, t) \quad \text{Top}(CTS) \in \widetilde{ts} \quad a \in \widetilde{cl} \\ e_s = e\{\texttt{self} := \text{Top}(\sigma)\} \quad \beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), \text{Top}(\sigma), -\text{amount})\end{array}}{\langle a.f(\widetilde{x : v}), \beta, \sigma, \text{CTS}\rangle \rightarrow \langle e_s, \beta'', \sigma \cdot \text{Top}(\sigma), \text{CTS}\rangle} \text{ CALL}$$

$$\frac{\begin{array}{c}C \in \text{classes} \quad \text{fbody}(C, f, \tilde{x}) = (e, \widetilde{ts}, \widetilde{cl}, t) \quad \text{Top}(CTS) \in \widetilde{ts} \quad a \in \widetilde{cl} \\ e_s = e\{\texttt{self} := \text{Top}(\sigma)\} \quad \beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), \text{Top}(\sigma), -\text{amount})\end{array}}{\langle \text{try ? } (a.f(\widetilde{x : v})), \beta, \sigma, \text{CTS}\rangle \to \langle e_s, \beta'', \sigma \cdot \text{Top}(\sigma), \text{CTS}\rangle} \text{ TRY}$$

$$\frac{\begin{array}{c}C \in \text{classes} \quad \text{fbodyinit}(C, f, \tilde{x}) = (e, \widetilde{cl}, t) \quad a \in \widetilde{cl} \\ e_s = e\{\texttt{self} := \text{Top}(\sigma)\} \quad \beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), \text{Top}(\sigma), -\text{amount})\end{array}}{\langle a.f(\widetilde{x : v}), \beta, \sigma, \text{CTS}\rangle \to \langle e, \beta'', \sigma \cdot \text{Top}(\sigma), \text{CTS}\rangle} \text{ CALL-2}$$

$$\frac{\begin{array}{c}C \in \text{classes} \quad \text{fbody}(C, f, \widetilde{x : v}) = (e, \widetilde{ts}, \widetilde{cl}, t) \quad a \in \widetilde{cl} \quad \texttt{anystate} \in \widetilde{ts} \\ e_s = e\{\texttt{self} := \text{Top}(\sigma)\} \quad \beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), \text{Top}(\sigma), -\text{amount})\end{array}}{\langle a.f(\widetilde{x : v}), \beta, \sigma, \text{CTS}\rangle \to \langle e, \beta'', \sigma \cdot \text{Top}(\sigma), \text{CTS}\rangle} \text{ CALL-ANY}$$

$$\frac{\text{uptbal}(\beta, \text{Top}(\sigma), -\text{amount}) = \bot}{\langle a.f(\widetilde{x : v}), \beta, \sigma, \text{CTS}\rangle \to \langle \text{revert}, \beta, \sigma, \text{CTS}\rangle} \text{ CALL-SENDER-R}$$

Furthermore, we added a special top-level call function, where we can declare the address of the contract that is calling the function in question. All rules below, Call-Sender; Call-Sender-2; Call-Sender-Any; and Call-Sender-R, are equivalent to the ones above, being the only difference the added sender address a_s, that will be pushed to the top of σ.

$$\frac{\begin{array}{c}\beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ C \in \text{classes} \quad \text{fbody}(C, f, \tilde{x}) = (e, \widetilde{ts}, \widetilde{cl}, t) \\ \text{Top}(CTS) \in \widetilde{ts} \quad e_s = e\{\texttt{self} := a\} \quad a \in \widetilde{cl} \quad \text{Top}(\sigma) = \emptyset \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), a_s, -\text{amount})\end{array}}{\langle a.f(\widetilde{x : v}).\text{sender}(a_s), \beta, \sigma, \text{CTS}\rangle \to \langle e_s, \beta'', \sigma \cdot a_s, \text{CTS}\rangle} \text{ CALL-SENDER}$$

$$\frac{\begin{array}{c}\beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ C \in \text{classes} \quad \text{fbodyinit}(C, f, \tilde{x}) = (e, \widetilde{cl}, t) \\ e_s = e\{\texttt{self} := a\} \quad a \in \widetilde{cl} \quad \text{Top}(\sigma) = \emptyset \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), a_s, -\text{amount})\end{array}}{\langle a.f(\widetilde{x : v}).\text{sender}(a_s), \beta, \sigma, \text{CTS}\rangle \to \langle e_s, \beta'', \sigma \cdot a_s, \text{CTS}\rangle} \text{ CALL-SENDER-2}$$

$$\frac{\begin{array}{c}\beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ C \in \text{classes} \quad \text{fbody}(C, f, \tilde{x}) = (e, \widetilde{ts}, \widetilde{cl}, t) \quad \texttt{anystate} \in \widetilde{ts} \\ e_s = e\{\texttt{self} := a\} \quad a \in \widetilde{cl} \quad \text{Top}(\sigma) = \emptyset \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), a_s, -\text{amount})\end{array}}{\langle a.f(\widetilde{x : v}).\text{sender}(a_s), \beta, \sigma, \text{CTS}\rangle \to \langle e_s, \beta'', \sigma \cdot a_s, \text{CTS}\rangle} \text{ CALL-SENDER-ANY}$$

$$\frac{\text{uptbal}(\beta, \text{Top}(\sigma), -\text{amount}) = \bot}{\langle a.f(\widetilde{x:v}).\text{sender}(a_s), \beta, \sigma, \text{CTS}\rangle \to \langle \text{revert}, \beta, \sigma, \text{CTS}\rangle} \text{ CALL-SENDER-R}$$

Variable lookup. Rule StateSel returns the value of the state variable x of the contract with the address a. Var returns the variable x declared in $\{x_{vars}*\}$ of the contract address on top of σ.

$$\frac{\beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \quad x \in \tilde{s} \cup \tilde{c}}{\langle a.x, \beta, \sigma, \text{CTS}\rangle \to \langle v, \beta, \sigma, \text{CTS}\rangle} \text{ STATESEL}$$

$$\frac{\begin{array}{c} a = \text{Top}(\sigma) \quad x \in \tilde{y} \\ \beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \end{array}}{\langle x, \beta, \sigma, \text{CTS}\rangle \to \langle v, \beta, \sigma, \text{CTS}\rangle} \text{ VAR}$$

Variable assignment. Both these rules enable the mutation of variables. Rule State-Ass allows the alteration of the state variable x's value, while rule ASS modifies the value of the given variable.

$$\frac{\beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \quad x \in \tilde{s} \cup \tilde{c}}{\langle a.x = v, \beta, \sigma, \text{CTS}\rangle \to \langle v, \beta[a.x \mapsto v], \sigma, \text{CTS}\rangle} \text{ STATEASS}$$

$$\frac{\beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \quad x \in \tilde{y}}{\langle x = v, \beta, \sigma \cdot a, \text{CTS}\rangle \to \langle v, \beta[a.x \mapsto v], \sigma \cdot a, \text{CTS}\rangle} \text{ ASS}$$

Sequential composition. Rule Seq discards v, so we can go to the evaluation of e.

$$\frac{}{\langle v\ e, \beta, \sigma, \text{CTS}\rangle \to \langle e, \beta, \sigma, \text{CTS}\rangle} \text{ SEQ}$$

$$\frac{\sigma = \beta_0 \cdot \tilde{a}}{\langle \text{revert}\ e, \beta, \sigma, \text{CTS}\rangle \to \langle \text{revert}, \beta_0, \sigma, \text{CTS}\rangle} \text{ SEQ-R}$$

Become statement. Become rule modifies the current typestate of the program by adding ts to the top of the stack CTS.

$$\frac{}{\langle \text{become}\ ts, \beta, \sigma, \text{CTS}\rangle \to \langle \text{unit}, \beta, \sigma, \text{CTS} \cdot ts\rangle} \text{ BECOME}$$

Return statement. The Return rule is applied when the evaluation of a function was successful, and we can pop an element from σ.

$$\frac{}{\langle \text{return}\ v, \beta, \sigma \cdot a, \text{CTS}\rangle \to \langle v, \beta, \sigma, \text{CTS}\rangle} \text{ RETURN}$$

Contract initialization. INIT rule deploys a new contract C to the blockchain. We added it to our syntax in order to be able to call function `init` by specifying the contract we wish to invoke. We have to specify the contract's address a, that in the real world would be given by the blockchain.

The variables provided as input have to be the same, and in the same order, as the ones declared as arguments of function `init`. The provided address a cannot be associated to any other contract, since it has to be unique. All the `self` and $\tilde{x}$ instances in the expression returned by the auxiliary function `initc` will be substituted by a and $\tilde{v}$ respectively.

The contract is appended to the blockchain, with the state variables and the constant declared and with an initial balance of 0. If those variables have no initial value, they will be added to the blockchain with their respective default value, as their values will be added later when e_s, the function `init`'s body, is evaluated. The address of the contract will be added to the top of σ.

$$\frac{\begin{array}{c}\text{finit}(C, \widetilde{x:v}) = (\widetilde{s:v'}, \widetilde{c:v''}, e) \\ e_s = e\{\text{self} := a, \tilde{x} := \tilde{v}\} \quad a \notin \text{dom}(\beta) \\ cn = a \mapsto ((C, \widetilde{s:v'}, n), \widetilde{c:v''},)\end{array}}{\langle C.\text{init}(\widetilde{x:v}).n.a, \beta, \sigma, \text{CTS}\rangle \rightarrow \langle e_s, \beta \cdot cn, \sigma \cdot a, \text{CTS}\rangle}\ \text{INIT}$$

Mappings. The following rules pertain to the mappings.

$$\frac{}{\langle M[v_1], \beta, \sigma, \text{CTS}\rangle \rightarrow \langle M(v_1), \beta, \sigma, \text{CTS}\rangle}\ \text{MAPSEL}$$

$$\frac{M' = M \backslash \{(v_1, M(v_1))\} \cup \{v_1, v_2\}}{\langle M[v_1 : v_2], \beta, \sigma, \text{CTS}\rangle \rightarrow \langle M', \beta, \sigma, \text{CTS}\rangle}\ \text{MAPASS}$$

The rule MAPSEL evaluates the expression $M[v_1]$ by returning the value in M in which v_1 is the key. MAPASS assigns the value of v_2 to v_1 and appends it to M. If v_1 is already in M, than v_2 replaces the old value of v_1.

4.1.3 Implementation Examples in Racket

In this section we show some execution examples of smart contracts, their vulnerabilities and how we can fix them with Flint-2's protection blocks.

4.1.3.1 BlockKing

In this section we recall once again the BlockKing contract introduced in Chapter 1. Figure 4.5 presents BlockKing's code in our Flint-2 implementation in Racket. We simplified the code, but the main idea of the Block King is kept. We removed the ifs, as it would lead to many execution steps, and then took all three functions of our original contract, and encapsulated them in protection blocks. Each function has its own block, which is guarded by its own typestate. We thought best to have three typestates, as state `canEnter` denotes that the contract has finished processing the last play and it is currently available

```
1  (contract BlockKing (waiting canEnter processing){
2      (var warrior : Address)
3      (var warriorGold : Int)
4      (var king : Address)}
5      (BlockKing :: (anycaller) {
6          (public init ((warrior : Address) (warriorGold : Int)) {
7              (self -> warrior = warrior)
8              (self -> warriorGold = warriorGold)
9              (self -> king = warrior)
10             (become canEnter)})})
11     (BlockKing @ (canEnter) :: (anycaller) {
12         (public func enter ((warrior : Address) (warriorGold : Int)) -> Void {
13             (self -> warrior = warrior)
14             (self -> warriorGold = warriorGold)
15             (become waiting)
16             return unit})})
17     (BlockKing @ (waiting) :: (anycaller) {
18         (public func __callback () -> Void {
19             (become processing)
20             (process_payment ())
21             return unit})})
22     (BlockKing @ (processing) :: (anycaller) {
23         (public func process_payment () -> Void {
24             (self -> king = (self -> warrior))
25             (become canEnter)
26             return unit})})
27 )
```

Figure 4.5: BlockKing Flint-2 code

for a new player; `waiting` which indicates that it sent a request to Oraclize and is now waiting for its new random number; and finally `processing`, which can only be accessed after receiving the random number. Moreover, a caller protection has been added to the function `__callback` to only allow calls from the Oraclize server.

By having these three typestates, the contract is bound to behave accordingly, as we established an execution stream that does not allow the contract to take any other bets, only when the current one is correctly processed.

Racket Execution. In this section we are going to present a simple execution example of the BlockKing contract. As explained previously, the configuration of Flint-2 is defined by a triple $\langle e, \beta, \sigma, CTS \rangle$. As Redex's language specification does not support concurrency, we trigger the response of the Oraclize service by simply calling the `__callback` function to mimic concurrency.

In Figure 4.6, through lines 1 to 8, we have the execution statements: lines 1 through 4 are the initialisation of both clients, the Oraclize service and the BlockKing contract, respectively; lines 5 and 6 reference the calls of the clients to enter the gamble, whereas 7 and 8 mimic the Oraclize service response. Lines 9 and 10 are the empty blockchain and call stack environments; and finally, line 11 refers to the typestate environment. This last one is a mapping of contract addresses to their respective typestate stack, being that the

last element of the stack is their current typestate.

We start by initialising the contracts, which applies the rule INIT, that adds the contracts to the blockchain and executes the state variable and constant statements and the statements inside the function `init`.

```
1  ((EOC -> init () -> 10 -> aBx)
2   (EOC -> init () -> 10 -> aBy)
3   (EOC -> init () -> 10 -> a0)
4   (BlockKing -> init ((warrior : aBK) (warriorGold : 0)) -> 9 -> aBK)
5   (aBK -> enter ((warrior : aBx) (warriorGold : 3)) -> sender (aBx))
6   (aBK -> enter ((warrior : aBy) (warriorGold : 3)) -> sender (aBy))
7   (aBK -> __callback () -> sender (a0))
8   (aBK -> __callback () -> sender (a0)))
9  ()
10 ()
11 ()
```

Figure 4.6: BlockKing Execution Example Pt.1

After successfully initialising the contracts, we reach the state presented in Figure 4.7. Only two more statements remain (lines 1 and 2), the blockchain has the state of our contract, the call stack is empty, and the current typestate is `canEnter`.

The following step is to evaluate the statement in line 1 from Figure 4.7, which is a client with the address aBx that wants to enter the game, with a bid of 3.

```
1  ((aBK -> enter ((warrior : aBx) (warriorGold : 3)) -> sender (aBx))
2   (aBK -> enter ((warrior : aBy) (warriorGold : 3)) -> sender (aBy))
3   (aBK -> __callback () -> sender (a0))
4   (aBK -> __callback () -> sender (a0)))
5  ((aBx -> (EOC 10) -> ->)
6   (aBy -> (EOC 10) -> ->)
7   (a0 -> (EOC 10) -> ->)
8   (aBK -> (BlockKing (warrior : aBK) (warriorGold : 0) (king : aBK) (Oraclize
        : a0) 9) -> ->))
9  ((aBK -> canEnter))
```

Figure 4.7: BlockKing Execution Example Pt.2

As we can see in Figure 4.8, the client successfully entered the game. The state of our contract is now updated in the blockchain, as the `warrior` is now aBx and `warriorGold` is 3. The current typestate has also changed to `waiting`, as the request to the Oraclize service has been made, and is waiting for a response.

Figure 3.44 is the last state of our execution, as the calling of function `enter` does not meet all requirements, because of our typestate protection.

Racket execution of this program allows us to prove that the BlockKing contract does not behave as expected, as it does not come to an end, as well as the typestate guards guarantee an ill-behaved execution.

Furthermore, to demonstrate that the caller groups prevent unsolicited calls from non-authorised contracts, we present in Figure 4.10 an example where a contract aV, which

```
1  (((((return (unit)))))
2   (aBK -> enter ((warrior : aBy) (warriorGold : 3)) -> sender (aBy))
3   (aBK -> __callback () -> sender (a0))
4   (aBK -> __callback () -> sender (a0)))
5  ((aBx -> (EOC 9) -> ->)
6   (aBy -> (EOC 10) -> ->)
7   (a0 -> (EOC 10) -> ->)
8   (aBK ->
9   (BlockKing (warrior : aBx) (warriorGold : 3) (king : aBK) (Oraclize : a0)
        10) -> -> (warrior : aBx) (warriorGold : 3)))
10 ((aBK -> canEnter waiting))
```

Figure 4.8: BlockKing Execution Example Pt.3

```
1  ((aBK -> enter ((warrior : aBy) (warriorGold : 3)) -> sender (aBy))
2   (aBK -> __callback () -> sender (a0))
3   (aBK -> __callback () -> sender (a0)))
4  ((aBx -> (EOC 9) -> ->)
5   (aBy -> (EOC 10) -> ->)
6   (a0 -> (EOC 10) -> ->)
7   (aBK -> (BlockKing (warrior : aBx) (warriorGold : 3) (king : aBK) (Oraclize
        : a0) 10) -> -> (warrior : aBx) (warriorGold : 3)))
8  ((aBK -> canEnter waiting))
```

Figure 4.9: BlockKing Execution Pt.5

is not assigned as `Oraclize`, calls the function `__callback`. We call special attention to the current typestate, which is the same as the one in the protection block of `__callback`. However, the Oraclize variable has a different address, and for that reason, if we run it on Racket, the program stops as there is no permitted next step.

```
1 (((aBK -> __callback () -> sender (aV)))
2  ((aBK -> (BlockKing
3             (warrior : aBK)
4             (warriorGold : 3)
5             (king : aBK)
6             (Oraclize : a0)
7           0) -> ->))
8  ()
9  ((aBK -> waiting)))
```

Figure 4.10: BlockKing Execution Example - Caller Groups

4.1.3.2 Concurrent Counter

This next example is a simple concurrent counter implemented in Solidity [36], as presented in Figure 4.11. Function `get` returns the current value of the variable `balance`, and function `set` updates the balance, sends back the previous amount and returns it.

If multiple contracts interact with the Counter at the same time, there is no guarantee that the value of `balance` will be *synchronised* through all interactions. This problem is

caused by the shared-memory in the blockchain, variable balance.

```
contract Counter {
  address public id;
  uint private balance;

  function get() returns (uint) {
    return balance;
  }

  function set() returns (uint) {
    uint t = balance;
    balance = msg.value;
    msg.sender.send(t);
    return t;
  }}
```

Figure 4.11: Solidity Concurrent Counter

As well as before, we made some modifications to the contract when implementing it to our version of Flint-2, as presented in Figure 4.12. As far as we know, there is no equivalent to Solidity's `msg.value`, so the value is passed as an input argument. Additionally, we did not understand how Flint-2 manages the gas of the contracts, thus we do not send money to the contract's callee. To solve the synchronisation problem, we added a simple locking mechanism to the contract, by using a typestate. Before modifying the variable balance, it enters the state cannotSet, which does not allow any other contract to re-write it, only when it reaches the state canSet.

The use of typestates prevents the modification of the variable by multiple contracts at the same time, as well as guarantying the atomicity of the procedure.

```
1  (contract Counter (canSet cannotSet){
2      (var balance : Int = 0) }
3      (Counter :: (anycaller) {
4          (public init () {
5              (become canSet)})
6          (public func get () -> Int {
7              return (self -> balance)})}
8      )
9      (Counter @ (canSet) :: (anycaller) {
10         (public func set ((value : Int)) -> Int {
11             (become cannotSet)
12             (var t : Int = (self -> balance))
13             (self -> balance = value)
14             (become canSet)
15             return t})}
16     )
17 )
```

Figure 4.12: Flint-2 code of the Solidity Counter

Racket Execution Similar to the example presented in Section 4.1.3.1, in Figure 4.13 we have an example of execution of the contract Counter, which is comprised of the contract

initialisation and function call of set and get. Just like before, our environments start empty.

```
(((Counter -> init () -> aCounter)
  (aCounter -> set ((value : 3)) -> sender (aCounter))
  (aCounter -> get () -> sender (aCounter)))
 ()
 ()
 ()
 ((contract Counter (canSet cannotSet)
   ((var balance : Int = 0))
   (Counter :: (anycaller) (
        (public init () (
            (become canSet)))
        (public func get () -> Int (
            return (self -> balance)))))
   (Counter @ (canSet) :: (anycaller) (
        (public func set ((value : Int)) -> Int (
            (become cannotSet)
            (var t : Int = (self -> balance))
            (self -> balance = value)
            (become canSet)
            return t)))))))
```

Figure 4.13: Counter Execution Example Pt.1

After the first statements is evaluated, we can see in Figure 4.14 that the contract was deployed to the blockchain, and that our current typestate is canSet.

```
(((aCounter -> set ((value : 3)) -> sender (aCounter))
  (aCounter -> get () -> sender (aCounter)))
 ((aCounter -> (Counter
                 (balance : 0)
                0) -> ->))
 ()
 ((aCounter -> canSet))
 ((contract Counter (canSet cannotSet)
   ((var balance : Int = 0))
   (Counter :: (anycaller) (
        (public init () (
            (become canSet)))
        (public func get () -> Int (
            return (self -> balance)))))
   (Counter @ (canSet) :: (anycaller) (
        (public func set ((value : Int)) -> Int (
            (become cannotSet)
            (var t : Int = (self -> balance))
            (self -> balance = value)
            (become canSet)
            return t)))))))
```

Figure 4.14: Counter Execution Example Pt.2

Figure 4.15, represents the state right after the CallSender rule was applied to retrieve the body of function set. We can observe that by evaluating these next statements,

the current typestate will change twice, as it locks and unlocks the ability to call the function set.

```
1  ((((become cannotSet)
2     ((var t : Int = (aCounter -> balance))
3     ((aCounter -> balance = 3)
4     ((become canSet)
5     (return (t))))))
6    (aCounter -> get () -> sender (aCounter)))
7   ((aCounter -> (Counter
8                  (balance : 0)
9                 0) -> -> (value : 3)))
10  (aCounter)
11  ((aCounter -> canSet))
12  ((contract Counter (canSet cannotSet)
13    ((var balance : Int = 0))
14    (Counter :: (anycaller) (
15         (public init () (
16             (become canSet)))
17         (public func get () -> Int (
18             return (self -> balance)))))
19    (Counter @ (canSet) :: (anycaller) (
20         (public func set ((value : Int)) -> Int (
21             (become cannotSet)
22             (var t : Int = (self -> balance))
23             (self -> balance = value)
24             (become canSet)
25             return t)))))))
```

Figure 4.15: Counter Execution Example Pt.3

The change in the typestate stack is visible in Figure 4.16, as well as the contracts balance was updated to 3. The evaluation continues and successfully ends by returning the value 3.

As Racket's step-by-step evaluation of statements is atomic, we are not able to show the concurrency this example needs. Instead, we now present a similar example where we try to call function set and the current typestate is cannotEnter, to prove that the use of this typestate will prevent the modification of the variable by multiple contracts at the same time.

When trying to evaluate the code presented in Figure 4.17, Racket cannot apply any rule, as the function set can only be called if the current typestate is canEnter, which in this example it is not.

4.1.3.3 Auction

In this example we have an Auction, in which Clients can bid to win, and if they lose or their bid is no longer the highest one, they can withdraw.

We modelled this example after Sylvain Conchon, Alexandrina Komeva and Fatiha Zaidi example of the automaton [12], as presented in Figure 4.18. It describes a simple contract of a bidding auction, where there is an Owner who sets up the auction and defines

```
(((aCounter -> get () -> sender (aCounter)))
 ((aCounter -> (Counter
                (balance : 3)
               0) -> -> (value : 3) (t : 0)))
 ()
 ((aCounter -> canSet cannotSet canSet))
 ((contract Counter (canSet cannotSet)
   ((var balance : Int = 0))
   (Counter :: (anycaller) (
        (public init () (
            (become canSet)))
        (public func get () -> Int (
            return (self -> balance)))))
   (Counter @ (canSet) :: (anycaller) (
        (public func set ((value : Int)) -> Int (
            (become cannotSet)
            (var t : Int = (self -> balance))
            (self -> balance = value)
            (become canSet)
            return t)))))))
```

Figure 4.16: Counter Execution Example Pt.4

```
((aCounter -> set ((value : 3)) -> sender (aCounter))
 ((aCounter -> (Counter
                (balance : 0)
               0) -> ->))
 ()
 ((aCounter -> cannotSet))
 ((contract Counter (canSet cannotSet)
   ((var balance : Int = 0))
   (Counter :: (anycaller) (
        (public init () (
            (become canSet)))
        (public func get () -> Int (
            return (self -> balance)))))
   (Counter @ (canSet) :: (anycaller) (
        (public func set ((value : Int)) -> Int (
            (become cannotSet)
            (var t : Int = (self -> balance))
            (self -> balance = value)
            (become canSet)
            return t)))))))
```

Figure 4.17: Counter Execution Example - Typestate

when it ends, as well as multiple clients that interact with the contract in order to win the auction. These clients can bid as long as the auction is running. Only the client who made the highest bid will win, and the others will be refunded their bids.

The auction has five shared state variables: `HBidder`, which denotes the highest bidder; `HBid`, the highest bid; $\texttt{PR}_i$, pending returns of the i^{th} client; `Ended`, represents if the auction is over; and finally `Owner`, which is the owner of the auction in progress. Additionally, the actions of the auction are represented in red in Figure 4.18. These actions, or entrypoints, are the following: `bid`, which has `v` as a parameter that represents the

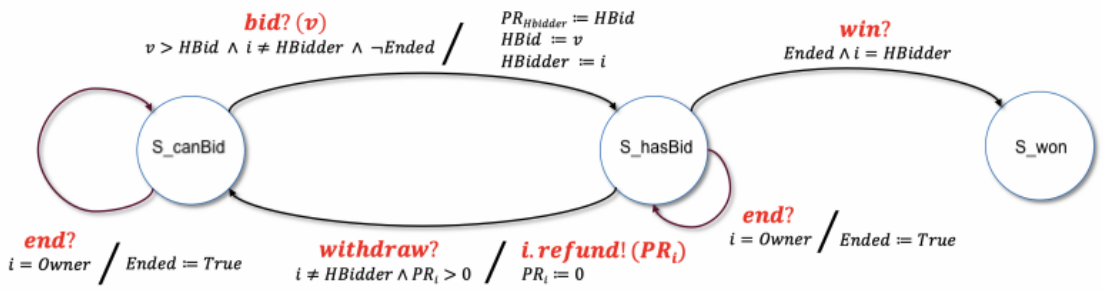

Figure 4.18: Example of Auction by Sylvain Conchon, Alexandrina Komeva and Fatiha Zaidi

amount of money being bid; `withdraw`; `win` and end.

There are three states, S_canBid; S_hasBid; and S_won. The first one, can be accessed by clients that have no pending returns, i.e., or have not bid yet or have already withdrawn their money. From this state, the client can `bid` if and only if the following conditions are all true: `v` is greater than the highest bid; the client bidding is not the highest bidder already; and the auction is still running. If so, the state variables `HBidder` and `HBid` will be updated with the client's index and the amount that was bid, and also the previous highest bid will be added to the pending returns of the preceding highest bidder.

The following state, S_hasBid, can be accessed only by clients who had bid beforehand. The client stays in this state until the auction ends, or it is no longer the highest bidder and has pending returns which, at this point, can `withdraw` the money, which will be `refunded` by the contract, and go back to state S_canBid. If the auction ends, and it is the highest bidder, then it can execute `win` and reclaim its prize by accessing the state S_won. If not, it will have to `withdraw`.

It is important to note that the owner of the auction can at any point end it by setting the variable Ended to True.

In Figure 4.19, we show a snippet of the code in Ligo [4], a smart contract language for Tezos, of the Auction presented in 4.18.

The three functions represent the ones in the automaton, and the typestates and pre-conditions presented in the example are checked in the if statements.

The code is verbose, and its prone to errors, as some pre-conditions can be left out by mistake.

Our approach with Flint-2 is much more compact and simple, as we use the typestates to prevent some undesired behaviours.

We made some adjustments to the contract when adapting it to Flint-2, but the behaviour of both is equivalent. Firstly, we divided the auction into two, Auction and Client. This separation between the auction and its clients is important as their behaviours are distinct, and so are their typestates.

[4] https://ligolang.org

```
52 let bid (bidding, storage : bidding * storage) : return =
53   let no_op : operation list = [] in
54   match Map.find_opt bidding.address storage.store with
55     Some (b) -> if b > 0 then
56     (failwith ("Already bid, need to withdraw first") : return) else
57       if bidding.address = 0 then
58         (failwith ("Owner cannot bid") : return)
59       else if storage.auctionEnded = 1 then
60         (failwith ("Auction is over") : return)
61       else if bidding.address = storage.highestBidder then
62         (failwith ("Client is already the highest bidder") : return)
63       else if bidding.amount <= 0 then
64         (failwith ("Value bid needs to be greater than 0") : return)
65       else
66         let s: storage = {storage with store = Map.update bidding.address (
               Some bidding.amount) storage.store} in
67         (([] : operation list), s)
68   | None -> (failwith ("Error"): return)
69
70 let withdraw (a, storage : act * storage) : return =
71   let no_op : operation list = [] in
72   match Map.find_opt a.address storage.store with
73     Some (b) -> if b > 0 then
74     (failwith ("Client has no pending returns") : return) else
75       if a.address = 0 then
76         (failwith ("Owner cannot withdraw") : return)
77       else if a.address = storage.highestBidder then
78         (failwith ("Client is the highest bidder, cannot withdraw") : return)
79       else if b > 0 then
80         (failwith ("Client has no pending returns") : return)
81       else
82         let s : store = Map.update a.address (Some 0) storage.store in
83         (no_op, {storage with store = s})
84   | None -> (failwith ("Error"): return)
85
86 let win (a, storage : act * storage) : return =
87   let no_op : operation list = [] in
88   match Map.find_opt a.address storage.store with
89     Some (b) -> if b = 0 then
90     (failwith ("Prize was already collected") : return) else
91       if a.address = 0 then
92         (failwith ("Owner cannot win") : return)
93       else if storage.auctionEnded = 0 then
94         (failwith ("Auction is still running") : return)
95       else if a.address <> storage.highestBidder then
96         (failwith ("Isnt the winner of the auction") : return)
97       else
98         let s : store = Map.update a.address (Some 0) storage.store in
99         (no_op, {storage with store = s})
100  | None -> (failwith ("Error"): return)
```

Figure 4.19: Auction Ligo code

Firstly, the Auction contract keeps the information about the owner and the all the bids, while the Client only stores its own address and bid.

Additionally, the typestate of the auction is `auctionEnded`, if the auction has ended, or `auctionRunning` if it has not. As it can be seen in Figure 4.20, there are many functions that can be called regardless of the state of the auction. When the auction is running, `bid` and `endAuction` can be called, although only the latter can only be called by the `owner` of the auction. Furthermore, function `win()` is only available whenever the current state is `aunctionEnded`.

In Figure 4.21, we present the automaton that models the behaviour of this Auction. Note that the functions presented in the constructor block are omitted, as they can be called whenever. The automata presented here was created by using Typestate Editor [5].

On the other hand, the Client's are greater in number, so inevitably the automaton is slightly more elaborate, as is presented in Figure 4.22. The typestates are `canBid`; `hasBid`; `won` (just like the ones in Figure 4.18), in addition to `lost`; `canWithdraw`; and `ended`.

When the contract is initiated, it begins as `canBid`. From it, the client can only bid, and if has the highest bid, then it changes to `hasBid`, otherwise it goes to `canWithdraw`.

From `canWithdraw`, the client withdraws its previous bid, and checks if the auction has ended. In case it is still running, it goes back to `canBid`, whereas if it has already ended, it goes to `lost`.

When in state `hasBid`, the client checks if it is still the highest bidder and if the auction is running. If it is the highest bidder and the auction has ended, the state changes to `won`, but if it is still going, then it remains in that state. If the client is no longer the highest bidder then it goes to `canWithdraw`.

We easily modelled the automaton in Figure 4.22 into a Flint-2 contract, which is presented in Figure 4.23. The state `S_checkEnded` is omitted as we can simply check if the auction is running in an if statement.

4.1.3.4 Racket Execution

In this section we present and discuss the execution of the auction shown in the previous section. This example consists in an auction `aAuction`, and two clients, `aC1` and `aC2`, who interact with the auction to win the bidding. This example starts with the initialisation of the auction, which owner is `a0wner`, and the clients, as presented in Figure 4.24, as well as empty environments. Apropos of this example, we omit the contracts declaration as it would be too extensive and repetitive.

In Figure 4.25, we can observe that the blockchain environment now has the auction and both the clients. The auction store is empty, it has no highest bidder or bid, and has not ended. And the clients' bids are also set to zero. The typestate environment informs us that the auction is running, and both the clients can proceed to bid. What follows is client `aC1` bids with the value 8.

[5] https://typestate-editor.github.io

```
(contract Auction (auctionEnded auctionRunning) {
    (var store : (Address : Int))
    (var owner : Address)
    (var ended : Bool)
    (var highestBid : Int)
    (var highestBidder : Address)}
    (Auction :: (anycaller) {
        (public init ((owner : Address)) {
            (self -> owner = owner)
            (self -> ended = false)
            (self -> highestBid = 0)
            (self -> highestBidder = aNull)
            (self -> store = ())
            (become auctionRunning)})
        (public func getHBidder () -> Address {
            return (self -> highestBidder)})
        (public func getHBid () -> Int {
            return (self -> highestBid)})
        (public func hasEnded () -> Bool {
            return (self -> ended)})
        (public func withdraw ((bidder : Address)) -> Bool {
            return (
                if (((bidder != (self -> highestBidder)) && (((self -> store)[
                    bidder ::]) != 0)))
                    {((self -> store = ((self -> store)[bidder :: 0])) (true))
                        }
                else {false})})) })
    (Auction @ (auctionRunning) :: (owner) {
        (public func endAuction () -> Void {
        (self -> ended = true)
        (become auctionEnded)
        return unit})} )
    (Auction @ (auctionRunning) :: (anycaller) {
        (public func bid ((bid : Int) (bidder : Address)) -> Address {
            (if ((((self -> store)[bidder ::]) == 0))
                {((self -> store = ((self -> store)[bidder :: bid]))
                (if ((bid > (self -> highestBid)))
                    {((self -> highestBid = bid) (self -> highestBidder =
                        bidder))}
                else {unit}))}
            else {unit})
            return (self -> highestBidder)}) })
    (Auction @ (auctionEnded) :: (anycaller) {
        (public func win ((bidder : Address)) -> Bool {
            return
                (if (((bidder == (self -> highestBidder)) && (((self -> store)
                    [bidder ::]) != 0)))
                    {((self -> store = ((self -> store)[bidder :: 0])) (true))
                        }
                else {false})})})))
```

Figure 4.20: Auction Code Flint-2

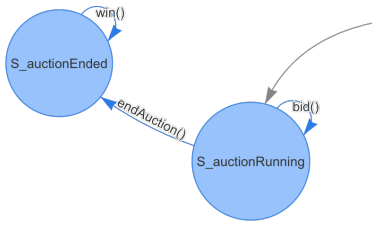

Figure 4.21: Automaton Auction

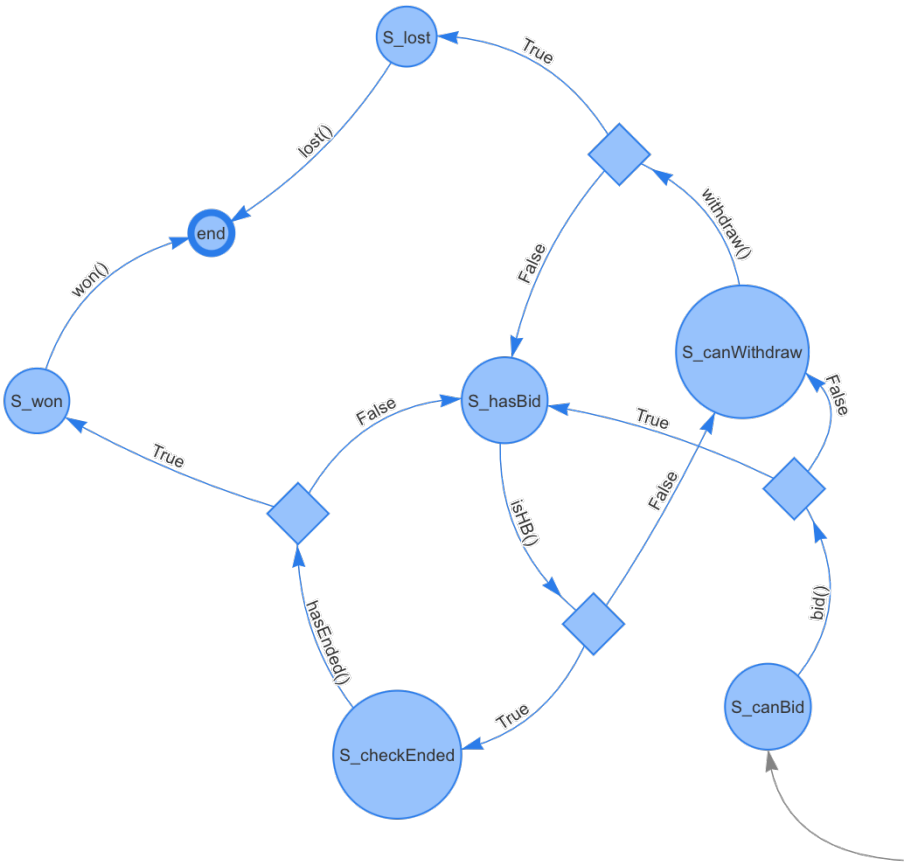

Figure 4.22: Automaton Client

```
(contract Client (canBid canWithdraw won lost hasBid ended) {
    (var auction : Address)
    (var bid : Int = 0)}
    (Client :: (anycaller) {
        (public init ((auction : Address)) {
            (self -> auction = auction)
            (become canBid)})
        (public func hasEnded () -> Bool {
            return ((self -> auction) -> ended)})})
    (Client @ (canBid) :: (anycaller) {
        (public func bid ((bid : Int)) -> Int {
            (self -> bid = bid)
            (if ((((self -> auction) -> bid ((bid : bid) (bidder : self))) ==
                self))
                {(become hasBid)}
            else {(become canWithdraw)})
            return bid})})
    (Client @ (canWithdraw) :: (anycaller) {
        (public func withdraw () -> Void {
            ((self -> auction) -> withdraw ((bidder : self)))
            (if (((self -> auction) -> hasEnded ()))
                {(become lost)}
            else {(become canBid)})
            return unit})})
    (Client @ (hasBid) :: (anycaller) {
        (public func isHB () -> Bool {
            return (if ((((self -> auction) -> getHBidder ()) == self))
                {(if (((self -> auction) -> hasEnded ()))
                    {((become won) (true))}
                else {true})}
            else {((become canWithdraw) (false))})})})
    (Client @ (won) :: (anycaller) {
        (public func won () -> Bool {
            return (if (((self -> auction) -> win ((bidder : self))))
                {(become ended)}
            else {false})})})))
```

Figure 4.23: Client Code Flint-2

```
((Auction -> init ((owner : aOwner)) -> aAuction)
  (Client -> init ((auction : aAuction)) -> aC1)
  (Client -> init ((auction : aAuction)) -> aC2)
  (aC1 -> bid ((bid : 8)) -> sender (aC1))
  (aC2 -> bid ((bid : 8)) -> sender (aC2))
  (aC2 -> withdraw () -> sender (aC2))
  (aC2 -> bid ((bid : 10)) -> sender (aC2))
  (aAuction -> endAuction () -> sender (aOwner))
  (aC1 -> isHB () -> sender (aC1))
  (aC1 -> withdraw () -> sender (aC1))
  (aC2 -> isHB () -> sender (aC2))
  (aC2 -> won () -> sender (aC2)))
()
()
()
```

Figure 4.24: Auction Example Pt.1

```
((aC1 -> bid ((bid : 8)) -> sender (aC1))
 (aC2 -> bid ((bid : 8)) -> sender (aC2))
 (aC2 -> withdraw () -> sender (aC2))
 (aC2 -> bid ((bid : 10)) -> sender (aC2))
 (aAuction -> endAuction () -> sender (aOwner))
 (aC1 -> isHB () -> sender (aC1))
 (aC1 -> withdraw () -> sender (aC1))
 (aC2 -> isHB () -> sender (aC2))
 (aC2 -> won () -> sender (aC2)))
((aAuction -> (Auction
                (store : ())
                (owner : aOwner)
                (ended : false)
                (highestBid : 0)
                (highestBidder : aNull)
               0) -> ->)
  (aC1 -> (Client
                (auction : aAuction)
                (bid : 0)
               0) -> ->)
  (aC2 -> (Client
                (auction : aAuction)
                (bid : 0)
               0) -> ->))
()
((aAuction -> auctionRunning)
 (aC1 -> canBid)
 (aC2 -> canBid))
```

Figure 4.25: Auction Example Pt.2

The auction has in its `store` the bid, and also, by being the first bid and greater than 0, the new highest bid is now 8, and belongs to `aC1`. Additionally, the typestate of `aC1` has now changed to `hasBid`; and `aC2` is ready to its won bidding with the same value.

As the bid of `aC2` was not greater than the highest bid, `aC2`'s typestate indicates that the client has to withdraw their previous bid, as in Figure 4.27. As a result, the next step on our evaluation is `aC2` withdrawing its bid.

Figure 4.28 represents the state of the blockchain after `aC2` withdraw. At this point, the typestate of `aC2` is back to `canBid`, and is ready to bid again, now with a bid of value 10.

As it can be seen in Figure 4.29, the auction's highest bid and bidder has changed to 10 and `aC2`, respectively. Additionally, the current typestate of client `aC2` is now `hasBid`; and the owner is about to end the auction.

After `aOwner` ends the auction, the typestate of the auction changes to `auctionEnded`, as in Figure 4.30, which in turn means that the clients can no longer bid.

As `aC1` asked if it was still the highest bidder and receiving that was no longer true, via calling `isHB()`, its typestate changed to `canWithdraw` as it has to be refund its previous bid.

After client `aC1` withdraws, the value stored in the blockchain for `aC1` bid is reset to

```
((aC2 -> bid ((bid : 8)) -> sender (aC2))
  (aC2 -> withdraw () -> sender (aC2))
  (aC2 -> bid ((bid : 10)) -> sender (aC2))
  (aAuction -> endAuction () -> sender (aOwner))
  (aC1 -> isHB () -> sender (aC1))
  (aC1 -> withdraw () -> sender (aC1))
  (aC2 -> isHB () -> sender (aC2))
  (aC2 -> won () -> sender (aC2)))
((aAuction -> (Auction
                (store : ((aC1 : 8)))
                (owner : aOwner)
                (ended : false)
                (highestBid : 8)
                (highestBidder : aC1)
               0) -> -> (bid : 8) (bidder : aC1))
  (aC1 -> (Client
                (auction : aAuction)
                (bid : 8)
               0) -> -> (bid : 8))
  (aC2 -> (Client
                (auction : aAuction)
                (bid : 0)
               0) -> ->))
()
((aAuction -> auctionRunning)
  (aC1 -> canBid hasBid)
  (aC2 -> canBid))
```

Figure 4.26: Auction Example Pt.3

zero, and its typestate is set to `lost`, as seen in Figure 4.32, which means that the client can no longer execute any calls other than `hasEnded`.

As client aC2 got the confirmation that it is the winner of the auction, its typestate changed to won, as shown in Figure 4.33. The only transaction remaining is for aC2 to claim its prize.

By evaluating the last expression, our execution comes to a halt as we reach terminal value `unit`. We can examine Figure 4.34 and see that the auction has no bids left to refund, our call stack is empty, and that client aC2's typestate is set to ended.

4.1.3.5 Traffic Light

This example was based on a traffic light case study in the Flint-2's Git repository [6]. The original code is presented in Figure 4.35, and the colour changing functions of the light are guarded with typestates. The light can only turn red or green if the current state is amber, and only turn amber if the light is red or green.

In Figure 4.36, we present the Racket implementation of this code. They are very similar, apart from the only missing statement, the `mutates` expression, which we have not implemented for simplicity purposes.

[6] https://github.com/flintlang/flint-2/blob/master/tests/behaviour_tests/traffic_lights.flint

```
((aC2 -> withdraw () -> sender (aC2))
  (aC2 -> bid ((bid : 10)) -> sender (aC2))
  (aAuction -> endAuction () -> sender (aOwner))
  (aC1 -> isHB () -> sender (aC1))
  (aC1 -> withdraw () -> sender (aC1))
  (aC2 -> isHB () -> sender (aC2))
  (aC2 -> won () -> sender (aC2)))
((aAuction -> (Auction
                (store : ((aC1 : 8)
                          (aC2 : 8)))
                (owner : aOwner)
                (ended : false)
                (highestBid : 8)
                (highestBidder : aC1)
               0) -> -> (bid : 8) (bidder : aC2))
  (bid : 8)
  (bidder : aC1)
  (aC -> (Client
                (auction : aAuction)
                (bid : 8)
               0) -> -> (bid : 8))
  (aC2 -> (Client
                (auction : aAuction)
                (bid : 8)
               0) -> -> (bid : 8)))
()
((aAuction -> auctionRunning)
  (aC1 -> canBid hasBid)
  (aC2 -> canBid canWithdraw))
```

Figure 4.27: Auction Example Pt.4

Now that we have introduced the Racket code, we are ready to show the step-by-step evaluation of a program that changes the light's colour. As it can be seen in Figure 4.37, all environments are initially empty before the contract initialisation.

Once the contract has been deployed, its typestate is set to Red, as represented in Figure 4.38. The next step on the program's evaluation is its change to the colour amber (line 1), which is permitted due to the current typestate being Red.

Figure 4.39 represents the program state after the light has turn amber. Line 5 shows that the current typestate is Amber, as the traffic light's following function call is to turn green, and then amber again.

At the end of the execution, which is represented in Figure 4.40, line 4 describes the evolution of the traffic light's states, and we can assume that the program behaved correctly as there was no changes from Red to Green, or vice-versa, without passing through Amber.

4.2 Type System

In this section we defined a type system for Flint-2, an original presentation based on that of Featherweight Solidity [14].

```
((aC2 -> bid ((bid : 10)) -> sender (aC2))
  (aAuction -> endAuction () -> sender (aOwner))
  (aC1 -> isHB () -> sender (aC1))
  (aC1 -> withdraw () -> sender (aC1))
  (aC2 -> isHB () -> sender (aC2))
  (aC2 -> won () -> sender (aC2)))
((aAuction -> (Auction
                (store : ((aC1 : 8) (aC2 : 0)))
                (owner : aOwner)
                (ended : false)
                (highestBid : 8)
                (highestBidder : aC1)
              0) -> -> (bid : 8) (bidder : aC2))
  (bidder : aC2)
  (bid : 8)
  (bidder : aC1)
  (aC1 -> (Client
                (auction : aAuction)
                (bid : 8)
              0) -> -> (bid : 8))
  (aC2 -> (Client
                (auction : aAuction)
                (bid : 8)
              0) -> -> (bid : 8)))
()
((aAuction -> auctionRunning)
 (aC1 -> canBid hasBid)
 (aC2 -> canBid canWithdraw canBid))
```

Figure 4.28: Auction Example Pt.5

Before proceeding, we recall the types of our syntax, as defined in Figure 4.2:

$$(\textit{Types}) \quad t \quad ::= \text{Int} \mid \text{Address} \mid \text{Bool} \mid \text{Void} \mid (t : t)$$

A type system of a given language is usually an inductively defined relation, via rules following the syntax of the language. A rule is represented as a number of premise judgements above a horizontal line, and below it a conclusion judgement [10]. A rule is valid if all the premises hold. These judgements define the relation between types, terms and environments. Thus, we define the judgements of this type system as the following relation:

$$\Gamma \vdash e : t \triangleright \Gamma'$$

where:

- Γ and Γ' are the context input and output respectively;
- and $e : t$ means that the expression e has type t in Γ.

We define a context Γ as a function assigning types to variables and the type Address to addresses.

```
((aAuction -> endAuction () -> sender (aOwner))
  (aC1 -> isHB () -> sender (aC1))
  (aC1 -> withdraw () -> sender (aC1))
  (aC2 -> isHB () -> sender (aC2))
  (aC2 -> won () -> sender (aC2)))
((aAuction -> (Auction
                (store : ((aC1 : 8) (aC2 : 10)))
                (owner : aOwner)
                (ended : false)
                (highestBid : 10)
                (highestBidder : aC2)
               0) -> -> (bid : 10) (bidder : aC2))
  (bid : 8)
  (bidder : aC2)
  (bidder : aC2)
  (bid : 8)
  (bidder : aC1)
  (aC1 -> (Client
                (auction : aAuction)
                (bid : 8)
              0) -> -> (bid : 8))
  (aC2 -> (Client
                (auction : aAuction)
                (bid : 10)
              0) -> -> (bid : 10))
  (bid : 8))
()
((aAuction -> auctionRunning)
 (aC1 -> canBid hasBid)
 (aC2 -> canBid canWithdraw canBid hasBid))
```

Figure 4.29: Auction Example Pt.6

$$(\textit{Type environment}) \quad \Gamma \quad ::= \quad \emptyset \mid \Gamma, x : t \mid \Gamma, a : \text{Address}$$

In the following sections, we will present the typing rules that define the Flint-2 type system.

4.2.1 Typing Rules

In this section we present the typing rules we formalised for Flint-2.

Axioms. These rules all have the expected meaning.

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool} \triangleright \Gamma}\ \text{T-True} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool} \triangleright \Gamma}\ \text{T-False} \qquad \frac{}{\Gamma \vdash \texttt{unit} : \texttt{Void} \triangleright \Gamma}\ \text{T-Unit}$$

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n\colon \texttt{Int} \triangleright \Gamma}\ \text{T-Int} \qquad \frac{}{\Gamma, a\colon \texttt{Address} \vdash a\colon \texttt{Address} \triangleright \Gamma}\ \text{T-Address}$$

$$\frac{}{\Gamma, x : t \vdash x : t \triangleright \Gamma}\ \text{T-Var} \qquad \frac{}{\Gamma, a : C \vdash a : C \triangleright \Gamma}\ \text{T-Ref}$$

```
((aC1 -> isHB () -> sender (aC1))
  (aC1 -> withdraw () -> sender (aC1))
  (aC2 -> isHB () -> sender (aC2))
  (aC2 -> won () -> sender (aC2)))
((aAuction -> (Auction
                (store : ((aC1 : 8) (aC2 : 10)))
                (owner : aOwner)
                (ended : true)
                (highestBid : 10)
                (highestBidder : aC2)
               0) -> -> (bid : 10) (bidder : aC2))
  (bid : 8)
  (bidder : aC2)
  (bidder : aC2)
  (bid : 8)
  (bidder : aC1)
  (aC1 -> (Client
            (auction : aAuction)
            (bid : 8)
           0) -> -> (bid : 8))
  (aC2 -> (Client
            (auction : aAuction)
            (bid : 10)
           0) -> -> (bid : 10))
  (bid : 8))
()
((aAuction -> auctionRunning auctionEnded)
 (aC1 -> canBid hasBid)
 (aC2 -> canBid canWithdraw canBid hasBid))
```

Figure 4.30: Auction Example Pt.7

Standard Rules. The rules include the return expression, sequential composition, variable declaration and assignment, and the if expression. These rules have the standard meaning.

$$\frac{\Gamma \vdash e : t \triangleright \Gamma}{\Gamma \vdash \texttt{return}\ e : t \triangleright \Gamma}\ \text{T-Return} \qquad \frac{\Gamma \vdash e_1 : t_1 \triangleright \Gamma' \quad \Gamma' \vdash e_2 : t_2 \triangleright \Gamma''}{\Gamma \vdash e_1 ; e_2 : t_2 \triangleright \Gamma''}\ \text{T-Seq}$$

$$\frac{\Gamma, x : t \vdash e : t \triangleright \Gamma'}{\Gamma \vdash \texttt{var}\ x : t = e : t \triangleright \Gamma'}\ \text{T-DeclVar} \qquad \frac{}{\Gamma \vdash \texttt{var}\ x : t : t \triangleright \Gamma, x : t}\ \text{T-DeclVar2}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \triangleright \Gamma \quad \Gamma \vdash e_2 : t \triangleright \Gamma \quad \Gamma \vdash e_3 : t \triangleright \Gamma}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t \triangleright \Gamma}\ \text{T-If}$$

$$\frac{\Gamma, x : t \vdash e : t \triangleright \Gamma'}{\Gamma \vdash \texttt{let}\ x : t = e : t \triangleright \Gamma'}\ \text{T-DeclCons}$$

$$\frac{}{\Gamma \vdash \texttt{let}\ x : t : t \triangleright \Gamma, x : t}\ \text{T-DeclCons2} \qquad \frac{\Gamma \vdash x : t \quad \Gamma \vdash e : t}{\Gamma \vdash x = e : t \triangleright \Gamma}\ \text{T-Ass}$$

```
((aC1 -> withdraw () -> sender (aC1))
  (aC2 -> isHB () -> sender (aC2))
  (aC2 -> won () -> sender (aC2)))
 ((aAuction
   ->
   (Auction
    (store : ((aC1 : 8) (aC2 : 10)))
    (owner : aOwner)
    (ended : true)
    (highestBid : 10)
    (highestBidder : aC2)
    0)
   ->
   ->
   (bid : 10)
   (bidder : aC2))
  (bid : 8)
  (bidder : aC2)
  (bidder : aC2)
  (bid : 8)
  (bidder : aC1)
  (aC1
   ->
   (Client (auction : aAuction) (bid : 8) 0)
   ->
   ->
   (bid : 8))
  (aC2
   ->
   (Client (auction : aAuction) (bid : 10) 0)
   ->
   ->
   (bid : 10))
  (bid : 8))
 ()
 ((aAuction -> auctionRunning auctionEnded)
  (aC1 -> canBid hasBid canWithdraw)
  (aC2 -> canBid canWithdraw canBid hasBid))
```

Figure 4.31: Auction Example Pt.8

State variables. The rules listed below are for the access of a contract's state variables.

$$\frac{\Gamma \vdash e : C \triangleright \Gamma \quad \text{sv}(C) = (\widetilde{x : t}) \quad x \in \tilde{x}}{\Gamma \vdash e.x : t \triangleright \Gamma} \text{ T-StateSel} \qquad \frac{\Gamma \vdash e_1.x : t \triangleright \Gamma \quad \Gamma \vdash e_2 : t \triangleright \Gamma}{\Gamma \vdash e_1.x = e_2 : t \triangleright \Gamma} \text{ T-StateAss}$$

Mapping. The typing rules presented below are for typing the mapping values as well as the read and write of them. The type t for $\Gamma \vdash \{v_n | n \in \mathbb{N}_0\} : t$ is fixed for all values in v_n.

$$\frac{M = \widetilde{k : v} \quad \Gamma \vdash \tilde{k} : t_k \triangleright \Gamma \quad \Gamma \vdash \tilde{v} : t_v \triangleright \Gamma}{\Gamma \vdash M : (t_k : t_v) \triangleright \Gamma} \text{ T-Mapping}$$

$$\frac{\Gamma \vdash e_1 : (t_k : t_v) \triangleright \Gamma \quad \Gamma \vdash e_2 : t_k \triangleright \Gamma \quad \Gamma \vdash e_3 : t_v \triangleright \Gamma}{\Gamma \vdash e_1[e_2 : e_3] : (t_k : t_v) \triangleright \Gamma} \text{ T-MapAss}$$

```
((aC2 -> isHB () -> sender (aC2))
(aC2 -> won () -> sender (aC2)))
 ((aAuction -> (Auction
                (store : ((aC1 : 0) (aC2 : 10)))
                (owner : aOwner)
                (ended : true)
                (highestBid : 10)
                (highestBidder : aC2)
               0) -> -> (bid : 10) (bidder : aC1))
  (bidder : aC2)
  (bid : 8)
  (bidder : aC2)
  (bidder : aC2)
  (bid : 8)
  (bidder : aC1)
  (aC1 -> (Client
                (auction : aAuction)
                (bid : 8)
               0) -> -> (bid : 8))
  (aC2 -> (Client
                (auction : aAuction)
                (bid : 10)
               0) -> -> (bid : 10))
  (bid : 8))
()
((aAuction -> auctionRunning auctionEnded)
  (aC1 -> canBid hasBid canWithdraw lost)
  (aC2 -> canBid canWithdraw canBid hasBid))
```

Figure 4.32: Auction Example Pt.9

$$\frac{\Gamma \vdash e_1 : (t_k : t_v) \triangleright \Gamma \quad \Gamma \vdash e_2 : t_k \triangleright \Gamma}{\Gamma \vdash e_1[e_2] : t_v \triangleright \Gamma} \text{ T-MapSel}$$

Call functions. The following rules are for functions, as well as contract deployment to the blockchain. Function f must be defined in the code of contract C, as `fbody` returns both the type of the function and the return statement.

$$\frac{C \in \text{classes} \quad \text{sv}(C) = \widetilde{x : t} \quad \Gamma \vdash \widetilde{e : t} \triangleright \Gamma}{\Gamma \vdash C.\text{init}(\widetilde{x : e}).a : \texttt{Void} \triangleright \Gamma, a : \texttt{Address}} \text{ T-Init}$$

$$\frac{\Gamma \vdash e_0.f(\widetilde{x : e}) : t \quad \Gamma \vdash a : \texttt{Address} \triangleright \Gamma}{\Gamma \vdash e_0.f(\widetilde{x : e}).\texttt{sender}(a) : t \triangleright \Gamma} \text{ T-CallSender}$$

$$\frac{\Gamma \vdash e_0 : \texttt{Address} \quad \text{fbody}(C, f, x) = (e'' \cdot \texttt{return}\ e, \widetilde{cl}, \widetilde{ts}, t) \quad \Gamma \vdash \widetilde{e' : t'} \triangleright \Gamma}{\Gamma \vdash e_0.f(\widetilde{x : e'}) : t \triangleright \Gamma} \text{ T-Call}$$

$$\frac{\Gamma \vdash e_0 : \texttt{Address} \quad \text{fbodyinit}(C, f, x) = (e'' \cdot \texttt{return}\ e, \widetilde{cl}, t) \quad \Gamma \vdash \widetilde{e' : t'} \triangleright \Gamma}{\Gamma \vdash e_0.f(\widetilde{x : e'}) : t \triangleright \Gamma} \text{ T-Call-2}$$

```
((aC2 -> won () -> sender (aC2)))
((aAuction -> (Auction
                (store : ((aC1 : 0) (aC2 : 10)))
                (owner : aOwner)
                (ended : true)
                (highestBid : 10)
                (highestBidder : aC2)
               0) -> -> (bid : 10) (bidder : aC1))
  (bidder : aC2)
  (bid : 8)
  (bidder : aC2)
  (bidder : aC2)
  (bid : 8)
  (bidder : aC1)
  (aC1 -> (Client
                (auction : aAuction)
                (bid : 8)
               0) -> -> (bid : 8))
  (aC2 -> (Client
                (auction : aAuction)
                (bid : 10)
               0) -> -> (bid : 10))
  (bid : 8))
()
((aAuction -> auctionRunning auctionEnded)
 (aC1 -> canBid hasBid canWithdraw lost)
 (aC2 -> canBid canWithdraw canBid hasBid won))
```

Figure 4.33: Auction Example Pt.10

$$\frac{\Gamma \vdash e_0 : \texttt{Address} \triangleright \Gamma \quad \text{fbody}(C, f, x) = (e'' \cdot \texttt{return}\ e, \widetilde{cl}, \widetilde{ts}, t) \qquad \Gamma \vdash \widetilde{e' : t'} \triangleright \Gamma \quad \texttt{anystate} \in \widetilde{ts}}{\Gamma \vdash e_0.f(\widetilde{x : e'}) : t \triangleright \Gamma} \text{ T-Call-Any}$$

4.2.2 Example of Typing Derivations

In the following sections we will prove the well typification of the examples presented in Section 4.1.3.

4.2.2.1 BlockKing

In this section we try to prove that the code presented in Figure 4.6 is well-typed. Because of presentation purposes, we do omit some premises presented in the judgements, such as $C \in \mathit{classes}$ and $\text{sv}(C) = \widetilde{x : t}$, nevertheless our typechecker guarantees all of them. To that extent, the expression that we are going to evaluate, which is a segment from before, is:

$$e = \quad \texttt{BlockKing.init}((\texttt{warrior} : aBK);(\texttt{warriorGold} : 0))(aBK)$$
$$e' = \quad \texttt{aBK.enter}((\texttt{warrior} : aBx);(\texttt{warriorGold} : 3)).\texttt{sender}(aBx)$$

The initial context is $\Gamma = \emptyset, \mathit{Oraclize} : \texttt{Address}, aBx : \texttt{Address}$, and $\Gamma' = \Gamma, aBK : \texttt{Address}$. (1) represents another derivation tree, that is presented separately.

```
1  (unit
2   ((aAuction -> (Auction
3                  (store : ((aC1 : 0) (aC2 : 0)))
4                  (owner : aOwner)
5                  (ended : true)
6                  (highestBid : 10)
7                  (highestBidder : aC2)
8                 0) -> -> (bid : 10) (bidder : aC2))
9    (bidder : aC1)
10   (bidder : aC2)
11   (bid : 8)
12   (bidder : aC2)
13   (bidder : aC2)
14   (bid : 8)
15   (bidder : aC1)
16   (aC1 -> (Client
17                 (auction : aAuction)
18                 (bid : 8)
19                0) -> -> (bid : 8))
20   (aC2 -> (Client
21                 (auction : aAuction)
22                 (bid : 10)
23                0) -> -> (bid : 10))
24   (bid : 8))
25  ()
26  ((aAuction -> auctionRunning auctionEnded)
27   (aC1 -> canBid hasBid canWithdraw lost)
28   (aC2 -> canBid canWithdraw canBid hasBid won ended))
```

Figure 4.34: Auction Example Pt.11

$$\text{T-Seq}\dfrac{(3)\quad \text{T-Init}\dfrac{\dfrac{\checkmark}{\Gamma' \vdash aBK : \texttt{Address} \triangleright \Gamma'}\ \text{T-Address}\quad (1)\quad (2)}{\Gamma \vdash e : \texttt{Void} \triangleright \Gamma, aBK : \texttt{Address}}}{\Gamma \vdash e; e' : \texttt{Void} \triangleright \Gamma'}$$

Where (1) and (2) are, is where the evaluation of the parameters of the expression:

$$(1)\dfrac{\checkmark}{\Gamma' \vdash aBx : \texttt{Address} \triangleright \Gamma'}\ \text{T-Address}\quad (2)\dfrac{\checkmark}{\Gamma' \vdash 0 : \texttt{Int} \triangleright \Gamma'}\ \text{T-Int}$$

After the evaluation of the left branch, we proceed to the following:

(3)

$$\dfrac{\dfrac{\checkmark}{\Gamma' \vdash aBx : \texttt{Address} \triangleright \Gamma'}\ \text{T-Address}\quad (4)}{\Gamma' \vdash e' : \texttt{Void} \triangleright \Gamma'}\ \text{T-CallSender}$$

Judgement T-CallSender evaluates the address of the sender and the expression $e'' = aBK.\texttt{enter}(\texttt{(warrior : } aBx\texttt{); (warriorGold : 3))}$, which derivation tree is presented in (4):

(4)

$$\dfrac{\dfrac{\checkmark}{\Gamma' \vdash aBK : \texttt{Address} \triangleright \Gamma'}\ \text{T-Address}\quad (5)\quad (6)}{\Gamma' \vdash e'' : \texttt{Void} \triangleright \Gamma'}\ \text{T-Call}$$

```
1  contract TrafficLights (Red, Amber, Green) {
2      var signal: Int = 0
3  }
4
5  TrafficLights :: (any) {
6      public init() {
7          become Red
8      }
9
10     public func getSignal() -> Int {
11         return signal
12     }
13 }
14
15 TrafficLights @(Red, Green) :: (any) {
16     public func moveToAmber() mutates (signal) {
17         signal = 1
18         become Amber
19     }
20 }
21
22 TrafficLights @(Amber) :: (any) {
23     public func moveToGreen() mutates (signal) {
24         signal = 2
25         become Green
26     }
27
28     public func moveToRed() mutates (signal) {
29         signal = 0
30         become Red
31     }
32 }
```

Figure 4.35: Traffic Light contract

The evaluation of both input arguments of function `enter` is on (5) and (6).

$$(5)\frac{\checkmark}{\Gamma' \vdash aBx : \mathsf{Address} \triangleright \Gamma'}\ \textsc{T-Address} \qquad (6)\frac{\checkmark}{\Gamma' \vdash 3 : \mathsf{Int} \triangleright \Gamma'}\ \textsc{T-Int}$$

This exercise proves the well-typification of the code presented in Figure 4.6, whilst the example we demonstrate next shows that our type system rejects ill-formed expressions.

In this next example, instead of providing an address as input for `warrior`, we give an integer: $e''' = \mathtt{BlockKing.init((warrior : 3); (warriorGold : 0))}(aBK)$.

$$\frac{\dfrac{(7)\ \dfrac{}{\Gamma' \vdash 3 : \mathsf{Address} \triangleright \Gamma'}\ \textsc{T-Address} \quad (2)}{\Gamma \vdash e''' : \mathsf{Void} \triangleright \Gamma, aBK : \mathsf{Address}}\ \textsc{T-Init} \quad (3)}{\Gamma \vdash e'''; e' : \mathsf{Void} \triangleright \Gamma'}\ \textsc{T-Seq}$$

However, when we proceed to the evaluation of the branch of the `warrior` variable, because it expects an address from the premise $\mathrm{sv}(C) = \mathtt{warrior : Address, warriorGold : Int}$,

```
(contract TrafficLights (Red Amber Green) {
    (var signal : Int = 0)
}

(TrafficLights :: (anycaller) {
    (public init () {
        (become Red)
    })

    (public func getSignal () -> Int {
        return signal
    })
})

(TrafficLights @ (Red Green) :: (anycaller) {
    (public func moveToAmber () -> Void {
        (signal = 1)
        (become Amber)
        return unit
    })
})

(TrafficLights @ (Amber) :: (anycaller) {
    (public func moveToGreen () -> Void {
        (signal = 2)
        (become Green)
        return unit
    })

    (public func moveToRed () -> Void {
        (signal = 0)
        (become Red)
        return unit
    })
}))
```

Figure 4.36: Racket's implementation of Traffic Light contract

```
((TrafficLights -> init () -> 10 -> aTL)
 (aTL -> moveToAmber () -> sender (aTL))
 (aTL -> moveToGreen () -> sender (aTL))
 (aTL -> moveToAmber () -> sender (aTL)))
()
()
()
```

Figure 4.37: Initial State of Traffic Light

```
((aTL -> moveToAmber () -> sender (aTL))
 (aTL -> moveToGreen () -> sender (aTL))
 (aTL -> moveToAmber () -> sender (aTL)))
((aTL -> (TrafficLights (signal : 0) 10) -> ->))
((aTL -> (TrafficLights (signal : 0) 10) -> ->))
((aTL -> Red))
```

Figure 4.38: After contract deployment

```
((aTL -> moveToGreen () -> sender (aTL))
(aTL -> moveToAmber () -> sender (aTL)))
((aTL -> (TrafficLights (signal : 1) 10) -> ->))
((aTL -> (TrafficLights (signal : 1) 10) -> ->))
((aTL -> Red Amber))
```

Figure 4.39: Light is amber

```
unit
((aTL -> (TrafficLights (signal : 1) 10) -> ->))
((aTL -> (TrafficLights (signal : 2) 10) -> ->))
((aTL -> Red Amber Green Amber))
```

Figure 4.40: Final state

the typechecker throws an error message that says: "`type mismatch : expected type Address but got type Int`".

4.2.2.2 Auction

In this example we try to prove that the program is well-typed by using part of the code in Section 4.1.3.4. The expression we are evaluating is:

$$\begin{aligned} e := \quad & Client.\texttt{init}((auction : aAuction))(aC1); \\ & aC1.\texttt{bid}((bid : 8)) \end{aligned}$$

We start with context $\Gamma = aAuction : \texttt{Address}$, and $e' = aC1.\texttt{bid}((\texttt{bid} : 8))$. In the branch of T-Init, we omit the premises $Client \in classes$ and $\text{sv}(C) = auction : \text{Address}$, for visualisation purposes; nevertheless they are both ensured by our typechecker.

$$\dfrac{\dfrac{\dfrac{\checkmark}{\Gamma \vdash aAuction : \text{Address}}\ (\text{T-Address})}{\Gamma \vdash Client.\texttt{init}((auction : aAuction))(aC1) : \text{Void} \dashv \Gamma, aC1 : \text{Address}}\ (\text{T-Init}) \quad \dfrac{(1)}{\Gamma' \vdash e' : \text{Int}}}{\Gamma \vdash Client.\texttt{init}((auction : aAuction))(aC1); e' : \text{Int}}\ (\text{T-Se}$$

After the evaluation of the left branch, $\Gamma' = \Gamma, aC1$:Address. Where (1) is, the derivation is:

$$\dfrac{\dfrac{\checkmark}{\Gamma' \vdash aC1 : \text{Address}}\ (\text{T-Address}) \quad \dfrac{\checkmark}{\Gamma' \vdash 8 : \text{Int}}\ (\text{T-Int})}{\Gamma' \vdash aC1.\texttt{bid}((bid : 8)) : \text{Int}}\ (\text{T-Call})$$

We also omit the pre-condition $\text{fbody}(Client, bid, bid) = (e' \cdot \text{return } bid, \widetilde{cl}, \widetilde{ts}, \text{Int})$, which is also guaranteed by our typechecker. As we can see, this exercise proves that the program presented in Section 4.1.3.4 is well-typed, whereas the one we describe next demonstrates that our type system rejects ill-formed expressions.

In this next example, instead of providing an integer as input for function `bid`, we give a boolean: $e = Client.init((auction : aAuction))(aC1); aC1.bid((bid : true))$. The context starts as $\Gamma : aAuction : \texttt{Address}$.

$$\dfrac{(2)\quad \dfrac{(3)\quad \Gamma' \vdash \texttt{true} : \texttt{Int} \triangleright \Gamma'}{\Gamma' \vdash aC1.bid((bid : true)) : \texttt{Int} \triangleright \Gamma'}\ \text{T-Call}}{\Gamma \vdash e'; aC1.bid((bid : true)) : \texttt{Int} \triangleright \Gamma'}\ \text{(T-Seq)}$$

The derivation trees assigned as (5) and (6) are the same as the ones presented in the exercise above. As we reach the evaluation of the statement `true : Int`, the typechecker throws the following error message: "`type mismatch : expected type Int but got type Bool`".

We now present the derivation for an example with maps, to demonstrate the mapping typification rules.

The initial context is $\Gamma = aAuction : \texttt{Address}, aC1 : \texttt{Address}, aC2 : \texttt{Address}$, and expression we are evaluating is:

$$e = \quad aAuction.\texttt{store} = aAuction.\texttt{store}[aC1 : 0]$$

where $aAuction.\texttt{store} = [(aC1 : 8), (aC2 : 10)]$.

$$\dfrac{\dfrac{\dfrac{\checkmark}{\Gamma \vdash aAuction : \texttt{Client} \triangleright \Gamma}\ \text{T-Address}}{\Gamma \vdash aAuction.\texttt{store} : (\texttt{Client} : \texttt{Int}) \triangleright \Gamma}\ \text{T-StateSel} \quad (4)}{\Gamma \vdash aAuction.\texttt{store} = aAuction.\texttt{store}[aC1 : 0] : (\texttt{Client} : \texttt{Int}) \triangleright \Gamma}\ \text{T-StateAss}$$

$$(4)\ \dfrac{(5)\quad \dfrac{\checkmark}{\Gamma \vdash aC1 : \texttt{Client} \triangleright \Gamma}\ \text{T-Address} \qquad \dfrac{\checkmark}{\Gamma \vdash 0 : \texttt{Int} \triangleright \Gamma}\ \text{T-Int}}{\Gamma \vdash [(aC1 : 8), (aC2 : 10)][aC1 : 0] : (\texttt{Client} : \texttt{Int}) \triangleright \Gamma}\ \text{T-MapAss}$$

Derivation tree (5) evaluates all elements in $aAuction.\texttt{store}$.

$$(5)\ \dfrac{\dfrac{\checkmark}{\Gamma \vdash aC1 : \texttt{Client} \triangleright \Gamma}\ \text{T-Address} \qquad \dfrac{\checkmark}{\Gamma \vdash aC2 : \texttt{Client} \triangleright \Gamma}\ \text{T-Address} \qquad \dfrac{\checkmark}{\Gamma \vdash 8 : \texttt{Int} \triangleright \Gamma}\ \text{T-Int} \qquad \dfrac{\checkmark}{\Gamma \vdash 10 : \texttt{Int} \triangleright \Gamma}\ \text{T-Int}}{\Gamma \vdash [(aC1 : 8), (aC2 : 10)](\texttt{Client} : \texttt{Int}) \triangleright \Gamma}\ \text{T-Mapp}$$

4.3 Extension of the Type System with Usages

In this section, based on the work of Vasconcelos and Ravara [42] of MOOL language, a small object-oriented programming language that integrates behavioural types in the form of usages, we present an extension proposal to Flint-2, completed with syntax formalisation, operational semantics and type system.

4.3.1 Syntax

In this version of Flint-2 with usages, we keep most of the syntax presented in 4.2 intact, apart from the following items:

- In the contract declaration, we add a construct for the usage declaration u.
- There is no longer a distinction between constants and variables so, we simplified it and we only have variable declarations as well as there is no longer an environment for the constants in the blockchain.
- We removed the *constructor block*, and we only use the syntax of the *protection blocks*.
- The type contract is now defined as $C[u\ \widetilde{F}]$, which is detailed below.
- We removed the dictionary structure and the expressions associated with its lookups and modification, in order to simplify the new implementation.

In Figure 4.41, we present the revised syntax of our implementation of Flint-2 with Usages, and below we describe the new additions to our syntax.

(*Contract Declaration*)	CD	::=	`contract` $C\ (u)\ \{\widetilde{F}\}\ \widetilde{PB}$
(...)			
(*Types*)	t	::=	(...) $\mid C[u;\widetilde{F}]$
(*Field Declaration*)	F	::=	$(f\ t)$
(*Class Session Types*)	u	::=	$q\{\widetilde{m:u}\} \mid < u+u > \mid \mu X.u \mid !X$
(*Usage Qualifiers*)	q	::=	`lin` | un

Figure 4.41: Revised Syntax of the Racket Implementation of Flint-2 with Usages

Types. The new type of contract $C[u;\ \widetilde{F}]$ now contains information of contract C usage type u, as well as the state variables (or fields) and its respective types.

Field declaration. As previously mentioned, contains information over each state variable type.

Class session types. $q\{\widetilde{m:u}\}$ denotes the branch type, which specifies every method available $\widetilde{m}$ and their continuation $\widetilde{u}$. Variant types are defined over $< u+u >$, which is indexed by the values returned by the prior function called, `true` or `false`. Recursive types are denoted by $\mu X.u$, which are required to be contractive, i.e. the sub-expression $\mu X_1.\cdots\mu X_n.X_1$ cannot be contained [22].

Usage qualifiers. Usage branch types are annotated with an usage qualifier q for aliasing purposes. The qualifier `lin`, which stands for linear, specifies that the contract can only be used by only one client, unlike the qualifier un which allows for it to be shared among multiple clients. The latter stands for unrestricted or shared.

4.3.2 Type System

In this section, we will present the formalisation of the type system of Flint-2 with usages.

4.3.2.1 Judgements

In the implementation of the type system of Flint-2 with usages, we define a new judgement for usage types, in addition to the previously defined one. The two judgements are shown below:

$$\Theta;\Gamma \vdash_C u \triangleright \Gamma' \quad \Gamma \vdash e : t \triangleright \Gamma'$$

In the usage typing relation defined above, Γ and Γ' are the initial and final type environments, respectively, when checking if the type usage u is valid in contract C.

Furthermore, we introduced a new construct Θ, which is detailed below, as well as extended the typing environment Γ.

Environment Γ. What was previously defined as Γ, is now Σ, and we now define it as:

$$\Gamma ::= \Sigma \mid < \Gamma + \Gamma >$$

The latter construct represents a pair of maps for the variant type, as the map on the left is value `true` and the one on the right is for the value `false`.

Environment Θ. This new environment maps usage types u to typing environments Γ, as it prevents cycles on recursive usage types by keeping record of previously visited ones [8].

$$\Theta ::= \emptyset \mid \Theta,\ u : \Gamma$$

4.3.2.2 Typing Rules

The typing rules are similar to the ones presented in the work of Vasconcelos and Ravara [42] for the revision of the MOOL language, as we had to make minor adjustments to be able to implement them in Flint-2. In the following sections, we introduce our formalisation of some typing rules that we thought relevant. Nevertheless, all typing rules of Flint-2 new typing system are in Appendix D.

Usage Typing Rules The use of usages is the main difference between our previous type system and the new one. We now present these new rules, that are equivelant to the ones presented by Vasconcelos and Ravara [42].

$$\frac{\begin{array}{c}\forall i \in I \\ C \in \text{classes} \quad \text{fbody}(C, f, \widetilde{x}) = (e, \widetilde{ts}, \widetilde{cl}, t) \\ \texttt{self} : C[u; \widetilde{F}], \widetilde{x : t'} \quad \Gamma \vdash \widetilde{x : t'} \quad \text{un}(t') \quad \Gamma \vdash \texttt{self} : C[u_i; \widetilde{F_i}] \quad \Theta; \Gamma \vdash u_i \triangleright \Gamma'\end{array}}{\Theta; C[u; \widetilde{F}] \vdash \{m_i : u_i\}_{i \in I} \triangleright \Gamma'} \text{ T-Branch}$$

$$\frac{}{\Theta; \Gamma \vdash \triangleright \Gamma} \text{ T-BranchEnd} \qquad \frac{\Theta; \Gamma' \vdash u_t \triangleright \Gamma \quad \Theta; \Gamma'' \vdash u_f \triangleright \Gamma}{\Theta; < \Gamma' + \Gamma'' > \vdash < u_t + u_f > \triangleright \Gamma} \text{ T-Variant}$$

$$\frac{}{(\Theta, X : \Gamma); \Gamma \vdash X \triangleright \Gamma} \text{ T-UsageVar} \qquad \frac{(\Theta, X : \Gamma); \Gamma \vdash u \triangleright \Gamma'}{\Theta; \Gamma \vdash \mu X.u \triangleright \Gamma'} \text{ T-Rec}$$

Variable Lookups. The main difference between our previous implementation is that now variables can be linear or shared. Because of this, if a variable is linear it has to be removed from the environment Γ in order for no one to access it.

$$\frac{\text{lin}(t)}{\Gamma, x : t \vdash x : t \triangleright \Gamma} \text{ T-LinVar} \qquad \frac{\text{un}(t)}{\Gamma, x : t \vdash x : t \triangleright \Gamma, x : t} \text{ T-UnVar}$$

Call functions. The biggest distinction in the calling rules is that the method being called has to be part of the set of functions allowed by the current usage. Function `allows` does exactly that, where it checks if the method `f` belongs to the set of functions in usage u.

$$\frac{\begin{array}{c}\text{fbody}(C, f, x) = (e, \widetilde{cl}, \widetilde{ts}, t) \quad e_0 \neq \texttt{self} \\ \Gamma \vdash \widetilde{e' : t'} \triangleright \Gamma' \quad \Gamma' \vdash e : t \triangleright \Gamma'' \\ \Gamma' \vdash e_0 : C[u; \widetilde{F}] \quad u.\text{allows}(f) = u'\end{array}}{\Gamma \vdash e_0.f(\widetilde{x : e'}) : t \triangleright \Gamma''\{e_0 \mapsto C[u; \widetilde{G}]\}} \text{ T-Call}$$

4.3.3 Typechecking Example

In this section we will present the BlockKing example where our typechecker catches a wrong sequence of operations, to prove that it can prevent execution errors.

In the BlockKing problem, we recall that with the typestates we were able to catch an execution errors. In Figure 4.42, we demonstrate an execution where the client aBK tries to enter the gamble two times in a row (lines 4 and 5).

When running this program with Racket, the execution comes to halt as the second call for `enter` is blocked due to the typestate protection. which states that this method can only be called if and only if the current state is `canEnter`.

Figure 4.43 demonstrates this last execution state, where the the client aBK current state is `waiting`, preventing it from accessing the `enter` function.

But the use of typestates can only guarantee dynamic safety. Thus, we propose the introduction of usages in the type system to ensure safety before running the code.

```
((EOC -> init () -> 10 -> aC)
 (EOC -> init () -> 10 -> a0)
 (BlockKing -> init ((warrior : aBK) (warriorGold : 3)) -> 9 -> aBK)
 (aBK -> enter ((warrior : aC) (warriorGold : 3)) -> sender (aC))
 (aBK -> enter ((warrior : aC) (warriorGold : 3)) -> sender (aC))
 (aBK -> __callback () -> sender (a0)))
()
()
```

Figure 4.42: BlockKing initial state

```
(((aBK -> enter ((warrior : aC) (warriorGold : 3)) -> sender (aC))
  (aBK -> __callback () -> sender (a0)))
 ((aC -> (EOC 9) -> ->)
  (a0 -> (EOC 10) -> ->)
  (aBK -> (BlockKing
            (warrior : aC)
            (warriorGold : 3)
            (king : aBK)
            (Oraclize : a0)
          10) -> -> (warrior : aC) (warriorGold : 3)))
 ((aBK -> canEnter waiting))
```

Figure 4.43: BlockKing execution stopped

```
contract main (lin{(main : un{})}) {
    main (anystate) :: (anycaller) {
        public func main () -> Void {
            var bk : BlockKing[lin{(enter : lin{(callback : lin{(
                processpayment : ! canEnter)})})} ] = BlockKing.init(9, aBK);
            bk.enter((warrior : 2); (warriorGold : 3));
            bk.enter((warrior : 2); (warriorGold : 3));
        }
    }
}
```

Figure 4.44: OCaml Main Contract

In Figure 4.44, we show the contract `main`, which has the same behaviour of the statement presented in Figure 4.42: the contract `BlockKing` is initialised and then it proceeds to do two calls in a row of function `enter`.

Additionally, we introduce in Figure 4.45 the BlockKing's usage. It states that after calling the constructor method `BlockKing`, the sequence of function calls must be: `enter`; `callback`; and `processpayment`, which at this point it returns back to `enter` and so on.

```
contract BlockKing (lin{(BlockKing : mu canEnter.lin{(enter : lin{(callback :
    lin{(processpayment : ! canEnter)})})})}) {
(...)
}
```

Figure 4.45: BlockKing Usage

```
1 (bk BlockKing[lin{(callback : lin{(processpayment : ! canEnter)})} ])
2  |- bk.enter((warrior : 2); (warriorGold : 3)) : Void
3 (T-CALL)
4 //argument evaluation
5 (bk BlockKing[lin{(callback : lin{(processpayment : ! canEnter)})} ]);
6  |- bk :  BlockKing[lin{(callback : lin{(processpayment : ! canEnter)})} ]
7 (T-VAR)
8 Call to method is not permitted
9 Fatal error: exception TypecheckerUsages.InvalidMethod("enter")
```

Figure 4.46: Output of OCaml Typechecker

When testing this code with our OCaml typechecker, the execution stops when evaluating the second call of function `enter`, and an exception is thrown as this method is not permitted. Figure 4.46 shows the execution of the typechecker. In line 2 we have the statement that we want to evaluate, i.e. the second call to `enter`, which applies the typing rule T-Call. Line 4 represents the evaluation of the input arguments of the function; and lines 1 and 5 represent the typing context Γ. If we take a close look at the usage type of the variable `bk`, we can observe its usage type differs from the one previously introduced in Figure 4.45. Due to the update of the variable's usage type as methods are called, they are removed in order to guarantee the correct sequence of events.

Thus, when trying to call function `enter`, the type system will check if the method is available in its usage, therefore guaranteeing statically that the program will behave correctly.

5

Conclusions

Due to the challenges posed by the development of programming languages, specially smart contract languages, our work reinforces that the use of behavioural types and model checking has to become a key component of smart contract languages, as they can guarantee safety and correctness of the code executed.

Some languages, such as Flint-2, have the expressive power to prevent specific behaviours. The use of typestates can guarantee dynamically the correct execution of a program, and prevent some execution errors.

However, as smart contracts cannot be modified once they are deployed to the blockchain, it is important to ensure safety prior to compilation. Static type verification can be enforced with the use of session types.

Thus, we combined the typestate feature of Flint-2, which guarantees the dynamical safety throughout the protocols execution, with session types which enforce is statically, as we believe that both are complementary.

It is also necessary to develop tools to promote safe smart contracts.. A tool like Racket, can be very useful for: (i) programmers, to visually debug their code, checking the status of the relevant environments; (ii) language developers, as it allows a quick and intuitive execution environment without the need of a compiler, telling them if their language has all the mechanisms to stop undesired behaviours. Furthermore, a typechecker allows for the possibility of detecting execution errors statically, i.e. prior to the deployment.

Racket provides a very useful laboratory for defining reduction semantics and type systems. Although the lack of visualisation of the type-checker lead us to implement independent checkers in OCaml. This task, for small languages, is straightforward.

5.1 Future Work

The formalisation of a programming language, smart or not, is a weary and time consuming task. Although our process can be helpful in their design and also the development of new contracts, we believe it can be improved. That is why we propose a tool that will take the functional requirements in the assertion and our behavioural type protocol to

automatically verify the contracts correctness, which will be supported by a deductive verification platform like Why3[1], or by an automata based model checker like Cubicle[2], which would enforce dynamically functional requirements.

As future work, we will also apply our approach to other languages for other blockchain systems, to have a better coverage and thus validation of our proof-of-concept exercise.

[1] http://why3.lri.fr
[2] http://cubicle.lri.fr

Bibliography

[1] URL: https://kucoinblog.com/real-world-examples-of-smart-contracts-and-blockchain-in-industry/ (cit. on pp. 1, 8, 9).

[2] URL: http://www.dcs.gla.ac.uk/research/mungo/index.html (cit. on p. 15).

[3] A. J. Ahern. "Code mobility and Java RMI". In: (2007) (cit. on p. 18).

[4] J. Aldrich et al. "Typestate-oriented programming". In: *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. 2009, pp. 1015–1022 (cit. on p. 15).

[5] Apple. *Swift Language*. URL: https://swift.org (cit. on p. 49).

[6] N. Atzei, M. Bartoletti, and T. Cimoli. "A Survey of Attacks on Ethereum Smart Contracts SoK". In: *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. Berlin, Heidelberg: Springer-Verlag, 2017, pp. 164–186. ISBN: 9783662544549. DOI: 10.1007/978-3-662-54455-6_8. URL: https://doi.org/10.1007/978-3-662-54455-6_8 (cit. on pp. 2, 9, 10, 45).

[7] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. USA: Prentice-Hall, Inc., 1990. ISBN: 013711821X (cit. on p. xv).

[8] J. C. Campos. "Linear and shared objects in concurrent programming". PhD thesis. 2010 (cit. on pp. 16, 17, 89).

[9] S. Capecchi et al. "Amalgamating sessions and methods in object-oriented languages with generics". In: *Theoretical Computer Science* 410.2-3 (2009), pp. 142–167 (cit. on p. 16).

[10] L. Cardelli. "Type systems". In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 263–264 (cit. on p. 77).

[11] *Common Weakness Enumeration*. URL: https://cwe.mitre.org/index.html (cit. on p. 10).

[12] S. Conchon, A. Komeva, and F. Zaidi. "Verifying Smart Contracts with Cubicle". In: *FMBC'19: Workshop on Formal Methods for Blockchains*. 2019 (cit. on p. 66).

[13] A. Das et al. *Resource-Aware Session Types for Digital Contracts*. 2019. arXiv: 1902.06056 [cs.PL] (cit. on p. 1).

[14] M. Di Pirro. "How solid is Solidity? An in-dept study of solidity's type safety". In: (2018) (cit. on pp. 18–22, 25, 45, 49, 54, 76).

[15] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009 (cit. on p. 10).

[16] M. Felleisen et al. "The Racket Manifesto". In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Ed. by T. Ball et al. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 113–128. ISBN: 978-3-939897-80-4. DOI: 10.4230/LIPIcs.SNAPL.2015.113. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5021 (cit. on p. 10).

[17] X. Feng. *Operational Semantics*. URL: http://staff.ustc.edu.cn/~xyfeng/teaching/TOPL/lectureNotes/06_operational.pdf (cit. on pp. 53, 54).

[18] R. B. Findler, C. Klein, and B. Fetscher. "Redex: Practical semantics engineering". In: (2014) (cit. on p. 10).

[19] *Flint Language Guide*. https://github.com/flintlang/flint-2/blob/master/docs/guide.md#external-calls. [Online; accessed 12-January-2021] (cit. on p. 50).

[20] Flintlang. *flintlang/flint*. URL: https://github.com/flintlang/flint (cit. on p. 49).

[21] R. Garcia et al. "Foundations of typestate-oriented programming". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36.4 (2014), pp. 1–44 (cit. on p. 15).

[22] S. J. Gay et al. "Modular session types for distributed object-oriented programming". In: *ACM Sigplan Notices* 45.1 (2010), pp. 299–312 (cit. on pp. 16, 17, 88).

[23] S. Gilbert and N. Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601 (cit. on p. 5).

[24] *Glossary*. [Online; accessed 20-11-2019]. 2019. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.4/glossary.html (cit. on pp. 5, 8).

[25] H. Hüttel et al. "Foundations of Session Types and Behavioural Contracts". In: *ACM Comput. Surv.* 49.1 (2016), 3:1–3:36. DOI: 10.1145/2873052. URL: https://doi.org/10.1145/2873052 (cit. on pp. 14, 15).

[26] A. Igarashi, B. C. Pierce, and P. Wadler. "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.3 (2001), pp. 396–450 (cit. on p. 18).

[27] D. Kouzapas et al. "Typechecking protocols with Mungo and StMungo: A session type toolchain for Java". In: *Science of Computer Programming* 155 (Apr. 2018), pp. 52–75. DOI: 10.1016/j.scico.2017.10.006. URL: https://doi.org/10.1016/j.scico.2017.10.006 (cit. on pp. 15, 16).

[28] L. Lamport, R. Shostak, and M. Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176. URL: https://doi.org/10.1145/357172.357176 (cit. on p. 6).

[29] E. Meijer and P. Drayton. "Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages". In: Jan. 2004 (cit. on p. xv).

[30] R. Mitra and R. Mitra. *Tezos VS Ethererum: [The Ultimate Comparison Guide]*. Aug. 2019. URL: https://blockgeeks.com/guides/tezos-vs-ethereum-the-ultimate-comparison-guide/ (cit. on p. 18).

[31] M. Neubauer and P. Thiemann. "An implementation of session types". In: *International Symposium on Practical Aspects of Declarative Languages*. Springer. 2004, pp. 56–70 (cit. on p. 16).

[32] F. Nielson and H. R. Nielson. "From CML to process algebras". In: *International Conference on Concurrency Theory*. Springer. 1993, pp. 493–508 (cit. on p. 14).

[33] B. C. Pierce and C. Benjamin. *Types and programming languages*. MIT press, 2002 (cit. on pp. 10, 11, 54).

[34] F. Schrans, S. Eisenbach, and S. Drossopoulou. "Writing safe smart contracts in flint". In: *Conference companion of the 2nd international conference on art, science, and engineering of programming*. 2018, pp. 218–219 (cit. on p. 49).

[35] F. Schrans et al. *Flint for Safer Smart Contracts*. 2019. arXiv: 1904.06534 [cs.PL] (cit. on p. 49).

[36] I. Sergey and A. Hobor. "A concurrent perspective on smart contracts". In: *International Conm5erence on Financial Cryptography and Data Security*. Springer. 2017, pp. 478–493 (cit. on pp. 2, 63).

[37] D. Siegel. *Understanding The DAO Hack for Journalists*. July 2016. URL: https://medium.com/@pullnews/understanding-the-dao-hack-for-journalists-2312dd43e993 (cit. on p. 1).

[38] *Solidity¶*. URL: https://solidity.readthedocs.io/en/v0.4.24/index.html (cit. on pp. 18, 49).

[39] R. E. Strom and S. Yemini. "Typestate: A programming language concept for enhancing software reliability". In: *IEEE Transactions on Software Engineering* 1 (1986), pp. 157–171 (cit. on p. 15).

[40] N. Szabo. "Formalizing and Securing Relationships on Public Networks". In: *First Monday* 2.9 (1997). ISSN: 13960466. DOI: 10.5210/fm.v2i9.548. URL: https://ojphi.org/ojs/index.php/fm/article/view/548 (cit. on p. 8).

[41] A. B. Tucker, ed. *The Computer Science and Engineering Handbook*. CRC Press, 1997. ISBN: 0-8493-2909-4 (cit. on pp. xv, 14).

[42] C. Vasconcelos and A. Ravara. "A Revision of the Mool Language". In: *arXiv preprint arXiv:1604.06245* (2016) (cit. on pp. 16, 87, 89).

[43] C. Vasconcelos and A. Ravara. "From Object-Oriented Code with Assertions to Behavioural Types". In: *Proceedings of the Symposium on Applied Computing*. SAC '17. Marrakech, Morocco: Association for Computing Machinery, 2017, pp. 1492–1497. ISBN: 9781450344869. DOI: 10.1145/3019612.3019733. URL: https://doi.org/10.1145/3019612.3019733 (cit. on p. 14).

[44] V. T. Vasconcelos. "Fundamentals of session types". In: *Information and Computation* 217 (2012), pp. 52–70 (cit. on p. 16).

[45] A. K. Wright and M. Felleisen. "A syntactic approach to type soundness". In: *Information and computation* 115.1 (1994), pp. 38–94 (cit. on p. 53).

[46] R. Zhang, R. Xue, and L. Liu. "Security and Privacy on Blockchain". In: *ACM Comput. Surv.* 52.3 (July 2019). ISSN: 0360-0300. DOI: 10.1145/3316481. URL: https://doi.org/10.1145/3316481 (cit. on pp. 1, 6–8).

A

Revised Typing Rules of Featherweight Solidity

Axioms

$$\frac{}{\Gamma, c : C, \Gamma' \vdash c : C \triangleright \Gamma, c : C, \Gamma'}\ \textsc{Ref}$$

$$\frac{}{\Gamma, x : T, \Gamma' \vdash x : T \triangleright \Gamma, x : T, \Gamma'}\ \textsc{Var}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool} \triangleright \Gamma}\ \textsc{True} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool} \triangleright \Gamma}\ \textsc{False}$$

$$\frac{}{\Gamma, a : \texttt{address}, \Gamma' \vdash a : \texttt{address} \triangleright \Gamma, a : \texttt{address}, \Gamma'}\ [\textsc{Address}]$$

$$\frac{}{\Gamma \vdash \texttt{u} : \texttt{unit} \triangleright \Gamma}\ \textsc{Unit} \qquad \frac{}{\Gamma \vdash \texttt{revert} : T \triangleright \Gamma}\ \textsc{Revert}$$

$$\frac{n \in \mathbb{N}^+}{\Gamma \vdash n : \texttt{uint} \triangleright \Gamma}\ \textsc{Nat}$$

Standard rules

$$\frac{\Gamma \vdash e : \texttt{address} \triangleright \Gamma}{\Gamma \vdash \texttt{balance}(e) : \texttt{uint}}\ \text{Bal} \qquad \frac{\Gamma \vdash e : C \triangleright \Gamma}{\Gamma \vdash \texttt{address}(e) : \texttt{address}}\ \text{Addr}$$

$$\frac{\Gamma \vdash e : T \triangleright \Gamma}{\Gamma \vdash \texttt{return}\ e : T \triangleright \Gamma}\ \textsc{Return} \qquad \frac{\Gamma \vdash x : T \triangleright \Gamma \quad \Gamma \vdash e : T \triangleright \Gamma}{\Gamma \vdash x = e : T \triangleright \Gamma}\ \text{Ass}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma' x : T_1 \vdash e_2 : T_2}{\Gamma \vdash T_1\ x = e_1; e_2 : T_2 \triangleright \Gamma}\ \textsc{Decl}$$

$$\frac{\Gamma \vdash e_1 : T_1 \triangleright \Gamma' \quad \Gamma' \vdash e_2 : T_2 \triangleright \Gamma''}{\Gamma \vdash e_1; e_2 : T_2 \triangleright \Gamma''}\ \textsc{Seq}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \triangleright \Gamma \quad \Gamma \vdash e_2 : T \triangleright \Gamma \quad \Gamma \vdash e_3 : T \triangleright \Gamma}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : T \triangleright \Gamma}\ \textsc{If}$$

Mappings

$$\frac{M = \{(\widetilde{k, v})\} \quad \Gamma \vdash \tilde{k} : T_1 \triangleright \Gamma \quad \Gamma \vdash \tilde{v} : T_2 \triangleright \Gamma}{\Gamma \vdash M : \texttt{mapping}(T_1 \Rightarrow T_2) \triangleright \Gamma} \textsc{Mapping}$$

$$\frac{\Gamma \vdash e_1 : \texttt{mapping}(T_1 \Rightarrow T_2) \triangleright \Gamma \quad \Gamma \vdash e_2 : T_1 \triangleright \Gamma \quad \Gamma \vdash e_3 : T_2 \triangleright \Gamma}{\Gamma \vdash e_1[e_2 \rightarrow e_3] : \texttt{mapping}(T_1 \Rightarrow T_2) \triangleright \Gamma} \textsc{MappAss}$$

$$\frac{\Gamma e_1 : \texttt{mapping}(T_1 \Rightarrow T_2) \triangleright \Gamma \quad \Gamma \vdash e_2 : T_1 \triangleright \Gamma}{\Gamma \vdash e_1[e_2] : T_2 \triangleright \Gamma} \textsc{MappSel}$$

Contract instantiation and access

$$\frac{\Gamma \vdash e : C \triangleright \Gamma \quad \text{sv}(C) = \widetilde{T\ s} \quad s_i \in \tilde{s}}{\Gamma \vdash e.s_i : T_i \triangleright \Gamma} \textsc{StateSel}$$

$$\frac{\Gamma \vdash e_1.s : T \triangleright \Gamma \quad \Gamma \vdash e_2 : T \triangleright \Gamma}{\Gamma \vdash e_1.s = e_2 : T \triangleright \Gamma} \textsc{StateAss}$$

$$\frac{\text{sv}(C = \widetilde{T\ s} \quad)\Gamma \vdash \tilde{e} : \tilde{T} \triangleright \Gamma \quad |\tilde{e}| = |\tilde{s}| \quad \Gamma \vdash e' : \texttt{uint} \triangleright \Gamma}{\Gamma \vdash \texttt{new}\ C.\texttt{value}(e')(\tilde{e}) : C \triangleright \Gamma} \textsc{New}$$

Functions

$$\frac{\Gamma \vdash c : C \triangleright \Gamma \quad \text{ftype}(C, f) = \tilde{T_1} \rightarrow T_2}{\Gamma \vdash c.f : T_1 \xrightarrow{\sim} T_2 \triangleright \Gamma} \textsc{Fun}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : C \triangleright \Gamma \quad \Gamma \vdash e_2 : \texttt{uint} \triangleright \Gamma \\ \text{ftype}(C, f) = \tilde{T_1} \rightarrow T_2 \quad \Gamma \vdash \tilde{e} : \tilde{T_1} \triangleright \Gamma \\ |\tilde{e}| = |\tilde{T_1}|\end{array}}{\Gamma \vdash e_1.f.\texttt{value}(e_2)(\tilde{e}) : T_2 \triangleright \Gamma} \textsc{Call}$$

$$\frac{\Gamma \vdash e_3 \texttt{address} \triangleright \Gamma \quad \Gamma \vdash e_1.f.\texttt{value}(e_2)(\tilde{e} : T_2) \triangleright \Gamma}{\Gamma \vdash e_1.f.\texttt{value}(e_2).\texttt{sender}(e_3)(\tilde{e} : T_2) \triangleright \Gamma} \textsc{CallTopLevel}$$

B

Operational Rules of Flint-2

If expression

$$\frac{}{\langle \text{if true then } e_1 \text{ else } e_2, \beta, \sigma, \text{CTS}\rangle \to \langle e_1, \beta, \sigma, \text{CTS}\rangle}\ \text{IF-TRUE}$$

$$\frac{}{\langle \text{if false then } e_1 \text{ else } e_2, \beta, \sigma, \text{CTS}\rangle \to \langle e_2, \beta, \sigma, \text{CTS}\rangle}\ \text{IF-FALSE}$$

Variable and constant declaration.

$$\frac{\begin{array}{c}\text{Top}(\sigma) = a \quad x \notin \tilde{x} \\ \beta(a) = (s, c, \widetilde{x : v}) \\ \beta' = \beta[a \mapsto (s, c, \widetilde{x : v} \cdot x : v)]\end{array}}{\langle \text{var } x : t = v, \beta, \sigma, \text{CTS}\rangle \to \langle v, \beta', \sigma, \text{CTS}\rangle}\ \text{DECLVAR}$$

$$\frac{\begin{array}{c}\text{Top}(\sigma) = a \quad x \notin \tilde{c} \\ \beta(a) = ((C, \tilde{s}, n), \widetilde{c : v}, \tilde{y}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \widetilde{c : v} \cdot x : v, \tilde{y})]\end{array}}{\langle \text{let } x : t = v, \beta, \sigma, \text{CTS}\rangle \to \langle v, \beta', \sigma, \text{CTS}\rangle}\ \text{DECLCONS}$$

$$\frac{\begin{array}{c}\text{Top}(\sigma) = a \quad x \notin \tilde{x} \\ \beta(a) = ((C, \tilde{s}, n), \tilde{c}, \widetilde{x : v}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \widetilde{x : v} \cdot x : v_t)]\end{array}}{\langle \text{var } x : t, \beta, \sigma, \text{CTS}\rangle \to \langle v_t, \beta', \sigma, \text{CTS}\rangle}\ \text{DECLVAR-T}$$

$$\frac{\begin{array}{c}\text{Top}(\sigma) = a \quad x \notin \tilde{c} \\ \beta(a) = ((C, \tilde{s}, n), \widetilde{c : v}, \tilde{y}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \widetilde{c : v_t} \cdot x : v_t, \tilde{y})]\end{array}}{\langle \text{let } x : t, \beta, \sigma, \text{CTS}\rangle \to \langle v_t, \beta', \sigma, \text{CTS}\rangle}\ \text{DECLCONS-T}$$

Function call.

$$\frac{\begin{array}{c}C \in \text{classes} \quad \text{fbody}(C, f, \tilde{x}) = (e, \widetilde{ts}, \widetilde{cl}, t) \quad \text{Top}(CTS) \in \widetilde{ts} \quad a \in \widetilde{cl} \\ e_s = e\{\texttt{self} := \text{Top}(\sigma)\} \quad \beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), \text{Top}(\sigma), -\text{amount})\end{array}}{\langle a.f(\widetilde{x : v}), \beta, \sigma, \text{CTS}\rangle \to \langle e_s, \beta'', \sigma \cdot \text{Top}(\sigma), \text{CTS}\rangle}\ \text{CALL}$$

$$\frac{\begin{array}{c}C \in \text{classes} \quad \text{fbody}(C, f, \tilde{x}) = (e, \widetilde{ts}, \widetilde{cl}, t) \quad \text{Top}(CTS) \in \widetilde{ts} \quad a \in \widetilde{cl} \\ e_s = e\{\texttt{self} := \text{Top}(\sigma)\} \quad \beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), \text{Top}(\sigma), -\text{amount})\end{array}}{\langle \text{try ? } (a.f(\widetilde{x : v})), \beta, \sigma, \text{CTS}\rangle \to \langle e_s, \beta'', \sigma \cdot \text{Top}(\sigma), \text{CTS}\rangle} \text{ TRY}$$

$$\frac{\begin{array}{c}C \in \text{classes} \quad \text{fbodyinit}(C, f, \tilde{x}) = (e, \widetilde{cl}, t) \quad a \in \widetilde{cl} \\ e_s = e\{\texttt{self} := \text{Top}(\sigma)\} \quad \beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), \text{Top}(\sigma), -\text{amount})\end{array}}{\langle a.f(\widetilde{x : v}), \beta, \sigma, \text{CTS}\rangle \to \langle e, \beta'', \sigma \cdot \text{Top}(\sigma), \text{CTS}\rangle} \text{ CALL-2}$$

$$\frac{\begin{array}{c}C \in \text{classes} \quad \text{fbody}(C, f, \widetilde{x : v}) = (e, \widetilde{ts}, \widetilde{cl}, t) \\ a \in \widetilde{cl} \quad \texttt{anystate} \in \widetilde{ts} \\ e_s = e\{\texttt{self} := \text{Top}(\sigma)\} \quad \beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), \text{Top}(\sigma), -\text{amount})\end{array}}{\langle a.f(\widetilde{x : v}), \beta, \sigma, \text{CTS}\rangle \to \langle e, \beta'', \sigma \cdot \text{Top}(\sigma), \text{CTS}\rangle} \text{ CALL-ANY}$$

$$\frac{\text{uptbal}(\beta, \text{Top}(\sigma), -\text{amount}) = \bot}{\langle a.f(\widetilde{x : v}), \beta, \sigma, \text{CTS}\rangle \to \langle \text{revert}, \beta, \sigma, \text{CTS}\rangle} \text{ CALL-SENDER-R}$$

$$\frac{\begin{array}{c}\beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ C \in \text{classes} \quad \text{fbody}(C, f, \tilde{x}) = (e, \widetilde{ts}, \widetilde{cl}, t) \\ \text{Top}(CTS) \in \widetilde{ts} \quad e_s = e\{\texttt{self} := a\} \\ a \in \widetilde{cl} \quad \text{Top}(\sigma) = \emptyset \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), a_s, -\text{amount})\end{array}}{\langle a.f(\widetilde{x : v}).\text{sender}(a_s), \beta, \sigma, \text{CTS}\rangle \to \langle e_s, \beta'', \sigma \cdot a_s, \text{CTS}\rangle} \text{ CALL-SENDER}$$

$$\frac{\begin{array}{c}\beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ C \in \text{classes} \quad \text{fbodyinit}(C, f, \tilde{x}) = (e, \widetilde{cl}, t) \\ e_s = e\{\texttt{self} := a\} \quad a \in \widetilde{cl} \quad \text{Top}(\sigma) = \emptyset \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), a_s, -\text{amount})\end{array}}{\langle a.f(\widetilde{x : v}).\text{sender}(a_s), \beta, \sigma, \text{CTS}\rangle \to \langle e_s, \beta'', \sigma \cdot a_s, \text{CTS}\rangle} \text{ CALL-SENDER-2}$$

$$\frac{\begin{array}{c}\beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \\ C \in \text{classes} \quad \text{fbody}(C, f, \tilde{x}) = (e, \widetilde{ts}, \widetilde{cl}, t) \quad \texttt{anystate} \in \widetilde{ts} \\ e_s = e\{\texttt{self} := a\} \quad a \in \widetilde{cl} \quad \text{Top}(\sigma) = \emptyset \\ \beta' = \beta[a \mapsto ((C, \tilde{s}, n), \tilde{c}, \tilde{y} \cdot \widetilde{x : v})] \\ \beta'' = \text{uptbal}(\text{uptbal}(\beta', a, \text{amount}), a_s, -\text{amount})\end{array}}{\langle a.f(\widetilde{x : v}).\text{sender}(a_s), \beta, \sigma, \text{CTS}\rangle \to \langle e_s, \beta'', \sigma \cdot a_s, \text{CTS}\rangle} \text{ CALL-SENDER-ANY}$$

$$\frac{\text{uptbal}(\beta, \text{Top}(\sigma), -\text{amount}) = \bot}{\langle a.f(\widetilde{x : v}).\text{sender}(a_s), \beta, \sigma, \text{CTS}\rangle \to \langle \text{revert}, \beta, \sigma, \text{CTS}\rangle} \text{ CALL-SENDER-R}$$

Variable lookup.

$$\frac{\beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \quad x \in \tilde{s} \cup \tilde{c}}{\langle a.x, \beta, \sigma, \text{CTS}\rangle \to \langle v, \beta, \sigma, \text{CTS}\rangle} \text{ STATESEL}$$

$$\frac{\begin{array}{c} a = \text{Top}(\sigma) \quad x \in \tilde{y} \\ \beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \end{array}}{\langle x, \beta, \sigma, \text{CTS}\rangle \to \langle v, \beta, \sigma, \text{CTS}\rangle} \text{ VAR}$$

Variable assignment.

$$\frac{\beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \quad x \in \tilde{s} \cup \tilde{c}}{\langle a.x = v, \beta, \sigma, \text{CTS}\rangle \to \langle v, \beta[a.x \mapsto v], \sigma, \text{CTS}\rangle} \text{ STATEASS}$$

$$\frac{\beta(a) = ((C, \tilde{s}, n), \tilde{c}, \tilde{y}) \quad x \in \tilde{y}}{\langle x = v, \beta, \sigma \cdot a, \text{CTS}\rangle \to \langle v, \beta[a.x \mapsto v], \sigma \cdot a, \text{CTS}\rangle} \text{ ASS}$$

Sequential composition.

$$\frac{}{\langle v\ e, \beta, \sigma, \text{CTS}\rangle \to \langle e, \beta, \sigma, \text{CTS}\rangle} \text{ SEQ}$$

$$\frac{\sigma = \beta_0 \cdot \tilde{a}}{\langle \text{revert } e, \beta, \sigma, \text{CTS}\rangle \to \langle \text{revert}, \beta_0, \sigma, \text{CTS}\rangle} \text{ SEQ-R}$$

Become statement.

$$\frac{}{\langle \text{become } ts, \beta, \sigma, \text{CTS}\rangle \to \langle \text{unit}, \beta, \sigma, \text{CTS} \cdot ts\rangle} \text{ BECOME}$$

Return statement.

$$\frac{}{\langle \text{return } v, \beta, \sigma \cdot a, \text{CTS}\rangle \to \langle v, \beta, \sigma, \text{CTS}\rangle} \text{ RETURN}$$

Contract initialization.

$$\frac{\begin{array}{c} \text{finit}(C, \widetilde{x : v}) = (\widetilde{s : v'}, \widetilde{c : v''}, e) \\ e_s = e\{\text{self} := a, \tilde{x} := \tilde{v}\} \quad a \notin \text{dom}(\beta) \\ cn = a \mapsto ((C, \widetilde{s : v'}, n), \widetilde{c : v''},) \end{array}}{\langle C.\text{init}(\widetilde{x : v}).n.a, \beta, \sigma, \text{CTS}\rangle \to \langle e_s, \beta \cdot cn, \sigma \cdot a, \text{CTS}\rangle} \text{ INIT}$$

Mappings.

$$\frac{}{\langle M[v_1], \beta, \sigma, \text{CTS}\rangle \to \langle M(v_1), \beta, \sigma, \text{CTS}\rangle} \text{ MAPSEL}$$

$$\frac{M' = M \backslash \{(v_1, M(v_1))\} \cup \{v_1, v_2\}}{\langle M[v_1 : v_2], \beta, \sigma, \text{CTS}\rangle \to \langle M', \beta, \sigma, \text{CTS}\rangle} \text{ MAPASS}$$

C

Typing Rules of Flint-2

Axioms

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool} \triangleright \Gamma}\ \text{T-True} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool} \triangleright \Gamma}\ \text{T-False}$$

$$\frac{}{\Gamma \vdash \texttt{unit} : \texttt{Void} \triangleright \Gamma}\ \text{T-Unit}$$

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n\colon \texttt{Int} \triangleright \Gamma}\ \text{T-Int} \qquad \frac{\Gamma' = \Gamma, a :: \texttt{Address}}{\Gamma, a\colon \texttt{Address} \vdash a\colon \texttt{Address} \triangleright \Gamma'}\ \text{T-Address}$$

$$\frac{}{\Gamma, x : t \vdash x : t \triangleright \Gamma, x : t}\ \text{T-Var} \qquad \frac{}{\Gamma, a : C \vdash a : C \triangleright \Gamma, a : C}\ \text{T-Ref}$$

Standard Rules

$$\frac{\Gamma \vdash e : t \triangleright \Gamma}{\Gamma \vdash \texttt{return}\ e : t \triangleright \Gamma}\ \text{T-Return} \qquad \frac{\Gamma \vdash e_1 : t_1 \triangleright \Gamma' \quad \Gamma' \vdash e_2 : t_2 \triangleright \Gamma''}{\Gamma \vdash e_1; e_2 : t_2 \triangleright \Gamma''}\ \text{T-Seq}$$

$$\frac{\Gamma, x : t \vdash e : t \triangleright \Gamma'}{\Gamma \vdash \texttt{var}\ x : t = e : t \triangleright \Gamma'}\ \text{T-DeclVar} \qquad \frac{}{\Gamma \vdash \texttt{var}\ x : t : t \triangleright \Gamma, x : t}\ \text{T-DeclVar2}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \triangleright \Gamma \quad \Gamma \vdash e_2 : t \triangleright \Gamma \quad \Gamma \vdash e_3 : t \triangleright \Gamma}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t \triangleright \Gamma}\ \text{T-If}$$

$$\frac{\Gamma, x : t \vdash e : t \triangleright \Gamma'}{\Gamma \vdash \texttt{let}\ x : t = e : t \triangleright \Gamma'}\ \text{T-DeclCons}$$

$$\frac{}{\Gamma \vdash \texttt{let}\ x : t : t \triangleright \Gamma, x : t}\ \text{T-DeclCons2} \qquad \frac{\Gamma \vdash x : t \quad \Gamma \vdash e : t}{\Gamma \vdash x = e : t \triangleright \Gamma}\ \text{T-Ass}$$

State variables

$$\frac{\Gamma \vdash e : C \triangleright \Gamma \quad \text{sv}(C) = (\widetilde{x : t}) \quad x \in \tilde{x}}{\Gamma \vdash e.x : t \triangleright \Gamma}\ \text{T-StateSel}$$

$$\frac{\Gamma \vdash e_1.x : t \triangleright \Gamma \quad \Gamma \vdash e_2 : t \triangleright \Gamma}{\Gamma \vdash e_1.x = e_2 : t \triangleright \Gamma}\ \text{T-StateAss}$$

Mapping

$$\frac{M = \widetilde{k : v} \quad \Gamma \vdash \tilde{k} : t_k \triangleright \Gamma \quad \Gamma \vdash \tilde{v} : t_v \triangleright \Gamma}{\Gamma \vdash M : (t_k : t_v) \triangleright \Gamma} \text{ T-Mapping}$$

$$\frac{\Gamma \vdash e_1 : (t_k : t_v) \triangleright \Gamma \quad \Gamma \vdash e_2 : t_k \triangleright \Gamma \quad \Gamma \vdash e_3 : t_v \triangleright \Gamma}{\Gamma \vdash e_1[e_2 : e_3] : (t_k : t_v) \triangleright \Gamma} \text{ T-MapAss}$$

$$\frac{\Gamma \vdash e_1 : (t_k : t_v) \triangleright \Gamma \quad \Gamma \vdash e_2 : t_k \triangleright \Gamma}{\Gamma \vdash e_1[e_2] : t_v \triangleright \Gamma} \text{ T-MapSel}$$

Call functions

$$\frac{C \in \text{classes} \quad \text{sv}(C) = \widetilde{x : t} \quad \Gamma \vdash \widetilde{e : t} \triangleright \Gamma}{\Gamma \vdash C.\text{init}(\widetilde{x : e}).a : \texttt{Void} \triangleright \Gamma, a : \texttt{Address}} \text{ T-Init}$$

$$\frac{\Gamma \vdash e_0.f(\widetilde{x : e}) : t \quad \Gamma \vdash a : \texttt{Address} \triangleright \Gamma}{\Gamma \vdash e_0.f(\widetilde{x : e}).\texttt{sender}(a) : t \triangleright \Gamma} \text{ T-CallSender}$$

$$\frac{\Gamma \vdash e_0 : \texttt{Address} \quad \text{fbody}(C, f, x) = (e'' \cdot \texttt{return } e, \widetilde{cl}, \widetilde{ts}, t) \quad \Gamma \vdash \widetilde{e' : t'} \triangleright \Gamma}{\Gamma \vdash e_0.f(\widetilde{x : e'}) : t \triangleright \Gamma} \text{ T-Call}$$

$$\frac{\Gamma \vdash e_0 : \texttt{Address} \quad \text{fbodyinit}(C, f, x) = (e'' \cdot \texttt{return } e, \widetilde{cl}, t) \quad \Gamma \vdash \widetilde{e' : t'} \triangleright \Gamma}{\Gamma \vdash e_0.f(\widetilde{x : e'}) : t \triangleright \Gamma} \text{ T-Call-2}$$

$$\frac{\Gamma \vdash e_0 : \texttt{Address} \triangleright \Gamma \quad \text{fbody}(C, f, x) = (e'' \cdot \texttt{return } e, \widetilde{cl}, \widetilde{ts}, t) \quad \Gamma \vdash \widetilde{e' : t'} \triangleright \Gamma \quad \texttt{anystate} \in \widetilde{ts}}{\Gamma \vdash e_0.f(\widetilde{x : e'}) : t \triangleright \Gamma} \text{ T-Call-Any}$$

D

Typing Rules of Flint-2 with Usages

Typing Rules for Contracts

$$\frac{\text{check}(\emptyset, u) \quad C[u;\emptyset] \vdash u \triangleright C[u;\widetilde{F}] \quad \text{un}(\widetilde{F})}{\vdash \texttt{contract} C\ (u)\ \{\widetilde{F}\}\ \widetilde{PB}}\ \textsc{T-Contract}$$

Usage Typing Rules

$$\frac{\begin{array}{c}\forall i \in I \\ C \in \text{classes} \quad \text{fbody}(C, f, \widetilde{x}) = (e, \widetilde{ts}, \widetilde{cl}, t) \\ \texttt{self} : C[u;\widetilde{F}], \widetilde{x : t'} \quad \Gamma \vdash \widetilde{x : t'} \quad \text{un}(t') \quad \Gamma \vdash \texttt{self} : C[u_i;\widetilde{F_i}] \quad \Theta;\Gamma \vdash u_i \triangleright \Gamma'\end{array}}{\Theta; C[u;\widetilde{F}] \vdash \{m_i : u_i\}_{i\in I} \triangleright \Gamma'}\ \textsc{T-Branch}$$

$$\frac{}{\Theta;\Gamma \vdash \triangleright\Gamma}\ \textsc{T-BranchEnd} \qquad \frac{\Theta;\Gamma' \vdash u_t \triangleright \Gamma \quad \Theta;\Gamma'' \vdash u_f \triangleright \Gamma}{\Theta; <\Gamma' + \Gamma''> \vdash < u_t + u_f > \triangleright \Gamma}\ \textsc{T-Variant}$$

$$\frac{}{(\Theta, X : \Gamma);\Gamma \vdash X \triangleright \Gamma}\ \textsc{T-UsageVar} \qquad \frac{(\Theta, X : \Gamma);\Gamma \vdash u \triangleright \Gamma'}{\Theta;\Gamma \vdash \mu X.u \triangleright \Gamma'}\ \textsc{T-Rec}$$

Axioms

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool} \triangleright \Gamma}\ \textsc{T-True} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool} \triangleright \Gamma}\ \textsc{T-False}$$

$$\frac{}{\Gamma \vdash \texttt{unit} : \texttt{Void} \triangleright \Gamma}\ \textsc{T-Unit}$$

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n\text{: } \texttt{Int} \triangleright \Gamma}\ \textsc{T-Int} \qquad \frac{\Gamma' = \Gamma, a :: \texttt{Address}}{\Gamma, a\text{: } \texttt{Address} \vdash a\text{: } \texttt{Address} \triangleright \Gamma'}\ \textsc{T-Address}$$

$$\frac{\text{lin}(t)}{\Gamma, x : t \vdash x : t \triangleright \Gamma}\ \textsc{T-LinVar} \qquad \frac{\text{un}(t)}{\Gamma, x : t \vdash x : t \triangleright \Gamma, x : t}\ \textsc{T-UnVar}$$

Standard Rules

$$\frac{\Gamma \vdash e_1 : t_1 \triangleright \Gamma' \quad \Gamma' \vdash e_2 : t_2 \triangleright \Gamma''}{\Gamma \vdash e_1; e_2 : t_2 \triangleright \Gamma''}\ \textsc{T-Seq} \qquad \frac{\Gamma, x : t \vdash e : t \triangleright \Gamma'}{\Gamma \vdash \texttt{var}\ x : t = e : t \triangleright \Gamma'}\ \textsc{T-DeclVar}$$

$$\frac{\Gamma \vdash x : t \quad \Gamma \vdash e : t \triangleright \Gamma'}{\Gamma \vdash x = e : t \triangleright \Gamma'}\ \textsc{T-Ass} \qquad \frac{\Gamma \vdash e_1 : \texttt{Bool} \triangleright \Gamma \quad \Gamma \vdash e_2 : t \triangleright \Gamma \quad \Gamma \vdash e_3 : t \triangleright \Gamma}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t \triangleright \Gamma}\ \textsc{T-If}$$

State variables

$$\frac{\Gamma \vdash \texttt{this} : C[u;\widetilde{F}] \quad \widetilde{F}(f) = t \quad \texttt{lin}(t)}{\Gamma \vdash \texttt{this}.f : t \triangleright \Gamma\{\texttt{self} \mapsto C[u;(\widetilde{F} \setminus f)]\}} \text{ T-LinStateSel}$$

$$\frac{\Gamma \vdash \texttt{this} : C[u;\widetilde{F}] \quad \widetilde{F}(f) = t \quad \mathsf{un}(t)}{\Gamma \vdash \texttt{this}.f : t \triangleright \Gamma} \text{ T-UnStateSel}$$

$$\Gamma \vdash e : g\Gamma' \frac{\Gamma' \vdash \texttt{self} : C[u;\widetilde{F}] \quad \mathrm{sv}(C) = \widetilde{(x:t)} \quad f \in \widetilde{x}}{\Gamma \vdash \texttt{self}.f = e : \mathsf{Void}\Gamma'\{self \mapsto C[u;(\widetilde{F} \cup (f\,t))]\}} \text{ T-StateAss}$$

Call functions

$$\frac{\begin{array}{c} C \in \mathrm{classes} \quad \mathrm{sv}(C) = \widetilde{x:t} \quad \Gamma \vdash \widetilde{e:t} \triangleright \Gamma \\ C.\mathrm{usage} = \texttt{lin}[C;u] \end{array}}{\Gamma \vdash C.\mathrm{init}(\widetilde{x:e}).a : C[u] \triangleright \Gamma, a : \texttt{Address}} \text{ T-Init}$$

$$\frac{\Gamma \vdash e_0.f(\widetilde{x:e}) : t \quad \Gamma \vdash a : \texttt{Address} \triangleright \Gamma}{\Gamma \vdash e_0.f(\widetilde{x:e}).\texttt{sender}(a) : t \triangleright \Gamma} \text{ T-CallSender}$$

$$\frac{\begin{array}{c} \mathrm{fbody}(C,f,x) = (e,\widetilde{cl},\widetilde{ts},t) \quad e_0 \neq \texttt{self} \\ \Gamma \vdash \widetilde{e':t'} \triangleright \Gamma' \quad \Gamma' \vdash e : t \triangleright \Gamma'' \\ \Gamma' \vdash e_0 : C[u;\widetilde{F}] \quad u.\mathrm{allows}(f) = u' \end{array}}{\Gamma \vdash e_0.f(\widetilde{x:e'}) : t \triangleright \Gamma''\{e_0 \mapsto C[u;\widetilde{G}]\}} \text{ T-Call}$$

$$\frac{\begin{array}{c} \mathrm{fbody}(C,f,x) = (e,\widetilde{cl},\widetilde{ts},t) \quad e_0 \neq \texttt{self} \\ \Gamma \vdash \widetilde{e':t'} \triangleright \Gamma' \quad \Gamma' \vdash e : t \triangleright \Gamma'' \\ \Gamma' \vdash \texttt{self} : C[u;\widetilde{F}] \end{array}}{\Gamma \vdash \texttt{self}.f(\widetilde{x:e'}) : t \triangleright \Gamma''} \text{ T-SelfCall}$$

Featherweight Solidity - Original Reduction Rules

If expression

$$\frac{}{\langle \text{if true then } e_1 \text{else } e_2, \beta, \sigma\rangle \to \langle e_1, \beta, \sigma\rangle}\ \text{IF-TRUE}$$

$$\frac{}{\langle \text{if true then } e_1 \text{else } e_2, \beta, \sigma\rangle \to \langle e_2, \beta, \sigma\rangle}\ \text{IF-FALSE}$$

Sequential composition

$$\frac{\sigma = \beta_0}{\langle v; e, \beta, \sigma\rangle \to \langle e, \beta, \beta\rangle}\ \text{SEQ-C} \qquad \frac{\sigma = \beta_0}{\langle \text{revert}; e, \beta, \sigma\rangle \to \langle \text{revert}, \beta_0, \sigma\rangle}\ \text{SEQ-R}$$

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

Variables

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

$$\frac{}{\langle x, \beta, \sigma\rangle \to \langle \beta(x), \beta, \sigma\rangle}\ \text{VAR} \qquad \frac{x \in \text{dom}(\beta)}{\langle x = v, \beta, \sigma\rangle \to \langle v, \beta[x \mapsto v], \sigma\rangle}\ \text{ASS}$$

Mappings

$$\frac{}{\langle M[v_1], \beta, \sigma\rangle \to \langle M(v_1), \beta, \sigma\rangle}\ \text{MAPPSEL} \qquad \frac{M' = M \setminus \{(v_1, M(v_1))\} \cup \{(v_1, v_2)\}}{\langle M[v_1 \to v_2], \beta, \sigma\rangle \to \langle M', \beta, \sigma\rangle}\ \text{MAPPASS}$$

Address

$$\frac{\hat{\beta}(c) = a}{\langle \text{address}(c), \beta, \sigma\rangle \to \langle a, \beta, \sigma\rangle}\ \text{ADDRESS}$$

Contract instantiation

$$\frac{(c, a) \notin \text{dom}(\beta) \quad \text{sv}(C) = \tilde{T}s \quad |\tilde{v}| = |\tilde{s}| \quad \text{Top}(\sigma) = \emptyset}{\langle \text{new } C.\text{value}(n)(\tilde{v}), \beta, \sigma\rangle \to \langle c, \beta \cdot [(c, a) \mapsto (C, s\tilde{:}v, n)], \sigma\rangle}\ \text{NEW-2}$$

State variables

$$\frac{\beta(c) = (C, s \tilde{:} v, n) \quad s \in \tilde{s}}{\langle c.s, \beta, \sigma \rangle \to \langle v, \beta, \sigma \rangle} \text{ STATESEL}$$

$$\frac{\beta(c) = (C, s \tilde{:} v, n) \quad s \in \tilde{s}}{\langle c.s = v', \beta, \sigma \rangle \to \langle v', \beta[c.s \mapsto v'], \sigma \rangle} \text{ STATEASS}$$

Money transfer

$$\frac{\begin{array}{c}\beta^C(a) = C \quad \text{fbody}(C, fb, \{\}) = (\{\}, e) \\ \beta' = \text{uptbal}(\text{uptbal}(\beta, a, n), \text{Top}(\sigma), -n)\end{array}}{\langle a.\text{transfer}(n), \beta, \sigma \rangle \to \langle e\{\text{this} := c, \text{msg.sender} := \text{Top}(\sigma), \text{msg.value} := n\}, \beta, \sigma \rangle} \text{ TRANSFER}$$

Function calls

$$\frac{\begin{array}{c}\hat{\beta}(c) = a \quad \beta^C(c) = C \quad \text{fbody}(C, f, \tilde{v}) = (\tilde{x}, e) \quad \tilde{x} \notin \text{dom}(\beta) \\ \beta' = \text{uptbal}(\text{uptbal}(\beta, a, n), \text{Top}(\sigma), -n) \cdot [\tilde{x} \mapsto \tilde{v}] \\ e_s = e\{\text{this} := c, \text{msg.sender} := \text{Top}(\sigma), \text{msg.value} := n\}\end{array}}{\langle c.f.\text{value}(n)(\tilde{v}), \beta, \sigma \rangle \to \langle e_s, \beta', \sigma \cdot a \rangle} \text{ CALL}$$

$$\frac{\begin{array}{c}\hat{\beta}(c) = a \quad \beta^C(c) = C \quad \text{fbody}(C, f, \tilde{v}) = (\tilde{x}, e) \quad \tilde{x} \notin \text{dom}(\beta) \\ \beta' = \text{uptbal}(\text{uptbal}(\beta, a, n), a', -n) \cdot [\tilde{x} \mapsto \tilde{v}] \quad \text{Top}(\sigma) = \emptyset \\ te_s = e\{\text{this} := c, \text{msg.sender} := a', \text{msg.value} := n\}\end{array}}{\langle c.f.\text{value}(n).\text{sender}(a')(\tilde{v}), \beta, \sigma \rangle \to \langle e_s, \beta', \sigma \cdot a \rangle} \text{ CALL-TOP-LEVEL}$$

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

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