Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

CORSO DI LAUREA MAGISTRALE IN INFORMATICA

How Solid is Solidity?

An In-dept Study of Solidity's Type Safety

Master thesis

Supervisor
Prof. Silvia Crafa

Author
Matteo Di Pirro

SEPTEMBER 2018

Matteo Di Pirro:
How Solid is Solidity?
An In-dept Study of Solidity's Type Safety,
Corso di Laurea Magistrale in Informatica, © September 2018

Dedicated to my family,
for their love and endless support

Abstract

Blockchain has evolved a lot in the last years: one of the most important features is the possibility, for mutually untrusted parties, to interact with one another without relying on a third party trusted entity. This interaction is made possible by the so-called smart contracts, passive arbitrary programs executed in a decentralized network and usually manipulating money. One of the main platforms in this sense is Ethereum, and a number of programming languages exist in its ecosystem, all with points of strength and flaws. Of these, the most widely used is for sure Solidity. In spite of its high potential, repeated security concerns have undercut the trust in this way of handling money. Bugs and undesired behaviors are worsened by the impossibility of patching a contract once it is deployed on the blockchain. As a consequence, many analysis tools have been developed by researchers. However, those operating on Solidity lack a real formalization of the core of this language.

We aim to fill the gap with Featherweight Solidity (FS). To the best of our knowledge, this is the first calculus including the semantics as well as the type system. Thanks to it, we proved the theorem of Type Safety for Solidity (claimed in the official documentation, although not supported by any public proof). We also formalized, and proved, an extended Type Safety statement addressing groups of transactions. During this process, we found out that Solidity's type system is far from being safe with respect to any type of error: in many occasions, contract interfaces are not consulted at compile-time, and this makes the execution raise an exception and the user waste money. Sometimes, in particular when transferring money from one party to another, exceptions can be avoided by simply looking at, at compile-time, contract interfaces.

We also propose an extension of the type system, FS^{+}, that targets this undesired behavior. We prove that Type Safety is maintained, but we formalize additional theorems stating new safety properties, too. In particular, but not only, FS^{+} statically detects, and consequently rules out, ill-formed money transfers made by means of the Solidity's built-in `transfer` function. We compared it with Solidity, and showed that including this extension does not change radically the way of writing smart contracts, whereas it makes them much safer.

FS has its limitations, and some aspects of Solidity have not been modeled yet, but we believe that this first attempt of formalization could trace an important path to further investigate new language features and contracts vulnerabilities.

The future belongs to those who
believe in the beauty of their dreams.

Eleanor Roosevelt

Acknowledgments

Of all the sections of a thesis this is always the most difficult to write, and no words will ever suffice to thank the people I've met during these two years.

First and foremost, I want to thank my parents, Daniela e Paolo, to whom I owe everything. Although I don't usually express it, they have always supported me in all my endeavors, and have encouraged me to aim high. I can't believe this day has finally come, and I want you to know that this wouldn't have been possible without you. Though no amount of thanks will suffice, thank you both for giving me strength to reach for the starts and chase my dreams; thank you for always being there whenever I needed it, and even when I though I didn't need it; thank you for your endless and unconditional love, support, and wisdom. To Mum and Dad, I love you so much!

Thanks to my beloved Elena, who suffered a great deal of my nonsense during these five years of University. I know I've stressed you out in many occasions, but you have always been by my side, even when there were more than a thousand kilometers between us. Thanks for your love, your letters, and for our long Skype calls. You've always encouraged me to aim high, giving me, day by day, the strength and the love to make my dreams come true. Sometimes I still pinch myself to make sure you're real: never in my wildest dreams I thought that something like this would happen. But it did, and we're together. Thank you for making my life wonderful and for keeping me in yours.

I owe a special thanks to all my friends, especially the people I met in Belgium. You helped me more times than you'll ever know and made my staying so pleasant and amazing that I'll never forget you. I grew up as a person and as a man during those five months and I'm so happy to have spent so much time with all of you. Thanks for letting me be part of your life and for cheering me up during the darkest times.

Last but not least, a deep thanks to my supervisor, Silvia Crafa, whose work and work ethic inspired me so much. You gave me a golden opportunity when you let me undertake this work with you, and I hope to have repaid your faith. I've been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly. Thank you for offering me the right advices when I needed them the most and for always asking the difficult questions whenever I presented my work in front of you.

Matteo Di Pirro

The future belongs to those who
believe in the beauty of their dreams.

Eleanor Roosevelt

Ringraziamenti

Di tutte le sezioni di una tesi, questa è sempre la più difficile da scrivere, e nessuna frase sarà mai sufficiente per ringraziare le persone che ho incontrato durante questi anni.

Prima di tutto voglio ringraziare i miei genitori, Daniela e Paolo, ai quali devo tutto. Sebbene non sia solito dirlo, loro mi hanno sempre supportato in tutti i miei sforzi e mi hanno incoraggiato a puntare in alto. Non riesco a credere che questo giorno sia finalmente arrivato, e voglio che sappiate che tutto questo non sarebbe stato possibile senza di voi. Anche se nessun ringraziamento sarà mai sufficiente, grazie ad entrambi per avermi dato la forza di raggiungere traguardi sempre più alti e di inseguire i miei sogni; grazie per esserci sempre stati ogniqualvolta avessi bisogno, e anche quando pensavo di non averne bisogno; grazie per il vostro infinito ed incondizionato amore, supporto e saggezza. A Mamma e Papà, vi voglio un sacco di bene!

Grazie alla mia amata Elena, che ha sopportato molte mie sfuriate durante questi cinque anni di Università. So di averti stressata in molte occasioni, ma tu sei sempre rimasta al mio fianco, anche quando c'erano più di mille chilometri tra noi. Grazie per il tuo amore, per le tue lettere e per le nostre lunghe chiamate su Skype. Mi hai sempre incoraggiato a puntare in alto, dandomi, giorno per giorno, la forza e l'amore per rendere i miei sogni realtà. A volte mi pizzico per essere sicuro che tu sia reale: mai, nei miei sogni più nascosti, avrei pensato che qualcosa del genere potesse accadere. Ma è successo, e siamo insieme. Grazie di rendere la mia vita meravigliosa e di tenermi nella tua.

Devo un ringraziamento speciale a tutti i miei amici, specialmente a quelli incontrati in Belgio. Mi avete aiutato più volte di quante possiate immaginare, e avete reso il mio soggiorno così piacevole che non vi dimenticherò mai. Durante quei cinque mesi sono cresciuto come persona e come uomo, e sono felicissimo di tutto il tempo che ho passato con voi. Grazie per avermi permesso di essere parte della vostra vita e per avermi tirato su il morale nei periodi più duri.

Infine voglio ringraziare profondamente la mia relatrice, Silvia Crafa, il cui lavoro e la cui etica lavorativa mi hanno molto ispirato. Mi ha dato una grandissima opportunità quando mi ha lasciato intraprendere questo lavoro con Lei, e spero di aver ripagato la sua fiducia. Mi sento estremamente fortunato ad avere avuto una relatrice che si interessasse così tanto al mio lavoro, e che rispondesse velocemente a tutte le mie domande. Grazie per avermi dato i giusti consigli quando mi servivano di più e per avermi fatto le domande più spinose quando presentavo il mio lavoro di fronte a Lei.

Matteo Di Pirro

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Blockchain technology was proposed to support financial transactions in the Bitcoin [34] system, but it has grown a lot over the last few years. In fact, it has become increasingly important in many industries, thanks to the possibility of executing Turing-complete programs that store their results on the history of the blockchain. For example, mutually untrusted parties may transfer money to one another without relying on a centralized third party. These programs are often referred to as *smart contracts*, and Solidity is one of the main languages used nowadays to implement them. It is compiled down into bytecode running on an Ethereum Virtual Machine (EVM) [53], the running environment for the Ethereum [9] blockchain.

However, this new class of programs poses a number of new criticalities: in fact, smart contracts are used not only for industrial applications, but also in critical domains, such as intelligent transportation systems [54] [46], insurance and energy industry [16], smart cities [38], and health care [5]. For these reasons, it is highly important to provide additional guarantees on their execution. For example, Solidity states that it is a type-safe language, but no public proof is provided to support such a claim, and, even worse, no precise statement of what "type-safe" means in this context has been made. Consequently, no guarantees are given on the execution of smart contracts. Bearing in mind that they can manipulate billions of dollars, this is particularly undesirable. The recent history is full of examples of contracts not executing as expected, exploited, or blocking money due to a clumsy implementation [18] [20]. Furthermore, Solidity let programmers use harmful primitives, such as `delegatecall` or inline assembly, that work around the type system, making the former safety totally useless.

Clearly, Solidity's security aspects had not received much attention initially, until the first bugs and issues arose. After that, a number of analysis tools have been developed, but none of them relies on a sound formalization of the language. In this work, we seek to set a milestone in the formal aspect of defining smart contracts. Our aim is two-fold. First, we strive to provide a full and sound formalization of the core part of Solidity, in order to study its type safety. Then, we want to use such formalization to investigate on the unexpected behaviors, looking for a way to avoid them.

Many attempts of formalizing the EVM have been made in the recent years [23] [4] [24] [21]. Nonetheless, they all rely on a trustworthy logical framework capable of expressing safety properties, but do not provide a platform-independent formalization. Furthermore, their target is the virtual machine, and consequently the bytecode, instead of the Solidity code. This has the obvious advantage that the formalization can apply no matter the actual programming language (alongside Solidity are many other languages, such as LLL, Serpent, Viper, and so forth), since the EVM executes bytecode. Never-

theless, we believe that this approach has also an important limitation: programmers write smart contracts in high-level languages that have to be carefully analyzed, too. Bugs are almost always introduced during the implementation phase, and it is worth to investigate on how we could statically identify buggy programs with the goal of ruling them out.

With this reasoning in mind we propose Featherweight Solidity (FS), the first abstract calculus modeling the core of Solidity including the operational semantics as well as the type system. FS is completely "platform-independent" and relies on no trustworthy logical framework. On our way to the proof of Type Safety we found that when an exception occurs during the execution of a contract, the entire program stops and returns a *revert*. The runtime detects this situation and rolls back any changes made to the blockchain during the execution, thus leaving the former as if the contract had never run. However, one change does remain: the nodes executing programs have to be paid, in advance, an amount of money which is not reimbursed in case of error. Hence, it is of interest of anyone "activating" a smart contract (i.e. making a contract execute) that the execution proceeds without any exceptions. One of the most sensitive scenarios in this sense is the transfer of money, where a "special" function of the target contract (the so-called *fallback*) is invoked.

Perhaps surprisingly, Solidity's compiler *does not* check that such function is actually defined in the target contract. In this latter case an exception is thrown, making users waste money with no possibility of reimbursement. This is the reason why we also propose an extension of our calculus, FS$^+$, detecting and ruling out this behavior. The key idea is letting a contract specify (in a fine-grained manner) the intended interface of the contracts interacting with it. Bearing in mind that programmers hate writing programs in tedious and verbose languages, in the design of FS$^+$ we strive to keep the syntax as simple as possible, making, on the other hand, the compiler quite more sophisticate and complex. The resulting language leaves the way of writing contracts almost unaltered. As a pleasant side-effect, FS$^+$ functions have a precise knowledge of the contracts invoking them, and thus can make assumptions on their behavior. This means that FS$^+$ contracts invariants can be proven much easier than before, avoiding the possibility of undefined, or unexpected, behaviors.

1.1 Organization of this document

The rest of this document is organized as follows:

- Chapter 2 gives an overview on the concepts treated by this thesis: blockchain, Ethereum, and Solidity.
- Chapter 3 provides a survey on the recent related work.
- Chapter 4 defines the syntax of our calculus, Featherweight Solidity.
- Chapter 5 defines the run-time syntax of FS as well as its operational semantics.
- Chapter 6 formalizes the type system, stating and proving the theorem of Type Safety.
- Chapter 7 elaborates on the current limits of Solidity and proposes an extension, discussing the pros and cons of adding it to Solidity. This chapter also formalizes the modifications to the type system and proves the safety of the latter.

- Chapter 8 concludes with some proposal of further work and personal considerations.

Chapter 2

Background

This chapter gives a background on the topics of this thesis. The focus of this part is very much on the presentation of the key, relevant, characteristics of Ethereum and Solidity, with additional information about transactions. We shall therefore go through it using examples and interaction patterns.

2.1 Transactions

The word "**transaction**" has been given many definitions over the last years, strongly depending on the context. In general, it can be thought of as a sequence of operations that forms a single step, transforming data from one consistent state to another. Such definition implies that a transaction always ends either successfully or not. In the former case, any changes made by the transaction itself are safely stored on persistent storage, and are thus recoverable even if something goes wrong later on. Such operation is often referred to as "**commit**". In the latter case, on the other hand, no changes are made to the current state, thus leaving it as if the transaction had never run ("**abort**"). It is also important to make sure that no transaction can see other transactions' uncommitted changes, so that its result will not depend on possibly aborted partial results. We can thus point out, a bit more formally, the four fundamental properties of transactions, the so-called ACID properties:

- **Atomicity**. Transactions execute with an "all or nothing" semantics: changes are either all committed or all canceled;
- **Consistency**. Transactions always leave the system in a consistent state, without any uncommitted partial results. Sometimes *eventual consistency*, a weaker property, is achieved: it may happen that the system is in an inconsistent state, but eventually all the changes will be propagated. This is usually the case in systems giving more importance to scalability;
- **Isolation**. Transactions never have a chance to see uncommitted changes. In other words, they run without interference from other transactions;
- **Durability**. Committed changes are always guaranteed to persist system fails, and are thus recoverable.

Ensuring such properties in a distributed system is generally hard, and many algorithms have been studied to this end. Generally, a consensus system is required, in

order to get the changes correctly propagated to all the nodes. This is even more complex when there is no centralized authority, since an arbitrary number of nodes have to agree on what to do and when to do it. Blockchain falls into this latter category.

2.2 Blockchain

A **blockchain** is basically a continuously growing database distributed over many computers (the so-called nodes). It is a digitized, decentralized, public ledger of all cryptocurrency transactions: the innovation represented by the word blockchain is the specific ability of this network database to reconcile the order of transactions, even when a few nodes on the network receive transactions in various order. This is done with a combination of four main technologies:

- **peer to peer networks**: nodes can send messages to each other without the need of a centralized authority managing such messages. In this way there is no single point of failure;
- **asymmetric cryptography**: these messages are encrypted in such a way that anyone can verify the sender's authenticity, but only intended recipients can read the message contents;
- **cryptographic hashing**: allowing the creation, for any data, of a small fingerprint to make sure the data has not been tampered with;
- **distributed consensus**: allowing nodes in the network to collectively agree on a set of canonical updates to the state of the ledger. Nodes can freely enter into the consensus process, solving the political problem of deciding who gets to influence the consensus. It does so by substituting a formal barrier (e.g. being on a particular list) with an economic barrier. The latter can be determined in various ways, and two have been so far the main alternatives:
  - *proof of work*: the weight of a single node in the consensus voting process is directly proportional to the computing power that the node brings, and
  - *proof of stake*: the weight of a node is proportional to its currency holdings.

In 2009 Satoshi Nakamoto combined the former three elements and added the latter to create Bitcoin [34].

2.2.1 Blocks

A **block** in a blockchain is basically composed of four main elements:

- a block number;
- the nonce, a numeric value we will explain shortly;
- the previous block's hash, which is basically a pointer pointing to the previous block in the blockchain and used for the sake of consistency;
- some data.

Mining Mining a block is essentially the process of adding some data to the block to be included in the blockchain and finding a nonce satisfying a certain condition, which is largely dependent of the actual blockchain instance. Figure 2.1 depicts a simple two-blocks long blockchain. The `data` field is filled depending on the application and the purpose the blockchain is being used for. Note how the `prev` field in $Block_2$ points back to the hash of $Block_1$. Also note that both the two hashes begin with four zeros.

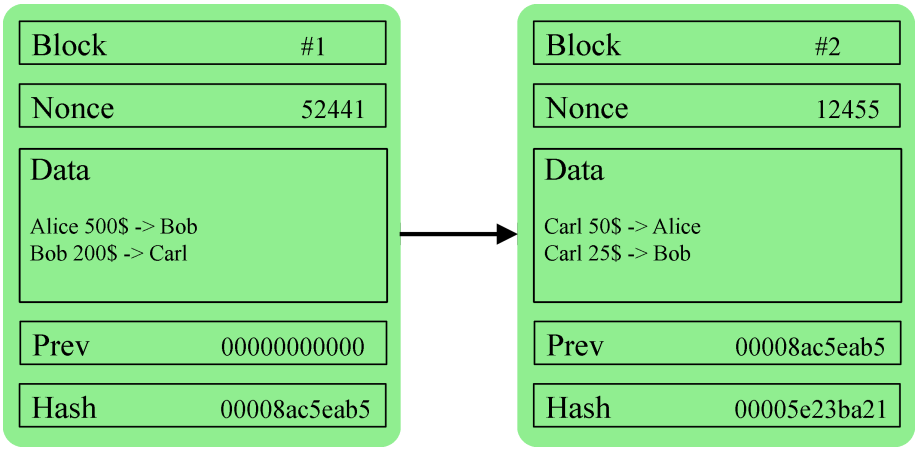

Figure 2.1: Example of a correct two-blocks long blockchain

This is a practical example of what we meant with "certain condition" above, and shows what the nonce is used for. The miner computes this numerical value to make the block's hash begin with four zeros. The search for a suitable value may take a long time, and is by far the most computational-intensive part of the entire mining process, since many hashes might have to be computed before finding the one satisfying the four-zeros-head condition. Once such nonce is found, it is saved into the block itself. Afterwards, all the other miners in the networks have to validate the very same block to make it actually appended to the chain. It is clear that the more blocks are added after a certain $Block_n$ the harder is to tamper $Block_n$ with. The explanation is that an attacker would have to change $Block_n$, recompute its hash (thus finding a suitable nonce for that block), and then repeat this process for all the subsequent blocks. If the networks is sufficiently large, this is extremely computational expensive and, even if theoretically feasible, practically unfeasible. An attacker would need the 51% of the entire network's computational power to make such an attack successful (or to control the 51% of the network) and, again, if the network is sufficiently large, this is all but easy to accomplish. This way of mining is what we called "proof of work" above.

Blocks in Figure 2.1 are correct and are indeed green-colored. Figure 2.2, on the other hand, shows an incorrect blockchain. Now Trudy, an attacker, has attempted to modify $Block_1$ making the two transactions transfer money to her account. This has caused a change in $Block_1$'s `hash`, which does not match with $Block_2$'s `prev` anymore. Hence, the latter is not a valid block, and neither is the blockchain. Trudy has to remine $Block_2$, too, and the more the blocks on top of $Block_1$ the more difficult is for Trudy to cheat.

Normally, blocks have to be mined at a given mining rate. To ensure so, the challenge is continuously adapted using another parameter, the difficulty. In our small example we can imagine it as the number of zeros at the beginning of the hash, which

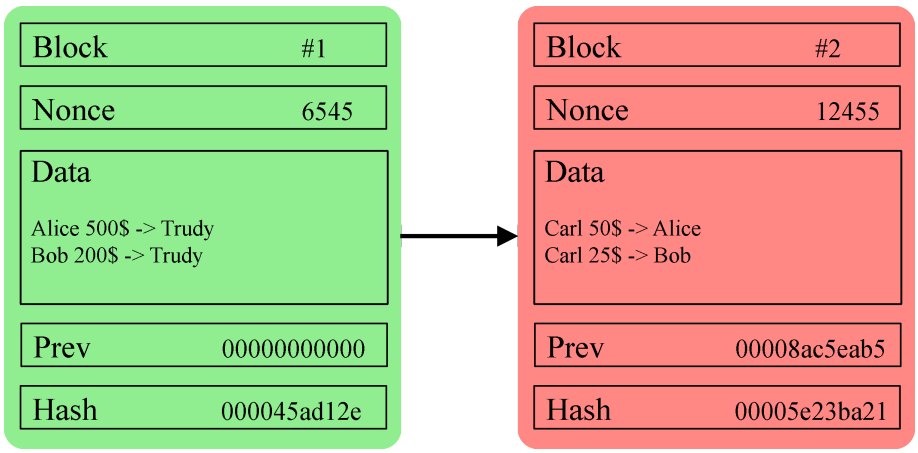

Figure 2.2: Example of an incorrect two-blocks long blockchain

is continuously increased or decreased to make the mining ratio constant. When blocks are mined too fast, the number of head-zeros is increased, thus making it harder to find a suitable nonce. Vice versa, when the ratio is too slow, the number of head-zero is decreased. Miners are given a compensation (measured in cryptocurrency) for their computational effort.

State transition function Figure 2.1 and Figure 2.2 show an arrow linking $Block_1$ and $Block_2$. We can, in fact, imagine the ledger of a cryptocurrency (such as Bitcoin) as a state transition system, where the "state" consists of the ownership status of all existing money and a "state transition function" that takes a state and a transaction and outputs a new state which is the result. Such function can be thought of as follows:

`APPLY(S, TX)` $\rightarrow$ `S' or ERROR`

A simple `ERROR` condition can be an insufficient balance to accomplish a given transaction (e.g. Alice's balance is $<$500 in Figure 2.1). Under this point of view, a block contains a list of transactions that can take the current state to either another valid state or raise an error.

Block validation After being mined, blocks have to be validated by all the other nodes in the network. The validation algorithm (for $Block_n$) roughly works as follows:

1. Check the validity of $Block_{n-1}$, which means checking if it exists and if its `hash` matches with $Block_n$'s `prev`.
2. Validate the proof of work (i.e. the challenge). In our toy example, this step means computing the $Block_n$'s hash and checking if it begins with four zeros.
3. Let `S[0]` be the state at the end of $Block_{n-1}$.
4. For each transaction ($\texttt{TX}_1 \ldots \texttt{TX}_m$) in $Block_n$, let `S[i]=APPLY(S[i-1],` $\texttt{TX}_i$`)`. If any application returns `ERROR`, stop and return false.
5. If every transaction in $Block_n$ returns a valid state, return true.

If such algorithm returns true, then the block is considered valid and appended to the blockchain. If any of the nodes on the network returns false, then the block is not included, since all the nodes must agree.

Stale blocks Consider two miners, A and B, mining, *at the same time*, a block. If A is faster than B, its block will be propagated before B's one, thus making the latter end up wasted without any contribution to the blockchain. B's block is said to be **stale**, since it has actually been mined, but it is discarded because it arrived late. Blockchains with fast confirmation times currently suffer from reduced security due to a high **stale rate**. In fact, there is a centralization issue: if miner A is a mining pool more powerful (i.e. faster) than B, A will have a risk of producing a stale block lower than B, since it can mine blocks at a higher rate. Thus, if the block interval is short enough for the stale rate to be high, A will be substantially more efficient simply by virtue of its size. With these two effects combined, blockchains producing blocks quickly are very likely to lead to one mining pool having a large enough percentage of the network power to have de facto control over the mining process.

2.3 Ethereum

Ethereum [9] is a decentralized platform that runs programs called **smart contracts**: applications that run exactly as programmed without any possibility of downtime (i.e. there will always be at least a machine capable of running them), censorship, fraud or third-party interference. Contracts are written in a low-level Turing-complete bytecode language, running in a virtual machine called Ethereum Virtual Machine (EVM) [53]. Contracts are roughly similar to classes in the object-oriented programming paradigm: they comprise a state and a set of functions, all defined by a sequence of bytecode instructions.

2.3.1 Comparison with Bitcoin

Bitcoin also has a scripting language allowing the creation of a raw and weak version of smart contracts. It is out of the scope of this work to provide details about Bitcoin's scripting language, but it is worth to point out the main differences between it and Ethereum's. The main limitation is that the former lacks Turing-completeness. In particular, no loops can be written, just to avoid infinite loops during transaction verification. This is easily (and theoretically) overcame for script programmers, since the underlying code can be replicated many times with an if statement, but such a solution does create very space-inefficient scripts. Secondly, in Bitcoin there is no way for a script to provide fine-grained control over the amount that can be withdrawn, and no state can survive over two, or more, transactions. This is a big limitation, since such a language only enables the creation of one-off contracts. Lastly, scripts cannot see any blockchain data, such as the nonce, the timestamp or the previous block's hash. Ethereum, on the other hand, provides a Turing-complete language enabling the creation of stateful and blockchain-aware contracts. Several high-level programming languages compiling to bytecode are provided: the most popular of such languages is Solidity [48]. We shall go deeper into its details later on in this chapter.

2.3.2 Accounts

Two are the kinds of **accounts** in Ethereum: externally owned accounts (EOAs), controlled by public-private key pairs (i.e. humans or computers), and contract accounts, controlled by the code stored together with the account itself. The address of an external account can be recovered from the public key, whereas the address of a contract is determined at the time the contract is created (it is derived from the creator's address and the number of transactions sent from that address, the so-called "nonce"). In both cases, the pair *(account, address)* is unique: an address corresponds to one and only one account, and vice versa. These two types of accounts are treated equally by the EVM, and they both have a balance in Ether (Ethereum's cryptocurrency) that can be modified by sending transactions containing Ether. Accounts also have a **storage area** (initially empty and further discussed in Section 2.4.2) and a **nonce** (not to be confused with the one of blocks in a blockchain) used to make sure each transaction is processed only once. In the context of Ethereum, transactions can carry Ether and binary data. The recipient (or target) may contain code (i.e. it is a smart contract and not an EOA); if it does, the code is executed using the binary data as input. The target address may be `0`, in which case a new contract is created. Any transactions are fired from externally owned accounts and the execution needs to be completely deterministic: its only context is the position of the block on the blockchain (i.e. the block number) and the data sent along with the transaction. Instances of different smart contracts may communicate via **messages** in the context of a transaction. Messages can be thought of as function calls, and they are virtual object that are never serialized and exist only in the Ethereum execution environment. They are essentially like transactions, and contain the same fields (see below); the only exception is that they are produced by contracts and not by external actors.

2.3.3 Transactions

Every Ethereum's transaction contains the following fields:

1. the target (i.e. the recipient account);
2. a signature identifying the sender (i.e. the caller account);
3. `VALUE`, the amount of Ether to be transfered to the recipient;
4. an optional data field;
5. `STARTGAS`, the maximum number of computational steps the execution of this transaction is allowed to take;
6. `GASPRICE`, the amount of Ether the caller is willing to pay for each unit of gas.

Gas Every transaction is charged, upon creation, with a certain amount of **gas**, the fuel of Ethereum's blockchain. Any bytecode instructions come with a gas fee ([53]) specifying how much it costs to execute that operation. The account initiating the transaction (caller) has to specify the maximum amount of gas (`STARTGAS`) and the price (`GASPRICE`) it is willing to pay. Hence, the total fee is computed as `GASPRICE * STARTGAS`. The caller is actually paying the miner, in advance, for the computation, regardless of whether the transaction succeeds or fails. This mechanism is also used to prevent infinite loops: any operations executed by the EVM decrease the `STARTGAS`

and, if it reaches 0, an `Out of Gas` exception is thrown to immediately block the execution. The transaction fees go to the miner who mines the block containing the transaction. When miners mine a block they have to decide the transactions to include in it. They can choose to include no transactions, or they can choose to randomly select them. Most miners follow a very simple strategy: they first sort the received transactions from the highest `GASPRICE` to the lowest; secondly, they include them until either the block is full or they reach one that has a `GASPRICE` set lower than they are willing to bother with. Hence, in general, the higher the `GASPRICE` the higher the probability to have the transaction included in the next block.

State transition function Transactions in Ethereum are clearly more complex than the ones we defined before: they do not simply represent an exchange of money from two accounts, but involve a number of fields and possibly code. Thus, the state transition function (`APPLY(S, TX)` $\rightarrow$ `S' or ERROR`) changes accordingly, as described below:

1. Check if the transaction is well-formed (i.e. all the required fields are defined), if the signature is correct and if the nonce matches the sender's one.
2. Calculate the transaction fee as `STARTGAS * GASPRICE`, and determine the sending address from the signature. Subtract the fee from the sender's account balance and increment the sender's nonce. If there is not enough balance to spend, return an `ERROR`.
3. Let `GAS = STARTGAS` and subtract an amount of gas corresponding to the number of bytes composing the transaction.
4. Transfer the `VALUE` from the sender to the recipient. If the latter does not yet exist, create it; if it is a contract, run its code either to completion or until the execution runs out of gas. (We shall see what code is actually run shortly.)
5. If the value transfer failed (either due to a sender's insufficient balance or because the code execution ran out of gas) revert all state changes except the payment of the fees, and add the fees to the miner's account.
6. Otherwise, refund the fees for all remaining gas to the sender, and send the fees paid for gas consumed to the miner.

It is clear that miners are paid even if the transaction is actually reverted. It is one of the goals of this work to enhance Solidity's type system in order to detect, at compile-time, some of the errors that may occur at run-time.

2.3.4 Blocks and validation

Blocks in Ethereum are quite similar to the ones we described before, with two key differences. First, two more values, the **difficulty** and a **timestamp**, are stored in the block itself. Secondly, blocks do not contain only a transaction list, but also the most recent state (i.e. all the active accounts together with their balance and their code, if any). This might seem space-inefficient, but Ethereum uses a particular data structure, known as Patricia Tree, to handle such a blockchain efficiently [10]. We can now give the algorithm to validate a block ($Block_n$) in Ethereum:

1. Check the validity of $Block_{n-1}$.

2. Check that the timestamp of $Block_n$ is greater than that of $Block_{n-1}$ and less than 15 minutes into the future.

3. Check the validity of some low-level fields (such as block number, difficulty, gas limit, and the root of the tree).

4. Validate the proof of work.

5. Let `S[0]` be the state at the end of $Block_{n-1}$.

6. For each transaction ($\texttt{TX}_1 \ldots \texttt{TX}_m$) in $Block_n$, let `S[i]=APPLY(S[i-1],` $\texttt{TX}_i$`)`. If any application returns `ERROR`, stop and return false. Return false also if the total gas consumed so far reaches `STARTGAS`.

7. Let $\texttt{S}_f$ be equal to `S[m]`, but adding the payment to the miner.

8. Check if the root of the tree in $\texttt{S}_f$ is equal to the one stored in $Block_n$. If so, the block is valid; otherwise it is not.

In Ethereum, the proof of work is slightly different than the four-zeros one we defined before. In fact, the hash of each block, seen as a decimal number, has to be lower that a given value, identified by the difficulty field, stored along with the block's header.

In brief, given a block of successful transactions (mined by a miner node), if any of them returns `ERROR` the validation process fails and the entire block is discarded. Note that, even if every validation node executes `APPLY` (i.e. runs the transaction again) the fee (payed by means of gas) is given only to the miner node the first time the transaction is carried out. That is, every change to the balance of the accounts involved in the transaction, as well as the miner's one, are only checked to be sure they are valid. No money transfers are performed at validation time.

2.3.5 Code execution

The aim of Ethereum is to allow programmers to define smart contracts. The latter are very similar to classes in object-oriented programming languages and generally model the back-end of a bigger application. This implies that the interaction between the "external world" (i.e. everything outside the Ethereum's blockchain) and smart contracts, and among smart contracts themselves is very important. At first this may sound a bit confusing, since many parties play a role in a "standard" Ethereum application. To clarify, suppose we want to develop a rock-paper-scissors game, modeled by a contract named `RPS`: when a player makes a choice they are charged a fixed amount of Ether and wait for the next player to make the second choice. The winner takes the entire amount reduced by a small fee payed to the contract `RPS`. Below, we outline the steps required to build such an application:

1. we write the smart contract `RPS` using one of Ethereum's high-level languages. The most known is Solidity, but many others are available (e.g. Serpent or LLL);

2. we compile down, using one of Ethereum's libraries, our contract `RPS` into bytecode and send it within a transaction to deploy the contract. Note that the bytecode is necessary in order to accomplish the deploy. An instance C of `RPS` is now on the blockchain waiting for someone to interact with it;

3. an Ethereum account A_1 (no matter if it is an EOA or a contract) wishes to play with C. It invokes a function to send its choice along with the amount of Ether to be paid, and waits for a second player to make a move;
4. when another Ethereum account A_2 makes the second move, C is able to figure out the winner (either A_1 or A_2) and to pay it back the award reduced by a small fee (kept in the C's balance);
5. optionally, a designated account A_3 with more privileges (e.g. the one which deployed the instance of `RPS`) can withdraw Ether from C's balance.

You can see different parties playing different roles in this sequence of steps. First, we, as programmers, develop `RPS` (for example in Solidity). Secondly, we write code to deploy it (such code may be written in many programming languages, from JavaScript to Haskell). The latter will run locally to our own computer or server, without the need to store a copy of the blockchain or to be a miner. The transaction send to deploy an instance of `RPS`, on the other hand, will be carried out by an Ethereum miner. Note that there is no assurance that this transaction will be executed immediately: miners can choose what to include in the block they are mining. Optionally, the design of our contract may keep track of the account deploying it (i.e. A_3), in order to let it withdraw the money (Step 5). A_1 and A_2 are totally not known at compile/deploy-time, and vary during the lifetime of C (i.e. two players play the first time, but then other two players can play again, without even knowing about the former two). They can be either contracts or EOAs, and the functions of `RPS` invoked to make their move will be executed, again, by a miner. Later in this Chapter, and precisely in Example 2.1, we shall show how to define a smart contract, how to deploy it and how to interact with it.

To sum up, we explicitly give an answer to two crucial questions: what happens when the code is executed and where does it run? The latter is simpler: if a transaction is added into $Block_n$ the code execution spawned by that transaction will be executed by all nodes, now and in the future, that download and validate $Block_n$. At first, a miner node will chose to add the transaction in a block and will execute it. Then, every node validating the same block will execute the same transaction to validate it. The former question is a bit more complex. First, code written in high-level languages, such as Solidity, is compiled down into bytecode (EVM code). Such bytecode consists of a series a operations working on three memory spaces:

- the **stack**;
- the **memory**, an infinitely expandable byte array;
- the contract's long-term **storage**, which is basically a list of key/value pairs. Unlike stack and memory, which are volatile and reset when, respectively, the transaction ends or a function returns, storage is persistent.

The code can access information regarding the current sender, the value, the current block and transaction. EVM's computational state can be defined by a tuple `(block_state, transaction, message, code, memory, stack, pc, gas)`, where `block_state` is the global state containing all accounts and includes balances and storage. The current instruction is found by taking the byte of `code` corresponding to the program counter `pc` (or 0 if `pc >= len(code)`), and each instruction has its own definition in terms of how it affects the tuple (number of push and pops on the stack, modifications to memory and storage, and so on) [53].

We can now take a closer look to Solidity, to understand how smart contracts are defined.

2.4 Solidity

Solidity is a statically-typed programming language, whose syntax has been strongly inspired by JavaScript, enabling programmers to define smart contracts. Their definition resembles that of classes in the object-oriented programming paradigm. However, programming in Solidity (and, more in general, defining smart contracts) presents some thorny points to pay attention to. We will not list any security or programming best practices, but smart contracts manipulate money, and clumsy implementations make room to irreversible bugs that may have serious consequences. An example is the contract The DAO [47]. It implemented a crowd-funding platform, which raised more or less $150M before being attacked on June 18th, 2016 [39]. An attacker managed to put about $60M under their control, until the hard-fork of the blockchain nullified the effects of the transactions involved in the attack. In the context of blockchain, a hard-fork is a rule change such that the software validating according to the old rules will see the blocks produced according to the new rules as invalid. In case of a hard-fork, all nodes meant to work in accordance with the new rules need to upgrade their software.

Contracts may contain state variables, functions, function modifiers, events, struct types, and enum types. Contracts may also inherit from other contracts. Example 2.1 shows a simple, yet complete, example of smart contract written in Solidity.

Example 2.1 (Simple smart contract)**.**
Listing 2.1 lists a basic smart contract modeling a person.

```
1  contract Person {
2      struct Address {
3          string city;
4          string street;
5      }
6
7      string public name;
8      uint public birthyear;
9      Address public addr;
10     bool public gender; // false = male, true = female
11     address public owner;
12
13     modifier ownerOnly {
14         require(msg.sender == owner);
15         _;
16     }
17
18     event NewAddress(Address _address);
19
20     constructor (string _name, uint _year, string _city,
21         string _street, bool _gender) public {
22         name = _name;
23         birthyear = _year;
24         addr = Address({city: _city, street: _street});
25         gender = _gender;
26         owner = msg.sender;
27     }
28
29     function setAddress(string _city, string _street) public
           ownerOnly {
30         addr = Address({city: _city, street: _street});
31         emit NewAddress(addr);
32     }
33 }
```

Listing 2.1: Person contract in Solidity

First, we define a struct `Address` containing the details of an address, the `city` and the name of the `street`. Secondly, all the state variables are listed: the `name` of the person, their `birth-year`, address (`addr`), and `gender`. We then add a variable of type `address`, `owner`, which is used as a very basic form of access control. The constructor initializes it with the address of the account instantiating the contract (stored automatically in `msg.sender`, an implicit variable set by the runtime), and subsequently read in the modifier `ownerOnly`. The `require` statement requires the `owner` to be equal to the current `msg.sender`. At run-time, the special symbol `_;` will be replaced with the body of `setAddress()`. In this contract, the only function using such access control mechanism is `setAddress()`, which allows the `owner` to change their address. This function also `emits` an event, `NewAddress`, actually logging the address change in the blockchain. An event allows a contract to signal that something particularly relevant has just happened. In this case, an instance of `Person` signals everyone monitoring the blockchain that its `address` has changed. This allows for an event-driven programming paradigm, since external applications may be monitoring the blockchain and subsequently react to events of interest.

It is now time to deploy the contract. To this end, we can use either Remix, the Solidity official on-line editor[1], whose explanation goes beyond the scope of this example, or an Ethereum compatible library. Here we shall use the JavaScript implementation, web3.js[2], but many Application Program Interfaces (APIs) exist for other popular programming languages, such as Web3.py[3] for Python, hs-web3[4] for Haskell, web3j[5] for Java, and web3j-scala[6] for Scala. Contracts can also be deployed directly in Solidity code, using `new` (e.g. `new Person(``Matteo Di Pirro'', 1994, ...)`). In this case, if contract `A` is instantiating contract `B`, the code of `B` must be know by the EVM when an instance of `A` is deployed (i.e. the code of `B` have to be deployed together with `A`'s).

We shall show how to use web3.js to deploy a contract and interact with it. Contracts are just like objects, and many different instances of the same contract may be on the blockchain at the same time, similarly to what happens for class instances in object-oriented programs. However, interacting with a deployed contract is slightly more complex than just using an object. First consider Listing 2.2, showing how deploy is done, supposing the smart contract code is stored in `Person.sol`.

```
1  // Compilation
2  var code = fs.readFileSync('Person.sol').toString(); // reads the
       source code
3  var solc = require('solc');
4  var compiledCode = solc.compile(code);  // and compiles it
5
6  /*
7  compiledCode.contracts[':Person'].bytecode contains the code which
       will be deployed
8
9  compiledCode.contracts[':Person'].interface contains the interface
       or template of the contract (called abi) telling the contract
       user what methods are available in the contract.
10 */
11 var byteCode = compiledCode.contracts[':Person'].bytecode;
```

[1] https://remix.ethereum.org/
[2] https://web3js.readthedocs.io/en/1.0/
[3] https://web3py.readthedocs.io/en/stable/
[4] http://hackage.haskell.org/package/web3
[5] https://web3j.io/
[6] https://github.com/mslinn/web3j-scala

```
12 var abi = JSON.parse(compiledCode.contracts[':Person'].interface);
13
14 var myAccountAddress = '0x...';
15
16 // Deploy
17 var Person = web3.eth.contract(abi);
18 var deployedContract = Person.new(
19     'Matteo Di Pirro', '1994', 'Padua', 'Via Trieste', false,
20     {data: byteCode, from: myAccountAddress, gas: 4700000,
           gasPrice: 200000000}
21 );
22
23 // deployedContract.address contains the address of the contract
       just deployed
24
25 var contractInstance = Person.at(deployedContract.address);
```

Listing 2.2: Deploying an instance of Person

This short listing contains a lot of things to analyze. Let us start with Lines 2-4, that simply read the contract source code and compile it, with the additional JavaScript library `solc`. After that, the variable `compiledCode` has two fundamental fields: `bytecode`, containing the actual bytecode resulting from the compilation (Line 11), and `interface` (Line 12). The latter represents the Application Binary Interface (ABI) of the contract, that basically is a JSON (JavaScript Object Notation) object representing the set of functions and events exposed by the contract. `myAccountAddress` (Line 14) defines the address of the account we are using to interact with the blockchain. We can now deploy an instance of `Person`, first interpreting the ABI interface as a JSON object and then using it as a parameter for the function `web3.eth.contract`, that returns a contract "class" representing our contract (Line 17). We use it to create a concrete `Person` object (i.e. a deployed contract) via the `new` method (Line 18). Here we also specify the parameters required by the constructor of the contract (Line 19) and then additional parameters necessary to the deploy (Line 20). In particular, `data` contains the bytecode of the contract to be deployed, `from` is the account initiating the transaction (i.e. the sender). This is because the blockchain has to keep track of who deployed the contract. Lastly, `gas` is the amount of gas we are willing to pay to deploy the contract. The balance in the `from` account will be used to buy gas, whose price is set by `gasPrice`. Finally, Line 25 shows how to get the contract instance by its address (which is the one of the contract we have just deployed).

Clearly, every time this snippet of code is executed a new instance of `Person` is deployed on the blockchain. Note that, in order to deploy the instance, the bytecode is necessary. In this example we defined `Person` from scratch, but we could actually reuse the source code of a contract we did not write. In fact, many smart contracts make publicly available their source code[7], which can be copied and reused to deploy other instances, by anybody.

The last thing we need to understand is how to interact with a deployed contract using web3.js. To this end, suppose we want to use a contract we **did not** deployed by ourselves. Listing 2.3 shows how to do it.

```
1 var abi = JSON.parse('abi'); // parse the abi as before
2 var Person = web3.eth.contract(abi); // get a contract object
3 var contractInstance = Person.at('0x...'); // get an instance by
      its address
```

[7]Visit `https://etherscan.io/contractsVerified` for a list of deployed Ethereum smart contracts

```
4  var myAccountAddress = '0x...';
5
6  // parameters used to interact with the smart contract
7  var transactionObject = {
8      from: myAccountAddress,
9      gas: 4700000,
10     gasPrice: 200000000
11 };
12
13 // JavaScript functions to interact with `contractInstance`
14 function getPersonName() {
15     contractInstance.getName(transactionObject, function(error,
           result) {
16         if(!error) {
17             // do something with `result`
18         } else {
19             // handle `error`
20         }
21     });
22 }
23
24 function changePersonAddress(city, street) {
25     contractInstance.setAddress(city, street, transactionObject);
26 }
```

Listing 2.3: Interacting with an instance of Person

Lines 1-3 are more or less as before. Given an ABI definition we obtain a contract instance knowing its address. Note, however, that now both the ABI and the address are hard-coded in the source code. As we said, `Person` has previously been deployed, thus the instance has got a valid address (e.g. `deployedContract.address` in Listing 2.2) and its ABI is accessible at `https://etherscan.io/`. As before, we know our own account address and we store it in `myAccountAddress`. Hard-coding in this way our address is not the only possible way: web3.js provides a property, `web3.eth.defaultAccount`, which will work as a default value whenever the `from` property is missing (see, for instance, Line 20 of Listing 2.2). `transactionObject` contains the parameters we will use when invoking functions on `contractInstance`, and is here declared to avoid code duplication. We are now ready to show how to communicate with a smart contract. To this end, we define two JavaScript functions, `getPersonName()` (Line 14) and `changePersonAddress()` (Line 24), to interact with `Person`. This code is usually part of the front-end of a web application, and thus generally manipulates things on a web page. In this case, for instance, `getPersonName()` could, in case of success, display the name on the page. Similarly, the actual parameters of `changePersonAddress()` could come from a form filled in by the user. Lastly, note how `getPersonName()` uses the JavaScript's mechanism of callbacks (Line 15) to wait for the result of `getName()`. This would happen also for `changePersonAddress()`, but we did not specify any callbacks because we are not interested in the result of this invocation. We shall see, at the end of this chapter and with a bigger understanding of Solidity, that function calls are not treated always in the same way. Finally notice that the function's actual parameters are specified before the transaction's ones, which are in turn specified before the callback. This also means that, in this example, each JavaScript function causes an Ethereum transaction.

Please note that, for the sake of simplicity, this example showed minimal snippets of code using web3.js. The main aim was not to explain the details of how this library works, but instead to give an overview to better understand the choices we made while

formalizing a core calculus of Solidity. Many of the methods shown here, such as the way of invoking functions of a smart contract, have many alternatives, each of which is best suited for different scenarios. Furthermore, many frameworks simplifying the programming pattern exist[8]. They provide built-in functions to compile, deploy, manage, and test smart contracts and their interactions.

This simple example pointed out some peculiarities of Solidity. First, getter functions are automatically added for each `public` state variable. We shall talk more in details about visibility modifiers later on in this chapter. Note that Solidity only adds getters, and not setters. The latter have to be explicitly defined by the programmer, if necessary. Secondly, modifiers can be used to restrict the ability of invoking some functions. In Example 2.1 we allowed only the owner to change their address, but modifiers can be used to implement more advanced restrictions. They can also accept parameters, such as functions, so that they do not only depend on the implicit variable `msg`. We shall explain more in detail `msg` below, but for now it can be thought of as a special variable containing information about the received message, such as the address of its sender or the amount of Ether it contains. We here make use of `require`, which is a short form for `if (msg.sender != owner) revert();`. `require` inputs a boolean expression which, if false, throws an exception. We will go into exceptions below. Thirdly, events are used to explicitly log something on the blockchain. This might seem unuseful here, but many external applications monitor the blockchain looking for events they can react to. Lastly, note the special syntax used to initialize a variable of a struct type (`addr` in this case). We use a notation that reminds the one used to deal with JSON objects, with every field of the struct explicitly associated with its value. In this way the order of the actual parameters is not important, and we could have written `Address(street: _street, city: _city)` without any changes in meaning. We could also have used a more "natural" notation, setting `addr` as follows: `addr = Address(_city, _street)`, but it is order-dependent and discouraged by the official documentation.

2.4.1 Implicit variables

Solidity provides a number of implicit variables and functions. The latter have to do, for instance, with mathematical and cryptographic tasks. The former, on the other hand, provide useful information about the current transaction and the current state of the blockchain (remember that Ethereum is a blockchain-aware platform, as said before). Three are the main implicit variables, whose scope is the contract definition:

- `msg`, storing information about the current message. As we said, accounts communicate by sending each other messages containing two main information: `sender`, the address of the account sending the message, and `value`, the amount of Wei that has been sent. Wei is the smallest sub-currency of Ether (1 Ether = 10^{18} Wei). This variable is first set when a transaction begins (i.e. `sender` contains the address of the account initiating the transaction) and is updated any time a contract invokes a function belonging to another contract. The amount of gas is also readable, via the implicit function `gasleft()`;

- `block`, storing information about the current block, such as its `number`, `gaslimit`, `difficulty`, the `timestamp`, and the address of its miner

[8] An example is Truffle, `https://truffleframework.com/`

(`coinbase`). Programmers can also read a given block's hash, via the function `blockhash(uint)`, specifying the number of the required block. Bearing in mind that miners can decide which transactions to include in a block and when, "current block" refers to the block the transaction is being included in, and hence the values of the fields of `block` vary based on when the miner executes the transaction.

- `tx`, storing information about the current transaction: its `gasprice` and the address of the account that has started it (`origin`). Remember that a transaction begins when an externally owned account invokes a function on a contract, whereas any interactions among contracts cause a message call. Hence, `origin` never changes during the execution of the transaction itself, and is equal to the sender of the first message call. As we will point out shortly, fields of `msg` can change due to particular function calls, but `tx` never does.

2.4.2 Types

Solidity is a statically-typed language, which means that variables are given a type at compile-time. Such type can be either a value type or a reference type. The former is named after its call-by-value semantics: variables are always copied when they are used as function arguments or in assignments. The latter comprises all those complex types whose copy can be expensive, and is associated with two memory areas: memory (not persisting) and storage (persisting). Every variable of a reference type has to be annotated with its location using the keywords `memory` or `storage`. Depending on the context, there is always a default value. Data locations are important because they change how assignments behave: assignments between storage and memory and also to a state variable (even from other state variables) always create an independent copy. Assignments to local storage variables only assign a reference though, and this reference always points to the state variable even if the latter is changed in the meantime. On the other hand, assignments from a memory stored reference type to another memory-stored reference type do not create a copy.

Storage vs memory The key difference to keep in mind is that memory is temporary, whereas storage is persisting. For example, one would perform intermediate computations using memory, and then save the result to storage. The latter is really expensive: it costs 20000 gas to set a storage location from zero to a value, 5000 gas to change its value, and 200 gas to read a word. The reason is that a contract's storage values are stored on the blockchain forever, which has a real-world cost. Hence, storage has to be used when its really mandatory, that is, when values have to be persisted among different contract calls, in the same way as state variables are. The storage of one contract can also be read by another contract, or by querying the blockchain. Memory is much cheaper than storage: it costs 3 gas to read or write a word, plus some gas if we are expanding memory. For a few KB it is very cheap indeed, but the cost goes up quadratically the more we use: a MB of memory will cost a couple of million gas. The drawback is that this area is only accessible during the contract execution: when the latter is over, the memory is wiped out. It can be thought of as a general workhorse, and can be used for everything non-permanent.

Listing 2.4[9] lists a contract example clarifying the concept of data location. There

[9]This example is taken from `https://solidity.readthedocs.io/en/v0.4.24/types.html#data-location`

are some keywords we have not seen yet. First, `delete a` is an operator assigning to `a` the default value for its type, for instance `a=0` for integers. It can also be used on arrays, where it assigns a dynamic array of length zero or a static array of the same length with all elements reset. For structs, it assigns a struct with all members reset. Secondly, `var` is a keyword used to let Solidity deduct the type of the variable being initialized. Lastly, `internal` is a function visibility modifiers, and we shall explain its meaning below.

```
1  contract C {
2      uint[] x; // the data location of x is storage
3
4      // the data location of memoryArray is memory
5      function f(uint[] memoryArray) public {
6          x = memoryArray; // works, copies the whole array to
               storage
7          var y = x; // works, assigns a pointer, data location of y
               is storage
8          y[7]; // fine, returns the 8th element
9          y.length = 2; // fine, modifies x through y
10         delete x; // fine, clears the array, also modifies y
11         // The following does not compile; it would need to create
               a new temporary /
12         // unnamed array in storage, but storage is "statically"
               allocated:
13         // y = memoryArray;
14         // This does not compile either, since it would "reset"
               the pointer, but there
15         // is no sensible location it could point to.
16         // delete y;
17         g(x); // calls g, handing over a reference to x
18         h(x); // calls h and creates an independent, temporary
               copy in memory
19     }
20
21     function g(uint[] storage storageArray) internal {}
22     function h(uint[] memoryArray) public {}
23 }
```

Listing 2.4: Reference types and data location in Solidity

Value types Many value types are the usual common ones found in many programming languages, such as `booleans`, `integers`, static `arrays`, `strings`, `enums`, and so on and so forth. Two are the most interesting:

- `address`, holding 20 byte values and representing Ethereum accounts. Values of type `address` have two main properties: `address.balance`, allowing programmers to query the balance of an address, and `address.transfer(unsigned_integer)`, to transfer Ether (in units of Wei) from one address to another. There are more properties, such as `send`, which is similar to, but slightly lower-level than, `transfer`, and `call`, `delegatecall`, and `callcode`, that are low-level alternatives to the "classic" function call. We will need additional information to explain in detail why `send` differs from `transfer`, and we shall do so in Section 2.4.4 and Example 2.5. Consider now the other three properties, and in particular `call`: `a.call(bytes4(keccak256(``fun(uint256) '')), b)` and `a.fun(b)` are totally equivalent in the sense that they both invoke the function `fun` on the contract corresponding to address `a`, passing `b` as an argument

(`keccak256` returns an hash). `call`, anyway, does return `false` if the invocation encounters an exception, and `true` otherwise. As you can see, the former is more complex, offers no type safe, and does not allow the (possible) return value of `fun` to be used. Furthermore, the use of `bytes4(keccak256(''fun(uint256)''))` is necessary since Solidity looks up the function to be invoked using the first four bytes of its hash. `call`, `delegatecall` and `callcode` are to be used only as a last resort, since they break type safety[10] (no interface is available to check what is being invoked). At the time of writing (version 0.4.24), contracts inherit from the address type, thus one can access all the properties stated above using a contract reference instead. For instance, things like the following are accepted:

```
  B b = new B();
int x = b.balance;
 b.transfer(10);
```

An implicit cast `B`→`address` applies in the cases above. This is currently discouraged and the best practice is to *explicitly* make the cast and subsequently access the required property (i.e. `address(b).balance;`). As of version 0.5.0, however, contracts will not derive from `address` anymore, but they can still be converted to it.

- function types, allowing programmers to use functions as first-class values, thus passing them as parameters, assigning them to variables, and returning them from function calls. However, Solidity does not support lambda expressions, and thus functions values can only refer to functions defined in any contract deployed on the blockchain. We have not talked about visibility modifiers yet, but we do give a first explanation about two of such modifiers here: `internal` and `external`. These are the sole two visibility modifiers allowed for function types. Values of the former type can only be called inside the current contract because they cannot be executed outside of the context of the current contract. They are implemented as basic jumps to the function inside the EVM: the current memory is not cleared, and the implicit variable `msg` is not changed. The latter type does not have this restriction, and can be used both through different contracts or in the context of the current contract. They are implemented via a message call to the function, thus causing a change in `msg`. The syntax of a function type is as follows:

```
  function (<parameter types>) internal|external
[pure|constant|view|payable] [returns (<return types>)]
```

We shall dive into all these modifiers in the paragraph about contracts below. Note that a function can return a tuple of values.

We now give some examples to give a better understanding of what we just explained. Example 2.2 goes deeper on the behavior of `call`, `callcode`, and `delegatecall`; Example 2.3 clarifies the behavior of internal and external functions with respect to `msg` and `tx`; lastly, in Example 2.4 we show how to use functions as values. Note

[10] Refer to `https://solidity.readthedocs.io/en/develop/contracts.html#functions` for more details

that Featherweight Solidity (FS), the language we formalize to study the behavior of Solidity, does not take into account the functionality shown in these two examples. The reader interested only in the core features modeled in FS may skip the following part.

Example 2.2 (`call`, `callcode`, and `delegatecall`)**.** Listing 2.5 defines three contract explaining the behavior of `call`, `callcode`, and `delegatecall`. `A` simply wraps these three functions and use them to invoke `B`'s `setN()`. `C` invokes `A`'s `delegatecallSetN()`, which in turn call `setN()` of `E`. Suppose that all the addresses correctly refer to the expected contracts (i.e. when we use `_a` we assume it points to an instance of `A`, and so forth). In this example we shall use capital letters, `A`, `B`, and `C` to indicate, as usual, contract names. For the sake of clarity, we shall abuse the use this letters to identify also contract instances. Hence, when we write "`A calls B`" we mean "any instance of `A calls` an instance of `B`", and when we write "`msg.sender` inside `B` is `A`" we mean "`msg.sender` inside the execution of a certain function of `B` corresponds to an instance of `A`".

```
1  contract A {
2      uint public n;
3      address public sender;
4
5      function callSetN(address _b, uint _n) public {
6          // B's storage is set, A is not modified
7          _b.call(bytes4(keccak256("setN(uint256)")), _n);
8      }
9
10     function callcodeSetN(address _b, uint _n) public {
11         // A's storage is set, B is not modified
12         _b.callcode(bytes4(keccak256("setN(uint256)")), _n);
13     }
14
15     function delegatecallSetN(address _b, uint _n) public {
16         // A's storage is set, B is not modified
17         _b.delegatecall(bytes4(keccak256("setN(uint256)")), _n);
18     }
19 }
20
21 contract B {
22     uint public n;
23     address public sender;
24
25     function setN(uint _n) public {
26         n = _n;
27         sender = msg.sender;
28     }
29 }
30
31 contract C {
32     function f(A _a, B _b, uint _n) public {
33         _a.callSetN(_b, _n);
34     }
35
36     function g(A _a, B _b, uint _n) public {
37         _a.callcodeSetN(_b, _n);
38     }
39
40     function h(A _a, B _b, uint _n) public {
41         _a.delegatecallSetN(_b, _n);
42     }
43
44     function abort() public {
```

```
45          revert();
46      }
47
48      function callAbort() public returns (bool) {
49          return this.call(bytes4(keccak256("abort()")));
50      }
51  }
```

Listing 2.5: `call`, `callcode`, and `delegatecall` in action

`delegatecall` basically says that a contract `A` is allowing (delegating) another contract `B` to do whatever it wants with `A`'s storage. `delegatecall` is a security risk for the sending contract, which needs to trust that the receiving contract will treat the storage well. It was introduced to fix a bug in `callcode`, which did not preserve neither `msg.sender` nor `msg.value`. If `C` invokes `A` who `delegatecalls` `B`, the `msg.sender` in the `delegatecall` is `C` (whereas if `callcode` was used the `msg.sender` would be `A`).

Let us explain so referring to Listing 2.5. When `A calls B`, the code runs in the context of `B`, and `B`'s storage is used. This means that `B`'s `n` will mutate. On the other hand, when `A callcodes B`, the code runs in the context of `A`, as if the code of `B` was in `A`. Whenever the code writes to storage, it writes to the storage of contract `A`, instead of `B`, and thus `A`'s `n` will mutate. Furthermore, when `A callcodes B`, `msg.sender` inside `B` is `A`. When `C` invokes `A`, and `A delegatecalls B`, `msg.sender` inside `B` is `C`; on the contrary, it is `A` if `callcode` is used. In the former case, `B` has the same `msg.sender` and `msg.value` as `A`, which are correctly preserved since `B`'s `setN()` is not marked as `external` (remember that marking a function as `external` modifies the variable `msg` since it generates a new message call).

The following tables sum up the results of various calls. Table 2.1 lists the results on the interaction between `A` and `B`; Table 2.2 shows how `msg.sender` varies. All these calls are made considering variables initialized at zero, that is integers are 0 and addresses are $0x00\ldots0$.

a.callSetN(b, 1)		**a.callcodeSetN(b, 2)**		**a.delegatecallSetN(b, 3)**	
a.n	0	a.n	2	a.n	3
b.n	1	b.n	0	b.n	0
b.sender	a	b.sender	a	b.sender	a

Table 2.1: Interactions between `A` and `B`. When `A` uses `callcode` or `delegatecall`, its own storage is modified instead of `B`'s. On the other hand, `call` works as expected and as a normal function call `b.setN(1)`.

As you can see, `C` contains other two functions, `abort()` and `callAbort()`: the former simply throws a `revert()` and the latter calls it by means of `call` and returns a boolean flag. Note that for the sake of this discussion, the behavior of `call`, `callcode`, and `delegatecall` does not change. So far, all the flags returned by these three functions have always been `true`, to signal that no exceptions were raised. However, when `callAbort()` is invoked, an exception actually occurs, and make the function return `false`. `false` is returned also if the invoked function does not exist.

So far, the behavior of these three functions seems predictable. However, we cheated a bit, calling the state variables of `A` and `B` in the same way and declaring them in the same order. Furthermore, we accurately avoided many important questions: what

c.callSetN(a, b, 1)		c.callcodeSetN(a, b, 2)		c.delegatecallSetN(a, b, 3)	
a.n	0	a.n	2	a.n	3
b.n	1	b.n	0	b.n	0
a.sender	0x0	a.sender	a	a.sender	c
b.sender	a	b.sender	0x0	b.sender	0x0

Table 2.2: Interactions between `C` and `B` using `A` as a man-in-the-middle. When `C` uses `callSetN()` everything is as expected, with only the storage of `B` mutating and the `msg.sender` set to `a`. When it uses `callcodeSetN()` the `mag.sender` is modified in `A` even though the function `setN()` is not `external` (and thus calling it should preserve the original sender). This is solved by using `delegatecallSetN()`, which correctly leaves unaltered `msg.sender`. As seen before, only `callSetN()` operates on `B`'s storage.

would happen if `A` did not define any integer variable `n`? What would happen if `A`'s state variables were declared in a different order than `B`'s? What would happen if the order remained the same, but with different identifiers? In these cases the results would be a lot more surprising than before. The fact is, when `B` writes to `A`'s storage, it does so taking into account its own variable ordering. Consider two contracts with the opposite order of state variables declaration (i.e. `A` declares `sender` and then `n` and `B` does the opposite, first `n` and then `sender`). Suppose we invoke `a.delegatecall(b, 100)` (it would be the same using `callcode`). `B` writes on `A`'s storage, but the latter is organized in a different way, and thus `n` in `A` becomes equal to the decimal representation of `msg.sender`, whereas `sender` in `A` becomes equal to the hexadecimal representation of 100, that is $0x0000000000000000000000000000000000000064$. Note that here the order of variables assignments in `setN()` does not matter. `B` actually makes `A`'s storage dirty, changing not only its state variables but also any other values that have been stored there. What is worse is that programmers do not receive any warnings (nor the function returns `false` to signal that something bad just happened), and could consequently assume that everything went fine even if the storage of their contract is totally messed up. Since the latter is written on the blockchain, there is no "undo" button or easy way out. This is the reason why `call`, `callcode`, and `delegatecall` should be used extremely carefully and only as a last resort. They not only are more complex then "normal" function calls, but are also not type-safe and can seriously damage a contract's storage.

Example 2.3 (Internal vs external functions)**.**
Listing 2.6 lists the contract used in this example, together with a list of function calls to be evaluated.

```
1  contract InternalExternal {
2      function f() internal returns (address) {
3          return msg.sender;
4      }
5
6      function g() external returns (address) {
7          return f();
8      }
9
10     function h() external returns (address) {
11         return tx.origin;
12     }
13
```

```
14      function f1() public returns (address) {
15          return f();
16      }
17
18      function g1() public returns (address) {
19          return this.g();
20      }
21
22      function h1() public returns (address) {
23          return this.h();
24      }
25  }
26
27  c.f1();
28  c.g();
29  c.g1();
30  c.h();
31  c.h1();
```

Listing 2.6: Internal and external functions in a nutshell

This simple example shows a contract defining six functions. Some are `internal` and some others are `external`. Let an instance `c` of `InternalExternal` be deployed on the blockchain with address a_1, and suppose we are making the invocations listed below the contract definition using an account with address a_2. First, neither a_1 nor a_2 have absolutely nothing to do with the address of the miner, which actually need not to correspond to an Ethereum account. Here, a_2 is just the address of the account *initiating* the transaction, which of course cannot be the miner. Secondly, note that there is no `c.f()` since `f()` is not visible outside the context of `c`, due to its `internal` modifier. Also note that we have to explicitly use `this` when calling an external function from the same contract defining it (i.e. the body of `g1()` and `h1()`). The following addresses are returned:

- `c.f1()` $\rightarrow$ a_2
- `c.g()` $\rightarrow$ a_2
- `c.g1()` $\rightarrow$ a_1
- `c.h()` $\rightarrow$ a_2
- `c.h1()` $\rightarrow$ a_2

The third call, `c.g1()`, returns the address of the deployed contract `c` instead of the one of the account we are making the calls from. This might seem surprising, but actually it is not if we consider what we said about external functions. In fact, they are implemented as new message calls, and thus `msg.sender` is modified accordingly: we are invoking `g()` from the context of `c`, and hence the sender becomes equal to the address of `c`. Indeed, when `g()` is directly called from the "outside", we see the address of our account, as we would expect. On the contrary, note that `h()` always returns the address of our account, no matter who invokes it: this is because it reads `tx.origin`, which always stores the initiator of the transaction, and it of course never changes.

Example 2.4 (Functions as first-class values in Solidity)**.** Listing 2.7 models a trivial contract, *Applier*, storing only one state variable and exposing an interface composed by a single method, *apply*, that inputs a function and applies it to the state variable, returning the result. *Test* is only meant to show how to use *Applier*. It defines a reference to the latter contract and two functions, $f1$ and $f2$, to compute the square and the double of the number 10. As you can see, function types in Solidity are quite complex. Only `internal` or `external` functions can be used as values. The function

type defines the list of keywords that must appear in the definition of the parameters. Indeed, both *square* and *double* are marked as `view` (so they do not modify any state) and `external` (note the use of `this` to reference them). A mismatch in this keyword list makes Solidity's type system reject the function being used.

```
1  contract Applier {
2      uint private state;
3
4      constructor(uint _state) public {
5          state = _state;
6      }
7
8      function apply(function (uint) view external returns (uint) f)
           view public returns (uint) {
9          return f(state);
10     }
11 }
12
13 contract Test {
14     Applier private app;
15
16     constructor() public {
17         app = new Applier(10);
18     }
19
20     function f1() view public returns (uint) {
21         return app.apply(this.square);
22     }
23
24     function f2() view public returns (uint) {
25         return app.apply(this.double);
26     }
27
28     function square(uint n) view external returns (uint) {
29         return n * n;
30     }
31
32     function double(uint n) view external returns (uint) {
33         return n + n;
34     }
35 }
```

Listing 2.7: Functions as values in Solidity

Lastly note that, for what we said before, the definition of `Applier` has to be known when `Test` is compiled and deployed. In this example we can assume that both the contracts are developed at the same time on the same machine, but in Ethereum this is not necessary: an instance of `Applier` may already be on the blockchain when `Test` is compiled and deployed. We shall go deeper on Ethereum's separated compilation and how to deal with it in Section 7.8; for now we can assume that the contracts we use are written by the same programmers, and compiled and developed at the same time.

Reference types Among the reference types we find dynamic `arrays` and `structs`. The definition of the latter cannot contain a member of its own type, as the size of the struct has to be finite.

Another interesting type is `mapping`. A `mapping` is declared as `mapping(_KeyType ⇒ _ValueType)`, where `_KeyType` can be almost any type except for a `mapping`, a dynamically sized array, a contract, an enum, and a struct.

`_ValueType` can actually be any type, including mappings. They can be seen as hash tables virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is all zeros: a type's default value, but the key data is not actually stored in a mapping. Because of this, they do not have a length and cannot be iterated over using loops. Mappings cannot be created locally to a function and are only allowed for state variables: if they are used within a function, they have to reside on the storage and point to a mapping state variable. It is possible to mark mappings `public` and have Solidity create a getter. The `_KeyType` will become a required parameter for the getter and it will return `_ValueType`. Listing 2.8 sums up what we just said (please take note that this contract does not compile).

```
1  contract TestMapping {
2      mapping (uint => bool) private map;
3
4      function f1() public {
5          mapping (uint => bool) wrong; // this does not compile
6      }
7
8      function f2() public {
9          mapping (uint => bool) correct = map;
10         correct[0] = true; // this also modifies map[0]
11     }
12
13     // this function would be automatically created if `map` were
           public
14     function map(uint n) public view returns (bool) {
15         return map[n];
16     }
17 }
```

Listing 2.8: Mappings in Solidity

2.4.3 Exceptions

Solidity uses state-reverting exceptions to handle errors. They are meant to undo all changes made to the state in the current transaction and also flag an error to the caller. When exceptions happen in a sub-call, they are rethrown automatically, and cannot be caught. The reason for reverting is that there is no safe way to continue the execution, because an expected effect did not occur. In fact, to retain atomicity of transactions, the safest thing to do is to revert all changes and make the whole transaction without effect. There are four ways to do so[11]:

- `require(bool)`: used to ensure valid conditions, such as inputs, or contract state variables are met, or to validate return values from calls to external contracts;
- `assert(bool)`: used to test for internal errors, such as division by zero or out-of-bounds access to an array, and to check invariants;
- `revert()`: used to raise an error without providing a boolean expression;
- `revert(string)`, also providing a string message about the error.

[11]The EVM raises `require`-like or `assert`-like exceptions in a number of scenarios. Refer to `https://solidity.readthedocs.io/en/develop/control-structures.html#error-handling-assert-require-revert-and-exceptions` for more information.

Listing 2.9 provides an example of use of these four statements.

```
1  contract Exceptions {
2      address private owner;
3      address[] private array;
4
5      // constructor to initialize `owner` and `array` not shown
6
7      function onlyForTheOwner() public {
8          require (msg.sender == owner);
9          // do something
10     }
11
12     function arrayAt(uint i) public returns (address) {
13         assert (i < array.length);
14         return array[i];
15     }
16
17     function onlyForTheOwnerExplicit() public {
18         if (msg.sender != owner) {
19             revert();
20         }
21         // do something
22     }
23
24     function onlyForTheOwnerExplicitWithMessage() public {
25         if (msg.sender != owner) {
26             revert("You are not the owner of this contract!");
27         }
28         // do something
29     }
30 }
```

Listing 2.9: Exceptions in Solidity

2.4.4 Contracts

As said, contracts are similar to classes, and can be created "from outside" via Ethereum transactions (using, for instance, the JavaScript library web3.js) or from within Solidity contracts, via `new`. The full code of the contract being created has to be known in advance, so recursive creation-dependencies are not possible. Programmers can specify an amount of Wei to send along with the creation (and such amount will become the balance of the new contract), but they cannot specify the amount of gas. Contracts may define one and only one constructor (hence, there is no overloading), which is executed when the contract is being instantiated.

Visibility Solidity provides four types of visibilities for functions and state variables:

- `external`: external functions are part of the contract interface and can be called from the outside, using transactions;
- `public`: public functions are similar to external ones, but can be called via messages (and not via a new transaction). For public state variables, an external getter is automatically added. Some types require these function to input parameters. Examples are arrays and mappings. In general, given a public state variable `x` of a simple type `T`, the getter function has the following signature: `function x() external returns (T)`. If `T` requires an additional parameter, such

as for mappings, the signature is as follows: `function x(KeyType k) external returns (ValueType)`. The reason behind the external visibility is that, from within the current contract, such variables can be accessed in two ways: with `this` (i.e. `this.x()`) to call the getter and without it (i.e. `x`) to access directly the variable without calling the getter;

- `internal`: internal members (i.e. functions and state variables) can be accessed from within the current contract or those deriving from it;
- `private`: private members are similar to internal ones, but can be accessed only from within the current contract.

Others types of function Besides the visibility, Solidity allows programmers to annotate in many ways a function signature. First, `payable` enables a function to receive Ether, that will be added to the contract's balance. If any value is sent to a non-`payable` function an exception is thrown. Secondly, functions not modifying the state[12] can be marked as `view`. The automatically-generated getters fall into this category. Function also not reading[13] the state can be marked as `pure`. In general, they are really rare. Solidity's compiler do not check the compliance of a function with the markers `view` and `pure`, and it is not possible to prevent functions from reading the state at the level of the EVM.

Every contract may contain only one **fallback** function, which has no name, no parameters and returns no value. It is executed whenever a contract is sent Ether without invoking a specific function (i.e. using, for example, `transfer` or `send`), and must consequently be `payable`, or whenever a call does not match with any of the function in the contract's interface. If it does not exist, the contract cannot receive Ether through regular transactions. It can only rely on 2300 gas, leaving not much room to perform operations except basic logging. For instance, writing to storage, creating a new contract or sending Ether consume more than 2300 gas. It is worth to mention that a contract might receive Ether as a consequence of another contract's `selfdestruct`. `selfdestruct` is an operation allowing a contract to terminate itself, sending all its balance to another account. A contract not defining any fallback is allowed to receive an amount of Ether this way, but may not reject them.

Now that we know what a fallback function is, we can explain the difference between `send` and `transfer`. The behavior of the former is quite similar to the latter's: an amount of Wei can be sent to another account, and if the latter is a contract, its fallback function, if any, is executed. If no fallback is defined, `transfer` raises a `revert` and aborts the execution, whereas `send` simply returns `false` to the caller. Hence, checking the returned boolean is extremely important, since programmers have to rely exclusively on it to get to know the result of the money transfer. If it failed, they should explicitly abort the transaction or try to recover from the error. Nonetheless, clumsy implementations might not check this boolean value, thus introducing dangerous bugs in the contract itself. Example 2.5 shows the two functions in action.

Example 2.5 (`send` vs `transfer`)**.** Consider Listing 2.10 pointing out the differences between functions `send` and `transfer` in Solidity. `A`, `B`, and `C` are three simple contracts defining only a fallback function, with some differences:

[12] For a complete list of the forbidden operations see `https://solidity.readthedocs.io/en/develop/contracts.html#view-functions`

[13] For a complete list of the forbidden operations see `https://solidity.readthedocs.io/en/develop/contracts.html#pure-functions`

- `A`'s fallback has an empty body;
- `B`'s fallback is ill-defined, since it is not marked as `payable`;
- `C`'s fallback is well-defined, but its body contains a call to `revert()`.

```
1  contract A {
2      function() public payable {}
3  }
4
5  contract B {
6      function() public {}
7  }
8
9  contract C {
10     function() public payable {
11         revert();
12     }
13 }
14
15 contract Test {
16     address a;
17
18     constructor(address _a) public payable {
19         a = _a;
20     }
21
22     function transfer() public {
23         a.transfer(address(this).balance);
24     }
25
26     function send() public returns(bool) {
27         return a.send(address(this).balance);
28     }
29 }
```

Listing 2.10: `send` and `transfer` in action

Now consider `Test`: it is initialized with a contract's address and provides two functions, `transfer` and `send`, invoking the functions of the same name. The results of their invocation with addresses pointing to an instance of `A`, `B`, and `C`, respectively, are as follows:

- `A`:
  - `transfer` → No revert
  - `send` → `true`
- `B`:
  - `transfer` → revert
  - `send` → `false`
- `C`:
  - `transfer` → revert
  - `send` → `false`

`A`'s results are as expected: `transfer` executes correctly and so does `send`, which in fact returns `true`. On the other hand, `B` and `C` make `transfer` raise a `revert`, thus propagating the exception, whereas `send` simply returns `false`. If this boolean is not correctly handled, the caller may erroneously assume that everything went fine.

Inheritance Solidity supports multiple-inheritance and polymorphism by copying code. All function calls are virtual, and hence the most-derived suitable function is

called, except when the contract's name is explicitly provided. When a contract inherits from multiple contracts, only a single contract is created on the blockchain, and the code from all the base contracts is copied into the created contract.

The reader interested only in the functionality of Solidity modeled in FS may skip this part, as FS does not include any multiple-inheritance.

Before diving into the way function calls work, it is worth to mention a problem that affects languages allowing multiple-inheritance: the diamond problem, sometimes referred to as "deadly diamond of death" [32]. Example 2.6 explains it.

Example 2.6 (Diamond problem)**.** Consider the hierarchy shown in Figure 2.3. Con-

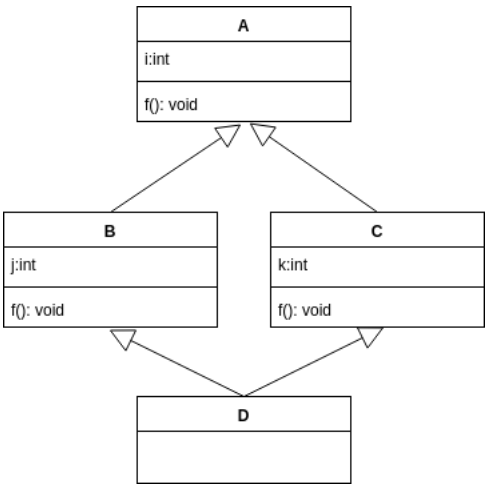

Figure 2.3: Class hierarchy showing the diamond problem

tract `A` defines a state variable `i` as well as a function `f`. It is then extended by two more contracts, `B` and `C`, defining, respectively, state variables `j` and `k`. They also override `f`. Lastly, contract `D` inherits from both `B` and `C`. Two are the problem arising from this hierarchy. First, if an instance of `D` call `f`, which `f` is to be invoked, `B`'s one or `C`'s one? Secondly, which value of `i` is actually visible in `D`? Both `B` and `C` inherit this state variable from `A`, and the value they assign to it is, in general, different.

These two problems are solved by Solidity in the same way as Python does. Using C3 linearization programmers force a specific inheritance order, so that no ambiguity can arise. Such an order is specified after the `is` keyword, from "most base-like" to "most derived". This mechanism rules out some hierarchies, but effectively solves the diamond problem. Bearing this in mind, Example 2.7 explains how virtual calls work, and how the former aforementioned ambiguity is solved.

Example 2.7 (Function calls with multiple-inheritance)**.** Consider the contracts listed in Listing 2.11, and the corresponding calls at the end. We use a lowercase letter (e.g. `a`) to indicate an instance of the contract named with the same, capital, letter (e.g. `A`).

```
1  contract A {
2      string public x;
3
4      function f() public  {
5          x = "A";
6      }
```

```
7  }
8
9  contract B is A {
10     function f() public { x = "B"; }
11 }
12
13 contract C1 is A {
14     function f() public { x = "C1"; }
15 }
16
17 contract C2 is A {
18     function f() public { super.f(); }
19 }
20
21 contract D1 is B, C1 {}
22
23 contract D2 is B, C2 {}
24
25 a.f();
26 b.f();
27 c1.f();
28 c2.f();
29 d1.f();
30 d2.f();
```

Listing 2.11: Virtual function calls in Solidity

Note how Solidity specifies inheritance: the keyword `is` is used to introduce the inheritance list. Initially, `x` is given `string`'s default value (i.e. the empty string ""). Afterwards, every call to `f` modifies such value in some way. `a.f()`, `b.f()` and `c1.f()` have the expected effect, and set `x` to, respectively, "A", "B" and "C1". `super` has the expected meaning: it refers to the immediate supercontract. For `c2.f()` the immediate supercontract is quite evident: it is `A`, and the function sets `x` to "A". More confusing are the cases regarding `D1` and `D2`. Remember the order in the inheritance list goes from the most-base to the most-derived contract. Hence, `d1.f()` calls `f` in `C1`, and sets `x` to "C1". `d2.f()` is more interesting. It causes the call of `f` in `C2`, which, in turn, calls the `f` of its `super`. Nonetheless, this `super` is not `C2`'s supercontract, but instead the next contract in the inheritance list of `D2`: `B`. Hence, `d2.f()` sets `x` to "B".

Subcontracts need to explicitly call the constructor of the contracts they derive. Such constructors are then invoked in the order specified in the inheritance list. There are two ways to do so:

- in the inheritance list itself: `contract Derived is Base(base_args);`
- in the base contract constructor's signature: `constructor (base_args, derived_args) Base (base_args) {}`.

Specifying arguments in both ways is an error. Constructors may be `public` or `internal`. In the latter case, the contract is turned abstract, and so is if it does not provide all the parameters required by its base contract(s).

Contracts are marked as abstract when at least one of their functions lacks an implementation. They cannot be deployed, but can be used as a base constructor. As usual, if a contract inherits from an abstract one and does not implement all the unimplemented functions, it is marked as abstract, too. Abstract contract provide implementation inheritance and facilitate patterns such as the Template method. As in Java, alongside abstract contracts are interfaces. Many restrictions apply to interfaces:

- no implemented functions;
- cannot inherit other contracts or interfaces;
- cannot define constructor;
- cannot define variables;
- cannot define structs;
- cannot define enums.

Casts In Solidity, given a value of type `address`, it is possible to retrieve the corresponding contract instance residing on the blockchain. This operation can be thought of as a cast `address`→`Contract`, but it presents many differences with casts in object-oriented languages, and its behavior can lead to bugs difficult to catch.

Let us begin with a high-level description of how a cast is carried out in Solidity. Consider a statement like the following: `C(a)`, where `C` is a contract name and `a` is an address. First, in statically typed languages, it would compile if and only if `a` really corresponds to an instance of `C`. In Solidity this is not true at all. `C(a)` **always** compiles, even if `a` refers to another contract `D`, but there is more. Suppose that the definition of `C`, as well as the one of `D`, contains a function `f()`, and suppose there is another contract declaration `E`, which does not contain any `f()`. Now consider the statement `C(a).f()`; according to the actual instance `a` refers to three things may happen:

- if `a` refers to an instance of `C` everything works as expected, and `f()` in `C` is called;
- if `a` refers to an instance of `D`, the cast compiles and executes successfully, as well as the function invocation. However, since `a` points to a contract of type `D`, `f()` in `D` is called without any warning to the programmer;
- even more surprisingly, if `a` refers to an instance of `E` the cast still is successful. Nonetheless, at run-time, no `f()` is found since the declaration of `E` does not contain it. Hence, a `revert` is thrown and the transaction is aborted.

Example 2.8 sums up this behavior.

Example 2.8 (Casts behavior in Solidity)**.** Listing 2.12 contains three simple contracts, `A`, `B`, and `C`, defining a function (`f()` for `A` and `B`, `g()` for `C`) invoked by a fourth contract, `Cast`. The latter contains three `address`es as state variables: suppose they are correctly initialized, that is `a` refers to an instance of `A`, `b` to an instance of `B`, and `c` to an instance of `C`.

```
1  contract A {
2      function f() public pure returns (string) {
3          return "A";
4      }
5  }
6
7  contract B {
8      function f() public pure returns (string) {
9          return "B";
10     }
11 }
```

```
12
13  contract C {
14      function g() public pure returns (string) {
15          return "C";
16      }
17  }
18
19  contract Cast {
20      address private a;
21      address private b;
22      address private c;
23
24      constructor(address _a, address _b, address _c) public {
25          a = _a;
26          b = _b;
27          c = _c;
28      }
29
30      function f1() public view returns (string) {
31          return A(a).f();
32      }
33
34      function f2() public view returns (string) {
35          return B(a).f();
36      }
37
38      function f3() public view returns (string) {
39          return A(b).f();
40      }
41
42      function f4() public view returns (string) {
43          return A(c).f();
44      }
45
46      /*
47      f5() does not compile!
48      function f5() public view returns (string) {
49          return C(a).f();
50      }
51      */
52
53      function f6() public view returns (string) {
54          return C(a).g();
55      }
56  }
```

Listing 2.12: Casts behavior in Solidity

We start our discussion from `f5()`, which does not compile. The reason is that, after a cast, the static type of `C(a)` is `C`, and `C` does not contain any function named `f`. The compiler detects the violation and makes the contract not compile. The other function calls are all successful, and produce results as follows:

- `f1()` → "A"
- `f2()` → "A"
- `f3()` → "B"
- `f4()` → revert
- `f6()` → revert

`f1()` behaves as expected, since `a` points to an instance of `A`. Also the second call results in an invocation of `f()` in `A`, because, as we said, `a` refers to an instance of `A` and at run-time the code inspected to find a suitable function is the one of `A`, not `B`,

even though the static type of `B(a)` is `B`. `f3()` confirms this behavior, and now `f()` in `B` is invoked (since the address `b` points to an instance of `B`). `f4()` and `f6()` throw a `revert`. Starting from the former, `c` refers to a contract of type `C`, whose code does not contain any declaration of `f()`. At run-time, the cast will be successful, but the function call will not. The same applies to the latter. The comparison of `f5()` and `f6()` reveals that the compiler does check if the function to be called is defined in the contract corresponding to the static type, but it does nothing to enforce that the address used as a parameter actually points to the desired contract. Perhaps worse, the compiler does not give any warnings.

As we saw, casts in Solidity are extremely difficult to deal with and can lead to subtle bugs. In Chapter 5 we shall model a slightly safer version of this, where `reverts` are raised if the address does not refer to the desired contract (and a warning is given at compile-time). Then, in Chapter 7, we shall propose an extension to better solve this issue.

Events To further illustrate a possible field of application for events, Example 2.9 lists a simple oracle. Oracles in general work as follows: an application calls an oracle's function to specify the URL (Uniform Resource Locator) of an operation to be invoked. Such function emits an event, caught by an external blockchain-monitoring application, which takes the URL, invokes the operation, and calls back the original contract to provide the result. In this way a value depending on external factors is sent as a parameter to a contract's function, thus preserving the determinism of the transaction. Random number are ofter generated in such a way.

Example 2.9 (Oracle)**.** The snipped in Listing 2.13 models a very simple application using a built-in oracle to continuously monitor the temperature of a room.

```
1  contract Oracle {
2      event Execute(address, string);
3
4      function execute(string url) external {
5          emit Execute(msg.sender, url);
6      }
7  }
8
9  contract Room {
10     uint public temperature;
11     Oracle oracle;
12
13     constructor(uint _temperature, address _oracle) public {
14         temperature = _temperature;
15         oracle = Oracle(_oracle);
16     }
17
18     function getTemperature() public {
19         oracle.execute("...");
20     }
21
22     function callback(uint _temperature) public {
23         temperature = _temperature;
24     }
25 }
```

Listing 2.13: Oracle in Solidity

`Oracle` models the oracle contract and defines only a function, `execute()`, taking a string representing a URL and, ideally, getting the result of the operation identified

by such URL. To this end, the function simply emits an event and lets an external application deal with it. It should then invoke `Room`'s `callback`, which handles the result.

Even though this approach works fine in theory, it should be clear that it is really fragile. What happens if `Room` does not define any callbacks or if the address used as a parameter in `Room`'s constructor does not actually correspond to an instance of `Oracle`? In these cases the execution cannot further proceed and a `revert` is raised. We shall see how to solve it in Chapter 7.

Additional examples of code Appendix D contains additional examples of smart contracts written in Solidity.

2.5 Decentralized applications

Normally, web applications consist of a web-based front-end taking commands from users and sending them to a server-side back-end, located somewhere. This server side creates a point of centralization that can fail or suddenly become unavailable. This is solved by Ethereum and Decentralized Applications (DApps). A DApp is an application using Ethereum and smart contracts as a back-end, which now runs in a distributed fashion on the blockchain, thus avoiding the centralization issue. In this configuration, smart contracts implement only a small part of the entire application (i.e. the business logic), since executing things on the blockchain is computational and time expensive. Hence, DApps should typically have their own suite of associated contracts on the blockchain which they use to encode business logic and allow persistent storage of their consensus-critical state.

The front-end is generally browser-based and built using the JavaScript library web3.js, of which we have seen (and will see soon) an example, but this is not mandatory. As we pointed out, many programming languages, from Python to Java and from Scala to Haskell offer a library to interact with Ethereum, and can thus be used to build non-browser-based applications interacting with smart contracts and the blockchain.

Below we go a bit deeper on how to build a DApp. Example 2.1 explained how to deploy and interact with a contract, but provided no additional information on how a Dapp performs the actions regarding the blockchain. To this end, a Dapp has to communicate with an Ethereum node (i.e. a computer with a copy of the Ethereum blockchain mining and validating transactions), which, in web3.js, translates in using an instance of the class `Web3`. For browser-based applications, this is best done using a browser extension known as MetaMask[14]: it implements a `Web3` provider that communicates with the browser extension, which in turn sends API calls to whatever node the user has chosen. It may seem counterintuitive to use an intermediary like MetaMask rather than communicating directly with a node (which is done creating an instance of `HttpProvider`), but MetaMask performs an important function: it keeps a user's private key secure. Ethereum transactions need to be signed with an account's private key, but allowing an application unlimited access to that private key would mean that a malicious application could drain a user's account. Instead, MetaMask intercepts each operation that requires a signature, prompts the user to approve that operation, and then creates the signature using the private key. This way, users are in full control of how their private key is used.

[14] `https://metamask.io/`

There are many ways to use MetaMask when getting an instance of `Web3`: the recommended one is shown in Listing 2.14, and should be executed when the page is fully loaded, to give MetaMask the chance to inject the variable `web3`.

```
1 if (typeof(web3) === "undefined") {
2   error("Unable to find web3, please run MetaMask");
3 } else {
4   web3 = new Web3(window.web3.currentProvider);
5 }
```

Listing 2.14: Creating an instance of Web3 using MetaMask

Now that we know a bit more about how Solidity works, we can show a better way to interact with smart contracts. Remember what we said about `view` functions: their body does not mutate the contract's state. At a first glance this modifier seems unuseful, since the compiler does nothing to enforce it (at the time of writing, Remix just gives a warning), but it is extremely important for application interacting with a smart contract. Not mutating any state means that `view` functions can be safely computed by any node with an up-to-date copy of the blockchain, without even requiring a transaction. This makes `view` function calls fast and free (no gas beforehand payment is required). On the other hand, functions mutating a contract's state require sending a transaction to the blockchain and waiting for confirmation. When they are invoked from the front-end of a DApp, MetaMask will prompt the user to approve the transaction, including the attached gas and the gas price. Figure 2.4 depicts such approval request.

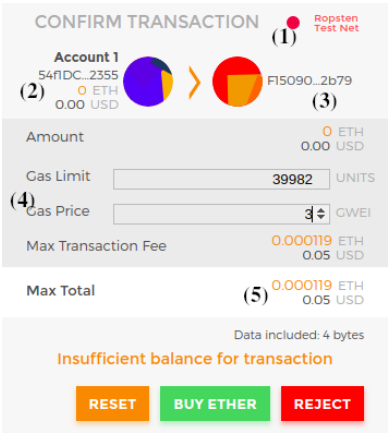

Figure 2.4: MetaMask asking for a transaction confirmation: (1) represents the Ethereum network the transaction is being sent to (a number of test networks are available to test contracts before deploying them to the "official" one); (2) represents our account (i.e. the one sending the transaction); (3) is the recipient; (4) allows users to change the maximum amount of gas and the price for one unit of gas and shows the amount of Ether being sent; and lastly (5) sums up the maximum cost of the current transaction (i.e. *Amount* + *Max Transition Fee*).

To conclude this Chapter, Example 2.10 define a blood bank smart contract (written in Solidity) and shows how to invoke `view` and non-`view` functions in web3.js.

Example 2.10 (A basic DApp)**.** Listing 2.15 models a blood bank.

```
1 contract BloodBank {
```

```
2       mapping (address => bool) private healty;
3       address public doctor;
4       uint public blood;
5
6       constructor() public {
7           doctor = msg.sender;
8       }
9
10      function setHealth(address _donor, bool _isHealty) public {
11          require(msg.sender == doctor);
12          healty[_donor] = _isHealty;
13      }
14
15      function isHealty(address _donor) public view returns (bool) {
16          require (msg.sender == doctor);
17          return healty[_donor];
18      }
19
20      function donate(uint _amount) public returns (bool) {
21          uint donorBlood = Donor(msg.sender).blood();
22          if (healty[msg.sender] && donorBlood > 3000 && donorBlood
                - _amount > 0) {
23              blood += _amount;
24              return true;
25          }
26          return false;
27      }
28  }
29
30  contract Donor {
31      uint public blood;
32      BloodBank public bank;
33
34      constructor(address _bank) public {
35          blood = 5000;
36          bank = BloodBank(_bank);
37      }
38
39      function donate(uint _amount) public {
40          if (bank.donate(_amount)) {
41              blood -= _amount;
42          }
43      }
44  }
```

Listing 2.15: Donor and BloodBank contracts in Solidity syntax

Since not every donor is allowed to donate, `BloodBank` contains a mapping `address => bool` to keep track, for each possible donor, of their health state, here represented by a `bool` (set by default to `false`). Only the `doctor` may change such values, as shown in `setHealth()`. If such function is called by someone else, the execution is stopped with an error. This is a common pattern in Solidity, where access control is modeled keeping track of the addresses allowed to do something and aborting the transaction if there is a mismatch. To explain how to call `view` functions from an external application, we also defined a function `isHealty()` to query the healthiness of a donor.

Allowed donors may donate an amount of blood of their choice. A donation might not succeed for two reasons: either the donor is not in health or they have too low blood in their body. The `blood()` function is a getter automatically generated by Solidity. Every successful donation increments the state variable `blood`.

`Donor` represents a human donor, characterized by an amount of blood (`blood`, set to five liters for the sake of simplicity) and by an explicit reference to its blood bank. This reference is set in `Donor`'s constructor from a value of type `address`. A donor donates by invoking `donate()`, which in turn attempts to call `BloodBank`'s `donate()` function. If the donation was successful, the amount of blood is reduced.

Note that the function `donate()` in `BloodBank` might be invoked also by EOAs or any other contract instances, instead of `Donor`'s. In this case, as pointed out in Example 2.8, the cast would still be successful, but, if no function `blood()` was defined by the caller, the runtime would throw a `revert`, thus aborting the transaction.

Now consider Listing 2.16, representing a very basic DApp using the two contracts defined above. First, we compile and deploy them, as we did in Example 2.1, and then we show two different interactions, from two different EOA: the doctor (identified by `doctorAddress`), and the donor (identified by `humanDonorAddress`). The following scenario is shown:

- a doctor (the same `doctor` stored in the instance of `BloodBank`) checks the health status of a patient and afterwards sets it to `true`;
- the very same patient attempts to donate 500ml of blood using the functions exposed by `Donor`.

Generally, the same DApp shows two different interfaces to, respectively, the doctor and the donor, allowing them to carry out different operations. For the sake of simplicity, we omit the specifications of gas parameters. Furthermore, we wrap the interaction with smart contracts in JavaScript functions: in a normal application, the latter are meant to be invoked as a reaction to an event (such as a pressed button).

```
1  // for the sake of simplicity gas parameters are not shown here
2
3  // Compilation
4  var bankCode = fs.readFileSync('BloodBank.sol').toString();
5  var donorCode = fs.readFileSync('Donor.sol').toString();
6  var solc = require('solc');
7  var compiledBankCode = solc.compile(bankCode);
8  var compiledDonorCode = solc.compile(donorCode);
9
10 var bankByteCode = compiledBankCode.contracts[':BloodBank'].
       bytecode;
11 var bankAbi = JSON.parse(compiledBankCode.contracts[':BloodBank'].
       interface);
12 var donorByteCode = compiledDonorCode.contracts[':Donor'].bytecode
       ;
13 var donorAbi = JSON.parse(compiledDonorCode.contracts[':Donor'].
       interface);
14
15 // EOA addresses
16 var doctorAddress = '0x...';
17 var humanDonorAddress = '0x...'
18
19 // Deploy
20 var BloodBank = web3.eth.contract(bankAbi);
21 var deployedBank = BloodBank.new({
22     data: bankByteCode,
23     from: doctorAddress
24 });
25
26 var Donor = web3.eth.contract(donorAbi);
27 var deployedDonor = Donor.new(
```

```
28      deployedBank.address, {
29          data: donorByteCode,
30          from: humanDonorAddress
31      }
32  );
33
34  // contract instances
35  var bloodbank = BloodBank.at(deployedBank.address);
36  var donor = Donor.at(deployedDonor.address);
37
38  /*
39  Simulating interaction from different parties:
40      1) a doctor checks the health status of a patient and
            afterwards sets it to 'true'.
41      2) a patient donates 500ml of their blood.
42  The sequence of calls is chained using the mechanism of callbacks.
43  */
44
45  // party 1: the doctor
46  // check the health status without any transactions
47  function isHealty() {
48      bloodbank.isHealty.call(deployedDonor.address, {from:
            doctorAddress}, function (err, healthStatus) {
49          if (err) {
50              // do something to handle the error
51          } else {
52              // do something with 'healthStatus'
53          }
54      });
55  }
56
57  // set the health status to 'true' via a transaction
58  function setHealth() {
59      bloodbank.setHealth.sendTransaction(deployedDonor.address,
            true, {from: doctorAddress},
60          function (err, hash) {
61              if (err) {
62                  // do something to handle the error
63              } else {
64                  // wait for the transaction confirmation checking
                        its status via the method
65                  // web3.eth.getTransactionReceipt(hash, callback)
66              }
67          }
68      );
69  }
70
71
72  // party 2: the human donor
73  function donate() {
74      donor.donate.sendTransaction(500, {from: humanDonorAddress},
75          function (err, hash) {
76              if (err) {
77                  // do something to handle the error
78              } else {
79                  // wait for the transaction confirmation checking
                        its status via the method
80                  // web3.eth.getTransactionReceipt(hash, callback)
81              }
82          }
83      );
```

```
84 }
```

Listing 2.16: Basic DApp using BloodBank and Donor

Starting from the doctor, note the difference between a `view` function call, `isHealty()`, and a non-`view` one, `setHealth()`. In Example 2.1 we used a different way to call a smart contract, omitting `call()` and `sendTransaction()`: this is all right and simply let the run-time choose whether to send a transaction or not, if it is not necessary. Consider the call to `isHealty()` at Line 48: specifying `call()` after the name of the function, we tell the run-time that we do not want to send a transaction. Bearing in mind that `isHealty()` is marked as `view`, any node can immediately execute it without mutating the blockchain, and hence the result is immediately available. On the contrary, when invoking `setHealth()` (Line 59), a transaction is necessary since the function modifies the contract's state, and we make it explicit with `sendTransaction()`. In this case, what we get in the callback is not the value returned from `setHealth()`, but simply the transaction's hash, which can be used to monitor the transaction status (has it been processed? was it successful?) with the function `web3.eth.getTransactionReceipt(hash, callback)`. This method calls a callback with two parameters, `error` and `receipt`, where the latter is `null` as long as the transaction is pending. Also note that we specify `deployedDonor.address` as a parameter. We could have used `humanAddress` instead, but the contract `Donor` is intended to mediate between an EOA (here represented by `humanAddress`) and an instance of `BloodBank`. During the donation, it will be an instance of `Donor` that invokes `donate()` on an instance of `BloodBank`: setting the health status of `humanAddress` is then incorrect. Furthermore, if we had used `donor` instead of `deployedDonor.address`, that would have been incorrect too, since the parameter of the function `setHealty()` in `BloodBank` is of type `address`.

The donor part is no different, with an EOA calling a function of `Donor` via a transaction.

This example went deeper on how web3.js works and explained the difference in usage between `view` and non-`view` functions. It also confirmed a very interesting thing: web3.js allows us to specify the sender of a function invocation programmatically with the `from` property. We shall see shortly that this possibility inspired and guided us while modeling our calculus.

Chapter 3

Related work

In this chapter we outline various paper that address, in one way or another, the reliability of smart contracts. We aim to give a comprehensive view of all the related work, focusing on, but not limiting to, the formal aspect.

3.1 Blockchain and design of smart contracts

Delmolino et al. [17] outline many differences between normal programs and smart contracts. They analyzed the most common errors that introduce vulnerabilities or bugs in Solidity code, and they have been the first group to prove that writing smart contracts is by far more sensitive than implementing normal programs.

Parizi, Dehghantanha, et al. [36] have conducted a usability and vulnerability study comparing Solidity with other programming languages, Pact[1] and Liquidity[2], designed for two different blockchains. Measuring these two parameters is important because it tells us how intuitive a language is and how probable is to introduce bugs during its usage. The results say that Solidity is more usable than the others, with a very low average implementation time, but it is also the most vulnerable. Indeed, Pact and Liquidity contracts developed during the experiment did not have any vulnerabilities, whereas Solidity ones suffered from reentrancy (the same vulnerability as The DAO) and Denial of Service caused by unexpected reverts. Sad to say, Solidity is the most popular and most used among the three. Hence, studying and formalizing it is a very good way to develop tools and methods to help programmers reduce the number of vulnerabilities in their code.

As common in programming languages, a set of patterns exist to limit the threats a smart contract can suffer from. The official documentation of Solidity lists a couple of them, but Wöhrer and Zdun [52] state and explain a series of design pattern to help programmers avoid common errors or wrong designs. The idea is exactly the same as in object-oriented programming, where design patterns are widespread: each contract should accomplish a single functionality and it should have a single responsibility. Decoupling the code makes it easier to read and modify it, reducing the risk of including vulnerabilities. Using design patters is a solution of an higher level than the one we address: patters might make the code simpler, but in Solidity there is the need of being 100% sure that a contract does not contain any backdoor. The good properties of these patters should be formally proven, and our work can help in this direction.

[1] `http://kadena.io/#pactModal`
[2] `http://www.liquidity-lang.org/`

Atzei, Bartoletti, and Cimoli [6] provide a list of many smart contract vulnerabilities in Ethereum. Along with reentrancy and exceptions are many other subtle bugs that can make a contract misbehave, such as casts or a poor design. In recent years, many works have focused on developing tools aiming to detect some of these vulnerabilities. However, the vast majority does so by using heuristics, with false positives and negatives, that cannot be used to be 100% sure about the absence of bugs. Tikhomirov et al. [49] provide an in-depth classification of many issues Solidity code can suffer from, classifying them into four categories: security (they can lead to malicious exploits), functional (the intended functionality is not achieved), operational (they can lead to run-time problems, such as poor performance), and developmental (making the code difficult to read, maintain, and improve).

Sergey and Hobor [44] give practical examples of concurrent behaviors at the level of the blockchain during the execution of smart contracts. In fact, the order of the transactions included into a block is not determined when a transaction runs, and, thus, the outcome can largely depend on the ordering with respect to other transactions, as already pointed out by Luu et al. [31]. Sergey and Hobor make the parallelism "Accounts using smart contracts in a blockchain are like threads using concurrent objects in shared memory", and identify many issues that can arise from this (implicit) concurrency.

Pettersson and Edström [37] propose a library for the programming language Idris[3] that allows for the development of secure smart contracts using dependent and polymorphic types. They extend the existing Idris compiler with a generator for Serpent[4] code (a Python-like high-level language for Ethereum smart contracts). This compiler is a proof of concept and fails in compiling more advanced contracts (as it cannot handle recursion). Furthermore, Serpent is explicitly not recommended for developing smart contracts.

Mavridou and Laszka [33] design a kind of graphical editor for collaborative development of smart contracts based on finite state machines. Automata are a well known tool in computer science, and this type of development process could be more intuitive for some programmers. The tool also provides some built-in patterns to avoid common vulnerabilities. Nonetheless, the work lacks formal foundations and there is no proof of correctness of the translation from the state machines to Solidity. FS could help in this way: a similar tool generating FS code could be developed. Thanks to our formalization, every translation step (graphical editor $\rightarrow$ FS $\rightarrow$ Solidity/bytecode) could be proven correct. We believe that this path is a very promising one, as programmers often do not know what they are doing: an intuitive, well designed, and correct by construction, graphical editor could reduce the number of bugs and security vulnerabilities.

3.2 Formalizations of the Ethereum Virtual Machine

KEVM [23], by Hildenbrandt et al., is the first fully executable semantics for the EVM. It has been created in the $\mathbb{K}$ framework [42], and various tools for contracts verification have been generated from it. $\mathbb{K}$ is an executable semantic framework in which programming languages, type systems and formal analysis tools can be defined. Taking such formal language definitions as input, $\mathbb{K}$ generates a variety of tools for the defined language, without any other piece of knowledge about the given language except its formal syntax and semantics. KEVM not only has served as a verification tool for smart contracts, but it has also revealed many ambiguities and potential sources of er-

[3] https://www.idris-lang.org/

[4] https://github.com/ethereum/serpent

ror in the existing formalization of the EVM semantics [53], such as `delegatecall` or overflows. KEVM have passed all the tests for EVM implementations. Before this work, no rigorous and complete formalization of the virtual machine had existed, leaving a lack of rigor to base verification tools on.

KEVM is not the only existing formalization. Hirai [24] does the same thing using Isabelle/HOL[5], a higher-order logic theorem proving environment well suited for formal verification of software (i.e. proving properties of computer languages and protocols). Hirai proves safety properties and invariants of Ethereum contracts in the presence of reentrancy. As a side effect, 13 inconsistencies (with respect to [53]) have been discovered. This work has posed the basis for many other works. In particular, Amani et al. [4] extended [24] covering smart contract correctness properties and giving a separate universal treatment of termination based on Ethereum's concept of execution "gas". This program logic allows the verification of smart contracts at the bytecode level.

A third work [21], by Grishchenko, Maffei, and Schneidewind, formalizes a small-step semantics for EVM bytecode and a formalization in F*[6] of a large part of this semantics. The aim is the same as before: formalizing, and validating, security properties on smart contracts.

Formalizing the virtual machine is a research path running in parallel with the formalization of Solidity. The two are strictly related, since Solidity code is compiled down into bytecode, and expressing properties on the latter is of great utility. However, we believe that programmers do not have to do directly with the virtual machine. Instead of aiming for correction by construction (i.e. writing smart contracts and then analyzing the bytecode they produce), ameliorating the languages operating on a high level, closer to programmers, could be of greater utility in the direction of writing better contracts. This the reason why FS formalizes the core part of Solidity, as well as its semantics and type system, and proposes some improvements.

3.3 Smart contracts analysis

Many recent works address smart contract implementations from a formal point of view. Two are the possible approaches: analyzing the EVM bytecode or working on a higher level, taking into account, for example, Solidity code. Both have pros and cons. The former, for example, has the advantage that the bytecode is stable and that it does not change (or it does in a quite limited way) over time. Many high level languages can be designed, but, in order to run in Ethereum, all have to be finally "translated" into bytecode. Programmers, however, do no develop programs using the bytecode. Instead, they use high level languages such as Solidity. Hence, formalizing and working on these languages narrows the gap between programmers and Ethereum. Pattern and best practices can be tailored in one language, and formal methods can help programmers in writing code respecting these patterns. Nonetheless, languages like Solidity are less mature, and changes may introduce deep modifications from one release to another. Furthermore, also the compiler have to be analyzed and proven bug-free. Currently, the Solidity compiler is written in C++, and importing its definition in a theorem prover is nearly impossible. In fact, the definition of the whole C++11 language has not been formalized yet, although some of the hardest aspects of the language, such as concurrency [7] or inheritance [41], have been addressed. Hence,

[5] `https://isabelle.in.tum.de/`

[6] `https://www.fstar-lang.org/#introduction`

formal verification of Solidity code has to operate outside theorem provers: this is possible, but rather difficult, since many aspects of Solidity's semantics might change over time. The way we chose is to formalize a calculus modeling the core part of this language, proving properties of its type system and proposing extensions addressing some of its flaws.

Program analysis can be either static or dynamic. The latter is basically like testing, and can reveal only the presence of bugs, not prove their absence. The former, instead, examines the code without running it. The process provides an understanding of the code structure, and can help ensure that the code adheres to certain properties. Usually, static analysis behaves as follows:

1. an intermediate representation (IR), such as an abstract syntax tree, is built from the source code;

2. the IR is enriched with additional information, using algorithms such as control- and dataflow analysis, taint analysis, symbolic execution, or abstract interpretation, depending on what the IR is used for;

3. vulnerability detection with respect to a database of patterns, which define vulnerability criteria in IR terms.

3.3.1 Static analysis

Le et al. [29] address the conditional termination of smart contracts. Even though Solidity uses gas to make sure that every function eventually terminates (possibly with a `revert` due to an out-of-gas exception), a mechanism to prove conditional termination can be useful in other languages compiling down into EVM bytecode. Furthermore, it can be applied also to Solidity contracts to solve the problem by construction: letting a contract compile when one, or more, of its functions will *for sure* terminate only thanks to gas is a waste of money and also an error that should be corrected as soon as possible. FS assumes every program as a terminating one, but we could include this work to prove conditional termination.

One of the first static analysis tools for Solidity is OYENTE [31]. Luu et al. [31] provided a list of common vulnerabilities, proposing a better design (which requires all clients in the network to upgrade) and a tool to help programmers develop better contracts. OYENTE is based on symbolic execution, which represents each program variable as a symbolic expression. Each execution path is then expressed in terms of a logic formula built over the symbolic expressions. In order for the execution to follow the path, all the actual values must satisfy that formula. Of course, if there are no values satisfying those constraints, the path will never be taken at run-time. Symbolic execution can achieve, in principle, a better precision and a lower false positives rate (with respect to, for example taint or data flow analysis) but, in general, it also gets a lower code coverage. OYENTE's main aim was to prove that the semantics of Solidity is subtle, and that the vulnerabilities Luu et al. identified actually happened in practice. Their thoughts were confirmed: out of 19366 analyzed contracts, 8833 contained at least a vulnerability (according to OYENTE). It was possible to collect the actual code of only 175 of these contracts, and the false positives rate was of 6.4% (i.e. 10 cases out of 175).

OYENTE has been extended in many ways. One of them is ETHIR [2], by Albert et al., a tool for decompilation of EVM bytecode into a high-level representation in a rule-based form. This form makes it easier to apply the existing tools to infer properties

of the bytecode, because the control and the data flow are explicit. It does so by initially using OYENTE to produce a set of blocks that store the information needed to represent the control flow graph of a set of EVM instructions. It then translates this graph (or better, each block of this graph) into a rule-based representation.

Rosu [43] collects the recent progress in using the $\mathbb{K}$ framework to verify smart contracts semantics. The work formalized the semantics of Vyper in $\mathbb{K}$, and found several bugs and inconsistencies. Vyper[7] is a novel programming language, compiling to EVM, for smart contracts that aim for increased security, simplicity, and human readability. Along with Vyper, a novel consensus protocol, Casper [11], is being developed. It is meant to save wasteful electricity expenditures and at the same time provide greatly increased security. Verifying Casper behavior and Vyper code is very important, since they will play a key role for Ethereum in the near future. Rosu are also formalizing the actual protocol in Coq[8] and Isabelle.

Chen et al. have identified first 7 [13] and then 24 [12] anti-patterns in smart contract design. They define an anti-pattern as an EVM operation sequence that can be replaced with another one that has the same semantics but needs less gas. Hence, their work focus on finding methods to detect and reduce the waste of gas, making Ethereum users spend less money. It might seem that detecting gas waste is not as crucial as detecting other vulnerabilities, but this is not true. Smart contracts are so critical that their implementation should be deeply reasoned about and optimized, in order not to perform unnecessary operations that may introduce bugs. Chen et al. [12] have developed GASREDUCER, a tool that analyzes contracts looking for these anti-patterns. They have analyzed all the deployed smart contracts (i.e., 599,959 as of 10 June, 2017), and detected 9,490,768 instances of anti-patterns wasting 2,040,892,224 units of gas. The calculus we are to propose can integrate these anti-patterns definitions for proving that a given contract does not suffer from any of them. In fact, currently GASREDUCER works on the bytecode. "Merging" it with Featherweight Solidity would mean reducing the gap between EVM code and Solidity code.

Tikhomirov et al. [49] provide a static analysis tool, SMARTCHECK, using lexical and syntactical analysis on Solidity source code. It generates an XML parse tree as an intermediate representation, and detects vulnerability patterns by using XPath[9] queries on it. The tool thus provides full coverage: the analyzed code is fully translated to the IR, and all its elements can be reached with XPath matching. The advantage of this method is that new languages can be added leaving the IR-level algorithms unchanged. On the other hand, XPath queries can easily lead to false positives (when applied, for instance, to reentrancy or timestamp dependencies). The result of their experiment, conducted on 4600 contracts, reveals that SMARTCHECK incurs in more true positives than OYENTE and SECURIFY. This means that SMARTCHECK is well suited to detect certain kind of vulnerabilities, but it also incur in more false positives due to the use of XPath.

In blockchain, an invocation is a run of a smart contract. Depending on the actual input values, the execution path may vary. Hence, looking at a *single* invocation is not enough to discover all the vulnerabilities a contract can suffer from. An alternative approach consists of looking at a trace of invocations. Nikolic et al. [35] have developed a tool, MAIAN, using systematic techniques to find contracts that violate specific properties of traces. Violations are either of liveness properties, asserting that there exists a trace from a specified blockchain state that causes the contract to violate certain

[7] https://github.com/ethereum/vyper
[8] https://coq.inria.fr/
[9] https://www.w3schools.com/xml/xml_xpath.asp

conditions, and of safety properties, asserting whether some actions cannot be taken in any execution starting from a specified blockchain state. They defined three categories of contracts:

- the greedy, those contracts that remain alive and lock Ether indefinitely, allowing it be released under no conditions;
- the prodigal, those contracts returning funds to accounts they had never had to do before, that is, an arbitrary address;
- the suicidal, those contracts invoking the `SUICIDE` instruction (that terminates a contract's life) transferring Ether to an account they had never had to do before.

Their approach extends OYENTE adding a semantics taking into account invocation traces. They also formalize, into logical formulas, properties characterizing greedy, prodigal, and suicidal contracts, and use symbolic execution to check their satisfiability. They analyzed 970,898 smart contracts, obtained by downloading the Ethereum blockchain from the first block until block number 4,799,998. Out of these, it was only possible to download the code 9,825 contracts (about 1% of the total), highlighting the usefulness of analyzing the bytecode. The percentage of true positives, with respect to prodigal and suicidal contracts, is 97% and 99%, respectively. On the other hand, greedy contracts have a false positive rate of 31%, quite high. This is due to many causes, but in general finding a trace that leads to an Ether transfer may require three or more traces.

Another static analyzer is SECURIFY [51], by Tsankov et al. It was born from two key observations: first, symbolic execution, on which other tools are based on, has many false positives, requires a long time to inspect large contracts and usually gets a low code coverage; secondly, many security properties can be expressed as patterns on the data flow graph. SECURIFY states such properties with two kinds of patterns: compliance patterns, which imply the satisfaction of the property, and violation patterns, which imply its negation. This tool works as follows: first it parses the EVM code, decompiling it into a static single-assignment form, then it looks for violation patterns. Such patterns are, for example, writing to storage after having invoked another function, or not validating the arguments. The results of their experiments show that this approach is more effective than symbolic execution.

Alongside these tools are many others static analyzers well suited for certain security properties. Examples are Zeus [27], by Kalra et al., and EtherTrust [22], by Grishchenko, Maffei, and Schneidewind. They operate at different levels, either analyzing Solidity code, abstract implementations of the latter, or EVM bytecode. However, static analysis often puts in practice heuristics, leaving room for false positives or negatives. Furthermore, the adoption of these tools could be limited: many programmers could ignore them, making the research progresses useless. We think that security properties, as well as contract invariants or safety properties, have to be checked directly by the compiler whenever it is possible. Smart contracts are not like normal programs that may be patched if something turns out to be wrong. Once a contract is deployed, no modifications or patches can be applied. These programs must be correct by construction, and we strongly believe that analysis tools should play a fundamental role during the development phase. This is the reason why we decided to rely on the compiler. By operating on the type system we make the compilation (as well as the compiler itself) more complex and selective, but, on the other hand, we are able to rule out or detect dangerous patterns. Of course, not every property is enforceable at compile-time, or it is without making the language excessively complex. "Heavy" type systems have

the only effect of making a language too verbose, thus giving programmers a reason to abandon it. As we saw in this section, many are the static analysis tools working on the code *after* its development, but none of them targets the type system. Solidity has the precise and explicit aim to be a type-safe language, and, to the best of our knowledge, this is the first work aiming to prove so also proposing some modifications to make the language sounder.

3.3.2 Dynamic analysis

Jiang, Liu, and Chan [26] have developed CONTRACTFUZZER, a tool analyzing smart contracts with fuzzing in order to detect common vulnerabilities. Fuzzing is based on the dynamic generation of a set of values used as an input for the program to test. Along with reentrancy and gasless send (i.e. invoking a fallback function that requires more than 2300 gas), it is designed to detect other, less common, vulnerabilities, such as timestamp or block number dependency (i.e. using the block timestamp or number, respectively, for critical operations, such as a random number generation) as well as the dangerous use of `delegatecall` (about which we discussed in Section 2.4.2). The results of CONTRACTFUZZER, applied to 6991 contracts, are very good: it falls into false positives very rarely, and only when detecting timestamp or block number dependency. When compared to OYENTE, CONTRACTFUZZER has a lower rate of false positives and an higher rate of false negatives. The former is due to the difficulty, for OYENTE, of symbolically analyze certain types of operations. On the other hand, CONTRACTFUZZER relies on the dynamic generation of input to dynamically test a smart contract, a process that could require a long time to detect something. Hence, with a limited analysis time, some bugs are not detected. Furthermore, dynamic analysis can only prove the presence of vulnerabilities, not their absence, and this is an important limitation in this context.

A similar work, even though very preliminary and limited to reentrancy vulnerabilities, comes from Liu et al. [30]. They have developed a fuzzing tool, REGUARD, focused on finding reentrancy, and analyzed 5 contracts, each for 20 minutes, comparing their results with the ones of OYENTE. It turned out that REGUARD suffers from false positives and negatives less than OYENTE. Even though the results are encouraging, the test is not significant enough to say that REGUARD may be a useful (or complementary) tool.

Fuzzing (and dynamic analysis in general) is very dependent on the amount of test time, which cannot be too high. However, tools like REGUARD or CONTRACTFUZZER may serve as a first alarm. They are for sure more lightweight than more complex tools operating, statically, on the source code (including bytecode), but they are not a replacement.

3.4 Translating Solidity into other languages

The first work in this direction comes from Bhargavan et al. [8], that translate both Solidity and EVM (when Solidity code is not available) info F*, a general-purpose functional programming language with effects aimed at program verification. Its type-system includes dependent types, monadic effects, refinement types, and a weakest precondition calculus. Together, these features allow expressing precise and compact specifications for programs, including functional correctness and security properties. Although this was only a preliminary work, it first tranced the path to verify, statically,

smart contracts implementations.

Instead of analyze Solidity or EVM code, Sergey, Kumar, and Hobor [45] propose a brand new intermediate language, SCILLA, that provides a clean separation between the communication aspect of smart contracts on a blockchain and a programming component. SCILLA is not meant to be a high-level language, but instead to be a target for Solidity code, to perform analysis and verification before compiling it down to bytecode. The key point is that *communication* is separated from *computation*: each contract is represented by an automaton where computations are carried out as standalone (i.e. not involving any other parties) transitions. These transitions end whenever the interaction with another party is required. Such separation enables for a much clearer reasoning about contracts. They implement SCILLA into Coq and use the latter to reason about the properties of a contract. Once the contracts is formalized in Coq, SCILLA allows programmers to prove many different properties, from contract invariants or temporal properties (including liveness). Being able to reason about contracts behavior at the point of proving invariants is a great achievement: programmers can now be sure that their programs will behave in a given way, having a mathematical proof of this. However, at the time of writing, there is no tool to compile Solidity code down to SCILLA, as well as there is no automated way to translate the latter into Coq. These are important limitations, because SCILLA is not meant to be a language directly used by programmers. Furthermore, even supposing an automated way to translate smart contracts written in Solidity into Coq (using SCILLA as a man-in-the-middle), programmers would have to learn how Coq works deep enough to become capable of proving things. Such knowledge is non-trivial at all, and many programmers could find it difficult to acquire it, with the only consequence of a limited use of SCILLA, which would then be useless, since a lot of buggy smart contracts would be deployed anyways.

IELE [28], by Kasampalis et al., is another intermediate language specifically designed to serve five different purposes:

- *security*, eliminating, by construction, many vulnerabilities, such as integer overflow and execution of data as code, to remove possible attack vectors;
- *formal verification*, to help programmers detect software bugs and prove the correctness of the contracts they write;
- *human readability*, storing readable code helps different parties agree on the behavior of a given contract;
- *determinism*, the virtual machine specification defines everything, leaving no room for undefined or implementation-dependent behavior;
- a *gas model* different than Ethereum's one, where there is no limit to execution, but the cost increase as long as the contract requires more resources.

The language is formalized into $\mathbb{K}$, and a compiler exists to generate IELE code from Solidity, as well as an IELE Virtual Machine to execute the former. Thanks to these tools, Kasampalis et al. have successfully deployed and executed IELE smart contracts on a Ethereum-based blockchain. The language has been proven secure, complete and correct, and it represents a great improvement of the execution environment (i.e. the virtual machine). By avoiding the drawbacks of the EVM pointed out by Hildenbrandt et al. in the formalization of KEVM [23], it set a new, higher, level for the development and design of blockchain virtual machines. Unfortunately, at the time of writing, it has not been used for real-world contracts.

The use of intermediate languages to validate properties of smart contracts is a good way to help Solidity grow. When developing or formalizing an alternative language, many limits of the original one come up, but we believe that before diving into brand new intermediate languages we should have a formal foundation of Solidity. FS attempts to provide such formal foundation, even though it is not its aim to validate arbitrary security properties or help programmers detect flaws. More important, it is an abstract calculus that can be extended in many ways to investigate the soundness of new features before adding them in the "real" language. Running behind an immature language such as Solidity could be an energy loss. On the other hand, using a formal foundations to prove, by construction, the safeness of a certain aspect could be much more useful. This is the reason why we chose to focus on Solidity itself, without working on alternatives that have to continuously adapt to a living language.

Chapter 4

Syntax

Featherweight Solidity (FS) is a minimal core calculus for Solidity [48]. It is intended to study various aspects of smart contract programming, such as new contracts deployment, interaction among deployed contracts, and money transfers. It takes inspiration from other calculi, such as DJ [1] and FJ [25]. The grammar of FS is presented in Figure 4.1. The syntax is intended to be liberal: ill-formed terms, such as $\mathsf{balance}(5)$, are ruled out by type rules.

4.1 Grammar of FS

$(Contract\ declaration)$	SC	$::=$	$\mathsf{contract}\ C\ \{\tilde{T}\,s;\ K\ \tilde{F}\}$
$(Constructor\ declaration)$	K	$::=$	$C\ (\tilde{T}\,x)\ \{\mathsf{this}.\tilde{s}\ =\ \tilde{x}\}$
$(Function\ declaration)$	F	$::=$	$T\ f\ (\tilde{T}\,x)\ \{\mathsf{return}\ e\}\ \mid\ \mathsf{unit}\ fb\ ()\ \{\mathsf{return}\ e\}$
$(Contract\ table)$	CT	$::=$	$\emptyset\ \mid\ CT \cdot [C \mapsto SC]$
$(Blockchain)$	β	$::=$	$\emptyset\ \mid\ \beta \cdot [(c,a) \mapsto (C, s\tilde{:}v, n)]\ \mid\ \beta \cdot [x \mapsto v]$
$(Program)$	$\mathcal{P}$	$::=$	(CT, β, e)
$(Expressions)$	e	$::=$	$v\ \mid\ x\ \mid\ \mathsf{this}\ \mid\ \mathsf{this}.f\ \mid\ \mathsf{msg.sender}\ \mid\ \mathsf{msg.value}\ \mid$ $\mathsf{balance}(e)\ \mid\ \mathsf{address}(e)\ \mid\ e.s\ \mid\ e.\mathsf{transfer}(e)\ \mid$ $\mathsf{new}\ C.\mathsf{value}(e)(\tilde{e})\ \mid\ C(e)\ \mid\ e; e\ \mid\ T\ x = e;\ e\ \mid$ $x = e; e\ \mid\ e.s = e; e\ \mid\ e[e]\ \mid\ e[e \to e]\ \mid$ $e.f.\mathsf{value}(e)(\tilde{e})\ \mid\ e.\mathsf{value}(e)(\tilde{e})\ \mid\ \mathsf{revert}\ \mid$ $e.f.\mathsf{value}(e).\mathsf{sender}(e)(\tilde{e})\ \mid\ \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$
$(Values)$	v	$::=$	$\mathsf{true}\ \mid\ \mathsf{false}\ \mid\ n\ \mid\ a\ \mid\ \mathsf{u}\ \mid\ M\ \mid\ c\ \mid\ c.f$
$(Types)$	T	$::=$	$\tilde{T} \to T\ \mid\ \mathsf{bool}\ \mid\ \mathsf{uint}\ \mid\ \mathsf{address}\ \mid$ $\mathsf{unit}\ \mid\ \mathsf{mapping}(T \Rightarrow T)\ \mid\ C$

Figure 4.1: The static syntax of the Featherweight Solidity language

The language presented here contains only a subset of Solidity. To simplify the formal system and associated proofs we removed looping primitives, arrays, structs, events, modifiers, and subtyping. Nevertheless, we include conditional expressions, and looping is allowed by recursion. We do not take into account the different types of visibility (represented in Solidity with `external`, `internal`, `public`, and `private`). We removed `pure` and `view` from function declaration, since Solidity's documentation explicitly states that restrictions associated with those two modifiers are not enforced yet. Lastly, we consider every function as `payable`, and we do not model gas.

As we shall explain later on, contracts in FS model both externally owned and code-driven Ethereum accounts. We also include primitives to initiate a transaction that are the only FS code not included in a contract definition.

Below we explain in detail the user syntax; runtime syntax (i.e. those terms occurring only at run-time, and thus that cannot be written by programmers as part of their source code) will be addressed in Chapter 5.

Remark (Tuple notation in FS syntax)**.** Figure 4.1 contains many different notations about tuples. In general, we identify a tuple of elements by means of the symbol $\tilde{}$, that is $\tilde{e}$ represents a tuple $e_1; \ldots; e_n$, for some $n \in \mathbb{N}$. Similarly, $\tilde{T} \to T$ indicates a type like the following: $T_1; \ldots; T_n \to T$, where $T_1; \ldots; T_n; T$ are all types as defined in Figure 4.1. The same applies also when $\tilde{}$ is in the middle of two characters, such as in $\tilde{T}\,s$. It indicates a tuple like $T_1\ s_1; \ldots; T_n\ s_n$, for some $n \in \mathbb{N}$. $\mathsf{this}.\tilde{s}\ =\ \tilde{x}$ represents, in constructor declarations, a sequence of assignments as the following: $\mathsf{this}.s_1 = v_1; \ldots; \mathsf{this}.s_n = v_n$. Lastly, $s\tilde{:}v$ in the definition of β represents a tuple of the following form: $s_1 : v_1; \ldots; s_n : v_n$, $\exists n \in \mathbb{N}$. This means that for every s_i, $1 \leq i \leq n$ there is a v_i corresponding to it.

Contract declaration We write $\mathsf{contract}\ C\ \{\tilde{T}\,s;\ K\ \tilde{F}\}$ to denote a smart contract declaration. It indicates that contract C contains a sequence, possibly empty, $T_1\ s_1; \ldots; T_n\ s_n$; of typed state variables, whose names are different to each other, a constructor K, and some function definitions $\tilde{F}$. $\mathsf{unit}\ fb\ ()\ \{\mathsf{return}\ e\}$ is the so-called fallback function, and it is mandatory in order to allow C to receive an amount of Ether. In Solidity, it takes no parameters and returns no value, and is implicitly invoked whenever C receives money. Note that our definition of fb actually returns unit. As we said, the syntax is intended to be liberal, and type rules enforce various restrictions (such as forcing fb to return a value of type unit). Functions in Solidity are very similar to methods in usual object-oriented programming languages. $T\ f\ (\tilde{T}x)\ \{\mathsf{return}\ e\}$ defines a function named f that takes a sequence, possibly empty, of typed parameters $(T_1\ x_1, \ldots, T_n\ x_n)$ and whose body returns a value of type T. Functions in Solidity may return tuples (i.e. sequences of typed values) or structures, but, for the sake of simplicity, we chose not to model this behavior. Generalizing the syntax, and the corresponding rules, to those cases is trivial.

In FS, fb is a reserved identifier identifying the fallback function, whereas f is a metavariable for function names. For the sake of simplicity, we give an identifier to the fallback function even if in Solidity it does not have one.

In Solidity an account can be either controlled by a an external party or by code. In the latter case we talk about contracts. We chose not to add explicitly this difference in FS, and the former category is defined by Definition 1.

Definition 1 (Externally owned account)**.** We consider an externally owned account as an empty contract, that is, one containing only a constructor and a fallback function, both empty. Its form is the following:

```
contract EOC {
    EOC() {}
    unit fb() {return u}
}
```

Where EOC stands for Externally Owned Contract, that is an Externally Owned Account modeled as a contract.

We assume that at every instant the blockchain contains as many EOC instances as the number of active externally owned accounts. This definition reflects what can be done by EOAs in Ethereum: starting a transaction or receiving an amount of Ether. The former is accomplished via the expression $c.f.\mathsf{value}(n).\mathsf{sender}(a_{EOC})(\tilde{e})$, where c is a reference to the contract to interact with, f is one of its functions, n is an amount of Ether (potentially 0), and a_{EOC} is the address of an instance of EOC. Furthermore, since EOC defines no functions, it can be used only for the two ends mentioned above: its code does not interact with any other contracts, and no function, but the fallback fb, can be invoked. Note that there is no rule in FS limiting what can be done with EOC: its limitations are a consequence of its definition.

Contract table A contract table, CT, is a mapping from contract names to contract definitions. As in Ethereum, each contract definition is unique in the blockchain. A contract table is inductively defined with the concatenation operator $\cdot$, allowing a new entry to be appended to CT. Definition 2 defines its domain.

Definition 2 (Domain of a contract table)**.** The domain of a contract table CT is defined as follows:

$$\begin{aligned} \mathrm{dom}(\emptyset) &= \emptyset \\ \mathrm{dom}(CT \cdot [C \mapsto SC]) &= \{C\} \cup \mathrm{dom}(CT) \end{aligned}$$

Blockchain β represents the blockchain. As pointed out by Definition 3, β maps pairs (c, a) to triples $(C, \tilde{sv}, n)$, and variable identifiers $x \in Var$ to values v, where Var represents an infinite set of variable identifiers. This formalization follows the Ethereum's one: as we said in Section 2.3.2, accounts contain a nonce, a balance, the contract's code, and the storage. β indeed associates to any pair (c, a) the name of the contract (C, used to retrieve the code, when necessary), a basic storage with the contract's state variables ($\tilde{sv}$), and its balance (n). c and a are, respectively, a contract reference and its address. Note that this pair must be unique in the domain of β, as we pointed out in Section 2.3.2. In other words, once a contract has been deployed on the blockchain only one address corresponds to it, and, vice versa, an address uniquely identifies only one contract deployed on the blockchain. Formally, $\forall (c, a), (c', a') \in \mathrm{dom}(\beta), c = c' \Rightarrow a = a'$ and $a = a' \Rightarrow c = c'$. Leveraging this uniqueness property, we shall abuse the access notation as follows:

$$\begin{aligned} \beta(c) &\triangleq \beta(c, a) \; \exists!\, a \,.\, (c, a) \in \mathrm{dom}(\beta) \\ \beta(a) &\triangleq \beta(c, a) \; \exists!\, c \,.\, (c, a) \in \mathrm{dom}(\beta) \end{aligned}$$

In the rest of this work, we shall often need to retrieve a reference to a contract knowing its address or vice versa. We define such operation as follows:

$$\begin{aligned} \hat{\beta}(c) &= a \quad if\ (c, a) \in \mathrm{dom}(\beta) \\ \hat{\beta}(a) &= c \quad if\ (c, a) \in \mathrm{dom}(\beta) \end{aligned}$$

Lastly, we define another abuse of notation to retrieve the code of a contract given either its address or its reference:

$$\beta^C(c) = \beta^C(a) \quad = \quad C \quad if\ \beta(c,a) = (C, s\tilde{:}v, n)$$

We chose not to model the possibility of saving arbitrary variables on the blockchain. As said in Section 2.4.2, in fact, variables may be stored either in the storage or in the memory area, but our definition of β does not support this feature. For simplicity we do not model an explicit difference between memory and storage; otherwise, we would have to complicate the formalization to keep track of scopes. Hence, the only differentiation is the following:

- the **storage** keeps state variables, contract balances and code. Nonetheless, it does not contain any variables except the state ones;
- the **memory** contains local variables only, and they are added to β each time a function declares them. Differently from Ethereum, we do not garbage-collect the local variables. However, since we assumed an infinite set Var of variable identifiers, not removing them from β is not a problem.

Definition 3 formally defines $\mathrm{dom}(\beta)$.

Definition 3 (Domain of β)**.**

$$\begin{aligned}
\mathrm{dom}(\emptyset) &= \emptyset \\
\mathrm{dom}(\beta \cdot [(c,a) \mapsto (C, s\tilde{:}v, n)]) &= \{(c,a)\} \cup \mathrm{dom}(\beta) \\
\mathrm{dom}(\beta \cdot [x \mapsto v]) &= \{x\} \cup \mathrm{dom}(\beta)
\end{aligned}$$

β's codomain contains the persistent state (i.e. what is, in Ethereum, stored in the blockchain) of each contract and a set of values referenced to by variable identifiers. The former comprises the name of the contract (C), the values of its state variables ($s\tilde{:}v$) and its balance (n). Again, we shall use the concatenation operator to append new entries to the blockchain.

Program $\mathcal{P}$ denotes a program composed by an expression e operating on the blockchain β, which contains instances of the contracts defined in CT. In Ethereum, the operations interacting with the blockchain (i.e. the transactions) can be either calls to contract functions, money transfers, or contract instantiations. As we said in Chapter 2, they are not written in Solidity; instead, usually they are JavaScript code (or code written in another language supported by Ethereum) interacting with an Ethereum Virtual Machine (EVM) thanks to the web3.js library (see, for instance, Listing 2.3 in Example 2.1 or Listing 2.16 in Example 2.10). Hence, a sort of two-level syntax exists: one for the expressions allowed at the top-level (i.e. to model the code interacting with the blockchain) and one for the ones used to define smart contracts (i.e. to model the Solidity code). For the sake of simplicity, this is not true anymore in FS. We unify the syntax and suppose that not only contracts, but also transactions, follow the grammar given in Figure 4.1. Hence, an FS program has the form $(CT, \beta, e_1; \ldots; e_n)$, where $e_1; \ldots; e_n$ is a sequence of operations interacting with the blockchain β, possibly modifying its state. In the paragraph below we clarify the meaning of e.

In brief, $\mathcal{P}$ represents a DApp: it comprises a list of transactions, the contract table containing, for any given contract name its code, and an initial blockchain. The execution operates and modifies the latter, which thus can grow as long as the program runs. Differently from Ethereum, our blockchain does not contain the bytecode of the deployed contracts.

Expressions The metavariable e ranges over expressions. v denotes a value, x a variable, this a local variable referring to the contract being invoked. We assume an infinite set of variable names Var and such that $x \in Var$, and x may also be this, msg.value, or msg.sender. this is local to functions, and makes no sense outside a function definition. this.f is a pointer to a function defined in the context of the current contract (referenced to by this). We shall further explain the meaning of such function pointers below. msg is a variable containing information about the current function call: msg.sender contains the address of the caller, while msg.value represents the amount of Wei (the smallest Ether's sub-unit) sent to the callee: its scope is a transaction. Assuming e evaluates to a contract reference c, address(e) explicitly gives its address, $e.s$ reads the value of the state variable s. The expression $e.f.\mathsf{value}(e_2)(\tilde{e'})$ is the function call, which not only calls f in c, but also transfers e_2 Wei from msg.sender to c. We shall use $e.f(\tilde{e'})$ as a short form of $e.f.\mathsf{value}(0)(\tilde{e'})$. The variant with sender ($e.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e})$) is allowed only in a very specific scenario, and we shall further discuss it in Chapter 5. When e evaluates to an address, balance(e) returns its balance, that is, the amount of Wei it currently contains and $e.\mathsf{transfer}(e_2)$ transfers n Wei to it, supposing e_2 evaluates to n. The expression new $C.\mathsf{value}(e_2)(\tilde{e})$ deploys a new contract C on the blockchain, with an initial balance of e_2, whereas $C(e)$ retrieves the reference to the contract identified by the address obtained evaluating e, if any. As we said before, there is always at most only one contract reference corresponding to an address, and vice versa. Again, new $C(\tilde{e})$ is a short form for new $C.\mathsf{value}(0)(\tilde{e})$. The expression $e; e$ is sequential composition, $T\ x = e; e'$ is variable declaration (binding x in e'), $x = e$ and $e.s = e$ are variable assignment and state variable assignment, respectively. When e evaluates to a mapping value, $e[e']$ reads the value corresponding to the evaluation of key e', whereas $e[e' \to e'']$ modifies it. The expression if e then e else e defines the usual `if` expression. For the sake of simplicity, FS does not include any boolean or arithmetic operators: their behavior is well known, and so is their type safety. $e_1.\mathsf{value}(e_2)(\tilde{e})$ represents a call to a function used as a value, supposing e_1 evaluates to $c.f$ and e_2 to n. This expression not only invokes the function f in c with parameters $\tilde{e}$, but also sends n Wei to c. Lastly, revert is a term used to signal erroneous situations. It is very similar to exceptions, but it cannot be caught. Hence, when a revert is thrown the current transaction is aborted, and every partial result is discarded (to maintain the consistency and atomicity properties of transactions). We shall further discuss it in Chapter 5. revert can be thrown both by the runtime and by programmers (thus modeling Solidity's `revert`, `require`, and `assert`).

Types T ranges over types. bool and unit have the expected meaning. uint indicates unsigned integers, formally defined by $\mathbb{N}^+ = \{n \in \mathbb{N} \mid n \geq 0\}$. address represents Ethereum's addresses, whose values range in set $\mathbb{A} = \{0x\ n_{hex} \mid n \in \mathbb{N} \wedge |n_{hex}| = 40\}$. In brief, $\mathbb{A}$ is the set of all the natural numbers, taken in their hexadecimal representation, whose length (computed as the number of hexadecimal digits) is equal to 40. Clearly this is a finite set, but there are 2^{160} addresses in total, and they are practically enough not to worry about their exhaustion.

We allow three complex types: functions, mappings and contracts. The latter is represented by C. $\tilde{T} \to T$ indicates functions, taking as input a tuple of typed values and returning a single value of type T. unit is used as a return type when the function does not return anything. mapping$(T_1 \Rightarrow T_2)$ indicates a total function $T_1 \to T_2$. T_1 can only be a simple type, i.e. unit, bool, address, and uint. In Solidity, a mapping is similar to a hash table, where every possible key is initialized to the "zero" value of the corresponding value type. This "zero" function is defined by Definition 4. This is

also the reason why we represent mappings as total functions: for each value v_1 in T_1 there is always one and only one value v_2 in T_2 matching v_1: v_2 is either the "zero" (or default) value, or a value previously set. Assigning another value v_2' in T_2 to the key v_1 simply overwrites the previous one, exactly as what happens when changing the value corresponding to a given key in a hash table.

Definition 4 (Zero)**.** The zero function is inductively defined over types as follows:

$$
\begin{array}{ll}
\mathrm{zero}(\mathsf{bool}) & \triangleq \ \mathsf{false} \\
\mathrm{zero}(\mathsf{unit}) & \triangleq \ \mathsf{u} \\
\mathrm{zero}(\mathsf{uint}) & \triangleq \ 0 \\
\mathrm{zero}(\mathsf{address}) & \triangleq \ 0_a \\
\mathrm{zero}(\mathrm{C}) & \triangleq \ \mathrm{null} \\
\mathrm{zero}(\tilde{\mathrm{T}_1} \to \mathrm{T}_2) & \triangleq \ 0_{T_2} \\
\mathrm{zero}(\mathsf{mapping}(\mathrm{T}_1 \Rightarrow \mathrm{T}_2)) & \triangleq \ 0_{\{\}}
\end{array}
$$

Where 0_a represents the address composed by only zeros and 0_{T_2} a constant function returning the "zero" value of T_2. $0_{\{\}}$ indicates the total function mapping every value k of type T_1 in 0_{T_2}.

Values v ranges over values. true and false have the expected meaning. n represents unsigned integers and $a \in \mathbb{A}$ indicates an address. u is the only value allowed for type unit. M indicates a mapping (i.e. a total function) and can be thought of as a set of key-value pairs, where the value can either have been defined by a user or be the "zero" one. Keys and values are, from a syntactic point of view, values v in FS. Lastly, c denotes references to contract instances and $c.f$ indicates a reference to a function defined in the declaration of the contract c. Solidity does not allow lambda expressions yet, and the only possible values for the function type are represented by "references" to contract functions. This is the meaning of $c.f$. Note there can be no ambiguity between such "function pointers" and function calls, not even considering the short syntax. When f is followed by either value or parenthesis, and possibly some parameters, that is a function call. When it is not, then it is a function pointer. In both cases, type rules will allow only functions defined in the contract declaration of c. Also note that programmers can reference fb.

4.2 Auxiliary functions

Here we define some auxiliary functions that extract information from FS programs:

- The function fv extracts the free variables of an expression in FS. Its full definition is given in Figure 4.2.

- The function fn extracts the free names (free references and free addresses) of an expression in FS. Its full definition is given in Figure 4.2.

Term		fv	fn
true	=	$\emptyset$	$\emptyset$
false	=	$\emptyset$	$\emptyset$
n	=	$\emptyset$	$\emptyset$
a	=	$\emptyset$	$\{a\}$
u	=	$\emptyset$	$\emptyset$
M	=	$\emptyset$	$\emptyset$
c	=	$\emptyset$	$\{c\}$
$c.f$	=	$\emptyset$	$\{c\}$
x	=	$\{x\}$	$\emptyset$
this	=	this	$\emptyset$
msg.sender	=	msg.sender	$\emptyset$
msg.value	=	msg.value	$\emptyset$
balance(e)	=	$\mathrm{fv}(e)$	$\mathrm{fn}(e)$
address(e)	=	$\mathrm{fv}(e)$	$\mathrm{fn}(e)$
$e.s$	=	$\mathrm{fv}(e)$	$\mathrm{fn}(e)$
e_1.transfer(e_2)	=	$\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)$	$\mathrm{fn}(e_1) \cup \mathrm{fn}(e_2)$
new C.value$(e_1)(\tilde{e})$; e_2	=	$\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \cup \bigcup \mathrm{fv}(e_i)$	$\mathrm{fn}(e_1) \cup \mathrm{fn}(e_2) \cup \bigcup \mathrm{fn}(e_i)$
$C(e)$	=	$\mathrm{fv}(e)$	$\mathrm{fn}(e)$
$e_1;\ e_2$	=	$\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)$	$\mathrm{fn}(e_1) \cup \mathrm{fn}(e_2)$
$e_1.f$.value$(e_2)(\tilde{e_3})$	=	$\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \cup \bigcup \mathrm{fv}(e_{3i})$	$\mathrm{fn}(e_1) \cup \mathrm{fn}(e_2) \cup \bigcup \mathrm{fn}(e_{3i})$
$e_1.f$.value(e_2).sender$(e_3)(\tilde{e_4})$	=	$\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \cup \mathrm{fv}(e_3) \cup \bigcup \mathrm{fv}(e_{4i})$	$\mathrm{fn}(e_1) \cup \mathrm{fn}(e_2) \cup \mathrm{fn}(e_3) \cup \bigcup \mathrm{fn}(e_{4i})$
e_1.value$(e_2)(\tilde{e_3})$	=	$\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \cup \bigcup \mathrm{fv}(e_{3i})$	$\mathrm{fn}(e_1) \cup \mathrm{fn}(e_2) \cup \bigcup \mathrm{fn}(e_{3i})$
$T\ x = e_1;\ e_2$	=	$\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \backslash \{x\}$	$\mathrm{fn}(e_1) \cup \mathrm{fn}(e_2)$
$x = e$	=	$\{x\} \cup \mathrm{fv}(e)$	$\mathrm{fn}(e)$
$e_1.s = e_2$	=	$\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)$	$\mathrm{fn}(e_1) \cup \mathrm{fn}(e_2)$
$e_1[e_2]$	=	$\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)$	$\mathrm{fn}(e_1) \cup \mathrm{fn}(e_2)$
$e_1[e_2 \to e_3]$	=	$\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \cup \mathrm{fv}(e_3)$	$\mathrm{fn}(e_1) \cup \mathrm{fn}(e_2) \cup \mathrm{fn}(e_3)$
if e_1 then e_2 else e_3	=	$\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \cup \mathrm{fv}(e_3)$	$\mathrm{fn}(e_1) \cup \mathrm{fn}(e_2) \cup \mathrm{fn}(e_3)$
$\emptyset$	=	$\emptyset$	$\emptyset$
$\beta \cdot [x \mapsto v]$	=	$\mathrm{fv}(\beta) \cup \{x\}$	$\mathrm{fn}(\beta) \cup \mathrm{fn}(v)$
$\beta \cdot [(c, a) \mapsto (C, \tilde{sv}, b)]$	=	$\mathrm{fv}(\beta)$	$\mathrm{fn}(\beta) \cup \{c\} \cup \{a\} \cup \mathrm{fn}(v)$

Figure 4.2: Free variables and names

Lastly, Figure 4.3 defines the substitution of free variables in an expression in FS. In such definition x is meant to be a variable identifier, and it may also be this, msg.sender, and msg.value.

Remark (Syntactic assumptions)**.** We make some assumptions regarding what can be written in FS. As we said, the syntax is intended to be liberal, with the type system ruling out the ill-formed expression. However, some syntactic restrictions, analogous to the ones in Solidity, are not checked with types:

- the expression $e_1.f$.value(e_2).sender$(e_3)(\tilde{e})$ models the invocation of a transaction. Hence, we assume it cannot be contained within the code of FS contracts, but only in the top-level expression e of a given program (CT, β, e);
- the expressions $e_1.f$.value$(e_2)(\tilde{e})$ and e_1.transfer(e_2) can be contained only into FS contracts and thus, given a program (CT, β, e), they cannot appear in the expression e.

For instance, the first one models the syntax of web3.js requiring an explicit specification of the sender (which it indicates with `from`) of each transaction. This parameter will be the first value for the variable `msg.sender`. On the other hand, such variable will be automatically set by the runtime for every further function (i.e. message) call,

$$
\begin{array}{lcl}
\mathsf{true}\{x := e'\} & = & \mathsf{true} \\
\mathsf{false}\{x := e'\} & = & \mathsf{false} \\
n\{x := e'\} & = & n \\
a\{x := e'\} & = & a \\
\mathsf{u}\{x := e'\} & = & \mathsf{u} \\
M\{x := e'\} & = & M \\
c\{x := e'\} & = & c \\
c.f\{x := e'\} & = & c.f \\
x\{x := e'\} & = & e' \\
y\{x := e'\} & = & y \qquad (y \neq x) \\
\mathsf{this}.f\{x := e'\} & = & \mathsf{this}.f \qquad (x \neq \mathsf{this}) \\
\mathsf{this}.f\{\mathsf{this} := e'\} & = & e'.f \\
\mathsf{balance}(e)\{x := e'\} & = & \mathsf{balance}(e\{x := e'\}) \\
\mathsf{address}(e)\{x := e'\} & = & \mathsf{address}(e\{x := e'\}) \\
e.s\{x := e'\} & = & e\{x := e'\}.s \\
(e_1.\mathsf{transfer}(e_2))\{x := e'\} & = & e_1\{x := e'\}.\mathsf{transfer}(e_2\{x := e'\}) \\
(\mathsf{new}\ C.\mathsf{value}(e_1)(\tilde{e}))\{x := e'\} & = & \mathsf{new}\ C.\mathsf{value}(e_1\{x := e'\})(e_i\{x := e'\}_{1 \leq i \leq n}) \\
(C(e))\{x := e'\} & = & e\{x := e'\} \\
(e_1; e_2)\{x := e'\} & = & e_1\{x := e'\}; e_2\{x := e'\} \\
(T\ y = e)\{x := e'\} & = & T\ y = e\{x := e'\} \qquad \text{if } y \notin \mathrm{fv}(e') \\
(y = e)\{x := e'\} & = & y = e\{x := e'\} \\
(e_1.s = e_2)\{x := e'\} & = & e_1\{x := e'\}.s = e_2\{x := e'\} \\
(e_1[e_2])\{x := e'\} & = & e_1\{x := e'\}[e_2\{x := e'\}] \\
(e_1[e_2 \to e_3])\{x := e'\} & = & e_1\{x := e'\}[e_2\{x := e'\} \to e_3\{x := e'\}] \\
(e_1.f.\mathsf{value}(e_2)(\tilde{e}))\{x := e'\} & = & e_1\{x := e'\}.f.\mathsf{value}(e_2\{x := e'\})(e_{i1 \leq i \leq n}\{x := e'\}) \\
(e_1.\mathsf{value}(e_2)(\tilde{e}))\{x := e'\} & = & e_1\{x := e'\}.\mathsf{value}(e_2\{x := e'\})(e_{i1 \leq i \leq n}\{x := e'\}) \\
(e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}))\{x := e'\} & = & e_1\{x := e'\}.f.\mathsf{value}(e_2\{x := e'\}). \\
 & & \quad \mathsf{sender}(e_3\{x := e'\})(e_{i1 \leq i \leq n}\{x := e'\}) \\
(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)\{x := e'\} & = & \mathsf{if}\ e_1\{x := e'\}\ \mathsf{then}\ e_2\{x := e'\}\ \mathsf{else}\ e_3\{x := e'\} \\
\mathsf{revert}\{x := e'\} & = & \mathsf{revert} \\
\mathsf{return}\ e\{x := e'\} & = & e
\end{array}
$$

Figure 4.3: Substitution of free variables in FS expressions

and will not be settable by programmers. Similarly, `transfer` may be invoked only in the context of a contract (i.e. within one of the functions it defines), and may not be called explicitly from the code interacting with the blockchain (e.g. the one written in JavaScript).

Differently from Ethereum, our `new` does not allow the specification of a top-level sender. As with any other transaction, in Ethereum a `new` has to specify the address of the initiating account (e.g. the parameter `from` in web3.js) that pays the transaction fee as well as may send Ether to the newly deployed instance. For the sake of simplicity we model only one version of `new`, $\mathsf{new}\ C(\tilde{e})$.

4.3 FS by example

We now informally compare the expressiveness of Solidity and FS. Example 4.1 analyzes the implementation of the functionality of a basic bank contract allowing users to deposit, withdraw or transfer an amount of Ether. Example 4.2 highlights deeper differences between the two languages. It models a blood bank, where donors may donate blood upon explicit doctor authorization. It shows how two contracts can interact together. Lastly, in Example 4.3, we demonstrate how to use function as values.

Example 4.1 (The first FS smart contract)**.** Listing 4.1 shows a very basic bank contract written in Solidity. It contains only one state variable, `balances`, representing the

balance for each account. An amount of Wei is sent via the deposit function (indeed marked as payable), which reads the amount of money from msg.value and increments the balance of msg.sender. A getter function, getBalance, allows each user to get to know their balance. withdraw and transfer are very similar to each other, and allow users to transfer an amount of Wei to their own Ethereum account (that is from an account internal to Bank to an Ethereum account) or someone else's Bank account (that is from an account internal to Bank to another), respectively. Note that there is no fallback function, since the only legal way to send money to this bank is through the deposit function (thus incrementing someone's balance). As a final note, when an account's balance is insufficient to accomplish either a withdrawal or a money transfer, no error or exception is thrown: functions simply do nothing.

```
1  contract Bank {
2      mapping (address => uint) private balances;
3
4      constructor() public {}
5
6      function deposit() external payable {
7          balances[msg.sender] += msg.value;
8      }
9
10     function transfer(address to, uint amount) external {
11         if (balances[msg.sender] >= amount) {
12             balances[msg.sender] -= amount;
13             balances[to] += amount;
14         }
15     }
16
17     function getBalance() external view returns (uint) {
18         return balances[msg.sender];
19     }
20
21     function withdraw(uint amount) external {
22         if (balances[msg.sender] >= amount) {
23             balances[msg.sender] -= amount;
24             msg.sender.transfer(amount);
25         }
26     }
27 }
```

Listing 4.1: Basic Bank contract in Solidity syntax

Listing 4.2, on the other hand, shows the same contract, but written using FS syntax. The very first big difference is that FS, in contrast to Solidity, explicitly has to initialize every state variable, including a mapping. Note also the use of this to disambiguate the formal parameter and the state variable. Secondly, the syntax for defining functions is a little different, the one of FS being more similar to Java. Furthermore, unit is explicitly used to represent void. Function bodies consist of a single expression, return e, where e is composed in an inductive way, following the rules given in Figure 4.1. Taking transfer's body as an example, note that the if expression is used right after return and that u (the only value of type unit) is explicitly added as a last term.

```
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
    }
}
```

Listing 4.2: Basic Bank contract in FS syntax

Example 4.1 clearly shows that our small language is expressive enough to encode, with slight modifications, a simple though powerful, Solidity contract.

Example 4.2 (DApp translation in FS)**.** We translate the code in Example 2.10 in FS syntax. The syntax we have defined is a bit more restrictive than Solidity's, and does not allow some of these things. This is clearly shown in Listing 4.3.

```
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
    }
}
```

Listing 4.3: Donor and BloodBank contracts in FS syntax

Starting from BloodBank, first note that each state variable value must be explicitly passed as a parameter to the constructor, since there is no default value. Thus, both

healty and blood become parameters. Secondly, a constructor body is limited to a sequence of initializations, and arbitrary expressions (what we called e in Figure 4.1) are not allowed. Hence, the doctor (i.e. msg.sender) becomes a parameter, too. Thirdly, no getter function is automatically added (remember FS does not take into account visibility keywords), so two new functions, getDoctor and getBlood, have to be explicitly implemented. To disambiguate, we name them with the usual nomenclature, instead of using the same name as the variables they refer to. Donor shows another limitation. As we said, no expression is allowed in a constructor body, so we cannot just use BloodBank(bank) to initialize this.bank. We chose to retrieve the contract reference each time donate is called. In general, due to this limitation, contracts in FS do not contain state variables with a contract type. Instead, those variables have type address, and an explicit cast is used to obtain the required functionality. This may decrease the global efficiency, but is semantically equivalent to what is shown in Listing 2.15. Lastly, `require` has to be "expanded" in FS, using an if expression and explicitly issuing a revert in one branch.

In Example 5.1 we shall show the evaluation of the following expression:

$$
\begin{aligned}
e := \quad & EOC\ x_{doctor} = \mathsf{new}\ EOC();\\
& EOC\ w_{humanDonor} = \mathsf{new}\ EOC();\\
& BloodBank\ y_{bank} = \mathsf{new}\ BloodBank(0_{\{\}}, \mathsf{address}(x_{doctor}), 0);\\
& Donor\ z_{donor} = \ \mathsf{new}\ Donor(5000, \mathsf{address}(y_{bank}));\\
& y_{bank}.setHealth.\mathsf{sender}(\mathsf{address}(x_{doctor}))(\mathsf{address}(z_{donor}), \mathsf{true});\\
& z_{donor}.donate.\mathsf{sender}(\mathsf{address}(w_{humanDonor}))(500)
\end{aligned}
$$

With a few changes, e is the translation in FS of Listing 2.16. First we deploy two instances of EOC, x_{doctor} and $w_{humanDonor}$. This was different in our DApp, because in Ethereum EOAs do not correspond to any contracts, but the use we make of these two instances is exactly the same, as we shall see soon. Then we deploy an instance of $BloodBank$ specifying the address of x_{doctor} as a parameter. This notation may seem a little heavy, but corresponds to the use of the variable `doctorAddress` in Listing 2.16 (Line 16). Lastly, we deploy an instance of $Donor$ (Lines 26-32).

e does not contain the call to $isHealty$, but it does contain the one to $setHealth$ to set at true the health status of z_{donor}. This is an example of a top-level code (and of a top-level transaction), and translates in web3.js as a function call using `sendTransaction`. Note how our syntax slightly differs from the one we saw before: in FS everything is a transaction and there are no `view` functions, thus there is no need to specify `call` or `sendTransaction`. We have also modeled less transaction parameters than web3.js: the value (i.e. the amount of Wei to be sent) and the sender (i.e. `from` in web3.js) are explicitly written before the function to be called. Here our syntax is a little heavier than the one we have seen before, since we have to pass address(x_{doctor}) instead of just `doctorAddress`. However, this is unavoidable since we modeled EOAs with contracts, and find a direct correspondence with the web3.js way to get an address given a contract instance (`Contract.at(deployedContract.address)`). In fact, notice that the way we pass the address of the donor as a parameter of `setHealth`: in Listing 2.16, at Line 59, we used `deployedDonor.address` as well as in e we used address(z_{donor}).

Afterwards, we deploy another instance of EOC, corresponding to `humanDonorAddress` in Listing 2.16, and specify it as a sender of a second top-

level call (i.e. a transaction) to $Donor.donate$ to donate 500ml of blood. Again, we have to specify the sender explicitly using $\mathsf{address}(w_{humanDonor})$.

In Example 4.2 we went deeper in the functionality offered by Solidity. We highlighted the differences with FS, but figured out that the latter is still expressive enough to encode a more complex contract.

Example 4.3 (Functions as values in FS)**.** Here we show how to translate Example 2.4 in FS. The main difference is that we do not take into account any function modifiers, and thus the code becomes simpler. The specification of the function type is also more readable, changing from `function (uint) view external returns (uint) f` to $\mathsf{uint} \to \mathsf{uint}$. The contracts are listed in Listing 4.4.

```
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
    }
}
```

Listing 4.4: Functions as values in FS syntax

In Example 5.3 we shall show the evaluation of the following expression:

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

Chapter 5

Operational Semantics

In this chapter we describe the operational semantics of FS. Before going into the rules, we introduce the run-time syntax of FS, explain the form of our judgments and formally define transactions. We then introduce some lookup functions extracting information from a contract definition and auxiliary predicates used to evaluate expressions. Lastly, we formalize the semantics and give some evaluation examples.

5.1 Run-time syntax

Figure 5.1 presents the run-time syntax of the FS language.

$$
\begin{array}{lrcl}
(Call\ stack) & \sigma & ::= & \beta \mid \sigma \cdot a \\
(Configuration) & \mathcal{C} & ::= & \langle \beta, \sigma, e \rangle
\end{array}
$$

Figure 5.1: The run-time syntax of the Featherweight Solidity language

The paragraphs that follow explain in detail σ, and $\mathcal{C}$.

Call stack σ is a call stack tracking the nesting of function calls (actually, the addresses of their enclosing contracts) within the execution of a transaction. To simplify the operational semantics when executing a transaction, we require the initial element of the call stack to be a blockchain, namely a copy of the blockchain at the time the (top-level) transaction starts. Such a copy, held at the bottom of the call stack, will be useful in case of abort, when the state of the blockchain has to be unrolled to the beginning of the transaction itself. On the other hand, if the transaction successfully commits, the modified blockchain will be copied over the σ's copy.

Hence, the call stack is initially empty (β) and grows as long as contracts interact with one another. If a top-level call targets a contract C_1 with address a_1, σ is β before the top-level call and becomes $\beta \cdot a_1$ when the function of C_1 is executed. If, later on, C_1 interacts with another contract C_2 at address a_2, σ becomes $\beta \cdot a_1 \cdot a_2$ in the context of C_2, and so on and so forth. When a function returns an element is popped from σ, until it becomes β again.

Configuration At any moment the state of the execution is described by a triple $\langle \beta, \sigma, e \rangle$, comprising a temporary working copy of the blockchain, the call stack, and

the expression to be evaluated. Note how, at the beginning of each (top-level) transaction, the configuration is of the form $\langle \beta_0, \beta_0, e \rangle$. A successfully evaluated transaction ends in $\langle \beta, \beta_0, v \rangle$ and the updated blockchain β is committed. Otherwise, if an error occurs, a configuration of the form $\langle \beta, \beta_0 \cdot \tilde{a}, \mathsf{revert} \rangle$ is reached, and the modified blockchain β is discarded in order to restore the initial one (β_0), thus modeling the rollback of the aborted transaction.

5.1.1 Free variables and names

Now that we have explained the run-time syntax, we can extend the definitions of functions fv and fn, given in Figure 4.2, to support σ.

Term		fv	fn
$\sigma \cdot a$	$=$	$\mathrm{fv}(\sigma)$	$\mathrm{fn}(\sigma) \cup \{a\}$
$\langle \beta, \sigma, e \rangle$	$=$	$\mathrm{fv}(\beta) \cup \mathrm{fv}(\sigma) \cup \mathrm{fv}(e)$	$\mathrm{fn}(\beta) \cup \mathrm{fn}(\sigma) \cup \mathrm{fn}(e)$

Figure 5.2: Free variables and names for FS run-time syntax

5.2 Judgments

The transition relation and the operational semantics of FS are formally defined by Definition 5 and Definition 6, respectively.

Definition 5 (Transition relation)**.** The transition relation representing one step of the evolution of an expression in FS has the following form:

$$\mathcal{C} \longrightarrow \mathcal{C}$$

that is:

$$\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$$

Where β and σ change according to the actual expression represented by e. This transition relation is inductively defined over the axioms and rules in Section 5.5.

Given a program $\mathcal{P} = (CT, \beta, e)$, the initial configuration is $\langle \beta, \beta, e \rangle$.

Definition 6 (Operational semantics)**.** The operational semantics is defined as the reflective, transitive closure of $\longrightarrow$. It is indicated with $\longrightarrow^*$ and formally defined as follows:

$$\mathcal{C} \longrightarrow^* \mathcal{C} \qquad \frac{\mathcal{C} \longrightarrow^* \mathcal{C}' \qquad \mathcal{C}' \longrightarrow \mathcal{C}''}{\mathcal{C} \longrightarrow^* \mathcal{C}''}$$

The semantics for transactions defined in this way is simple: a top-level expression (i.e. a transaction) e_i operates on the blockchain, modifying it. If there is another expression e_{i+1}, the latter operates on this modified copy. Otherwise, if there is no e_{i+1}, e_i evaluates to the final state. Definition 7 formalizes this transition relation.

Definition 7 (Transition relation for transactions)**.** The transition relation representing the evaluation of a list of transactions in FS has the following form:

$$\langle \beta_{i-1}, e_i; \ldots; e_n \rangle \Longrightarrow \langle \beta_i, e_{i+1}; \ldots; e_n \rangle$$

Where $1 \leq i \leq n$, and e_i represents a ***single*** transaction. The initial state for a program $\mathcal{P} = (CT, \beta, e)$ is $\langle \beta, e \rangle$. Note that the tuple $e_1; \ldots; e_n$ represents the sequence of

top-level transactions to be executed. The transition relation $\Rightarrow$ models the operational semantics of the Ethereum nodes, and is inductively defined over the axioms and rules in Figure 5.3.

$$\text{(SUCCESS)}\quad \frac{\langle \beta, \beta, e\rangle \longrightarrow^* \langle \beta', \beta, v\rangle}{\langle \beta, e\rangle \Longrightarrow \langle \beta', v\rangle} \qquad \text{(FAILURE)}\quad \frac{\langle \beta, \beta, e\rangle \longrightarrow^* \langle \beta', \beta, \mathsf{revert}\rangle}{\langle \beta, e\rangle \Longrightarrow \langle \beta, \mathsf{revert}\rangle}$$

$$\text{(COMMIT)}\quad \frac{\langle \beta, \beta, e_1\rangle \longrightarrow^* \langle \beta', \beta, v_1\rangle}{\langle \beta, e_1; e_2\rangle \Longrightarrow \langle \beta', e_2\rangle} \qquad \text{(ABORT)}\quad \frac{\langle \beta, \beta, e_1\rangle \longrightarrow^* \langle \beta', \beta, \mathsf{revert}\rangle}{\langle \beta, e_1; e_2\rangle \Longrightarrow \langle \beta, \mathsf{revert}\rangle}$$

Figure 5.3: Operational semantics rules for transactions in FS

The initial configuration for $\mathcal{P}$ is, as we said, $\langle \beta, e_1; \ldots; e_n\rangle$, and the evaluation may lead to either a value or a revert. In the former case everything was successfully evaluated and the program terminates with a final value; in symbols: $\langle \beta_n, v\rangle$. In the latter, something went wrong and the evaluation could not further proceed: $\langle \beta_i, \mathsf{revert}\rangle$. In Ethereum, miner nodes discard erroneous transactions and do not include them into their blocks; validation nodes, on the other hand, discard the entire block if any transactions return an error. The semantics in Definition 7 formalizes the latter behavior.

COMMIT models a successful transaction: e_1 evaluates toward a value v_1 and makes β become β'. v_1 is discarded and the execution proceeds with e_2, evaluated over β'. ABORT models the opposite. Here e_1 evaluates toward a revert: any changes made to β are discarded and the execution does not further proceed. Bearing in mind what we said in Section 2.3.4, $\Longrightarrow$ actually models the behavior of validation nodes: if any transactions returns a revert the entire process is stopped without further proceeding with the subsequent transactions.

Lastly, Definition 8 formalizes the operational semantics for transactions in FS.

Definition 8 (Operational semantics for transactions)**.** The operational semantics for transactions is defined as the reflective, transitive closure of $\Longrightarrow$. It is indicated with $\Longrightarrow^*$ and formally defined as follows:

$$\langle \beta, e\rangle \Longrightarrow^* \langle \beta, e\rangle \qquad \frac{\langle \beta, e\rangle \Longrightarrow^* \langle \beta', e'\rangle \qquad \langle \beta', e'\rangle \Longrightarrow \langle \beta'', e''\rangle}{\langle \beta, e\rangle \Longrightarrow^* \langle \beta'', e''\rangle}$$

5.3 Lookup functions

Here we define the following three functions:

- $\mathrm{sv}(C)$, defined over contract names and returning the tuple of state variables defined in the declaration of C;
- $\mathrm{fbody}(C, f, \tilde{v})$, returning a pair $(\tilde{x}, e)$ corresponding to the formal parameters and the body of the function f in the definition of C. It also checks whether $|\tilde{x}| = |\tilde{v}|$;
- $\mathrm{ftype}(C, f)$, returning the type of the function f contained in C;

These functions are formally defined in Figure 5.4.

State variable lookup:

$$\frac{CT(C) = \mathsf{contract}\ C\ \{\tilde{T}\,s;\ K\ \tilde{F}\}}{\mathrm{sv}(C) = \tilde{T}\,s}$$

Function body lookup:

$$\frac{CT(C) = \mathsf{contract}\ C\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \quad T\ f\ (\tilde{T}\,x)\ \{\mathsf{return}\ \ e\} \in \tilde{F} \quad |\tilde{x}| = |\tilde{v}|}{\mathrm{fbody}(C, f, \tilde{v}) = (\tilde{x}, \mathsf{return}\ e)}$$

$$\frac{CT(C) = \mathsf{contract}\ C\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \quad f \notin \tilde{F} \vee (T\ f\ (\tilde{T}\,x)\ \{\mathsf{return}\ \ e\} \in \tilde{F} \wedge |\tilde{v}| \neq |\tilde{x}|)}{\mathrm{fbody}(C, f, \tilde{v}) = (\{\}, \mathsf{return\ revert})}$$

Function signature lookup:

$$\frac{CT(C) = \mathsf{contract}\ C\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \qquad B\ f\ (\tilde{A}\,x)\ \{\mathsf{return}\ e;\} \in \tilde{F}}{\mathrm{ftype}(C, f) = \tilde{A} \to B}$$

Figure 5.4: Lookup functions

fbody is a bit complex and its definition is not trivial. Consider the first case, $\mathrm{fbody}(C, f, \tilde{v})$. It checks whether the function f appears in the list of functions F of C. If so, it also checks the length of the actual parameters tuple ($\tilde{v}$), comparing it with the one of the formal parameters tuple ($\tilde{x}$). If they match, everything is fine and the body of f is correctly returned, together with the list of formal parameters. Otherwise, the function is undefined, either because f does not appear in F or because of a mismatch in the tuples length. In this case no body can be retrieved and we signal the error by throwing a revert.

5.4 Auxiliary predicates

Here we define some predicates used to make the rules simpler and more readable. They extract information from the components of a configuration $\mathcal{C}$, in particular:

- $\mathrm{uptbal}(\beta, a, n)$ (Figure 5.5) returns a blockchain where the balance of a has been incremented/decremented of n;

- $\mathrm{Top}(\sigma)$ (Figure 5.6), defined on σ, returns the top of the call stack, if any (i.e. $\sigma = \sigma' \cdot a$), or $\emptyset$, if the call stack is empty.

$$\mathrm{uptbal}(\beta, a, n) = \begin{cases} \beta[(c, a) \mapsto (C, s\tilde{:}v, n' + n)] & \textit{if } \hat{\beta}(a) = c \wedge \\ & \wedge \beta(a, c) = (C, s\tilde{:}v, n') \wedge \\ & \wedge n' + n > 0 \ (n \in \mathbb{Z}) \\ \bot & \textit{if } \hat{\beta}(a) = c \wedge \\ & \wedge \beta(a, c) = (C, s\tilde{:}v, n') \wedge \\ & \wedge n' + n < 0 \\ \bot & \textit{if } a \notin \mathrm{dom}(\beta) \\ \bot & \textit{if } a = \emptyset \end{cases}$$

Figure 5.5: Balance update

Note that $\mathrm{dom}(\mathrm{uptbal}(\beta, \mathrm{a}, \pm\mathrm{n})) = \mathrm{dom}(\beta)$. In other words, it is not defined over addresses not contained in β. Before decrementing, $\mathrm{updatebalance}$ also checks whether the balance is greater or equal to n. Here, we suppose $n \in \mathbb{Z}$, and return $\bot$ if the balance is not sufficient to accomplish the decrement as well as if the address a is not known (i.e. $a \notin \mathrm{dom}(\beta)$ or $a = \emptyset$). As we shall see soon, this will result in a revert at run-time.

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

Figure 5.6: Top of the call stack

5.5 Operational semantics rules

In this section we list the operational semantics rules of the FS language. For a comprehensive view of them, see Appendix A.

Remark. In the following rules, whenever the call stack is empty ($\sigma = \beta$, or, alternatively, $\mathrm{Top}(\sigma) = \emptyset$) the expression poses as a transaction. When necessary, the rules will make a distinction between empty and non-empty stack. If no distinction is explicitly done, the rule applies to both cases.

If expression The rules modeling the if expression are as follows:

$$\text{(IF-TRUE)}\quad \frac{}{\langle \beta, \sigma, \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \rangle \longrightarrow \langle \beta, \sigma, e_1 \rangle}$$

$$\text{(IF-FALSE)}\quad \frac{}{\langle \beta, \sigma, \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \rangle \longrightarrow \langle \beta, \sigma, e_2 \rangle}$$

These rules have the standard meaning. The only thing that is worth to point out is that both the branches (*then* and *else*) are mandatory and cannot be omitted.

Sequential composition The rules for sequential composition are as follows:

$$\text{(SEQ-C)}\quad \frac{\sigma = \beta_0}{\langle \beta, \sigma, v; e \rangle \longrightarrow \langle \beta, \beta, e \rangle} \qquad \text{(SEQ-R)}\quad \frac{\sigma = \beta_0}{\langle \beta, \sigma, \mathsf{revert}; e \rangle \longrightarrow \langle \beta_0, \sigma, \mathsf{revert} \rangle}$$

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

These rules have to take into account different scenarios. First, when an expression like $v; e$ is found at the top level ($\mathrm{Top}(\sigma) = \emptyset$) then the last transaction was successful. In fact, it evaluated to a value v, which can be discarded to proceed with the next transaction e. Furthermore, since evaluating to a value is equivalent to a commit, we apply the changes and proceed with β. This scenario is modeled by SEQ-C. Secondly, $v; e$ might be at an inner level ($\mathrm{Top}(\sigma) = a$). In this case (SEQ) we cannot apply any changes, since the transaction is not over yet. Hence, we just proceed with the evaluation of e, discarding v. Lastly, a revert may have been thrown during the evaluation, so instead of $v; e$ the expression involving sequential composition is $\mathsf{revert}; e$ (SEQ-R). In this case we do not further proceed in evaluating the other transactions: this models the behavior of validation nodes, as said before.

We shall see later on in this Chapter how we handle a revert occurring not at the top level.

Variables The rules regarding variables are as follows:

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

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

When a variable x is declared we add it to β: this behavior is modeled by DECL. VAR and ASS simply access β to read or modify, respectively, the value corresponding to a variable x.

Mappings The rules regarding mappings are as follows:

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

As we said, a mapping can be seen as a total function $T_1 \to T_2$, where $\forall v \in \mathrm{dom}(M), \exists v_2$ such that $M(v_1) = v_2$. $\mathrm{dom}(M) = \{v \mid v \text{ is a value of type } T_1\}$, and v_2 is either a user-defined value or the "zero" value, according to Definition 4. Leveraging this property, MAPPSEL evaluates $M[v_1]$ by retrieving the corresponding v_2. MAPPASS works in a similar way. Considering M as a set of key-value pairs ($M = \{(v, \tilde{M}(v))\}$), we define a new mapping M' where every pair remains the same, except for the one having v_1 as a key. It is updated so that it becomes (v_1, v_2).

BALANCE and ADDRESS The rules regarding the expressions balance and address are as follows:

$$\text{(BALANCE)}\quad \frac{\beta(a) = (C, s\tilde{:}v, n)}{\langle \beta, \sigma, \mathsf{balance}(a)\rangle \longrightarrow \langle \beta, \sigma, n\rangle} \qquad \text{(ADDRESS)}\quad \frac{\hat{\beta}(c) = a}{\langle \beta, \sigma, \mathsf{address}(c)\rangle \longrightarrow \langle \beta, \sigma, a\rangle}$$

In Solidity, balance is a property of address values returning the balance of the account referred to by each address. To avoid confusion with contract state variables, we chose to define balance as an expression of the language. Note the short form in the abuse of notation to query β without an explicit reference to a contract. The complete form of this hypothesis is as follows: $\beta(c, a) = (C, s\tilde{:}v, b)$, $\exists c$ such that $(c, a) \in \mathrm{dom}(\beta)$. Furthermore, in Solidity an implicit cast is applied whenever a contract reference is used instead of an address value. To model such a behavior in FS, which does not have any implicit casts, we defined an explicit global function, address, extracting the address corresponding to a given reference, as said in Section 4.1.

Contract instantiation The rules regarding contract instantiation are as follows:

(NEW-1)
$$\frac{(c,a) \notin \mathrm{dom}(\beta) \qquad \mathrm{sv}(C) = \tilde{T}\,s \qquad |\tilde{v}| = |\tilde{s}| \qquad \mathrm{Top}(\sigma) \neq \emptyset}{\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) \cdot [(c,a) \mapsto (C, s\tilde{:}v, n)], \sigma, c \rangle}$$

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

(NEW-R)
$$\frac{\mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) = \bot \qquad \mathrm{Top}(\sigma) \neq \emptyset}{\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle}$$

NEW rules deploy a new contract on the blockchain. The number of typed values provided as input must be the same as the number of state variables in C. This is consistent with the definition of K, whose body contains only a sequence of assignments. This is slightly different from what Solidity allows. For example, we cannot explicitly reference either msg.sender or msg.value, or initialize the state starting from values with different types. In FS there must be a one-to-one correspondence between parameters and state variables. We already saw this behavior in Example 4.2, where BloodBank's constructor contained more parameters then the one written in Solidity. A contract may be deployed with an initial balance. To this end we allow programmers to specify the amount of Wei to send along with the call to new. Again, we shall omit the $\mathsf{value}(n)$ part whenever $n == 0$. NEW-1 appends to β a pair (c, a), where c is a fresh contract reference and a is a fresh address. In this context, "fresh" means "not in the domain". As we already pointed out, the uniqueness of the single components of the pair is ensured by definition. The rule ensures that the creator's balance is greater than the amount of Wei being sent. If not, the instantiation fails and a revert is thrown (rule NEW-R). NEW-1 requires the stack to be non-empty (i.e. $\mathrm{Top}(\sigma) \neq \emptyset$). This means that the call to new happens in the context of a contract (i.e. a function f of the contract at address $\mathrm{Top}(\sigma)$ is being executed). Hence, the creator (i.e. the contract calling new) is always well defined, and the balance update operates on an address $\mathrm{Top}(\sigma) \in \mathrm{dom}(\beta)$. On the contrary, if the call stack is empty (i.e. $\mathrm{Top}(\sigma) = \emptyset$), rule NEW-2 applies. In this case we suppose the amount of Wei comes from an external source and do not check any balances. In Ethereum, instead, the balance of the contract posing as a sender is decreased.

Cast The rules about casts are as follows:

(CONTRRETR)
$$\frac{\beta^C(a) = C \qquad \hat{\beta}(a) = c}{\langle \beta, \sigma, C(a) \rangle \longrightarrow \langle \beta, \sigma, c \rangle}$$

(CONTRRETR-R)
$$\frac{\beta^C(a) = C' \qquad C' \neq C}{\langle \beta, \sigma, C(a) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle}$$

Solidity allows programmers to retrieve a reference to a deployed contract: one can think of this operation as a cast. Given an address a, if it is stored on the blockchain,

$C(a)$ returns the contract reference c corresponding to a (CONTRRETR). If a corresponds to a contract with a different definition ($C' \neq C$), a revert is thrown (CONTRRETR-R).

State variables The rules to read and mutate state variables are as follows:

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

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

Given a contract reference and a state variable identifier, STATESEL returns the value of the required variable. In a similar way, STATEASS enables the mutation of such value.

Money transfer Money transfers are modeled by the following rules:

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

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

Money transfers are modeled by TRANSFER. In Solidity, whenever a contract sends Wei to another, the recipient's fallback function is invoked (note fb in the call of fbody). If the latter is not defined, fbody returns a revert. A call to transfer modifies the balance of both the caller and the callee in the expected way, decreasing the former and increasing the latter. What we said before about σ and e applies here, too. This rule requires σ to be non-empty (thus, if empty the evaluation goes stuck). This could seem a limitation, since also the top level contract may send Wei to another contract, but actually it is not: from the top-level the fallback function is directly invokable and allows programmers to send n Wei to a specific contract.

Again, we throw a revert if the balance is not enough (TRANSFER-R).

As said in Section 2.4.2, Solidity actually provides another way to send Wei: the function send. We also said transfer is the safe counterpart of send: if something fails during the fallback evaluation, a revert is thrown and then propagated until the top level to make the entire transaction fail. send does not do that, and simply returns a boolean flag, which is `false` if an exception had happened while running the fallback function. Except for this, their behavior is the same. Hence, we chose to model transfer because it is safer: the use of send is discouraged and may lead to harmful situations as well as subtle bugs.

Function calls Function calls are modeled by the following rules:

(CALL)

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

(CALLTOPLEVEL)

$$\frac{\hat{\beta}(c) = a \qquad \beta^C(c) = C \qquad \text{fbody}(C, f, \tilde{v}) = (\tilde{x}, e) \qquad \tilde{x} \notin \text{dom}(\beta) \\ \beta' = \text{uptbal}(\text{uptbal}(\beta_w, a, n), a', -n) \cdot [\tilde{x} \mapsto \tilde{v}] \qquad \text{Top}(\sigma) = \emptyset \\ e_s = e\{\mathsf{this} := c, \mathsf{msg.sender} := a', \mathsf{msg.value} := n\}}{\langle \beta, \sigma, c.f.\mathsf{value}(n).\mathsf{sender}(a')(\tilde{v}); e' \rangle \longrightarrow \langle \beta', \sigma \cdot a, e_s; e' \rangle}$$

(CALL-R)

$$\frac{\text{uptbal}(\beta, \text{Top}(\sigma), -n) = \bot}{\langle \beta, \sigma, c.f.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle}$$

(RETURN)

$$\frac{}{\langle \beta, \sigma \cdot a, \mathsf{return}\, v \rangle \longrightarrow \langle \beta, \sigma, v \rangle}$$

(CALLTOPLEVEL-R)

$$\frac{\text{uptbal}(\beta, a', -n) = \bot \qquad \text{Top}(\sigma) = \emptyset}{\langle \beta, \sigma, c.f.\mathsf{value}(n).\mathsf{sender}(a')(\tilde{v}); e' \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert}; e' \rangle}$$

(RETURN-R)

$$\frac{}{\langle \beta, \sigma \cdot a, \mathsf{return\ revert} \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle}$$

Function calls are modeled by CALL. value has the same meaning as before, and allows programmers to indicate the amount of Wei to send along with the invocation. The blockchain is modified by changing the balances of both the caller and the callee (decrementing, and respectively incrementing, their balance by n Wei). σ is extended in the expected way, by pushing the callee's address. Note that σ has to be non-empty. The purpose of RETURN is to pop an element from σ. We apply this behavior not only in the context of a "successful" return, but also when the expression being returned is a revert (RETURN-R).

CALLTOPLEVEL is a particular rule allowed only in a very specific scenario. CALL allows contracts to invoke functions, but it cannot be applied from the top level since there is no address to use as a sender. Hence, we let programmers specify the callee address a' in a function invocation so that we can pretend a' to be the actual callee. This way of calling functions is intended to model EOC contracts, which do not contain any code, invoking other deployed contracts. Thus, it models transactions started by EOAs. Note that this rule only applies at the top level, since only there the runtime operates on an empty stack (i.e. $\sigma = \beta$, or, alternatively, $\text{Top}(\sigma) = \emptyset$).

Again, before proceeding with the evaluation of the function's body, we check the callee's balance to see if it is grater that the amount of Wei being sent. If not, we throw a revert (CALL-R and CALLTOPLEVEL-R).

We assume a correct use of top and inner-level calls: a simple static inspection of the program can detect such violations and stop the compilation process.

Computation rules The only two FS computation rules are as follows:

$$\text{(CONG)}\quad \frac{\langle \beta, \sigma, e\rangle \longrightarrow \langle \beta', \sigma', e'\rangle}{\langle \beta, \sigma, E[e]\rangle \longrightarrow \langle \beta', \sigma', E[e']\rangle} \qquad \text{(REVERT)}\quad \frac{}{\langle \beta, \sigma, E[\mathsf{revert}]\rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert}\rangle}$$

In the two rules above we made use of evaluation context, formally defined by Figure 5.7, to reduce the number of computation rules. Contexts contain a single hole, written [], inside them. $E[e]$ represents the expression obtained by replacing the hole in context E with the expression e.

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

Figure 5.7: Evaluation context

5.6 Operational semantics by example

This section shows how the above rules are applied. Example 5.1 is about the evaluation of contract creations and function calls, Example 5.2 describes money transfers, and Example 5.3 shows how function values are used.

Example 5.1. We make use of the FS code listed in Example 4.2, and show how the following expression is evaluated in $\langle \emptyset, \emptyset, e\rangle$:

$$\begin{aligned} e := \quad & EOC\ x_{doctor} = \mathsf{new}\ EOC(); \\ & BloodBank\ y_{bank} = \mathsf{new}\ BloodBank(0_{\{\}}, \mathsf{address}(x_{doctor}), 0); \\ & Donor\ z_{donor} = \ \mathsf{new}\ Donor(5000, \mathsf{address}(y_{bank})); \\ & y_{bank}.setHealth.\mathsf{sender}(\mathsf{address}(x_{doctor}))(\mathsf{address}(z_{donor}), \mathsf{true}) \end{aligned}$$

Where EOC was defined by Definition 1.

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

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

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

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

We omit the evaluation of $BloodBank\ y_{bank} = \mathsf{new}\ BloodBank(0_{\{\}}, \mathsf{address}(x_{doctor}), 0)$ and $Donor\ z_{donor} = \mathsf{new}\ Donor(5000, \mathsf{address}(y_{bank}))$, which is as before. Let
$\beta_1 = \sigma_1 = \emptyset \cdot [(c_{doctor}, a_{doctor}) \mapsto (EOC, \epsilon, 0)] \cdot [x_{doctor} \mapsto c_{doctor}]$
$\cdot [(c_{bank}, a_{bank}) \mapsto (BloodBank, healty = 0_{\{\}}; doctor = a_{doctor}; blood = 0, 0)]$
$\cdot [y_{bank} \mapsto c_{bank}] \cdot [(c_{donor}, a_{donor}) \mapsto (Donor, blood : 5000; bank : a_{bank}, 0)]$
$\cdot [z_{donor} \mapsto c_{donor}]$. The evaluation goes on as follows:

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

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

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

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

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

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

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

Let β_2 be the blockchain at this point.
$\langle \beta_2, \underline{\sigma_1}, \mathsf{u} \rangle$

Note the application of CALLTOPLEVEL when $y_{bank}.setHealth.\mathsf{sender}(x_{doctor})(\mathsf{address}(z_{donor}), \mathsf{true})$ is evaluated. Thanks to this rule, programmers may explicitly control msg.sender, thus pretending x_{doctor} to be the actual sender. Also note the call-by-value semantics, where all the actual parameters must be reduced to values before the function call.

e evaluates to u, so it does not actually return anything to the caller. Instead, it modifies the blockchain β by allowing c_{donor} to donate blood. Below we list the evaluation steps for $e' = z_{donor}.donate.\mathsf{sender}(\mathsf{address}(w_{humanDonor}))(500)$, where $EOC\ w_{humanDonor} = \mathsf{new}\ EOC()$. Let $\beta_3 = \sigma_3\beta_2 \cdot [(c_{humanDonor}, a_{humanDonor}) \mapsto (EOC, \epsilon, 0)] \cdot [w_{humanDonor} \mapsto c_{humanDonor}]$.

The transition relation $\Longrightarrow$ behaves as follows:

$$\langle \beta_2, c_{humanDonor}; e' \rangle \Longrightarrow^{\text{COMMIT}} \langle \beta_3, e' \rangle$$

And e' is evaluated in the following way:

$$\langle \beta_3, \sigma_3, z_{donor}.donate.\mathsf{sender}(\mathsf{address}(w_{humanDonor}))(500); \mathsf{u} \rangle \longrightarrow^*$$

$$\langle \beta_3 \cdot \underline{[\_amount \mapsto 500]}, \sigma_3 \cdot \underline{a_{donor}},$$
$$(\mathsf{return\ if}\ c_{bank}.donate(\_amount)\ \mathsf{then}\ c_{donor}.blood = c_{donor}.blood - \_amount; \mathsf{u}$$
$$\mathsf{else\ u}) \rangle \longrightarrow \langle \beta_3 \cdot [\_amount \mapsto 500] \cdot \underline{[amount' \mapsto 500]}, \sigma_3 \cdot a_{donor} \cdot \underline{a_{bank}},$$
$$(\mathsf{return\ if}$$
$$(\mathsf{return\ uint}\ donorBlood = Donor(a_{donor}).getBlood();$$
$$\mathsf{if}\ c_{bank}.healty[a_{donor}] \&\& donorBlood > 3000$$
$$\mathsf{then}\ c_{bank}.blood = c_{bank}.blood + \_amount'; \mathsf{true}$$
$$\mathsf{else\ false})$$
$$\mathsf{then}\ c_{donor}.blood = c_{donor}.blood - \_amount; \mathsf{u\ else\ u}) \rangle \longrightarrow^*$$

$$\langle \beta_3 \cdot [\_amount \mapsto 500] \cdot [amount' \mapsto 500] \cdot \underline{[donorBlood \mapsto 5000]}, \sigma_3 \cdot a_{donor} \cdot a_{bank},$$
$$(\mathsf{return\ if}$$
$$(\mathsf{return\ if\ true} \&\& 5000 > 3000\ \mathsf{then}\ c_{bank}.blood = c_{bank}.blood + \_amount'; \mathsf{true}$$
$$\mathsf{else\ false})$$
$$\mathsf{then}\ c_{donor}.blood = c_{donor}.blood - \_amount; \mathsf{u\ else\ u}) \rangle \longrightarrow^*$$

$$\langle \beta_3 \cdot [\_amount \mapsto 500], \sigma_3 \cdot a_{donor} \cdot a_{bank},$$
$$(\mathsf{return\ if}$$
$$(\mathsf{return}\ c_{bank}.blood = 0 + 500; \mathsf{true})$$
$$\mathsf{then}\ c_{donor}.blood = c_{donor}.blood - \_amount; \mathsf{u\ else\ u}) \rangle \longrightarrow^*$$

$$\langle \emptyset \cdot [(c_{doctor}, a_{doctor}) \mapsto (EOC, \epsilon, 0)] \cdot [x_{doctor} \mapsto c_{doctor}]$$
$$\cdot[(c_{bank}, a_{bank}) \mapsto (BloodBank, healty = \{(a_{donor}, \mathsf{true})\};$$
$$doctor = a_{doctor}; \underline{blood = 500}, 0)] \cdot [y_{bank} \mapsto c_{bank}]$$
$$\cdot[(c_{donor}, a_{donor}) \mapsto (Donor, \underline{blood : 4500}; bank : a_{bank}, 0)] \cdot [z_{donor} \mapsto c_{donor}]$$
$$\cdot[\_donor \mapsto a_{donor}] \cdot [\_isHealty \mapsto \mathsf{true}] \cdot [(c_{humanDonor}, a_{humanDonor})$$
$$\mapsto (EOC, \epsilon, 0)] \cdot [w_{humanDonor} \mapsto c_{humanDonor}] \cdot [\_amount \mapsto 500], \sigma_3, \mathsf{u}\rangle$$

Note that we added a new EOC contract, pointed to by $w_{humanDonor}$, to start the transaction. $w_{humanDonor}$ is intended to represent the person behind the contract z_{donor}, even though z_{donor} does not contain any reference to them. Actually, anybody could call z_{donor}.donate() on $w_{humanDonor}$'s behalf. Here, we were not interested in showing how to implement access control mechanisms, shown in set_Healty(), but in giving an example of how the interaction between contracts behaves.

Example 5.2. Here we make use of the code listed in Example 4.1, and show the evaluation of the following expression:

$$\begin{aligned} e := \quad & EOC\ x_{eoa} = \mathsf{value}(1000).\mathsf{new}\ EOC(); \\ & Bank\ y_{bank} = \mathsf{new}\ Bank(0_{\{\}}); \\ & y_{bank}.deposit.\mathsf{value}(500).\mathsf{sender}(\mathsf{address}(x_{eoa}))(); \\ & y_{bank}.withdraw.\mathsf{sender}(\mathsf{address}(x_{eoa}))(100) \end{aligned}$$

Basically, x_{eoa} first deposits 500 and then withdraws 100 Wei.

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

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

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

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

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

Let $\beta = \sigma = \emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, \underline{500})] \cdot [x_{eoa} \mapsto c_{eoa}] \cdot [(c_{bank}, a_{bank}) \mapsto$
$\underline{(Bank, \{(a_{eoa}, 500)\}, 500)}] \cdot [y_{bank} \mapsto c_{bank}]$.

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

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

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

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

Example 5.3. This example uses the code listed in Example 4.3 and shows the evaluation of the following expression:

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

In this latter evaluation we shall skip the vast majority of the steps to show only the ones more concerned with function pointers.

$$\langle \emptyset, \emptyset, e \rangle \longrightarrow^*$$

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

Let $\beta = \sigma = \emptyset \cdot [(c_{eoa}, a_{eoa}) \mapsto (EOC, \epsilon, 0)] \cdot [x_{eoa} \mapsto c_{eoa}] \cdot [(c_{app}, a_{app}) \mapsto (Applier, state = 10, 0)] \cdot [y_{app} \mapsto c_{app}] \cdot [(c_{test}, a_{test}) \mapsto (Test, app = c_{app}, 0)] \cdot [x_{test} \mapsto c_{test}]$. The evaluation proceeds as follows:

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

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

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

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

This evaluation shows that function pointers behave like normal functions.

Chapter 6

Type system

In this chapter we describe the type system of FS, first giving the form of the judgments and then the rules. FS is strongly and statically typed, meaning that all the types of an expression are known before evaluating it, and that programmers cannot work around the restrictions imposed by our type system. This is slightly different in Solidity, where some expressions, for instance inline assembly or call, may break type safety. FS does not model these aspects.

6.1 The goals of the type system of FS

Usually, type systems are used to give programmers some guarantees. In particular, they help in rejecting all the ill-formed expressions, that is the ones having a sort of "undefined" meaning, such as $\mathsf{balance}(5)$ or $\mathsf{true.transfer}(\mathsf{false})$. Taking the former as an example, the implicit function $\mathrm{balance}$ is defined only over addresses and makes no sense on natural numbers. One of the most important guarantees is type safety. Informally, in a type-safe language, if an expression is closed (i.e. it does not contain any free variables) and well-typed (i.e. not ill-formed) then it does not go stuck, which means that the computation is not blocked (but it may still not terminate due to, for example, an infinite loop or an ill-founded recursion). On the other hand, a stuck expression cannot be further evaluated, and the computation is blocked. Normally, this is a serious issue, because the program does not terminate correctly.

In Solidity, at run-time, the following errors may appear:

1. accessing a state variable or function not defined by the contract;
2. wrong parameters passing, either in number or type. This category includes not only function calls, but also if expressions, mapping reads and writes, assignment, and so forth;
3. transferring an amount of Wei to a contract without a (payable) fallback function;
4. sending an amount of Wei to a non-payable constructor during a contract instantiation;
5. wrong cast $C(a)$. This happens when a refers to an instance of a contract C', with $C \neq C'$;
6. transferring an amount of Wei from a contract with an insufficient balance;

7. the transaction ran out of gas.

Solidity's compiler rules out Errors 1, 2, and 4. Solidity actually allows a workaround that makes them possible. In fact, programmers can invoke functions either in the usual way (i.e. on a variable mapping to a contract reference) or by using `call`, `delegatecall`, or `callcode`. We explained in detail their behavior, and the harmful consequences, in Example 2.2: in fact, the documentation [48] explicitly states to use them with care and as a last resort, since they break the type safety of Solidity. Unfortunately, except for this textual advice, the compiler does not warn programmers. This may, of course, lead to unexpected behaviors, and FS does not contain such primitives. Hence, we shall consider only Solidity code not using any of them, and thus ruling out Errors 1, 2, and 4.

Errors 1 and 2, in particular, correspond to the "message not understood" error in the object-oriented programming paradigm. The proof of the typing properties of FS (see Appendix E) is the first formal proof that well-typed Solidity contracts do not contain such errors.

Errors 3 and 5 have to do with clumsy uses of addresses. Starting with Error 3, an amount of Wei can be transfered from one account to another by using either `transfer` or `send`. The latter is actually the low-level counterpart of the former (returning a `bool` to indicate whether or not the transfer was successful), and its use is discouraged. They are both defined on values of type address. This choice enables msg.sender to have type address and simplifies the writing of contracts. Furthermore, EOAs in Solidity do not correspond to contracts, thus trying to give a contract type to msg.sender might pose some problems. On the other hand, this could introduce errors of the third category. As we said, a contract must define a payable fallback function in order to receive any amount of Wei, but Solidity does not check which code is associated with an address when `transfer` (or `send`) is invoked[1]. Hence, it may happen that the recipient does not define such a function, thus causing Error 3. Error 5 originates from a similar issue. When we compile a contract we cannot know if a given address will correspond to a given contract definition C. If it does, the cast will be successful at run-time, but if it does not, the behavior entirely depends on the actual contract pointed to by the address, as we saw in Section 2.4.2.

Lastly, Error 6 appears when a contract is transferring money to another, but its balance is insufficient. Since contracts cannot have negative balances, the execution cannot further proceed.

Solidity solves these issues by introducing the concept of errors and reverts. Whenever something goes wrong and a transaction cannot terminate, Solidity can cause a revert, thus aborting the transaction itself. This happens, for instance, when the amount of gas is not enough to make the code run to completion or when an amount of Wei is sent to a contract without a (payable) fallback function. Programmers also can abort the current transaction. Usually, this is done either implicitly, with `require` or `assert`, which validate preconditions, or explicitly, with `revert`.

We shall differentiate the two categories as follows:

- "internal" reverts issued by Solidity are referenced to as **errors**;
- reverts issued by programmers are referenced to as **exceptions**. They appear explicitly into the code and (should) entirely depend on the business logic. For instance, an auction contract might `require` that a new bid is greater than the

[1] Actually, `transfer` and `send` can be invoked also on contract references, but the compiler gives a warning saying it is deprecated.

current one. If such entry guard is false, the amount of Wei sent along with the function invocation is sent back to the caller. The miner, on the other hand, still receives a payment corresponding to the effort it took to validate the condition. These reverts cannot be avoided using only a type system, because they depend on values known only at run-time.

With this solution, transactions always terminate, either with a value or with an abort. Remember that, in Solidity, a transaction cannot loop indefinitely thanks to the system of gas. Nonetheless, an abort still leaves tracks of the transaction: any changes to the blockchain (both to state variables and balances) are reverted, except for the price paid to the miner. In fact, miners keep the amount of Wei corresponding to the amount of gas needed to mine the transaction, even if the latter actually failed.

FS presents the same run-time errors as Solidity, except for the out-of-gas one. We assume that every program runs with an amount of gas sufficient to make it run to completion. This assumption is not too strict, since providing enough gas is just a matter of paying more Wei to execute a transaction. These errors are dealt with in the same way as Solidity does: Errors 1, 2, and 4 are ruled out by the type system, whereas Errors 3, 5, and 6 raise a revert and abort the transaction. Note that every function in FS is implicitly `payable`, and so we do not check this aspect when looking for a fallback function.

6.2 Judgments

Definition 9 defines the context Γ, a list of typed variables, addresses, and contract references.

Definition 9 (Context)**.**

$$(Type\ environment) \quad \Gamma \quad ::= \quad \emptyset \mid \Gamma,\ x : T \mid \Gamma,\ a : \mathsf{address} \mid \Gamma,\ c : C$$

Definition 10 then extends Definition 2 and Definition 3 to include Γ.

Definition 10 (Domains with type environment)**.** Let Γ be well formed (see Section 6.3). Then its domain is defined as follows:

$$\begin{array}{lcl} \mathrm{dom}(\emptyset) & = & \emptyset \\ \mathrm{dom}(\Gamma,\ x : T) & = & \{x\} \cup \mathrm{dom}(\Gamma) \\ \mathrm{dom}(\Gamma,\ a : \mathsf{address}) & = & \{a\} \cup \mathrm{dom}(\Gamma) \\ \mathrm{dom}(\Gamma,\ c : C) & = & \{c\} \cup \mathrm{dom}(\Gamma) \end{array}$$

Hence, $\mathrm{dom}(\Gamma)$ is defined as the set of variables, addresses, and contract references contained in Γ.

The main goal of a type system is to give types to expressions. We use the following types of judgments:

- **Type environment** ($\Gamma \vdash \langle\rangle$), to indicate that Γ is well formed;
- **Contract and function** ($T_2\ f\ (\tilde{T_1\ x})\ \{\mathsf{return}\ e\}\ OK\ in\ C$ and $\mathsf{contract}\ C\ \{\tilde{T\ s};\ K\ \tilde{F}\}\ OK$), to indicate that both contract and function definitions are well formed;
- **Blockchain** ($\Gamma \vdash \beta$), to indicate that β is compliant with its type definition;

- **Transaction** ($\Gamma \vdash \sigma$), to indicate that σ is well-formed;
- **Configuration and program** ($\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ and $\Gamma \vdash (CT, \beta, e)T$), to ensure that programs and configurations are well formed;
- **Expressions** ($\Gamma \vdash e : T$), to give a type to each meaningful expression of the FS language.

In the rules that follow, T, T_1, and T_2 range over expression types, as defined in Figure 4.1, whereas C represents contract names.

6.3 Type rules

In this section we list all the rules defining the type system of FS. Most of them are standard and have the expected meaning. Others reflect some peculiarities of Solidity. Note that FS does not support subtyping. For a comprehensive view of all these rules, see Appendix B.

Remark (Tuple notation in FS type judgments)**.** As said in Chapter 4, the symbol $\tilde{}$ represents a tuple: in the rules that follow we shall make further use of this symbol in various ways. $\Gamma \vdash \tilde{v} : \tilde{T}$ represents $n \in \mathbb{N}$ distinct judgments with the form $\Gamma \vdash v_i : T_i$, $1 \leq i \leq n$, that is every value v_i is required to have a type T_i possibly different than all the others T_j in $\tilde{T}$. On the other hand, $\Gamma \vdash \tilde{v} : T$ also represents n distinct judgments, but the type T is fixed and does not vary, that is: $\Gamma \vdash v_i : T$, $1 \leq i \leq n$. Lastly, we use $\{\tilde{(k, v)}\}$ to indicate a tuple of pairs (k, v): $(k_1, v_1); \ldots; (k_n; v_n)$.

Well-formedness of Γ The rules to check the well-formedness of the type environment Γ, and making sure that every element in $\mathrm{dom}(\Gamma)$ is unique, are as follows:

$$\frac{}{\emptyset \vdash \langle\rangle}\ (\textsc{EmptyEnvironment}) \qquad \frac{\Gamma \vdash \langle\rangle \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : T \vdash \langle\rangle}\ (\textsc{VariableEnvironment})$$

$$\frac{\Gamma \vdash \langle\rangle \qquad a \notin \mathrm{dom}(\Gamma)}{\Gamma, a : \mathsf{address} \vdash \langle\rangle}\ (\textsc{AddressEnvironment}) \qquad \frac{\Gamma \vdash \langle\rangle \qquad c \notin \mathrm{dom}(\Gamma)}{\Gamma, c : C \vdash \langle\rangle}\ (\textsc{ContractEnvironment})$$

Well-formedness for contracts and functions The following rules give a type to contract and function definitions. Note that $T\ f\ \tilde{(T\ x)}\ \{\mathsf{return}\ e\}\ OK\ in\ C$ captures also fb.

$$\frac{\text{this} : C, \text{msg.sender} : \text{address}, \text{msg.value} : \text{uint}, \tilde{x} : \tilde{T_1} \vdash e : T_2}{T_2\ f\ (\tilde{T_1\ x})\ \{\text{return}\ e\}\ OK\ in\ C}\ \text{(F OK IN C)}$$

$$\frac{K = C\ (\tilde{T\ x})\ \{\text{this}.\tilde{s}\ =\ \tilde{x}\} \qquad \tilde{F}\ \text{OK in}\ C}{\text{contract}\ C\ \{\tilde{T\ s};\ K\ \tilde{F}\}\ OK}\ \text{(C OK)}$$

Well-formedness for blockchains The rules used to check the well-formedness of β are the following:

$$\frac{}{\Gamma \vdash \emptyset}\ \text{(EMPTYBLOCKCHAIN)} \qquad \frac{\Gamma \vdash \beta \qquad x \notin \text{dom}(\beta) \qquad \Gamma \vdash x : T \qquad \Gamma \vdash v : T}{\Gamma \vdash \beta \cdot [x \mapsto v]}\ \text{(VARIABLE)}$$

$$\frac{\Gamma \vdash \beta \qquad (c, a) \notin \text{dom}(\beta) \qquad \Gamma \vdash c : C \qquad \Gamma \vdash a : \text{address} \qquad \text{sv}(C) = \tilde{T\ s} \qquad \Gamma \vdash \tilde{v} : \tilde{T} \qquad \Gamma \vdash n : \text{uint} \qquad |\tilde{s}| = |\tilde{v}|}{\Gamma \vdash \beta \cdot [(c, a) \mapsto (C, \tilde{s : v}, n)]}\ \text{(CONTRACT)}$$

VARIABLE makes sure that every "new" variable is unique in β (i.e. $\nexists x \in \text{dom}(\beta)$) and that the type of x is the same of the one of v, the value being associated to x. CONTRACT is similar: we check the type of c and a (respectively, a contract reference of C and an address), that the balance n is of type uint and that the state variables $\tilde{s}$ match in number and type with the values $\tilde{v}$ associated with them.

Well-formedness for call stacks The rule used to give a type to a call stack is defined as follows:

$$\frac{\Gamma \vdash \sigma \qquad \Gamma \vdash a : \text{address}}{\Gamma \vdash \sigma \cdot a}\ \text{(CALLSTACK)}$$

σ is composed of two parts: a blockchain β and a sequence of addresses a pushed upon the latter. CALLSTACK, which checks the type of each of these addresses, eventually reaches its base case, $\Gamma \vdash \beta$. At this point, the rules to give a type to β apply.

Well-formedness for configurations and programs The rules used to check configurations and programs well-formedness are defined as follows:

$$\text{(CONFIGURATION)}\ \frac{\Gamma \vdash \beta \qquad \Gamma \vdash \sigma \qquad \Gamma \vdash e : T}{\Gamma \vdash \langle \beta, \sigma, e \rangle : T} \qquad \text{(PROGRAM)}\ \frac{C\ OK\ \forall\, C \in CT \qquad \Gamma \vdash \langle \beta, \beta, e \rangle : T}{\Gamma \vdash (CT, \beta, e) : T}$$

CONFIGURATION is used to give a type to $\mathcal{C}$ in FS. Its formalization aims to be general: note, indeed, the first premise $\Gamma \vdash \beta$. It may seem redundant, since it is also implied by $\Gamma \vdash \sigma$ (remember the base case of CALLSTACK), but this is not always true. In fact, the first element of $\mathcal{C}$ is equal to the β in σ only at the beginning of the execution, but after that it may change. A rule without this first premise would fail in giving a type to the evolving β.

PROGRAM gives a type to an FS program, starting from the configuration $\langle \beta, \beta, e \rangle$.

Types of expressions We now state the rules to give a type to FS expressions.

Axioms The axioms have all a standard meaning. Note how we require $\Gamma \vdash \langle\rangle$ in all of them, to make sure that the context we operate on is well-formed.

$$\text{(REF)}\ \frac{\Gamma, c : C, \Gamma' \vdash \langle\rangle}{\Gamma, c : C, \Gamma' \vdash c : C} \qquad \text{(VAR)}\ \frac{\Gamma, x : T, \Gamma' \vdash \langle\rangle}{\Gamma, x : T, \Gamma' \vdash x : T} \qquad \text{(TRUE)}\ \frac{\Gamma \vdash \langle\rangle}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}$$

$$\text{(FALSE)}\ \frac{\Gamma \vdash \langle\rangle}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \qquad \text{(ADDRESS)}\ \frac{\Gamma, a : \mathsf{address}, \Gamma' \vdash \langle\rangle}{\Gamma, a : \mathsf{address}, \Gamma' \vdash a : \mathsf{address}} \qquad \text{UNIT}\ \frac{\Gamma \vdash \langle\rangle}{\Gamma \vdash \mathsf{u} : \mathsf{unit}}$$

$$\text{(REVERT)}\ \frac{\Gamma \vdash \langle\rangle}{\Gamma \vdash \mathsf{revert} : T} \qquad \text{(NAT)}\ \frac{n \in \mathbb{N}^+ \qquad \Gamma \vdash \langle\rangle}{\Gamma \vdash n : \mathsf{uint}}$$

$\Gamma \vdash \mathsf{revert} : T$ is well-typed regardless of T. We do not give a unique type to this expression, since it may be used in any context whatsoever. For instance, $\Gamma \vdash$ if true then 10 else revert : uint and $\Gamma \vdash$ if false then u else revert : unit are both valid expression in FS, but the former has type uint and the latter has type unit.

Standard rules Rules with a standard meaning are listed below. They comprise the if expression, variable assignment and declaration as well as sequential composition. BAL and ADDR are about the expressions $\mathsf{balance}(e)$ and $\mathsf{address}(e)$, respectively. As said in Section 2.4.2, the former operates on addresses, whereas the latter is used to explicitly get the address identifying a given contract reference.

$$\text{(BAL)}\ \frac{\Gamma \vdash e : \mathsf{address}}{\Gamma \vdash \mathsf{balance}(e) : \mathsf{uint}} \qquad \text{(ADDR)}\ \frac{\Gamma \vdash e : C}{\Gamma \vdash \mathsf{address}(e) : \mathsf{address}} \qquad \text{(RETURN)}\ \frac{\Gamma \vdash e : T}{\Gamma \vdash \mathsf{return}\, e : T}$$

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

$$\text{(IF)}\ \frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T} \qquad \text{(ASS)}\ \frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T}{\Gamma \vdash x = e : T}$$

Mappings The following rules give a type to mapping values, as well as to read and write accesses on them. As we said at the beginning of this section, $\Gamma \vdash \tilde{v} : T$ represents $n \in \mathbb{N}$ distinct judgments where the type T is fixed and does not vary, that is: $\Gamma \vdash v_i : T, 1 \leq i \leq n$.

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

$$\text{(MAPPASS)}\ \frac{\Gamma \vdash e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2) \qquad \Gamma \vdash e_2 : T_1 \qquad \Gamma \vdash e_3 : T_2}{\Gamma \vdash e_1[e_2 \to e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)}$$

$$\text{(MAPPSEL)}\ \frac{\Gamma \vdash e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2) \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1[e_2] : T_2}$$

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

$$\text{(STATESEL)}\ \frac{\Gamma \vdash e : C \qquad \mathrm{sv}(C) = \tilde{T\, s} \qquad s_i \in \tilde{s}}{\Gamma \vdash e.s_i : T_i} \qquad \text{(STATEASS)}\ \frac{\Gamma \vdash e_1.s : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1.s = e_2 : T}$$

$$\text{(NEW)}\ \frac{\mathrm{sv}(C) = \tilde{T\, s} \qquad \Gamma \vdash \tilde{e} : \tilde{T} \qquad |\tilde{e}| = |\tilde{s}| \qquad \Gamma \vdash e' : \mathsf{uint}}{\Gamma \vdash \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : C}$$

Functions The following rules regard functions.

$$\frac{\Gamma \vdash c : C \qquad \text{ftype}(C, f) = \tilde{T_1} \to T_2}{\Gamma \vdash c.f : \tilde{T_1} \to T_2} \text{(FUN)}$$

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

$$\frac{\Gamma \vdash e_3 : \mathsf{address} \qquad \Gamma \vdash e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2}{\Gamma \vdash e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T_2} \text{(CALLTOPLEVEL)}$$

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

Rule FUN imposes that what we called a "function pointer" ($c.f$) must be defined in the code of c. It retrieves the type of f via ftype, defined in Figure 5.4. If $f \notin \tilde{F}$, $\text{ftype}(C, f)$ is not defined, and FUN does not apply anymore. Thus, to use a function pointer as a value, it must appear in the code of c. Note how CALLVALUE's first premise, $\Gamma \vdash e_1 : \tilde{T_1} \to T_2$ reduces to FUN.

CALL and CALLTOPLEVEL have the expected meaning; the latter also checks that the expression specified as sender is of type address.

Casts and money transfers The following rules are about casts and money transfers.

$$\frac{\Gamma \vdash e : \mathsf{address} \qquad \textit{warning}}{\Gamma \vdash C(e) : C} \text{(CONTRRETR)} \qquad \frac{\Gamma \vdash e_1 : \mathsf{address} \qquad \Gamma \vdash e_2 : \mathsf{uint}}{\Gamma \vdash e_1.\mathsf{transfer}(e_2) : \mathsf{unit}} \text{(TRANSFER)}$$

Note the limitation in TRANSFER. The transfer operation is defined on addresses (its definition on contract references is deprecated in Solidity), and at compile-time, when type rules are enforced (remember FS is statically typed), we have no information about the contract residing at a given address. Hence, we cannot check if the code of c contains a fallback function. Lastly, the type system does not ensure that a top-level function call (the one specifying the sender) actually appears only at the top level. Furthermore, Solidity does not contain such distinction, since top-level calls actually come from a different programming language (typically from JavaScript through the web3.js library). Hence, we assume a correct use of top and inner-level calls: a simple static inspection of the program (prior to the type check) can detect such violations and stop the compilation process.

CONTRRETR raise a warning when a cast is compiled. As we explained in Section 2.4.2, Ethereum adopts a very liberal policy when it deals with casts from address to contract types. Furthermore, Solidity's compiler does not give any warnings to signal that something strange may happen at run-time. We chose, instead, to warn programmers at compile-time and to adopt a more conservative behavior when evaluating expressions like $C(a)$. As shown in Section 5.5 (Rule CONTRRETR-R), we raise a revert when we detect that a refers to an instance of $C' \neq C$. With this type system this is all we can do. We shall see soon, in Chapter 7, a possible way of making casts really type-safe.

These limitations inspired and convinced us to investigate a better version of the type system, which is discussed in detail in Chapter 7.

Example 6.1 shows the type derivation for a simple FS expression.

Example 6.1. In this example we shall prove the well-typing of a program using the code of Example 4.1. Remember that a program is a triple (CT, β, e), where CT contains the code of EOC (Definition 1) and of $Bank$ (Listing 4.2) and e is as follows:

$$\begin{aligned} e := \quad & EOC\ x = \mathsf{new}\ EOC.\mathsf{value}(500)(); \\ & Bank\ y = \mathsf{new}\ Bank(0_{\{\}}); \\ & y.deposit.\mathsf{value}(100).\mathsf{sender}(\mathsf{address}(x))() \end{aligned}$$

We suppose $\beta = \emptyset$ and depict the type derivation for this program. We make use of placeholders (such as (1)) to indicate a derivation which is shown separately.

$$\dfrac{\dfrac{(1)}{C\ OK\ \forall\ C \in CT} \qquad \dfrac{(2)}{\emptyset \vdash \langle \emptyset, \emptyset, EOC\ x = \mathsf{new}\ EOC.\mathsf{value}(500)(); e' \rangle : \mathsf{unit}}}{\emptyset \vdash (CT, \emptyset, e) : \mathsf{unit}}$$

Where (1) is as follows. We show only the derivation for $Bank$ and omit the one for EOC, which is trivial.

$$\dfrac{\dfrac{\checkmark}{K = Bank(\mathsf{mapping}(\mathsf{address} \Rightarrow \mathsf{unit})\ balances)\ \{\dots\}} \qquad \dfrac{(6)}{\tilde{F}\ \text{OK in}\ C}}{Bank\ OK}$$

Regarding the premise $\tilde{F}$ OK in C, we shall show only the derivation of the first one, $deposit$.
Let $e_d = (\mathsf{this}.balances = \mathsf{this}.balances[\mathsf{msg.sender} \rightarrow \mathsf{this}.balances[\mathsf{msg.sender}] + \mathsf{msg.value}])$, $eass_d = \mathsf{this}.balances[\mathsf{msg.sender} \rightarrow \mathsf{this}.balances[\mathsf{msg.sender}] + \mathsf{msg.value}]$, $T_{map} = \mathsf{mapping}(\mathsf{address} \Rightarrow \mathsf{unit})$, and
$\Gamma = \mathsf{this} : Bank, \mathsf{msg.sender} : \mathsf{address}, \mathsf{msg.value} : \mathsf{uint}$; (2) is as follows:

$$\dfrac{\dfrac{\dfrac{(3)}{\Gamma \vdash \mathsf{this}.balances : T_{map}} \qquad \dfrac{(4)}{\Gamma \vdash eass_d : T_{map}}}{\Gamma \vdash e_d : T_{map}} \qquad \dfrac{\checkmark}{\Gamma \vdash \mathsf{u} : \mathsf{unit}}}{\dfrac{\Gamma \vdash e_d; \mathsf{u} : \mathsf{unit}}{\tilde{F}\ \text{OK in}\ C}}$$

Where (3) is as follows:

$$\frac{\dfrac{\checkmark}{\Gamma \vdash \mathsf{this} : Bank} \qquad balances \in \mathrm{sv}(Bank)}{\Gamma \vdash \mathsf{this}.balances : T_{map}}$$

And (4) is as follows:

$$\frac{\dfrac{(3)}{\Gamma \vdash \mathsf{this}.balances : T_{map}} \qquad \dfrac{\checkmark}{\Gamma \vdash \mathsf{msg.sender} : \mathsf{address}} \qquad (5)}{\Gamma \vdash eass_d : T_{map}}$$

Supposing the existence of a rule giving a type to an addition of integers, (5) is the following derivation tree:

$$\frac{\dfrac{\dfrac{(3)}{\Gamma \vdash \mathsf{this}.balances : T_{map}} \qquad \dfrac{\checkmark}{\Gamma \vdash \mathsf{msg.sender} : \mathsf{address}}}{\Gamma \vdash \mathsf{this}.balances[\mathsf{msg.sender}] : \mathsf{uint}} \qquad \dfrac{\checkmark}{\Gamma \vdash \mathsf{msg.value} : \mathsf{uint}}}{\Gamma \vdash \mathsf{this}.balances[\mathsf{msg.sender}] + \mathsf{msg.value} : \mathsf{uint}}$$

We can now look at the derivation of the expression e $((6))$.

$$\frac{\dfrac{\dfrac{\checkmark}{\emptyset \vdash 500 : \mathsf{uint}}}{\Gamma \vdash \mathsf{new}\ EOC.\mathsf{value}(500)() : EOC} \qquad \dfrac{(7)}{\emptyset, x : EOC \vdash e_1 : \mathsf{unit}}}{\emptyset \vdash EOC\ x = \mathsf{new}\ EOC.\mathsf{value}(500)() : \mathsf{unit}; e_1}$$

(7) is the following derivation tree:

$$\frac{\dfrac{(8)}{\emptyset, x : EOC \vdash \mathsf{new}\ Bank(0_{\{\}}) : Bank} \qquad \dfrac{(9)}{\emptyset, x : EOC, y : Bank \vdash e_2 : \mathsf{unit}}}{\emptyset, x : EOC \vdash Bank\ y = \mathsf{new}\ Bank(); e_2 : \mathsf{unit}}$$

Where (8) is as follows:

$$\frac{\dfrac{\checkmark}{\emptyset, x : EOC \vdash 0_{\{\}} : \mathsf{mapping}(\mathsf{address} \Rightarrow \mathsf{uint})}}{\emptyset, x : EOC \vdash \mathsf{new}\ Bank(0_{\{\}}) : Bank}$$

For the sake of clarity, in (7) and (8) we omitted the trivial premises regarding the instantiation of EOC and $Bank$.

Let $\Gamma = \emptyset, x : EOC, y : Bank$; (9) is as follows:

$$\frac{\dfrac{\dfrac{\checkmark}{\Gamma \vdash x : EOC}}{\Gamma \vdash \mathsf{address}(x) : \mathsf{address}} \qquad \dfrac{\dfrac{\checkmark}{\Gamma \vdash y : Bank} \qquad \dfrac{\checkmark}{\Gamma \vdash 100 : \mathsf{uint}}}{\Gamma \vdash y.deposit.\mathsf{value}(100)() : \mathsf{unit}}}{\Gamma \vdash y.deposit.\mathsf{value}(100).\mathsf{sender}(\mathsf{address}(x))() : \mathsf{unit}}$$

$$\frac{\Gamma \vdash y.deposit.\mathsf{value}(100).\mathsf{sender}(\mathsf{address}(x))() : \mathsf{unit}}{\Gamma \vdash y.deposit.\mathsf{value}(100).\mathsf{sender}(\mathsf{address}(x))() : \mathsf{u}}$$

In (9), while giving a type to $y.deposit.\mathsf{value}(100)()$ we omitted the premises regarding the parameters of $deposit$ because there were no formal parameters.

If Example 6.1 showed how the type system gives a type to a legitimate expression, Example 6.2 does the opposite, showing how, in FS, ill-formed expressions are ruled out.

Example 6.2. This example is based on the very same CT as Example 6.1, and thus we omit the proof of its well-formedness. We shall show how the following e is rules out, since incorrect.

$$\begin{aligned} e := \quad & EOC\ x = \mathsf{new}\ EOC.\mathsf{value}(500)(); \\ & Bank\ y = \mathsf{new}\ Bank(0_{\{\}}); \\ & y.deposit.\mathsf{value}(100).\mathsf{sender}(\mathsf{address}(x))(10) \end{aligned}$$

Note that e is as in Example 6.1, with the only difference that $deposit$ is given an argument, 10. Since this function does not accept any parameters, e should not be accepted by the type system. The type derivation until (9), as well as the context Γ, is as in Example 6.1: we shall only show the difference in the judgment $\Gamma \vdash y.deposit.\mathsf{value}(100)(10) : \mathsf{unit}$.

$$\frac{\dfrac{}{\Gamma \vdash y : Bank}\checkmark \quad \dfrac{}{\Gamma \vdash 100 : \mathsf{uint}}\checkmark \quad \mathrm{ftype}(Bank, deposit) = \{\} \to \mathsf{unit} \quad \dfrac{}{\Gamma \vdash 10 : \emptyset}\ 0 = 1}{\Gamma \vdash y.deposit.\mathsf{value}(100)(10) : \mathsf{unit}}$$

The rule fails when it comes to give a type to the parameter 10. This cannot be done, since the type of $deposit$ in $Bank$ is $\{\} \to \mathsf{unit}$.

6.4 Properties of the type system

We can now state some important properties of our type system. Our main goal is to prove the type safety of FS, of which we give two versions: one applying to FS configurations (Theorem 1) and one applying to programs (Theorem 2). In order to define closed programs we need an additional function retrieving the contract names in β (Definition 11).

Definition 11 (Contract names in β)**.**

$$\begin{aligned} \mathrm{cn}(\emptyset) &= \emptyset \\ \mathrm{cn}(\beta \cdot [(c, a) \mapsto (C, \tilde{sv}, n)]) &= \{C\} \cup \mathrm{cn}(\beta) \\ \mathrm{cn}(\beta \cdot [x \mapsto v]) &= \mathrm{cn}(\beta) \end{aligned}$$

We can now define closed programs (Definition 12) and configurations (Definition 13).

Definition 12 (Closed programs)**.** Let $\mathcal{P} = (CT, \beta, e)$ be an FS program. $\mathcal{P}$ is closed if $\mathrm{fv}(e) \cup \mathrm{fn}(e) \subseteq \mathrm{dom}(\beta)$ and $\mathrm{cn}(\beta) \subseteq \mathrm{dom}(CT)$.

Definition 13 (Closed configurations)**.** Let e be an FS expression and consider the configuration $\langle \beta, \sigma, e \rangle$. The latter is closed if $\mathrm{fv}(e) \cup \mathrm{fn}(e) \subseteq \mathrm{dom}(\beta)$, that is, any free variables or names appear in β. The same applies to $\langle \beta, e \rangle$.

Lemma 1 explicits a property connecting closed programs and configurations.

Lemma 1 (Closedness)**.**
Let $\mathcal{P} = (CT, \beta, e)$ be a closed FS program. Then $\langle \beta, e \rangle$ and $\langle \beta, \beta, e \rangle$ are closed, too.

Proof. The proof immediately follows from the first property of closed programs. In fact, $\mathcal{P}$ implies $\mathrm{fv}(e) \cup \mathrm{fn}(e) \subseteq \mathrm{dom}(\beta)$, which in turn means that $\langle \beta, e \rangle$ and $\langle \beta, \beta, e \rangle$ are closed. □

Definition 12 and Definition 13 slightly abuse the notation of $\subseteq$, because $\mathrm{fn}(e)$ does not return pairs (c, a), but only one component of such pair (i.e. c or a, depending on e). Hence $\mathrm{fn}(e) = \{c\} \subseteq \mathrm{dom}(\beta)$ actually means $\exists a$ such that $(c, a) \in \mathrm{dom}(\beta)$, and, respectively, $\mathrm{fn}(e) = \{a\} \subseteq \mathrm{dom}(\beta)$ actually means $\exists c$ such that $(c, a) \in \mathrm{dom}(\beta)$.

Theorem 1 (Type safety for configurations)**.**
If $\Gamma \vdash \langle \beta, \beta, e \rangle : T$, $\langle \beta, \beta, e \rangle$ is closed, and $\exists (\beta', \sigma', e')$ such that $\langle \beta, \beta, e \rangle \longrightarrow^* \langle \beta', \sigma', e' \rangle$, with $\langle \beta', \sigma', e' \rangle \not\longrightarrow$, then either $e' = v$, where v is a value, or $e' =$ revert.

Proof. We prove this theorem in Appendix E.6. □

Theorem 2 (Type safety for programs)**.**
Let $\mathcal{P} = (CT, \beta, e_1; \ldots; e_n)$ be an FS program. If $\Gamma \vdash (CT, \beta, e_1; \ldots; e_n) : T$, $\mathcal{P}$ is closed, and $\exists (\beta', e')$ such that $\langle \beta, e_1; \ldots; e_n \rangle \Longrightarrow^* \langle \beta', e' \rangle$, with $\langle \beta', e' \rangle \not\Longrightarrow$, then either $e' = v$ or $e' =$ revert.

Proof. We prove this theorem in Appendix E.7. □

In order to prove these two theorems, we need many other results: Inversion, Permutation, Weakening, Substitution, and Canonical Forms lemmas are formalized by Lemma 2 to 6.

Lemma 2 (Inversion)**.**

1. *If $\Gamma \vdash$ true $: T$ can be derived, then $T =$ bool and Γ is well-formed.*
2. *If $\Gamma \vdash$ false $: T$ can be derived, then $T =$ bool and Γ is well-formed.*
3. *If $\Gamma \vdash n : T$ can be derived, then $T =$ uint and Γ is well-formed.*
4. *If $\Gamma \vdash$ u $: T$ can be derived, then $T =$ unit and Γ is well-formed.*
5. *If $\Gamma \vdash$ revert $: T$ can be derived, then Γ is well-formed.*
6. *If $\Gamma \vdash a : T$ can be derived, then $T =$ address and $\Gamma = \Gamma', a :$ address$, \Gamma''$ is well formed.*
7. *If $\Gamma \vdash c.f : T$ can be derived, then $\exists T_1, T_2$ such that $T = \tilde{T_1} \to T_2$, $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$, and $\Gamma \vdash c : C$ is derivable.*
8. *If $\Gamma \vdash M : T$ can be derived, then $\exists T_1, T_2$ such that $T =$ mapping$(T_1 \Rightarrow T_2)$.*
9. *If $\Gamma \vdash c : T$ can be derived, then $T = C$ and $\Gamma = \Gamma', c : C, \Gamma''$ is well formed.*
10. *If $\Gamma \vdash x : T$ can be derived, then $\Gamma = \Gamma', x : T, \Gamma''$ is well formed.*
11. *If $\Gamma \vdash$ balance$(e) : T$ can be derived, then $T =$ uint and $\Gamma \vdash e :$ address is derivable.*

12. *If* $\Gamma \vdash \mathsf{address}(e) : T$ *can be derived, then* $T = \mathsf{address}$ *and* $\Gamma \vdash e : C$ *is derivable.*

13. *If* $\Gamma \vdash \mathsf{return}\, e : T$ *can be derived, then* $\Gamma \vdash e : T$ *is derivable.*

14. *If* $\Gamma \vdash e.s : T$ *can be derived, then* $\Gamma \vdash e : C$ *and* $\exists T_i$ *such that* $\mathrm{sv}(C) = \tilde{T}s$ *and* $T = T_i$.

15. *If* $\Gamma \vdash e_1.\mathsf{transfer}(e_2) : T$ *can be derived, then* $T = \mathsf{unit}$, $\Gamma \vdash e_1 : \mathsf{address}$, *and* $\Gamma \vdash e_2 : \mathsf{uint}$.

16. *If* $\Gamma \vdash \mathsf{new}\, C.\mathsf{value}(e')(\tilde{e}) : T$ *can be derived, then* $T = C$, $\Gamma \vdash e' : \mathsf{uint}$, *and* $\Gamma \vdash \tilde{e} : \tilde{T}$, *where* $\mathrm{sv}(C) = \tilde{T}s$.

17. *If* $\Gamma \vdash C(e) : T$ *can be derived, then* $T = C$ *and* $\Gamma \vdash e : \mathsf{address}$.

18. *If* $\Gamma \vdash e_1; e_2 : T_2$ *can be derived, then* $\Gamma \vdash e_2 : T_2$ *and* $\exists T_1$ *such that* $\Gamma \vdash e_1 : T_1$.

19. *If* $\Gamma \vdash T_1\ x = e; e' : T_2$ *can be derived, then* $\Gamma, x : T_1 \vdash e' : T_2$ *and* $\Gamma \vdash e : T_1$.

20. *If* $\Gamma \vdash x = e : T$ *can be derived, then* $\Gamma \vdash x : T$ *and* $\Gamma \vdash e : T$.

21. *If* $\Gamma \vdash e_1.s = e_2 : T$ *can be derived, then* $\Gamma \vdash e_1 : C$, $\Gamma \vdash e_1.s : T$, *and* $\Gamma \vdash e_2 : T$.

22. *If* $\Gamma \vdash e_1[e_2] : T$ *can be derived, then* $\exists T_1, T_2$ *such that* $T = T_2$, $\Gamma \vdash e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, *and* $\Gamma \vdash e_2 : T_2$.

23. *If* $\Gamma \vdash e_1[e_2 \to e_3] : T$ *can be derived, then* $\exists T_1, T_2$ *such that* $\Gamma \vdash e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash e_2 : T_1$, $\Gamma \vdash e_3 : T_2$, *and* $T = \mathsf{mapping}(T_1 \Rightarrow T_2)$.

24. *If* $\Gamma \vdash e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T$ *can be derived, then* $\exists \tilde{T_1}, T_2$ *such that* $T = \tilde{T_1} \to T_2$, $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$, $\Gamma \vdash e_1 : C$, $\Gamma \vdash e_2 : \mathsf{uint}$, *and* $\Gamma \vdash \tilde{e} : \tilde{T}$.

25. *If* $\Gamma \vdash e_1.\mathsf{value}(e_2)(\tilde{e}) : T$ *can be derived, then* $\exists T_1, T_2$ *such that* $T = \tilde{T_1} \to T_2$, $\Gamma \vdash e_1 : \tilde{T_1} \to T_2$, $\Gamma \vdash e_2 : \mathsf{uint}$, *and* $\Gamma \vdash \tilde{e} : \tilde{T}$.

26. *If* $\Gamma \vdash e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T$ *can be derived, then* $\exists T_1, T_2$ *such that* $T = \tilde{T_1} \to T_2$, $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$, $\Gamma \vdash e_1 : C$, $\Gamma \vdash e_2 : \mathsf{uint}$, $\Gamma \vdash e_3 : \mathsf{address}$, *and* $\Gamma \vdash \tilde{e} : \tilde{T}$.

27. *If* $\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ *can be derived, then* $\Gamma \vdash e_1 : \mathsf{bool}$ *and* $\Gamma \vdash e_2, e_3 : T$.

Proof. The proof immediately follows from the type rules for FS expressions rules given in Section 6.3. □

Lemma 3 (Permutation)**.**
If $\Gamma \vdash e : T$ *can be derived and* Δ *is a permutation of* Γ, *then the judgment* $\Delta \vdash e : T$ *can be derived and the derivation has the same height of the previous one.*

Proof. We prove this lemma in Appendix E.1. □

Lemma 4 (Weakening)**.**
Let $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ *be a derivable judgment, and let* Δ *be such that* $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$ *(i.e.* Γ *and* Δ *have no elements in common). Then* $\Gamma \cdot \Delta \vdash \langle \beta, \sigma, e \rangle : T$ *can be derived and its derivation has the same height as the previous one.*

Proof. We prove this lemma in Appendix E.2. □

Lemma 5 (Substitution)**.**
If $\Gamma, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}, \mathsf{msg.value} : \mathsf{uint} \vdash e : T$*,* $\Gamma \vdash c : C$*,* $\Gamma \vdash a : \mathsf{address}$*, and* $\Gamma \vdash n : \mathsf{uint}$*, then* $\Gamma \vdash e\{\mathsf{this} := c, \mathsf{msg.sender} := a, \mathsf{msg.value} := n\} : T$.

Proof. We prove this lemma in Appendix E.3. □

Lemma 6 (Canonical forms)**.**

1. *If* v *is a value of type* bool*, then* v *is either* true *or* false*.*
2. *If* v *is a value of type* uint*, then* v *is an non-negative integer number* n*.*
3. *If* v *is a value of type* unit*, then* v *is* u*.*
4. *If* v *is a value of type* $\mathsf{mapping}(T_1 \Rightarrow T_2)$*, then* v *is a total function from* T_1 *to* T_2*.*
5. *If* v *is a value of type* C*, then* v *is a contract reference* c*.*
6. *If* v *is a value of type* $\tilde{T_1} \to T_2$*, then* v *is a pointer to a function defined into a contract reference* c*.*
7. *If* v *is a value of type* address*, then* v *is an address* a*.*

Proof. The proof immediately follows from the type rules for FS expressions in Section 6.3 and Lemma 2. □

The proof of Theorem 1 uses the theorems of Progress and Preservation, formalized by Theorems 3 and 4.

Theorem 3 (Progress)**.**
If $\langle \beta, \sigma, e \rangle$ *is closed (Definition 13) and well-typed (i.e.* $\exists \Gamma$ *such that* $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ *is derivable), then either* $e = v$*,* $e = \mathsf{revert}$*, or* $\exists (\beta', \sigma', e')$ *such that* $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$*.*

Proof. We prove this theorem in Appendix E.4. □

Theorem 4 (Subject Reduction)**.**
If $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ *with* $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ *then* $\exists \Delta$ *such that* $\Gamma' = \Gamma \cdot \Delta$ *and* $\Gamma' \vdash \langle \beta', \sigma', e' \rangle : T$*.*

Proof. We prove this theorem in Appendix E.5. □

In other words, Theorem 2 states that a well-typed group of transactions is never stuck in the middle of a transaction: it either evaluates toward a final value (i.e. it reaches the final state $\langle \beta_n, v \rangle$) or throws a revert somewhere during the evaluation (i.e. $\langle \beta_i, \mathsf{revert} \rangle$).

Chapter 7

Extending the type system

What we have defined so far formalizes a version of FS where every terminating program terminates either successfully or with a revert. The type system we have defined rules out some errors, but cannot do anything against the following ones:

1. transferring Wei to a contract without a (payable) fallback function;
2. wrong cast $C(a)$. This happens when $\beta(a) = (C', s, v, n)$, with $C' \neq C$;
3. transferring Wei from a contract with an insufficient balance;
4. the transaction ran out of gas.

As we said before, we do not take into account Case 4, since we do not model gas. Furthermore, we cannot deal with Case 3 at compile-time, since we do not have any information about any balances. Fortunately, we can actually do something to make the behavior of FS more predictable for what concerns Cases 1 and 2. In fact, an extension of the type system can propagate the information about the contract definition corresponding to each address, thus making it possible to check, without executing the program, if transfers and casts are correct.

To this end, we propose an extension of FS, FS$^+$, in which the address type is annotated with a contract name, so that it will be easy to get the actual code corresponding to it. In this way, given an address a, we are able to know the exact interface exposed by the contract reference pointed to by a. We shall introduce subtyping on contract and address types, by means of which we shall provide retro-compatibility with the legacy code. We shall discuss in more detail the impact of our changes later on in this chapter.

7.1 Syntax

FS$^+$ requires some changes in the syntax of FS. As we said, we introduce nominal subtyping for contract definitions and we replace the type address with a new type address$\langle$C$\rangle$ keeping track of the contract definition corresponding to each address. Figure 7.1 lists the modifications applied to the syntax.

$$
\begin{array}{llll}
(Contract\ declaration) & SC & ::= & \mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \\
(Constructor\ declaration) & K & ::= & C\ (\tilde{T_1}\,y, \tilde{T_2}\,x)\ \{\mathsf{super}(\tilde{y}); \mathsf{this}.\tilde{s}\ =\ \tilde{x}\} \\
(Function\ declaration) & F & ::= & T\ f\langle C\rangle\ (\tilde{T}\,x)\ \{\mathsf{return}\ e\}\ | \\
 & & & \mathsf{unit}\ fb\langle C\rangle\ ()\ \{\mathsf{return}\ e\} \\
(Types) & T & ::= & \tilde{T} \to T\ |\ \mathsf{bool}\ |\ \mathsf{uint}\ |\ \mathsf{address}\langle \mathsf{C}\rangle\ |\ \mathsf{unit}\ | \\
 & & & \mathsf{mapping}(T \Rightarrow T)\ |\ C
\end{array}
$$

Figure 7.1: Changes to the syntax in FS$^+$

The new SC reflects the nominal subtyping, where the supertype is specified in the contract declaration after the keyword is. $\mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\}$ means C inherits from (or extends) D, where D must also be declared in the contract table CT. C inherits any fields and functions declared in D, and we assume that the identifiers $\tilde{x}$ are not used in D, so that there is no shadowing of state variables. As we will see, overriding is supported. The constructor K now includes also the parameters, $\tilde{T_1}\,y$, for the constructor of the supertype, invoked via $\mathsf{super}(\tilde{y})$. F changed, too. Note how we modify the signature with the annotation $\langle C\rangle$: this is to explicitly state the required maximum supertype allowed for the implicit variable msg.sender. We shall validate this constraint at compile-time, supposing msg.sender as an expression of type $\mathsf{address}\langle \mathsf{C}\rangle$ and checking the subtyping for each call of f. Lastly, types T have been modified replacing address with $\mathsf{address}\langle \mathsf{C}\rangle$. No further modifications are needed to implement our extension.

Note that mutually recursive definitions are allowed. In other words, it may happen that two contracts C and D are defined as follows:

$$\mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\}, \text{ and}$$

$$\mathsf{contract}\ D\ \mathsf{is}\ C\ \{\tilde{T}\,s;\ K\ \tilde{F}\}$$

Even if this not explicitly forbidden by the grammar, we assume non-mutually recursive contract definitions. A violation in this sense can be easily found (and consequently ruled out) via a static code inspection.

As pointed out by Section 2.4.4, Solidity supports multiple inheritance: we discussed about the issues related to this choice and the existing solutions. It is not the aim of FS$^+$ to provide a detailed formalization of Solidity's subtyping, and indeed we do not introduce any multiple inheritance, interfaces of abstract contracts. In FS$^+$ the subtyping (for what concerns smart contracts programmers) is really simple: when a contract is defined, it must specify its direct supercontract, with no exceptions. The following Section explains it in detail.

7.2 Subtyping

From the definition of SC in Figure 7.1 follows that any contract in FS$^+$ now must extend one and only one contract. We assume the existence of two contracts, Top and $\mathsf{Top_{fb}}$. The former is similar to Object in Java, in the sense that each hierarchy begins with Top, but there is no declaration for it in CT and there are no Top instances

on the blockchain. Contracts implementing a fallback function also implicitly extend $\mathsf{Top_{fb}}$. Such extension is actually defined by structural subtyping and is handled by the compiler, which checks if the contract being compiled defines a fallback. This kind of subtyping is named "structural" since it is based on the *structure* of a contract (i.e. its definition), and not explicitly declared by means of a keyword (e.g. is in FS). Thus, programmers can neither control it in any ways nor create hierarchies by structural subtyping. The reason behind $\mathsf{Top_{fb}}$ will be clarified later on in this chapter.

Definition 14 defines the subtyping relation in FS$^+$.

Definition 14 (Subtyping relation)**.** Subtyping ($<:$) in FS$^+$ is a reflexive and transitive relation inductively defined as follows:

$$\frac{}{C <: \mathsf{Top}}\ (\textsc{Top}) \qquad \frac{\mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\}}{C <: D}\ (\textsc{Contract})$$

$$\frac{}{C <: C}\ (\textsc{Reflexivity}) \qquad \frac{C <: D \qquad D <: E}{C <: E}\ (\textsc{Transitivity})$$

Where Top and Reflexivity are the base cases. If $C <: D$ we say that D is a supercontract for C and, vice versa, C is a subcontract of D. We define the relation as covariant on expressions of type $\mathsf{address}\langle\mathsf{C}\rangle$, that is $\mathsf{address}\langle\mathsf{C}\rangle <: \mathsf{address}\langle\mathsf{D}\rangle$ if and only if $C <: D$. Bearing in mind the Liskov substitution principle in object-oriented programming[1], an expression of type $\mathsf{address}\langle\mathsf{C}\rangle$ should be correctly used whenever an expression of type $\mathsf{address}\langle\mathsf{D}\rangle$ is required, if $C <: D$. One may wonder why covariance is correct instead of contravariance. The reason is that, in FS, as well as in Solidity and FS$^+$, addresses are used to carry out casts and money transfers (i.e. expressions like $C(a)$ and $a.\mathsf{transfer}(n)$, respectively, where a is an address and n an integer). As we have already said, these operations require a knowledge about the interface of the contract pointed to by a given address. In the former case , $C(a)$, the operation should be successful if and only if a points to a contract that is *at least* C (or better, that expose *at least* the same interface as C does). The very same applies to $a.\mathsf{transfer}(n)$, that requires the contract pointed to by a (say, an instance of C) to define a fallback function. However, C may not define a fallback *itself*, but it may inherit it from one of its supercontracts. If this is the case, the operation can be compiled and executed with no worries, since the hierarchy provides a valid definition of the fallback. In brief, for $\mathsf{address}\langle\mathsf{C}\rangle <: \mathsf{address}\langle\mathsf{D}\rangle$ to be true, C must present *at least* the same interface as D does, that is $C <: D$. This is known as covariance. Contravariance, on the other hand, says the opposite: $\mathsf{address}\langle\mathsf{C}\rangle <: \mathsf{address}\langle\mathsf{D}\rangle$ is true if and only if $D <: C$, but this would mean that addresses pointing to a *supertype* could be used when an address pointing to a *subtype* is required. Taking $D(a)$ as an example, this would have the consequence that a would refer to a contract whose interface is *at most* D's one. Covariance translates in symbols as follows:

$$\frac{C <: D}{\mathsf{address}\langle\mathsf{C}\rangle <: \mathsf{address}\langle\mathsf{D}\rangle}\ (\textsc{Address-Covariance})$$

[1] *Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.*

Lastly, whenever a contract C implements a fallback function, it automatically extends $\mathsf{Top_{fb}}$, as follows:

(STRUCTURAL FALLBACK - 1)

$$\frac{CT(C) = \mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \qquad \mathsf{unit}\ fb\ ()\ \{\mathsf{return}\ e\} \in \tilde{F}}{C <: \mathsf{Top_{fb}}}$$

(STRUCTURAL FALLBACK - 2)

$$\frac{CT(C) = \mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \qquad D <: \mathsf{Top_{fb}}}{C <: \mathsf{Top_{fb}}}$$

7.3 Operational semantics

Some changes are due in the lookup functions defined in Figure 5.4: C's state variables are not only the ones declared in C, but also those defined in the contract C inherits from. We have supposed no variable shadowing, so identifiers are assumed as unique. A similar reasoning applies to function lookup (both of type or body), and overriding is implemented in the expected way: when a function f of C is called, we first look for it in C. If C defines f, its body is executed. Otherwise, we go up in the hierarchy to find a suitable definition of f, if any. Figure 7.2 shows the new definitions.

State variable lookup:

$$\mathrm{sv}(\mathsf{Top}) = \emptyset$$

$$\frac{CT(C) = \mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \qquad \mathrm{sv}(D) = \tilde{T_1}\,r}{\mathrm{sv}(C) = \tilde{T_1}\,r; \tilde{T_2}\,s}$$

Function body lookup:

$$\mathrm{fbody}(\mathsf{Top}, fb, \tilde{v}) = (\{\}, \mathsf{return\ revert})$$

$$\frac{CT(C) = \mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \quad T\ f\langle C\rangle\ (\tilde{T}\,x)\ \{\mathsf{return}\ e\} \in \tilde{F} \quad |\tilde{x}| = |\tilde{v}|}{\mathrm{fbody}(C, f, \tilde{v}) = (\tilde{x}, \mathsf{return}\, e)}$$

$$\frac{CT(C) = \mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \quad T\ f\langle C\rangle\ (\tilde{T}\,x)\ \{\mathsf{return}\ e\} \notin \tilde{F} \vee |\tilde{x}| \neq |\tilde{v}|}{\mathrm{fbody}(C, f, \tilde{v}) = \mathrm{fbody}(D, f, \tilde{v})}$$

Function signature lookup:

$$\frac{CT(C) = \mathsf{contract}\ C\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \qquad B\ f\ (\tilde{A}\,x)\ \{\mathsf{return}\, e;\} \in \tilde{F}}{\mathrm{ftype}(C, f) = \tilde{A} \to B}$$

$$\frac{CT(C) = \mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \qquad T\ f\langle C\rangle\ (\tilde{T}\,x)\ \{\mathsf{return}\ e\} \notin \tilde{F}}{\mathrm{ftype}(C, f) = \mathrm{ftype}(D, f)}$$

Figure 7.2: Redefined lookup functions

Function sv is as described: it takes the state variables from C and recursively from all the contracts above it in the hierarchy. The base case is defined on Top, that has no state variables. $\mathrm{fbody}(C, f, \tilde{v})$ looks for f in C as in Figure 5.4. If no suitable f is defined in C (i.e. due to no definition of f at all or different number of parameters), fbody recursively tries to look f up in its supercontract D. As before, if a fallback function lookup get to Top a revert is thrown. ftype behaves in a similar way, as expected.

In order to prove that FS$^+$ is safer than FS, we leave almost unaltered the semantics given in Figure 5.3 and Section 5.5: in this way, the conditions under which reverts are thrown remain the same. However, the type system will be more sophisticated and consequently able to check Cases 1 and 2 (of the list at the beginning of this chapter). The only change we make regards rules CONTRRETR and CONTRRETR-R, whose new version is given below.

(CONTRRETR)
$$\frac{\beta^C(a) = D' \qquad \hat{\beta}(a) = c \qquad D' <: D}{\langle \beta, \sigma, D(a) \rangle \longrightarrow \langle \beta, \sigma, c \rangle}$$

(CONTRRETR-R)
$$\frac{\beta^C(a) = D' \qquad D' \not<: D}{\langle \beta, \sigma, D(a) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle}$$

7.4 Type system

The vast majority of the changes are applied to the type system, which has to be adapted to the subtyping and to the new address⟨C⟩.

First, we adapt Γ as shown in Definition 15.

Definition 15 (Context)**.**

$$(\mathit{Type\ environment}) \quad \Gamma \quad ::= \quad \emptyset \mid \Gamma,\ x : T \mid \Gamma,\ a : \mathsf{address}\langle \mathsf{C} \rangle \mid \Gamma,\ c : C$$

Definition 16 then extends Definition 2 and Definition 3 to include the new Γ.

Definition 16 (Domains with type environment)**.** Let Γ be well formed (i.e. $\Gamma \vdash \langle\rangle$). Then its domain is now defined as follows:

$$\begin{array}{lcl} \mathrm{dom}(\emptyset) & = & \emptyset \\ \mathrm{dom}(\Gamma,\ x : T) & = & \{x\} \cup \mathrm{dom}(\Gamma) \\ \mathrm{dom}(\Gamma,\ a : \mathsf{address}\langle \mathsf{C} \rangle) & = & \{a\} \cup \mathrm{dom}(\Gamma) \\ \mathrm{dom}(\Gamma,\ c : C) & = & \{c\} \cup \mathrm{dom}(\Gamma) \end{array}$$

Well-formedness for contracts and functions We now redefine the rules given in Section 6.3 and add an additional predicate, override checking the well-formedness of method overriding.

$$\frac{\mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}\langle \mathsf{C}' \rangle, \mathsf{msg.value} : \mathsf{uint}, \tilde{x} : \tilde{T_1} \vdash e : T_2' \qquad T_2' <: T_2 \qquad CT(C) = \mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T s};\ K\ \tilde{F}\} \qquad \mathrm{override}(f, D, \tilde{T_1} \to T_2)}{T_2\ f\langle C' \rangle\ (\tilde{T_1 x})\ \{\mathsf{return}\ e\}\ OK\ in\ C}$$

$$\frac{\text{If ftype}(f, D) = \tilde{T_1}' \to T_2' \text{ then } \tilde{T_1}' = \tilde{T_1} \text{ and } T_2' = T_2}{\mathrm{override}(f, D, \tilde{T_1} \to T_2)}$$

$$\frac{K = C\ (\tilde{T_1 y}, \tilde{T_2 x})\ \{\mathsf{super}(\tilde{y}); \mathsf{this}.\tilde{s}\ =\ \tilde{x}\} \qquad \mathrm{sv}(D) = \tilde{T_1 r} \qquad |\tilde{r}| = |\tilde{y}| \qquad \tilde{F}\ \text{OK in}\ C}{\mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T_2 s};\ K\ \tilde{F}\}\ OK}$$

The definition of a contract C is well-formed if its constructor K calls the constructor of the supercontract with the right parameters, correctly initializes all the state variables of C, and all the functions in C are well-formed. A function f is well-formed in C if its body respect its signature. Note how we use the additional annotation $\langle C' \rangle$ on the function signature: Γ is enlarged with the implicit variable msg.sender which is given the type $\mathsf{address}\langle \mathsf{C}' \rangle$. We shall check at every invocation that the actual sender has *at least* this type. By subtyping, the dynamic type of msg.sender can be also a subtype of C'. Bearing this in mind, we can say that FS corresponds to FS$^+$ where every function is annotated with Top. The predicate override checks if a function override is well-defined: if a function of type $\tilde{T_1} \to T_2$ overrides a function of the supercontract D, then the signature of the same function in D is equal. Note how override is true also for the functions declared by C itself.

Well-formedness for blockchains The new rules used to check the well-formedness of β are listed below:

(EMPTYBLOCKCHAIN)

$$\frac{}{\Gamma \vdash \emptyset}$$

(VARIABLE)

$$\frac{\Gamma \vdash \beta \qquad x \notin \mathrm{dom}(\beta) \qquad \Gamma \vdash x : T \qquad \Gamma \vdash v : T' \qquad T' <: T}{\Gamma \vdash \beta \cdot [x \mapsto v]}$$

(CONTRACT)

$$\frac{\Gamma \vdash \beta \qquad (c,a) \notin \mathrm{dom}(\beta) \qquad \Gamma \vdash c : C \qquad \Gamma \vdash a : \mathsf{address\langle C \rangle} \qquad \mathrm{sv}(C) = \tilde{T}\,s \qquad \Gamma \vdash \tilde{v} : \tilde{T}' \qquad \tilde{T}' <: \tilde{T} \qquad \Gamma \vdash n : \mathsf{uint} \qquad |\tilde{s}| = |\tilde{v}|}{\Gamma \vdash \beta \cdot [(c,a) \mapsto (C, \tilde{s:v}, n)]}$$

The main difference with respect to Section 6.3 is that now the actual types of the values stored in the blockchain (for either user-declared or state variables) can be subtypes of the declared ones.

Well-formedness for call stacks The new rule used to give a type to a call stack is defined as follows:

(CALLSTACK)

$$\frac{\Gamma \vdash \sigma \qquad \Gamma \vdash a : \mathsf{address\langle C \rangle}}{\Gamma \vdash \sigma \cdot a}$$

The only difference regards the introduction of the new type $\mathsf{address\langle C \rangle}$.

Well-formedness for configurations The main difference is the introduction of subtyping in e.

(CONFIGURATION)

$$\frac{\Gamma \vdash \beta \qquad \Gamma \vdash \sigma \qquad \Gamma \vdash e : T' \qquad T' <: T}{\Gamma \vdash \langle \beta, \sigma, e \rangle : T}$$

Types of expressions The type system here defined is algorithmic: we do not give a subsumption rule, but instead add directly and explicitly the subtyping in those rules which need it. In this way there is always at most one type rule applying.

In the rules that follow we make use of one more lookup function, defined below:

Sender maximum type lookup:

$$\frac{CT(C) = \mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \qquad T_2\ f\langle C' \rangle\ (\tilde{T_1\,x})\ \{\mathsf{return}\ e\} \in \tilde{F}}{\mathrm{fsender}(C,f) = C'}$$

$$\frac{CT(C) = \mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \qquad f \notin \tilde{F}}{\mathrm{fsender}(C,f) = \mathrm{fsender}(D,f)}$$

$\mathrm{fsender}(C, f)$ retrieves the required type of msg.sender. It looks for a definition of f in C and, if none is found, it goes on the supercontract of C, and so on until Top.

In Section 6.3 we listed the rules used to give types to expressions in FS. We now give the new ones in FS$^+$. Some of the former have remained the same and are not showed below. We use C and D to indicate contract types, whereas T, T_1, T_2, and so on represent types as defined in Figure 7.1.

Changes due to the new address type or to algorithmic subtyping Below we list the rules that have changed just to support subtyping or the new type $\mathsf{address}\langle\mathsf{C}\rangle$.

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

(ADDR)
$$\frac{\Gamma \vdash e : C}{\Gamma \vdash \mathsf{address}(e) : \mathsf{address}\langle\mathsf{C}\rangle}$$

(MAPPING)
$$\frac{M = \{(\tilde{k}, \tilde{v})\} \quad \Gamma \vdash \tilde{k} : \tilde{T_1'} \quad \Gamma \vdash \tilde{v} : \tilde{T_2'} \quad \tilde{T_1'} <: T_1 \quad \tilde{T_2'} <: T_2}{\Gamma \vdash M : \mathsf{mapping}(T_1 \Rightarrow T_2)}$$

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

(BAL)
$$\frac{\Gamma \vdash e : \mathsf{address}\langle\mathsf{C}\rangle}{\Gamma \vdash \mathsf{balance}(e) : \mathsf{uint}}$$

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

(ASS)
$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash e : T' \quad T' <: T}{\Gamma \vdash x = e : T'}$$

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

(MAPPASS)
$$\frac{\Gamma \vdash e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2) \quad \Gamma \vdash e_2 : T_1' \quad \Gamma \vdash e_3 : T_2' \quad T_1' <: T_1 \quad T_2' <: T_2}{\Gamma \vdash e_1[e_2 \to e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)}$$

(MAPPSEL)
$$\frac{\Gamma \vdash e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2) \quad \Gamma \vdash e_2 : T_1' \quad T_1' <: T_1}{\Gamma \vdash e_1[e_2] : T_2}$$

(NEW)
$$\frac{\mathrm{sv}(C) = \tilde{T}\,s \qquad \Gamma \vdash \tilde{e} : \tilde{T}' \qquad \tilde{T}' <: \tilde{T} \qquad |\tilde{e}| = |\tilde{s}| \qquad \Gamma \vdash e' : \mathsf{uint}}{\Gamma \vdash \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : C}$$

Casts and money transfers Below we list the most interesting rules of FS$^+$, CONTRRETR and TRANSFER.

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

(TRANSFER)
$$\frac{\Gamma \vdash e_1 : \mathsf{address}\langle \mathsf{C} \rangle \qquad \mathrm{ftype}(C, fb) = \{\} \to \mathsf{unit} \qquad \Gamma \vdash e_2 : \mathsf{uint} \qquad \Gamma \vdash \mathsf{this} : C' \qquad C' <: \mathrm{fsender}(C, fb)}{\Gamma \vdash e_1.\mathsf{transfer}(e_2) : \mathsf{unit}}$$

Consider CONTRRETR first. We now have a precise information about the actual contract definition behind the expression e. Indeed, $D(e)$ is correctly given type D if and only if $\Gamma \vdash e : \mathsf{address}\langle \mathsf{C} \rangle$, where $C <: D$ is also true. In other words, we can cast an address pointing to a subcontract of D to a reference of type D. If $C <: D$ was not true, then the cast would not be safe, since we would be trying to get a reference to a contract of type D using an address pointing to a contract that has nothing to do with D.

TRANSFER is similar. Note the different hypothesis about the recipient: $e_1.\mathsf{transfer}(e_2)$ is safe if it exists a C such that $\Gamma \vdash e_1 : \mathsf{address}\langle \mathsf{C} \rangle$ and $\mathrm{ftype}(C, fb) = \{\} \to \mathsf{unit}$. In other words, e_1 must be an address pointing to a reference of a contract defining a fallback function. Note that if C itself does not define any fallback functions, the recursive definition of ftype looks for it in its supertype, and so on until Top. Nonetheless, provided that ftype is undefined on Top the expression does not compile, as desired. The last hypothesis checks the type of the sender. This latter hypothesis is the one ensuring the well-typing of msg.sender. Let this be an instance of C_1, and $\mathrm{fsender}(C, f) = C_2$. $C_1 <: C_2$ means that this refers to a contract that is subtype of C_2. f asks that every address used as a sender refers to a contract which is *at least* C_2. Since this : C_1 and $C_1 <: C_2$ then its use as a sender is safe. Note that if this was not defined (i.e. the expression occurs as a top-level transaction), the expression would not compile.

Functions Below we list the rules giving a type to functions and function calls.

(FUN)

$$\frac{\Gamma \vdash c : C \qquad \text{ftype}(C, f) = \tilde{T_1} \to T_2 \qquad \Gamma \vdash \mathsf{this} : C' \qquad C' <: \text{fsender}(C, f)}{\Gamma \vdash c.f : \tilde{T_1} \to T_2}$$

(CALL)

$$\frac{\begin{array}{llll}\Gamma \vdash e_1 : C & \Gamma \vdash e_2 : \mathsf{uint} & \text{ftype}(C, f) = \tilde{T_1} \to T_2 & \\ \Gamma \vdash \tilde{e} : \tilde{T_1'} & \tilde{T_1'} <: \tilde{T_1} \qquad |\tilde{e}| = |\tilde{T_1}| & \Gamma \vdash \mathsf{this} : C' & C' <: \text{fsender}(C, f)\end{array}}{\Gamma \vdash e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2}$$

(CALLTOPLEVEL)

$$\frac{\begin{array}{llll}\Gamma \vdash e_1 : C & \Gamma \vdash e_2 : \mathsf{uint} & \text{ftype}(C, f) = \tilde{T_1} \to T_2 & C' <: \text{fsender}(C, f) \\ \Gamma \vdash \tilde{e} : \tilde{T_1'} & \tilde{T_1'} <: \tilde{T_1} \qquad |\tilde{e}| = |\tilde{T_1}| & \Gamma \vdash e_3 : \mathsf{address}\langle \mathsf{C}' \rangle & \end{array}}{\Gamma \vdash e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T_2}$$

(CALLVALUE)

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

The very same reasoning as TRANSFER applies to FUN and CALL. CALLTOPLEVEL is a bit different. Here, an expression is explicitly provided as a sender, e_3, which is supposed to be an address. We retrieve the required type for msg.sender in f, and check $\Gamma \vdash e_3 : \mathsf{address}\langle \mathsf{C}' \rangle$, where $C' <: \text{fsender}(C, f)$.

7.5 Properties of the type system

Changing the type rules requires a change in the Inversion Lemma, too. We formalize the new version in Lemma 7.

Lemma 7 (Inversion)**.**

1. *If* $\Gamma \vdash \mathsf{true} : T$ *can be derived, then* $T = \mathsf{bool}$.
2. *If* $\Gamma \vdash \mathsf{false} : T$ *can be derived, then* $T = \mathsf{bool}$.
3. *If* $\Gamma \vdash n : T$ *can be derived, then* $T = \mathsf{uint}$.
4. *If* $\Gamma \vdash \mathsf{u} : T$ *can be derived, then* $T = \mathsf{unit}$.
5. *If* $\Gamma \vdash a : T$ *can be derived, then* $\exists C$ *such that* $T = \mathsf{address}\langle \mathsf{C} \rangle$ *and* $a : \mathsf{address}(C) \in \Gamma$.

6. *If* $\Gamma \vdash c.f : T$ *can be derived, then* $\exists T_1, T_2$ *such that* $T = \tilde{T_1} \to T_2$, $\text{ftype}(C, f) = \tilde{T_1} \to T_2$, *and* $\Gamma \vdash c : C$. *Furthermore, the signature of* f *requires a* msg.sender *pointing to a contract of type at least* $C' = \text{fsender}(C, f)$, *and* this *refers to a subcontract* C'' *of such* C'*:* $\Gamma \vdash \mathsf{this} : C''$ *can be derived and it is true that* $C'' <: C'$.

7. *If* $\Gamma \vdash M : T$ *can be derived, then* $\exists T_1, T_2$ *such that* $T = \mathsf{mapping}(T_1 \Rightarrow T_2)$, *and all the keys and values are well-typed with a subtype of, respectively,* T_1 *and* T_2.

8. *If* $\Gamma \vdash c : T$ *can be derived, then* $T = C$ *and* $c : C \in \Gamma$.

9. *If* $\Gamma \vdash x : T$ *can be derived, then* $x : T \in \Gamma$.

10. *If* $\Gamma \vdash \mathsf{balance}(e) : T$ *can be derived, then* $T = \mathsf{uint}$ *and* $\exists C$ *such that* $\Gamma \vdash e : \mathsf{address}\langle \mathsf{C} \rangle$.

11. *If* $\Gamma \vdash \mathsf{address}(e) : T$ *can be derived, then* $\exists C$ *such that* $T = \mathsf{address}\langle \mathsf{C} \rangle$ *and* $\Gamma \vdash e : C$.

12. *If* $\Gamma \vdash \mathsf{return}\ e : T$ *can be derived, then* $\exists T'$ *such that* $T' <: T$ *and* $\Gamma \vdash e : T'$ *is derivable.*

13. *If* $\Gamma \vdash e.s : T$ *can be derived, then* $\Gamma \vdash e : C$ *and* $\exists T_i$ *such that* $\text{sv}(C) = \tilde{T} s$ *and* $T = T_i$.

14. *If* $\Gamma \vdash e_1.\mathsf{transfer}(e_2) : T$ *can be derived, then* $T = \mathsf{unit}$, $\Gamma \vdash e_1 : \mathsf{address}\langle \mathsf{C} \rangle$, *and* $\Gamma \vdash e_2 : \mathsf{uint}$. *Furthermore* $\text{ftype}(C, fb) = \{\} \to \mathsf{unit}$ *ensures that* C *contains a fallback* fb, *whose signature requires a* msg.sender *pointing to a contract of type at least* $C' = \text{fsender}(C, fb)$, *and the current contract* C'' *is a subcontract of such* C'*:* $\Gamma \vdash \mathsf{this} : C''$ *can be derived and it is true that* $C'' <: C'$.

15. *If* $\Gamma \vdash \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : T$ *can be derived, then* $T = C$, $\Gamma \vdash e' : \mathsf{uint}$, *and* $\Gamma \vdash \tilde{e} : \tilde{T'}$, *where* $\text{sv}(C) = \tilde{T} s$ *and* $\tilde{T'} <: \tilde{T}$.

16. *If* $\Gamma \vdash D(e) : T$ *can be derived, then* $T = D$ *and* $\exists C$ *such that* $\Gamma \vdash e : \mathsf{address}\langle \mathsf{C} \rangle$ *and* $C <: D$.

17. *If* $\Gamma \vdash e_1; e_2 : T_2$ *can be derived, then* $\Gamma \vdash e_2 : T_2$ *and* $\exists T_1$ *such that* $\Gamma \vdash e_1 : T_1$.

18. *If* $\Gamma \vdash T_1\ x = e; e' : T_2$ *can be derived, then* $\Gamma, x : T_1 \vdash e' : T_2$ *and* $\exists T_1' <: T_1$ *such that* $\Gamma \vdash e : T_1'$.

19. *If* $\Gamma \vdash x = e : T$ *can be derived, then* $\Gamma \vdash x : T$ *and* $\exists T' <: T$ *such that* $\Gamma \vdash e : T'$.

20. *If* $\Gamma \vdash e_1.s = e_2 : T$ *can be derived, then* $\Gamma \vdash e_1 : C$, $\Gamma \vdash e_1.s : T$, *and* $\exists T' <: T$ *such that* $\Gamma \vdash e_2 : T'$.

21. *If* $\Gamma \vdash e_1[e_2] : T$ *can be derived, then* $\exists T_1, T_2$ *such that* $T = T_2$, $\Gamma \vdash e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, *and* $\exists T_1' <: T_1$ *such that* $\Gamma \vdash e_2 : T_1'$.

22. *If* $\Gamma \vdash e_1[e_2 \to e_3] : T$ *can be derived, then* $\exists T_1, T_2$ *such that* $\Gamma \vdash e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash e_2 : T_1$, $T = \mathsf{mapping}(T_1 \Rightarrow T_2)$, *and* $\exists T_1' <: T_1, T_2' <: T_2$ *such that* $\Gamma \vdash e_2 : T_1'$ *and* $\Gamma \vdash e_3 : T_2'$.

23. *If* $\Gamma \vdash e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T$ *can be derived, then* $\exists \tilde{T_1}, T_2$ *such that* $T = \tilde{T_1} \to T_2$, $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$, $\Gamma \vdash e_1 : C$, $\Gamma \vdash e_2 : \mathsf{uint}$, *and* $\Gamma \vdash \tilde{e} : \tilde{T_1'}$, *where* $\tilde{T_1'} <: \tilde{T_1}$. *Furthermore, the signature of* f *requires a* msg.sender *pointing to a contract of type at least* $C' = \mathrm{fsender}(C, f)$, *and* this *refers to a subcontract* C'' *of such* C'*:* $\Gamma \vdash \mathsf{this} : C''$ *can be derived and it is true that* $C'' <: C'$.

24. *If* $\Gamma \vdash e_1.\mathsf{value}(e_2)(\tilde{e}) : T$ *can be derived, then* $\exists T_1, T_2$ *such that* $T = \tilde{T_1} \to T_2$, $\Gamma \vdash e_1 : \tilde{T_1} \to T_2$, $\Gamma \vdash e_2 : \mathsf{uint}$, *and* $\Gamma \vdash \tilde{e} : \tilde{T_1'}$, *where* $\tilde{T_1'} <: T_1$.

25. *If* $\Gamma \vdash e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T$ *can be derived, then* $\exists T_1, T_2$ *such that* $T = \tilde{T_1} \to T_2$, $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$, $\Gamma \vdash e_1 : C$, $\Gamma \vdash e_2 : \mathsf{uint}$, $\Gamma \vdash e_3 : \mathsf{address}$, *and* $\Gamma \vdash \tilde{e} : \tilde{T}$. *Furthermore, the signature of* f *requires a* msg.sender *pointing to a contract of type at least* $C' = \mathrm{fsender}(C, f)$, *and* e_3 *is an address referring to a subcontract of* C'*:* $\Gamma \vdash e_3 : \mathsf{address}\langle \mathsf{C}' \rangle$.

26. *If* $\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ *can be derived, then* $\exists T_2, T_3$ *such that* $\Gamma \vdash e_1 : \mathsf{bool}$, $\Gamma \vdash e_2 : T_2$, $\Gamma \vdash e_3 : T_3$ *with* $T_2, T_3 <: T$.

Proof. The proof immediately follows from the type rules for FS$^+$ expressions in Section 7.4. □

Lemma 8 formalizes the canonical form for the new type $\mathsf{address}\langle \mathsf{C} \rangle$.

Lemma 8 (Canonical forms)**.**

1. *If* v *is a value of type* $\mathsf{address}\langle \mathsf{C} \rangle$, *for some* C, *then* v *is an address* a.

Proof. The proof immediately follows from the type rules for FS$^+$ expressions in Section 7.4 and Lemma 7. □

Lemma 5 slightly changes as a consequence to the new type $\mathsf{address}\langle \mathsf{C} \rangle$ as stated in Lemma 9.

Lemma 9 (Substitution)**.**
If $\Gamma, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}\langle \mathsf{D} \rangle, \mathsf{msg.value} : \mathsf{uint} \vdash e : T$, $\Gamma \vdash c : C'$, $\Gamma \vdash a : \mathsf{address}\langle \mathsf{D}' \rangle$, *and* $\Gamma \vdash n : \mathsf{uint}$, *with* $D' <: D$, *then*
$\Gamma \vdash e\{\mathsf{this} := c, \mathsf{msg.sender} := a, \mathsf{msg.value} := n\} : T$.

Proof. We prove this lemma in Appendix F.3. □

The other properties stated in Chapter 6, and recalled below, do not change and are proven in Appendix F.

Theorem 3 (Progress)**.**
If $\langle \beta, \sigma, e \rangle$ *is closed (Definition 13) and well-typed (i.e.* $\exists \Gamma$ *such that* $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ *is derivable), then either* $e = v$, $e = \mathsf{revert}$, *or* $\exists (\beta', \sigma', e')$ *such that* $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$.

Proof. We prove this theorem in Appendix F.4. □

Theorem 4 (Subject Reduction)**.**
If $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ *with* $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ *then* $\exists \Delta$ *such that* $\Gamma' = \Gamma \cdot \Delta$ *and* $\Gamma' \vdash \langle \beta', \sigma', e' \rangle : T$.

Proof. We prove this theorem in Appendix F.5. □

Theorem 1 (Type safety for configurations)**.**
If $\Gamma \vdash \langle \beta, \beta, e \rangle : T$*,* $\langle \beta, \beta, e \rangle$ *is closed, and* $\exists(\beta', \sigma', e')$ *such that* $\langle \beta, \beta, e \rangle \longrightarrow^* \langle \beta', \sigma', e' \rangle$*, with* $\langle \beta', \sigma', e' \rangle \not\longrightarrow$*, then either* $e' = v$*, where* v *is a value, or* $e' = \mathsf{revert}$*.*

Proof. We prove this theorem in Appendix F.6. □

Theorem 2 (Type safety for programs)**.**
Let $\mathcal{P} = (CT, \beta, e_1; \ldots; e_n)$ *be an FS program. If* $\Gamma \vdash (CT, \beta, e_1; \ldots; e_n) : T$*,* $\mathcal{P}$ *is closed, and* $\exists(\beta', e')$ *such that* $\langle \beta, e_1; \ldots; e_n \rangle \Longrightarrow^* \langle \beta', e' \rangle$*, with* $\langle \beta', e' \rangle \not\Longrightarrow$*, then either* $e' = v$ *or* $e' = \mathsf{revert}$*.*

Proof. We prove this theorem in Appendix F.6. □

In addition, we can prove two more theorems on this type system, as stated below.

Cast safety Definition 17 defines cast-safe programs.

Definition 17 (Cast-safe programs)**.** Let $\mathcal{P} = (CT, \beta, e)$ be an FS$^+$ program. $\mathcal{P}$ is cast-safe if the evaluation of e does not make use of the rule CONTRRETR-R.

Example 7.1 shows two FS programs: the former is cast safe, whereas the latter is not.

Example 7.1 (Cast safety)**.** Consider the FS code in Listing 7.1, defining two very simple contracts A and B.

```
contract A{
    uint f() {
        return 5
    }
}

contract B{
    uint g() {
        return 0
    }
}
```

Listing 7.1: Simple FS contracts to demonstrate cast safety

Suppose $CT = A \cdot B$, and consider now the following three programs:

$$\begin{array}{lcl} \mathcal{P}_1 & := & (CT, \emptyset, e_1) \\ \mathcal{P}_2 & := & (CT, \emptyset, e_2) \end{array}$$

Where e_1, e_2, and e_3 are as follows:

$$\begin{array}{lll} e_1 := & \mathsf{address}\ a = \ldots; & \\ & A(a).f.\mathsf{value}().\mathsf{sender}(\ldots)() & \end{array}$$

$$\begin{array}{lll} e_2 := & \mathsf{address}\ b = \ldots; & \\ & A(b).f.\mathsf{value}().\mathsf{sender}(\ldots)() & \end{array}$$

We used ... as a sender since it is not important here. Supposing a points to an instance of A and b points to an instance of B, $\mathcal{P}_1$ is cast safe. On the other hand, $\mathcal{P}_2$ is not: it attempts to execute a wrong cast, since a does not point to an instance of B, ad evaluates into a revert by rule CONTRRETR-R.

With cast-safe programs, programmers can be sure that each address they use points to the intended contract reference, avoiding unexpected aborts and, consequently, money loss. Theorem 5 formalizes which programs are cast-safe in FS$^+$.

Theorem 5 (Cast safety)**.**
If $\mathcal{P} = (CT, \beta, e)$ is closed and well-typed then $\mathcal{P}$ is cast-safe.

Proof. We prove this theorem in Appendix F.7. □

Transfer safety Definition 18 defines transfer-safe programs.

Definition 18 (Transfer-safe programs)**.** Let $\mathcal{P} = (CT, \beta, e)$ be an FS$^+$ program. $\mathcal{P}$ is transfer-safe if the evaluation of e does not make use of the rule TRANSFER where $\mathrm{fbody}(C, fb, \{\}) = (\{\}, \mathsf{return\ revert})$.

With transfer-safe programs, programmers can be sure that money transfers target only contracts defining a fallback function, thus avoiding unexpected aborts and, consequently, money loss. Note that Definition 18 only takes into account reverts generated due to the lack of a fallback function: it does not consider either those thrown due to an insufficient balance or those explicitly raised by programmers in the body of a valid fallback function. Theorem 6 formalizes which program are transfer-safe in FS$^+$. Example 7.2 shows three FS programs: $\mathcal{P}_1$ and $\mathcal{P}_2$ are transfer safe, whereas $\mathcal{P}_3$ is not.

Example 7.2 (Transfer safety)**.** Consider the FS code in Listing 7.2 defining three very simple contracts A, B, and C.

```
contract A{
    unit fb() {
        return u
    }
}

contract B{
    unit fb() {
        revert
    }
}

contract C{}
```

Listing 7.2: Simple FS contracts to demonstrate transfer safety

Suppose $CT = A \cdot B$, and consider now the following three programs:

$$\begin{array}{lcl} \mathcal{P}_1 & := & (CT, \emptyset, e_1) \\ \mathcal{P}_2 & := & (CT, \emptyset, e_2) \\ \mathcal{P}_3 & := & (CT, \emptyset, e_3) \end{array}$$

Where e_1, e_2, and e_3 are as follows:

$$\begin{aligned} e_1 := \quad & A\ a = \mathsf{new}\ A(); \\ & a.\mathsf{transfer}(10) \end{aligned}$$

$$\begin{aligned} e_2 := \quad & B\ b = \mathsf{new}\ B(); \\ & b.\mathsf{transfer}(10) \end{aligned}$$

$$\begin{aligned} e_3 := \quad & C\ c = \mathsf{new}\ C(); \\ & c.\mathsf{transfer}(10) \end{aligned}$$

For the sake of simplicity we suppose that $\mathrm{transfer}$ can be invoked at the top level. In real FS code, the same behavior as e_1 can be achieved with the statement $a.fb.\mathsf{value}(10).\mathsf{sender}(\dots)()$, an similarly for e_2 and e_3. $\mathcal{P}_1$ is transfer safe, since A correctly defines a fallback function. Also $\mathcal{P}_2$ is transfer safe, since revert is legitimately thrown by programmers themselves as part of fb's body. On the other hand, $\mathcal{P}_3$ is not transfer safe, because it does not define any fallbacks and the execution evaluates toward a revert by rule TRANSFER ($\mathrm{fbody}(C, fb, \{\})$ returns return revert).

Theorem 6 (Transfer safety)**.**
If $\mathcal{P} = (CT, \beta, e)$ is closed and well-typed then $\mathcal{P}$ is transfer-safe.

Proof. We prove this theorem in Appendix F.8. □

Benefits of cast- and transfer-safe programs are clear. As we said, these two errors can lead to transaction aborts and money loss. The extension we are proposing can effectively rule out both of such errors, making smart contracts sounder.

7.6 Detecting vulnerabilities

Many recent works aiming to detect vulnerabilities in smart contracts written in Solidity have been presented over the last few months. Most of them investigate on the reentrancy bug, the one behind the DAO attack [47], or on Solidity's way to deal with exceptions. Some do so analyzing the EVM bytecode [31][2]; others look at the problem from a concurrent point of view [44] or compile down Solidity code into languages with more powerful type systems in order to benefit from their expressiveness [8]. Atzei, Bartoletti, and Cimoli [6] provide a survey of attacks and vulnerabilities in Solidity code. Alongside reentrancy and exceptions are many other subtle bugs that can make a contract misbehave. To the best of our knowledge, there are no works investigating the type safety of casts and transfers. As we said, aborting transactions containing incorrect casts or transfers does not harm any account, but the sender of the transaction itself. A malicious contract can clumsily implement such operations with no other aim of making a miner richer and to make the accounts initiating the transaction waste money. On the other hand, a sound contract would have all the interest in providing an implementation enhanced with some additional safety guarantees. Example 7.3 clarifies such a scenario.

Example 7.3 (Cast and transfer vulnerabilities in smart contracts)**.** Consider the contract *Unsafe* in Listing 7.3.

```
contract A {
    A(){}

    uint f(){
        return 5;
    }
}

contract Unsafe{
    mapping(address ⇒ uint) balances;

    Unsafe(mapping(address ⇒ uint) _balances) {
        this.balances = _balances;
    }

    unit deposit() {
        return this.balances[msg.sender → this.balances[msg.sender] + msg.value]; unit
    }

    uint unsafeCast() {
        return A(msg.sender).f();
    }

    unit unsafeWithdraw() {
        return uint b = this.balances[msg.sender];
            this.balances[msg.sender → 0];
            msg.sender.transfer(b);
            unit
    }
}
```

Listing 7.3: Unsafe transfer and cast operations

It implements a basic form of the withdrawal pattern[2] allowing users to deposit and afterwards withdraw an amount of Ether. Suppose this contract has an invariant stating that *a user can always withdraw the amount of Ether they have deposited so far*. The problem here is that the function *unsafeWithdraw* calls transfer on msg.sender without knowing if the contract corresponding to msg.sender actually defines a fallback. Hence, the withdrawal may fail at run-time. Even worse is that the amount of Ether that sender has deposited remains indefinitely locked in *Unsafe*, since a deployed contract has no way to change itself to define a suitable fallback. Hence, *Unsafe* falls into the category of the greedy contracts defined by Nikolic et al. [35]. In fact, Ether sent to *Unsafe* from contract lacking a fallback cannot be withdrew, thus failing the aforementioned invariant. Such behavior is surely not malicious, but *Unsafe* could provide additional guarantees.

Another unsafe use of Solidity is in function *unsafeCast*. It is really trivial and simply attempts to retrieve an instance of a contract A using msg.sender. For the sake of simplicity, A defines just a function, f, returning the constant integer 5. The problem here is that *Unsafe* does not know if msg.sender will actually point to an instance of A. The documentation advices to make sure of that before going with the cast, but provides no additional or automatic ways to ensure such constraint, and nor does the compiler. Hence, if msg.sender corresponds to another contract, say B, a revert will be raised at run-time, aborting the transaction and making that sender lose money just

[2] https://solidity.readthedocs.io/en/develop/common-patterns.html#withdrawal-from-contracts

due to someone else's clumsy (or not robust enough) implementation.

Consider now the snippet in Listing 7.4, showing the same *Unsafe* (now renamed as *Safe*) contract using the full power of FS$^+$. A has remained the same and is thus omitted.

```
contract Safe is Top{
    mapping(address⟨Top_fb⟩ ⇒ uint) balances;

    Safe(mapping(address⟨Top_fb⟩ ⇒ uint) _balances) {
        this.balances = _balances;
    }

    unit deposit⟨Top_fb⟩() {
        return this.balances[msg.sender → this.balances[msg.sender] + msg.value]; unit
    }

    uint safeCast⟨A⟩() {
        return A(msg.sender).f();
    }

    unit safeWithdraw⟨Top_fb⟩() {
        return uint b = this.balances[msg.sender];
            this.balances[msg.sender → 0];
            msg.sender.transfer(b);
            unit
    }
}
```

Listing 7.4: Safe transfer and cast operations

Note the major change in the function signatures and in the declaration of *balances*. We impose that the key type must be address⟨Top$_{fb}$⟩. We do the same in the functions using this state variables (i.e. *deposit* and *safeWithdraw*). Similarly, we force callers of *safeCast* to point to an instance of A. The benefit is clear. Consider first the withdrawal of money. Remember that, due to the possible lack of the fallback, with the unsafe definition there was no chance to ensure that an amount of Ether would eventually be withdrew. This has now changed. The compiler checks that *every* call either to *deposit* or *safeWithdraw* comes from a contract defining a fallback. On the one hand this reduces the addresses able to interact with *Safe*: any interaction coming from a contract lacking a fallback will be ruled out at compile-time. On the other hand, the contract's invariant can be now proven and ensured for the entire life of an instance of *Safe*. The job is done by rules FUN, CALL, and CALLTOPLEVEL in Section 6.3. They take the sender (either this or an explicit expression) and make sure its type is a subtype of the required one. Bearing in mind that the latter is Top$_{fb}$, and looking at rules `StructuralFallback` in Definition 14, it is clear that such a type will define for sure a fallback: otherwise STRUCTURALFALLBACK (either 1 or 2) would not apply and the subtyping relation would not hold, thus making the compilation fail.

A similar reasoning applies to *safeCast*, where we impose msg.sender to refer to a subcontract of A. In this case the rules checking the subtyping is CONTRACT in Definition 14 instead of STRUCTURALFALLBACK, but the result does not change: if the subtyping does not hold the compilation fails. At run-time we will be sure that msg.sender can be safely cast to an instance of A, since it points to a contract that also is an instance of A.

7.7 Impact on Solidity

Two are the main additions made by this extension: annotating addresses with contract names and subtyping. The latter is a little more limited than Solidity's one. It is not the aim of this extension to investigate and formalize the way Solidity handles subtyping. In fact, we did not include either multiple inheritance, abstract contracts, or interfaces. In FS$^+$, inheritance is only used for the sake of retro-compatibility and to make it possible a direct mapping from existing contracts to the new type system we are proposing. The rules STRUCTURAL FALLBACK actually introduces a very limited form of multiple inheritance. In fact, a contract implementing a fallback function extends, at the same time, both the contract indicated in its declaration (i.e. contract C is D), by nominal subtyping, and $\mathsf{Top}_{\mathsf{fb}}$ by structural subtyping. Nonetheless, this form of multiple inheritance is a very controlled one, and no ambiguities can arise (since $\mathsf{Top}_{\mathsf{fb}}$ is used only to check a given property of C).

Consider now the new address type: $\mathsf{address}\langle\mathsf{C}\rangle$. It tells the compiler that a value of that type points to an instance of contract C, thus allowing it to inspect its interface. This is, of course, not compatible with Solidity, where address does not keep any information about the contract it refers to. Nonetheless, a direct default mapping is easily definable: remember the contract Top, defined as the base contract for any hierarchy. It is hence natural to map the type address to $\mathsf{address}\langle\mathsf{Top}\rangle$. This provides no guarantees, since we actually still do not know which contract that address refers to. On the other hand, the new code written using this extension allows the compiler to make fine-grained checks. Thus, no changes are required to the already deployed contracts: only the compiler has to be made more complex (as described above) to deal with the additional binding. We will not be able to apply our fine-grained check to the existing contracts, that cannot be modified, but we will be able to improve the reliability of the new ones. We shall also provide a flag (`--notopcast`) in the new compiler to disable rule CONTRRETR in Section 7.4 when $C = \mathsf{Top}$ and allow programmers to compile using rule CONTRRETR in Section 6.3. We do this for the sake of retro-compatibility: CONTRRETR in Section 7.4 would rule out any cast having $\mathsf{address}\langle\mathsf{Top}\rangle$ as actual type of e, since $\forall D$ type . $\mathsf{Top} \not<: D$. All the other casts will be correctly verified.

The major effort required to programmers is to annotate each function with the required type of its sender. This somehow reduces the flexibility provided by default by Solidity, since programmers have to specify in advance, and without the possibility to change it, a maximum supertype for the caller of their functions. This might seem annoying, but actually allows the definition of sounder contracts (as we saw above). One of the most known benefits of types is that they get programmers to know which operations can be invoked on each value. The compiler then checks that every value is used according to its type, i.e. that only allowed operations are invoked on it. Types might sometimes be too rigid, but they are a fundamental tool to make the code clearer, more readable and more correct. On the other hand, the flexibility of dynamically-typed languages (where types are not checked at compile-time) is well-known and largely appreciated by many programmers. Nonetheless, such languages provide weaker guarantees and allow for run-time errors, avoidable thanks to a compiler check. Even worse, such errors might appear in certain executions and might not in others, making it extremely difficult to catch the bug. Catching (and correcting) bugs is almost impossible in Solidity, where contracts may not be redeployed and the only (extreme) solution to dangerous bugs is the suicide. The main problem in Solidity's type safety is the type address, that can be compared to a pointer to `void` (`void *`) in C. Such pointers allow for an extreme flexibility, but are really difficult to deal with since the compiler

does not give any hint on how to use them: programmers have to know what they are doing and how to do so, in order to avoid subtle bugs. They are widely used to implement generic functions, when the precise target type is not known in advance. Nonetheless, the real question is: "Would you allow a `void*` to be cast to a completely arbitrary type, without knowing if that cast will cause run-time errors"? The answer in C is yes, and that can lead to many bugs if not done carefully. Void pointers are a dangerous backdoor in C's type safety.

address is similar: it can refer to an instance of any accounts, and neither Solidity nor the EVM provides additional information on that. As we saw above, this is quite dangerous and can lead to Ether indefinitely locked into a contract or to unexpected run-time reverts. Hence, the main benefit of specifying the *most general required* sender's supertype in a function's signature is that the compiler has a precise knowledge about the operations programmers wish to invoke on the contract the sender refers to. In the context of function *safeCast* above, the compiler blocks any attempts to call it from a contract not extending A, thus making sure that any successful invocation will not raise any reverts due to an unsuccessful cast. In other words, we ask programmers to specify the minimum required interface to accomplish the function's postcondition. Such specification makes the address type safe again, and simplify the task of proving properties and invariants on Solidity's smart contracts and their functions.

Functions in Solidity can, in general, be annotated in many ways: the visibility (`external`, `internal`, `public`, and `private`), `payable`, and an indication of what the body does not do (`view` if it does not modify the state or `pure` if it does not read the state). Using such markers on function signatures is a common practice in Solidity, and makes the code more readable. We provide the following two new annotations as a syntactic sugar:

- `payback` to indicate that the function possibly sends Ether to the sender. This annotation is translated requiring $\mathsf{Top}_{\mathsf{fb}}$ as a most general type;

- `any` to indicate that the function does not impose any restrictions to the sender. This annotation is translated requiring Top as a most general type, and corresponds to the default mapping of legacy code.

It will be then a compiler's task to translate such annotation ensuring the constraints they correspond to. In a similar way, `address` is translated into `address<`Top`>` and `payableaddress` into `address<`$\mathsf{Top}_{\mathsf{fb}}$`>`.

Example 7.4 compares the current Solidity with our enhanced version.

Example 7.4 (Comparison on transfer safety)**.** Let us consider the contract $Unsafe$ in Example 7.3. The same contract (omitting, for the sake of clarity *safeCast*) would be written in Solidity as in Listing 7.5. Listing 7.6, on the other hand, shows the very same contract without our modifications.

```
contract SafeWithdrawal {
    mapping (payableaddress => uint) private balances;

    function deposit() public payable payback {
        balances[msg.sender] += msg.value;
    }

    function withdraw() public payback {
        uint b = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(b);
    }
}
```

Listing 7.5: Safe withdrawal in Solidity

```
contract UnsafeWithdrawal {
    mapping (address => uint) private balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() public {
        uint b = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(b);
    }
}
```

Listing 7.6: Unsafe withdrawal in Solidity

The code looks very similar, with a few differences. First, the key type of `balances` is `payableaddress` and not `address` anymore. This means that only addresses referring to a contract with a fallback function may deposit money in (and successively possibly withdraw from) the contract. Note that the key set space will remain the same: we cannot know in advance which address will correspond to a suitable contract, and thus the set of keys of the two mappings (the one in Listing 7.5 and the one in Listing 7.6) will be the same. Some keys will never be used in the former, but this is not a problem since unused keys are not stored anywhere. Secondly, to enforce the constraint on the mapping's key type, `payback` is used on the two functions managing money. With these additional annotations the code remains readable, and the effort required to programmers is really limited.

Lastly, Example7.5 shows a use-case taking advantage of the full power of the new type system.

Example 7.5 (Requiring a callback interface)**.** This example shows the same application as Example 2.9. Remember that we said there may be some troubles with the interaction of `Oracle` and `Room`: the latter may not define `callback` or may use a wrong address to initialize its state variable `oracle`. Listing 7.7 shows how our extension solves the problem.

```
interface Oracle {
    function execute<Callbackable>(string) external;
}

interface Callbackable {
    function callback(uint) public;
}

contract ConcreteOracle is Oracle {
    event Execute(address<Callbackable>, string);

    function execute<Callbackable>(string url) external {
```

```
13          emit Execute(msg.sender, url);
14      }
15  }
16
17  contract Room is Callbackable {
18      uint public temperature;
19      Oracle oracle;
20
21      constructor(uint _temperature, address<Oracle> _oracle) public
            {
22          temperature = _temperature;
23          oracle = Oracle(_oracle);
24      }
25
26      function getTemperature() {
27          oracle.execute("...");
28      }
29
30      function callback(uint _temperature) public {
31          temperature = _temperature;
32      }
33  }
```

Listing 7.7: Asynchronous interaction of contracts in Solidity

There are two interfaces defining a set of minimum operations: `Oracle` exposes a function `execute()` taking a string representing a URL and, ideally, getting the result of the operation identified by such URL; `Callbackable` defines the function that will be eventually invoked. `ConcreteOracle` then implements `Oracle` as we described above, but note how simple is to specify that the caller of `execute()` must be compliant with the `Callbackable` interface. The compiler will check for us that any attempts to call `execute()` comes from a contract implementing `Callbackable`, thus ensuring that the result can be eventually delivered to it.

An example of this is given by the contract `Room`. It imposes a constraint on a parameter of its constructor, forcing the oracle to adhere to the `Oracle` interface: consequently, the cast at line 23 will always be safe. It then implements the function `callback()` and invokes the oracle via `getTemperature()`. Note that the function `execute()` is `external`: when `getTemperature()` calls it, the implicit variable `msg` will change and its `sender` field will contain the address of the instance of `Room`, instead of the one of the caller of `getTemperature()`.

Again, ensuring properties on addresses using this extension is really simple, with small and limited modifications in the source code (see Listing 2.13), but with large benefits in terms of type safety and provable properties.

As a last note, EOAs in FS$^+$ are modeled by means of contracts (EOC in Definition 1), but in Ethereum are not. Hence, the compiler must block any attempt of cast from an address pointing to an EOA. Internally, the former could still represent these accounts with the type EOC and rule out erroneous casts ($D(a)$) by simply adding the hypothesis $C \neq EOC$, where $\Gamma \vdash a : \mathsf{address}\langle \mathsf{C} \rangle$. Using EOC to internally model EOAs also allows the latter to receive Ether via `transfer`: in fact, by rule StructuralFallback-1, $EOC <: \mathsf{Top}_{\mathsf{fb}}$.

7.8 Separated compilation of smart contracts

In Ethereum, smart contracts are deployed independently of each other. This arises some difficulties for our paradigm, since when a contract is compiled the code of the

contracts it communicates with is not available. This causes behaviors somewhat surprising, as shown by Example 7.6.

Example 7.6 (Separated compilation of smart contracts in Solidity)**.** Consider contract `A` listed in Listing 7.8.

```
1  contract A {
2      uint private x;
3
4      function setX(uint _x) public {
5          x = _x;
6      }
7
8      function getX() view public returns (uint) {
9          return x;
10     }
11
12     function () public payable {}
13 }
```

Listing 7.8: Example of a deployed contract

This simple contract just exposes two functions, `setX()` and `getX()`, allowing programmers to manipulate state variable `x`. It also defines a fallback function. Now consider contract `B` below (Listing 7.9).

```
1  interface AInterface {
2      function setX(uint) public;
3      function getX() public view returns(uint);
4  }
5
6  contract B {
7      AInterface private a;
8
9      constructor (address _addressA) public {
10         a = AInterface(_addressA);
11     }
12
13     function inc() public {
14         a.setX(a.getX() + 1);
15     }
16
17     function read() public view returns (uint) {
18         return a.getX();
19     }
20 }
```

Listing 7.9: Correct interaction with a deployed contract

Programmers of `B` cannot directly refer to `A`, since its code is not available nor is its name known to the compiler compiling `B`. The pattern is then as follows:

- An interface (`AInterface`) is defined exposing the functions of `A` needed by `B`. Such interface is compiled with the latter, which can thus see all the function names it needs. As we saw in Example 2.1, the contract's ABI can be used to know what functions `A` exposes. However, the Solidity code of many contracts is often available at etherscan.com.
- `B` interacts with `A` via `AInterface`. It declares, in this example, a state variable (with static type `AInterface`), initialized through a cast from an address, obtained as a parameter.

- The code compiles, since the compiler checks `B` against `AInterface`, of which it knows all it needs. Nonetheless, Solidity does not check the address used as a parameter for the cast, and lets it compile. At run-time the behavior might be unexpected. The address used as a parameter is known, and the cast is executed regardless of the contract it refers to. If its definition contains a function with the same signature of the one being invoked, the latter is executed. If, on the other hand, no matching function is defined, a `revert` is raised.

Lastly, consider contract `C` below (Listing 7.10).

```
1  interface AWrongInterface {
2      function set(uint) public;
3      function get() public view returns(uint);
4  }
5
6  contract C {
7      AWrongInterface private a;
8
9      constructor (address _addressA) public {
10         a = AWrongInterface(_addressA);
11     }
12
13     function inc() public {
14         a.set(a.get() + 1);
15     }
16
17     function read() public view returns (uint) {
18         return a.get();
19     }
20 }
```

Listing 7.10: Incorrect interaction with a deployed contract

`C` has the same meaning as `B`, with an important difference: it uses `AWrongInterface`, which is similar to `AInterface` above, but misnames the name of the two functions. When either `inc()` or `read()` is invoked, no function is found in `A` with the same signature. This throws a `revert`, even if `A` defines a fallback. Hence, the guarantees given at compile-time fall at run-time, and different things may happen according to the actual address being used as a parameter. This allows for unexpected behaviors and is quite error-prone.

Our proposal makes it possible to solve this issue. Our type $\mathsf{address}\langle \mathsf{C} \rangle$ keeps all the information needed to ensure that a cast is correct. Every address is forced to refer to a contract instance satisfying a given interface. If it does not, it is not compiled. This works under two assumptions: having available all the contract interfaces and implementing structural subtyping. The former is necessary, since we must know all the functions exposed by every contract. The latter is necessary because subtyping has to be handled implicitly by the compiler. When a cast $AInterface(a)$ is carried out, the compiler has to do the following things:

1. Retrieve the interface of the contract A referred to by a. Let I be such interface.
2. Compare every function f in I with those defined in $AInterface$.
3. If every function in $AInterface$ matches with one function in I, then A is a subtype of $AInterface$ ($A <: AInterface$) and the cast compiles. At run-time, we will be sure that the invoked function has a match in A, and it will be executed.

4. Otherwise, the cast does not compile, since A is not a subtype of $AInterface$.

Even though it is a non-trivial modification in the way of implementing smart contracts, this works in any scenario. By *always* imposing such constraints we can be sure that only acceptable addresses are used as parameters. This reasoning applies also for the implicit variable msg.sender. Suppose A is deployed on the blockchain, and suppose B contains a function with the following signature: uint $f\langle AInterface\rangle$ () $\{\dots\}$. f imposes the constraint on its sender since it aims to use the interface exposed by $AInterface$. The key fact is that $AInterface$ exposes the functions of A that B needs. $AInterface$ is defined together with B and is not deployed on the blockchain together with A. A was, on the other hand, compiled and deployed on its own, without taking into account the future contract potentially interacting with it. B and $AInterface$ are compiled together, so that the compiler can check B's code against $AInterface$. When, later on, an address a is input to B, the interface of the contract it refers to is checked by the compiler as described above. By structural subtyping, if it defines all the function in $AInterface$, we are sure that no reverts will be thrown. On the other hand, if there are any mismatches, *that* invocation does not compile, thus avoiding a run-time revert. This latter point is important: A, B, and $AInterface$ successfully compile. Only the invocation of f from a wrong sender does not.

This behavior is similar to Java RMI [1]. The first step in using RMI is to define remote interfaces (extending `Remote`), which are later on implemented by remote objects. Such objects are afterwards deployed on an RMI registry, waiting for other objects to invoke their methods. Clients connects to the registry and look remote objects up to use them. They know nothing about the actual implementation of the methods they will invoke: they just know their signature. The code has eventually to be available, and it is usually distributed via a `.jar` file deployed somewhere and downloaded by clients. What we propose is just a little different. The actual implementation is not deployed somewhere, but is deployed on the blockchain and is available to every miner. We can assume the interface is publicly available to programmers. This is not too strict, since they have to know the signatures of the functions they want to invoke. The last thing we need is a kind of RMI registry, which, in FS$^+$, is the context Γ used to give a type to programs. The compiler use it to look up any addresses referenced in the source code and check their interfaces against the desired ones.

Note that, for the sake of simplicity, the type system we formalized above does not take into account this kind of structural subtyping. However, it could be introduced by adding the following rule:

$$\frac{\mathsf{contract}\ C\ \mathsf{is}\ C'\ \{\tilde{T\,s};\ K\ \tilde{F_C}\} \qquad \mathsf{contract}\ D\ \mathsf{is}\ D'\ \{\tilde{T\,r};\ K\ \tilde{F_D}\} \quad \forall f \in \tilde{F_D}\ .\ \mathrm{structural\_override}(C, D, f)}{C <: D}\ (\textsc{StructuralSubtyping})$$

Where structural_override is defined as follows:

(STRUCTURALOVERRIDE)

$$\frac{\text{If } \text{ftype}(f, C) = \tilde{T_1} \to T_2 \wedge \text{ftype}(f, D) = \tilde{T_1}' \to T_2' \text{ then } \tilde{T_1}' = \tilde{T_1} \text{ and } T_2' = T_2}{\text{structural\_override}(C, D, f)}$$

7.9 Coexistence of addresses and contract references

To conclude this chapter, it is worth to note that there is no need anymore for Solidity to keep both contract references and addresses. These two constructs are, at the time of writing, both included in Solidity because they represent two sides of the same coin: they both allow programmers to refer to an account instance, but have some subtle differences. As we have seen, addresses are not expressive enough to describe a contract's interface, whereas contract references are not expressive enough to be used to refer to any Ethereum accounts. Our new type, $\mathsf{address}\langle\mathsf{C}\rangle$, merges the good aspects of both, and makes one of them redundant. Given a value a of type $\mathsf{address}\langle\mathsf{C}\rangle$, it now keeps all the necessary information about a contract interface and, at the same time, can be used to refer to a contract instance on the blockchain. Furthermore, when it comes to invoke a function on a, an implicit cast $C(a)$ can be applied. The very same reasoning applies to contract references c. They can be directly used to invoke functions, and an implicit conversion $\mathsf{address}(c)$ might be carried out to obtain the corresponding address, while preserving the information about the interface. Figure 7.3 depicts this relation.

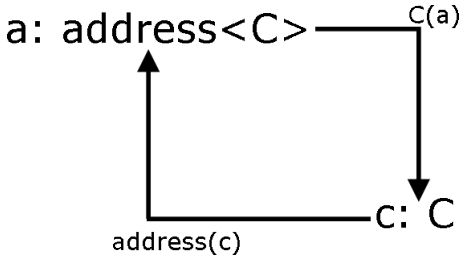

Figure 7.3: Circular relation between addresses and contract references in Solidity

These two ways of referring to an instance have a direct correspondence in how two famous object-oriented programming languages manage heap-allocated objects: C++ and Java. For the former, the `new` operation returns a pointer to the instance, and the programmer can invoke functions by dereferencing it (for example, via the `->` operator). Also the latter contains pointers, but they are carefully taken out of the programmer's hands: `new` returns a contract reference, and the use of pointers to manage the memory is hidden. Both solutions have pros and cons, although the way Java handles references is less error-prone than C++'s one, mainly because it abstracts from machine details programmers are, in general, not interested in. Note we are talking about heap-allocated objects: in fact, C++ also allows for objects to reside on the stack, but this a different situation not handled by Java, where objects always reside on the heap.

In Ethereum and Solidity we could imagine the blockchain as a enhanced type of heap: any contract instance here deployed (allocated) is persistent and has got a unique address (pointer), which allows other instances to communicate with it. The question that arises is: if Solidity used one of the two aforementioned solutions, which one

would be better? Java's way is by far safer, since pointers are known to be harmful and error-prone. On the other hand, C++'s way enables for more flexibility. Furthermore, removing contract references would be more "Ethereum-oriented". Consider a Solidity-like language, but without addresses (i.e. expressions like a : address⟨C⟩). Such a language would lack one of the Solidity's peculiarities, that is the possibility to directly reference accounts only based on their unique identifier (i.e. their address). Hence, no hard-coded addresses might be written, and interacting with EOAs would be infeasible. On the other hand, a version without contract references (i.e. expressions like $c : C$) would be easily usable and would preserve all the functionality of Solidity. This C++-like version would make any new return an address instead of a reference. Actually, no cast would be necessary, since a contract implementation may be retrieved using only its address. Example 7.7 shows the same code as Listing 7.7, but without contract references.

Example 7.7 (Oracle without contract references)**.** Listing 7.11 omits the definition of `Oracle`, `Callbackable`, and `ConcreteOracle`, and shows only the implementation of `Room`.

```
1  contract Room is Callbackable {
2      uint public temperature;
3      address<Oracle> oracle;
4
5      constructor(uint _temperature, address<Oracle> _oracle) public
           {
6          temperature = _temperature;
7          oracle = _oracle;
8      }
9
10     function getTemperature() {
11         oracle.execute("...");
12     }
13
14     function callback(uint _temperature) public {
15         temperature = _temperature;
16     }
17 }
```

Listing 7.11: Room without contract references

There are two main changes. Note the state variable `oracle` has now type `address<Oracle>` instead of just `Oracle` (Line 4). This allows us to remove the cast in Line 7. No other changes are needed in this contract: `getTemperature()` can invoke `execute()` as before, but on an address instead of on a reference.

This modification eliminates redundant casts: addresses are now treated as pointers in C++, but are by far safer. Programmers can reference deployed contracts or EOAs and, at the same time, impose on them the needed constraints. There are no leaks, since nothing can be removed (de-allocated) from the blockchain, and there is nothing like address-arithmetic that makes it difficult to reason about these identifiers. On the other hand, a source code without casts is easier to read and to understand. An explicit type, such as EOC is now mandatory in order to refer to EOAs.

Example 7.8 further shows the simplicity of the resulting code.

Example 7.8 (Blood bank contract with the extended syntax)**.** Here we revise the `BloodBank` and the `Donor` contracts first listed in Example 2.10. The new version is shown in Listing 7.12. This example not only shows the new syntax, but clarifies what we explained in Section 7.8.

```
1  // Deployed on the blockchain at block number n
2
3  interface DonorInterface {
4      function blood() public view returns (uint);
5  }
6
7  contract BloodBank {
8      mapping (address<DonorInterface> => bool) private healty;
9      address public doctor;
10     uint public blood;
11
12     constructor() public {
13         doctor = msg.sender;
14     }
15
16     function setHealth(address<DonorInterface> _donor, bool
           _isHealty) public {
17         require(msg.sender == doctor);
18         healty[_donor] = _isHealty;
19     }
20
21     function donate<DonorInterface>(uint _amount) public returns (
           bool) {
22         uint donorBlood = msg.sender.blood();
23         if (healty[msg.sender] && donorBlood > 3000 && donorBlood
               - _amount > 0) {
24             blood += _amount;
25             return true;
26         }
27         return false;
28     }
29 }
30
31 // Deployed on the blockchain at block number m > n,
32 // indipendently of DonorInterface and BloodBank
33
34 interface BloodBankInterface {
35     function donate<Donor>(uint _amount) public returns (bool);
36 }
37
38 contract Donor {
39     uint private blood;
40     BloodBankInterface public bank;
41
42     constructor(address<BloodBankInterface> _bank) public {
43         blood = 5000;
44         bank = _bank;
45     }
46
47     function donate(uint _amount) public {
48         if (bank.donate(_amount)) {
49             blood -= _amount;
50         }
51     }
52
53     function blood() public view returns (uint) {
54         return blood;
55     }
56 }
```

Listing 7.12: Donor and BloodBank contracts in extended Solidity syntax

At $Block_n$, `BloodBank` is deployed on the blockchain. It defines an interface,

`DonorInterface`, exposing the required functionality for a donor (i.e. the function `blood()`) and then used to impose a type constraint on the mapping `healty`. The expressiveness of the new syntax is shown in `donate()`. This function requires its `sender` to refer to a contract satisfying `DonorInterface`, and calls `blood()` without making any casts. Such invocation will be correct, at run-time, since the constraint will be ensured when compiling the contract invoking `donate()`, of which we know both static type and interface. For this end, consider the contract `Donor` and the interface `BloodBankInterface`. The idea is the same, but note the latter defines only a subset of the functions in `BloodBank` (just `donate()`). Also note that neither `BloodBank` nor `Donor` explicitly inherits from, respectively, `BloodBankInterface` and `DonorInterface`: the subtyping is implicitly handled by the compiler. Again, `Donor` uses its state variable `bank`, of type `address<BloodBankInterface>`, as if it was a contract reference, using it to invoke the function `donate()`. It is in this function that the compiler checks the compliance of `Donor` with `DonorInterface`, and it is also in this function that the compiler makes sure the address `bank` points to a contract instance exposing a function with the required signature. When the address `_bank` in the constructor is first used, the compiler checks the contract it refers to in order to see if it adheres to `BloodBankInterface`. This address will likely point to an instance of `BloodBank`, which has already been deployed and that defines all the functions listed in `BloodBankInterface`. Hence, when `donate()` is later on invoked, the runtime will for sure find a match in the actual contract being invoked.

Example 7.8 clearly shows how simple the new code is. A unique type working as contract reference and identifier, `address<C>`, enables for many simplification, thus making smart contracts more readable. Furthermore, the added expressiveness makes the compiler able to effectively check if addresses refer to contracts with the required functionality, avoiding unexpected and difficult-to-track `reverts` at run-time.

Chapter 8

Conclusions

This thesis proposed a new calculus, called Featherweight Solidity, to allow the formal study of smart contracts. Our specific focus was to model the Solidity language, that comprises non-trivial features and is widely used in practice in the context of the Ethereum blockchain. Our formalization took into account the trickiest parts of this language, and aimed to prove its type safety. Solidity code is not directly run. It is instead compiled down into bytecode, which is afterwards executed by Ethereum's nodes known as miners. Many research works have been targeting the bytecode itself: this makes sense, because Solidity is not stable yet. In fact, many (minor) changes have been introduced since the beginning of this thesis, and much more (major) changes will come. Nonetheless, we think that a simple, yet powerful and expressive, formalization such as FS could be of great help and serve as a basis to explore more and more functionality of both Ethereum and Solidity.

The FS calculus is a small object-oriented-like language with explicit conscience of the blockchain. It has been inspired from two other calculi aiming to formalize, respectively, Java and Java RMI: FJ [25] and DJ [1]. Distribution is not explicitly modeled, even though FS is miner-aware. On the one hand, we omitted the consensus protocol and made no difference between mining and validation nodes. On the other hand we formalized a two-level semantics to execute lists of transactions (what we called top-level expressions) and inner function calls. We strove to make our calculus simple and easy to reason about, slightly simplifying the semantics, which has some small differences with respect to EVM's one. For example, we supposed a unified compilation of smart contracts, which is unreal. Furthermore, we omitted some constructs, such as the primitives `call`, `delegatecall`, `callcode`, and `send` (which are discouraged by the documentation itself), functions and variables visibility, implicit variables (for blocks and transactions), gas, memory and storage, events, and function modifiers. Supposing every function as `public` is not too strict, and generalizing our work in this direction is straightforward. The same applies to the implicit variables `block` and `tx`, referring to, respectively, the current block and the current transaction. We showed the behavior of `msg`, and the same techniques can be used to add the other two variables. Function modifiers find a direct mapping using if expressions and revert, and adding them would provide just a bit of syntactic sugar. Events are similar to function calls, with the difference that they actually write something on the blockchain. Our definition of β does not resemble exactly a blockchain, since it also contains local variables, and does not allow programmers to store events of interest. Furthermore, our β merges the concepts of memory (for local variables) and storage (for state variables, balances, and contracts code). Lastly, gas is, often, just a matter of paying more Ether to have a

function executed, because transaction initiators may decide the amount they are willing to pay beforehand. The only, notable, exception to this affirmation regards fallback functions. When the latter is executed via `transfer` or `send` (not modeled), it does so with a "balance" of 2300 gas. Such an amount is sufficient just for a few operations, and does not allow contracts to write to storage (thus modifying state variables), creating other contracts, and sending Ether. If a fallback function does need to execute with more that such an amount, programmers have to work around the type system using `call`. This is, however, indeed just a workaround and should be used just as a last resort because it breaks type safety.

To avoid a too complex formalization, we supposed Ethereum's externally owned accounts as controlled by code, defining a EOC contract. It may receive Ether and may be used as a sender in top-level function calls. Another big difference is about the separation of concern intrinsic in DApps. In fact, smart contracts are just passive entities waiting for external (e.g. JavaScript) code to call them. This poses a big issue, since to model an entire Ethereum "program" we also need to deal with this additional, external, part. We decided to suppose that the entire program is written in FS. This is the motivation behind the two types of function calls (top-level and non-top-level), and the use of the stack to distinguish between them.

We stated and proved many properties of type theory, such as substitution, permutation, progress, and subject reduction. We formalized a two-fold type safety, for FS configurations and programs. Our aim was to show that no expression (and consequently no transaction) can go stuck (i.e. the virtual machine cannot proceed any further in its evaluation). We succeed, but figured out that Solidity's type system provides poor guarantees and does not rule out many avoidable errors. Although the latter is statically typed, the duality between contract references and addresses arises many ambiguous (and consequently error-prone) situations. In particular, among the others, no static check is done on the fallback function when a `transfer`, as well as a `send`, operation is invoked. Similarly, and as aforementioned, the behavior in presence of explicit casts from "contract address" to "contract instance" is all but sound, with reverts that may arise at any time, making room for bugs extremely difficult to cast out. Even though we know that, so far, no serious problems have arisen from that, we strongly believe that it is clue of a clumsy implementation. It also turned out that, to the best of our knowledge and at the time of writing, no research work has addressed such scenarios. Furthermore, many recent papers have focused on many security flaws with heuristic techniques. Type systems, on the other hand, do not have false positives: a term is either well-typed, meaning that it will behave as expected, or it is not. In this latter case the source code does not even compile, and hence the risk of run-time ambiguities is reduced. However, they do have false negatives: if an expression does not compile, it might not evaluate toward an error.

This is the reason why we went further and formalized a second version of our calculus, which we named FS$^+$, trying to fill the gap between references and addresses. The main idea behind this is that the `address` type does not preserve any information about the account it refers to. Thinking of this in a language like C, addresses are like pointers to `void`: they refer to something, but there is actually no way to be sure of what they point to. This is, of course, undesirable, because allows programmers to work around the type system. We proposed to annotate addresses with the name of the contract they refer to. This way we can always check the interface exposed by a contract and provide additional guarantees. To make the new language more expressive, we also introduced nominal subtyping, omitting, at the same time, many unnecessary features, such as interfaces, abstract contracts, and multiple inheritance.

A minimal form of subtyping describes the core of our idea and provides additional flexibility. For instance, when a cast $D(a)$ is found, it is successfully compiled if and only if a refers to a subcontract of D. However, casts were not our main concern: knowing a contract's interface allows us to check for the presence of a fallback function when a `transfer` is invoked. The main drawback is that any address has to be annotated, including `msg.sender`. Unfortunately, forcing this implicit variable to be of a given type would be too restrictive, because the sender (i.e. the caller of a function) may be any other contract or even an externally owned account. Subtyping helps us out here. We can annotate any function with a minimum required type, so that any, future, sender must be respecting, and possibly specializing, its interface. We discussed for long about the changes this would make on the legacy Solidity code, speaking about the syntactic sugar to make the changes easier to implement. We also reasoned about how to adapt our calculus to the separated compilation of smart contracts. In such a context, the main limitation of our approach is that contracts cannot explicitly inherit from other contracts: the compilation happens in a machine that is not necessarily part of the blockchain. Hence, the compiler would find some unknown names and could not guarantee the well-definition of function overrides. Hence, we formalized structural subtyping, even though the main approach is based on the nominal one, to simplify the compiler's task. FS$^+$ supposes that the interface of every contract being used or referenced by name is known at compile-time. This assumption is only meant to keep the calculus simple, and it would not be necessary in Solidity.

In the context of FS$^+$, we aimed to prove the same properties we formalized for FS. In addition, FS$^+$ provides more guarantees in terms of cast safety and transfer safety. If a contract successfully compiles, programmers are sure that any casts or `transfers` in it will never raise a `revert`. This implies that no money is wasted (i.e. the amount of gas due to the miner) as a consequence of a naive implementation.

8.1 Further work

This thesis is not meant to be a standalone work and opens many different research branches: in this section we outline some possibilities.

8.1.1 Other Solidity features

As aforementioned, many are the Solidity features we did not take into account either in FS or in FS$^+$. On the one hand, some of them, such as function modifiers or visibility and the other implicit variables, do not have a great impact on type safety, and hence are not of interest. On the other hand, some may help in formalizing and enforcing additional properties on Solidity code.

Gas We can think of gas as a kind of fuel to power functions, that can run only if they are given a sufficient amount of gas. We said that providing the latter is only a matter of paying more Ether, but inexperienced programmers could write unoptimized functions wasting money. Since the cost for each low-level operation is known, and tools are available to estimate gas consumption [50], we could enhance the function type with an annotation of the maximum gas allowed, in order to warn, at compile-time, developers if the expected amount exceeds the required one. This would help, among the other things, in detecting those fallbacks exceeding the 2300 threshold.

Interaction with events Events are an important, although not fundamental, feature in Solidity. They can be emitted anywhere in a function body, and will then appear in the transaction log. As we saw, other applications, such as oracles, monitor the blockchain looking for events of interest to react to. Modeling them in FS could be of great help in formalizing such interaction, possibly proving interesting properties. For instance, recalling Example 7.5, let CO and R be instances of, respectively, `ConcreteOracle` and `Room`, and suppose that an external application $oracle$ is associated with `ConcreteOracle` and monitors the blockchain looking for events of "type" `Execute`. A desirable property could be the following:

Whenever R asks for the temperature of the room, it will eventually get the value via a function call to `callback`.

This is likely the invariant for a contract of type `Room`: whenever an instance asks for a temperature it gets it, sooner or later. However, this coarse-grained property silently implies a sequence of calls involving CO: "asking for the temperature" translates to a function call to `ConcreteOracle.execute()`, which in turns emits an event `Execute`. $oracle$, that is continuously monitoring the blockchain, detects the latter, does something to get to know the final value (note that $oracle$ does not even need to know the value is a temperature) and eventually calls `Room.callback()`, providing the value as a parameter. Being able to formally prove that this sequence of operations always happens may be fundamental in a number of scenarios.

Lambda expressions At the time of writing, Solidity (v0.4.24) does not support lambda expressions, although the documentation states their addition is planned. Before adding them to the language core, it could be a possibility to analyze their impact on the existing semantics using FS or FS$^+$, proving they do not break type safety. Currently, FS, as well as Solidity, treats functions as first class values, but they must be defined by a contract; that is, function values are actually pointers to a function defined somewhere in the context of a contract and have the form $c.f$, where c is a contract reference and f is the name of the function. Thus, invoking such function value has the same effect than invoking the function f of c explicitly, since the execution takes into account the state variables of c, if any is referenced. With inline functions this would be a little more complex, due to the need of a closure capturing any variables free in f. Furthermore, lambda expressions could come from the outside (i.e. not from a smart contract), in which case closing the free variables is even more important. Studying the best way to introduce this construct in Solidity on an abstract calculus would avoid many possible flaws.

Memory and storage As said in Section 2.4.2, smart contracts are given two memory areas to store the values they operate on: memory and storage. FS does not explicitly model this distinction, but it could, if we also formalized gas. Even though this does not directly provide additional safety guarantees, it would help in making the type system more expressive and powerful. We could do so relying not only on the type system, but also on other static techniques. For example, a sound formalization of storage may help in ensuring that a contract never messes up another contract's storage, as we saw in Example 2.2.

Such a formalization could also help formalize the garbage collection and the cleaning of the memory area.

8.1.2 Typestate-oriented programming

Typestate-oriented programming, first proposed by Aldrich et al. [3], consists of an extension of the object-oriented programming paradigm. In fact, objects come with a defined behavior, which does not change during their life: in brief, programmers cannot enforce a given protocol. Files are a very familiar example to explain states. Two are the main operations influencing a file's state: `open` and `close`. In an *open* state, one may read, write to or `close` it, but not invoke `open` again. In a *close* state, the file may only be `open`ed. `open` and `close` cause a state transition changing the intended behavior of the file.

Being unable to capture things like this in a programming language is an important limitation. Types are useful because they specify a set of operations allowed on a given value. For example, if the value 5 was given the type of integer numbers, we would know we could safely use it as a parameter of any arithmetical operation. Similarly, if 5 was given a string type, the set of operations would be completely different, with, for instance, concatenation of strings replacing the sum of numbers. The object-oriented paradigm allows programmers to define their own types, specifying the set of safe operations and their intended behavior. However, programmers cannot specify *when* such operations can be invoked.

This *when* dimension is well provided by states. Indeed, states are a widely used abstractions in computer science, engineering and science in general: any system can be described with a state machine and a set of transitions modifying its behavior. Having a states-aware type system would be helpful for enforcing desired protocols on objects: what would be the effect of reading a closed file or re-open an open file?

Ethereum is no different, and it would take benefit of a states-aware type system. In fact, the official documentation of the Solidity language shows an example[1] explaining how to implement a state machine. Furthermore, a blockchain platform named Obsidian [40] has been designed to support typestate-oriented programming by construction [14] [15]. However it is more recent and "immature" than Ethereum: less documentation is available, the language is currently being designed, and less research work has targeted it. For these reasons, migrating to it just for the support to states may be an hazard: we propose to effectively integrate typestate programming in Solidity, making it states-aware.

8.1.3 Block-aided programming

Leveraging the calculus we formalized, a research possibility is about the develop of another, simpler, programming language for Ethereum. In fact, the main problem with Solidity is that defining smart contracts is by far more difficult (and different) than developing "usual" applications. As pointed out by Delmolino et al. [17], programmers have to pay attention to a number of things that have an importance much more limited in the other types of programs. Furthermore, bugs in smart contracts last (almost) forever, and thus any small error could cause terrible issues. Even worse, not every drawback and implementation flaw is known to date, and hence many bugs could (and probably will) arise in the years to come. Indeed, the official documentation contains a section describing common patterns to avoid known issues[2], but there is no guarantee on their use by programmers. This justifies, in part, the number of works on this topic

[1] `https://solidity.readthedocs.io/en/v0.4.24/common-patterns.html#state-machine`

[2] `https://solidity.readthedocs.io/en/v0.4.24/common-patterns.html`

that have been published over the last few months and years. Nonetheless, such a huge amount of work (mostly focused on static analysis of either Solidity or EVM code) creates confusion and will be likely ignored by "real-word" developers.

A feasible direction of future work could be simplifying the task of programming smart contracts, reducing the expressiveness of the language itself. By using techniques of visual programming, the new language could allow the definition of a smart contract by simply combining a bunch of predefined blocks and generate, behind the scenes, the real Solidity code. This way we could formally prove the compliance of the resulting code to any security or design best practice, using FS as an intermediate step.

As we said in Chapter 3, there has already been an attempt of developing a graphical tool to develop smart contracts [33], but, as we pointed out, it lacks any formal foundations or correctness proof. Using FS as a formal basis for such a tool could be an interesting research path.

8.1.4 Mining

At the time of writing, FS does not really formalize the two levels of a DApp. In Chapter 5 we provided two types of semantics, one intended to operate on transactions (i.e. operations coming from the "external world" invoking a contract) and one intended to model the interaction among smart contracts. However, we did not model the fact that the former is not necessarily executed by an Ethereum node, whereas the latter is. A formalization going in this direction could result in a calculus for DApps, which, to the best of our knowledge, nobody has studied yet. Furthermore, we could also include gas to model the choice of the transactions to be included in a block as well as the waiting time a DApp suffers when it waits for its transactions to be processed. Applications with (strict) timing requirements could find this useful to estimate their worst execution time, or study the impact of writing on the blockchain. Lastly, remember that we did not model the difference between *call* and *transaction* (see Example 2.10): with an explicit concept of Ethereum node and miner it could be possible to do so.

8.1.5 Link with object-oriented programming

In Chapter 7 we introduced an important modification in the syntax of FS. In order to control the caller (i.e. the sender), we annotated each function with a contract name. This allowed us to impose a type constraint that, in brief, meant: "if a function f specifies as a minimum type for its sender C, a contract C' is allowed to invoke f if and only if $C' <: C$". This is important in FS (and consequently in Solidity), because functions often need to know something about their caller (e.g. the existence of a fallback function). Nonetheless, this could be quite useful also in object-oriented programming, where, generally, the caller, if needed, is explicitly passed as a parameter. However, having it as a formal parameter might be confusing, and it might not be immediately clear what that parameter actually represents. Thus, a possible direction of future work could be investigating the use of this technique in object-oriented programming. This addition would be two-fold: on the one hand the code could be easier to read. On the other hand, it would simplify the implementation of functions and probably reduce all the errors caused by a wrong value passed as an argument.

8.2 Personal conclusions

At the beginning of this thesis, my research experience was really limited. I had the pleasure and the honor to participate in other projects, but I had never undertaken a work as monumental as this. Once, a Professor of mine said, during a lecture, that researchers should instead be called "finders", because their work is finding things rather than searching for them. This sentence perfectly sums up what I felt over the last months: it might be easy to see the room for improvements and to explore previously unexplored paths, but getting useful and sound results requires many iterations and hours of work to make sure that everything goes as expected. This thesis is no different: it has gone through a number of revisions and has been subject of innumerable modifications.

When formalizing a language as complex (and immature) as Solidity, the abstract calculus needs continuous improvements to best adhere to the reality. When I first begun working on it, I thought I perfectly knew what I had to do and how to do it, but I was wrong. Deeply. I thought that modeling a programming language was only about testing functionality and finding the best way to abstract it. This is only partially true. As common in computer science, many are the ways to deal with and solve a problem: all of them may be correct, but only a few will be a good fit. I experienced it so many times. Almost every time I thought I found a good way I ended up realizing that something in my solution could be ameliorated. The syntax and the semantics (and also the type system, even though to a lower extent) got through many revisions, due to errors, misunderstandings, and changes in Solidity behavior. It took me a long time to clearly see the big picture.

Andrew Wiles, the mathematician who proved the Fermat's last theorem[3], once said:

> "Perhaps I could best describe my experience of doing mathematics in terms of entering a dark mansion. You go into the first room and it's dark, completely dark. You stumble around, bumping into the furniture. Gradually, you learn where each piece of furniture is. And finally, after six months or so, you find the light switch and turn it on. Suddenly, it's all illuminated and you can see exactly where you were. Then you enter the next dark room..."

I felt exactly like this when it came the time to prove theorems on FS. I realized that many things I was totally sure about were wrong. I found myself making changes to the semantics, simplifying the syntax, and question myself about my entire work. I found out that FS couldn't solve all the errors I thought it could. Under the supervision of Professor Silvia Crafa I explored many ways to extend it, finally ending up with FS$^+$. To the best of our knowledge, it is the first calculus trying to enhance Solidity's type system without proposing a new language. We interrogated ourselves many times about the feasibility of our proposal, and strive to make it less invasive as possible.

Since the beginning, our aim wasn't just to formalize a language to prove the type safety of a core part of it. We wanted to achieve a result with practical consequences, something that real-world programmers could and would use everyday, to make their code and their programs better. Programming languages are one of the most powerful tools we can count on nowadays. Computers and automated systems control our world

[3]No three positive integers a, b, and c satisfy the equation $a^n + b^n = c^n$ for any integer value of n greater than 2.

to a great extent. They not only help us at work or at home, but since Bitcoin they also have managed our money. We trust a network of tens of thousands of nodes, without even knowing the position of any of them. We trust software written by people we don't know, we send our money to contracts that may contain the worst of the bugs. This is awesome as long as it works, but when something goes wrong the consequences may be awful. I cited the DAO contract during this work, and I talked a little about the bug behind this tremendous page of Ethereum's story. Ethereum had to hard-fork itself to restore the situation before that bug was exploited, and this set a dangerous precedent. Nowadays we trust software more than people, but we seem to have forgotten that behind a software (from the most trivial script to the most sophisticated program) there are people. People we know nothing about that could write a buggy software for fun, profit, or incompetence.

When the software becomes so critical that it handles our own money, we should make 100% sure about the correctness of every operation it carries out. And since we know nothing about programmers and their competences, it is our duty to provide them with tools capable of detecting potential vulnerabilities; tools that cannot be worked around and that have no false positives or negatives. We have been experiencing so with language-based security: there are no external tools that can be ignored, but it is the language itself that rules out harmful code patterns. David Evans and David Larochelle wrote[4]:

> "So why do developers keep making the same mistakes? Instead of relying on programmers' memories, we should strive to produce tools that codify what is known about common security vulnerabilities and integrate it directly into the development process."

Type systems are a perfect example of such tools, if integrated with a compiler. If a snippet of code does not compile, it cannot be executed and hence can create no harm. However, type systems risk not to be adopted if they are too heavy, or if they are too invasive. This is the reason why we strove to make FS^+ as simple and less invasive as possible. Our goal was to create a tool that can be easily integrated with Solidity, causing no incompatibility with the legacy code, but, at the same time, being simple enough to be fully understood and used. We strongly believe that the modifications proposed in FS^+ could make their way into Solidity, helping programmers in writing safer code. This is the greatest aim of this work.

This thesis has been a long journey, during which I learned many things. Since my first year at University, I had tried to avoid the most theoretical part of computer science, labeling it as not a good fit for me, and for a long time I had preferred just programming. But during the fourth year, the first of my Master program, something changed, and I started getting interested in all the theoretical stuff. I honestly got enthralled by the logics and the mathematics behind the set of keywords we call "programming languages", and I then decided to undertake a thesis on this topic. Now that I am through with it, I don't regret at all my choice. It's been a long and difficult journey, where I often found myself doubting about my work, but also where I learned what being a researcher (or a "finder") looks like. I felt happy whenever I was able to make a little improvement on this thesis, I felt happy whenever I saw a little more clearly something, and I felt enthusiastic whenever I got a sound proof or a confirmation that I was walking on the right direction.

These six months of work taught me an important lesson: our life always challenges us, at work, at the University, at school, any time and anywhere. Every day we face the

[4] "Improving security using extensible lightweight static analysis" [19]

same dilemma: either to keep striving to ameliorate things, to make our job, or other people's job, better, easier, or faster, or to settle for what we already have. The former may be difficult and tiring, but it ensures us we're doing the right thing, that we are on the right path, and when we succeed in getting something better we feel satisfied and ready to start it again. It is exactly like climbing a mountain: we go up and climb to get to the peak, but over there, right after the peak we just conquered, there's something higher to achieve. We just take a minute to rest, and then we start moving again.

This is how I felt while working on this topic. It's been worth it, and the difficulties made me understand so many things that I wouldn't have understood otherwise. For this, and for all the help she gave to me, I pay a debt of immense gratitude to my supervisor, Professor Silvia Crafa, that let me guide this work, make decisions and mistakes in autonomy, but that also never left me alone and helped me whenever I needed it.

Appendix A

Operational semantics rules for FS

This Appendix lists the operational semantics rules of the FS language.

(IF-TRUE)

$$\frac{}{\langle \beta, \sigma, \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \rangle \longrightarrow \langle \beta, \sigma, e_1 \rangle}$$

(IF-FALSE)

$$\frac{}{\langle \beta, \sigma, \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \rangle \longrightarrow \langle \beta, \sigma, e_2 \rangle}$$

(SEQ-C)

$$\frac{\sigma = \beta_0}{\langle \beta, \sigma, v; e \rangle \longrightarrow \langle \beta, \beta, e \rangle}$$

(SEQ-R)

$$\frac{\sigma = \beta_0}{\langle \beta, \sigma, \mathsf{revert}; e \rangle \longrightarrow \langle \beta_0, \sigma, \mathsf{revert} \rangle}$$

(SEQ)

$$\frac{\mathrm{Top}(\sigma) = a}{\langle \beta, \sigma, v; e \rangle \longrightarrow \langle \beta, \sigma, e \rangle}$$

(DECL)

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

(VAR)

$$\frac{}{\langle \beta, \sigma, x \rangle \longrightarrow \langle \beta, \sigma, \beta(x) \rangle}$$

(ASS)

$$\frac{x \in \mathrm{dom}(\beta)}{\langle \beta, \sigma, x = v \rangle \longrightarrow \langle \beta[x \mapsto v], \sigma, v \rangle}$$

(MAPPSEL)

$$\frac{}{\langle \beta, \sigma, M[v_1] \rangle \longrightarrow \langle \beta, \sigma, M(v_1) \rangle}$$

(MAPPASS)

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

(BALANCE)

$$\frac{\beta(a) = (C, s\tilde{:}v, n)}{\langle \beta, \sigma, \mathsf{balance}(a) \rangle \longrightarrow \langle \beta, \sigma, n \rangle}$$

(ADDRESS)

$$\frac{\hat{\beta}(c) = a}{\langle \beta, \sigma, \mathsf{address}(c) \rangle \longrightarrow \langle \beta, \sigma, a \rangle}$$

(NEW-1)

$$\frac{(c,a) \notin \mathrm{dom}(\beta) \qquad \mathrm{sv}(C) = \tilde{T}\,s \qquad |\tilde{v}| = |\tilde{s}| \qquad \mathrm{Top}(\sigma) \neq \emptyset}{\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) \cdot [(c,a) \mapsto (C, s\tilde{:}v, n)], \sigma, c \rangle}$$

(NEW-2)

$$\frac{(c,a) \notin \mathrm{dom}(\beta) \qquad \mathrm{sv}(C) = \tilde{T}\,s \qquad |\tilde{v}| = |\tilde{s}| \qquad \mathrm{Top}(\sigma) = \emptyset}{\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta \cdot [(c,a) \mapsto (C, s\tilde{:}v, n)], \sigma, c \rangle}$$

(NEW-R)

$$\frac{\mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) = \bot \qquad \mathrm{Top}(\sigma) \neq \emptyset}{\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle}$$

(CONTRRETR)

$$\frac{\beta^C(a) = C \qquad \hat{\beta}(a) = c}{\langle \beta, \sigma, C(a) \rangle \longrightarrow \langle \beta, \sigma, c \rangle}$$

(CONTRRETR-R)

$$\frac{\beta^C(a) = C' \qquad C' \neq C}{\langle \beta, \sigma, C(a) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle}$$

(STATESEL)

$$\frac{\beta(c) = (C, s\tilde{:}v, n) \qquad s \in \tilde{s}}{\langle \beta, \sigma, c.s \rangle \longrightarrow \langle \beta, \sigma, v \rangle}$$

(STATEASS)

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

(TRANSFER)

$$\frac{\begin{array}{c}\beta^C(a) = C \qquad \mathrm{fbody}(C, fb, \{\}) = (\{\}, e) \\ \beta' = \mathrm{uptbal}(\mathrm{uptbal}(\beta, a, n), \mathrm{Top}(\sigma), -n)\end{array}}{\langle \beta, \sigma, a.\mathsf{transfer}(n) \rangle \longrightarrow \langle \beta', \sigma \cdot a, e\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\} \rangle}$$

(CALL)

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

(CALLTOPLEVEL)

$$\frac{\begin{array}{l}\hat{\beta}(c) = a \qquad \beta^C(c) = C \qquad \mathrm{fbody}(C, f, \tilde{v}) = (\tilde{x}, e) \qquad \tilde{x} \notin \mathrm{dom}(\beta) \\ \beta' = \mathrm{uptbal}(\mathrm{uptbal}(\beta_w, a, n), a', -n) \cdot [\tilde{x} \mapsto \tilde{v}] \qquad \mathrm{Top}(\sigma) = \emptyset \\ e_s = e\{\mathsf{this} := c, \mathsf{msg.sender} := a', \mathsf{msg.value} := n\}\end{array}}{\langle \beta, \sigma, c.f.\mathsf{value}(n).\mathsf{sender}(a')(\tilde{v}); e' \rangle \longrightarrow \langle \beta', \sigma \cdot a, e_s; e' \rangle}$$

(TRANSFER-R)
$$\frac{\mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) = \bot}{\langle \beta, \sigma, a.\mathsf{transfer}(n) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle}$$

(CALL-R)
$$\frac{\mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) = \bot}{\langle \beta, \sigma, c.f.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle}$$

(CALLTOPLEVEL-R)
$$\frac{\mathrm{uptbal}(\beta, a', -n) = \bot \qquad \mathrm{Top}(\sigma) = \emptyset}{\langle \beta, \sigma, c.f.\mathsf{value}(n).\mathsf{sender}(a')(\tilde{v}); e' \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert}; e' \rangle}$$

(RETURN)
$$\frac{}{\langle \beta, \sigma \cdot a, \mathsf{return}\ v \rangle \longrightarrow \langle \beta, \sigma, v \rangle}$$

(RETURN-R)
$$\frac{}{\langle \beta, \sigma \cdot a, \mathsf{return\ revert} \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle}$$

(CONG)
$$\frac{\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle}{\langle \beta, \sigma, E[e] \rangle \longrightarrow \langle \beta', \sigma', E[e'] \rangle}$$

(REVERT)
$$\frac{}{\langle \beta, \sigma, E[\mathsf{revert}] \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle}$$

Appendix B

Type system rules for FS

This Appendix lists the type system rules of the FS language.

(EMPTYENVIRONMENT)
$$\frac{}{\emptyset \vdash \langle\rangle}$$

(VARIABLEENVIRONMENT)
$$\frac{\Gamma \vdash \langle\rangle \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : T \vdash \langle\rangle}$$

(ADDRESSENVIRONMENT)
$$\frac{\Gamma \vdash \langle\rangle \qquad a \notin \mathrm{dom}(\Gamma)}{\Gamma, a : \mathsf{address} \vdash \langle\rangle}$$

(CONTRACTENVIRONMENT)
$$\frac{\Gamma \vdash \langle\rangle \qquad c \notin \mathrm{dom}(\Gamma)}{\Gamma, c : C \vdash \langle\rangle}$$

(F OK IN C)
$$\frac{\mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}, \mathsf{msg.value} : \mathsf{uint}, \tilde{x} : \tilde{T_1} \vdash e : T_2}{T_2\ f\ (\tilde{T_1 x})\ \{\mathsf{return}\ e\}\ OK\ in\ C}$$

(C OK)
$$\frac{K = C\ (\tilde{Tx})\ \{\mathsf{this}.\tilde{s}\ =\ \tilde{x}\} \qquad \tilde{F}\ \mathsf{OK\ in}\ C}{\mathsf{contract}\ C\ \{\tilde{Ts};\ K\ \tilde{F}\}\ OK}$$

(CALLSTACK)
$$\frac{\Gamma \vdash \sigma \qquad \Gamma \vdash a : \mathsf{address}}{\Gamma \vdash \sigma \cdot a}$$

(EMPTYBLOCKCHAIN)
$$\frac{}{\Gamma \vdash \emptyset}$$

(VARIABLE)
$$\frac{\Gamma \vdash \beta \qquad x \notin \mathrm{dom}(\beta) \qquad \Gamma \vdash x : T \qquad \Gamma \vdash v : T}{\Gamma \vdash \beta \cdot [x \mapsto v]}$$

(CONTRACT)
$$\frac{\Gamma \vdash \beta \qquad (c,a) \notin \mathrm{dom}(\beta) \qquad \Gamma \vdash c : C \qquad \Gamma \vdash a : \mathsf{address} \qquad \mathrm{sv}(C) = \tilde{Ts} \qquad \Gamma \vdash \tilde{v} : \tilde{T} \qquad \Gamma \vdash n : \mathsf{uint} \qquad |\tilde{s}| = |\tilde{v}|}{\Gamma \vdash \beta \cdot [(c,a) \mapsto (C, \tilde{s:v}, n)]}$$

(CONFIGURATION)
$$\frac{\Gamma \vdash \beta \qquad \Gamma \vdash \sigma \qquad \Gamma \vdash e : T}{\Gamma \vdash \langle \beta, \sigma, e \rangle : T}$$

(PROGRAM)
$$\frac{C\ OK\ \forall\, C \in CT \qquad \Gamma \vdash \langle \beta, \beta, e \rangle : T}{\Gamma \vdash (CT, \beta, e) : T}$$

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

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

(TRUE)
$$\frac{\Gamma \vdash \langle\rangle}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}$$

(FALSE)
$$\frac{\Gamma \vdash \langle\rangle}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

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

UNIT
$$\frac{\Gamma \vdash \langle\rangle}{\Gamma \vdash \mathsf{u} : \mathsf{unit}}$$

(FUN)
$$\frac{\Gamma \vdash c : C \qquad \mathrm{ftype}(C, f) = \tilde{T_1} \to T_2}{\Gamma \vdash c.f : \tilde{T_1} \to T_2}$$

(NAT)
$$\frac{n \in \mathbb{N}^+ \qquad \Gamma \vdash \langle\rangle}{\Gamma \vdash n : \mathsf{uint}}$$

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

(REVERT)
$$\frac{\Gamma \vdash \langle\rangle}{\Gamma \vdash \mathsf{revert} : T}$$

(BAL)
$$\frac{\Gamma \vdash e : \mathsf{address}}{\Gamma \vdash \mathsf{balance}(e) : \mathsf{uint}}$$

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

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

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

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

(MAPPSEL)
$$\frac{\Gamma \vdash e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2) \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1[e_2] : T_2}$$

(STATESEL)
$$\frac{\Gamma \vdash e : C \quad \mathrm{sv}(C) = \tilde{T s} \quad s_i \in \tilde{s}}{\Gamma \vdash e.s_i : T_i}$$

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

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

(MAPPASS)
$$\frac{\Gamma \vdash e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2) \qquad \Gamma \vdash e_2 : T_1 \qquad \Gamma \vdash e_3 : T_2}{\Gamma \vdash e_1[e_2 \to e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)}$$

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

(NEW)
$$\frac{\mathrm{sv}(C) = \tilde{T}\,s \quad \Gamma \vdash \tilde{e} : \tilde{T} \quad |\tilde{e}| = |\tilde{s}| \quad \Gamma \vdash e' : \mathsf{uint}}{\Gamma \vdash \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : C}$$

(CONTRRETR)
$$\frac{\Gamma \vdash e : \mathsf{address} \qquad \mathit{warning}}{\Gamma \vdash C(e) : C}$$

(TRANSFER)
$$\frac{\Gamma \vdash e_1 : \mathsf{address} \qquad \Gamma \vdash e_2 : \mathsf{uint}}{\Gamma \vdash e_1.\mathsf{transfer}(e_2) : \mathsf{unit}}$$

(CALL)
$$\frac{\Gamma \vdash e_1 : C \quad \Gamma \vdash e_2 : \mathsf{uint} \quad \mathrm{ftype}(C, f) = \tilde{T_1} \to T_2 \quad \Gamma \vdash \tilde{e} : \tilde{T_1} \quad |\tilde{e}| = |\tilde{T_1}|}{\Gamma \vdash e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2}$$

(CALLTOPLEVEL)
$$\frac{\Gamma \vdash e_3 : \mathsf{address} \qquad \Gamma \vdash e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2}{\Gamma \vdash e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T_2}$$

(CALLVALUE)
$$\frac{\Gamma \vdash e_1 : \tilde{T_1} \to T_2 \qquad \Gamma \vdash e_2 : \mathsf{uint} \qquad \Gamma \vdash \tilde{e} : \tilde{T_1} \qquad |\tilde{e}| = |\tilde{T_1}|}{\Gamma \vdash e_1.\mathsf{value}(e_2)(\tilde{e}) : T_2}$$

Appendix C

Type system rules for FS$^+$

This Appendix lists the type system rules of the FS$^+$ language.

(TOP)
$$\frac{}{C <: \mathsf{Top}}$$

(CONTRACT)
$$\frac{\mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\}}{C <: D}$$

(REFLEXIVITY)
$$\frac{}{C <: C}$$

(TRANSITIVITY)
$$\frac{C <: D \qquad D <: E}{C <: E}$$

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

(STRUCTURAL FALLBACK - 1)
$$\frac{CT(C) = \mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \qquad \mathsf{unit}\ fb\ ()\ \{\mathsf{return}\ e\} \in \tilde{F}}{C <: \mathsf{Top_{fb}}}$$

(STRUCTURAL FALLBACK - 2)
$$\frac{CT(C) = \mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\,s;\ K\ \tilde{F}\} \qquad D <: \mathsf{Top_{fb}}}{C <: \mathsf{Top_{fb}}}$$

(EMPTYENVIRONMENT)
$$\frac{}{\emptyset \vdash \langle\rangle}$$

(VARIABLEENVIRONMENT)
$$\frac{\Gamma \vdash \langle\rangle \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : T \vdash \langle\rangle}$$

(ADDRESSENVIRONMENT)
$$\frac{\Gamma \vdash \langle\rangle \qquad a \notin \mathrm{dom}(\Gamma)}{\Gamma, a : \mathsf{address} \vdash \langle\rangle}$$

(CONTRACTENVIRONMENT)
$$\frac{\Gamma \vdash \langle\rangle \qquad c \notin \mathrm{dom}(\Gamma)}{\Gamma, c : C \vdash \langle\rangle}$$

$$\frac{\mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}\langle \mathsf{C}' \rangle, \mathsf{msg.value} : \mathsf{uint}, \tilde{x} : \tilde{T_1} \vdash e : T_2' \qquad T_2' <: T_2 \qquad CT(C) = \mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T}\, s;\ K\ \tilde{F}\} \qquad \mathrm{override}(f, D, \tilde{T_1} \to T_2)}{T_2\ f\langle C' \rangle\ (\tilde{T_1}\, x)\ \{\mathsf{return}\ e\}\ OK\ in\ C}$$

$$\frac{K = C\ (\tilde{T_1}\, y, \tilde{T_2}\, x)\ \{\mathsf{super}(\tilde{y}); \mathsf{this}.\tilde{s}\ =\ \tilde{x}\} \qquad \mathrm{sv}(D) = \tilde{T_1}\, r \qquad |\tilde{r}| = |\tilde{y}| \qquad \tilde{F}\ \mathbf{OK}\ \mathbf{in}\ C}{\mathsf{contract}\ C\ \mathsf{is}\ D\ \{\tilde{T_2}\, s;\ K\ \tilde{F}\}\ OK}$$

(CALLSTACK)
$$\frac{\Gamma \vdash \sigma \qquad \Gamma \vdash a : \mathsf{address}\langle \mathsf{C} \rangle}{\Gamma \vdash \sigma \cdot a}$$

(EMPTYBLOCKCHAIN)
$$\frac{}{\Gamma \vdash \emptyset}$$

(VARIABLE)
$$\frac{\Gamma \vdash \beta \qquad x \notin \mathrm{dom}(\beta) \qquad \Gamma \vdash x : T \qquad \Gamma \vdash v : T' \qquad T' <: T}{\Gamma \vdash \beta \cdot [x \mapsto v]}$$

(CONTRACT)
$$\frac{\Gamma \vdash \beta \qquad (c, a) \notin \mathrm{dom}(\beta) \qquad \Gamma \vdash c : C \qquad \Gamma \vdash a : \mathsf{address}\langle \mathsf{C} \rangle \qquad \mathrm{sv}(C) = \tilde{T}\, s \qquad \Gamma \vdash \tilde{v} : \tilde{T'} \qquad \tilde{T'} <: \tilde{T} \qquad \Gamma \vdash n : \mathsf{uint} \qquad |\tilde{s}| = |\tilde{v}|}{\Gamma \vdash \beta \cdot [(c, a) \mapsto (C, \tilde{s{:}v}, n)]}$$

(CONFIGURATION)
$$\frac{\Gamma \vdash \beta \qquad \Gamma \vdash \sigma \qquad \Gamma \vdash e : T' \qquad T' <: T}{\Gamma \vdash \langle \beta, \sigma, e \rangle : T}$$

(PROGRAM)
$$\frac{C\ OK\ \forall\, C \in CT \qquad \Gamma \vdash \langle \beta, \beta, e \rangle : T}{\Gamma \vdash (CT, \beta, e) : T}$$

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

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

(TRUE)
$$\frac{\Gamma \vdash \langle\rangle}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}$$

(FALSE)
$$\frac{\Gamma \vdash \langle\rangle}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

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

UNIT
$$\frac{\Gamma \vdash \langle\rangle}{\Gamma \vdash \mathsf{u} : \mathsf{unit}}$$

(FUN)
$$\frac{\Gamma \vdash c : C \qquad \mathrm{ftype}(C, f) = \tilde{T_1} \to T_2 \qquad \Gamma \vdash \mathsf{this} : C' \qquad C' <: \mathrm{fsender}(C, f)}{\Gamma \vdash c.f : \tilde{T_1} \to T_2}$$

(NAT)
$$\frac{n \in \mathbb{N}^+ \qquad \Gamma \vdash \langle\rangle}{\Gamma \vdash n : \mathsf{uint}}$$

(REVERT)
$$\frac{\Gamma \vdash \langle\rangle}{\Gamma \vdash \mathsf{revert} : T}$$

(MAPPING)
$$\frac{M = \{(\tilde{k, v})\} \qquad \Gamma \vdash \tilde{k} : \tilde{T_1'} \qquad \Gamma \vdash \tilde{v} : \tilde{T_2'} \qquad \tilde{T_1'} <: T_1 \qquad \tilde{T_2'} <: T_2}{\Gamma \vdash M : \mathsf{mapping}(T_1 \Rightarrow T_2)}$$

(BAL)
$$\frac{\Gamma \vdash e : \mathsf{address}\langle \mathsf{C} \rangle}{\Gamma \vdash \mathsf{balance}(e) : \mathsf{uint}}$$

(ADDR)
$$\frac{\Gamma \vdash e : C}{\Gamma \vdash \mathsf{address}(e) : \mathsf{address}\langle \mathsf{C} \rangle}$$

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

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

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

(MAPPSEL)
$$\frac{\Gamma \vdash e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2) \qquad \Gamma \vdash e_2 : T_1' \qquad T_1' <: T_1}{\Gamma \vdash e_1[e_2] : T_2}$$

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

(ASS)
$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T' \qquad T' <: T}{\Gamma \vdash x = e : T'}$$

(STATESEL)
$$\frac{\Gamma \vdash e : C \qquad \mathrm{sv}(C) = \tilde{T}\, s \qquad s_i \in \tilde{s}}{\Gamma \vdash e.s_i : T_i}$$

(MAPPASS)
$$\frac{\Gamma \vdash e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2) \quad \Gamma \vdash e_2 : T_1' \quad \Gamma \vdash e_3 : T_2' \quad T_1' <: T_1 \quad T_2' <: T_2}{\Gamma \vdash e_1[e_2 \to e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)}$$

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

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

(NEW)
$$\frac{\mathrm{sv}(C) = \tilde{T}\, s \qquad \Gamma \vdash \tilde{e} : \tilde{T}' \qquad \tilde{T}' <: \tilde{T} \qquad |\tilde{e}| = |\tilde{s}| \qquad \Gamma \vdash e' : \mathsf{uint}}{\Gamma \vdash \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : C}$$

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

(CALL)
$$\frac{\begin{array}{l}\Gamma \vdash e_1 : C \qquad \Gamma \vdash e_2 : \mathsf{uint} \qquad \mathrm{ftype}(C, f) = \tilde{T}_1 \to T_2 \\ \Gamma \vdash \tilde{e} : \tilde{T}_1' \qquad \tilde{T}_1' <: \tilde{T}_1 \qquad |\tilde{e}| = |\tilde{T}_1| \qquad \Gamma \vdash \mathsf{this} : C' \qquad C' <: \mathrm{fsender}(C, f)\end{array}}{\Gamma \vdash e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2}$$

(CALLTOPLEVEL)
$$\frac{\begin{array}{l}\Gamma \vdash e_1 : C \qquad \Gamma \vdash e_2 : \mathsf{uint} \qquad \mathrm{ftype}(C, f) = \tilde{T}_1 \to T_2 \qquad C' <: \mathrm{fsender}(C, f) \\ \Gamma \vdash \tilde{e} : \tilde{T}_1' \qquad \tilde{T}_1' <: \tilde{T}_1 \qquad |\tilde{e}| = |\tilde{T}_1| \qquad \Gamma \vdash e_3 : \mathsf{address}\langle \mathsf{C}' \rangle\end{array}}{\Gamma \vdash e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T_2}$$

(CALLVALUE)
$$\frac{\Gamma \vdash e_1 : \tilde{T}_1 \to T_2 \qquad \Gamma \vdash e_2 : \mathsf{uint} \qquad \Gamma \vdash \tilde{e} : \tilde{T}_1' \qquad \tilde{T}_1' <: \tilde{T}_1 \qquad |\tilde{e}| = |\tilde{T}_1|}{\Gamma \vdash e_1.\mathsf{value}(e_2)(\tilde{e}) : T_2}$$

Appendix D

Additional Solidity examples

D.1 Cryptocurrency

Listing D.1 list the code of an interface, `Currency`, and an implementation `BasicCurrency`.

```
1  interface Currency {
2      function deposit() external payable;
3      function convert(uint) external view returns (uint);
4      function register() external;
5      function transfer(address, uint) external;
6      function getBalance() external view returns (uint);
7  }
8
9  contract BasicCurrency {
10
11     struct Account {
12         uint balance;
13         bool exists;
14     }
15
16     uint private change;
17
18     mapping (address => Account) private accounts;
19
20     event NewAccount(address _holder);
21     event NewDeposit(address _holder, uint _amount);
22     event NewTransfer(address _from, address _to, uint amount);
23
24     constructor(uint _change) public {
25         change = _change;
26     }
27
28     modifier notZero(uint _n) {
29         require(_n > 0);
30         _;
31     }
32
33     function deposit() external payable notZero(msg.value) {
34         require(accounts[msg.sender].exists);
35
36         uint amount = change * msg.value;
37         accounts[msg.sender].balance += amount;
38         emit NewDeposit(msg.sender, amount);
39     }
40
```

```
41      function convert(uint _amount) external view returns (uint) {
42          return change * _amount;
43      }
44
45      function register() external {
46          require(!accounts[msg.sender].exists);
47
48          accounts[msg.sender].exists = true;
49          emit NewAccount(msg.sender);
50      }
51
52      function transfer(address _to, uint _amount) external notZero(
            _amount) {
53          require(accounts[msg.sender].balance > _amount);
54
55          accounts[msg.sender].balance -= _amount;
56          accounts[_to].balance += _amount;
57          emit NewTransfer(msg.sender, _to, _amount);
58      }
59
60      function getBalance() external view returns (uint) {
61          return accounts[msg.sender].balance;
62      }
63  }
```

Listing D.1: Definition of a cryptocurrency

D.2 Plane tickets

Listing D.2 implements a contract modeling the seats on a plane. Passengers can buy tickets and, possibly, ask for a reimbursement. `Plane` defines various modifiers that are applied to its methods. To avoid the reentrancy problem that afflicted the DAO contract [47], `Plane` also implements the withdrawal pattern and uses the operator `delete` to make tickets available again in a simple way.

```
1   contract Plane {
2
3       struct Ticket {
4           uint8 seat;
5           bool bought;
6       }
7
8       uint8 constant private numtickets = 5; // number of seats
9       uint8 constant private ticketprice = 50; // 50 wei
10      address public company;
11      Ticket[numtickets] public tickets;
12
13      mapping (address => uint) private pendingWithdrawals; //
            withdrawal pattern
14
15      mapping (address => Ticket) private boughtTickets;
16
17      // to iterate over the bought tickets
18      address[numtickets] private passengers;
19
20      uint8 private nextAvailableTicket;
21
22      modifier stillSpot() {
23          require(nextAvailableTicket < numtickets);
24          _;
```

```
25      }
26
27      modifier enoughWei {
28          require(msg.value >= ticketprice);
29          pendingWithdrawals[msg.sender] += msg.value - ticketprice;
30          _;
31      }
32
33      modifier hasTicket {
34          require(boughtTickets[msg.sender].bought);
35          _;
36      }
37
38      modifier noOtherTickets {
39          require(!boughtTickets[msg.sender].bought);
40          _;
41      }
42
43      modifier companyOnly {
44          require(msg.sender == company);
45          _;
46      }
47
48      constructor() public {
49          company = msg.sender;
50          for(uint8 i = 0; i < numtickets; i++) {
51              tickets[i].seat = i;
52          }
53      }
54
55      function buyTicket() public payable stillSpot enoughWei
            noOtherTickets {
56          tickets[nextAvailableTicket].bought = true;
57          boughtTickets[msg.sender] = tickets[nextAvailableTicket];
58          passengers[nextAvailableTicket] = msg.sender;
59          nextAvailableTicket = getNextAvailableTicket();
60      }
61
62      function getNextAvailableTicket() private view returns (uint8)
            {
63          uint8 i = 0;
64          while (i < numtickets && tickets[i].bought) {i++;}
65          return i;
66      }
67
68      function askReimbursment() public hasTicket {
69          pendingWithdrawals[msg.sender] += ticketprice/2;
70          tickets[boughtTickets[msg.sender].seat].bought = false;
71          nextAvailableTicket = boughtTickets[msg.sender].seat;
72          delete boughtTickets[msg.sender];
73          delete passengers[boughtTickets[msg.sender].seat];
74      }
75
76      function withdraw() public {
77          uint amount = pendingWithdrawals[msg.sender];
78          pendingWithdrawals[msg.sender] = 0;
79          msg.sender.transfer(amount);
80      }
81
82      function getPassengers() public view returns(address[
            numtickets]) {
83          return passengers;
84      }
```

```
85
86      function takeOff() public companyOnly {
87          for (uint8 i = 0; i < numtickets; i++) {
88              tickets[i].bought = false;
89              delete boughtTickets[passengers[i]];
90          }
91          delete passengers;
92      }
93  }
```

Listing D.2: Solidity contract for plane tickets

Appendix E

Proving the properties of the type system

In this section we prove Lemmas of Permutation (Lemma 3), Weakening (Lemma 4), and Substitution (Lemma 5), and Theorems of Progress (Theorem 3), Subject Reduction (Theorem 4), and Safety for configurations (Theorem 1) and programs (Theorem 2).

In the proof that follows, we shall write $\Gamma \vdash_k e : T$ and $\langle \beta, \sigma, e \rangle \longrightarrow^k \langle \beta', \sigma', e' \rangle$ to indicate that the derivations of those judgments have height at most k.

E.1 Permutation Lemma

We recall Lemma 3:

Lemma 3 (Permutation)**.**
If $\Gamma \vdash e : T$ can be derived and Δ is a permutation of Γ, then the judgment $\Delta \vdash e : T$ can be derived and the derivation has the same height of the previous one.

Proof. We prove this lemma by induction on the height of the derivation of the judgment $\Gamma \vdash e : T$.

Base cases These cases correspond to the axioms in Section 6.3. The height of these derivations is always 1.

- REF. The judgment is $\Gamma \vdash c : C$. Since it can be derived, by Case 9 of Lemma 2 we know that $c : C \in \Gamma$. Let Δ be a permutation of Γ. Hence, Δ has exactly the same elements as Γ, but in a different order. This means that $c : C \in \Delta$, and that $\Delta \vdash c : C$ can be derived.

- VAR. This case is very similar to REF: the judgment is $\Gamma \vdash x : T$. Since it can be derived, by Case 10 of Lemma 2 we know that $x : T \in \Gamma$. Let Δ be a permutation of Γ. Hence, Δ has exactly the same elements as Γ, but in a different order. This means that $x : T \in \Delta$, and that $\Delta \vdash x : T$ can be derived.

- TRUE. The judgment is $\Gamma \vdash \mathsf{true} : \mathsf{bool}$. Since it can be derived regardless of Γ, using a permutation Δ of Γ instead of Γ does not change anything. Hence, the judgment $\Delta \vdash \mathsf{true} : \mathsf{bool}$ can be derived.

- FALSE. This case is very similar to TRUE: the judgment is $\Gamma \vdash \mathsf{false} : \mathsf{bool}$. Since it can be derived regardless of Γ, using a permutation Δ of Γ instead of Γ does not change anything. Hence, the judgment $\Delta \vdash \mathsf{false} : \mathsf{bool}$ is derivable.

- NAT. This case is very similar to TRUE: the judgment is $\Gamma \vdash n : \mathsf{uint}$. The only premise is $n \in \mathbb{N}^+$, and the judgment can be derived regardless of Γ. Hence, using a permutation Δ of Γ instead of Γ does not change anything: the judgment $\Delta \vdash n : \mathsf{uint}$ can be derived.

- ADDRESS. This case is very similar to REF: the judgment is $\Gamma \vdash a : \mathsf{address}$. Since it can be derived, by Case 6 of Lemma 2 we know $a : \mathsf{address} \in \Gamma$. Let Δ be a permutation of Γ. Hence, Δ has exactly the same elements as Γ, but in a different order. This means that $a : \mathsf{address} \in \Delta$, and that $\Delta \vdash a : \mathsf{address}$ can be derived.

- UNIT. The judgment is $\Gamma \vdash \mathsf{u} : \mathsf{unit}$. Since it can be derived regardless of Γ, using a permutation Δ of Γ instead of Γ does not change anything. Hence, the judgment $\Delta \vdash \mathsf{u} : \mathsf{unit}$.

- REVERT. The judgment is $\Gamma \vdash \mathsf{revert} : T$. Again, it is an axiom and can be derived regardless of Γ. Hence, using a permutation Δ of Γ instead of Γ does not change anything: the judgment $\Delta \vdash \mathsf{revert} : T$ can be derived.

Inductive cases Given a judgment J such that its derivation has height $k + 1$, we prove the inductive cases on the last rule used to derive J. We assume the lemma for the judgments with height at most k and we prove it for those with height $k + 1$.

- FUN. In this case $J = \Gamma \vdash_{k+1} c.f : \tilde{T_1} \to T_2$ was derived from the judgment $\Gamma \vdash_k c : C$, and the additional premise $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$. By inductive hypothesis, $\Delta \vdash_k c : C$, where Δ is a permutation of Γ, can be derived. Considering the judgment $\Delta \vdash_{k+1} c.f : \tilde{T_1} \to T_2$, we note that it can be derived from $\Delta \vdash_k c : C$, since the premise $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$ is still valid.

- MAPPING. In this case $J = \Gamma \vdash_{k+1} M : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from the judgments $\Gamma \vdash_k \tilde{k} : \tilde{T_1}$ and $\Gamma \vdash_k \tilde{v} : \tilde{T_2}$. By inductive hypothesis, also the judgments $\Delta \vdash_k \tilde{k} : \tilde{T_1}$ and $\Delta \vdash_k \tilde{v} : \tilde{T_2}$, where Δ is a permutation of Γ, are derivable. Hence, by applying MAPPING to the latter two judgments, $\Delta \vdash_{k+1} M : \mathsf{mapping}(T_1 \Rightarrow T_2)$.

- BAL. In this case $J = \Gamma \vdash_{k+1} \mathsf{balance}(e) : \mathsf{uint}$ was derived from $\Gamma \vdash_k e : \mathsf{address}$. By inductive hypothesis, also the judgment $\Delta \vdash_k e : \mathsf{address}$, where Δ is a permutation of Γ. Hence, by applying BAL to it we obtain $\Delta \vdash_{k+1} \mathsf{balance}(e) : \mathsf{uint}$.

- ADDR. This case is very similar to BAL: $J = \Gamma \vdash_{k+1} \mathsf{address}(e) : \mathsf{address}$ was derived from $\Gamma \vdash_k e : C$. By inductive hypothesis, also the judgment $\Delta \vdash_k e : C$, where Δ is a permutation of Γ, is derivable. Hence, by applying ADDR to it we obtain $\Delta \vdash_{k+1} \mathsf{address}(e) : \mathsf{address}$.

- RETURN. In this case: $J = \Gamma \vdash_{k+1} \mathsf{return}\, e : T$ was derived from $\Gamma \vdash_k e : T$. By inductive hypothesis, also the judgment $\Delta \vdash_k e : T$, where Δ is a permutation of Γ, is derivable. Hence, by applying RETURN to it we obtain $\Delta \vdash_{k+1} \mathsf{return}\, e : T$.

- IF. In this case $J = \Gamma \vdash_{k+1} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ was derived from $\Gamma \vdash_k e_1 : \mathsf{bool}$, $\Gamma \vdash_k e_2 : T$, and $\Gamma \vdash_k e_3 : T$. By inductive hypothesis, given a permutation Δ of Γ, the judgments $\Delta \vdash_k e_1 : \mathsf{bool}$, $\Delta \vdash_k e_2 : T$, and $\Delta \vdash_k e_3 : T$ can be derived. By applying IF to them we obtain $\Delta \vdash_{k+1} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_2 : T$.

- SEQ. This case is very similar to IF: $J = \Gamma \vdash_{k+1} e_1; e_2 : T$ was derived from $\Gamma \vdash_k e_1 : T$ and $\Gamma \vdash_k e_2 : T$. By inductive hypothesis, given a permutation Δ of Γ, the judgments $\Delta \vdash_k e_1 : T$ and $\Delta \vdash_k e_2 : T$ can be derived. By applying SEQ to them we obtain $\Delta \vdash_{k+1} e_1; e_2 : T$.

- DECL. In this case $J = \Gamma \vdash_{k+1} T_1\ x = e_1; e_2 : T_2$ was derived from $\Gamma \vdash_k e_1 : T_1$ and $\Gamma, x : T_1 \vdash_k e_2 : T_2$. The derivations of the latter judgments have height k. Let $\Gamma' = \Gamma, x : T_1$, Δ be a permutation of Γ, and Δ' be a permutation of Γ'; by inductive hypothesis we can derive $\Delta \vdash_k e_1 : T_1$. Nonetheless, we cannot directly apply the inductive hypothesis on $\Gamma' \vdash_k e_2 : T_2$, since we have $\Gamma' \neq \Gamma$. Still, we know that there exists a derivation, of height k, for this judgment. (Otherwise J would not be derivable, but this is a contradiction, since we assumed J has a valid derivation of height $k + 1$.) Hence, it comes from another judgment J' having a derivation of height $k - 1$ where any of the rules defined in Section 6.3 was applied as a last step. We can now apply the inductive hypothesis on J' considering Δ' as a permutation, concluding that $\Delta' \vdash_k e_2 : T_2$ is derivable. Lastly, applying DECL to $\Delta \vdash_k e_1 : T_1$ and $\Delta' \vdash_k e_2 : T_2$ we obtain $\Delta \vdash_{k+1} T_1\ x = e_1; e_2 : T_2$.

- MAPPSEL. In this case $J = \Gamma \vdash_{k+1} e_1[e_2] : T_2$ was derived from $\Gamma \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Gamma \vdash_k e_2 : T_1$. By inductive hypothesis, the judgments $\Delta \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Delta \vdash_k e_2 : T_1$ can be derived. Hence, applying MAPPSEL the them we obtain $\Delta \vdash_{k+1} e_1[e_2] : T_2$.

- STATESEL In this case $J = \Gamma \vdash_{k+1} e.s_i : T_i$ was derived from $\Gamma \vdash_k e : C$, with the additional premise stating $s_i \in \tilde{s}$, where $\mathrm{sv}(C) = \tilde{T}\,\tilde{s}$. Provided that the height of $\Gamma \vdash e : C$ is k, by inductive hypothesis, given a permutation Δ of Γ, we know we can derive $\Delta \vdash_k e : C$. The additional premise is still valid, and applying STATESEL we obtain $\Delta \vdash_{k+1} e.s_i : T_i$.

- ASS. In this case $J = \Gamma \vdash_{k+1} x = e : T$ was derived from $\Gamma \vdash_k x : T$ and $\Gamma \vdash_k e : T$. By inductive hypothesis, given a permutation Δ of Γ, we know we can derive both $\Delta \vdash_k x : T$ and $\Delta \vdash_k e : T$. Hence, by applying ASS to them we can derive $\Delta \vdash_{k+1} x = e : T$.

- MAPPASS. In this case $J = \Gamma \vdash_{k+1} e_1[e_2 \to e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from $\Gamma \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash_k e_2 : T_1$, and $\Gamma \vdash_k e_3 : T_2$. By inductive hypothesis, given a permutation Δ of Γ, we can derive $\Delta \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Delta \vdash_k e_2 : T_1$, and $\Delta \vdash_k e_3 : T_2$. By the application of MAPPASS we obtain $\Delta \vdash_{k+1} e_1[e_2 \to e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)$.

- STATEASS. This case is very similar to ASS: $J = \Gamma \vdash_{k+1} e_1.s = e_2 : T$ was derived from $\Gamma \vdash_k e_1.s : T$ and $\Gamma \vdash_k e_2 : T$. By inductive hypothesis, given a permutation Δ of Γ, we know we can derive both $\Delta \vdash_k e_1.s : T$ and $\Delta \vdash_k e_2 : T$. Hence, by applying STATEASS to them we can derive $\Delta \vdash_{k+1} e_1.s = e_2 : T$.

- NEW. In this case $J = \Gamma \vdash_{k+1} \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : C$ was derived from $\Gamma \vdash_k \tilde{e} : \tilde{T}$ and $\Gamma \vdash_k e' : \mathsf{uint}$, together with the additional premise $|\tilde{e}| = |\tilde{s}|$, where $\mathrm{sv}(C) = \tilde{T}\,s$. By inductive hypothesis, we know that, given a permutation Δ of Γ, we can derive both $\Delta \vdash_k \tilde{e} : \tilde{T}$ and $\Delta \vdash_k e' : \mathsf{uint}$. Provided that the additional premise checking the length of the tuples $\tilde{e}$ and $\tilde{s}$ is still valid, we can apply NEW to obtain $\Delta \vdash_{k+1} \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : C$.

- CONTRRETR. In this case $J = \Gamma \vdash_{k+1} C(e) : C$ was derived from $\Gamma \vdash_k e : \mathsf{address}$. By inductive hypothesis $\Delta \vdash_k e : \mathsf{address}$ is derivable, where Δ is a permutation of Γ. Considering $\Delta \vdash C(e) : C$, notice it can be derived from $\Delta \vdash_k e : \mathsf{address}$ applying CONTRRETR, and its derivation has height $k + 1$.

- TRANSFER. In this case $J = \Gamma \vdash_{k+1} e_1.\mathsf{transfer}(e_2) : \mathsf{unit}$ was derived from $\Gamma \vdash_k e_1 : \mathsf{address}$ and $\Gamma \vdash_k e_2 : \mathsf{uint}$. By inductive hypothesis, considering a permutation Δ of Γ, we can derive $\Delta \vdash_k e_1 : \mathsf{address}$ and $\Delta \vdash_k e_2 : \mathsf{uint}$. Lastly, we can apply TRANSFER to them to obtain $\Delta \vdash_{k+1} e_1.\mathsf{transfer}(e_2) : \mathsf{unit}$.

- CALL. In this case $J = \Gamma \vdash_{k+1} e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$ was derived from $\Gamma \vdash_k e_1 : C$, $\Gamma \vdash_k e_2 : \mathsf{uint}$, and $\Gamma \vdash_k \tilde{e} : \tilde{T_1}$, together with the additional premises checking the length of the tuple $\tilde{e}$ ($|\tilde{e}| = |\tilde{T_1}|$) and the type of f in C ($\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$). By inductive hypothesis on the three judgments of height k, and given a permutation Δ of Γ, also the following judgments can be derived: $\Delta \vdash_k e : C$, $\Delta \vdash_k e_2 : \mathsf{uint}$, and $\Delta \vdash_k \tilde{e} : \tilde{T_1}$. Applying CALL we derive $\Delta \vdash_{k+1} e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$.

- CALLTOPLEVEL. In this case $J = \Gamma \vdash_{k+1} e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T_2$ was derived from $\Gamma \vdash_k e_3 : \mathsf{address}$ and $\Gamma \vdash_k e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$. Let Δ be a permutation of Γ; by inductive hypothesis the judgments $\Delta \vdash_k e_3 : \mathsf{address}$ and $\Delta \vdash_k e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$. We then apply CALLTOPLEVEL and obtain a derivation of $\Delta \vdash_{k+1} e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T_2$.

- CALLVALUE. In this case $J = \Gamma \vdash_{k+1} e_1.\mathsf{value}(e_2)(\tilde{e}) : T_2$ was derived from $\Gamma \vdash_k e_1 : \tilde{T_1} \to T_2$, $\Gamma \vdash_k e_2 : \mathsf{uint}$, and $\Gamma \vdash_k \tilde{e} : \tilde{T_1}$, together with the premise checking the length of the tuple $\tilde{e}$ ($|\tilde{e}| = |\tilde{T_1}|$). By inductive hypothesis on the three judgments of height k, and given a permutation Δ of Γ, also the following judgments can be derived: $\Delta \vdash_k e_1 : \tilde{T_1} \to T_2$, $\Delta \vdash_k e_2 : \mathsf{uint}$, and $\Delta \vdash_k \tilde{e} : \tilde{T_1}$. Applying CALLVALUE we derive $\Delta \vdash_{k+1} e_1.\mathsf{value}(e_2)(\tilde{e}) : T_2$.

□

E.2 Weakening Lemma

Weakening Lemma is formalized over FS configurations $\langle \beta, \sigma, e \rangle$. To prove it, we shall first prove the validity of the same Lemma on the projections on β (Lemma 10), σ (Lemma 11), and e (Lemma 12). In the proof that follow, let $\Gamma' = \Gamma \cdot \Delta$.

E.2.1 Weakening of β

Here we prove the projection of the lemma of Weakening on the first component of $\langle \beta, \sigma, e \rangle$: β.

Lemma 10 (Weakening of β)**.**
Let $\Gamma \vdash \beta$ be a derivable judgment, and let Δ be such that $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$. Then $\Gamma \cdot \Delta \vdash \beta$ can be derived and its derivation has the same height as the previous one.

Proof. We prove this lemma by induction on the height of the derivation of the judgment $\Gamma \vdash \beta$.

Base case This case corresponds to the axiom EMPTYBLOCKCHAIN in Section 6.3. The height of the derivation is 1, and $\beta = \emptyset$, that is $\Gamma \vdash_1 \emptyset$. $\emptyset$ is well-formed regardless of Γ, and hence it is also true $\Gamma' \vdash_1 \emptyset$.

Inductive cases Given a judgment J such that its derivation has height $k+1$, we prove the inductive cases on the last rule used to derive J. We assume the lemma for the judgments with height at most k and we prove it for those with height $k+1$.

- VARIABLE. In this case $J = \Gamma \vdash_{k+1} \beta \cdot [x \mapsto v]$ was derived from $\Gamma \vdash_k \beta$, $\Gamma \vdash_k x : T$, and $\Gamma \vdash_k v : T$, where $x \notin \mathrm{dom}(\beta)$. By inductive hypothesis, also the judgments $\Gamma' \vdash_k \beta$, $\Gamma' \vdash_k x : T$, and $\Gamma' \vdash_k v : T$ are derivable. Applying VARIABLE we can thus derive $\Gamma' \vdash_{k+1} \beta \cdot [x \mapsto v]$, which is what we wanted to prove.
- CONTRACT. In this case $J = \Gamma \vdash_{k+1} \beta \cdot [(c, a) \mapsto (C, s\tilde{:}v, n)]$ was derived from $\Gamma \vdash_k \beta$, $\Gamma \vdash_k c : C$, $\Gamma \vdash_k a : \mathsf{address}$, $\Gamma \vdash_k n : \mathsf{uint}$, and $\Gamma \vdash_k \tilde{v} : \tilde{T}$, where $(c, a) \notin \mathrm{dom}(\beta)$ and $\mathrm{sv}(C) = \tilde{T}\, s$. By inductive hypothesis, also the judgments $\Gamma' \vdash_k \beta$, $\Gamma' \vdash_k c : C$, $\Gamma' \vdash_k a : \mathsf{address}$, $\Gamma' \vdash_k n : \mathsf{uint}$, and $\Gamma' \vdash_k \tilde{v} : \tilde{T}$ are derivable. Applying CONTRACT we can thus derive $\Gamma' \vdash_{k+1} \beta \cdot [(c, a) \mapsto (C, s\tilde{:}v, n)]$, which is what we wanted to prove.

□

E.2.2 Weakening of σ

Here we prove the projection of the lemma of Weakening on the second component of $\langle \beta, \sigma, e \rangle$: σ.

Lemma 11 (Weakening of σ)**.**
Let $\Gamma \vdash \sigma$ be a derivable judgment, and let Δ be such that $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$. Then $\Gamma \cdot \Delta \vdash \sigma$ can be derived and its derivation has the same height as the previous one.

Proof. We prove this lemma by induction on the height of the derivation of the judgment $\Gamma \vdash \sigma$.

Base case Rule CALLSTACK in Section 6.3 has, as a base case, the well-formedness of β, proven by Lemma 10.

Inductive case Let the judgment $\Gamma \vdash \sigma \cdot a$ have height $k+1$. We assume the lemma for the judgments with height at most k and we prove it for those with height at most $k+1$.

$\Gamma \vdash_{k+1} \sigma \cdot a$ was derived from $\Gamma \vdash_k \sigma$ and $\Gamma \vdash_k a : \mathsf{address}$. By inductive hypothesis we can derive $\Gamma' \vdash_k \sigma$ and $\Gamma' \vdash_k a : \mathsf{address}$, and applying CALLSTACK we obtain a derivation of $\Gamma' \vdash_{k+1} \sigma \cdot a$. □

E.2.3 Weakening of e

Here we prove the projection of the lemma of Weakening on the first component of $\langle \beta, \sigma, e \rangle$: e.

Lemma 12 (Weakening of e)**.**
Let $\Gamma \vdash e : T$ be a derivable judgment, and let Δ be such that $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$. Then $\Gamma \cdot \Delta \vdash e : T$ can be derived and its derivation has the same height as the previous one.

Proof. We prove this lemma by induction on the height of the derivation of the judgment $\Gamma \vdash e : T$.

Base cases These cases correspond to the axioms in Section 6.3. The height of these derivations is always 1.

- REF. The judgment is $\Gamma \vdash c : C$. From Case 9 of Lemma 2 we know that $c : C \in \Gamma$. We defined Γ' as $\Gamma \cdot \Delta$, so $c : C \in \Gamma \Rightarrow c : C \in \Gamma'$. Hence, $\Gamma' \vdash c : C$ is derivable.
- VAR. The judgment is $\Gamma \vdash x : T$. From Case 10 of Lemma 2 we know that $x : T \in \Gamma$. We defined Γ' as $\Gamma \cdot \Delta$, so $x : T \in \Gamma \Rightarrow x : T \in \Gamma'$. Hence, $\Gamma' \vdash x : T$ is derivable.
- TRUE. The judgment is $\Gamma \vdash \mathsf{true} : \mathsf{bool}$. From Case 1 of Lemma 2 we know that this judgment is derivable with height 1 regardless of Γ, and so is $\Gamma' \vdash \mathsf{true} : \mathsf{bool}$.
- FALSE. The judgment is $\Gamma \vdash \mathsf{false} : \mathsf{bool}$. From Case 2 of Lemma 2 we know that this judgment is derivable with height 1 regardless of Γ, and so is $\Gamma' \vdash \mathsf{false} : \mathsf{bool}$.
- NAT. The judgment is $\Gamma \vdash n : \mathsf{uint}$. From Case 3 of Lemma 2 we know that this judgment is derivable with height 1 whenever $n \in \mathbb{N}^+$, regardless of Γ, and so is $\Gamma' \vdash n : \mathsf{uint}$.
- ADDRESS. The judgment is $\Gamma \vdash a : \mathsf{address}$. From Case 6 of Lemma 2 we know that $a : \mathsf{address} \in \Gamma$. We defined Γ' as $\Gamma \cdot \Delta$, so $a : \mathsf{address} \in \Gamma \Rightarrow a : \mathsf{address} \in \Gamma'$. Hence, $\Gamma' \vdash a : \mathsf{address}$ is derivable.
- UNIT. The judgment is $\Gamma \vdash \mathsf{u} : \mathsf{unit}$. From Case 4 of Lemma 2 we know that this judgment is derivable with height 1 regardless of Γ, and so is
- REVERT. The judgment is $\Gamma \vdash \mathsf{revert} : T$. This judgment is derivable with height 1 regardless of Γ, and so is $\Gamma' \vdash \mathsf{revert} : T$.

Inductive cases Given a judgment J such that its derivation has height $k + 1$, we prove the inductive cases on the last rule used to derive J. We assume the lemma for the judgments with height at most k and we prove it for those with height $k + 1$.

- FUN. In this case $J = \Gamma \vdash_{k+1} c.f : \tilde{T_1} \to T_2$ was derived from the judgment $\Gamma \vdash_k c : C$ with the premise $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$. We can apply the inductive hypothesis to say that $\Gamma' \vdash_k c : C$ is derivable. Provided that the premise $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$ is still valid, applying FUN we conclude that also $\Gamma' \vdash_{k+1} c.f : \tilde{T_1} \to T_2$ is derivable.

- MAPPING. In this case $J = \Gamma \vdash_{k+1} M : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from $\Gamma \vdash_k \tilde{k} : \tilde{T_1}$ and $\Gamma \vdash_k \tilde{v} : \tilde{T_2}$. By induction hypothesis, also $\Gamma' \vdash_k \tilde{k} : \tilde{T_1}$ and $\Gamma' \vdash_k \tilde{v} : \tilde{T_2}$ can be derived. Hence, applying MAPPING we obtain a derivation of $\Gamma' \vdash_{k+1} M : \mathsf{mapping}(T_1 \Rightarrow T_2)$.

- BAL. In this case $J = \Gamma \vdash_{k+1} \mathsf{balance}(e) : \mathsf{uint}$ was derived from $\Gamma \vdash_k e : \mathsf{address}$. By inductive hypothesis, $\Gamma' \vdash_k e : \mathsf{address}$ is derivable. We then apply BAL to conclude that the judgment $\Gamma' \vdash_{k+1} \mathsf{balance}(e) : \mathsf{uint}$ can be derived.

- ADDR. In this case $J = \Gamma \vdash_{k+1} \mathsf{address}(e) : \mathsf{address}$ was derived from $\Gamma \vdash_k e : C$. By inductive hypothesis, $\Gamma' \vdash_k e : C$ is derivable. We then apply ADDR to conclude that the judgment $\Gamma' \vdash_{k+1} \mathsf{address}(e) : \mathsf{address}$ can be derived, too.

- RETURN. In this case $J = \Gamma \vdash_{k+1} \mathsf{return}\, e : T$ was derived from $\Gamma \vdash_k e : T$. By inductive hypothesis, $\Gamma' \vdash_k e : T$ is derivable. We then apply RETURN to conclude that the judgment $\Gamma' \vdash_{k+1} \mathsf{return}\, e : T$ can be derived, too.

- IF. In this case $J = \Gamma \vdash_{k+1} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ was derived from $\Gamma \vdash_k e_1 : \mathsf{bool}$, $\Gamma \vdash_k e_2 : T$, and $\Gamma \vdash_k e_3 : T$. We can thus apply the inductive hypothesis and say that $\Gamma' \vdash_k e_1 : \mathsf{bool}$, $\Gamma' \vdash_k e_2 : T$, and $\Gamma' \vdash_k e_3 : T$ are all derivable. We then apply IF to derive $\Gamma' \vdash_{k+1} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$.

- SEQ. In this case $J = \Gamma \vdash_{k+1} e_1; e_2 : T_2$ was derived from $\Gamma \vdash_k e_1 : T_1$ and $\Gamma \vdash_k e_2 : T_2$. By inductive hypothesis, these "smaller" judgments are derivable also under Γ', without any changes in height, that is $\Gamma' \vdash_k e_1 : T_1$ and $\Gamma' \vdash_k e_2 : T_2$. Applying SEQ we obtain a derivation $\Gamma' \vdash_{k+1} e_1; e_2 : T_2$.

- DECL. In this case $J = \Gamma \vdash_{k+1} T_1\ x = e_1; e_2 : T_2$ was derived from $\Gamma \vdash_k e_1 : T_1$ and $\Gamma, x : T_1 \vdash_k e_2 : T_2$. On the former we can apply the inductive hypothesis and say that $\Gamma' \vdash_k e_1 : T_1$ is derivable. On the contrary, we cannot apply the inductive hypothesis on the latter, since the context is $\Gamma, x : T_1$ and not Γ. Still, we know there exists a derivation of height k for this judgment, otherwise J would not be derivable, but this is a contradiction, since we assumed J has a valid derivation of height $k+1$. Hence, it comes from another judgment J' having a derivation of height $k-1$ where any of the rules defined in Section 6.3 was applied as a last step. We can now apply the inductive hypothesis on J', considering $\Gamma, x : T_1$ as a context and concluding that $\Gamma, x : T_1, \Delta \vdash_k e_2 : T_2$ is derivable. Lastly, applying DECL we obtain a derivation of $\Gamma' \vdash_{k+1} T_1\ x = e_1; e_2 : T_2$.

- MAPPSEL. In this case $J = \Gamma \vdash_{k+1} e_1[e_2] : T_2$ was derived from $\Gamma \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Gamma \vdash_k e_2 : T_1$. By inductive hypothesis also the judgments $\Gamma' \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Gamma' \vdash_k e_2 : T_1$ are derivable. Applying MAPPSEL we obtain a derivation of $\Gamma' \vdash_{k+1} e_1[e_2] : T_2$.

- STATESEL. In this case $J = \Gamma \vdash_{k+1} e.s_i : T_i$ was derived from $\Gamma \vdash_k e : C$, with the additional premise stating $s_i \in \tilde{s}$, where $\mathrm{sv}(C) = \tilde{T}\,\tilde{s}$. By inductive hypothesis we know that also $\Gamma' \vdash_k e : C$ is derivable. As a final step, applying STATESEL we obtain a derivation of $\Gamma' \vdash_{k+1} e.s_i : T_i$.

- ASS. In this case $J = \Gamma \vdash_{k+1} x = e : T$ was derived from $\Gamma \vdash_k x : T$ and $\Gamma \vdash_k e : T$. By inductive hypothesis we can derive with the same height also

$\Gamma' \vdash_k x : T$ and $\Gamma' \vdash_k e : T$, and applying ASS we obtain a derivation of $\Gamma' \vdash_{k+1} x = e : T$.

- MAPPASS. In this case $J = \Gamma \vdash_{k+1} e_1[e_2 \to e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from $\Gamma \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash_k e_2 : T_1$, and $\Gamma \vdash_k e_3 : T_2$. By inductive hypothesis we can derive with the same height also $\Gamma' \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma' \vdash_k e_2 : T_1$, and $\Gamma' \vdash_k e_3 : T_2$, and applying MAPPASS we obtain a derivation of $\Gamma' \vdash_{k+1} e_1[e_2 \to e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)$.

- STATEASS. In this case $J = \Gamma \vdash_{k+1} e_1.s = e_2 : T$ was derived from $\Gamma \vdash_k e_1.s : T$ and $\Gamma \vdash_k e_2 : T$. By inductive hypothesis we can derive with the same height also $\Gamma' \vdash_k e_1.s : T$ and $\Gamma' \vdash_k e_2 : T$, and applying STATEASS we obtain a derivation of $\Gamma' \vdash_{k+1} e_1.s = e_2 : T$.

- NEW. In this case $J = \Gamma \vdash_{k+1} \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : C$ was derived from $\Gamma \vdash_k \tilde{e} : \tilde{T}$ and $\Gamma \vdash_k e' : \mathsf{uint}$, together with the premise $|\tilde{e}| = |\tilde{s}|$, where $\mathrm{sv}(C) = \tilde{T}\,\tilde{s}$. By inductive hypothesis, $\Gamma' \vdash_k \tilde{e} : \tilde{T}$ and $\Gamma' \vdash_k e' : \mathsf{uint}$ can be derived. Furthermore, the premise checking the length of $\tilde{e}$ and $\tilde{s}$ is still valid, and we can apply NEW to derive $\Gamma' \vdash_{k+1} \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : C$.

- CONTRRETR. In this case $J = \Gamma \vdash_{k+1} C(e) : C$ was derived from $\Gamma \vdash_k e : \mathsf{address}$. By inductive hypothesis also $\Gamma' \vdash_k e : \mathsf{address}$ is derivable; applying CONTRRETR we then obtain a derivation of $\Gamma' \vdash_{k+1} C(e) : C$.

- TRANSFER. In this case $J = \Gamma \vdash_{k+1} e_1.\mathsf{transfer}(e_2) : \mathsf{unit}$ was derived from $\Gamma \vdash_k e_1 : \mathsf{address}$ and $\Gamma \vdash_k e_2 : \mathsf{uint}$. By inductive hypothesis we can derive $\Gamma' \vdash_k e_1 : \mathsf{address}$ and $\Gamma' \vdash_k e_2 : \mathsf{uint}$. We then apply TRANSFER to obtain a derivation of $\Gamma' \vdash_{k+1} e_1.\mathsf{transfer}(e_2) : \mathsf{unit}$.

- CALL. In this case $J = \Gamma \vdash_{k+1} e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$ was derived from $\Gamma \vdash_k e_1 : C$, $\Gamma \vdash_k e_2 : \mathsf{uint}$, and $\Gamma \vdash_k \tilde{e} : \tilde{T_1}$, together with the premises checking the length of the tuple $\tilde{e}$ ($|\tilde{e}| = |\tilde{T_1}|$) and the type of f in C ($\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$). By induction hypothesis we can derive $\Gamma' \vdash_k e_1 : C$, $\Gamma' \vdash_k e_2 : \mathsf{uint}$, and $\Gamma' \vdash_k \tilde{e} : \tilde{T_1}$. Furthermore, the other two premises are still valid, and thus we can apply CALL to derive $\Gamma' \vdash_{k+1} e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$.

- CALLTOPLEVEL. In this case $J = \Gamma \vdash_{k+1} e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T_2$ was derived from $\Gamma \vdash_k e_3 : \mathsf{address}$ and $\Gamma \vdash_k e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$. By induction hypothesis, the judgments $\Gamma' \vdash_k e_3 : \mathsf{address}$ and $\Gamma' \vdash_k e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$ are derivable. As a final step we apply CALLTOPLEVEL to derive $\Gamma' \vdash_{k+1} e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T_2$.

- CALLVALUE. In this case $J = \Gamma \vdash_{k+1} e_1.\mathsf{value}(e_2)(\tilde{e}) : T_2$ was derived from $\Gamma \vdash_k e_1 : \tilde{T_1} \to T_2$, $\Gamma \vdash_k e_2 : \mathsf{uint}$, and $\Gamma \vdash_k \tilde{e} : \tilde{T_1}$. There is another premise, checking the length of the tuple $\tilde{e}$ ($|\tilde{e}| = |\tilde{T_1}|$). By induction hypothesis, the judgments $\Gamma' \vdash_k e_1 : \tilde{T_1} \to T_2$, $\Gamma' \vdash_k e_2 : \mathsf{uint}$, and $\Gamma' \vdash_k \tilde{e} : \tilde{T_1}$ are derivable. Furthermore, the other premise is still valid, and we can thus apply CALLVALUE to obtain a derivation of $\Gamma' \vdash_{k+1} e_1.\mathsf{value}(e_2)(\tilde{e}) : T_2$.

□

E.2.4 Proof of the Lemma

We can now prove Lemma 4:

Lemma 4 (Weakening)**.**
Let $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ *be a derivable judgment, and let* Δ *be such that* $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$ *(i.e.* Γ *and* Δ *have no elements in common). Then* $\Gamma \cdot \Delta \vdash \langle \beta, \sigma, e \rangle : T$ *can be derived and its derivation has the same height as the previous one.*

Proof. By hypothesis $\Gamma \vdash_{k+1} \langle \beta, \sigma, e \rangle : T$: by rule CONFIGURATION this means that also $\Gamma \vdash_k \beta$, $\Gamma \vdash_k \sigma$, and $\Gamma \vdash_k e : T$ are derivable. By, respectively, Lemma 10, 11, and 12 we know that there exists a derivation for $\Gamma' \vdash_k \beta$, $\Gamma' \vdash_k \sigma$, and $\Gamma' \vdash_k e : T$. Hence, applying CONFIGURATION to the latter three judgments we can derive $\Gamma' \vdash_{k+1} \langle \beta, \sigma, e \rangle : T$, which is what we wanted to prove. □

E.3 Substitution Lemma

We recall Lemma 5:

Lemma 5 (Substitution)**.**
If $\Gamma, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}, \mathsf{msg.value} : \mathsf{uint} \vdash e : T$, $\Gamma \vdash c : C$, $\Gamma \vdash a : \mathsf{address}$, *and* $\Gamma \vdash n : \mathsf{uint}$, *then* $\Gamma \vdash e\{\mathsf{this} := c, \mathsf{msg.sender} := a, \mathsf{msg.value} := n\} : T$.

Proof. Let $\Gamma' = \Gamma, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}, \mathsf{msg.value} : \mathsf{uint}$. We prove this lemma by induction on the height of the judgment $\Gamma' \vdash e : T$.

We shall make use of the following notation to make the proof more readable:

$$\mathrm{subst}(e) = e\{\mathsf{this} := c, \mathsf{msg.sender} := a, \mathsf{msg.value} := n\}$$

Base cases These cases correspond to the axioms in Section 6.3. The height of these derivations is always 1.

- REF. In this case the judgment is $\Gamma' \vdash d : D$. There are two sub-cases:
  - $d \neq \mathsf{this}$. By Case 9 of Lemma 2 we obtain $d : D \in \Gamma'$. Since $d \neq \mathsf{msg.value}$, $d \neq \mathsf{msg.sender}$, and $d \neq \mathsf{this}$, from $d : D \in \Gamma'$ follows $d : D \in \Gamma$. Lastly, $d : D \in \Gamma \Rightarrow \Gamma \vdash d : D$ by applying REF, since $\mathrm{subst}(d) = d$.
  - $d = \mathsf{this}$. By Case 9 of Lemma 2 we obtain $D = C$. We are to prove $\Gamma \vdash \mathrm{subst}(\mathsf{this}) : C$, but $\mathrm{subst}(\mathsf{this}) = c$ and the judgment becomes $\Gamma \vdash c : C$, which is true by the second hypothesis.
- VAR. In this case the judgment is $\Gamma' \vdash x : T$ and $\mathrm{subst}(e) = \mathrm{subst}(x) = x$, where $x \neq \mathsf{msg.value}$, $x \neq \mathsf{msg.sender}$, and $x \neq \mathsf{this}$. By Case 10 of Lemma 2 we know $x : T \in \Gamma'$, and, for what we just said ($x \neq \mathsf{msg.value}$, $x \neq \mathsf{msg.sender}$, and $x \neq \mathsf{this}$), $x : T \in \Gamma$, too. We are to prove $\Gamma \vdash x : T$, which is true by VAR, since we just noticed $x : T \in \Gamma$.
- TRUE. In this case the judgment is $\Gamma' \vdash \mathsf{true} : \mathsf{bool}$ and $\mathrm{subst}(e) = \mathrm{subst}(\mathsf{true}) = \mathsf{true}$. We are to prove $\Gamma \vdash \mathsf{true} : \mathsf{bool}$, which is true by TRUE.

- FALSE. In this case the judgment is $\Gamma' \vdash \mathsf{false} : \mathsf{bool}$ and $\mathrm{subst}(e) = \mathrm{subst}(\mathsf{false})$ $= \mathsf{false}$. We are to prove $\Gamma \vdash \mathsf{false} : \mathsf{bool}$, which is true by FALSE.

- NAT. In this case the judgment is $\Gamma' \vdash m : \mathsf{uint}$, where $m \neq \mathsf{msg.value}$. In this case $\mathrm{subst}(m) = m$ and the judgment becomes $\Gamma \vdash m : \mathsf{uint}$, which is true by NAT.

- UNIT. In this case the judgment is $\Gamma' \vdash \mathsf{u} : \mathsf{unit}$ and $\mathrm{subst}(e) = \mathrm{subst}(\mathsf{u}) = \mathsf{u}$. We are to prove $\Gamma \vdash \mathsf{u} : \mathsf{unit}$, which is true by UNIT.

- ADDRESS. In this case the judgment is $\Gamma' \vdash a' : \mathsf{address}$, where $a' \neq \mathsf{msg.sender}$. In this case $\mathrm{subst}(a') = a'$. By Case 6 of Lemma 2 we know $a' : \mathsf{address} \in \Gamma'$, and thus $a' : \mathsf{address} \in \Gamma$. We are to prove $\Gamma \vdash a' : \mathsf{address}$, which is true because $a' : \mathsf{address} \in \Gamma$.

- REVERT. In this case the judgment is $\Gamma' \vdash \mathsf{revert} : T$ and $\mathrm{subst}(e) = \mathrm{subst}(\mathsf{revert}) = \mathsf{revert}$. We are to prove $\Gamma \vdash \mathsf{revert} : T$, which is true by rule REVERT.

Inductive cases Given a judgment J such that its derivation has height $k + 1$, we prove the inductive cases on the last rule used to derive J. We assume the lemma for the judgments with height at most k and we prove it for those with height $k + 1$.

- FUN. In this case $J = \Gamma' \vdash_{k+1} d.f : \tilde{T_1} \to T_2$ was derived from $\Gamma' \vdash_k d : D$. We again have two sub-cases:

  - if $d \neq \mathsf{this}$ then, by Case 9 of Lemma 2 we obtain $d : D \in \Gamma'$. We supposed $d \neq \mathsf{this}$, but it is also true that $d \neq \mathsf{msg.sender}$ and $d \neq \mathsf{msg.value}$. Hence, $d : D \in \Gamma' \Rightarrow d : D \in \Gamma \Rightarrow \Gamma \vdash d : D$. Furthermore, $\mathrm{subst}(d.f) = d.f$, and the judgment $\Gamma \vdash d.f : \tilde{T_1} \to T_2$ is derivable.
  - id $d = \mathsf{this}$ then, by Case 9 of Lemma 2 we know $D = C$. We are to prove $\Gamma \vdash \mathrm{subst}(\mathsf{this}.f) : D$, but $\mathrm{subst}(\mathsf{this}.f) = c.f$ and the judgment becomes $\Gamma \vdash c.f : \tilde{T_1} \to T_2$. By the second hypothesis we know we can derive $\Gamma \vdash c : C$, and applying FUN we can also derive $\Gamma \vdash c.f : \tilde{T_1} \to T_2$, which is what we wanted to prove since $\mathrm{subst}(\mathsf{this}.f) = c.f$.

- MAPPING. In this case $J = \Gamma' \vdash_{k+1} M : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from $\Gamma' \vdash_k \tilde{k} : T_1$ and $\Gamma' \vdash_k \tilde{v} : T_2$. Since both $\tilde{k}$ and $\tilde{v}$ are tuples of values, $\mathrm{subst}(\tilde{k}) = \tilde{k}$ and $\mathrm{subst}(\tilde{v}) = \tilde{v}$. By inductive hypothesis, $\Gamma \vdash \tilde{k} : T_1$ and $\Gamma \vdash \tilde{v} : T_2$, and applying MAPPING we derive $\Gamma \vdash M : \mathsf{mapping}(T_1 \Rightarrow T_2)$, as required.

- BAL. In this case $J = \Gamma' \vdash_{k+1} \mathsf{balance}(e) : \mathsf{uint}$ was derived from $\Gamma' \vdash_k e : \mathsf{address}$. By inductive hypothesis we have $\Gamma \vdash \mathrm{subst}(e) : \mathsf{address}$, and applying BAL we obtain $\Gamma \vdash \mathsf{balance}(\mathrm{subst}(e)) : \mathsf{uint}$, which is the same as $\Gamma \vdash \mathrm{subst}(\mathsf{balance}(e)) : \mathsf{uint}$ (see Figure 4.3).

- ADDR. In this case $J = \Gamma' \vdash_{k+1} \mathsf{address}(e) : \mathsf{address}$ was derived from $\Gamma' \vdash_k e : C$. By inductive hypothesis we have $\Gamma \vdash \mathrm{subst}(e) : C$, and applying ADDR we obtain $\Gamma \vdash \mathsf{address}(\mathrm{subst}(e)) : \mathsf{address}$, which is the same as $\Gamma \vdash \mathrm{subst}(\mathsf{address}(e)) : \mathsf{address}$ (see Figure 4.3).

- RETURN. In this case $J = \Gamma' \vdash_{k+1} \mathsf{return}\, e : T$ was derived from $\Gamma' \vdash_k e : T$. By inductive hypothesis we have $\Gamma \vdash \mathrm{subst}(e) : T$, and applying RETURN we obtain $\Gamma \vdash \mathsf{return}\, \mathrm{subst}(e) : T$, which is the same as $\Gamma \vdash \mathrm{subst}(\mathsf{return}\, e) : T$ (see Figure 4.3).

- IF. In this case $J = \Gamma' \vdash_{k+1} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ was derived from $\Gamma' \vdash_k e_1 : \mathsf{bool}$, $\Gamma' \vdash_k e_2 : T$ and $\Gamma' \vdash_k e_3 : T$. By inductive hypothesis, we can derive $\Gamma \vdash \mathrm{subst}(e_1) : \mathsf{bool}$, $\Gamma \vdash \mathrm{subst}(e_2) : T$ and $\Gamma \vdash \mathrm{subst}(e_3) : T$. We are to prove $\Gamma \vdash \mathrm{subst}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_2) : T$, which is derived applying IF to the latter judgments together with the equivalence $\mathsf{if}\ \mathrm{subst}(e_1)\ \mathsf{then}\ \mathrm{subst}(e_2)\ \mathsf{else}\ \mathrm{subst}(e_3) = \mathrm{subst}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_2)$.

- SEQ. In this case $J = \Gamma' \vdash_{k+1} e_1; e_2 : T_2$ was derived from $\Gamma' \vdash_k e_1 : T_1$ and $\Gamma' \vdash_k e_2 : T_2$. By inductive hypothesis we obtain $\Gamma \vdash \mathrm{subst}(e_1) : T_1$ and $\Gamma \vdash \mathrm{subst}(e_2) : T_2$; applying SEQ we derive $\Gamma \vdash \mathrm{subst}(e_1); \mathrm{subst}(e_2) : T_2$, which is what we wanted to prove since $\mathrm{subst}(e_1; e_2) = \mathrm{subst}(e_1); \mathrm{subst}(e_2)$.

- DECL. In this case $J = \Gamma' \vdash_{k+1} T_1\ x = e_1; e_2 : T_2$ was derived from $\Gamma' \vdash_k e_1 : T_1$ and $\Gamma', x : T_1 \vdash_k e_2 : T_2$. By inductive hypothesis we can derive $\Gamma \vdash \mathrm{subst}(e_1) : T_1$. To apply the inductive hypothesis to the latter judgment we need to rearrange the context. By Lemma 3 we know that $\Gamma, x : T_1, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}, \mathsf{msg.value} : \mathsf{uint} \vdash_k e_2 : T_2$ can be derived with the same height. Furthermore, we can apply Lemma 4 to the other three hypotheses to obtain a derivation of the judgments
  $\Gamma, x : T_1, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}, \mathsf{msg.value} : \mathsf{uint} \vdash_k c : C$,
  $\Gamma, x : T_1, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}, \mathsf{msg.value} : \mathsf{uint} \vdash_k a : \mathsf{address}$, and
  $\Gamma, x : T_1, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}, \mathsf{msg.value} : \mathsf{uint} \vdash_k n : \mathsf{uint}$. Now, by inductive hypothesis, also
  $\Gamma, x : T_1, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}, \mathsf{msg.value} : \mathsf{uint} \vdash \mathrm{subst}(e_2) : T_2$
  can be derived. Lastly, by applying DECL we obtain a derivation of $\Gamma \vdash T_1\ x = \mathrm{subst}(e_1); \mathrm{subst}(e_2) : T_2$, which is the same as $\Gamma \vdash \mathrm{subst}(T_1\ x = e_1; e_2) : T_2$ since $\mathrm{subst}(T_1\ x = e_1; e_2) = \mathrm{subst}(T_1\ x = e_1); \mathrm{subst}(e_2) =$
  $T_1\ x = \mathrm{subst}(e_1); \mathrm{subst}(e_2)$.

- MAPPSEL. In this case $J = \Gamma' \vdash_{k+1} e_1[e_2] : T_2$ was derived from $\Gamma' \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Gamma' \vdash_k e_2 : T_1$. By inductive hypothesis we derive $\Gamma \vdash \mathrm{subst}(e_1) : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Gamma \vdash \mathrm{subst}(e_2) : T_1$; then, applying MAPPSEL, we obtain a derivation of $\Gamma \vdash \mathrm{subst}(e_1)[\mathrm{subst}(e_2)] : T_2$, which is what we wanted to prove since $\mathrm{subst}(e_1[e_2] = \mathrm{subst}(e_1)[\mathrm{subst}(e_2)])$.

- STATESEL. In this case $J = \Gamma' \vdash_{k+1} e.s_i : T_i$ was derived from $\Gamma' \vdash_k e : C$, where $\mathrm{sv}(C) = \tilde{T}\,\tilde{s}$ and $s_i \in \tilde{s}$. By inductive hypothesis we obtain $\Gamma \vdash \mathrm{subst}(e) : C$; then, applying STATESEL we derive $\Gamma \vdash \mathrm{subst}(e).s_i : T_i$.

- ASS. In this case $J = \Gamma' \vdash_{k+1} x = e : T$ was derived from $\Gamma' \vdash_k x : T$ and $\Gamma' \vdash_k e : T$. By inductive hypothesis we have $\Gamma \vdash \mathrm{subst}(x) : T$ and $\Gamma \vdash \mathrm{subst}(e) : T$; applying ASS we obtain $\Gamma \vdash \mathrm{subst}(x) = \mathrm{subst}(e) : T$, which is the same as $\Gamma \vdash \mathrm{subst}(x = e) : T$ since $\mathrm{subst}(x) = x$ and $\mathrm{subst}(x = e) = (x = \mathrm{subst}(e))$.

- MAPPASS. In this case $J = \Gamma' \vdash_{k+1} e_1[e_2 \to e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from $\Gamma' \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma' \vdash_k e_2 : T_1$, and $\Gamma' \vdash_k$

$e_3 : T_2$. By inductive hypothesis we derive $\Gamma \vdash \text{subst}(e_1) : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash \text{subst}(e_2) : T_1$, and $\Gamma \vdash \text{subst}(e_3) : T_2$. By MAPPASS we obtain $\Gamma \vdash \text{subst}(e_1)[\text{subst}(e_2) \to \text{subst}(e_3)] : \mathsf{mapping}(T_1 \Rightarrow T_2)$, which is what we wanted to prove since $\text{subst}(e_1[e_2 \to e_3]) = \text{subst}(e_1)[\text{subst}(e_2) \to \text{subst}(e_3)]$.

- STATEASS. In this case $J = \Gamma' \vdash_{k+1} e_1.s = e_2 : T$ was derived from $\Gamma' \vdash_k e_1.s : T$ and $\Gamma' \vdash_k e_2 : T$. By inductive hypothesis we obtain $\Gamma \vdash \text{subst}(e_1.s) : T$ and $\Gamma \vdash \text{subst}(e_2) : T$; then, applying STATEASS we derive $\Gamma \vdash \text{subst}(e_1).s = \text{subst}(e_2) : T$, which is what we wanted to prove since $\text{subst}(e_1.s = e_2) = \text{subst}(e_1).s = \text{subst}(e_2)$.

- NEW. In this case $J = \Gamma' \vdash_{k+1} \mathsf{new}\ C.\mathsf{value}(e_1)(\tilde{e}) : C$ was derived from $\Gamma' \vdash_k \tilde{e} : \tilde{T}$ and $\Gamma' \vdash_k e_1 : \mathsf{uint}$, where $\text{sv}(C) = \tilde{T}\,s$ and $|\tilde{e}| = |\tilde{s}|$. By inductive hypothesis we obtain $\Gamma \vdash \text{subst}(\tilde{e}) : \tilde{T}$ and $\Gamma \vdash \text{subst}(e_1) : \mathsf{uint}$; applying NEW we conclude $\Gamma \vdash \text{subst}(\mathsf{new}\ C.\mathsf{value}(e_1)(\tilde{e})) : C$, since $\text{subst}(\mathsf{new}\ C.\mathsf{value}(e_1)(\tilde{e})) = \mathsf{new}\ C.\mathsf{value}(\text{subst}(e_1))(\text{subst}(\tilde{e}))$.

- CONTRRETR. In this case $J = \Gamma' \vdash_{k+1} C(e) : C$ was derived from $\Gamma' \vdash_k e : \mathsf{address}$. By inductive hypothesis we have $\Gamma \vdash \text{subst}(e) : \mathsf{address}$, and applying CONTRRETR we obtain $\Gamma \vdash C(\text{subst}(e)) : C$, which is the same as $\Gamma \vdash \text{subst}(C(e)) : C$.

- TRANSFER. In this case $J = \Gamma' \vdash_{k+1} e_1.\mathsf{transfer}(e_2) : \mathsf{unit}$ was derived from $\Gamma' \vdash_k e_1 : \mathsf{address}$ and $\Gamma' \vdash_k e_2 : \mathsf{uint}$. By inductive hypothesis we get $\Gamma \vdash \text{subst}(e_1) : \mathsf{address}$ and $\Gamma \vdash \text{subst}(e_2) : \mathsf{uint}$. Furthermore, applying TRANSFER we derive $\Gamma \vdash \text{subst}(e_1).\mathsf{transfer}(\text{subst}(e_2)) : \mathsf{unit}$, which is the same as $\Gamma \vdash \text{subst}(e_1.\mathsf{transfer}(e_2)) : \mathsf{unit}$.

- CALL. In this case $J = \Gamma' \vdash_{k+1} e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$ was derived from $\Gamma' \vdash_k e_1 : C$, $\Gamma' \vdash_k e_2 : \mathsf{uint}$, and $\Gamma' \vdash_k \tilde{e} : \tilde{T}_1$, where $\text{ftype}(C, f) = \tilde{T}_1 \to T_2$ and $|\tilde{e}| = |\tilde{T}_1|$. By inductive hypothesis we get $\Gamma \vdash \text{subst}(e_1) : C$, $\Gamma \vdash \text{subst}(e_2) : \mathsf{uint}$, and $\Gamma \vdash \text{subst}(\tilde{e}) : \tilde{T}_1$: applying CALL we derive $\Gamma \vdash \text{subst}(e_1).f.\mathsf{value}(\text{subst}(e_2))(\text{subst}(\tilde{e})) : T_2$, which is what we wanted to prove since $\text{subst}(e_1.f.\mathsf{value}(e_2)(\tilde{e})) = \text{subst}(e_1).f.\mathsf{value}(\text{subst}(e_2))(\text{subst}(\tilde{e}))$.

- CALLTOPLEVEL. In this case is very similar to the previous one, with the only addition of the judgment $\Gamma \vdash e_3 : \mathsf{address}$ checking the well-typing of the third parameter.

- CALLVALUE. In this case $J = \Gamma' \vdash_{k+1} e_1.\mathsf{value}(e_2)(\tilde{e}) : T_2$ was derived from $\Gamma' \vdash_k e_1 : \tilde{T}_1 \to T_2$, $\Gamma' \vdash_k e_2 : \mathsf{uint}$, and $\Gamma' \vdash_k \tilde{e} : \tilde{T}_1$, where $|\tilde{e}| = |\tilde{T}_1|$. By inductive hypothesis we get $\Gamma \vdash \text{subst}(e_1) : \tilde{T}_1 \to T_2$, $\Gamma \vdash \text{subst}(e_2) : \mathsf{uint}$, and $\Gamma \vdash \text{subst}(\tilde{e}) : \tilde{T}_1$: applying CALLVALUE we derive $\Gamma \vdash \text{subst}(e_1).\mathsf{value}(\text{subst}(e_2))(\text{subst}(\tilde{e})) : T_2$, which is what we wanted to prove since $\text{subst}(e_1.\mathsf{value}(e_2)(\tilde{e})) = \text{subst}(e_1).\mathsf{value}(\text{subst}(e_2))(\text{subst}(\tilde{e}))$.

□

E.4 Progress Theorem

Before proving Theorem 3, we shall prove an additional lemma stating that, for well-typed judgments, $\text{dom}(\Gamma) \supseteq \text{dom}(\beta)$, that is, every x, c, or $a \in \Gamma$ is also in β. Again,

we shall abuse the notation regarding pairs (c, a) in β.

Lemma 13. Let $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ be derivable. Then $\mathrm{dom}(\Gamma) \supseteq \mathrm{dom}(\beta)$.

Proof. Suppose, by contradiction, that $\exists x \in \mathrm{dom}(\beta)$ such as $x \notin \mathrm{dom}(\Gamma)$, and that e contains such x. This means that either x, $x = e'$, or both appears in e. In the former case, rule VAR in Section 6.3 apply to give a type to e. However, VAR requires x to be in Γ (i.e. $x : T' \in \Gamma$), but it does not by hypothesis. The latter case is similar: rule ASS applies, and it contains, as a premise, $\Gamma \vdash x : T'$, which in turn is handled by VAR. Hence, the type system cannot give a type to x, but this is a contradiction, since we assumed $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ as derivable. Hence, x must be in $\mathrm{dom}(\Gamma)$.

The same reasoning applies to a and c. □

We can now prove Theorem 3:

Theorem 3 (Progress)**.**
If $\langle \beta, \sigma, e \rangle$ is closed (Definition 13) and well-typed (i.e. $\exists \Gamma$ such that $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ is derivable), then either $e = v$, $e =$ revert, or $\exists (\beta', \sigma', e')$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$.

Proof. We prove this theorem by induction on the height of the judgment $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$.

Base cases These cases correspond to the axioms in Section 6.3. By rule CONFIGURATION, $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ implies $\Gamma \vdash \beta$ and $\Gamma \vdash \sigma$. These cases modify neither β nor σ, and hence we focus on the expression e (i.e. on the judgment $\Gamma \vdash e : T$).

- REF. The judgment is $\Gamma \vdash c : C$, and c is already a value.
- VAR. The judgment is $\Gamma \vdash x : T$. By hypothesis, we know $\Gamma \vdash x : T$, which implies, by rule VAR, $x : T \in \Gamma$, which means $x \in \mathrm{dom}(\Gamma)$. By Lemma 13, $x \in \mathrm{dom}(\Gamma) \Rightarrow x \in \mathrm{dom}(\beta)$, which makes $\beta(x)$ well-defined. By rule VAR in Section 5.5, $\exists e' = \beta(x)$ such that $\langle \beta, \sigma, x \rangle \longrightarrow \langle \beta, \sigma, \beta(x) \rangle$.
- TRUE. The judgment is $\Gamma \vdash$ true : bool, and the expression true is already a value.
- FALSE, NAT, ADDRESS, and UNIT. The judgments are $\Gamma \vdash$ false : bool, $\Gamma \vdash n$: uint, $\Gamma \vdash a$: address, or $\Gamma \vdash$ u : unit, respectively, and the expressions false, n, a, and u are already values.
- REVERT. The judgment is $\Gamma \vdash$ revert : T, and the expression is a revert.

Inductive cases Given a judgment J such that its derivation has height $k + 1$, we prove the inductive cases on the last rule used to derive J. We assume the theorem for the judgments with height at most k and we prove it for those with height $k + 1$.

- MAPPING. The judgment is $\Gamma \vdash M :$ mapping$(T_1 \Rightarrow T_2)$, and the expression M is already a value.
- FUN. The judgment is $\Gamma \vdash c.f : \tilde{T_1} \to T_2$, and the expression $c.f$ is already a value.

- BAL. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, \mathsf{balance}(e) \rangle : \mathsf{uint}$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e \rangle : \mathsf{address}$. By inductive hypothesis we know e is either a value, revert, or $\exists \beta', \sigma', e'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$. We distinguish the following cases:
  - if $e = v$ then, by Case 7 of Lemma 6, v is an address a. By rule BALANCE in Section 5.5, $\langle \beta, \sigma, \mathsf{balance}(a) \rangle \longrightarrow \langle \beta, \sigma, n \rangle$, where $\beta(a) = (C, \tilde{s:v}, n)$ is well-defined for what we said in Lemma 13: $\Gamma \vdash \langle \beta, \sigma, \mathsf{balance}(a) \rangle : \mathsf{uint}$ is well typed, and then $\exists c$ such that the pair $(c, a) \in \mathrm{dom}(\beta)$.
  - if $e = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{balance}(e) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, \mathsf{balance}(e) \rangle \longrightarrow \langle \beta', \sigma', \mathsf{balance}(e') \rangle$.
- ADDR. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, \mathsf{address}(e) \rangle : \mathsf{address}$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e \rangle : C$. By inductive hypothesis we know e is either a value, revert, or $\exists \beta', \sigma', e'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$. We distinguish the following cases:
  - if $e = v$ then, by Case 5 of Lemma 6, v is a contract reference c. By rule ADDRESS in Section 5.5, $\langle \beta, \sigma, \mathsf{address}(c) \rangle \longrightarrow \langle \beta, \sigma, a \rangle$, where $\hat{\beta}(c) = a$. The latter equivalence is well-defined by Lemma 13.
  - if $e = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{address}(e) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, \mathsf{address}(e) \rangle \longrightarrow \langle \beta', \sigma', \mathsf{address}(e') \rangle$.
- RETURN. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma \cdot a, \mathsf{return}\, e \rangle : T$ was derived from $\Gamma \vdash_k \langle \beta, \sigma \cdot a, e \rangle : T$. By inductive hypothesis we know e is either a value, revert, or $\exists \beta', \sigma', e'$ such that $\langle \beta, \sigma \cdot a, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$. We distinguish the following cases:
  - if $e = v$ then, by rule RETURN in Section 5.5, $\langle \beta, \sigma \cdot a, \mathsf{return}\, v \rangle \longrightarrow \langle \beta, \sigma, v \rangle$.
  - if $e = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma \cdot a, \mathsf{return\ revert} \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma \cdot a, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma \cdot a, \mathsf{return}\, e \rangle \longrightarrow \langle \beta', \sigma', \mathsf{return}\, e' \rangle$.
- IF. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rangle : T$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : \mathsf{bool}$, $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : T$, and $\Gamma \vdash_k \langle \beta, \sigma, e_3 \rangle : T$. By inductive hypothesis we know e_1 is either a value, a revert, or $\exists \beta', \sigma', e_1'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e_1' \rangle$. We distinguish the following cases:
  - if $e_1 = v$ then, by Case 1 of Lemma 6, v is either true or false. If it is true, by rule IF-TRUE in Section 5.5 $\langle \beta, \sigma, \mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3 \rangle \longrightarrow \langle \beta, \sigma, e_2 \rangle$. On the other hand, if it is false, by rule IF-FALSE in Section 5.5 $\langle \beta, \sigma, \mathsf{if\ false\ then}\ e_2\ \mathsf{else}\ e_3 \rangle \longrightarrow \langle \beta, \sigma, e_3 \rangle$.
  - if $e_1 = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{if\ revert\ then}\ e_2\ \mathsf{else}\ e_3 \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.

  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta', \sigma', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rangle \longrightarrow \langle \beta', \sigma', \mathsf{if}\ e_1'\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rangle$.

- SEQ. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1; e_2 \rangle : T_2$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : T_1$ and $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : T_2$. By inductive hypothesis we know e_1 is either a value, revert, or $\exists \beta', \sigma', e_1'$ such that $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta', \sigma', e_1' \rangle$. We distinguish the following cases:
  - if $e_1 = v$ then two scenarios are possible. First, if $\mathrm{Top}(\sigma) = \emptyset$, by rule SEQ-C in Section 5.5, $\langle \beta, \sigma, v; e_2 \rangle \longrightarrow \langle \beta, \beta, e_2 \rangle$. Secondly, if $\exists a$ such that $\mathrm{Top}(\sigma) = a$ then, by rule SEQ in Section 5.5, $\langle \beta, \sigma, v; e_2 \rangle \longrightarrow \langle \beta, \sigma, e_2 \rangle$.
  - if $e_1 = \mathsf{revert}$ then, again, two scenarios are possible. First, if $\mathrm{Top}(\sigma) = \emptyset$ (and, consequently, $\exists \beta_0$ such that $\sigma = \beta_0$), by rule SEQ-R in Section 5.5, $\langle \beta, \sigma, \mathsf{revert}; e_2 \rangle \longrightarrow \langle \beta_0, \beta_0, \mathsf{revert} \rangle$. Secondly, if $\exists a$ such that $\mathrm{Top}(\sigma) = a$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{revert}; e_2 \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta', \sigma', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, e_1; e_2 \rangle \longrightarrow \langle \beta', \sigma', e_1'; e_2 \rangle$.

- DECL. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, T_1\ x = e_1; e_2 \rangle : T_2$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : T_1$ and $\emptyset, x : T_1 \vdash_k \langle \beta, \sigma, e_2 \rangle : T_2$. By inductive hypothesis, e_1 is either a value, revert, or $\exists \beta_1', \sigma_1', e_1'$ such that $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$. We distinguish the following cases:
  - if $e_1 = v_1$ then, by rule DECL in Section 5.5, $\langle \beta, \sigma, T_1\ x = v_1; e_2 \rangle \longrightarrow \langle \beta \cdot [x \mapsto v], \sigma, v_1; e_2 \rangle$. Note that the premise $x \notin \mathrm{dom}(\beta)$ is always satisfied since variables can be renamed by α-equivalence.
  - if $e_1 = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, T_1\ x = \mathsf{revert}; e_2 \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert}; e_2 \rangle$. Note that this case does not take into account e_2 at all.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta', \sigma', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, T_1\ x = e_1; e_2 \rangle \longrightarrow \langle \beta', \sigma', T_1\ x = e_1'; e_2 \rangle$.

- ASS. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, x = e \rangle : T$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, x \rangle : T$ and $\Gamma \vdash_k \langle \beta, \sigma, e \rangle : T$. By inductive hypothesis, e is either a value, revert, or $\exists \beta', \sigma', e'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$. We distinguish the following cases:
  - if $e = v$ then, by rule ASS in Section 5.5, $\langle \beta, \sigma, x = v \rangle \longrightarrow \langle \beta[x \mapsto v], \sigma, v \rangle$. Note that $x \in \mathrm{dom}(\beta)$ is always satisfied for what we said in Lemma 13.
  - if $e = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, x = \mathsf{revert} \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, x = e \rangle \longrightarrow \langle \beta', \sigma', x = e' \rangle$.

- MAPPSEL. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1[e_2] \rangle : T_2$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : T_1$. By inductive hypothesis we know e_1 is either a value, revert, or $\exists \beta_1', \sigma_1', e_1'$ such that $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$. The same reasoning applies to e_2, thus e_2 is either a value, revert, or $\exists \beta_2', \sigma_2', e_2'$ such that $\langle \beta, \sigma, e_2 \rangle \longrightarrow \langle \beta_2', \sigma_2', e_2' \rangle$. We distinguish the following cases:

  - if $e_1 = v_1$ and $e_2 = v_2$ then, by Case 4 of Lemma 6, v_1 is a total function M from T_1 to T_2. By rule MAPPSEL in Section 5.5, $\langle \beta, \sigma, M[v_2] \rangle \longrightarrow \langle \beta, \sigma, M(v_2) \rangle$. Note that $M(v_2)$ is always well-defined because M is a total function.
  - if $e_1 = v_1$ and $e_2 = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, v_1[\mathsf{revert}] \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $e_1 = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{revert}[e_2] \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, e_1[e_2] \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1'[e_2] \rangle$. Note we do not have to specify what e_2 is.
  - if $e_1 = v_1$ and $\langle \beta, \sigma, e_2 \rangle \longrightarrow \langle \beta_2', \sigma_2', e_2' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, v_1[e_2] \rangle \longrightarrow \langle \beta_2', \sigma_2', v_1[e_2'] \rangle$.

- STATESEL. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e.s_i \rangle : T_i$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e \rangle : C$, where C is such that $\mathrm{SV}(C) = \tilde{T} s$ and $s_i \in \tilde{s}$. By inductive hypothesis we know e is either a value, revert, or $\exists \beta', \sigma', e'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$. We distinguish the following cases:
  - if $e = v$ then, by Case 5 of Lemma 6, v is a contract reference c. By rule STATESEL in Section 5.5, $\langle \beta, \sigma, c.s_i \rangle \longrightarrow \langle \beta, \sigma, v_i \rangle$, where $\beta(c) = (C, \tilde{sv}, n)$ and $c.s_i = v_i$. Note that $\beta(c) = (C, \tilde{sv}, n)$ is well-defined by Lemma 13.
  - if $e = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{revert}.s_i \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, e.s_i \rangle \longrightarrow \langle \beta', \sigma', e'.s_i \rangle$.

- MAPPASS. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1[e_2 \to e_3] \rangle : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : T_1$, and $\Gamma \vdash_k \langle \beta, \sigma, e_3 \rangle : T_2$. By inductive hypothesis we know e_1 is either a value, revert, or $\exists \beta_1', \sigma_1', e_1'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$. The same reasoning applies to e_2 and e_3. We distinguish the following cases:
  - if $e_1 = v_1$, $e_2 = v_2$, and $e_3 = v_3$ then, by Case 4 of Lemma 6, v_1 is a total function M from T_1 to T_2. By rule MAPPASS in Section 5.5, $\langle \beta, \sigma, M[v_2 \to v_3] \rangle \longrightarrow \langle \beta, \sigma, M' \rangle$, where $M' = M \backslash \{(v_1, M(v_1))\} \cup \{(v_1, v_2)\}$.
  - if either $e_2 = \mathsf{revert}$ or $e_3 = \mathsf{revert}$, with $e_1 = M$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, M[\mathsf{revert} \to v_3] \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$ or $\langle \beta, \sigma, M[v_2 \to \mathsf{revert}] \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$, respectively.
  - if $e_1 = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{revert}[e_2 \to e_3] \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, e_1[e_2 \to e_3] \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1'[e_2 \to e_3] \rangle$.
  - if $e_1 = v_1$ and $\langle \beta, \sigma, e_2 \rangle \longrightarrow \langle \beta_2', \sigma_2', e_2' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, v_1[e_2 \to e_3] \rangle \longrightarrow \langle \beta_2', \sigma_2', v_1[e_2' \to e_3] \rangle$.

  - if $e_1 = v_1$, $e_2 = v_2$, and $\langle \beta, \sigma, e_3 \rangle \longrightarrow \langle \beta'_3, \sigma'_3, e'_3 \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, v_1[v_2 \to e_3] \rangle \longrightarrow \langle \beta'_3, \sigma'_3, v_1[v_2 \to e'_3] \rangle$.

- STATEASS. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1.s_i = e_2 \rangle : T$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1.s_i \rangle : T$ and $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : T$. By inductive hypothesis we know e_1 is either a value, revert, or $\exists \beta'_1, \sigma'_1, e'_1$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta'_1, \sigma'_1, e'_1 \rangle$. The same reasoning applies to e_2. We distinguish the following cases:

  - if $e_1 = v_1$ and $e_2 = v_2$ then, by Case 21 of Lemma 2 we know $\Gamma \vdash v_1 : C$ can be derived. Hence, by Case 5 of Lemma 6, v_1 is a contract reference c. By rule STATEASS in Section 5.5, $\langle \beta, \sigma, c.s_i = v_2 \rangle \longrightarrow \langle \beta, \sigma, v_2 \rangle$, where $\beta(c) = (C, \tilde{sv}, n)$ and $c.s_i = v_i$. Note that $\beta(c)$ is well-defined by Lemma 13.
  - if $e_1 = v_1$ and $e_2 = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, e_1.s = \mathsf{revert} \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $e_1 = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{revert}.s_i = e_2 \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta'_1, \sigma'_1, e'_1 \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, e_1.s_i = e_2 \rangle \longrightarrow \langle \beta'_1, \sigma'_1, e'_1.s_i = e_2 \rangle$.
  - if $e_1 = v_1$ and $\langle \beta, \sigma, e_2 \rangle \longrightarrow \langle \beta'_2, \sigma'_2, e'_2 \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, v_1.s_i = e_2 \rangle \longrightarrow \langle \beta'_2, \sigma'_2, v_1.s_i = e'_2 \rangle$.

- NEW. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(e_1)(\tilde{e}) \rangle : C$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : \mathsf{uint}$, and $\Gamma \vdash_k \langle \beta, \sigma, \tilde{e} \rangle : \tilde{T}$. By inductive hypothesis we know e_1 is either a value, revert, or $\exists \beta'_1, \sigma'_1, e'_1$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta'_1, \sigma'_1, e'_1 \rangle$. The same reasoning applies to the tuple $\tilde{e}$: it is either a tuple of values ($\tilde{v}$), a tuple of values and expressions separated by a revert (i.e. $v_i^{\{0 \le i < m\}}, \mathsf{revert}, e_j^{\{m < j \le n\}}$), or it is in the form $v_i^{\{0 \le i < m\}}, e_j^{\{m \le j \le n\}}$ and $\exists \beta'_i, \sigma'_i, e'_j$ such that $\langle \beta, \sigma, v_i^{\{0 \le i < m\}}, e_j^{\{m \le j \le n\}} \rangle \longrightarrow \langle \beta'_i, \sigma'_i, v_i^{\{0 \le i < m\}}, e_j'^{\{m \le j \le n\}} \rangle$. We distinguish the following cases:

  - if $e_1 = v_1$ and $\tilde{e} = \tilde{v}$ then, by Case 2 of Lemma 6, $v_1 = n$. Three scenarios are possible here. If $\mathrm{Top}(\sigma) \neq \emptyset$, then either NEW-1 or NEW-R (as defined in Section 5.5) applies. If it is NEW-1 (note the check about the length of the tuples is true since it is also a premise of the rule NEW in Section 6.3) then the balance of the contract corresponding to $\mathrm{Top}(\sigma)$ is enough to create a new contract with an initial balance of n. Hence, $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) \cdot$ $\cdot [(c, a) \mapsto (C, \tilde{sv}, n)], \sigma, c \rangle$, where (c, a) is a fresh pair identifying the new contract in β. On the other hand, if NEW-R applies, then the balance of the contract corresponding to $\mathrm{Top}(\sigma)$ is not enough. Hence, $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$. Lastly, if $\mathrm{Top}(\sigma) = \emptyset$, then NEW-2 (as defined in Section 5.5) applies. Hence, $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle$ $\longrightarrow \langle \beta \cdot [(c, a) \mapsto (C, \tilde{sv}, n)], \sigma, c \rangle$, where (c, a) is, as before, a fresh pair identifying the new contract in β.
  - if either $e_1 = \mathsf{revert}$ or $e_1 = v_1$ and $\tilde{e} = v_i^{\{0 \le i < m\}}, \mathsf{revert}, e_j^{\{m < j \le n\}}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(\mathsf{revert})(\tilde{e}) \rangle \longrightarrow$ $\langle \beta, \sigma, \mathsf{revert} \rangle$ or $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(v_1)(\tilde{e} = v_i^{\{0 \le i < m\}}, \mathsf{revert}, e_j^{\{m < j \le n\}}) \rangle$ $\longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$, respectively.

  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(e_1)(\tilde{e}) \rangle \longrightarrow \langle \beta_1', \sigma_1', \mathsf{new}\ C.\mathsf{value}(e_1')(\tilde{e}) \rangle$.
  - if $e_1 = v_1, \tilde{e} = v_i^{\{0 \leq i < m\}}, e_j^{\{m \leq j \leq n\}}$, and $\langle \beta, \sigma, v_i^{\{0 \leq i < m\}}, e_j^{\{m \leq j \leq n\}} \rangle \longrightarrow \langle \beta_i', \sigma_i', v_i^{\{0 \leq i < m\}}, e_j'^{\{m \leq j \leq n\}} \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(v_1)(v_i^{\{0 \leq i < m\}}, e_j^{\{m \leq j \leq n\}}) \rangle \longrightarrow \langle \beta_i', \sigma_i', \mathsf{new}\ C.\mathsf{value}(v_1)(v_i^{\{0 \leq i < m\}}, e_j'^{\{m \leq j \leq n\}}) \rangle$.

- CONTRRETR. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, C(e) \rangle : C$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e \rangle : \mathsf{address}$. By inductive hypothesis, e is either a value, revert or $\exists \beta', \sigma', e'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$. We distinguish the following cases:

  - if $e = v$ then, by Case 7 of Lemma 6, v is an address a. Two are the possible scenarios. First, $\beta^C(a) = C$ (well-defined by Lemma 13) and rule CONTRRETR of Section 5.5 applies. a corresponds to a reference c to a contract C. Hence, $\langle \beta, \sigma, C(a) \rangle \longrightarrow \langle \beta, \sigma, c \rangle$. Secondly, $\beta^C(a) = C'$, with $C' \neq C$. In this case rule CONTRRETR-R of Section 5.5 applies, and $\langle \beta, \sigma, C(a) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$. Again, $\beta^C(a) = C'$ is well-defined by Lemma 13.
  - if $e = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, C(\mathsf{revert}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, C(e) \rangle \longrightarrow \langle \beta', \sigma', C(e') \rangle$.

- TRANSFER. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1.\mathsf{transfer}(e_2) \rangle : \mathsf{unit}$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : \mathsf{address}$ and $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : \mathsf{uint}$. By inductive hypothesis, e_1 is either a value, revert or $\exists \beta_1', \sigma_1', e_1'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$. The same reasoning applies to e_2. We distinguish the following cases:

  - if $e_1 = v_1$ and $e_2 = v_2$ then, by Case 7 and Case 2 of Lemma 6, v_1 is an address a and v_2 is a non-negative integer n. Let $\beta(a) = (C, \tilde{sv}, m)$, well-defined by Lemma 13. Two are the possible scenarios. First, if $m \geq n$ rule TRANSFER in Section 5.5 applies and $\langle \beta, \sigma, a.\mathsf{transfer}(n) \rangle \longrightarrow \langle \beta', \sigma \cdot a, e\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\} \rangle$, where $\beta' = \mathrm{uptbal}(\mathrm{uptbal}(\beta, a, n), \mathrm{Top}(\sigma), -n)$, $\hat{\beta}(c) = a$, and e is either the body of the callback function of C, if any, or $\mathsf{return\ revert}$. Secondly, if $m < n$ then TRANSFER-R applies and we obtain $\langle \beta, \sigma, a.\mathsf{transfer}(n) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if either $e_1 = \mathsf{revert}$ or $e_1 = v_1$ and $e_2 = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{revert}.\mathsf{transfer}(e_2) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$ or $\langle \beta, \sigma, v_1.\mathsf{transfer}(\mathsf{revert}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$, respectively.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, e_1.\mathsf{transfer}(e_2) \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1'.\mathsf{transfer}(e_2) \rangle$.
  - if $e_1 = v_1$ and $\langle \beta, \sigma, e_2 \rangle \longrightarrow \langle \beta_2', \sigma_2', e_2' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, v_1.\mathsf{transfer}(e_2) \rangle \longrightarrow \langle \beta_2', \sigma_2', v_1.\mathsf{transfer}(e_2') \rangle$.

  Note we did not make any reasoning about $\mathrm{Top}(\sigma)$. As said, we assumed $\mathrm{Top}(\sigma) \neq \emptyset$.

- CALL. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1.f.\mathsf{value}(e_2)(\tilde{e}) \rangle : T_2$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : C$, $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : \mathsf{uint}$, and $\Gamma \vdash_k \langle \beta, \sigma, \tilde{e} \rangle : \tilde{T_1}$, where $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$ and $|\tilde{e}| = |\tilde{T_1}|$. By inductive hypothesis, e_1 is either a value, revert or $\exists \beta', \sigma', e_1'$ such that $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$. The same reasoning applies to e_2 and to the tuple $\tilde{e}$. The former is either a value, revert or $\exists \beta'', \sigma'', e_2'$ such that $\langle \beta, \sigma, e_2 \rangle \longrightarrow \langle \beta_2', \sigma_2', e_2' \rangle$. The latter is either a tuple of values ($\tilde{v}$), a tuple of values and expressions separated by a revert (i.e. $v_i^{\{0 \le i < m\}}, \mathsf{revert}, e_j^{\{m < j \le n\}}$), or it is in the form $v_i^{\{0 \le i < m\}}, e_j^{\{m \le j \le n\}}$ and $\exists \beta_i', \sigma_i', e_j'$ such that $\langle \beta, \sigma, v_i^{\{0 \le i < m\}}, e_j^{\{m \le j \le n\}} \rangle \longrightarrow \langle \beta_i', \sigma_i', v_i^{\{0 \le i < m\}}, e_j'^{\{m \le j \le n\}} \rangle$. We distinguish the following cases:
  - $e_1 = v_1$, $e_2 = v_2$, and $\tilde{e} = \tilde{v}$. By Case 5 and Case 2 of Lemma 6, v_1 is a contract reference c and v_2 is a non-negative integer n. Let $\beta(c) = (C, \tilde{s:v}, m)$, well-defined by Lemma 13. Two are the possible scenarios. First, if $m \ge n$, then rule CALL in Section 5.5 applies. It retrieves f's body, e, and its formal parameters $\tilde{x}$, defines $\beta' = \mathrm{uptbal}(\mathrm{uptbal}(\beta, a, n), \mathrm{Top}(\sigma), -n) \cdot [x_i \mapsto v_i \ ^{x_i \in \tilde{x}, v_i \in \tilde{v}}]$ as explained in Chapter 5, and then evolves as follows: $\langle \beta, \sigma, c.f.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta', \sigma \cdot a, e\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\} \rangle$. Secondly, if $m < n$, then rule CALL-R in Section 5.5 applies, and $\langle \beta, \sigma, c.f.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if either $e_1 = \mathsf{revert}$, $e_1 = v_1$ and $e_2 = \mathsf{revert}$, or $e_1 = v_1$, $e_2 = v_2$ and $\tilde{e} = v_i^{\{0 \le i < m\}}, \mathsf{revert}, e_j^{\{m < j \le n\}}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{revert}.f.\mathsf{value}(e_2)(\tilde{e}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$, $\langle \beta, \sigma, v_1.f.\mathsf{value}(\mathsf{revert})(\tilde{e}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$, or $\langle \beta, \sigma, v_1.f.\mathsf{value}(v_2)(\tilde{e} = v_i^{\{0 \le i < m\}}, \mathsf{revert}, e_j^{\{m < j \le n\}}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$, respectively.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, e_1.f.\mathsf{value}(e_2)(\tilde{e}) \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1'.f.\mathsf{value}(e_2)(\tilde{e}) \rangle$.
  - if $e_1 = v_1$ and $\langle \beta, \sigma, e_2 \rangle \longrightarrow \langle \beta_2', \sigma_2', e_2' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, v_1.f.\mathsf{value}(e_2)(\tilde{e}) \rangle \longrightarrow \langle \beta_2', \sigma_2', v_1.f.\mathsf{value}(e_2')(\tilde{e}) \rangle$.
  - if $e_1 = v_1$, $e_2 = v_2$, $\tilde{e} = v_i^{\{0 \le i < m\}}, e_j^{\{m \le j \le n\}}$, and $\langle \beta, \sigma, v_i^{\{0 \le i < m\}}, e_j^{\{m \le j \le n\}} \rangle \longrightarrow \langle \beta_i', \sigma_i', v_i^{\{0 \le i < m\}}, e_j'^{\{m \le j \le n\}} \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, v_1.f.\mathsf{value}(v_2)(v_i^{\{0 \le i < m\}}, e_j^{\{m \le j \le n\}}) \rangle \longrightarrow \langle \beta_i', \sigma_i', v_1.f.\mathsf{value}(v_2)(v_i^{\{0 \le i < m\}}, e_j'^{\{m \le j \le n\}}) \rangle$.
- CALLTOPLEVEL. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) \rangle : T_2$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1.f.\mathsf{value}(e_2)(\tilde{e}) \rangle : T_2$ and $\Gamma \vdash_k \langle \beta, \sigma, e_3 \rangle : \mathsf{address}$. This case is very similar to the previous one, with the only addition of the parameter e_3 and considering CALLTOPLEVEL and CALLTOPLEVEL-R instead of CALL and CALL-R. What we said and proved previously applies here, too. We can apply the inductive hypothesis on e_3. If it is a value, then, by Case 7 of Lemma 6, it is an address a. If also e_1, e_2, and $\tilde{e}$ are values, either CALLTOPLEVEL or CALLTOPLEVEL-R (as defined in Section 5.5) applies, according to the balance of the caller. When it is a revert, rule REVERT in Section 5.5 applies. Lastly, when $e_1 = v_1$ and $e_2 = v_2$ and $\exists \beta', \sigma', e_3'$ such that $\langle \beta, \sigma, e_3 \rangle \longrightarrow \langle \beta', \sigma', e_3' \rangle$ we obtain $\langle \beta, \sigma, v_1.f.\mathsf{value}(v_2).\mathsf{sender}(e_3)(\tilde{e}) \rangle \longrightarrow \langle \beta', \sigma', v_1.f.\mathsf{value}(v_2).\mathsf{sender}(e_3')(\tilde{e}) \rangle$.

- CALLVALUE. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1.\mathsf{value}(e_2)(\tilde{e}) \rangle : T_2$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : \tilde{T_1} \to T_2$, $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : \mathsf{uint}$, and $\Gamma \vdash_k \langle \beta, \sigma, \tilde{e} \rangle : \tilde{T_1}$, all such that the height of their derivations is k. Also this case is similar to CALL, and the same rules (CALL and CALL-R of Section 5.5) are applied. The only difference is the expression e_1. By inductive hypothesis, it is either a value, revert or $\exists \beta', \sigma', e_1'$ such that $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta', \sigma', e_1' \rangle$. If it is a value, by Case 6 of Lemma 6, it is a function pointer $c.f$. From this point on, every case is equal to those discussed for CALL, with the application of the same rules CALL and CALL-R.

□

E.5 Subject Reduction Theorem

To prove Theorem 4 we shall need an additional lemma formalizing type preservation into evaluation contexts E.

Lemma 14 (Type preservation into E)**.** *If* $\Gamma \vdash E[e] : T$ *then* $\exists T'$ *such that* $\Gamma \vdash e : T'$*, and* $\forall e' \mid \Gamma \vdash e' : T' \Rightarrow \Gamma \vdash E[e'] : T$.

Proof. We prove this lemma by on the definition of evaluation contexts given in Section 6.3.

- $E = []$. This case is trivial: $[e] = e$, thus $\Gamma \vdash e : T$ implies trivially $T' = T$. The second conclusion follows immediately.

- $E = \mathsf{balance}(E')$, and $E[e] = \mathsf{balance}(e)$. In this case $\Gamma \vdash \mathsf{balance}(e) : T$ can be derived by hypothesis. By Case 11 of Lemma 2, $T = \mathsf{uint}$ and $\Gamma \vdash e : \mathsf{address}$, which is part of what we aimed to prove (setting $T' = \mathsf{address}$). Let e' be any expression such that $\Gamma \vdash e' : \mathsf{address}$, and consider rule BAL in Section 6.3: $\Gamma \vdash e' : \mathsf{address}$ satisfies its premise, and thus we can derive $\Gamma \vdash \mathsf{balance}(e') : \mathsf{uint}$, which is the same as $\Gamma \vdash \mathsf{balance}[e'] : \mathsf{uint}$.

- $E = \mathsf{address}(E')$, and $E[e] = \mathsf{address}(e)$. In this case $\Gamma \vdash \mathsf{address}(e) : T$ can be derived by hypothesis. By Case 12 of Lemma 2, $T = \mathsf{address}$ and $\Gamma \vdash e : C$, which is part of what we aimed to prove (setting $T' = C$). Let e' be any expression such that $\Gamma \vdash e' : C$, and consider rule ADDR in Section 6.3: $\Gamma \vdash e' : C$ satisfies its premise, and thus we can derive $\Gamma \vdash \mathsf{address}(e') : \mathsf{address}$, which is the same as $\Gamma \vdash \mathsf{address}[e'] : \mathsf{address}$.

- $E = E'.s$, and $E[e] = e.s$. In this case $\Gamma \vdash e.s : T$ can be derived by hypothesis. By Case 14 of Lemma 2, $\Gamma \vdash e : C$, which is part of what we aimed to prove (setting $T' = C$). Let e' be any expression such that $\Gamma \vdash e' : C$, and consider rule STATESEL in Section 6.3: $\Gamma \vdash e' : C$, together with the conclusions of Case 14 of Lemma 2 satisfy its premises, and thus we can derive $\Gamma \vdash e'.s : T_i$, which is the same as $\Gamma \vdash E[e'] : \mathsf{uint}$.

- $E = E'.\mathsf{transfer}(e_1)$, and $E[e] = e.\mathsf{transfer}(e_1)$. In this case $\Gamma \vdash e.\mathsf{transfer}(e_1) : T$ can be derived by hypothesis. By Case 15 of Lemma 2, $T = \mathsf{unit}$, $\Gamma \vdash e : \mathsf{address}$, and $\Gamma \vdash e_1 : \mathsf{uint}$. Setting $T' = \mathsf{address}$, $\Gamma \vdash e : \mathsf{address}$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : \mathsf{address}$, and consider rule TRANSFER in Section 6.3: $\Gamma \vdash e' : \mathsf{address}$ and $\Gamma \vdash e_1 : \mathsf{uint}$

satisfy its premises, and thus we can derive $\Gamma \vdash e'.\mathsf{transfer}(e_1) : \mathsf{unit}$, which is what we aimed to prove since $(E.\mathsf{transfer}(e_1))[e'] = e'.\mathsf{transfer}(e_1)$.

- $E = a.\mathsf{transfer}(E')$, and $E[e] = a.\mathsf{transfer}(e)$. In this case $\Gamma \vdash a.\mathsf{transfer}(e) : T$ can be derived by hypothesis. By Case 15 of Lemma 2, $T = \mathsf{unit}$, $\Gamma \vdash a : \mathsf{address}$, and $\Gamma \vdash e : \mathsf{uint}$. Setting $T' = \mathsf{uint}$, $\Gamma \vdash e : \mathsf{uint}$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : \mathsf{uint}$, and consider rule TRANSFER in Section 6.3: $\Gamma \vdash a : \mathsf{address}$ and $\Gamma \vdash e' : \mathsf{uint}$ satisfy its premises, and thus we can derive $\Gamma \vdash a.\mathsf{transfer}(e') : \mathsf{unit}$, which is the same as $\Gamma \vdash E[e'] : \mathsf{unit}$.

- $E = \mathsf{new}\ C.\mathsf{value}(E')(\tilde{e})$, and $E[e] = \mathsf{new}\ C.\mathsf{value}(e)(\tilde{e})$. In this case $\Gamma \vdash \mathsf{new}\ C.\mathsf{value}(e)(\tilde{e}) : T$ can be derived by hypothesis. By Case 16 of Lemma 2, $T = C$, $\Gamma \vdash e : \mathsf{uint}$, and $\Gamma \vdash \tilde{e} : \tilde{T}$, where $\mathrm{sv}(C) = \tilde{T}\,s$. Setting $T' = \mathsf{uint}$, $\Gamma \vdash e : \mathsf{uint}$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : \mathsf{uint}$, and consider rule NEW in Section 6.3: $\Gamma \vdash e' : \mathsf{uint}$, together with the conclusions of Case 16 of Lemma 2, satisfy its premises, and thus we can derive $\Gamma \vdash \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : C$, which is the same as $\Gamma \vdash E[e'] : C$.

- $E = \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}, E', \tilde{e})$, and $E[e] = \mathsf{new}\ C.\mathsf{value}(a)(\tilde{v}, e, \tilde{e})$. In this case $\Gamma \vdash \mathsf{new}\ C.\mathsf{value}(a)(\tilde{v}, e, \tilde{e}) : T$ can be derived by hypothesis. By Case 16 of Lemma 2, $T = C$, $\Gamma \vdash n : \mathsf{uint}$, and $\Gamma \vdash v_j^{\{0 \leq j < i\}}, e_i, e_j^{\{i < j \leq m\}} : \tilde{T}$, where $\mathrm{sv}(C) = \tilde{T}\,s$. Setting $T' = T_i$, $\Gamma \vdash e_i : T_i$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : T_i$, and consider rule NEW in Section 6.3: $\Gamma \vdash a : \mathsf{uint}$, together with $\Gamma \vdash v_j^{\{0 \leq j < i\}}, e', e_j^{\{i < j \leq m\}} : \tilde{T}$, satisfy its premises, and thus we can derive $\Gamma \vdash \mathsf{new}\ C.\mathsf{value}(a)(\tilde{v}, e, \tilde{e}) : C$, which is the same as $\Gamma \vdash E[e'] : C$.

- $E = C(E')$, and $E[e] = C(e)$. In this case $\Gamma \vdash C(e) : T$ can be derived by hypothesis. By Case 17 of Lemma 2, $T = C$ and $\Gamma \vdash e : \mathsf{address}$, which is part of what we aimed to prove (setting $T' = \mathsf{address}$). Let e' be any expression such that $\Gamma \vdash e' : \mathsf{address}$, and consider rule CONTRRETR in Section 6.3: $\Gamma \vdash e' : \mathsf{address}$ satisfies its premise, and thus we can derive $\Gamma \vdash C(e') : C$, which is the same as $\Gamma \vdash \mathsf{C}[e'] : C$, which is indeed the same as $\Gamma \vdash E[e'] : C$.

- $E = E'; e_1$, and $E[e] = e; e_1$. In this case $\Gamma \vdash e; e_1 : T$ can be derived by hypothesis. By Case 18 of Lemma 2, $\exists T_1$ such that $\Gamma \vdash e : T_1$ and $\Gamma \vdash e_1 : T$. Setting $T' = T_1$, $\Gamma \vdash e : T_1$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : T_1$, and consider rule SEQ in Section 6.3: $\Gamma \vdash e' : T_1$ and $\Gamma \vdash e_1 : T$ satisfy its premises, and thus we can derive $\Gamma \vdash e'; e : T$, which is what we aimed to prove since $(E; e_1)[e'] = e'; e_1$.

- $E = E.f.\mathsf{value}(e_1)(\tilde{e})$, and $E[e] = e.f.\mathsf{value}(e_1)(\tilde{e})$. In this case $\Gamma \vdash e.f.\mathsf{value}(e_1)(\tilde{e}) : T$ can be derived by hypothesis. By Case 24 of Lemma 2, $\exists \tilde{T_1}, T_2$ such that $T = \tilde{T_1} \to T_2$, $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$, $\Gamma \vdash e : C$, $\Gamma \vdash e_1 : \mathsf{uint}$, and $\Gamma \vdash \tilde{e} : \tilde{T}$. Setting $T' = C$, $\Gamma \vdash e : C$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : C$, and consider rule CALL in Section 6.3: $\Gamma \vdash e' : C$, together with the conclusions of Case 24 of Lemma 2, satisfy its premises, and thus we can derive $\Gamma \vdash e.f.\mathsf{value}(e_1)(\tilde{e}) : T_2$, which is the same as $\Gamma \vdash E[e'] : T_2$.

- $E = c.f.\mathsf{value}(E)(\tilde{e})$, and $E[e] = c.f.\mathsf{value}(e)(\tilde{e})$. In this case $\Gamma \vdash c.f.\mathsf{value}(e)(\tilde{e}) : T$ can be derived by hypothesis. By Case 24 of Lemma 2, $\exists \tilde{T_1}, T_2$ such that $T = \tilde{T_1} \to T_2$, $\text{ftype}(C, f) = \tilde{T_1} \to T_2$, $\Gamma \vdash c : C$, $\Gamma \vdash e : \mathsf{uint}$, and $\Gamma \vdash \tilde{e} : \tilde{T}$. Setting $T' = \mathsf{uint}$, $\Gamma \vdash e : \mathsf{uint}$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : \mathsf{uint}$, and consider rule CALL in Section 6.3: $\Gamma \vdash e' : \mathsf{uint}$, together with the conclusions of Case 24 of Lemma 2, satisfy its premises, and thus we can derive $\Gamma \vdash c.f.\mathsf{value}(e')(\tilde{e}) : T_2$, which is the same as $\Gamma \vdash E[e'] : T_2$.

- $E = c.f.\mathsf{value}(n)(\tilde{v}, E, \tilde{e})$, and $E[e] = c.f.\mathsf{value}(n)(\tilde{v}, e, \tilde{e})$. In this case $\Gamma \vdash c.f.\mathsf{value}(n)(\tilde{v}, e, \tilde{e}) : T$ can be derived by hypothesis. By Case 24 of Lemma 2, $\exists \tilde{T_1}, T_2$ such that $T = \tilde{T_1} \to T_2$, $\text{ftype}(C, f) = \tilde{T_1} \to T_2$, $\Gamma \vdash c : C$, $\Gamma \vdash n : \mathsf{uint}$, and $\Gamma \vdash c.f.\mathsf{value}(n)(v_j^{\{0 \le j < i\}}, e_i, e_j^{\{i < j \le m\}}) : \tilde{T}$. Setting $T' = T_i$, $\Gamma \vdash e_i : T_i$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : T_i$, and consider rule CALL in Section 6.3: we know $\Gamma \vdash e' : T_i$, and hence we can replace e_i in $v_j^{\{0 \le j < i\}}, e_i, e_j^{\{i < j \le m\}}$ with e', in such a way that we obtain $\Gamma \vdash v_j^{\{0 \le j < i\}}, e', e_j^{\{i < j \le m\}} : \tilde{T}$. This, together with the other conclusions of Case 24 of Lemma 2, satisfy the premises of CALL, and thus we can derive $\Gamma \vdash c.f.\mathsf{value}(n)(v_j^{\{0 \le j < i\}}, e', e_j^{\{i < j \le m\}}) : T_2$, which is the same as $\Gamma \vdash E[e'] : T_2$.

- $E = E.\mathsf{value}(e_1)(\tilde{e})$, and $E[e] = e.\mathsf{value}(e_1)(\tilde{e})$. In this case $\Gamma \vdash e.\mathsf{value}(e_1)(\tilde{e}) : T$ can be derived by hypothesis. By Case 25 of Lemma 2, $\exists \tilde{T_1}, T_2$ such that $T = \tilde{T_1} \to T_2$, $\Gamma \vdash e : \tilde{T_1} \to T_2$, $\Gamma \vdash e_1 : \mathsf{uint}$, and $\Gamma \vdash \tilde{e} : \tilde{T}$. Setting $T' = \tilde{T_1} \to T_2$, $\Gamma \vdash e : \tilde{T_1} \to T_2$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : \tilde{T_1} \to T_2$, and consider rule CALLVALUE in Section 6.3: $\Gamma \vdash e' : \tilde{T_1} \to T_2$, together with the conclusions of Case 25 of Lemma 2, satisfy its premises, and thus we can derive $\Gamma \vdash e'.\mathsf{value}(e_1)(\tilde{e}) : T_2$, which is the same as $\Gamma \vdash E[e'] : T_2$.

- The cases with $E = E'.f.\mathsf{value}(e_1).\mathsf{sender}(e_2)(\tilde{e})$, $E = c.f.\mathsf{value}(E').\mathsf{sender}(e_1)(\tilde{e})$, and $E = E.a.\mathsf{value}(c).\mathsf{sender}(n)(\tilde{v}, E', \tilde{e})$ are the same as discussed above for call. The only difference relies on the additional hypothesis about the sender. For what concerns these three cases, the sender's expression (respectively e_2, e_1 and a) is just an hypothesis used to derive the second conclusion of this lemma. The remaining case, where the sender is actually the target expression, is discussed below.

- $E = c.f.\mathsf{value}(n).\mathsf{sender}(E')(\tilde{e})$, and $E[e] = c.f.\mathsf{value}(n).\mathsf{sender}(e)(\tilde{e})$. In this case $\Gamma \vdash c.f.\mathsf{value}(n).\mathsf{sender}(E)(\tilde{e}) : T$ can be derived by hypothesis. By Case 26 of Lemma 2, $\exists \tilde{T_1}, T_2$ such that $T = \tilde{T_1} \to T_2$, $\text{ftype}(C, f) = \tilde{T_1} \to T_2$, $\Gamma \vdash c : C$, $\Gamma \vdash n : \mathsf{uint}$, $\Gamma \vdash e : \mathsf{address}$, and $\Gamma \vdash \tilde{e} : \tilde{T}$. Setting $T' = \mathsf{address}$, $\Gamma \vdash e : \mathsf{address}$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : \mathsf{address}$, and consider rule CALLTOPLEVEL in Section 6.3: $\Gamma \vdash e' : \mathsf{address}$ satisfies its first premise. The second one refers back to CALL, whose premises are satisfied by the conclusions of Case 24 of Lemma 2. Hence, we can derive $\Gamma \vdash c.f.\mathsf{value}(n).\mathsf{sender}(e)(\tilde{e}) : T_2$.

- $E = (T_1\ x = E'; e_1)$, and $E[e] = (T_1\ x = e; e_1)$. In this case $\Gamma \vdash T_1\ x = e; e_1 : T$ can be derived by hypothesis. By Case 19 of Lemma 2, $\Gamma \vdash e : T_1$ and $\Gamma, x : T_1 \vdash e_1 : T$. Setting $T' = T_1$, $\Gamma \vdash e : T_1$ proves the first conclusion of

this lemma. Let e' be any expression such that $\Gamma \vdash e' : T_1$, and consider rule DECL in Section 6.3: $\Gamma \vdash e' : T_1$ and $\Gamma, x : T_1 \vdash e_1 : T$ satisfy its premises, and thus we can derive $\Gamma \vdash T_1\ x = e'; e_1 : T$, which is what we wanted to prove, since $(T_1\ x = E; e_1)[e'] = T_1\ x = e'; e_1$

- $E = (x = E')$, and $E[e] = (x = e)$. In this case $\Gamma \vdash x = e : T$ can be derived by hypothesis. By Case 20 of Lemma 2, $\Gamma \vdash x : T$ and $\Gamma \vdash e : T$. Setting $T' = T$, $\Gamma \vdash e : T$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : T$, and consider rule ASS in Section 6.3: its premises are satisfied by $\Gamma \vdash x : T$ and $\Gamma \vdash e' : T$; we can then derive $\Gamma \vdash x = e' : T$, which is what we aimed to prove since $(x = E')[e'] = (x = e')$.

- $E = (E'.s = e_1)$, and $E[e] = (e.s = e_1)$. In this case $\Gamma \vdash e.s = e_1 : T$ can be derived by hypothesis. By Case 21 of Lemma 2, $\Gamma \vdash e : C$, $\Gamma \vdash e.s : T$, and $\Gamma \vdash e_1 : T$. Setting $T' = C$, $\Gamma \vdash e : C$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : C$, and consider rule STATEASS in Section 6.3. The second premise is satisfied by $\Gamma \vdash e_1 : T$; the first one is also satisfied, since $\Gamma \vdash e' : C$ makes it possible to derive $\Gamma \vdash e'.s : T$. Hence, we can derive $\Gamma \vdash e'.s = e_1 : T$, which is what we aimed to prove since $(E.s = e_1)[e'] = (e'.s = e_1)$.

- $E = (c.s = E')$, and $E[e] = (c.s = e)$. In this case $\Gamma \vdash c.s = e : T$ can be derived by hypothesis. By Case 21 of Lemma 2, $\Gamma \vdash c : C$, $\Gamma \vdash c.s : T$, and $\Gamma \vdash e : T$. Setting $T' = T$, $\Gamma \vdash e : T$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : T$, and consider rule STATEASS in Section 6.3. The second premise is satisfied by $\Gamma \vdash e' : T$, whereas the first one is so by $\Gamma \vdash c.s : T$. Hence, we can derive $\Gamma \vdash c.s = e' : T$, which is what we aimed to prove since $(c.s = E)[e'] = (c.s = e')$.

- $E = E'[e_1]$, and $E[e] = e[e_1]$. In this case $\Gamma \vdash e[e_1] : T$ can be derived by hypothesis. By Case 22 of Lemma 2, $\exists T_1, T_2$ such that $T = T_2$, $\Gamma \vdash e : \mathsf{mapping}(T_1 \Rightarrow T_2)$, and $\Gamma \vdash e_1 : T_1$. Setting $T' = \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash e : \mathsf{mapping}(T_1 \Rightarrow T_2)$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : \mathsf{mapping}(T_1 \Rightarrow T_2)$, and consider rule MAPPSEL in Section 6.3. The first premise is satisfied by $\Gamma \vdash e' : \mathsf{mapping}(T_1 \Rightarrow T_2)$, whereas the second one is so by $\Gamma \vdash e_1 : T_1$. Hence, we can derive $\Gamma \vdash e'[e_1] : T_2$, which is what we aimed to prove since $(E[e_1])[e'] = e'[e_1]$.

- $E = M[E']$, and $E[e] = M[e]$. In this case $\Gamma \vdash M[e] : T$ can be derived by hypothesis. By Case 22 of Lemma 2, $\exists T_1, T_2$ such that $T = T_2$, $\Gamma \vdash M : \mathsf{mapping}(T_1 \Rightarrow T_2)$, and $\Gamma \vdash e : T_1$. Setting $T' = T_1$, $\Gamma \vdash e : T_1$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : T_1$, and consider rule MAPPSEL in Section 6.3. The first premise is satisfied by $\Gamma \vdash M : \mathsf{mapping}(T_1 \Rightarrow T_2)$, whereas the second one is so by $\Gamma \vdash e' : T_1$. Hence, we can derive $\Gamma \vdash M[e'] : T_2$, which is what we aimed to prove since $(M[E])[e'] = M[e']$.

- $E = E'[e_1 \to e_2]$, and $E[e] = e[e_1 \to e_2]$. In this case $\Gamma \vdash e[e_1 \to e_2] : T$ can be derived by hypothesis. By Case 23 of Lemma 2, $\exists T_1, T_2$ such that $T = \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash e : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash e_1 : T_1$, and $\Gamma \vdash e_2 : T_2$. Setting $T' = \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash e : \mathsf{mapping}(T_1 \Rightarrow T_2)$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash$

e' : $\mathsf{mapping}(T_1 \Rightarrow T_2)$, and consider rule MAPPASS in Section 6.3. The first premise is satisfied by $\Gamma \vdash e' : \mathsf{mapping}(T_1 \Rightarrow T_2)$, whereas the others are is so by Case 23 of Lemma 2. Hence, we can derive $\Gamma \vdash e'[e_1 \to e_2] : \mathsf{mapping}(T_1 \Rightarrow T_2)$, which is what we aimed to prove since $(E[e_1 \to e_2])[e'] = e'[e_1 \to e_2]$.

- $E = M[e \to e_1]$, and $E[e] = M[e \to e_1]$. In this case $\Gamma \vdash M[e \to e_1] : T$ can be derived by hypothesis. By Case 23 of Lemma 2, $\exists T_1, T_2$ such that $T = \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash M : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash e : T_1$, and $\Gamma \vdash e_1 : T_2$. Setting $T' = T_1$, $\Gamma \vdash e : T_1$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : T_1$, and consider rule MAPPASS in Section 6.3. The second premise is satisfied by $\Gamma \vdash e' : T_1$, whereas the others are is so by Case 23 of Lemma 2. Hence, we can derive $\Gamma \vdash M[e' \to e_1] : \mathsf{mapping}(T_1 \Rightarrow T_2)$, which is what we aimed to prove since $(M[E \to e_1])[e'] = M[e' \to e_1]$.

- $E = M[v \to e]$, and $E[e] = M[v \to e]$. In this case $\Gamma \vdash M[v \to e'] : T$ can be derived by hypothesis. By Case 23 of Lemma 2, $\exists T_1, T_2$ such that $T = \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash M : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash v : T_1$, and $\Gamma \vdash e : T_2$. Setting $T' = T_2$, $\Gamma \vdash e : T_2$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : T_2$, and consider rule MAPPASS in Section 6.3. The third premise is satisfied by $\Gamma \vdash e' : T_2$, whereas the others are is so by Case 23 of Lemma 2. Hence, we can derive $\Gamma \vdash M[v \to e'] : \mathsf{mapping}(T_1 \Rightarrow T_2)$, which is what we aimed to prove since $(M[v \to E])[e'] = M[v \to e']$.

- $E = \mathsf{if}\ E'\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$, and $E[e] = \mathsf{if}\ e\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$. In this case $\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ can be derived by hypothesis. By Case 27 of Lemma 2, $\Gamma \vdash e : \mathsf{bool}$ and $\Gamma \vdash e_2, e_3 : T$. Setting $T' = \mathsf{bool}$, $\Gamma \vdash e : \mathsf{bool}$ proves the first conclusion of this lemma. Let e' be any expression such that $\Gamma \vdash e' : \mathsf{bool}$, and consider rule IF in Section 6.3: $\Gamma \vdash e' : \mathsf{bool}$ and $\Gamma \vdash e_2, e_3 : T$ satisfy its premises, and thus we can derive $\Gamma \vdash \mathsf{if}\ e'\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$, which is what we aimed to prove since $(\mathsf{if}\ E\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)[e'] = \mathsf{if}\ e'\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$

- $E = \mathsf{return}\ E'$, and $E[e] = \mathsf{return}\ e$. In this case $\Gamma \vdash \mathsf{return}\ e : T$ can be derived by hypothesis. By Case 13 of Lemma 2, $\Gamma \vdash e : T$ can be derived, which is part of what we aimed to prove (setting $T' = T$). Let e' be any expression such that $\Gamma \vdash e' : T$, and consider rule RETURN in Section 6.3: $\Gamma \vdash e' : T$ satisfies its premise, and thus we can derive $\Gamma \vdash e' : T$, which is what we aimed to prove since $(\mathsf{return}\ E)[e'] = \mathsf{return}\ e'$.

□

We can now prove Theorem 4:

Theorem 4 (Subject Reduction)**.**
If $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ *with* $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ *then* $\exists \Delta$ *such that* $\Gamma' = \Gamma \cdot \Delta$ *and* $\Gamma' \vdash \langle \beta', \sigma', e' \rangle : T$.

Proof. We prove this theorem by induction on the height of the derivation of one reduction step $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta, \sigma, e' \rangle$.

Base cases These cases correspond to the axioms in Section 5.5, where the height of the derivation corresponding to the computational step is 0.

- IF-TRUE. In this case $\langle \beta, \sigma, \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \rangle \longrightarrow \langle \beta, \sigma, e_1 \rangle$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 : T$, and by Case 27 of Lemma 2 we know $\Gamma \vdash e_1 : T$, which makes it derivable $\Gamma' \vdash \langle \beta, \sigma, e_1 \rangle : T$, since β and σ are well-typed by hypothesis, setting $\Delta = \emptyset$.

- IF-FALSE. This case is dual to IF-TRUE.

- SEQ-C. In this case $\langle \beta, \sigma, v; e_1 \rangle \longrightarrow \langle \beta, \beta, e_1 \rangle$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, v; e_1 \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash v; e_1 : T$, and by Case 18 of Lemma 2 we know $\Gamma \vdash e_1 : T$, which makes it derivable $\Gamma' \vdash \langle \beta, \beta, e_1 \rangle : T$, since β is well-typed by hypothesis, setting $\Delta = \emptyset$.

- SEQ-R. In this case $\langle \beta, \sigma, \mathsf{revert}; e_1 \rangle \longrightarrow \langle \beta_0, \sigma, \mathsf{revert} \rangle$, where $\sigma = \beta_0$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, \mathsf{revert}; e_1 \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash \sigma$ (and consequently $\Gamma \vdash \beta_0$). Hence, bearing in mind that revert is well-typed regardless of the actual T (as stated by rule REVERT in Section 6.3), we have $\Gamma \vdash \mathsf{revert} : T$, which implies $\Gamma' \vdash \langle \beta_0, \sigma, \mathsf{revert} \rangle : T$, since $\sigma = \beta_0$ is well-typed by hypothesis, setting $\Delta = \emptyset$.

- SEQ. In this case $\langle \beta, \sigma, v; e_1 \rangle \longrightarrow \langle \beta, \sigma, e_1 \rangle$, where $\mathrm{Top}(\sigma) = a$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, v; e_1 \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash v; e_1 : T$, and by Case 18 of Lemma 2 we know $\Gamma \vdash e_1 : T$, which makes it derivable $\Gamma' \vdash \langle \beta, \sigma, e_1 \rangle : T$, since β and σ are well-typed by hypothesis, setting $\Delta = \emptyset$.

- STATESEL. In this case $\langle \beta, \sigma, c.s \rangle \longrightarrow \langle \beta, \sigma, v \rangle$, where $\beta(c) = (C, \tilde{s:v}, n)$ and $s \in \tilde{s}$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, c.s \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash c.s : T$. Furthermore, v has same type T; that is $\Gamma \vdash v : T$. Hence, $\Gamma' \vdash \langle \beta, \sigma, v \rangle : T$, since β and σ are well-typed by hypothesis and $\Delta = \emptyset$.

- DECL. In this case $\langle \beta, \sigma, T\ x = v; e_1 \rangle \longrightarrow \langle \beta \cdot [x \mapsto v], \sigma, v; e_1 \rangle$, where $x \notin \mathrm{dom}(\beta)$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, T\ x = v; e_1 \rangle : T_2$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash T\ x = v; e_1 : T_2$, and by Case 19 of Lemma 2 we know $\Gamma \vdash e_1 : T_2$ and $\Gamma \vdash v : T_1$. Let $\Delta = x : T_1$ and $\Gamma' = \Gamma \cdot \Delta$, and consider $\Gamma' \vdash \beta \cdot [x \mapsto v]$. Rule VARIABLE requires $\Gamma' \vdash \beta$, which is derivable by applying Lemma 10 to the hypothesis $\Gamma \vdash \beta, x \notin \mathrm{dom}(\beta)$, which is true by hypothesis of DECL, $\Gamma' \vdash x : T_1$, and $\Gamma' \vdash v : T_1$. The former is true by rule VAR in Section 6.3, since $\Delta = x : T_1$. The latter is true by Lemma 12 applied to $\Gamma \vdash v : T_1$. Hence, by rule VARIABLE, $\Gamma' \vdash \beta \cdot [x \mapsto v]$ is true. Furthermore, by Lemma 11, given $\Gamma \vdash \sigma$ we know that also $\Gamma' \vdash \sigma$ can be derived. Hence, by rule CONFIGURATION, we can derive $\Gamma' \vdash \langle \beta \cdot [x \mapsto v], \sigma, v; e_1 \rangle : T_2$, which is what we wanted to prove.

- VAR. In this case $\langle \beta, \sigma, x \rangle \longrightarrow \langle \beta, \sigma, \beta(x) \rangle$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, x \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash x : T$, and by Case 10 of Lemma 2 $x : T \in \Gamma$. By rule VARIABLE, each value v pointed to by a variable identifier x has the same type of x itself. This means that $\Gamma \vdash \beta(x) : T$. Hence, $\Gamma' \vdash \langle \beta, \sigma, \beta(x) \rangle : T$, since β and σ are well-typed by hypothesis and setting $\Delta = \emptyset$.

- BALANCE. In this case $\langle \beta, \sigma, \mathsf{balance}(a) \rangle \longrightarrow \langle \beta, \sigma, n \rangle$, where $\beta(a) = (C, s\tilde{:}v, n)$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, \mathsf{balance}(a) \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash \mathsf{balance}(a) : T$, and by Case 11 of Lemma 2 we know $T = \mathsf{uint}$ and $\Gamma \vdash a : \mathsf{address}$. From rule CONTRACT in Section 6.3 follows $\Gamma \vdash n : \mathsf{uint}$, which makes it derivable $\Gamma' \vdash \langle \beta, \sigma, n \rangle : \mathsf{uint}$, since β and σ are well-typed by hypothesis, setting $\Delta = \emptyset$.

- ADDRESS. In this case $\langle \beta, \sigma, \mathsf{address}(c) \rangle \longrightarrow \langle \beta, \sigma, a \rangle$, where $\hat{\beta}(c) = a$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, \mathsf{address}(c) \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash \mathsf{address}(c) : T$, and by Case 12 of Lemma 2 we know $T = \mathsf{address}$ and $\Gamma \vdash c : C$. We defined $\hat{\beta}(c) = a$ if $(c, a) \in \mathrm{dom}(\beta)$. $\Gamma \vdash \beta$ implies, by rule CONTRACT in Section 6.3, that every pair $(c, a) \in \mathrm{dom}(\beta)$ is well-formed, and hence, by Lemma 13, $\Gamma \vdash a : \mathsf{address}$. From this follows $\Gamma' \vdash \langle \beta, \sigma, a \rangle : \mathsf{address}$, since β and σ are well-typed by hypothesis, setting $\Delta = \emptyset$.

- ASS. In this case $\langle \beta, \sigma, x = v \rangle \longrightarrow \langle \beta[x \mapsto v], \sigma, v \rangle$, where $x \in \mathrm{dom}(\beta)$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, x = v \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash x = v : T$, and by Case 20 of Lemma 2, $\Gamma \vdash x : T$ and $\Gamma \vdash v : T$. Provided that the type of x and v is T, β remains well-typed. In fact VARIABLE (Section 6.3) applies and makes it derivable $\Gamma' \vdash \langle \beta[x \mapsto v], \sigma, v \rangle : T$, setting $\Delta = \emptyset$.

- STATEASS In this case $\langle \beta, \sigma, c.s = v' \rangle \longrightarrow \langle \beta[c.s \mapsto v'], \sigma, v' \rangle$, where $\beta(c) = (C, s\tilde{:}v, n)$ and $s \in \tilde{s}$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, c.s = v' \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash c.s = v' : T$, and by Case 21 of Lemma 2 we know $\Gamma \vdash c.s : T$ and $\Gamma \vdash v' : T$. Provided that the type of $c.s$ and v' is T, β remains well-typed. CONTRACT (Section 6.3) applies and makes it derivable $\Gamma' \vdash \langle \beta, \sigma, c.s = v' \rangle : T$, setting $\Delta = \emptyset$.

- MAPPSEL. In this case $\langle \beta, \sigma, M[v_1] \rangle \longrightarrow \langle \beta, \sigma, M(v_1) \rangle$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, M[v_1] \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash M[v_1] : T$, and by Case 22 of Lemma 2, $\exists T_1, T_2$ such that $\Gamma \vdash M : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $T = T_2$, and $\Gamma \vdash v_1 : T_1$. Since a mapping is a total function $T_1 \to T_2$, $M[v_1]$ is always well-defined and has type T_2, that is $\Gamma' \vdash \langle \beta, \sigma, M(v_1) \rangle : T_2$, where β and σ are well-typed by hypothesis and $\Delta = \emptyset$.

- MAPPASS. In this case $\langle \beta, \sigma, M[v_1 \to v_2] \rangle \longrightarrow \langle \beta, \sigma, M' \rangle$, where $M' = M \backslash \{(v_1, M(v_1))\} \cup \{(v_1, v_2)\}$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, M[v_1 \to v_2] \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash M[v_1 \to v_2] : T$, and by Case 23 of Lemma 2, $\exists T_1, T_2$ such that $\Gamma \vdash M : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $T = \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash v_1 : T_1$, and $\Gamma \vdash v_2 : T_2$. Looking at M', we note it is obtained from M substituting the pair $(v_1, M(v_1))$ with (v_1, v_2). The types of both v_1 and v_2 are correct with respect to $\mathsf{mapping}(T_1 \Rightarrow T_2)$, and so we can derive $\Gamma' \vdash \langle \beta, \sigma, M' \rangle : \mathsf{mapping}(T_1 \Rightarrow T_2)$, since β and σ are well-typed by hypothesis and $\Delta = \emptyset$.

- NEW-1. In this case $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) \cdot [(c, a) \mapsto (C, s\tilde{:}v, n)], \sigma, c \rangle$, where $(c, a) \notin \mathrm{dom}(\beta)$, $\mathrm{sv}(C) = \tilde{T}\,\tilde{s}$, and $|\tilde{v}| = |\tilde{s}|$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) : T$, and by Case 16 of Lemma 2 we know $T = C$, $\Gamma \vdash n : \mathsf{uint}$, and $\Gamma \vdash \tilde{v} : \tilde{T}$. Let $\Delta =$

$c : C, a : \mathsf{address}$ and $\Gamma' = \Gamma \cdot \Delta$, and consider $\Gamma' \vdash \beta \cdot [(c, a) \mapsto (C, \tilde{sv}, n)]$. Rule CONTRACT requires $\Gamma' \vdash \beta$, which is derivable by applying Lemma 10 to the hypothesis $\Gamma \vdash \beta$, $(c, a) \notin \mathrm{dom}(\beta)$, which is true by hypothesis of NEW-1, $\Gamma' \vdash c : C$, $\Gamma' \vdash a : \mathsf{address}$, $\Gamma' \vdash \tilde{v} : \tilde{T}$ (where $\mathrm{sv}(C) = \tilde{Ts}$), and $\Gamma' \vdash n : \mathsf{uint}$. $\Gamma' \vdash c : C$ and $\Gamma' \vdash a : \mathsf{address}()$ are derivable by rules REF and ADDRESS, respectively. $\Gamma' \vdash \tilde{v} : \tilde{T}$ and $\Gamma' \vdash n : \mathsf{uint}$ are, on the other hand, derivable by Lemma 10 applied to the hypotheses $\Gamma \vdash n : \mathsf{uint}$ and $\Gamma \vdash \tilde{v} : \tilde{T}$. Furthermore, uptbal only modifies β incrementing or decrementing the balance n, thus preserving its well-formedness. We can here suppose that such operation is successful, since the case where uptbal returns $\bot$ is dealt with by NEW-R (see below). Hence, by rule CONTRACT in Section 6.3 it follows $\Gamma' \vdash \mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) \cdot [(c, a) \mapsto (C, \tilde{sv}, n)]$. By Lemma 11, given $\Gamma \vdash \sigma$ also $\Gamma' \vdash \sigma$ is true. This makes it derivable, by rule CONFIGURATION, $\Gamma' \vdash \langle \mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) \cdot [(c, a) \mapsto (C, \tilde{sv}, n)], \sigma, c \rangle : C$, which is what we wanted to prove.

- NEW-2. This case is very similar to NEW-1, with the only difference that no balance updates are made on β.

- NEW-R. In this case $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$, with $\mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) = \bot$. By rule REVERT in Section 6.3 $\Gamma \vdash \mathsf{revert} : T$, which implies $\Gamma' \vdash \langle \beta, \sigma, \mathsf{revert} \rangle : T$, where β and σ are well-typed by hypothesis and $\Delta = \emptyset$.

- CALL. In this case $\langle \beta, \sigma, c.f.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta', \sigma \cdot a, e\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\} \rangle$, where $\hat{\beta}(c) = a$, $\beta^C(c) = C$, $\mathrm{fbody}(C, f, \tilde{v}) = (\tilde{x}, e)$, $(x_i, a)^{x_i \in \tilde{x}} \notin \mathrm{dom}(\beta)$, and $\beta' = \mathrm{uptbal}(\mathrm{uptbal}(\beta, a, n), \mathrm{Top}(\sigma), -n) \cdot [x_i \mapsto v_i\ ^{x_i \in \tilde{x}, v_i \in \tilde{v}}]$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, c.f.\mathsf{value}(n)(\tilde{v}) \rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash c.f.\mathsf{value}(n)(\tilde{v}) : T$, and by Case 24 of Lemma 2 $\exists \tilde{T_1}, T_2$ such that $T = \tilde{T_1} \to T_2$, $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$, $\Gamma \vdash c : C$, $\Gamma \vdash n : \mathsf{uint}$, and $\Gamma \vdash \tilde{v} : \tilde{T_1}$. Since $\hat{\beta}(c) = a$ is well-defined by hypothesis, $\exists (c, a) \in \mathrm{dom}(\beta)$ such that $\Gamma \vdash a : \mathsf{address}$: from it follows $\Gamma \vdash \sigma \cdot a$. Also note that $\Gamma \vdash \sigma \Rightarrow \Gamma \vdash \mathrm{Top}(\sigma) : \mathsf{address}$, where $\mathrm{Top}(\sigma) \neq \emptyset$ (otherwise CALLTOPLEVEL would apply). From rule F OK IN C (Section 6.3) we know $\mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}, \mathsf{msg.value} : \mathsf{uint}, \tilde{x} : \tilde{T_1} \vdash e : T_2$. By Lemma 5, considering the judgments $\Gamma \vdash c : C$, $\Gamma \vdash \mathrm{Top}(\sigma) : \mathsf{address}$, and $\Gamma \vdash n : \mathsf{uint}$ (which we have already proven as true), follows $\Gamma \vdash e\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\} : T_2$. The last thing we have to deal with is β'. First, as we already pointed out, uptbal only modifies the balances via arithmetical operations, thus preserving the well-formedness of β. Again, we can suppose that such operation is successful, since the case where uptbal returns $\bot$ is dealt with by CALL-R (see below). Secondly, we append to β a list of fresh pairs of local variables, together with the address of the contract identifying the invoked function. From $\Gamma \vdash \tilde{v} : \tilde{T_1}$ follows that each value v_i has the same type of the formal parameter x_i it refers to, and we have already proven $\Gamma \vdash \mathrm{Top}(\sigma) : \mathsf{address}$. Hence, by rule VARIABLE in Section 6.3, $\Gamma \vdash \beta'$, and we can finally derive $\Gamma' \vdash \langle \beta', \sigma \cdot a, e\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\} \rangle : T_2$, where $\Delta = \emptyset$.

- CALLTOPLEVEL. In this case $\langle \beta, \sigma, c.f.\mathsf{value}(n).\mathsf{sender}(a')(\tilde{v}); e' \rangle \longrightarrow \langle \beta', \sigma \cdot a, e\{\mathsf{this} := c, \mathsf{msg.sender} :=$

$a', \mathsf{msg.value} := n\}; e'\rangle$, where $\hat{\beta}(c) = a$, $\beta^C(c) = C$, $\mathrm{fbody}(C, f, \tilde{v}) = (\tilde{x}, e)$, $(x_i, a)^{x_i \in \tilde{x}} \notin \mathrm{dom}(\beta)$, and $\beta' = \mathrm{uptbal}(\mathrm{uptbal}(\beta, a, n), \mathrm{Top}(\sigma), -n) \cdot [x_i \mapsto v_i \ ^{x_i \in \tilde{x}, v_i \in \tilde{v}}]$. This case is very similar to CALL, the only difference relying on the sender, which is here explicitly set to a' (since $\mathrm{Top}(\sigma) = \emptyset$). We shall prove only the correctness of such sender, referring to the previous case for the rest of the proof. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, c.f.\mathsf{value}(n).\mathsf{sender}(a')(\tilde{v}); e'\rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash c.f.\mathsf{value}(n).\mathsf{sender}(a')(\tilde{v}); e' : T$, and by Case 26 of Lemma 2, $\Gamma \vdash a' : \mathsf{address}$. Setting $\Delta = \emptyset$, we obtain $\Gamma' \vdash a' : \mathsf{address}$. The proof is now the same as before, using a' instead of $\mathrm{Top}(\sigma)$.

- CALL-R. This case is the same as NEW-R:
  $\langle \beta, \sigma, c.f.\mathsf{value}(n)(\tilde{v})\rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert}\rangle$, with
  $\mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) = \bot$. By rule REVERT in Section 6.3 $\Gamma \vdash \mathsf{revert} : T$, which implies $\Gamma' \vdash \langle \beta, \sigma, \mathsf{revert}\rangle : T$, since β and σ are well-typed by hypothesis and $\Delta = \emptyset$.

- CALLTOPLEVEL-R. This case is the same as CALL-R.

- TRANSFER. This case is the same as CALL, the only difference relying on the absence of formal parameters. Furthermore, the return type is unit. Hence, β' is well-formed for what we said before, and so are $\sigma \cdot \mathrm{Top}(\sigma)$ and $e\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\}$. Also note that $\mathrm{Top}(\sigma) \neq \emptyset$, and hence using it to enlarge σ and to form β' is safe.

- TRANSFER-R. This case is the same as CALL-R.

- RETURN. In this case $\langle \beta, \sigma \cdot a, \mathsf{return}\, v\rangle \longrightarrow \langle \beta, \sigma, v\rangle$. $\Gamma \vdash \sigma \cdot a \Rightarrow \Gamma \vdash \sigma$ by rule CALLSTACK in Section 6.3. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, \mathsf{return}\, v\rangle : T$; this implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash \mathsf{return}\, v : T$. Hence, by Case 13 of Lemma 2, $\Gamma \vdash v : T$ and we can derive $\Gamma' \vdash \langle \beta, \sigma, v\rangle : T$, where $\Delta = \emptyset$.

- RETURN-R. In this case $\langle \beta, \sigma \cdot a, \mathsf{return\ revert}\rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert}\rangle$. By rule CONFIGURATION in Section 6.3, $\Gamma \vdash \mathsf{revert} : T$ is derivable, and by Case 5 of lemma 2, we obtain $\Gamma \vdash \mathsf{revert} : T$. Furthermore β and σ are well-typed for what we previously said for RETURN, and thus we can derive $\Gamma' \vdash \langle \beta, \sigma, \mathsf{revert}\rangle : T$, where $\Delta = \emptyset$.

- CONTRRETR. In this case $\langle \beta, \sigma, C(a)\rangle \longrightarrow \langle \beta, \sigma, c\rangle$, where $\beta^C(a) = C$ and $\hat{\beta}(a) = c$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, C(a)\rangle : T$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash C(a) : T$, and by Case 17 of Lemma 2 we know $T = C$ and $\Gamma \vdash a : \mathsf{address}$. Since $\hat{\beta}(a) = c$ is well-defined by hypothesis, $\exists (c, a) \in \mathrm{dom}(\beta)$, and since $\Gamma \vdash \beta$ we get $\Gamma \vdash c : C$. Hence, $\Gamma' \vdash \langle \beta, \sigma, c\rangle : C$, considering that also σ is well-typed by hypothesis and setting $\Delta = \emptyset$.

- CONTRRETR-R. In this case $\langle \beta, \sigma, C(a)\rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert}\rangle$, where $\beta^C(a) = C'$ and $C' \neq C$. By rule REVERT in Section 6.3 $\Gamma \vdash \mathsf{revert} : T$, which implies $\Gamma' \vdash \langle \beta, \sigma, \mathsf{revert}\rangle : T$, where β and σ are well-typed by hypothesis, setting $\Delta = \emptyset$.

- REVERT. In this case $\langle \beta, \sigma, E[\mathsf{revert}]\rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert}\rangle$. Whatever E is, by rule REVERT in Section 6.3 $\Gamma \vdash \mathsf{revert} : T$. Hence, $\Gamma' \vdash \langle \beta, \sigma, \mathsf{revert}\rangle : T$, where β and σ are well-typed by hypothesis, setting $\Delta = \emptyset$.

Inductive case This case takes into account only rule CONG, the sole computational rule of our semantics. In this case $\langle \beta, \sigma, E[e] \rangle \longrightarrow^{k+1} \langle \beta', \sigma', E[e'] \rangle$. The height of the derivation making this step possible is $k+1$, and its last judgment is $\langle \beta, \sigma, e \rangle \longrightarrow^{k} \langle \beta', \sigma', e' \rangle$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, E[e] \rangle : T$. To apply the inductive hypothesis we need to prove that, given a T', $\Gamma \vdash \langle \beta, \sigma, e \rangle : T'$. We do so using Lemma 14, whose hypothesis is satisfied by the hypotheses of this theorem. Hence, $\exists T'$ such that $\Gamma \vdash e : T'$ and $\forall e'$ such that $\Gamma \vdash e' : T'$ we have $\Gamma \vdash E[e'] : T$. The former is enough to prove $\Gamma \vdash \langle \beta, \sigma, e \rangle : T'$, since β and σ are well-typed by hypothesis (i.e. $\Gamma \vdash \beta$ and $\Gamma \vdash \sigma$). Bearing in mind that the premise of CONG states $\langle \beta, \sigma, e \rangle \longrightarrow^{k} \langle \beta', \sigma', e' \rangle$, we can apply the induction hypothesis to conclude $\Gamma \vdash \langle \beta', \sigma', e' \rangle : T'$. This implies, by rule CONFIGURATION in Section 6.3, $\Gamma \vdash e' : T'$. We now make use of the latter conclusion of Lemma 14: $\Gamma \vdash e' : T' \Rightarrow \Gamma \vdash E[e'] : T'$. Consequently, $\Gamma \vdash \langle \beta', \sigma', E[e'] \rangle : T$, which is what we aimed to prove. □

E.6 Type Safety Theorem for FS configurations

We recall the theorem of type safety for FS configurations:

Theorem 1 (Type safety for configurations)**.**
If $\Gamma \vdash \langle \beta, \beta, e \rangle : T$*,* $\langle \beta, \beta, e \rangle$ *is closed, and* $\exists (\beta', \sigma', e')$ *such that* $\langle \beta, \beta, e \rangle \longrightarrow^{*} \langle \beta', \sigma', e' \rangle$*, with* $\langle \beta', \sigma', e' \rangle \not\longrightarrow$*, then either* $e' = v$*, where* v *is a value, or* $e' =$ revert.

Proof. Let β', σ', e' be such that $\langle \beta, \beta, e \rangle \longrightarrow^{*} \langle \beta', \sigma', e' \rangle$, with $\langle \beta', \sigma', e' \rangle \not\longrightarrow$. The theorem states e' is either a value v or revert. From $\Gamma \vdash \langle \beta, \beta, e \rangle : T$ and Theorem 4 it follows $\Gamma \vdash \langle \beta', \sigma', e' \rangle : T$. Now, by Theorem 3 we have three cases: e' is either a value, revert, or $\exists \beta'', \sigma'', e''$ such that $\langle \beta', \sigma', e' \rangle \longrightarrow \langle \beta'', \sigma'', e'' \rangle$, but we assumed e' such that $e' \not\longrightarrow$. Hence, the third case does not apply and we conclude that e' is either a value v or revert. □

E.7 Type Safety Theorem for FS programs

We recall Theorem 2:

Theorem 2 (Type safety for programs)**.**
Let $\mathcal{P} = (CT, \beta, e_1; \ldots; e_n)$ *be an FS program. If* $\Gamma \vdash (CT, \beta, e_1; \ldots; e_n) : T$*,* $\mathcal{P}$ *is closed, and* $\exists (\beta', e')$ *such that* $\langle \beta, e_1; \ldots; e_n \rangle \Longrightarrow^{*} \langle \beta', e' \rangle$*, with* $\langle \beta', e' \rangle \not\Longrightarrow$*, then either* $e' = v$ *or* $e' =$ revert.

Proof. We prove this theorem by induction on n.

Base case In this case $n = 1$, and the program is (CT, β, e_1). By hypothesis $\Gamma \vdash (CT, \beta, e_1) : T$ and by rule PROGRAM in Section 6.3 we know $\Gamma \vdash \langle \beta, \beta, e_1 \rangle : T$. By hypothesis we also know that $\langle \beta, e_1 \rangle \Longrightarrow^{*} \langle \beta', e' \rangle$, with $\langle \beta', e' \rangle \not\Longrightarrow$. Since there are no other expression to evaluate after e_1 (i.e. no sequential composition), only SUCCESS and FAILURE (as defined in Figure 5.3) can be applied. This means that we are to evaluate e_1 according to the rules defined in Section 5.5, thus obtaining $\langle \beta, \beta, e_1 \rangle \longrightarrow^{*} \langle \beta', \beta, e' \rangle$, where $\langle \beta', \beta, e' \rangle \not\longrightarrow$. Two things are worth to be pointed out. First, the evaluation begins at the initial state $\langle \beta, \beta, e_1 \rangle$ because e_1 is the first expression in $e_1; \ldots; e_n$. Secondly, the evaluation of e_1 "creates" the blockchain β', which is not copied over the second component (σ) of the final configuration. This

happens because, in this case, we do not consider any sequential compositions after e_1, and thus the rules committing or discarding the changes to the blockchain (i.e. SEQ-C and SEQ-R in Section 5.5) do not apply. By Lemma 1, (CT, β, e_1) closed implies $\langle \beta, \beta, e_1 \rangle$ closed. Hence, by Theorem 1, e' is either a value v or revert. We analyze the two cases separately:

- if $e' = v$ then, by rule SUCCESS in Figure 5.3, $\langle \beta, e_1 \rangle \Longrightarrow \langle \beta', v \rangle$.
- if $e' = \mathsf{revert}$ then, by rule FAILURE in Figure 5.3, $\langle \beta, e_1 \rangle \Longrightarrow \langle \beta', \mathsf{revert} \rangle$.

Inductive case We assume the property true for the sequence $e_1; \ldots; e_{i-1}$ and are to prove it for e_i. The sequence we are taking into account is thus $e_1; \ldots; e_{i-1}; e_i$. We are considering a subset of the initial tuple of expressions without reducing either β or CT. Hence, the hypothesis of closedness for $\mathcal{P}$ remains valid. By inductive hypothesis, if $\exists \beta', e'$ such that $\langle \beta, e_1; \ldots; e_{i-1} \rangle \Longrightarrow^* \langle \beta', e' \rangle$ and $\langle \beta', e' \rangle \not\Longrightarrow$, then either $e' = v'$ or $e' = \mathsf{revert}$. Hence, applying this to the bigger picture including e_i leads to $\langle \beta, e_1; \ldots; e_{i-1}; e_i \rangle \Longrightarrow^* \langle \beta', e'; e_i \rangle$. We make a distinction according to the actual e':

- if $e' = v'$ we have $\langle \beta, e_1; \ldots; e_{i-1} \rangle \Longrightarrow^* \langle \beta', v' \rangle$. By rule COMMIT in Figure 5.3, $\langle \beta, e_1; \ldots; e_{i-1}; e_i \rangle \Longrightarrow^* \langle \beta', e_i \rangle$, and we are to evaluate e_i on β'. Provided that no reverts have been thrown, evaluating e_i on β' (i.e. $\langle \beta', e_i \rangle$) is equivalent to evaluating $e_1; \ldots; e_i$ starting from the blockchain β. Hence, we consider the sequence $e_1; \ldots; e_i$, and reason about the configuration $\langle \beta, \beta, e_1; \ldots; e_i \rangle$. By hypothesis we know that $e_1; \ldots; e_n$ is well-typed ($\Gamma \vdash (CT, \beta, e_1; \ldots; e_n) : T$), and so is $e_1; \ldots; e_i$, since it is a subset of the original sequence: $\Gamma \vdash \beta : (CT, \beta, e_1; \ldots; e_i) : T$. Furthermore, we know that $e_1; \ldots; e_i$ evaluates to something: $\langle \beta, e_1; \ldots; e_i \rangle \Longrightarrow^* \langle \beta', e_i \rangle$ by induction hypothesis and rule COMMIT in Figure 5.3. The application of COMMIT implies the application of SUCCESS, which in turns evaluates the expressions using the rules defined in Section 5.5; that is $\langle \beta, \beta, e_1; \ldots; e_{i-1}; e_i \rangle \longrightarrow^* \langle \beta', \beta', e_i \rangle$. Let β'', e'' be such that $\langle \beta', \beta', e_i \rangle \longrightarrow^* \langle \beta'', \beta', e'' \rangle$, and thus $\langle \beta, \beta, e_1; \ldots; e_{i-1}; e_i \rangle \longrightarrow^* \langle \beta'', \beta', e'' \rangle$, where $\langle \beta'', \beta', e'' \rangle \not\longrightarrow$. Again, note that changes of β'' are not copied over β' since no top-level sequential composition follows e_i. By Lemma 1, $(CT, \beta, e_1; \ldots; e_i)$ closed implies $\langle \beta, \beta, e_1; \ldots; e_i \rangle$ closed. By Theorem 1, e'' is either a value v'' or revert. In the former case, by rule SUCCESS in Figure 5.3, $\langle \beta', e_i \rangle \Longrightarrow \langle \beta'', v'' \rangle$. In the latter, by rule FAILURE in Figure 5.3, $\langle \beta', e_i \rangle \Longrightarrow \langle \beta', \mathsf{revert} \rangle$.
- Let $j < i$ be such that e_j is the first expression in $e_1; \ldots; e_{i-1}$ throwing revert. We make a distinction based on the actual value of j:
  - if $j = i - 1$ then the first expression throwing revert was e_{i-1}. In this case, β' is the blockchain containing all the changes made by $e_1; \ldots; e_{i-2}$ (possibly $\emptyset$), and we know that $\langle \beta', e_{i-1} \rangle \Longrightarrow \langle \beta', \mathsf{revert} \rangle$. By rule ABORT in Figure 5.3, if $\langle \beta', e_{i-1} \rangle \Longrightarrow \langle \beta', \mathsf{revert} \rangle$ then $\langle \beta', e_{i-1}; e_i \rangle \Longrightarrow \langle \beta', \mathsf{revert} \rangle$, without even evaluating e_i.
  - if $j < i - 1$ then an expression in the middle of the smaller sequence $(e_1; \ldots; e_{i-1})$ threw revert. In this case, β' is the blockchain containing all the changes made by $e_1; \ldots; e_{j-1}$ (possibly $\emptyset$), and we know that $\langle \beta', e_j \rangle \Longrightarrow \langle \beta', \mathsf{revert} \rangle$. By rule ABORT in Figure 5.3, if $\langle \beta', e_j \rangle \Longrightarrow$

$\langle \beta', \mathsf{revert} \rangle$ then $\langle \beta', e_j; e_{j+1} \rangle \implies \langle \beta', \mathsf{revert} \rangle$, without even evaluating $e_{j+1}; \ldots; e_{i-1}; e_i$.

□

Appendix F

Proving the safety of FS$^+$

In this section we prove Lemmas of Permutation (Lemma 3), Weakening (Lemma 4), and Substitution (Lemma 9), and Theorems of Progress (Theorem 3), Subject Reduction (Theorem 4), and Safety for configurations (Theorem 1) and programs (Theorem 2) in the context of the new type system described in Chapter 7. In addition, we shall prove the Theorem of Cast Safety (Theorem 5) and Transfer Safety (Theorem 6).

In the proof that follows, we shall write $\Gamma \vdash_k e : T$ and $\langle \beta, \sigma, e \rangle \longrightarrow^k \langle \beta', \sigma', e' \rangle$ to indicate that the derivations of those judgments have height at most k.

F.1 Permutation Lemma

We recall Lemma 3:

Lemma 3 (Permutation)**.**
If $\Gamma \vdash e : T$ can be derived and Δ is a permutation of Γ, then the judgment $\Delta \vdash e : T$ can be derived and the derivation has the same height of the previous one.

Proof. We prove this lemma by induction on the height of the derivation of the judgment $\Gamma \vdash e : T$. We shall rewrite the proof only for the rules in Section 7.4, that defines the changes with respect to the rules in Section 6.3. All the cases not showed here are as in Appendix E.1.

Base cases The only modified axiom is ADDRESS, whose derivation has height 1.

- ADDRESS. The judgment is $\Gamma \vdash a : \mathsf{address}\langle \mathsf{C} \rangle$. Since it can be derived, by Case 5 of Lemma 7 we know $a : \mathsf{address}\langle \mathsf{C} \rangle \in \Gamma$. Let Δ be a permutation of Γ. Hence, Δ has exactly the same elements as Γ, but in a different order. This means that $a : \mathsf{address}\langle \mathsf{C} \rangle \in \Delta$, and that $\Delta \vdash a : \mathsf{address}\langle \mathsf{C} \rangle$ can be derived.

Inductive cases Given a judgment J such that its derivation has height $k+1$, we prove the inductive cases on the last rule used to derive J. We assume the lemma for the judgments with height at most k and we prove it for those with height $k+1$.

- FUN. In this case $J = \Gamma \vdash_{k+1} c.f : \tilde{T_1} \to T_2$ was derived from the judgments $\Gamma \vdash_k c : C$ and $\Gamma \vdash_k \mathsf{this} : C'$, together with the additional premises $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$ and $C' <: \mathrm{fsender}(C, f)$. By inductive hypothesis, $\Delta \vdash_k c : C$ and $\Delta \vdash_k \mathsf{this} : C'$, where Δ is a permutation of Γ, can be derived.

Considering the judgment $\Delta \vdash_{k+1} c.f : \tilde{T_1} \to T_2$, we note that it can be derived from $\Delta \vdash_k c : C$ and $\Delta \vdash_k \mathsf{this} : C'$, since the other two premises are still valid.

- MAPPING. In this case $J = \Gamma \vdash_{k+1} M : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from the judgments $\Gamma \vdash_k \tilde{k} : T_1'$ and $\Gamma \vdash_k \tilde{v} : T_2'$, where $T_1' <: T_1$ and $T_2' <: T_2$. By inductive hypothesis, also the judgments $\Delta \vdash_k \tilde{k} : T_1'$ and $\Delta \vdash_k \tilde{v} : T_2'$, where Δ is a permutation of Γ, are derivable. Hence, by applying MAPPING to the latter two judgments and the two additional premises, $\Delta \vdash_{k+1} M : \mathsf{mapping}(T_1 \Rightarrow T_2)$.

- BAL. In this case $J = \Gamma \vdash_{k+1} \mathsf{balance}(e) : \mathsf{uint}$ was derived from $\Gamma \vdash_k e : \mathsf{address}\langle \mathsf{C} \rangle$. By inductive hypothesis, also the judgment $\Delta \vdash_k e : \mathsf{address}\langle \mathsf{C} \rangle$, where Δ is a permutation of Γ. Hence, by applying BAL to it we obtain $\Delta \vdash_{k+1} \mathsf{balance}(e) : \mathsf{uint}$.

- ADDR. This case is very similar to BAL: $J = \Gamma \vdash_{k+1} \mathsf{address}(e) : \mathsf{address}\langle \mathsf{C} \rangle$ was derived from $\Gamma \vdash_k e : C$. By inductive hypothesis, also the judgment $\Delta \vdash_k e : C$, where Δ is a permutation of Γ, is derivable. Hence, by applying ADDR to it we obtain $\Delta \vdash_{k+1} \mathsf{address}e : \mathsf{address}\langle \mathsf{C} \rangle$.

- IF. In this case $J = \Gamma \vdash_{k+1} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ was derived from $\Gamma \vdash_k e_1 : \mathsf{bool}$, $\Gamma \vdash_k e_2 : T_1$, and $\Gamma \vdash_k e_3 : T_2$, where $T_1 <: T$ and $T_2 <: T$. By inductive hypothesis, given a permutation Δ of Γ, the judgments $\Delta \vdash_k e_1 : \mathsf{bool}$, $\Delta \vdash_k e_2 : T_1$, and $\Delta \vdash_k e_3 : T_2$ can be derived. By applying IF to them we obtain $\Delta \vdash_{k+1} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_2 : T$.

- DECL. In this case $J = \Gamma \vdash_{k+1} T_1\ x = e_1; e_2 : T_2$ was derived from $\Gamma \vdash_k e_1 : T_1'$, where $T_1' <: T_1$, and $\Gamma, x : T_1 \vdash_k e_2 : T_2$. The derivations of the latter judgments have height k. Let $\Gamma' = \Gamma, x : T_1$, Δ be a permutation of Γ, and Δ' be a permutation of Γ'; by inductive hypothesis we can derive $\Delta \vdash_k e_1 : T_1'$. Nonetheless, we cannot directly apply the inductive hypothesis on $\Gamma' \vdash_k e_2 : T_2$, since we have $\Gamma' \neq \Gamma$. Still, we know that there exists a derivation, of height k, for this judgment. (Otherwise J would not be derivable, but this is a contradiction, since we assumed J has a valid derivation of height $k+1$.) Hence, it comes from another judgment J' having a derivation of height $k-1$ where any of the type rules we have defined was applied as a last step. We can now apply the inductive hypothesis on J' considering Δ' as a permutation, concluding that $\Delta' \vdash_k e_2 : T_2$ is derivable. Lastly, applying DECL to $\Delta \vdash_k e_1 : T_1'$ and $\Delta' \vdash_k e_2 : T_2$, together with the other premise, we obtain $\Delta \vdash_{k+1} T_1\ x = e_1; e_2 : T_2$.

- MAPPSEL. In this case $J = \Gamma \vdash_{k+1} e_1[e_2] : T_2$ was derived from $\Gamma \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Gamma \vdash_k e_2 : T_1'$, where $T_1' <: T_1$. By inductive hypothesis, the judgments $\Delta \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Delta \vdash_k e_2 : T_1'$ can be derived. Hence, applying MAPPSEL the them we obtain $\Delta \vdash_{k+1} e_1[e_2] : T_2$.

- ASS. In this case $J = \Gamma \vdash_{k+1} x = e : T$ was derived from $\Gamma \vdash_k x : T$ and $\Gamma \vdash_k e : T'$, where $T' <: T$. By inductive hypothesis, given a permutation Δ of Γ, we know we can derive both $\Delta \vdash_k x : T$ and $\Delta \vdash_k e : T'$. Hence, by applying ASS to them we can derive $\Delta \vdash_{k+1} x = e : T$.

- MAPPASS. In this case $J = \Gamma \vdash_{k+1} e_1[e_2 \to e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from $\Gamma \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash_k e_2 : T_1'$, and $\Gamma \vdash_k e_3 : T_2'$, where $T_1' <: T_1$ and $T_2' <: T_2$. By inductive hypothesis, given a permutation

Δ of Γ, we can derive $\Delta \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Delta \vdash_k e_2 : T_1'$, and $\Delta \vdash_k e_3 : T_2'$. By the application of MAPPASS we obtain $\Delta \vdash_{k+1} e_1[e_2 \rightarrow e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)$.

- STATEASS. This case is very similar to ASS: $J = \Gamma \vdash_{k+1} e_1.s = e_2 : T$ was derived from $\Gamma \vdash_k e_1.s : T$ and $\Gamma \vdash_k e_2 : T'$, with $T' <: T$. By inductive hypothesis, given a permutation Δ of Γ, we know we can derive both $\Delta \vdash_k e_1.s : T$ and $\Delta \vdash_k e_2 : T'$. Hence, by applying STATEASS to them we can derive $\Delta \vdash_{k+1} e_1.s = e_2 : T$.

- NEW. In this case $J = \Gamma \vdash_{k+1} \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : C$ was derived from $\Gamma \vdash_k \tilde{e} : \tilde{T}'$, with $\tilde{T}' <: \tilde{T}$, and $\Gamma \vdash_k e' : \mathsf{uint}$, together with the additional premise $|\tilde{e}| = |\tilde{s}|$, where $\mathrm{sv}(C) = \tilde{T}\,\tilde{s}$. By inductive hypothesis, we know that, given a permutation Δ of Γ, we can derive both $\Delta \vdash_k \tilde{e} : \tilde{T}'$ and $\Delta \vdash_k e' : \mathsf{uint}$. Provided that the additional premise checking the length of the tuples $\tilde{e}$ and $\tilde{s}$ is still valid, we can apply NEW to obtain $\Delta \vdash_{k+1} \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : C$.

- CONTRRETR. In this case $J = \Gamma \vdash_{k+1} D(e) : D$ was derived from $\Gamma \vdash_k e : \mathsf{address}\langle \mathsf{C} \rangle$. By inductive hypothesis $\Delta \vdash_k e : \mathsf{address}\langle \mathsf{C} \rangle$ is derivable, where Δ is a permutation of Γ. Considering $\Delta \vdash D(e) : D$, notice it can be derived from $\Delta \vdash_k e : \mathsf{address}\langle \mathsf{C} \rangle$ applying CONTRRETR, and its derivation has height $k + 1$.

- TRANSFER. In this case $J = \Gamma \vdash_{k+1} e_1.\mathsf{transfer}(e_2) : \mathsf{unit}$ was derived from $\Gamma \vdash_k e_1 : \mathsf{address}\langle \mathsf{C} \rangle$, $\Gamma \vdash_k \mathsf{this} : C'$, and $\Gamma \vdash_k e_2 : \mathsf{uint}$, together with the additional premise ensuring the definition of the fallback function in C ($\mathrm{ftype}(C, fb) = \{\} \rightarrow \mathsf{unit}$) and checking the required type for the sender ($C' <: \mathrm{fsender}(C, fb)$). By inductive hypothesis, considering a permutation Δ of Γ, we can derive $\Delta \vdash_k e_1 : \mathsf{address}\langle \mathsf{C} \rangle$, $\Delta \vdash_k \mathsf{this} : C'$, and $\Delta \vdash_k e_2 : \mathsf{uint}$. Lastly, we can apply TRANSFER to them to obtain $\Delta \vdash_{k+1} e_1.\mathsf{transfer}(e_2) : \mathsf{unit}$.

- CALL. In this case $J = \Gamma \vdash_{k+1} e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$ was derived from $\Gamma \vdash_k e_1 : C$, $\Gamma \vdash_k e_2 : \mathsf{uint}$, $\Gamma \vdash_k \mathsf{this} : C'$, and $\Gamma \vdash_k \tilde{e} : \tilde{T}_1'$, together with the additional premises checking the length of the tuple $\tilde{e}$ ($|\tilde{e}| = |\tilde{T}_1|$), the required type for the sender ($C' <: \mathrm{fsender}(C, f)$), and the type of f in C ($\mathrm{ftype}(C, f) = \tilde{T}_1 \rightarrow T_2$). By inductive hypothesis on the four judgments of height k, and given a permutation Δ of Γ, also the following judgments can be derived: $\Delta \vdash_k e : C$, $\Delta \vdash_k e_2 : \mathsf{uint}$, $\Delta \vdash_k \mathsf{this} : C'$, and $\Delta \vdash_k \tilde{e} : \tilde{T}_1'$. Applying CALL we derive $\Delta \vdash_{k+1} e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$.

- CALLTOPLEVEL. In this case $J = \Gamma \vdash_{k+1} e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T_2$ was derived from $\Gamma \vdash_k e_1 : C$, $\Gamma \vdash_k e_2 : \mathsf{uint}$, $\Gamma \vdash_k e_3 : \mathsf{address}\langle \mathsf{C}' \rangle$, and $\Gamma \vdash_k \tilde{e} : \tilde{T}_1'$, together with the additional premises checking the length of the tuple $\tilde{e}$ ($|\tilde{e}| = |\tilde{T}_1|$), the required type for the sender ($C' <: \mathrm{fsender}(C, f)$), and the type of f in C ($\mathrm{ftype}(C, f) = \tilde{T}_1 \rightarrow T_2$). By inductive hypothesis on the four judgments of height k, and given a permutation Δ of Γ, also the following judgments can be derived: $\Delta \vdash_k e : C$, $\Delta \vdash_k e_2 : \mathsf{uint}$, $\Delta \vdash_k e_3 : \mathsf{address}\langle \mathsf{C}' \rangle$, and $\Delta \vdash_k \tilde{e} : \tilde{T}_1'$. We then apply CALLTOPLEVEL and obtain a derivation of $\Delta \vdash_{k+1} e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T_2$.

- CALLVALUE. In this case $J = \Gamma \vdash_{k+1} e_1.\mathsf{value}(e_2)(\tilde{e}) : T_2$ was derived from $\Gamma \vdash_k e_1 : \tilde{T}_1 \rightarrow T_2$, $\Gamma \vdash_k e_2 : \mathsf{uint}$, and $\Gamma \vdash_k \tilde{e} : \tilde{T}_1'$, where $\tilde{T}_1' <: \tilde{T}_1$,

together with the premise checking the length of the tuple $\tilde{e}$ ($|\tilde{e}| = |\tilde{T_1}|$). By inductive hypothesis on the three judgments of height k, and given a permutation Δ of Γ, also the following judgments can be derived: $\Delta \vdash_k e_1 : \tilde{T_1} \to T_2$, $\Delta \vdash_k e_2 : \mathsf{uint}$, and $\Delta \vdash_k \tilde{e} : \tilde{T_1'}$. Applying CALLVALUE we derive $\Delta \vdash_{k+1} e_1.\mathsf{value}(e_2)(\tilde{e}) : T_2$.

□

F.2 Weakening Lemma

As said in Appendix E.2, Weakening Lemma is formalized over FS configurations $\langle \beta, \sigma, e \rangle$. To prove it, we shall first prove the validity of the same Lemma on the projections on β (Lemma 10), σ (Lemma 11), and e (Lemma 12). In the proof that follow, let $\Gamma' = \Gamma \cdot \Delta$.

F.2.1 Weakening of β

Here we prove the projection of the lemma of Weakening on the first component of $\langle \beta, \sigma, e \rangle$: β.

Lemma 10 (Weakening of β)**.**
Let $\Gamma \vdash \beta$ be a derivable judgment, and let Δ be such that $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$. Then $\Gamma \cdot \Delta \vdash \beta$ can be derived and its derivation has the same height as the previous one.

Proof. We prove this lemma by induction on the height of the derivation of the judgment $\Gamma \vdash \beta$.

Base case This case corresponds to the axiom EMPTYBLOCKCHAIN in Section 7.4. The height of the derivation is 1, and $\beta = \emptyset$, that is $\Gamma \vdash_1 \emptyset$. $\emptyset$ is well-formed regardless of Γ, and hence it is also true $\Gamma' \vdash_1 \emptyset$.

Inductive cases Given a judgment J such that its derivation has height $k + 1$, we prove the inductive cases on the last rule used to derive J. We assume the lemma for the judgments with height at most k and we prove it for those with height $k + 1$.

- VARIABLE. In this case $J = \Gamma \vdash_{k+1} \beta \cdot [x \mapsto v]$ was derived from $\Gamma \vdash_k \beta$, $\Gamma \vdash_k x : T$, and $\Gamma \vdash_k v : T'$, where $x \notin \mathrm{dom}(\beta)$ and $T' <: T$. By inductive hypothesis, also the judgments $\Gamma' \vdash_k \beta$, $\Gamma' \vdash_k x : T$, and $\Gamma' \vdash_k v : T'$ are derivable. Applying VARIABLE we can thus derive $\Gamma' \vdash_{k+1} \beta \cdot [x \mapsto v]$, which is what we wanted to prove.

- CONTRACT. In this case $J = \Gamma \vdash_{k+1} \beta \cdot [(c, a) \mapsto (C, s\tilde{:}v, n)]$ was derived from $\Gamma \vdash_k \beta$, $\Gamma \vdash_k c : C$, $\Gamma \vdash_k a : \mathsf{address}$, $\Gamma \vdash_k n : \mathsf{uint}$, and $\Gamma \vdash_k \tilde{v} : \tilde{T'}$, where $(c, a) \notin \mathrm{dom}(\beta)$, $\mathrm{sv}(C) = \tilde{T}\, s$, and $\tilde{T'} <: \tilde{T}$. By inductive hypothesis, also the judgments $\Gamma' \vdash_k \beta$, $\Gamma' \vdash_k c : C$, $\Gamma' \vdash_k a : \mathsf{address}$, $\Gamma' \vdash_k n : \mathsf{uint}$, and $\Gamma' \vdash_k \tilde{v} : \tilde{T'}$ are derivable. Applying CONTRACT we can thus derive $\Gamma' \vdash_{k+1} \beta \cdot [(c, a) \mapsto (C, s\tilde{:}v, n)]$, which is what we wanted to prove.

□

F.2.2 Weakening of σ

Here we prove the projection of the lemma of Weakening on the second component of $\langle \beta, \sigma, e \rangle$: σ.

Lemma 11 (Weakening of σ)**.**
Let $\Gamma \vdash \sigma$ be a derivable judgment, and let Δ be such that $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$. Then $\Gamma \cdot \Delta \vdash \sigma$ can be derived and its derivation has the same height as the previous one.

Proof. We prove this lemma by induction on the height of the derivation of the judgment $\Gamma \vdash \sigma$.

Base case Rule CALLSTACK in Section 7.4 has, as a base case, the well-formedness of β, proven by Lemma 10.

Inductive case Let the judgment $\Gamma \vdash \sigma \cdot a$ have height $k + 1$. We assume the lemma for the judgments with height at most k and we prove it for those with height at most $k + 1$.

$\Gamma \vdash_{k+1} \sigma \cdot a$ was derived from $\Gamma \vdash_k \sigma$ and $\Gamma \vdash_k a : \mathsf{address}\langle \mathsf{C} \rangle$. By inductive hypothesis we can derive $\Gamma' \vdash_k \sigma$ and $\Gamma' \vdash_k a : \mathsf{address}\langle \mathsf{C} \rangle$, and applying CALLSTACK we obtain a derivation of $\Gamma' \vdash_{k+1} \sigma \cdot a$. □

F.2.3 Weakening of e

Here we prove the projection of the lemma of Weakening on the first component of $\langle \beta, \sigma, e \rangle$: e.

Lemma 12 (Weakening of e)**.**
Let $\Gamma \vdash e : T$ be a derivable judgment, and let Δ be such that $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$. Then $\Gamma \cdot \Delta \vdash e : T$ can be derived and its derivation has the same height as the previous one.

Proof. We prove this lemma by induction on the height of the derivation of the judgment $\Gamma \vdash e : T$. We shall rewrite the proof only for the rules in Section 7.4, that defines the changes with respect to the rules in Section 6.3. All the cases not showed here are as in Appendix E.2.

Base cases The only modified axiom is ADDRESS, whose derivation has height 1.

- ADDRESS. The judgment is $\Gamma \vdash a : \mathsf{address}\langle \mathsf{C} \rangle$. From Case 5 of Lemma 7 we know that $a : \mathsf{address}\langle \mathsf{C} \rangle \in \Gamma$. We defined Γ' as $\Gamma \cdot \Delta$, so $a : \mathsf{address}\langle \mathsf{C} \rangle \in \Gamma \Rightarrow a : \mathsf{address}\langle \mathsf{C} \rangle \in \Gamma'$. Hence, $\Gamma' \vdash a : \mathsf{address}\langle \mathsf{C} \rangle$ is derivable.

Inductive cases Given a judgment J such that its derivation has height $k + 1$, we prove the inductive cases on the last rule used to derive J. We assume the lemma for the judgments with height at most k and we prove it for those with height $k + 1$.

- FUN. In this case $J = \Gamma \vdash_{k+1} c.f : \tilde{T_1} \to T_2$ was derived from the judgment $\Gamma \vdash_k c : C$ with the premise $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$. Furthermore, $\Gamma \vdash_k \mathsf{this} : C'$, where $C' <: \mathrm{fsender}(C, f)$. We can apply the inductive hypothesis to say that $\Gamma' \vdash_k c : C$ and $\Gamma \vdash_k \mathsf{this} : C'$ are derivable. Provided that the premises

$\text{ftype}(C, f) = \tilde{T_1} \to T_2$ and $C' <: \text{fsender}(C, f)$ are still valid, applying FUN we conclude that also $\Gamma' \vdash_{k+1} c.f : \tilde{T_1} \to T_2$ is derivable.

- MAPPING. In this case $J = \Gamma \vdash_{k+1} M : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from $\Gamma \vdash_k \tilde{k} : T_1'$ and $\Gamma \vdash_k \tilde{v} : T_2'$, where $T_1' <: T_1$ and $T_2' <: T_2$. By induction hypothesis, also $\Gamma' \vdash_k \tilde{k} : T_1'$ and $\Gamma' \vdash_k \tilde{v} : T_2'$ can be derived. Hence, applying MAPPING we obtain a derivation of $\Gamma' \vdash_{k+1} M : \mathsf{mapping}(T_1 \Rightarrow T_2)$.

- BAL. In this case $J = \Gamma \vdash_{k+1} \mathsf{balance}(e) : \mathsf{uint}$ was derived from $\Gamma \vdash_k e : \mathsf{address}(C)$. By inductive hypothesis, $\Gamma' \vdash_k e : \mathsf{address}\langle \mathsf{C} \rangle$ is derivable. We then apply BAL to conclude that the judgment $\Gamma' \vdash_{k+1} \mathsf{balance}(e) : \mathsf{uint}$ can be derived.

- ADDR. In this case $J = \Gamma \vdash_{k+1} \mathsf{address}(e) : \mathsf{address}\langle \mathsf{C} \rangle$ was derived from $\Gamma \vdash_k e : C$. By inductive hypothesis, $\Gamma' \vdash_k e : C$ is derivable. We then apply ADDR to conclude that the judgment $\Gamma' \vdash_{k+1} \mathsf{address}(e) : \mathsf{address}\langle \mathsf{C} \rangle$ can be derived, too.

- IF. In this case $J = \Gamma \vdash_{k+1} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ was derived from $\Gamma \vdash_k e_1 : \mathsf{bool}$, $\Gamma \vdash_k e_2 : T_1$, and $\Gamma \vdash_k e_3 : T_2$, with $T_1 <: T$ and $T_2 <: T$. We can thus apply the inductive hypothesis and say that $\Gamma' \vdash_k e_1 : \mathsf{bool}$, $\Gamma' \vdash_k e_2 : T_1$, and $\Gamma' \vdash_k e_3 : T_2$ are all derivable. We then apply IF to derive $\Gamma' \vdash_{k+1} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$.

- DECL. In this case $J = \Gamma \vdash_{k+1} T_1\ x = e_1; e_2 : T_2$ was derived from $\Gamma \vdash_k e_1 : T_1'$, where $T_1' <: T_1$, and $\Gamma, x : T_1 \vdash_k e_2 : T_2$. On the former we can apply the inductive hypothesis and say that $\Gamma' \vdash_k e_1 : T_1'$ is derivable. On the contrary, we cannot apply the inductive hypothesis on the latter, since the context is $\Gamma, x : T_1$ and not Γ. Still, we know there exists a derivation of height k for this judgment, otherwise J would not be derivable, but this is a contradiction, since we assumed J has a valid derivation of height $k + 1$. Hence, it comes from another judgment J' having a derivation of height $k - 1$ where any of the type rules was applied as a last step. We can now apply the inductive hypothesis on J', considering $\Gamma, x : T_1$ as a context and concluding that $\Gamma, x : T_1, \Delta \vdash_k e_2 : T_2$ is derivable. Lastly, applying DECL we obtain a derivation of $\Gamma' \vdash_{k+1} T_1\ x = e_1; e_2 : T_2$.

- MAPPSEL. In this case $J = \Gamma \vdash_{k+1} e_1[e_2] : T_2$ was derived from $\Gamma \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Gamma \vdash_k e_2 : T_1'$, where $T_1' <: T_1$. By inductive hypothesis also the judgments $\Gamma' \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Gamma' \vdash_k e_2 : T_1'$ are derivable. Applying MAPPSEL we obtain a derivation of $\Gamma' \vdash_{k+1} e_1[e_2] : T_2$.

- ASS. In this case $J = \Gamma \vdash_{k+1} x = e : T$ was derived from $\Gamma \vdash_k x : T$ and $\Gamma \vdash_k e : T'$, where $T' <: T$. By inductive hypothesis we can derive with the same height also $\Gamma' \vdash_k x : T$ and $\Gamma' \vdash_k e : T'$, and applying ASS we obtain a derivation of $\Gamma' \vdash_{k+1} x = e : T$.

- MAPPASS. In this case $J = \Gamma \vdash_{k+1} e_1[e_2 \to e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from $\Gamma \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash_k e_2 : T_1'$, and $\Gamma \vdash_k e_3 : T_2'$, where $T_1' <: T_1$ and $T_2' <: T_2$. By inductive hypothesis we can derive with the same height also $\Gamma' \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma' \vdash_k e_2 : T_1'$, and $\Gamma' \vdash_k e_3 : T_2'$, and applying MAPPASS we obtain a derivation of $\Gamma' \vdash_{k+1} e_1[e_2 \to e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)$.

- StateAss. In this case $J = \Gamma \vdash_{k+1} e_1.s = e_2 : T$ was derived from $\Gamma \vdash_k e_1.s : T$ and $\Gamma \vdash_k e_2 : T'$, where $T' <: T$. By inductive hypothesis we can derive with the same height also $\Gamma' \vdash_k e_1.s : T$ and $\Gamma' \vdash_k e_2 : T'$, and applying StateAss we obtain a derivation of $\Gamma' \vdash_{k+1} e_1.s = e_2 : T$.

- New. In this case $J = \Gamma \vdash_{k+1} \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : C$ was derived from $\Gamma \vdash_k \tilde{e} : \tilde{T}'$, where $\tilde{T}' <: \tilde{T}$, and $\Gamma \vdash_k e' : \mathsf{uint}$, together with the premise $|\tilde{e}| = |\tilde{s}|$, where $\mathrm{sv}(C) = \tilde{T}\,\tilde{s}$. By inductive hypothesis, $\Gamma' \vdash_k \tilde{e} : \tilde{T}'$ and $\Gamma' \vdash_k e' : \mathsf{uint}$ can be derived. Furthermore, the premise checking the length of $\tilde{e}$ and $\tilde{s}$ is still valid, and we can apply New to derive $\Gamma' \vdash_{k+1} \mathsf{new}\ C.\mathsf{value}(e')(\tilde{e}) : C$.

- ContrRetr. In this case $J = \Gamma \vdash_{k+1} D(e) : D$ was derived from $\Gamma \vdash_k e : \mathsf{address}\langle \mathsf{C} \rangle$, where $C <: D$. By inductive hypothesis also $\Gamma' \vdash_k e : \mathsf{address}\langle \mathsf{C} \rangle$ is derivable; applying ContrRetr we then obtain a derivation of $\Gamma' \vdash_{k+1} D(e) : D$.

- Transfer. In this case $J = \Gamma \vdash_{k+1} e_1.\mathsf{transfer}(e_2) : \mathsf{unit}$ was derived from $\Gamma \vdash_k e_1 : \mathsf{address}\langle \mathsf{C} \rangle$, $\Gamma \vdash_k e_2 : \mathsf{uint}$, and $\Gamma \vdash_k \mathsf{this} : C'$. Furthermore, $\mathrm{ftype}(C, fb) = \{\} \to \mathsf{unit}$ ensures that C defined a fallback, whereas $C' <: \mathrm{fsender}(C, fb)$ ensures that the sender a run-time will be a subcontract of the required one. By inductive hypothesis we can derive $\Gamma' \vdash_k e_1 : \mathsf{address}\langle \mathsf{C} \rangle$, $\Gamma' \vdash_k e_2 : \mathsf{uint}$, and $\Gamma \vdash_k \mathsf{this} : C'$. The other two premises are still valid, and applying Transfer we obtain a derivation of $\Gamma' \vdash_{k+1} e_1.\mathsf{transfer}(e_2) : \mathsf{unit}$.

- Call. In this case $J = \Gamma \vdash_{k+1} e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$ was derived from $\Gamma \vdash_k e_1 : C$, $\Gamma \vdash_k e_2 : \mathsf{uint}$, $\Gamma \vdash_k \tilde{e} : \tilde{T}_1$, where $\tilde{T}_1 <: \tilde{T}_1$, and $\Gamma \vdash_k \mathsf{this} : C'$, together with the premises checking the length of the tuple $\tilde{e}$ ($|\tilde{e}| = |\tilde{T}_1|$), the type of f in C ($\mathrm{ftype}(C, f) = \tilde{T}_1 \to T_2$), and the constraint on the sender's type ($C' <: \mathrm{fsender}(C, f)$). By induction hypothesis we can derive $\Gamma' \vdash_k e_1 : C$, $\Gamma' \vdash_k e_2 : \mathsf{uint}$, $\Gamma' \vdash_k \tilde{e} : \tilde{T}_1$, and $\Gamma \vdash_k \mathsf{this} : C'$. Furthermore, the other three premises are still valid, and thus we can apply Call to derive $\Gamma' \vdash_{k+1} e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$.

- CallTopLevel. In this case $J = \Gamma \vdash_{k+1} e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T_2$ was derived from $\Gamma \vdash_k e_1 : C$, $\Gamma \vdash_k e_2 : \mathsf{uint}$, $\Gamma \vdash_k \tilde{e} : \tilde{T}_1$, where $\tilde{T}_1 <: \tilde{T}_1$, and $\Gamma \vdash_k e_3 : \mathsf{address}\langle \mathsf{C}' \rangle$, where $C' <: \mathrm{fsender}(C, f)$, together with the premises checking the length of the tuple $\tilde{e}$ ($|\tilde{e}| = |\tilde{T}_1|$), the type of f in C ($\mathrm{ftype}(C, f) = \tilde{T}_1 \to T_2$). By induction hypothesis we can derive $\Gamma' \vdash_k e_1 : C$, $\Gamma' \vdash_k e_2 : \mathsf{uint}$, $\Gamma' \vdash_k \tilde{e} : \tilde{T}_1$, and $\Gamma \vdash_k e_3 : \mathsf{address}\langle \mathsf{C}' \rangle$. Furthermore, the other two premises are still valid, and thus we can apply CallTopLevel to derive $\Gamma' \vdash_{k+1} e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) : T_2$.

- CallValue. In this case $J = \Gamma \vdash_{k+1} e_1.\mathsf{value}(e_2)(\tilde{e}) : T_2$ was derived from $\Gamma \vdash_k e_1 : \tilde{T}_1 \to T_2$, $\Gamma \vdash_k e_2 : \mathsf{uint}$, and $\Gamma \vdash_k \tilde{e} : \tilde{T}_1'$, where $\tilde{T}_1' <: \tilde{T}_1$. There is another premise, checking the length of the tuple $\tilde{e}$ ($|\tilde{e}| = |\tilde{T}_1|$). By induction hypothesis, the judgments $\Gamma' \vdash_k e_1 : \tilde{T}_1 \to T_2$, $\Gamma' \vdash_k e_2 : \mathsf{uint}$, and $\Gamma' \vdash_k \tilde{e} : \tilde{T}_1'$ are derivable. Furthermore, the other premise is still valid, and we can thus apply CallValue to obtain a derivation of $\Gamma' \vdash_{k+1} e_1.\mathsf{value}(e_2)(\tilde{e}) : T_2$.

□

F.2.4 Proof of the Lemma

We can now prove Lemma 4:

Lemma 4 (Weakening)**.**
Let $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ *be a derivable judgment, and let* Δ *be such that* $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$ *(i.e.* Γ *and* Δ *have no elements in common). Then* $\Gamma \cdot \Delta \vdash \langle \beta, \sigma, e \rangle : T$ *can be derived and its derivation has the same height as the previous one.*

Proof. By hypothesis $\Gamma \vdash_{k+1} \langle \beta, \sigma, e \rangle : T$: by rule CONFIGURATION this means that also $\Gamma \vdash_k \beta$, $\Gamma \vdash_k \sigma$, and $\Gamma \vdash_k e : T'$, where $T' <: T$ are derivable. By, respectively, Lemma 10, 11, and 12 we know that there exists a derivation for $\Gamma' \vdash_k \beta$, $\Gamma' \vdash_k \sigma$, and $\Gamma' \vdash_k e : T'$. Hence, applying CONFIGURATION to the latter three judgments we can derive $\Gamma' \vdash_{k+1} \langle \beta, \sigma, e \rangle : T$, which is what we wanted to prove. □

F.3 Substitution Lemma

We recall Lemma 9:

Lemma 9 (Substitution)**.**
If $\Gamma, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}\langle \mathsf{D} \rangle, \mathsf{msg.value} : \mathsf{uint} \vdash e : T$, $\Gamma \vdash c : C'$, $\Gamma \vdash a : \mathsf{address}\langle \mathsf{D}' \rangle$*, and* $\Gamma \vdash n : \mathsf{uint}$*, with* $D' <: D$*, then*
$\Gamma \vdash e\{\mathsf{this} := c, \mathsf{msg.sender} := a, \mathsf{msg.value} := n\} : T$.

Proof. Let $\Gamma' = \Gamma, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}\langle \mathsf{D} \rangle, \mathsf{msg.value} : \mathsf{uint}$. We prove this lemma by induction on the height of the judgment $\Gamma' \vdash e : T$.

We shall make use of the following notation to make the proof more readable:

$$\mathrm{subst}(e) = e\{\mathsf{this} := c, \mathsf{msg.sender} := a, \mathsf{msg.value} := n\}$$

We shall use identifiers C, C', D, D', and E to indicate contract names.

Base cases Only one case, corresponding to rule ADDRESS, has changed with respect to Appendix E.3. The height of its derivation is 1.

- ADDRESS. In this case the judgment is $\Gamma' \vdash a' : \mathsf{address}\langle \mathsf{E} \rangle$, where $a' \neq \mathsf{msg.sender}$. In this case $\mathrm{subst}(a') = a'$. By Case 5 of Lemma 7 we know $a' : \mathsf{address}\langle \mathsf{E} \rangle \in \Gamma'$, and thus $a' : \mathsf{address}\langle \mathsf{E} \rangle \in \Gamma$, because $\Gamma' = \Gamma, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}\langle \mathsf{D} \rangle, \mathsf{msg.value} : \mathsf{uint}$, and $a' \neq \mathsf{this}$, $a' \neq \mathsf{msg.sender}$, and $a' \neq \mathsf{msg.value}$. We are to prove $\Gamma \vdash a' : \mathsf{address}\langle \mathsf{E} \rangle$, which is true because $a' : \mathsf{address}\langle \mathsf{E} \rangle \in \Gamma$.

Inductive cases Given a judgment J such that its derivation has height $k + 1$, we prove the inductive cases on the last rule used to derive J. We assume the lemma for the judgments with height at most k and we prove it for those with height $k + 1$. We shall omit the unchanged cases with respect to Appendix E.3.

- FUN. In this case $J = \Gamma' \vdash_{k+1} d.f : \tilde{T_1} \to T_2$ was derived from $\Gamma' \vdash_k d : E$ and $\Gamma' \vdash_k \mathsf{this} : C$, with the additional premise stating $C <: \mathrm{fsender}(E, f)$. We again have two sub-cases:

  - if $d \neq \mathsf{this}$ then, by Case 8 of Lemma 7 we obtain $d : E \in \Gamma'$. We supposed $d \neq \mathsf{this}$, but it is also true that $d \neq \mathsf{msg.sender}$ and $d \neq \mathsf{msg.value}$. Hence, $d : E \in \Gamma' \Rightarrow d : E \in \Gamma \Rightarrow \Gamma \vdash d : E$. Furthermore, $\mathrm{subst}(d.f) = d.f$, and the judgment $\Gamma \vdash d.f : \tilde{T_1} \to T_2$ is derivable, since the other premises have remained valid.
  - id $d = \mathsf{this}$ then, by Case 8 of Lemma 7 we know $E = C$. We are to prove $\Gamma \vdash \mathrm{subst}(\mathsf{this}.f) : E$, but $\mathrm{subst}(\mathsf{this}.f) = c.f$ and the judgment becomes $\Gamma \vdash c.f : \tilde{T_1} \to T_2$. By the second hypothesis we know we can derive $\Gamma \vdash c : C$ and applying FUN we can also derive $\Gamma \vdash c.f : \tilde{T_1} \to T_2$, which is what we wanted to prove since $\mathrm{subst}(\mathsf{this}.f) = c.f$.

- MAPPING. In this case $J = \Gamma' \vdash_{k+1} M : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from $\Gamma' \vdash_k \tilde{k} : \tilde{T_1'}$ and $\Gamma' \vdash_k \tilde{v} : \tilde{T_2'}$, where $\tilde{T_1'} <: T_1$ and $\tilde{T_2'} <: T_2$. Since both $\tilde{k}$ and $\tilde{v}$ are tuples of values, $\mathrm{subst}(\tilde{k}) = \tilde{k}$ and $\mathrm{subst}(\tilde{v}) = \tilde{v}$. By inductive hypothesis, $\Gamma \vdash \tilde{k} : \tilde{T_1'}$ and $\Gamma \vdash \tilde{v} : \tilde{T_2'}$, and applying MAPPING we derive $\Gamma \vdash M : \mathsf{mapping}(T_1 \Rightarrow T_2)$, as required.

- BAL. In this case $J = \Gamma' \vdash_{k+1} \mathsf{balance}(e) : \mathsf{uint}$ was derived from $\Gamma' \vdash_k e : \mathsf{address}\langle\mathsf{E}\rangle$. By inductive hypothesis we have $\Gamma \vdash \mathrm{subst}(e) : \mathsf{address}\langle\mathsf{E}\rangle$, and applying BAL we obtain $\Gamma \vdash \mathsf{balance}(\mathrm{subst}(e)) : \mathsf{uint}$, which is the same as $\Gamma \vdash \mathrm{subst}(\mathsf{balance}(e)) : \mathsf{uint}$ (see Figure 4.3).

- ADDR. In this case $J = \Gamma' \vdash_{k+1} \mathsf{address}(e) : \mathsf{address}\langle\mathsf{E}\rangle$ was derived from $\Gamma' \vdash_k e : E$. By inductive hypothesis we have $\Gamma \vdash \mathrm{subst}(e) : E$, and applying ADDR we obtain $\Gamma \vdash \mathsf{address}(\mathrm{subst}(e)) : \mathsf{address}\langle\mathsf{E}\rangle$, which is the same as $\Gamma \vdash \mathrm{subst}(\mathsf{address}(e)) : \mathsf{address}\langle\mathsf{E}\rangle$ (see Figure 4.3).

- IF. In this case $J = \Gamma' \vdash_{k+1} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ was derived from $\Gamma' \vdash_k e_1 : \mathsf{bool}$, $\Gamma' \vdash_k e_2 : T_1$ and $\Gamma' \vdash_k e_3 : T_2$, with $T_1, T_2 <: T$. By inductive hypothesis, we can derive $\Gamma \vdash \mathrm{subst}(e_1) : \mathsf{bool}$, $\Gamma \vdash \mathrm{subst}(e_2) : T_1$ and $\Gamma \vdash \mathrm{subst}(e_3) : T_2$. We are to prove $\Gamma \vdash \mathrm{subst}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_2) : T$, which is derived applying IF to the latter judgments together with the equivalence $\mathsf{if}\ \mathrm{subst}(e_1)\ \mathsf{then}\ \mathrm{subst}(e_2)\ \mathsf{else}\ \mathrm{subst}(e_3) = \mathrm{subst}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_2)$.

- DECL. In this case $J = \Gamma' \vdash_{k+1} T_1\ x = e_1; e_2 : T_2$ was derived from $\Gamma' \vdash_k e_1 : T_1'$ and $\Gamma', x : T_1 \vdash_k e_2 : T_2$, with $T_1' <: T_1$. By inductive hypothesis we can derive $\Gamma \vdash \mathrm{subst}(e_1) : T_1'$. To apply the inductive hypothesis to the latter judgment we need to rearrange the context. By Lemma 3 we know that $\Gamma, x : T_1, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}\langle\mathsf{D}\rangle, \mathsf{msg.value} : \mathsf{uint} \vdash_k e_2 : T_2$ can be derived with the same height. Furthermore, we can apply Lemma 4 to the other three hypotheses to obtain a derivation of the judgments $\Gamma, x : T_1, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}\langle\mathsf{D}\rangle, \mathsf{msg.value} : \mathsf{uint} \vdash_k c : C$, $\Gamma, x : T_1, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}\langle\mathsf{D}\rangle, \mathsf{msg.value} : \mathsf{uint} \vdash_k a : \mathsf{address}\langle\mathsf{D'}\rangle$, and $\Gamma, x : T_1, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}\langle\mathsf{D}\rangle, \mathsf{msg.value} : \mathsf{uint} \vdash_k n : \mathsf{uint}$. Now, by inductive hypothesis, also $\Gamma, x : T_1, \mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}\langle\mathsf{D}\rangle, \mathsf{msg.value} : \mathsf{uint} \vdash \mathrm{subst}(e_2) : T_2$ can be derived. Lastly, by applying DECL we obtain a derivation of $\Gamma \vdash T_1\ x = \mathrm{subst}(e_1); \mathrm{subst}(e_2) : T_2$, which is the same as $\Gamma \vdash \mathrm{subst}(T_1\ x = e_1; e_2) : T_2$ since $\mathrm{subst}(T_1\ x = e_1; e_2) = \mathrm{subst}(T_1\ x = e_1); \mathrm{subst}(e_2) =$ $T_1\ x = \mathrm{subst}(e_1); \mathrm{subst}(e_2)$.

- MAPPSEL. In this case $J = \Gamma' \vdash_{k+1} e_1[e_2] : T_2$ was derived from $\Gamma' \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Gamma' \vdash_k e_2 : T_1'$, where $T_1' <: T_1$. By inductive hypothesis we derive $\Gamma \vdash \mathrm{subst}(e_1) : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Gamma \vdash \mathrm{subst}(e_2) : T_1'$; then, applying MAPPSEL, we obtain a derivation of $\Gamma \vdash \mathrm{subst}(e_1)[\mathrm{subst}(e_2)] : T_2$, which is what we wanted to prove since $\mathrm{subst}(e_1[e_2] = \mathrm{subst}(e_1)[\mathrm{subst}(e_2)])$.

- ASS. In this case $J = \Gamma' \vdash_{k+1} x = e : T$ was derived from $\Gamma' \vdash_k x : T$ and $\Gamma' \vdash_k e : T'$, where $T' <: T$. By inductive hypothesis we have $\Gamma \vdash \mathrm{subst}(x) : T$ and $\Gamma \vdash \mathrm{subst}(e) : T'$; applying ASS we obtain $\Gamma \vdash \mathrm{subst}(x) = \mathrm{subst}(e) : T$, which is the same as $\Gamma \vdash \mathrm{subst}(x = e) : T$ since $\mathrm{subst}(x) = x$ and $\mathrm{subst}(x = e) = (x = \mathrm{subst}(e))$.

- MAPPASS. In this case $J = \Gamma' \vdash_{k+1} e_1[e_2 \to e_3] : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from $\Gamma' \vdash_k e_1 : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma' \vdash_k e_2 : T_1'$, and $\Gamma' \vdash_k e_3 : T_2'$, where $T_1' <: T_1$ and $T_2' <: T_2$. By inductive hypothesis we derive $\Gamma \vdash \mathrm{subst}(e_1) : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash \mathrm{subst}(e_2) : T_1'$, and $\Gamma \vdash \mathrm{subst}(e_3) : T_2'$. By MAPPASS we obtain $\Gamma \vdash \mathrm{subst}(e_1)[\mathrm{subst}(e_2) \to \mathrm{subst}(e_3)] : \mathsf{mapping}(T_1 \Rightarrow T_2)$, which is what we wanted to prove since $\mathrm{subst}(e_1[e_2 \to e_3]) = \mathrm{subst}(e_1)[\mathrm{subst}(e_2) \to \mathrm{subst}(e_3)]$.

- STATEASS. In this case $J = \Gamma' \vdash_{k+1} e_1.s = e_2 : T$ was derived from $\Gamma' \vdash_k e_1.s : T$ and $\Gamma' \vdash_k e_2 : T'$, where $T' <: T$. By inductive hypothesis we obtain $\Gamma \vdash \mathrm{subst}(e_1.s) : T$ and $\Gamma \vdash \mathrm{subst}(e_2) : T'$; then, applying STATEASS we derive $\Gamma \vdash \mathrm{subst}(e_1).s = \mathrm{subst}(e_2) : T$, which is what we wanted to prove since $\mathrm{subst}(e_1.s = e_2) = \mathrm{subst}(e_1).s = \mathrm{subst}(e_2)$.

- NEW. In this case $J = \Gamma' \vdash_{k+1} \mathsf{new}\ C.\mathsf{value}(e_1)(\tilde{e}) : C$ was derived from $\Gamma' \vdash_k \tilde{e} : \tilde{T}'$ and $\Gamma' \vdash_k e_1 : \mathsf{uint}$, where $\mathrm{sv}(C) = \tilde{T}\,s$, $|\tilde{e}| = |\tilde{s}|$, and $\tilde{T}' <: \tilde{T}$. By inductive hypothesis we obtain $\Gamma \vdash \mathrm{subst}(\tilde{e}) : \tilde{T}'$ and $\Gamma \vdash \mathrm{subst}(e_1) : \mathsf{uint}$; applying NEW we conclude $\Gamma \vdash \mathrm{subst}(\mathsf{new}\ C.\mathsf{value}(e_1)(\tilde{e})) : C$, since $\mathrm{subst}(\mathsf{new}\ C.\mathsf{value}(e_1)(\tilde{e})) = \mathsf{new}\ C.\mathsf{value}(\mathrm{subst}(e_1))(\mathrm{subst}(\tilde{e}))$.

- CONTRRETR. In this case $J = \Gamma' \vdash_{k+1} E(e) : E$ was derived from $\Gamma' \vdash_k e : \mathsf{address}\langle \mathsf{E}' \rangle$, where $E' <: E$. By inductive hypothesis we have $\Gamma \vdash \mathrm{subst}(e) : \mathsf{address}\langle \mathsf{E}' \rangle$, and applying CONTRRETR we obtain $\Gamma \vdash E(\mathrm{subst}(e)) : E$, which is the same as $\Gamma \vdash \mathrm{subst}(E(e)) : E$.

- TRANSFER. In this case $J = \Gamma' \vdash_{k+1} e_1.\mathsf{transfer}(e_2) : \mathsf{unit}$ was derived from $\Gamma' \vdash_k e_1 : \mathsf{address}\langle \mathsf{C} \rangle$, $\Gamma' \vdash_k e_2 : \mathsf{uint}$, and $\Gamma' \vdash_k \mathsf{this} : C'$, where $C' <: \mathrm{fsender}(C, fb)$. By inductive hypothesis we get $\Gamma \vdash \mathrm{subst}(e_1) : \mathsf{address}\langle \mathsf{C} \rangle$ and $\Gamma \vdash \mathrm{subst}(e_2) : \mathsf{uint}$. Furthermore, applying TRANSFER we derive $\Gamma \vdash \mathrm{subst}(e_1).\mathsf{transfer}(\mathrm{subst}(e_2)) : \mathsf{unit}$, which is the same as $\Gamma \vdash \mathrm{subst}(e_1.\mathsf{transfer}(e_2)) : \mathsf{unit}$.

- CALL. In this case $J = \Gamma' \vdash_{k+1} e_1.f.\mathsf{value}(e_2)(\tilde{e}) : T_2$ was derived from $\Gamma' \vdash_k e_1 : C$, $\Gamma' \vdash_k e_2 : \mathsf{uint}$, and $\Gamma' \vdash_k \tilde{e} : \tilde{T}_1'$, where $\mathrm{ftype}(C, f) = \tilde{T}_1 \to T_2$, $|\tilde{e}| = |\tilde{T}_1|$, and $\tilde{T}_1' <: \tilde{T}_1$. By inductive hypothesis we get $\Gamma \vdash \mathrm{subst}(e_1) : C$, $\Gamma \vdash \mathrm{subst}(e_2) : \mathsf{uint}$, and $\Gamma \vdash \mathrm{subst}(\tilde{e}) : \tilde{T}_1'$: applying CALL we derive $\Gamma \vdash \mathrm{subst}(e_1).f.\mathsf{value}(\mathrm{subst}(e_2))(\mathrm{subst}(\tilde{e})) : T_2$, which is what we wanted to prove since $\mathrm{subst}(e_1.f.\mathsf{value}(e_2)(\tilde{e})) = \mathrm{subst}(e_1).f.\mathsf{value}(\mathrm{subst}(e_2))(\mathrm{subst}(\tilde{e}))$.

- CALLTOPLEVEL. In this case is very similar to the previous one, with the only addition of the judgment $\Gamma \vdash e_3 : \mathsf{address}\langle \mathsf{C}' \rangle$ checking the well-typing of the third parameter.

- CALLVALUE. In this case $J = \Gamma' \vdash_{k+1} e_1.\mathsf{value}(e_2)(\tilde{e}) : T_2$ was derived from $\Gamma' \vdash_k e_1 : \tilde{T_1} \to T_2$, $\Gamma' \vdash_k e_2 : \mathsf{uint}$, and $\Gamma' \vdash_k \tilde{e} : \tilde{T_1'}$, where $|\tilde{e}| = |\tilde{T_1}|$ and $\tilde{T_1'} <: \tilde{T_1}$. By inductive hypothesis we get $\Gamma \vdash \mathrm{subst}(e_1) : \tilde{T_1} \to T_2$, $\Gamma \vdash \mathrm{subst}(e_2) : \mathsf{uint}$, and $\Gamma \vdash \mathrm{subst}(\tilde{e}) : \tilde{T_1'}$: applying CALLVALUE we derive $\Gamma \vdash \mathrm{subst}(e_1).\mathsf{value}(\mathrm{subst}(e_2))(\mathrm{subst}(\tilde{e})) : T_2$, which is what we wanted to prove since $\mathrm{subst}(e_1.\mathsf{value}(e_2)(\tilde{e})) = \mathrm{subst}(e_1).\mathsf{value}(\mathrm{subst}(e_2))(\mathrm{subst}(\tilde{e}))$.

□

F.4 Progress Theorem

We recall Theorem 3:

Theorem 3 (Progress)**.**
If $\langle \beta, \sigma, e \rangle$ is closed (Definition 13) and well-typed (i.e. $\exists \Gamma$ such that $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ is derivable), then either $e = v$, $e = \mathsf{revert}$, or $\exists (\beta', \sigma', e')$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$.

Proof. We prove this theorem by induction on the height of the judgment $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$.

Base cases These cases correspond to the axioms in Section 6.3. By rule CONFIGURATION, $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ implies $\Gamma \vdash \beta$ and $\Gamma \vdash \sigma$. These cases modify neither β nor σ, and hence we focus on the expression e (i.e. on the judgment $\Gamma \vdash e : T$).

- REF. The judgment is $\Gamma \vdash c : C$, and c is already a value.

- VAR. The judgment is $\Gamma \vdash x : T$. By hypothesis, we know $\Gamma \vdash x : T$, which implies, by rule VAR, $x : T \in \Gamma$, which means $x \in \mathrm{dom}(\Gamma)$. By Lemma 13, $x \in \mathrm{dom}(\Gamma) \Rightarrow x \in \mathrm{dom}(\beta)$, which makes $\beta(x)$ well-defined. By rule VAR in Section 5.5, $\exists e' = \beta(x)$ such that $\langle \beta, \sigma, x \rangle \longrightarrow \langle \beta, \sigma, \beta(x) \rangle$.

- TRUE. The judgment is $\Gamma \vdash \mathsf{true} : \mathsf{bool}$, and the expression true is already a value.

- FALSE, NAT, ADDRESS, and UNIT. The judgments are $\Gamma \vdash \mathsf{false} : \mathsf{bool}$, $\Gamma \vdash n : \mathsf{uint}$, $\Gamma \vdash a : \mathsf{address}\langle \mathsf{C} \rangle$, or $\Gamma \vdash \mathsf{u} : \mathsf{unit}$, respectively, and the expressions false, n, a, and u are already values.

- REVERT. The judgment is $\Gamma \vdash \mathsf{revert} : T$, and the expression is a revert.

Inductive cases Given a judgment J such that its derivation has height $k + 1$, we prove the inductive cases on the last rule used to derive J. We assume the theorem for the judgments with height at most k and we prove it for those with height $k + 1$. These cases correspond to the rules in Section 6.3, with the modified CONTRRETR given in Section 7.4.

- MAPPING. The judgment is $\Gamma \vdash M : \mathsf{mapping}(T_1 \Rightarrow T_2)$, and the expression M is already a value.

- FUN. The judgment is $\Gamma \vdash c.f : \tilde{T_1} \to T_2$, and the expression $c.f$ is already a value.

- BAL. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, \mathsf{balance}(e) \rangle : \mathsf{uint}$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e \rangle : \mathsf{address\langle C \rangle}$. By inductive hypothesis we know e is either a value, revert, or $\exists \beta', \sigma', e'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$. We distinguish the following cases:

  - if $e = v$ then, by Case 1 of Lemma 8, v is an address a. By rule BALANCE in Section 5.5, $\langle \beta, \sigma, \mathsf{balance}(a) \rangle \longrightarrow \langle \beta, \sigma, n \rangle$, where $\beta(a) = (C, s \tilde{:} v, n)$ is well-defined by Lemma 13.
  - if $e = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{balance}(e) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, \mathsf{balance}(e) \rangle \longrightarrow \langle \beta', \sigma', \mathsf{balance}(e') \rangle$.

- ADDR. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, \mathsf{address}(e) \rangle : \mathsf{address\langle C \rangle}$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e \rangle : C$. By inductive hypothesis we know e is either a value, revert, or $\exists \beta', \sigma', e'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$. We distinguish the following cases:

  - if $e = v$ then, by Case 5 of Lemma 6, v is a contract reference c. By rule ADDRESS in Section 5.5, $\langle \beta, \sigma, \mathsf{address}(c) \rangle \longrightarrow \langle \beta, \sigma, a \rangle$, where $\hat{\beta}(c) = a$. c belongs for sure to $\mathrm{dom}(\beta)$ by Lemma 13.
  - if $e = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{address}(e) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, \mathsf{address}(e) \rangle \longrightarrow \langle \beta', \sigma', \mathsf{address}(e') \rangle$.

- IF. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rangle : T$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : \mathsf{bool}$, $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : T_1$, and $\Gamma \vdash_k \langle \beta, \sigma, e_3 \rangle : T_2$, where $T_1 <: T$ and $T_2 <: T$. By inductive hypothesis we know e_1 is either a value, a revert, or $\exists \beta', \sigma', e_1'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e_1' \rangle$. We distinguish the following cases:

  - if $e_1 = v$ then, by Case 1 of Lemma 6, v is either true or false. On the one hand, if it is true, by rule IF-TRUE in Section 5.5 $\langle \beta, \sigma, \mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3 \rangle \longrightarrow \langle \beta, \sigma, e_2 \rangle$. On the other hand, if it is false, by rule IF-FALSE in Section 5.5 $\langle \beta, \sigma, \mathsf{if\ false\ then}\ e_2\ \mathsf{else}\ e_3 \rangle \longrightarrow \langle \beta, \sigma, e_3 \rangle$.
  - if $e_1 = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{if\ revert\ then}\ e_2\ \mathsf{else}\ e_3 \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta', \sigma', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rangle \longrightarrow \langle \beta', \sigma', \mathsf{if}\ e_1'\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rangle$.

- DECL. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, T_1\ x = e_1; e_2 \rangle : T_2$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : T_1'$, where $T_1' <: T_1$, and $\Gamma, x : T_1 \vdash_k \langle \beta, \sigma, e_2 \rangle : T_2$. By inductive hypothesis, e_1 is either a value, revert, or $\exists \beta_1', \sigma_1', e_1'$ such that $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$. We distinguish the following cases:

  - if $e_1 = v_1$ then, by rule DECL in Section 5.5, $\langle \beta, \sigma, T_1\ x = v_1; e_2 \rangle \longrightarrow \langle \beta \cdot [x \mapsto v], \sigma, v_1; e_2 \rangle$. Note that the premise $x \notin \mathrm{dom}(\beta)$ is always verified since variables can be renamed by α-equivalence.
  - if $e_1 = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, T_1\ x = \mathsf{revert}; e_2 \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert}; e_2 \rangle$. Note that this case does not take into account e_2 at all.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta', \sigma', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, T_1\ x = e_1; e_2 \rangle \longrightarrow \langle \beta', \sigma', T_1\ x = e_1'; e_2 \rangle$.

- ASS. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, x = e \rangle : T$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, x \rangle : T$ and $\Gamma \vdash_k \langle \beta, \sigma, e \rangle : T'$, where $T <: T'$. By inductive hypothesis, e is either a value, revert, or $\exists \beta', \sigma', e'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$. We distinguish the following cases:

  - if $e = v$ then, by rule ASS in Section 5.5, $\langle \beta, \sigma, x = v \rangle \longrightarrow \langle \beta[x \mapsto v], \sigma, v \rangle$. Note that $x \in \mathrm{dom}(\beta)$ is always satisfied for what we said in Lemma 13.
  - if $e = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, x = \mathsf{revert} \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, x = e \rangle \longrightarrow \langle \beta', \sigma', x = e' \rangle$.

- MAPPSEL. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1[e_2] \rangle : T_2$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : \mathsf{mapping}(T_1 \Rightarrow T_2)$ and $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : T_1'$, where $T_1' <: T_1$. By inductive hypothesis we know e_1 is either a value, revert, or $\exists \beta_1', \sigma_1', e_1'$ such that $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$. The same reasoning applies to e_2, thus e_2 is either a value, revert, or $\exists \beta_2', \sigma_2', e_2'$ such that $\langle \beta, \sigma, e_2 \rangle \longrightarrow \langle \beta_2', \sigma_2', e_2' \rangle$. We distinguish the following cases:

  - if $e_1 = v_1$ and $e_2 = v_2$ then, by Case 4 of Lemma 6, v_1 is a total function M from T_1 to T_2. By rule MAPPSEL in Section 5.5, $\langle \beta, \sigma, M[v_2] \rangle \longrightarrow \langle \beta, \sigma, M(v_2) \rangle$. Note that $M(v_1)$ is always well-defined since M is a total function.
  - if $e_1 = v_1$ and $e_2 = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, v_1[\mathsf{revert}] \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $e_1 = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{revert}[e_2] \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, e_1[e_2] \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1'[e_2] \rangle$. Note we do not have to specify what e_2 is.
  - if $e_1 = v_1$ and $\langle \beta, \sigma, e_2 \rangle \longrightarrow \langle \beta_2', \sigma_2', e_2' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, v_1[e_2] \rangle \longrightarrow \langle \beta_2', \sigma_2', v_1[e_2'] \rangle$.

- MAPPASS. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1[e_2 \to e_3] \rangle : \mathsf{mapping}(T_1 \Rightarrow T_2)$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : T_1'$, and $\Gamma \vdash_k \langle \beta, \sigma, e_3 \rangle : T_2'$, where $T_1' <: T_1$ and $T_2' <: T_2$. By inductive hypothesis we know e_1 is either a value, revert, or $\exists \beta_1', \sigma_1', e_1'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$. The same reasoning applies to e_2 and e_3. We distinguish the following cases:

  - if $e_1 = v_1$, $e_2 = v_2$, and $e_3 = v_3$ then, by Case 4 of Lemma 6, v_1 is a total function M from T_1 to T_2. By rule MAPPASS in Section 5.5, $\langle \beta, \sigma, M[v_2 \rightarrow v_3] \rangle \longrightarrow \langle \beta, \sigma, M' \rangle$, where $M' = M \backslash \{(v_1, M(v_1))\} \cup \{(v_1, v_2)\}$.
  - if either $e_2 = \mathsf{revert}$ or $e_3 = \mathsf{revert}$, with $e_1 = M$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, M[\mathsf{revert} \rightarrow v_3] \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$ or $\langle \beta, \sigma, M[v_2 \rightarrow \mathsf{revert}] \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$, respectively.
  - if $e_1 = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{revert}[e_2 \rightarrow e_3] \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, e_1[e_2 \rightarrow e_3] \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1'[e_2 \rightarrow e_3] \rangle$.
  - if $e_1 = v_1$ and $\langle \beta, \sigma, e_2 \rangle \longrightarrow \langle \beta_2', \sigma_2', e_2' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, v_1[e_2 \rightarrow e_3] \rangle \longrightarrow \langle \beta_2', \sigma_2', v_1[e_2' \rightarrow e_3] \rangle$.
  - if $e_1 = v_1$, $e_2 = v_2$, and $\langle \beta, \sigma, e_3 \rangle \longrightarrow \langle \beta_3', \sigma_3', e_3' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, v_1[v_2 \rightarrow e_3] \rangle \longrightarrow \langle \beta_3', \sigma_3', v_1[v_2 \rightarrow e_3'] \rangle$.

- NEW. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(e_1)(\tilde{e}) \rangle : C$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : \mathsf{uint}$, and $\Gamma \vdash_k \langle \beta, \sigma, \tilde{e} \rangle : \tilde{T}'$, where $\tilde{t}' <: \tilde{T}$. By inductive hypothesis we know e_1 is either a value, revert, or $\exists \beta_1', \sigma_1', e_1'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$. The same reasoning applies to the tuple $\tilde{e}$: it is either a tuple of values ($\tilde{v}$), a tuple of values and expressions separated by a revert (i.e. $v_i^{\{0 \leq i < m\}}, \mathsf{revert}, e_j^{\{m < j \leq n\}}$), or it is in the form $v_i^{\{0 \leq i < m\}}, e_j^{\{m \leq j \leq n\}}$ and $\exists \beta_i', \sigma_i', e_j'$ such that $\langle \beta, \sigma, v_i^{\{0 \leq i < m\}}, e_j^{\{m \leq j \leq n\}} \rangle \longrightarrow \langle \beta_i', \sigma_i', v_i^{\{0 \leq i < m\}}, e_j'^{\{m \leq j \leq n\}} \rangle$. We distinguish the following cases:

  - if $e_1 = v_1$ and $\tilde{e} = \tilde{v}$ then, by Case 2 of Lemma 6, $v_1 = n$. Three scenarios are possible here. If $\mathrm{Top}(\sigma) \neq \emptyset$, then either NEW-1 or NEW-R (as defined in Section 5.5) applies. If it is NEW-1 (note the check about the length of the tuples is true since it is also a premise of the rule NEW in Section 7.4) then the balance of the contract corresponding to $\mathrm{Top}(\sigma)$ is enough to create a new contract with an initial balance of n. Hence, $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta' \cdot [(c, a) \mapsto (C, \tilde{sv}, n)], \sigma, c \rangle$, where (c, a) is a fresh pair identifying the new contract in β and $\beta' = \mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n)$. On the other hand, if NEW-R applies, then the balance of the contract corresponding to $\mathrm{Top}(\sigma)$ is not enough. Hence, $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$. Lastly, if $\mathrm{Top}(\sigma) = \emptyset$, then NEW-2 (as defined in Section 5.5) applies. Hence, $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta \cdot [(c, a) \mapsto (C, \tilde{sv}, n)], \sigma, c \rangle$, where (c, a) is, as before, a fresh pair identifying the new contract in β.
  - if either $e_1 = \mathsf{revert}$ or $e_1 = v_1$ and $\tilde{e} = v_i^{\{0 \leq i < m\}}, \mathsf{revert}, e_j^{\{m < j \leq n\}}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(\mathsf{revert})(\tilde{e}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$ or $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(v_1)(\tilde{e} = v_i^{\{0 \leq i < m\}}, \mathsf{revert}, e_j^{\{m < j \leq n\}}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$, respectively.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(e_1)(\tilde{e}) \rangle \longrightarrow \langle \beta_1', \sigma_1', \mathsf{new}\ C.\mathsf{value}(e_1')(\tilde{e}) \rangle$.

  - if $e_1 = v_1$, $\tilde{e} = v_i^{\{0 \leq i < m\}}, e_j^{\{m \leq j \leq n\}}$, and $\langle \beta, \sigma, v_i^{\{0 \leq i < m\}}, e_j^{\{m \leq j \leq n\}} \rangle \longrightarrow \langle \beta_i', \sigma_i', v_i^{\{0 \leq i < m\}}, e_j'^{\{m \leq j \leq n\}} \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(v_1)(v_i^{\{0 \leq i < m\}}, e_j^{\{m \leq j \leq n\}}) \rangle \longrightarrow \langle \beta_i', \sigma_i', \mathsf{new}\ C.\mathsf{value}(v_1)(v_i^{\{0 \leq i < m\}}, e_j'^{\{m \leq j \leq n\}}) \rangle$.

- CONTRRETR. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, D(e) \rangle : D$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e \rangle : \mathsf{address}\langle \mathsf{C} \rangle$, where $C <: D$. By inductive hypothesis, e is either a value, revert or $\exists \beta', \sigma', e'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$. We distinguish the following cases:

  - if $e = v$ then, by Case 1 of Lemma 8, v is an address a. With the new rule just one scenario might happen: $\beta^C(a) = C <: D$ (well-defined by Lemma 13) and rule CONTRRETR of Section 7.3 applies. a corresponds to a reference c to a contract C. Hence, $\langle \beta, \sigma, D(a) \rangle \longrightarrow \langle \beta, \sigma, c \rangle$. On the other hand, the case with $\beta^C(a) = C < / : D$ is not possible. In fact, $\Gamma \vdash \langle \beta, \sigma, D(v) \rangle : T \Rightarrow \Gamma \vdash D(v) : T$ (by rule CONFIGURATION in Section 7.4). By Case 16 of Lemma 7, $C <: D$. This rules out rule CONTRRETR-R in Section 5.5, which does not apply anymore.
  - if $e = \mathsf{revert}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, D(\mathsf{revert}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, D(e) \rangle \longrightarrow \langle \beta', \sigma', D(e') \rangle$.

- TRANSFER. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1.\mathsf{transfer}(e_2) \rangle : \mathsf{unit}$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : \mathsf{address}\langle \mathsf{C} \rangle$ and $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : \mathsf{uint}$. By inductive hypothesis, e_1 is either a value, revert or $\exists \beta_1', \sigma_1', e_1'$ such that $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$. The same reasoning applies to e_2. We distinguish the following cases:

  - if $e_1 = v_1$ and $e_2 = v_2$ then, by Case 7 and Case 2 of Lemma 6, v_1 is an address a and v_2 is a non-negative integer n. Let $\beta(a) = (C, \tilde{sv}, m)$, well-defined by Lemma 13. Two are the possible scenarios. First, if $m \geq n$ rule TRANSFER in Section 5.5 applies and $\langle \beta, \sigma, a.\mathsf{transfer}(n) \rangle \longrightarrow \langle \mathrm{uptbal}(\mathrm{uptbal}(\beta, a, n), \mathrm{Top}(\sigma), -n), \sigma \cdot a, e' \rangle$, where $e' = e\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\}$, $\hat{\beta}(c) = a$, and e is either the body of the fallback function of C, if any, or return revert. Secondly, if $m < n$ then TRANSFER-R applies and we obtain $\langle \beta, \sigma, a.\mathsf{transfer}(n) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if either $e_1 = \mathsf{revert}$ or $e_1 = v_1$ and $e_2 = \mathsf{revert}$ then, by rule REVERT $\langle \beta, \sigma, \mathsf{revert}.\mathsf{transfer}(e_2) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$ or $\langle \beta, \sigma, v_1.\mathsf{transfer}(\mathsf{revert}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$, respectively.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, e_1.\mathsf{transfer}(e_2) \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1'.\mathsf{transfer}(e_2) \rangle$.
  - if $e_1 = v_1$ and $\langle \beta, \sigma, e_2 \rangle \longrightarrow \langle \beta_2', \sigma_2', e_2' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, v_1.\mathsf{transfer}(e_2) \rangle \longrightarrow \langle \beta_2', \sigma_2', v_1.\mathsf{transfer}(e_2') \rangle$.

  Note we did not make any reasoning about $\mathrm{Top}(\sigma)$. As said, we assumed $\mathrm{Top}(\sigma) \neq \emptyset$.

- CALL. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1.f.\mathsf{value}(e_2)(\tilde{e}) \rangle : T_2$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : C$, $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : \mathsf{uint}$, and $\Gamma \vdash_k \langle \beta, \sigma, \tilde{e} \rangle : \tilde{T_1}'$, where $\mathrm{ftype}(C, f) = \tilde{T_1} \to T_2$, $\tilde{T_1}' <: \tilde{T_1}$, and $|\tilde{e}| = |\tilde{T_1}|$. By inductive hypothesis, e_1 is either a value, revert or $\exists \beta_1', \sigma_1', e_1'$ such that $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$. The same reasoning applies to e_2 and to the tuple $\tilde{e}$. The former is either a value, revert or $\exists \beta_2', \sigma_2', e_2'$ such that $\langle \beta, \sigma, e_2 \rangle \longrightarrow \langle \beta_2', \sigma_2', e_2' \rangle$. The latter is either a tuple of values ($\tilde{v}$), a tuple of values and expressions separated by a revert (i.e. $v_i^{\{0 \le i < m\}}, \mathsf{revert}, e_j^{\{m < j \le n\}}$), or it is in the form $v_i^{\{0 \le i < m\}}, e_j^{\{m \le j \le n\}}$ and $\exists \beta_i', \sigma_i', e_j'$ such that $\langle \beta, \sigma, v_i^{\{0 \le i < m\}}, e_j^{\{m \le j \le n\}} \rangle \longrightarrow \langle \beta_i', \sigma_i', v_i^{\{0 \le i < m\}}, e_j'^{\{m \le j \le n\}} \rangle$. We distinguish the following cases:
  - $e_1 = v_1$, $e_2 = v_2$, and $\tilde{e} = \tilde{v}$. By Case 5 and Case 2 of Lemma 6, v_1 is a contract reference c and v_2 is a non-negative integer n. Let $\beta(c) = (C, s\tilde{:}v, m)$. Two are the possible scenarios. First, if $m \ge n$, then rule CALL in Section 5.5 applies. It retrieves f's body, e, and its formal parameters $\tilde{x}$, defines $\beta' = \mathrm{uptbal}(\mathrm{uptbal}(\beta, a, n), \mathrm{Top}(\sigma), -n) \cdot [x_i \mapsto v_i \; {}^{x_i \in \tilde{x}, v_i \in \tilde{v}}]$ as explained in Chapter 5, and then evolves as follows: $\langle \beta, \sigma, c.f.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta', \sigma \cdot a, e' \rangle$, where $e' = e\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\}$. Secondly, if $m < n$, then rule CALL-R in Section 5.5 applies, and $\langle \beta, \sigma, c.f.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$.
  - if either $e_1 = \mathsf{revert}$, $e_1 = v_1$ and $e_2 = \mathsf{revert}$, or $e_1 = v_1$, $e_2 = v_2$ and $\tilde{e} = v_i^{\{0 \le i < m\}}, \mathsf{revert}, e_j^{\{m < j \le n\}}$ then, by rule REVERT in Section 5.5, $\langle \beta, \sigma, \mathsf{revert}.f.\mathsf{value}(e_2)(\tilde{e}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$, $\langle \beta, \sigma, v_1.f.\mathsf{value}(\mathsf{revert})(\tilde{e}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$, or $\langle \beta, \sigma, v_1.f.\mathsf{value}(v_2)(\tilde{e} = v_i^{\{0 \le i < m\}}, \mathsf{revert}, e_j^{\{m < j \le n\}}) \rangle \longrightarrow \langle \beta, \sigma, \mathsf{revert} \rangle$, respectively.
  - if $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, e_1.f.\mathsf{value}(e_2)(\tilde{e}) \rangle \longrightarrow \langle \beta_1', \sigma_1', e_1'.f.\mathsf{value}(e_2)(\tilde{e}) \rangle$.
  - if $e_1 = v_1$ and $\langle \beta, \sigma, e_2 \rangle \longrightarrow \langle \beta_2', \sigma_2', e_2' \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, v_1.f.\mathsf{value}(e_2)(\tilde{e}) \rangle \longrightarrow \langle \beta_2', \sigma_2', v_1.f.\mathsf{value}(e_2')(\tilde{e}) \rangle$.
  - if $e_1 = v_1$, $e_2 = v_2$, $\tilde{e} = v_i^{\{0 \le i < m\}}, e_j^{\{m \le j \le n\}}$, and $\langle \beta, \sigma, v_i^{\{0 \le i < m\}}, e_j^{\{m \le j \le n\}} \rangle \longrightarrow \langle \beta_i', \sigma_i', v_i^{\{0 \le i < m\}}, e_j'^{\{m \le j \le n\}} \rangle$ then, by rule CONG in Section 5.5, $\langle \beta, \sigma, v_1.f.\mathsf{value}(v_2)(v_i^{\{0 \le i < m\}}, e_j^{\{m \le j \le n\}}) \rangle \longrightarrow \langle \beta_i', \sigma_i', v_1.f.\mathsf{value}(v_2)(v_i^{\{0 \le i < m\}}, e_j'^{\{m \le j \le n\}}) \rangle$.
- CALLTOPLEVEL. $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1.f.\mathsf{value}(e_2).\mathsf{sender}(e_3)(\tilde{e}) \rangle : T_2$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1.f.\mathsf{value}(e_2)(\tilde{e}) \rangle : T_2$ and $\Gamma \vdash_k \langle \beta, \sigma, e_3 \rangle : \mathsf{address}\langle \mathsf{C}' \rangle$, with additional hypothesis stating that C' must be a subtype of the minimum type required by f's signature. This case is very similar to the previous one, with the only addition of the parameter e_3 and considering CALLTOPLEVEL and CALLTOPLEVEL-R instead of CALL and CALL-R. What we said and proved previously applies here, too. We can apply the inductive hypothesis on e_3. If it is a value, then, by Case 1 of Lemma 8, it is an address a. If also e_1, e_2, and $\tilde{e}$ are values, either CALLTOPLEVEL or CALLTOPLEVEL-R (as defined in Section 5.5) applies, according to the balance of the caller. When it is a revert, rule REVERT in Section 5.5 applies. Lastly, when $e_1 = v_1$ and $e_2 = v_2$ and

$\exists \beta', \sigma', e_3'$ such that $\langle \beta, \sigma, e_3 \rangle \longrightarrow \langle \beta', \sigma', e_3' \rangle$ we obtain $\langle \beta, \sigma, v_1.f.\mathsf{value}(v_2).\mathsf{sender}(e_3)(\tilde{e}) \rangle \longrightarrow \langle \beta', \sigma', v_1.f.\mathsf{value}(v_2).\mathsf{sender}(e_3')(\tilde{e}) \rangle$.

- CALLVALUE. In this case $J = \Gamma \vdash_{k+1} \langle \beta, \sigma, e_1.\mathsf{value}(e_2)(\tilde{e}) \rangle : T_2$ was derived from $\Gamma \vdash_k \langle \beta, \sigma, e_1 \rangle : \tilde{T_1} \to T_2$, $\Gamma \vdash_k \langle \beta, \sigma, e_2 \rangle : \mathsf{uint}$, and $\Gamma \vdash_k \langle \beta, \sigma, \tilde{e} \rangle : \tilde{T_1}'$, where $\tilde{T_1}' <: \tilde{T_1}$. Also this case is similar to CALL, and the same rules (CALL and CALL-R of Section 5.5) are applied. The only difference is the expression e_1. By inductive hypothesis, it is either a value, revert or $\exists \beta', \sigma', e_1'$ such that $\langle \beta, \sigma, e_1 \rangle \longrightarrow \langle \beta', \sigma', e_1' \rangle$. If it is a value, by Case 6 of Lemma 6, it is a function pointer $c.f$. From this point on, every case is equal to those discussed for CALL, with the application of the same rules CALL and CALL-R.

□

F.5 Subject Reduction Theorem

We recall Theorem 4:

Theorem 4 (Subject Reduction)**.**
If $\Gamma \vdash \langle \beta, \sigma, e \rangle : T$ *with* $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta', \sigma', e' \rangle$ *then* $\exists \Delta$ *such that* $\Gamma' = \Gamma \cdot \Delta$ *and* $\Gamma' \vdash \langle \beta', \sigma', e' \rangle : T$.

Proof. We prove this theorem by induction on the height of the derivation of one reduction step $\langle \beta, \sigma, e \rangle \longrightarrow \langle \beta, \sigma, e' \rangle$.

Base cases These cases correspond to the axioms in Section 5.5, where the height of the derivation corresponding to the computational step is 0. We shall show only the cases where something changes with respect to the proof in Appendix E.5.

- IF-TRUE. In this case $\langle \beta, \sigma, \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \rangle \longrightarrow \langle \beta, \sigma, e_1 \rangle$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \rangle : T$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 : T$, and by Case 26 of Lemma 7 we know $\Gamma \vdash e_1 : T'$, where $T' <: T$, which makes it derivable $\Gamma' \vdash \langle \beta, \sigma, e_1 \rangle : T$, since β and σ are well-typed by hypothesis and $\Delta = \emptyset$.

- IF-FALSE. This case is dual to IF-TRUE.

- DECL. In this case $\langle \beta, \sigma, T_1\ x = v; e_1 \rangle \longrightarrow \langle \beta \cdot [x \mapsto v], \sigma, v; e_1 \rangle$, where $x \notin \mathrm{dom}(\beta)$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, T_1\ x = v; e_1 \rangle : T_2$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash T\ x = v; e_1 : T_2$, and by Case 18 of Lemma 7 we know $\Gamma \vdash e_1 : T_2$ and $\Gamma \vdash v : T_1'$, where $T_1' <: T_1$. Let $\Delta = x : T_1$ and $\Gamma' = \Gamma \cdot \Delta$, and consider $\Gamma' \vdash \beta \cdot [x \mapsto v]$. Rule VARIABLE requires $\Gamma' \vdash \beta$, which is derivable by applying Lemma 10 to the hypothesis $\Gamma \vdash \beta$, $x \notin \mathrm{dom}(\beta)$, which is true by hypothesis of DECL, $\Gamma' \vdash x : T_1$, and $\Gamma' \vdash v : T_1'$. The former is true by rule VAR in Section 6.3, since $\Delta = x : T_1$. The latter is true by Lemma 12 applied to $\Gamma \vdash v : T_1'$. Hence, by rule VARIABLE, $\Gamma' \vdash \beta \cdot [x \mapsto v]$ is true. Furthermore, by Lemma 11, given $\Gamma \vdash \sigma$ we know that also $\Gamma' \vdash \sigma$ can be derived. Hence, by rule CONFIGURATION, we obtain a derivation of $\Gamma' \vdash \langle \beta \cdot [x \mapsto v], \sigma, v; e_1 \rangle : T_2$, which is what we aimed to prove.

- VAR. In this case $\langle \beta, \sigma, x \rangle \longrightarrow \langle \beta, \sigma, \beta(x) \rangle$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, x \rangle : T$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash x : T$, and by Case 9 of Lemma 7 $x : T \in \Gamma$. By rule VARIABLE, each value v pointed to by a variable identifier x has a type $T' <: T$. This means that $\Gamma \vdash \beta(x) : T'$. Hence, $\Gamma' \vdash \langle \beta, \sigma, \beta(x) \rangle : T$, since β and σ are well-typed by hypothesis, setting $\Delta = \emptyset$.

- BALANCE. In this case $\langle \beta, \sigma, \mathsf{balance}(a) \rangle \longrightarrow \langle \beta, \sigma, n \rangle$, where $\beta(a) = (C, \tilde{s:v}, n)$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, \mathsf{balance}(a) \rangle : T$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash \mathsf{balance}(a) : T$, and by Case 10 of Lemma 7 we know $T = \mathsf{uint}$ and $\Gamma \vdash a : \mathsf{address}\langle \mathsf{C}' \rangle$, where $C' <: C$. From rule CONTRACT in Section 7.4 follows $\Gamma \vdash n : \mathsf{uint}$, which makes it derivable $\Gamma' \vdash \langle \beta, \sigma, n \rangle : \mathsf{uint}$, since β and σ are well-typed by hypothesis, where $\Delta = \emptyset$.

- ADDRESS. In this case $\langle \beta, \sigma, \mathsf{address}(c) \rangle \longrightarrow \langle \beta, \sigma, a \rangle$, where $\hat{\beta}(c) = a$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, \mathsf{address}(c) \rangle : T$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash \mathsf{address}(c) : T$, and by Case 11 of Lemma 7 we know $T = \mathsf{address}\langle \mathsf{C} \rangle$ and $\Gamma \vdash c : C$. We defined $\hat{\beta}(c) = a$ if $(c, a) \in \mathrm{dom}(\beta)$. $\Gamma \vdash \beta$ implies, by rule CONTRACT in Section 7.4, that every pair $(c, a) \in \mathrm{dom}(\beta)$ is well-formed, and hence, by Lemma 13, $\Gamma \vdash a : \mathsf{address}\langle \mathsf{C} \rangle$. From this follows $\Gamma' \vdash \langle \beta, \sigma, a \rangle : \mathsf{address}\langle \mathsf{C} \rangle$, since β and σ are well-typed by hypothesis, setting $\Delta = \emptyset$.

- ASS. In this case $\langle \beta, \sigma, x = v \rangle \longrightarrow \langle \beta[x \mapsto v], \sigma, v \rangle$, where $x \in \mathrm{dom}(\beta)$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, x = v \rangle : T'$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash x = v : T'$, and by Case 19 of Lemma 7, $\Gamma \vdash x : T$ and $\Gamma \vdash v : T'$, where $T' <: T$. β remains well-typed. In fact VARIABLE (Section 7.4) applies and makes it derivable $\Gamma' \vdash \langle \beta[x \mapsto v], \sigma, v \rangle : T$, where $\Delta = \emptyset$.

- STATEASS In this case $\langle \beta, \sigma, c.s = v' \rangle \longrightarrow \langle \beta[c.s \mapsto v'], \sigma, v' \rangle$, where $\beta(c) = (C, \tilde{s:v}, n)$ and $s \in \tilde{s}$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, c.s = v' \rangle : T'$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash c.s = v' : T'$, and by Case 20 of Lemma 7 we know $\Gamma \vdash c.s : T$ and $\Gamma \vdash v' : T'$, where $T' <: T$. Provided that the type of $c.s$ is T and that of v' is $T' <: T$, β remains well-typed. CONTRACT (Section 7.4) applies and makes it derivable $\Gamma' \vdash \langle \beta, \sigma, c.s = v' \rangle : T'$, where $\Delta = \emptyset$.

- MAPPSEL. In this case $\langle \beta, \sigma, M[v_1] \rangle \longrightarrow \langle \beta, \sigma, M(v_1) \rangle$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, M[v_1] \rangle : T$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash M[v_1] : T$, and by Case 21 of Lemma 7, $\exists T_1, T_2$ such that $\Gamma \vdash M : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $T = T_2$, and $\Gamma \vdash v_1 : T_1'$, where $T_1' <: T_1$. Since a mapping is a total function $T_1 \rightarrow T_2$, $M[v_1]$ is always well-defined and has type T_2, that is $\Gamma' \vdash \langle \beta, \sigma, M(v_1) \rangle : T_2$, where β and σ are well-typed by hypothesis, setting $\Delta = \emptyset$.

- MAPPASS. In this case $\langle \beta, \sigma, M[v_1 \rightarrow v_2] \rangle \longrightarrow \langle \beta, \sigma, M' \rangle$, where $M' = M \backslash \{(v_1, M(v_1))\} \cup \{(v_1, v_2)\}$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, M[v_1 \rightarrow v_2] \rangle : T$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash M[v_1 \rightarrow v_2] : T$, and by Case 22 of Lemma 7, $\exists T_1, T_2, T_1' <: T_1$, and $T_2' <: T_2$ such that $\Gamma \vdash M : \mathsf{mapping}(T_1 \Rightarrow T_2)$, $T = \mathsf{mapping}(T_1 \Rightarrow T_2)$, $\Gamma \vdash v_1 : T_1'$, and $\Gamma \vdash v_2 : T_2'$. Looking at M', we note it is obtained from M substituting the pair $(v_1, M(v_1))$ with (v_1, v_2). The types of both v_1 and v_2 are correct with respect

to $\mathsf{mapping}(T_1 \Rightarrow T_2)$, since $T_1' <: T_1$ and $T_2' <: T_2$, and so we can derive $\Gamma' \vdash \langle \beta, \sigma, M' \rangle : \mathsf{mapping}(T_1 \Rightarrow T_2)$, since β and σ are well-typed by hypothesis, setting $\Delta = \emptyset$.

- NEW-1. In this case $\langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) \cdot [(c, a) \mapsto (C, s\tilde{:}v, n)], \sigma, c \rangle$, where $(c, a) \notin \mathrm{dom}(\beta)$, $\mathrm{sv}(C) = \tilde{T}s$, and $|\tilde{v}| = |\tilde{s}|$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) \rangle : T$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash \mathsf{new}\ C.\mathsf{value}(n)(\tilde{v}) : T$, and by Case 15 of Lemma 7 we know $T = C$, $\Gamma \vdash n : \mathsf{uint}$, and $\Gamma \vdash \tilde{v} : \tilde{T}'$, where $\tilde{T}' <: \tilde{T}$. Let $\Delta = c : C, a : \mathsf{address}\langle \mathsf{C} \rangle$ and $\Gamma' = \Gamma \cdot \Delta$, and consider $\Gamma' \vdash \beta \cdot [(c, a) \mapsto (C, s\tilde{:}v, n)]$. Rule CONTRACT requires $\Gamma' \vdash \beta$, which is derivable by applying Lemma 10 to the hypothesis $\Gamma \vdash \beta$, $(c, a) \notin \mathrm{dom}(\beta)$, which is true by hypothesis of NEW-1, $\Gamma' \vdash c : C$, $\Gamma' \vdash a : \mathsf{address}\langle \mathsf{C} \rangle$, $\Gamma' \vdash \tilde{v} : \tilde{T}'$, and $\Gamma' \vdash n : \mathsf{uint}$. $\Gamma' \vdash c : C$ and $\Gamma' \vdash a : \mathsf{address}\langle \mathsf{C} \rangle$ are derivable by rules REF and ADDRESS, respectively. $\Gamma' \vdash \tilde{v} : \tilde{T}'$ and $\Gamma' \vdash n : \mathsf{uint}$ are, on the other hand, derivable by Lemma 12 applied to the hypotheses $\Gamma \vdash n : \mathsf{uint}$ and $\Gamma \vdash \tilde{v} : \tilde{T}'$. Furthermore, uptbal only modifies β incrementing or decrementing the balance n, thus preserving its well-formedness. We can here suppose that such operation is successful, since the case where uptbal returns $\bot$ is dealt with by NEW-R (see below). Hence, by rule CONTRACT in Section 7.4 it follows $\Gamma' \vdash \mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) \cdot [(c, a) \mapsto (C, s\tilde{:}v, n)]$. By Lemma 11, given $\Gamma \vdash \sigma$ also $\Gamma' \vdash \sigma$ is true. This makes it derivable, by rule CONFIGURATION, $\Gamma' \vdash \langle \mathrm{uptbal}(\beta, \mathrm{Top}(\sigma), -n) \cdot [(c, a) \mapsto (C, s\tilde{:}v, n)], \sigma, c \rangle : C$, which is what we wanted to prove.

- CALL. In this case $\langle \beta, \sigma, c.f.\mathsf{value}(n)(\tilde{v}) \rangle \longrightarrow \langle \beta', \sigma \cdot a, e\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\} \rangle$, where $\hat{\beta}(c) = a$, $\beta^C(c) = C$, $\mathrm{fbody}(C, f, \tilde{v}) = (\tilde{x}, e)$, $(x_i, a)^{x_i \in \tilde{x}} \notin \mathrm{dom}(\beta)$, and $\beta' = \mathrm{uptbal}(\mathrm{uptbal}(\beta, a, n), \mathrm{Top}(\sigma), -n) \cdot [x_i \mapsto v_i\ ^{x_i \in \tilde{x}, v_i \in \tilde{v}}]$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, c.f.\mathsf{value}(n)(\tilde{v}) \rangle : T$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash c.f.\mathsf{value}(n)(\tilde{v}) : T$, and by Case 23 of Lemma 7 $\exists \tilde{T}_1, T_2$ such that $T = \tilde{T}_1 \to T_2$, $\mathrm{ftype}(C, f) = \tilde{T}_1 \to T_2$, $\Gamma \vdash c : C$, $\Gamma \vdash n : \mathsf{uint}$, and $\Gamma \vdash \tilde{v} : \tilde{T}_1'$, where $\tilde{T}_1' <: \tilde{T}_1$. Since $\hat{\beta}(c) = a$ is well-defined by hypothesis, $\exists (c, a) \in \mathrm{dom}(\beta)$ such that $\Gamma \vdash a : \mathsf{address}\langle \mathsf{C} \rangle$: from it follows $\Gamma \vdash \sigma \cdot a$. Also note that $\Gamma \vdash \sigma \Rightarrow \Gamma \vdash \mathrm{Top}(\sigma) : \mathsf{address}\langle \mathsf{D}' \rangle$, where $\mathrm{Top}(\sigma) \neq \emptyset$ (otherwise CALLTOPLEVEL would apply). From rule F OK IN C (Section 7.4) we know $\mathsf{this} : C, \mathsf{msg.sender} : \mathsf{address}\langle \mathsf{D} \rangle, \mathsf{msg.value} : \mathsf{uint}, \tilde{x} : \tilde{T}_1 \vdash e : T_2$. By Case 23 of Lemma 7 we know $D' <: D$. By Lemma 5, considering the judgments $\Gamma \vdash c : C$, $\Gamma \vdash \mathrm{Top}(\sigma) : \mathsf{address}\langle \mathsf{D}' \rangle$, and $\Gamma \vdash n : \mathsf{uint}$ (which we have already proven as true), follows $\Gamma \vdash e\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\} : T_2$. The last thing we have to deal with is β'. First, as we already pointed out, uptbal only modifies the balances via arithmetical operations, thus preserving the well-formedness of β. Again, we can suppose that such operation is successful, since the case where uptbal returns $\bot$ is dealt with by CALL-R (see below). Secondly, we append to β a list of fresh pairs of local variables, together with the address of the contract identifying the invoked function. From $\Gamma \vdash \tilde{v} : \tilde{T}_1'$, where $\tilde{T}_1' <: \tilde{T}_1$ by Case 23 of Lemma 7, follows that each value v_i has a subtype of the formal parameter x_i it refers to, and we have already proven $\Gamma \vdash \mathrm{Top}(\sigma) : \mathsf{address}\langle \mathsf{D}' \rangle$. Hence, by rule VARIABLE in Section 7.4, $\Gamma \vdash \beta'$, and we can finally derive

$\Gamma' \vdash \langle \beta', \sigma \cdot a, e\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\}\rangle : T_2$, where $\Delta = \emptyset$.

- CALLTOPLEVEL. In this case $\langle \beta, \sigma, c.f.\mathsf{value}(n).\mathsf{sender}(a')(\tilde{v}); e'\rangle \longrightarrow \langle \beta', \sigma \cdot a, e\{\mathsf{this} := c, \mathsf{msg.sender} := a', \mathsf{msg.value} := n\}; e'\rangle$, where $\hat{\beta}(c) = a$, $\beta^C(c) = C$, $\mathrm{fbody}(C, f, \tilde{v}) = (\tilde{x}, e)$, $(x_i, a)^{x_i \in \tilde{x}} \notin \mathrm{dom}(\beta)$, and $\beta' = \mathrm{uptbal}(\mathrm{uptbal}(\beta, a, n), \mathrm{Top}(\sigma), -n) \cdot [x_i \mapsto v_i \ ^{x_i \in \tilde{x}, v_i \in \tilde{v}}]$. This case is very similar to CALL, the only difference relying on the sender, which is here explicitly set to a' (since $\mathrm{Top}(\sigma) = \emptyset$). We shall prove only the correctness of such sender, referring to the previous case for the rest of the proof. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, c.f.\mathsf{value}(n).\mathsf{sender}(a')(\tilde{v}); e'\rangle : T$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash c.f.\mathsf{value}(n).\mathsf{sender}(a')(\tilde{v}); e' : T$, and by Case 25 of Lemma 7, $\Gamma \vdash a' : \mathsf{address}\langle \mathsf{D}'\rangle$. By the same Case, f in C required a sender of type at least D ($\mathrm{fsender}(C, f) = D$), and $D' <: D$. Setting $\Delta = \emptyset$, we obtain $\Gamma' \vdash a' : \mathsf{address}\langle \mathsf{D}'\rangle$. The proof is now similar to the previous one.

- TRANSFER. This case is the same as CALL, the only difference relying on the absence of formal parameters. Furthermore, the return type is unit. Hence, β' is well-formed for what we said before, and so are $\sigma \cdot \mathrm{Top}(\sigma)$ and $e\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\}$. Also note that $\mathrm{Top}(\sigma) \neq \emptyset$, and hence using it to enlarge σ and to form β' is safe.

- CONTRRETR. In this case $\langle \beta, \sigma, D(a)\rangle \longrightarrow \langle \beta, \sigma, c\rangle$, where $\beta^C(a) = D'$, $D' <: D$, and $\hat{\beta}(a) = c$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, D(a)\rangle : T$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash D(a) : T$, and by Case 16 of Lemma 7 we know $T = D$ and $\Gamma \vdash a : \mathsf{address}\langle \mathsf{D}'\rangle$. Since $\hat{\beta}(a) = c$ is well-defined by hypothesis, $\exists (c, a) \in \mathrm{dom}(\beta)$, and since $\Gamma \vdash \beta$ we get $\Gamma \vdash c : D$. Hence, $\Gamma' \vdash \langle \beta, \sigma, c\rangle : D$, considering that σ is well-typed by hypothesis and $\Delta = \emptyset$.

Inductive case This case takes into account only rule CONG, the sole computational rule of our semantics. In this case $\langle \beta, \sigma, E[e]\rangle \longrightarrow^{k+1} \langle \beta', \sigma', E[e']\rangle$. The height of the derivation making this step possible is $k+1$, and its last judgment is $\langle \beta, \sigma, e\rangle \longrightarrow^{k} \langle \beta', \sigma', e'\rangle$. By hypothesis, $\Gamma \vdash \langle \beta, \sigma, E[e]\rangle : T$. To apply the inductive hypothesis we need to prove that, given a T', $\Gamma \vdash \langle \beta, \sigma, e\rangle : T'$. We do so using Lemma 14, whose hypothesis is satisfied by the hypotheses of this theorem. Hence, $\exists T'$ such that $\Gamma \vdash e : T'$ and $\forall e'$ such that $\Gamma \vdash e' : T'$ we have $\Gamma \vdash E[e'] : T$. The former is enough to prove $\Gamma \vdash \langle \beta, \sigma, e\rangle : T'$, since β and σ are well-typed by hypothesis (i.e. $\Gamma \vdash \beta$ and $\Gamma \vdash \sigma$). Bearing in mind that the premise of CONG states $\langle \beta, \sigma, e\rangle \longrightarrow^{k} \langle \beta', \sigma', e'\rangle$, we can apply the induction hypothesis to conclude $\Gamma \vdash \langle \beta', \sigma', e'\rangle : T'$. This implies, by rule CONFIGURATION in Section 7.4, $\Gamma \vdash e' : T'$. We now make use of the latter conclusion of Lemma 14: $\Gamma \vdash e' : T' \Rightarrow \Gamma \vdash E[e'] : T'$. Consequently, $\Gamma \vdash \langle \beta', \sigma', E[e']\rangle : T$, which is what we aimed to prove. □

F.6 Type Safety Theorems

Proven the validity of Progress (Theorem 3) and Subject Reduction (Theorem 4), the proof for the Theorems of Type Safety for configurations (Theorem 1) and programs (Theorem 2) remains the same as in Appendix E.6 and Appendix E.7, respectively.

F.7 Cast Safety Theorem

We recall Theorem 5:

Theorem 5 (Cast safety)**.**
If $\mathcal{P} = (CT, \beta, e)$ is closed and well-typed then $\mathcal{P}$ is cast-safe.

Proof. Let $\mathcal{P} = (CT, \beta, e)$ be a well-typed FS program. Consider the evaluation starting from the state $\langle \beta, \beta, e \rangle$, where β contains all the deployed contracts. (CT, β, e) closed means that $\langle \beta, \beta, e \rangle$ is closed, too (Lemma 1), and its being well-typed implies that $\exists \Gamma$ such that $\Gamma \vdash (CT, \beta, e) : T$. Consequently, by rule PROGRAM, $\Gamma \vdash \langle \beta, \beta, e \rangle : T$, and by Lemma 13, $\mathrm{dom}(\Gamma) \supseteq \mathrm{dom}(\beta)$. If the evaluation of e does not contain the rule CONTRRETR or CONTRRETR-R, the theorem is trivially true, since e does not contain any casts. Thus, let e contain at least a cast, such that $\langle \beta, \beta, e \rangle \longrightarrow^* \langle \beta', \sigma, D(a) \rangle$. We are to prove that the evaluation of $D(a)$ does not throw any reverts. By hypothesis, $\Gamma \vdash e : T$ is derivable. This means that also $D(a)$ is well-typed, i.e. the judgment $\Gamma \vdash D(a) : T'$ is derivable, too. By Case 16 of Lemma 7, $T' = D$ and $\exists D' <: D$ such that $\Gamma \vdash a : \mathsf{address}\langle \mathsf{D}' \rangle$ is derivable. Then, by Case 5 of Lemma 7, $a : \mathsf{address}\langle \mathsf{D}' \rangle \in \Gamma$ and hence a corresponds to an instance of $D' <: D$. $D(a)$ may evolve into two different terms, according to rules CONTRRETR and CONTRRETR-R in Section 7.3:

- If CONTRRETR applies then $c = \hat{\beta}'(a)$ must be an instance of a subcontract of $D'' <: D$, and we know that a refers exactly to an instance of $D' <: D$. Hence, setting $D'' = D'$, we can apply CONTRRETR and obtain $\langle \beta', \sigma, D(a) \rangle \longrightarrow \langle \beta', \sigma, c \rangle$.
- For CONTRRETR-R to apply it is necessary that $\beta'^C(a) = D'' \not<: D$, but this an absurd, since by hypothesis a points to an instance of $D' <: D$. Hence, this case does not apply and $\langle \beta', \sigma, D(a) \rangle \not\longrightarrow \langle \beta', \sigma, \mathsf{revert} \rangle$.

Note that we know that $\langle \beta', \sigma, D(a) \rangle$ cannot go stuck (i.e. neither CONTRRETR nor CONTRRETR-R applies), since by Theorem 1 any closed and well-typed configuration evolves toward either a value v or a revert. □

F.8 Transfer Safety Theorem

We recall Theorem 6:

Theorem 6 (Transfer safety)**.**
If $\mathcal{P} = (CT, \beta, e)$ is closed and well-typed then $\mathcal{P}$ is transfer-safe.

Proof. Let $\mathcal{P} = (CT, \beta, e)$ be a well-typed FS program. Consider the evaluation starting from the state $\langle \beta, \beta, e \rangle$, where β contains all the deployed contracts. (CT, β, e) closed means that $\langle \beta, \beta, e \rangle$ is closed, too (Lemma 1), and its being well-typed implies that $\exists \Gamma$ such that $\Gamma \vdash (CT, \beta, e) : T$. Consequently, by rule PROGRAM, $\Gamma \vdash \langle \beta, \beta, e \rangle : T$, and by Lemma 13, $\mathrm{dom}(\Gamma) \supseteq \mathrm{dom}(\beta)$. If e does not contain any $\mathrm{transfer}$ the theorem is trivially true. Thus, let e contain at least a $\mathrm{transfer}$, such that $\langle \beta, \beta, e \rangle \longrightarrow^* \langle \beta', \sigma, a.\mathsf{transfer}(n) \rangle$. We are to prove that the evaluation of $a.\mathsf{transfer}(n)$ does not throw any reverts. By hypothesis, $\Gamma \vdash e : T$ is derivable. This means that also $a.\mathsf{transfer}(n)$ is well-typed, i.e. $\Gamma \vdash a.\mathsf{transfer}(n) : T'$ is derivable, too. By Case 14 of Lemma 7 $T' = \mathsf{unit}$, $\Gamma \vdash e_1 : \mathsf{address}\langle \mathsf{C} \rangle$, $\Gamma \vdash e_2 : \mathsf{uint}$,

and $\Gamma \vdash \mathsf{this} : C''$. In addition, $\mathrm{ftype}(C, fb) = \{\} \to \mathsf{unit}$ means that fb is defined in C. Lastly, its signature requires C'' to be a subcontract of $C' = \mathrm{fsender}(C, fb)$ ($C'' <: C'$). $a.\mathsf{transfer}(n)$ may evolve into two different expressions, according to the result of $\mathrm{fbody}(C, fb, \{\})$ in rule TRANSFER in Section 5.5:

- If $\mathrm{fbody}(C, fb, \{\}) = (\{\}, \mathsf{return}\ e')$ and $\mathrm{uptbal}(\mathrm{uptbal}(\beta, a, n), \mathrm{Top}(\sigma), -n) = \beta'' \neq \bot$, then $\langle \beta', \sigma, a.\mathsf{transfer}(n) \rangle \longrightarrow \langle \beta'', \sigma \cdot a, e'\{\mathsf{this} := c, \mathsf{msg.sender} := \mathrm{Top}(\sigma), \mathsf{msg.value} := n\} \rangle$. The fallback is defined and balances are sufficient to accomplish the money transfer, thus the evaluation proceeds. Note that it is fine if e' contains a revert, since Definition 18 allows those raised by programmers.

- If, on the other hand, $\mathrm{fbody}(C, fb, \{\}) = (\{\}, \mathsf{return}\ e')$, but $\mathrm{uptbal}(\beta', \mathrm{Top}(\sigma), -n) = \bot$, $\langle \beta', \sigma, a.\mathsf{transfer}(n) \rangle \longrightarrow \langle \beta', \sigma, \mathsf{revert} \rangle$. Note that this case is fine, too, since Definition 18 accepts reverts thrown as a consequence of an insufficient balance.

- The case with $\mathrm{fbody}(C, fb, \{\}) = (\{\}, \mathsf{return\ revert})$ does not apply. In fact, we know that $\mathrm{ftype}(C, fb) = \{\} \to \mathsf{unit}$, which means that either C or any of its supercontracts contains a fallback function. Hence, for $\mathrm{fbody}(C, fb, \{\})$ to return $(\{\}, \mathsf{return\ revert})$, it is necessary that the lookup goes up the hierarchy until Top, but this cannot happen if any of the contracts in the hierarchy contains a valid fb. Hence, $\mathrm{fbody}(C, fb, \{\})$ cannot return $(\{\}, \mathsf{return\ revert})$.

Note that we know that $\langle \beta', \sigma, a.\mathsf{transfer}(n) \rangle$ cannot go stuck (i.e. TRANSFER does not apply), since by Theorem 1 any closed and well-typed configuration evolves toward either a value v or a revert. □

Bibliography

[1] Alexander Joseph Ahern. "Code mobility and Java RMI". PhD thesis. Citeseer, 2007.

[2] Elvira Albert et al. "EthIR: A Framework for High-Level Analysis of Ethereum Bytecode". In: *arXiv preprint arXiv:1805.07208* (2018).

[3] Jonathan Aldrich et al. "Typestate-oriented programming". In: *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM. 2009, pp. 1015–1022.

[4] Sidney Amani et al. "Towards verifying ethereum smart contract bytecode in Isabelle/HOL". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM. 2018, pp. 66–77.

[5] Suveen Angraal, Harlan M Krumholz, and Wade L Schulz. "Blockchain technology: applications in health care". In: *Circulation: Cardiovascular Quality and Outcomes* 10.9 (2017), e003800.

[6] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A survey of attacks on Ethereum smart contracts (SoK)". In: *International Conference on Principles of Security and Trust*. Springer. 2017, pp. 164–186.

[7] Mark Batty et al. "Mathematizing C++ concurrency". In: *ACM SIGPLAN Notices* 46.1 (2011), pp. 55–66.

[8] Karthikeyan Bhargavan et al. "Formal verification of smart contracts: Short paper". In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM. 2016, pp. 91–96.

[9] V. Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. Tech. rep. URL: `https://github.com/ethereum/wiki/wiki/White-Paper`.

[10] V. Buterin. *Patricia Tree*. URL: `https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-Patricia-Tree`.

[11] Vitalik Buterin and Virgil Griffith. "Casper the friendly finality gadget". In: *arXiv preprint arXiv:1710.09437* (2017).

[12] Ting Chen et al. "Towards saving money in using smart contracts". In: *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. ACM. 2018, pp. 81–84.

[13] Ting Chen et al. "Under-optimized smart contracts devour your money". In: *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE. 2017, pp. 442–446.

[14] Michael Coblenz. "Obsidian: a safer blockchain programming language". In: *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press. 2017, pp. 97–99.

[15] Michael Coblenz. "User-Centered Design of Permissions, Typestate, and Ownership in the Obsidian Blockchain Language". In: (2018).

[16] Alan Cohn, Travis West, and Chelsea Parker. "Smart After All: Blockchain, Smart Contracts, Parametric Insurance, and Smart Energy Grids". In: *Georgetown Law Technology Review* 1.2 (2017), pp. 273–304.

[17] Kevin Delmolino et al. "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2016, pp. 79–94.

[18] *Etherdice is down for maintenance*. URL: https://www.reddit.com/r/ethereum/comments/47f028/etherdice_is_down_for_maintenance_we_are_having/.

[19] David Evans and David Larochelle. "Improving security using extensible lightweight static analysis". In: *IEEE software* 1 (2002), pp. 42–51.

[20] *GovernMental's 1100 ETH jackpot payout is stuck because it uses too much gas*. URL: https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/.

[21] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. "A Semantic Framework for the Security Analysis of Ethereum smart contracts". In: *International Conference on Principles of Security and Trust*. Springer. 2018, pp. 243–269.

[22] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. "EtherTrust: Sound Static Analysis of Ethereum bytecode". In: ().

[23] Everett Hildenbrandt et al. *Kevm: A complete semantics of the ethereum virtual machine*. Tech. rep. 2017.

[24] Yoichi Hirai. "Defining the ethereum virtual machine for interactive theorem provers". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2017, pp. 520–535.

[25] Atsushi Igarashi, Benjamin C Pierce, and Philip 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.

[26] Bo Jiang, Ye Liu, and WK Chan. "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection". In: *arXiv preprint arXiv:1807.03932* (2018).

[27] Sukrit Kalra et al. "Zeus: Analyzing safety of smart contracts". In: NDSS. 2018.

[28] Theodoros Kasampalis et al. *IELE: An Intermediate-Level Blockchain Language Designed and Implemented Using Formal Semantics*. Tech. rep. 2018.

[29] Ton Chanh Le et al. "Proving Conditional Termination for Smart Contracts". In: *Proceedings of the 2nd ACM Workshop on Blockchains, Cryptocurrencies, and Contracts*. ACM. 2018, pp. 57–59.

[30] Chao Liu et al. "ReGuard: finding reentrancy bugs in smart contracts". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM. 2018, pp. 65–68.

[31] Loi Luu et al. "Making smart contracts smarter". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 254–269.

[32] Robert C Martin. “Java and C++ A critical comparison”. In: *Technical Note, Object Mentor* (1997).

[33] Anastasia Mavridou and Aron Laszka. “Designing secure Ethereum smart contracts: A finite state machine based approach”. In: *arXiv preprint arXiv:1711.09327* (2017).

[34] Satoshi Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: (2008).

[35] Ivica Nikolic et al. “Finding the greedy, prodigal, and suicidal contracts at scale”. In: *arXiv preprint arXiv:1802.06038* (2018).

[36] Reza M Parizi, Ali Dehghantanha, et al. “Smart Contract Programming Languages on Blockchains: An Empirical Evaluation of Usability and Security”. In: *International Conference on Blockchain*. Springer. 2018, pp. 75–91.

[37] Jack Pettersson and Robert Edström. “Safer smart contracts through type-driven development”. PhD thesis. Master’s thesis, Chalmers University of Technology, Department of Computer Science and Engineering, Sweden, 2016.

[38] Alessandra Pieroni et al. “Smarter City: Smart Energy Grid based on Blockchain Technology”. In: *International Journal on Advanced Science, Engineering and Information Technology* 8.1 (2018), pp. 298–306.

[39] Giulio Prisco. *The DAO Raises More Than $117 Million in World’s Largest Crowdfunding to Date*. https://bitcoinmagazine.com/articles/the-dao-raises-more-than-million-in-world-s-largest-crowdfunding-to-date-1463422191/. Accessed June, 2018.

[40] Obsidian Project. *Obsidian Platform White Paper*. Tech. rep. URL: https://drive.google.com/file/d/0B5ffnLUWqKkMMG1YTXUtNjNvOGc/view.

[41] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. “Formal verification of object layout for C++ multiple inheritance”. In: *ACM SIGPLAN Notices*. Vol. 46. 1. ACM. 2011, pp. 67–80.

[42] Grigore Rosu. *An overview of the K semantic framework*. Tech. rep. 2010.

[43] Grigore Rosu. “Formal Design, Implementation and Verification of Blockchain Languages (Invited Talk)”. In: *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 108. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[44] Ilya Sergey and Aquinas Hobor. “A concurrent perspective on smart contracts”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2017, pp. 478–493.

[45] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. “Scilla: a smart contract intermediate-level language”. In: *arXiv preprint arXiv:1801.00687* (2018).

[46] Pradip Kumar Sharma, Seo Yeon Moon, and Jong Hyuk Park. “Block-VN: A distributed blockchain based vehicular network architecture in smart City”. In: *Journal of Information Processing Systems* 13.1 (2017), p. 84.

[47] David Siegel. *Understanding The DAO Attack*. https://www.coindesk.com/understanding-dao-hack-journalists/. Accessed June, 2018.

[48] *Solidity documentation*. https://solidity.readthedocs.io/en/develop/index.html. Release 0.4.24.

[49] Sergei Tikhomirov et al. "SmartCheck: Static Analysis of Ethereum Smart Contracts". In: (2018).

[50] *Truffle suite*. URL: https://truffleframework.com/.

[51] Petar Tsankov et al. "Securify: Practical Security Analysis of Smart Contracts". In: *arXiv preprint arXiv:1806.01143* (2018).

[52] Maximilian Wöhrer and Uwe Zdun. "Design Patterns for Smart Contracts in the Ethereum Ecosystem". In: (2018).

[53] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151 (2014), pp. 1–32.

[54] Yong Yuan and Fei-Yue Wang. "Towards blockchain-based intelligent transportation systems". In: *Intelligent Transportation Systems (ITSC), 2016 IEEE 19th International Conference on*. IEEE. 2016, pp. 2663–2668.