Linear typing for resource-aware programming

Student: Giacomo Dal Sasso

Università degli Studi di Padova

DIPARTIMENTO MATEMATICA
DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

Department of Mathematics
Master degree in COMPUTER SCIENCE

Supervisor: Prof. Silvia Crafa

Academic year 2023-24
Graduation date 23/02/2024

Contents

Preface

Abstract

Smart contracts running on blockchain platforms are critical components of decentralized applications (dApps). Often, smart contracts manipulate valuable assets, so that the correctness and security of the code is a main concern. A programming error in a smart contract can lead to huge capital losses.

The recently developed Move language seems to be a step forward in term of smart contracts security and correctness. Move introduces the use of linear types to represent resources (like financial assets) in the language, in a way that permits to detect at compile time common errors on the manipulation of resources. In Move, the programmer can't successfully compile a program that loses a coin or forges a copy of a coin by mistake.

We provide a bird's eye comparison between Move, applied to the Sui blockchain, and Solidity, used in Ethereum, with the aim of better understanding the errors preventable by Move's linear types. Doing so, we give a clean explanation of the move semantics in Move. We formalize the operational semantics and the type system of a core subset of the Move language we call FM, which includes linear types, and we prove the language enjoys a Resource Preservation property similar to the the Resource Safety property stated and proved by Blackshear at al. for the Move bytecode. We mechanize the proofs we have done for FM with approximately 3000 lines of code in Agda, a proof assistant based on Martin-Löf intuitionistic type theory. In the formalization we clarify what does it mean to use a resource and how the type system constraining the use of variables by programmers, with linear typing, can guarantee resources are correctly used at runtime.

Document Structure

The document is organized as follows:

- In Chapter 1 we give a brief introduction to the blockchain technology and to smart contracts programming. We overview the Ethereum and the Sui blockchain platforms, and their main on-chain programming languages that are respectively Solidity and Move, from the perspective of a smart contract developer. We discuss the peculiar characteristics of Move language (particularly linear types) and the errors it can prevent that Solidity cannot.
- In Chapter 2 we formalize the core language FM giving its syntax, operational semantics and typing rules. At the beginning of the chapter we define a simple blockchain model in which we immagine FM to be executed, to give a concrete application context to the language.
- In Chapter 3 we state and discuss the properties of FM we have proved. The chapter is divided in two main sections: Basic properties and Resource Safety. The goal of the former is to prove a closed and well-typed term can't evolve into a stuck term at runtime (Theorem 1). The goal of the latter is to prove the Resource Preservation theorem (Theorem 2).
- In Chapter 4 we present the mechanization of FM in the Agda proof assistant.
- In Chapter 5 we draw some conclusions and discuss possible future works.

Acknowledgments

I thank my former employers, and now friends, Luciano and Andrea, for allowing me to continue my studies with maximum flexibility and for always supporting me. Without their trust, I could not have reached this milestone.

Special thanks to Prof. Alvise Spanò, who passionately introduced me to the world of formal methods with his "Functional Languages" course and passed on his love for the subject.

Thanks to Prof. Maria Emilia Maietti and Prof. Ingo Blechschmidt for introducing me to type theory and Agda, two topics that excite me and have greatly influenced this thesis and my thinking more generally.

I thank my supervisor, Prof. Silvia Crafa, for her expertise and precision. I thank you for always making me feel comfortable and allowing me to express my ideas freely. Thank you for guiding me through this research journey with great professionalism.

I would like to thank the research team composed of professors Sabina Rossi, Alvise Spanò, Silvia Crafa, Michele Bugliesi and doctoral student Lorenzo Benetollo, who welcomed me and helped me analyze the contents of this thesis.

Finally, I sincerely thank my parents, Anna and Paolo, my sisters Jessica, Cecilia, Benedetta and Speranza, and my whole family.

Ringraziamenti

Ringrazio i miei ex titolari, e ormai amici, Luciano ed Andrea, per avermi permesso di proseguire gli studi con la massima flessibilità e per avermi sempre supportato. Senza la loro fiducia, non avrei potuto raggiungere questo traguardo.

Un ringraziamento particolare al prof. Alvise Spanò che mi ha introdotto con passione al mondo dei metodi formali, con il suo corso "Functional Languages", e che mi ha trasmesso l'amore per la materia.

Grazie alla prof.ssa Maria Emilia Maietti ed al prof. Ingo Blechschmidt per avermi fatto scoprire la teoria dei tipi ed Agda, due argomenti che mi entusiasmano e che hanno influenzato molto questa tesi e più in generale il mio modo di pensare.

Ringrazio il mio relatore, la prof.ssa Silvia Crafa, per la sua generosa disponibilità, per la sua competenza e per la sua meticolosità. La ringrazio per avermi fatto sentire sempre a mio agio e per avermi permesso di esprimere le mie idee liberamente. Grazie per avermi guidato in questo percorso di ricerca con grande professionalità.

Ringrazio il gruppo di ricerca composto dai professori Sabina Rossi, Alvise Spanò, Silvia Crafa, Michele Bugliesi e dal dottorando Lorenzo Benetollo, che mi ha accolto e mi ha aiutato ad analizzare i contenuti di questa tesi.

Infine, ringrazio di cuore i miei genitori, Anna e Paolo, le mie sorelle Jessica, Cecilia, Benedetta e Speranza, e tutta la mia famiglia.

Chapter 1

Smart Contracts Programming

1.1 Introduction

In recent years, blockchains and smart contracts, are receiving a lot of attention. New platforms and programming languages continue to emerge proposing new solutions to the problems plaguing the blockchain environment. One of the main problems is the security and correctness of smart contracts developed using blockchain specific programming languages (on-chain languages), that constitute the backend of decentralized applications (dApps). Often, smart contracts manipulate valuable assets, so that a vulnerability in the code can lead to huge capital losses. Examples are the DAO attack in 2016, where an amount of ethereum worth 50 million USD was stolen exploiting a reentrancy vulnerability, the Parity wallet bug in 2017, where 31 million USD were loss, and the MyEtherWallet hack in 2018, where 17 million USD were stolen [10, p. 2].

The Move language, burn with the Diem blockchain and later inherited by the Aptos and Sui blockchains, seems to be a step forward in term of smart contracts security and correctness. As main novelty, Move introduces the use of linear types to represent resources (like financial assets) in the language, in a way that permits to detect at compile time common errors on the manipulation of resources. Assets like cryptocurrencies and NFTs are represented in Move as C-like structs, and the type system guaranties that instances of these structs are not duplicated or lost during the execution of the smart contract. In Move, the programmer can't successfully compile a program that loses a coin or creates a copy of a coin by mistake. This is a powerful feature for a blockchain programming language.

We provide a bird's eye comparison between Move, applied to the Sui blockchain, and Solidity, the most popular blockchain programming language, used in the Ethereum blockchain. In particular, we try to understand how the use of linear types in Move can improve the correctness of smart contracts, with respect to Solidity. Doing so, we provide a new viewpoint on the move semantics of Move, clarifying the mechanism underlying the linear types in the language.

We formalize the operational semantics and the type system of a subset of the Move language we call FM, which includes linear types. For the core language FM, in addition to proving the standard properties expected from a well-defined operational semantics and type system, inspired by the Resource Safety theorem

in the formalization of the Move bytecode [7], we prove an equivalent Resource Preservation theorem. The Resource Preservation theorem states that: in a program that compiles without errors, resources (assets in particular) can't be duplicated or accidentally lost at runtime. The formalization gives a better understanding of what does it mean to use a variable and a value (in particular a resource) stating precisely what are the operations that use (or consume) a value. We have found this concept not very clear in the literature, and our work attempts to shed some light on it. In addition, the core language FM takes from Move the linear typing mechanism only, separating it from the ownership/access-control mechanism which is left for future work. This helps to clarify the role of linear types in Move, that is what are the kinds of error prevented in Move that we can attribute to linear typing, and to understand the potential of linear types in the context of smart contracts programming.

We mechanize the proofs we have done for FM in approximately 3000 lines of Agda code. Agda is a proof assistant (like Coq) based on Martin-Löf intuitionistic type theory [18]. In particular, two important theorems we have proved using Agda are the Type Preservation lemma (Lemma 6) and the Resource Preservation lemma (Lemma 10). The code in Agda is not only a formal proof of the correctness of the type system and the operational semantics of FM, but is also a framework for future extensions of the language and for the investigation of new language properties.

1.2 Introduction to blockchains

Nowadays the term *blockchain* is overloaded of meanings and it is used in different contexts to denote different things. It is therefore necessary to make some clarifications that may seem trivial.

Blockchain - A blockchain is a particular kind of distributed data structure for storing information [13]. A blockchain is composed of a sequence of blocks, each containing a set of data and a reference to the previous block. The blockchain is append-only, that is, it grows from a root block, called "genesis block", and new blocks can only be added at the end of the chain. Cryptography and a consensus protocol are used to establish the order between blocks. The information stored in the blockchain is composed by the data contained in all its blocks. A blockchain can store any kind of information. The cryptographic mechanisms used in a blockchain are commonly used in other contexts, such as in PKI (Public Key Infrastructure) and in Secure Boot of embedded systems.

Blockchain system - A modern blockchain system, such as Ethereum, Sui, Aptos and Solana, is a platform which uses a blockchain to store its state [19]. A blockchain system can be thought of as one massive global computer where anyone can store data and execute code paying a fee. The global computer is distributed across a network of nodes, with no central authority, which cooperate to maintain the system. In a similar way to what happens in a standard PC, the blockchain system can execute programs on demand which update its state. Users of the system request the execution of programs by sending transactions to the network. A transaction is a message signed by the sender, which contains instructions for the system. The

blockchain data structure of a blockchain system is also called ledger.

In what follows we will use the term blockchain to refer both to the data structure and to the system. It will be clear from the context which of the two meanings we are referring to.

The state of a blockchain is synchronized across the nodes of the network and each state update must be agreed upon by the nodes. A consensus algorithm is used to reach consensus on state updates. As a first approximation, when a transaction is forwarded to the network, each node executes it and compares the resulting state with the state produced by other nodes. If the majority of the nodes agree on the next state in which the system should evolve the state update is applied.

Smart contract - As in the filesystem of a standard PC, the ledger may contain both data and programs. Programs stored in the ledger and invoked by transactions are called *smart contracts*. Usually, blockchains are born with a predefined set of smart contracts providing basic functionalities, while new and original smart contracts are developed by users who deploy them sending special transactions to the network. We say a smart contract is deployed on-chain.

Smart contracts are written in a blockchain specific programming language. Some blockchains support multiple languages, such as Solana which supports both Rust and C/C++, while others, sometimes due to their peculiar characteristics, only one, such as Sui and Aptos which support only Move. Some programming languages for smart contracts are general purpose, such as Rust, while others are devised specifically for the blockchain environment, such as Move and Solidity.

Languages used to write smart contracts are also called on-chain languages, to distinguish them from off-chain languages, which are used to write client side applications that interact with the blockchain reading the ledger and sending transactions. Off-chain languages are general purpose, like JavaScript, Python, Dart and run on a client side untrusted environment, while on-chain languages run on the blockchain trusted environment (the virtual machine).

Virtual machine - The blockchain executes the bytecode of a program in a virtual machine. Each blockchain may define its own virtual machine with a custom instruction set. Compared to the instruction set of a standard CPU, the instruction set of a blockchain is usually simpler and more limited. Generally, blockchain programs don't need great performance because they don't implement complex logic and algorithms [15]. Instead, they must be secure and reliable because they manipulate valuable assets. A simple and well formalized ISA (Instruction Set Architecture) is easier to implement and to verify, and it simplifies the writing of compilers for languages that target the blockchain. Many security bugs have been found within the Solidity compiler[1], and new bugs continue to be found in the Vyper compiler[2], which is a python-like smart contract language for Ethereum.

dApp - On-chain and off-chain languages are used together to write decentralized applications, or dApps. A dApp is not much different from a traditional webApp. In both cases we have a client side application, which runs on the user's device, and

[1] `https://docs.soliditylang.org/en/develop/bugs.html`

[2] `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-22419`

a server side application, which runs in a centralized server in the case of a webApp while it runs on a network of nodes in the case of a dApp. In a dApp, the backend is written with an on-chain language while the frontend with an off-chain language.

Transaction fees - Generally, users must pay a fee, in the blockchain native currency, to execute a transaction and the fee is proportional to the amount of code executed by the transaction. The fee may also depend on other factors, such as on the congestion state of the network: the more the network is congested the higher the fee. Due to this costs, and due to the latency introduced by the consensus algorithm, decentralized applications are difficult to scale. The more the users of a dApp, the more the transactions that must be executed, the higher the network congestion, the latency and the fees. Research is ongoing to mitigate this problem [26] [16] [11].

Smart contract Use cases - Blockchains are predominantly used to store, transfer and manipulate digital assets which are digital equivalents of physical (often valuable) objects. In a world where finance is mainly managed by banking institutions that can't keep up with the technological progress, blockchains would like to be a decentralized, secure and democratic alternative for the management of financial assets [19]. The most common example of digital asset is the cryptocurrency. Digital assets can be divided in fungible and non-fungible (NFT). Fungible assets are interchangeable, like a penny that is indistinguishable from another penny, while non-fungible assets are unique, like a painting. The ledger of a blockchain is mainly used to keep track of the assets owned by each user.

Smart contracts are mainly used to implement applications in which an agreement between untrusted parties is needed [17]. As an example, an online marketplace (such as Amazon) is an intermediary the buyer and the seller rely on to conclude a sale. The buyer trusts the marketplace (but not the seller) to receive the goods he has paid for, and the seller trusts the marketplace (but not the buyer) to receive the payment. The buyer is sure that his order and his money will not be lost. The seller is sure that he will get his money when the customer's receipt of the product has been confirmed. A smart contract establishes a trusted, reliable and regulated environment in which untrusted users can interact with each other. The blockchain ledger is a root of trust.

To name a few, a smart contract can implement a marketplace, a crowdfunding campaign, an auction, a lottery or a monthly payment. The paper [3] written by Bartoletti at al., to which we made a small contribution, shows the implementation of some of those use cases in different blockchains and programming languages.

Motivation - The economic value of assets manipulated during the execution of transactions can be high, thus the correctness of the code executed is crucial. A bug in a smart contract can lead to the loss of millions of dollars. Furthermore, since the bytecode of each smart contract (sometimes even the source code) is freely accessible by anyone due to the public nature of the ledger, and since the blockchain programming environment can still be considered relatively new, it is not difficult to find bugs [29]. One of the main problems that developers in this sector complain about is the lack of adequate tools and programming languages that can help them

prevent potentially disastrous errors [31].

Motivated by these observations we decide to explore the world of blockchain programming languages. The initial objective was that of understanding what are the fundamental characteristics a language aimed at the blockchain environment should have. A good language should help the developer write correct programs by preventing the most serious and common errors. Since smart contracts are rarely complex [15], it would be reasonable to prefer a simple language, with clear and predictable semantics and with good security properties. After this analysis we decided to focus on the Move language to deeply understand how it uses linear types to prevent certain kinds of errors in the manipulation of resources (in particular assets). For this purpose, we have formalized the FM core language.

Our focus remains on the language and errors that can be identified at compile-time, with an appropriate type system. We will not deal with and consider other issues typical of blockchains, such as the efficiency of consensus algorithms, scalability, privacy, etc.

This chapter proceeds by providing an overview of two different blockchains: Ethereum which today is probably the biggest and the most used, and Sui which is quite novel in the scene and introduces interesting features at the language level.

1.3 Ethereum

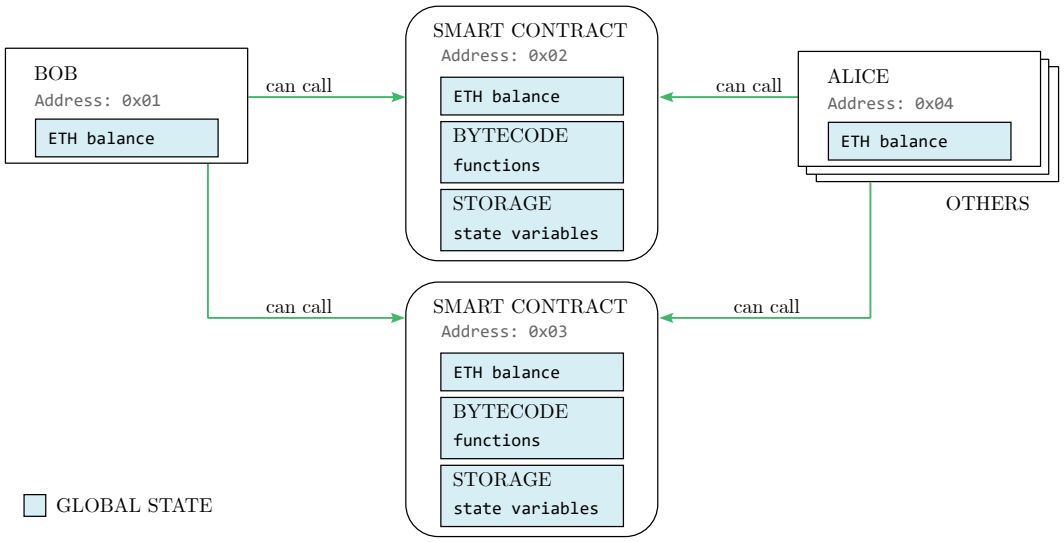

Figure 1.1: A simplified model of Ethereum. Bob and Alice are EOAs (Externally Owned Accounts) representing two users of the system. Each of them has its own ETH balance. Two different smart contracts are deployed on chain, each with its own balance, bytecode and storage. Any EOA can invoke functions of any contract. The balance of each address, together with the bytecode and the storage of each contract, compose the global state of Ethereum .

Apart from Bitcoin, that has limited capabilities (in terms of programmability) compared to those of modern blockchains, Ethereum is probably the best known and most used. It was launched in 2015 by Vitalik Buterin and partners, its native

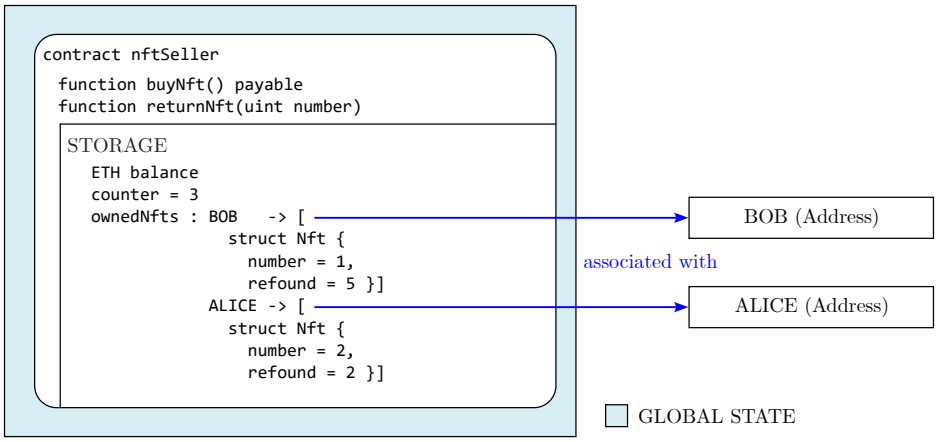

Figure 1.2: A possible state of an instance of the `nftSeller` contract (Listing 1) after some transactions. Bob bought the NFT with the number 1, paying 5 Wei (1 Wei = 10^{18} ETH) for it, while Alice bought the NFTs with number 2 paying 2 Wei. The counter of the contract is 3.

currency is the Ether (ETH), at the time this is written it counts 6795 nodes and it has a market cup of 303 billion USD [3]. The number of nodes and the market cup fluctuate heavily over time but their order of magnitude can be useful to understand the size and importance of Ethereum.

In Figure 1.1 we show a simplified model of Ethereum. Accounts in Ethereum are entities that can interact with the system. There are two types of accounts: Externally Owned Accounts (EOA), which are controlled by users, and Contract Accounts, which are controlled by smart contracts. Each account has a unique address, which is a 160-bit identifier, and a balance in ETH. EOAs send transactions to the network to transfer ETH and to invoke functions exposed by smart contracts.

Smart contracts - A smart contract has a balance in ETH, a bytecode fragment, which is the code executed by the Ethereum Virtual Machine (EVM) when the contract is invoked, and a persistent storage. The storage is a private memory area the contract can use to store data that persists between different invocations of the contract. The storage may save the list of users registered to a service, the list of items in a marketplace, or the state of a game. The balance, the bytecode and the storage of a contract are written in the ledger and contribute to the definition of the global state of the system. The bytecode of a contract is immutable while the balance and the private memory can change during the contract lifetime.

Solidity - The majority of Ethereum smart contracts are written in Solidity, a feature reach object oriented language with a C++ like syntax. In Listing 1 we show a toy example of a Solidity contract. Pretending that natural numbers are assets worth owning, the contract let users buy unique natural numbers as NFTs. A user can brag about owning the number 42. The minimum price of a number is proportional to its value, so the base price of the number 42 is twice that of the number 21. To buy a number, a user can pay the minimum price or more, as they

[3] `https://www.kraken.com/prices/ethereum`

```
pragma solidity ^0.8.13;
contract nftSeller {
  struct Nft {
    uint number;
    uint refound;
  }

  uint counter;
  mapping(address => Nft[]) public ownedNfts;

  constructor() { counter = 1; }

  function buyNft() public payable {
    require(msg.value >= counter, "Not enough Ether");
    Nft memory nft = Nft(counter, msg.value);
    ownedNfts[msg.sender].push(nft);
    counter++;
  }

  function returnNft(uint number) public {
    Nft[] storage Nfts = ownedNfts[msg.sender];
    for (uint i = 0; i < Nfts.length; i++) {
      if (Nfts[i].number == number) {
        Nft memory nft = Nfts[i];
        Nfts[i] = Nfts[Nfts.length - 1];
        Nfts.pop();
        (bool success, ) = msg.sender.call{value: nft.refound}("");
        require(success, "Refound failed.");
        break;
      }
    }
  }
}
```

Listing 1: Example of Solidity contract. Users can buy unique natural numbers from the contract and later delete the numbers they own to get their money back.

prefers. When the user realizes that owning numbers is not a great deal, he can give back the numbers he owns to get his money back.

The implementation of a Solidity contract is similar to the implementation of a class in an object oriented language. The `contract` keyword, followed by the name of the contract, starts the contract definition. The instance variable `counter` is part of the contract storage and saves the next NFT number to be created. It is set to 1 when the contract is initialized at deploy time. The map `ownedNfts` is also part of the contract storage and maps each user address to the NFTs it owns. Thus, the storage of the contract is composed of the contract's instance variables.

The `nftSeller` contract exposes two `public` functions to users: `buyNft` and `returnNft`. The former is `payable`, which means it can receive ETH from the caller. If the caller sends enough ETH, the function creates a new struct of type `Nft` (line 15), representing the number-asset purchased by the caller, and adds it to the list of the NFTs owned by the caller (line 16). The `Nft` struct records the amount of ETH paid by the buyer in the `refound` field. The ETH received by the contract are added to the contract balance. A user can invoke the `returnNft` function passing as input a number (`uint`). If the number is found in the list of the NFTs owned by the caller (line 23), the number is removed from the list and the user is refunded with the amount of ETH he paid for that number (line 27). The contract sends ETH to the sender calling `msg.sender.call{value: amount}("")`, where `msg.sender` is the address of the sender and `nft.refound` is the amount of ETH to send.

A `contract`, like a class in OOP, can be instantiated multiple times, and each instance has its own private balance, bytecode and storage. A smart contract published on chain is thus an instance of a Solidity `contract`, like an object is an instance of a class. The same Solidity `contract` can be deployed multiple times, by different users at different addresses. Apart from small exceptions, the storage of a contract instance can only be modified by the bytecode of the same instance. In Figure 1.2 we show a possible state of an instance of the `nftSeller` contract after some transactions.

Solidity contracts can't have class variables, that is variables shared by all the instances of the same Solidity `contract`. The Solidity `contract` of which the smart contract is an instance is not tracked on chain. Each contact is a standalone entity with a copy of its bytecode.

A contract, during its execution, can invoke functions of other contracts, and can create new contracts using the `new` keyword in the same way an object can allocate other objects in OOP. A contract created by another contract is no different form a contract created by an EOA: it is still standalone and it has no particular connection with its creator. It is not possible to allocate contracts with a limited scope, like a temporary object in OOP: all the contracts are allocated on the global state and they persist until they are explicitly deleted. A contract can delete itself from the blockchain by calling `selfdestruct` [4].

In Ethereum there is no built-in contract ownership mechanism. When a contract is published, every account (both a user or a another contract) can call its functions without restrictions. It is possible though to implement an access control mechanism in the code of the contract. Each function, before doing its job, can check if the caller is authorized to invoke the function, and if no, the contract can revert the

[4] `https://docs.soliditylang.org/en/v0.8.24/introduction-to-smart-contracts.html`

transaction. When a transaction is reverted, the state of the system is restored to the state before the transaction started, but the transaction is still recorded on the ledger and the caller is charged for the gas used to execute the transaction. This prevents a user from clogging the network with transactions that fail without paying the price.

1.4 Sui

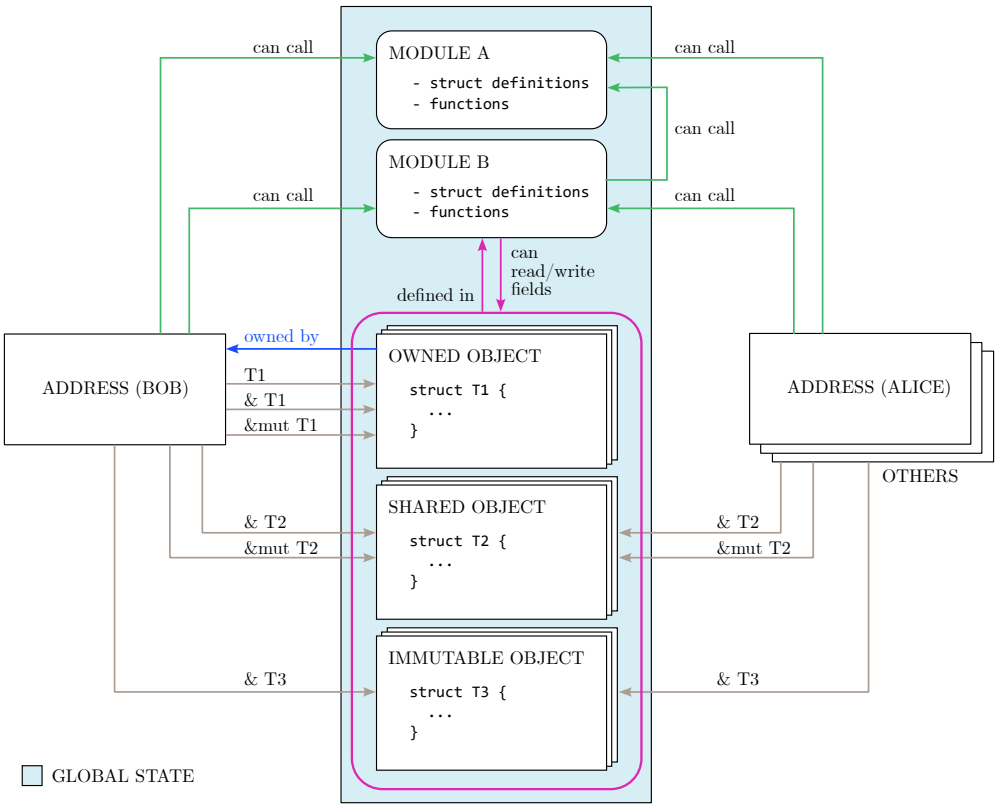

Figure 1.3: A simplified model of Sui. Bob and Alice are two users of the system. Each user owns some Sui objects that are Move structs. Two different modules are deployed on chain and each module can modify only the structs defined in it. Functions of a module can be invoked by any user. Only the owner of an *owned object* can consume it (passing the object by value to the transaction, i.e. `T1`), while *shared object* and *immutable object* are accessible (through a reference either mutable, i.e. `&T2`, or immutable, i.e. `&mut T3`) by all users.

Currently, the Sui blockchain is new in the scene [25] [27]. It was launched in 2022 by Mysten Labs, a group of former Meta (ex Facebook) employees. Sui native currency is the SUI, at the time this is written it counts 403 validator nodes and has a market cup of 899 million USD [5]. Sui is derived from Diem [2], a blockchain developed by Meta, announced in 2019 and recently abandoned. Sui shares some characteristics with Aptos, another young blockchain, because as the former, the latter was born

[5] `https://www.kraken.com/prices/sui`

```
struct Coin has key, store {
  id: UID,
  u64: amount,
}
```

```
struct Crowdfund has key {
  id: UID,
  endDonate: u64,
  goal: u64,
  receiver: address,
  donations: Coin,
}
```

Figure 1.4: Definition of a `Coin` and a `Crowdfund` struct in Move.

from the ashes of Diem and was founded by former Meta employees. Both Sui and Aptos are based on the Move language and the Move Virtual Machine (Move VM).

Benckmark - To better understand the Sui platform, we developed 11 smart contract examples representing common use cases. The examples were developed for the paper [3] written by Bartoletti at al., that carries out a qualitative evaluation of the main smart contract languages and blockchain platforms. In the paper, a set of significant use cases for smart contracts were selected and these use cases were implemented in different languages and for different blockchain platforms, with the goal of highlighting key differences, strengths and weaknesses of the languages and platforms.

In Appendix A we show three of the use cases we have implemented for Sui, with a brief description: Crowdfund, Auction and Escrow. The other use cases can be found in the github repository [22].

Modules - Modules are the type of smart contracts supported by Sui. A Module is a collection of functions and struct definitions.

Modules are developed in Sui Move, which is an adaptation of the original Move language used by Diem. Sui Move mainly replaces the primitives to interact with the global state that the original Move has, with different primitives that better fit the Sui storage model [14]. The Move language used by Aptos, instead, is more similar to the original one. For this reason, smart contracts written for Aptos [6], are not compatible with Sui, and vice versa. In what follows, when we talk about Move we always mean Sui Move, even though the feature of the language we are most interested on, that is linear types, belongs to the core of the language and doesn't depend on the specific Move adaptation.

Modules are grouped in packages. A package, with all its modules, is compiled off-chain and the bytecode of the package is deployed on-chain by a user sending it to the network using a specific transaction. When a package is published it receives a unique address and became immutable. A function published on-chain is identified by the triplet: package address, module name and function name. In the same way a struct definition (which is a type) published on-chain is identified by the triplet: address of the package, module name and struct name.

Sui Objects - Accounts in Sui are the users of the platform and have an associated 32 byte unique address, which is the public key of the account. Addresses owns Sui

[6] `https://aptos.dev/move/move-on-aptos`

objects that represent assets (or more generally resources), such as coins and non fungible tokens (NFT). A map stored on the ledger associates each address with the objects it owns. A Sui object is represented in Move as a `struct`, which is very similar to a C `struct`. Objects are created and published to the global state by Modules during the execution of transactions. In Figure 1.4 we show the definition of two structs in Move that can be published as Sui objects. On the left we have the definition of a `Coin` struct representing a pile of coins, which is similar to the definition of the native coin type in Sui (`sui::coin::Coin`). On the right we have the definition of a `Crowdfund` struct, which stands for a crowdfund campaign (see the Crowdfund use case in Appendix A.3). Each object has a unique identifier `UID`.

In Figure 1.3 we show a simplified model of Sui. In addition to *owned objects*, which have a single owner, Sui provides two types of object that are not owned by anyone: *shared objects* and *immutable objects*. A shared object is mutable while an immutable object can't be modified once published. Whether an object is owned, shared or immutable depends on the way the object is published.

Global State - The user sending a transaction designates the objects to be provided as input parameters telling their identifier (`UID`). Objects can be passed in input to a transaction in three ways: by value, by mutable reference or by immutable reference.

- When an object is passed by value, it is removed from the global state and the transaction can change the ownership of the object, modify or destroy it.
- When an object is passed by mutable reference it can't be destroyed by the transaction and its owner can't be changed, but its content can be modified.
- When an object is passed by immutable reference the transaction can only read the object.

We say a user *use* an object when he passes it in input to a transaction either by value, by mutable reference or by immutable reference. Owned objects can be used only by their owner, while shared objects and immutable objects can be used by anyone. This access control mechanism is enforced by the Move VM and it is represented in Figure 1.3 by the gray arrows.

Move - Move is a Rust-like procedural language [6]. As novelty, Move introduces *linear types* for the implementation of resources. Resources by their nature are scarce and valuable, so they should not be duplicated or accidentally lost. The resource construct in Move is used to represent digital assets such as coins and NFTs. By taking advantage of linear types, Move is able to prevent some common programming errors related to the use of resources that other languages (such as Solidity) are not able to detect. Move is purposely simple to be secure, easy to understand for developers and easy to analyze for static analysis tools. It comes with Move Prover [30]: a dedicated tool which allows to formally and automatically verify some correctness properties of the modules logic. Even if we think the Move Prover is an important tool that can be useful for reducing the number of bugs in smart contracts, for rising the confidence in the correctness of the code and for increasing the overall quality of smart contracts, we will not delve into it in this work.

In Listing 2 we show a simple Move module defining a seller of numbers as NFTs (the same example we saw for Solidity in Listing 1). It has no utility other than

```
module package::nftSeller {
  use sui::tx_context::{TxContext, sender}; use sui::object::{Self, UID};
  use sui::transfer::{transfer, public_transfer, share_object};
  use sui::sui::SUI; use sui::coin::{Self, Coin};

  struct State has key {
    id: UID,
    counter: u64,
  }

  struct Nft has key {
    id: UID,
    number: u64,
    refound: Coin<SUI>,
  }

  fun init(ctx: &mut TxContext) {
    let state = State { id: object::new(ctx),
      counter: 1,
    };
    share_object(move state);
  }

  public entry fun buyNft(state: &mut State, money : Coin<SUI>, ctx: &mut TxContext) {
    assert!(coin::value(&money) == state.counter, 0); // Check if the amount is ok
    let nft = Nft { id: object::new(ctx),
      number: state.counter,
      refound: move money,
    };
    state.counter = state.counter + 1;
    transfer(move nft, sender(ctx));
  }

  public entry fun returnNft(nft: Nft, ctx: &mut TxContext) {
    let Nft { id: id, number: _, refound: refound } = move nft;
    object::delete(id);
    public_transfer(move refound, sender(ctx));
  }
}
```

Listing 2: Simple Move module defining a contract that sells numbers as NFTs.

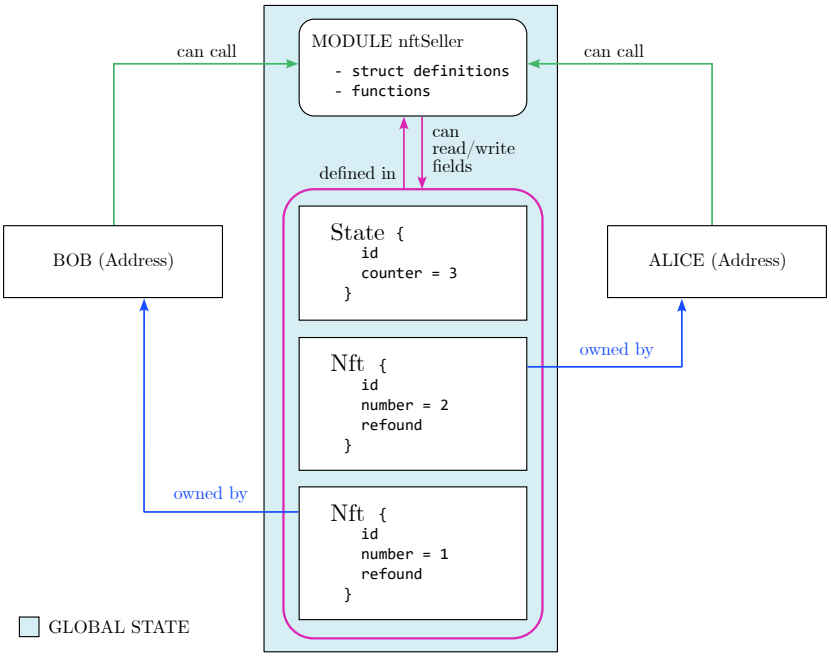

Figure 1.5: A configuration of the objects created by the module `nftSeller` (Listing 2). Bob bought and owns the NFT with the number 1, while Alice bought and owns the NFT with the number 2. The counter inside the State of the module is 3, so the next NFT to be sold will have the number 3.

showing the syntax and structure of a module.

The module name `nftSeller` is declared after the `module` keyword. Inside the scope of the module we find some imports, which allow to use functions and structs defined in other modules, two struct definitions and three functions. An instance of the `State` struct is created during the initialization of the module, in the `init` function, to hold the module's state: the value of the next `Nft` object to create (`counter`). The `init` function is called automatically when the module is published on-chain. The `State` struct is then published (line 19) on the Sui global state as a shared object, so that it is accessible by all the users of the system. Later, in Section 1.6, we will see in detail the semantics of the `move` operator, that is applied to the `state` variable during its publication. For now it is sufficient to know that `(move x)`, where `x` is a variable, denotes the value of `x`.

A user can call `buyNft`, passing as input a mutable reference to the unique and shared instance of `State`, and a certain amount of Sui money (in the `money` variable). Modules are stateless, so the state of the module (in the example the `State` instance), if needed, must be passed as input to the function by the caller. The `buyNft` function checks if the payed amount is enough and if so creates a new `Nft` struct. When we create a new struct we say that we are *packing* the struct (e.g. line 18 and 26), while when we decompose a struct in its components, as done in line 35 of Listing 2, we say that we are *unpacking* the struct. `buyNft` sets the `number` field of the new `Nft` to the current value of the counter, saves the `money` value in the `refound` filed of `nft`, increments the counter and transfers the `nft` object to the sender of the transaction (the user who called the function). The `Nft` struct is published on the global state as an owned object, so that only its owner can use it (line 31).

The owner of an `Nft` can delete it and obtain its money back by calling the `returnNft` function, which takes as input a value of type `Nft` and returns nothing. The `Nft` is *unpacked* inside the function (line 35) and the money in the `refound` field are transferred to the sender (line 37). When the user calls `returnNft`, the `Nft` object provided as input is removed from the global state. In Figure 1.5 is depicted a possible configuration of the objects created by the `nftSeller` module.

Note that transferring an amount of money to an address (line 37) means assigning to that address the ownership of a `Coin` object. So addresses don't have a balance of money, but they own a set of `Coin` objects. Users can split and merge the `Coin` objects they own to obtain `Coin` objects of the desired amount. A `Coin` is actually a pile of coins.

Summing up, Sui objects are instances of Move structs. A module initializes a struct, giving a value to all its fields, and publishes it as an owned, shared or immutable object. The type of a Sui object is the fully qualified name of its struct type, and it is recorded on-chain. Every address can own multiple objects of the same type.

Objects can be transferred from address to address by means of transactions. If we don't consider exceptions, this transfer can only be done by calling a function of the Module defining the object. *In general, the structs defined in a Module can be created, deleted, inspected, modified, and transferred only by the code of the same Module.* This ensures that properties of the structs which are invariant within the Module, are also invariant at the system level. For example, in the `nftSeller` module (Listing 2), the `counter` field of the `State` struct is initialized to 1 and it is always incremented; so it is always greater than 0. This property is guaranteed by the code of the module, and it is guaranteed at the system level because the `counter` field can only be modified by the module. Whenever a value of type `State` is passed as input to a function of the module, we can be sure the `counter` field is greater than 0.

Note that a function of a module different from `nftSeller` is allowed to receive as input a mutable reference to the `State` instance (`&mut State`), but the only thing it is allowed to do with that reference is to pass it as input to a function of `nftSeller`. In particular, it can't directly read or write fields of the `State` instance, and it can't destroy it.

The global state of the Sui system is composed by the set of modules with their bytecode, the set of shared objects and the set of owned objects with the mapping between object and owner.

1.4.1 Transaction example

The function `buyNft` of the `nftSeller` module can be executed by a client sending a transaction to the Sui network with the content shown on the left of Listing 3. The content of the transaction is show with a JSON-like syntax and it is not complete: we listed only the fields that are relevant for our discussion. We assume the package containing the function we want to call is published on-chain at the address `0x04`.

Follows the explanation of the transaction fields:

- **`method:`** The task we want to perform in the transaction. In this case we want

```
{ "transaction" : {
  "method"          : "moveCall",
  "package"         : "0x04",
  "module"          : "nftSeller",
  "function"        : "buyNft",
  "type_arguments" : [],
  "arguments"       : [
    "0x180", "0x283" ]
}}
```

```
{ "transaction" : {
  "method"          : "moveCall",
  "package"         : "0x04",
  "module"          : "nftSeller",
  "function"        : "returnNft",
  "type_arguments" : [],
  "arguments"       : [ "0x873" ]
}}
```

Listing 3: On the left: content of a Sui transaction that calls `0x04::nftSeller::buyNft`. On the right: content of a Sui transaction that calls `0x04::nftSeller::returnNft`.

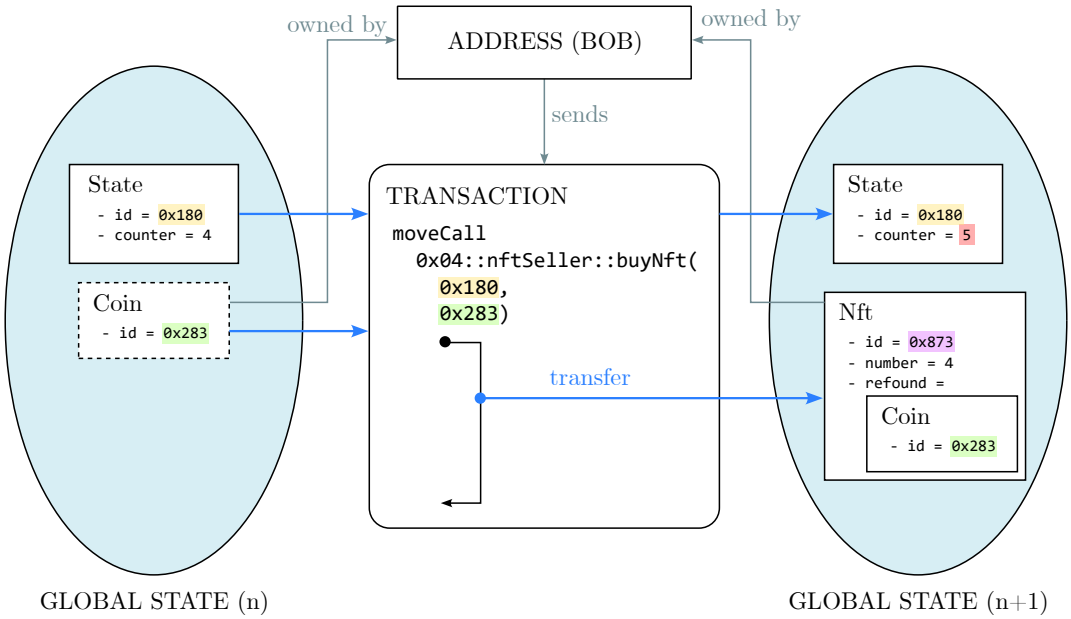

Figure 1.6: The evolution of the Sui global state during the esecution of the transaction shown on the left of Listing 3. The `State` (`0x180`) is passed by mutable reference to the function, while the `Coin` (`0x283`) is passed by value. The transaction modifies the `counter` filed of the `State` object and transfers a new `Nft` to the transaction sender.

to call a function of a module. Another possible value is `publish`, which is used to publish a Move package.

- **`package`** : The address of the package containing the function we want to call.
- **`module`** : The name of the module.
- **`function`** : The name of the function to call.
- **`type_arguments`** : The type arguments of the function. Move functions can be generic, so they can take arguments that are types. In this case the function we want to call is not generic, so the list is empty. Type arguments are specified as strings: for example, the `State` type of the `nftSeller` module is denoted with the string `"0x04::nftSeller::State"` (when the package containing the module is at the address `0x04`).
- **`arguments`** : The arguments of the function. `0x180` is the `UID` of a `State` object, while `0x283` is the `UID` of a `Coin<SUI>` object. The two objects become respectively the `state` and `money` parameters of the function. The last function argument (`ctx : &mut TxContext`) is not specified in the transaction, because it is automatically added by the Sui VM; it contains information about the transaction and the sender. Note that, the fact that the `state` object is passed by mutable reference to the `nftSeller`, while the `money` object is passed by value, is not specified in the transaction. The user knows the signature of the function and calling it, he accepts the way the function uses the objects passed as input.

In Figure 1.6, a diagram depicts the evolution of the Sui global state during the execution of the transaction just described.

On the right of Listing 3 we show the content of another transaction that calls `0x04::nftSeller::returnNft`. The function `returnNft` receives as argument the `Nft` object with identifier `0x873`, by value.

1.5 Other blockchains

We also did a preliminary study of Solana. The main goal of Solana is to solve the performance problems of blockchains. Its main programming language is Rust, a general purpose language for system programming. Although Rust is a recent language with a type system that can be considered advanced, we soon noticed that Solana does not use any particular feature to bring added value to the smart contract programmer. Rust is used as a low-level language on par with C/C++. The Solana API exposes many details of the underlying system, forcing the programmer to deal with marginal aspects such as deserialization of function parameters or explicit memory allocation for storing persistent data. We attribute this low level nature to the fact that Solana wants to be scalable and fast, and therefore does not want to allow itself the luxury of hiding implementation details. In our opinion, Solana does not provide an adequate level of abstraction for the development of smart contracts, and it is therefore not of interest for our work. For this reason we exclude it from the study.

More information and comments about Solana can be found in [3].

1.6 Move semantics

Move [6] [8] [24] is a strongly typed language where every variable and every value has a type. In Move, a variable is bounded to a memory location that may contain a value or not. A variable declaration creates a new empty memory location, while the assignment operator writes a value into the memory location of the left-hand side variable. There are programming languages, like Go or Java, where an uninitialized variable assumes a default value, and there are languages like C where an uninitialized variable assumes an undefined value (an unpredictable value). In Move, a variable that is not explicitly initialized has no value, and it cannot be used until it is assigned a value. This can be seen at line 2 of the following example.

```
1 let x : u64;
2 let y = x; // ERROR: Invalid usage of unassigned variable 'x'
3 x = 1;
4 let y = x; // OK
```

In Move, the expressions `(copy x)` and `(move x)` retrieve the value of the variable `x` in two different ways. The argument of the `copy` and `move` operators must be a variable.

Move - The expression `(move x)` evaluates to the value of `x` and clears its memory location. `(move x)` steals somehow the value of `x` and makes the variable unusable until it is assigned a new value. Using the `move` operator, a value can be moved from a memory location to another. *Every type of value can be moved.* In the next code snippet we have an error at line 3 because the value of `x` has been moved at line 2. When the program reaches line 3, the variable `x` is empty and cannot be used.

```
1 let x : u64 = 1;
2 (move x);
3 (move x); // ERROR: Invalid usage of previously moved variable 'x'.
```

Copy - The expression `(copy x)` evaluates to the value of `x` and leaves its memory location intact. Using the `copy` operator, a value can be duplicated. As we will see later, *not every type of value can be copied.* We say a type has the *copy* ability if its values can be copied. There are some types that can only be moved and so variables of those types can never appear as argument of a `copy` operator. In the following snippet the value of `x` is copied at line 2 and 3. The use of `x` at line 3 is valid because the copy at line 2 leaves the memory location of `x` intact.

```
1 let x : u64 = 1;
2 (copy x);
3 (copy x); // OK
```

The Move syntax permits to write `x` alone, instead of `(copy x)` or `(move x)`. The compiler automatically converts `x` to `(copy x)` or `(move x)` depending on the type of `x`. If `x` is of a copyable type, the compiler converts `x` to `(copy x)`, otherwise it converts `x` to `(move x)`. If the programmer wants to move a value of a copyable type, he must use the `move` operator explicitly. In any case, *when a value is retrieved from a variable,*

it is always either a copy or a move. Note that The Move Book[7], maintained by the Move core team, states a different rule for the automatic conversion of `x` to `(copy x)` or `(move x)`. In particular it states that `x` is converted to `(move x)` when the type of `x` is a struct with the copy ability, but this is not what we observed in the Sui Move compiler.

Whether a variable contains a value or not when the program reaches a given program-point its a runtime property. For example, in the following code, when the program reaches line 8, `x` contains a value only if the guard `g` is false, otherwise it is empty. Since the compiler can't statically determine wether the true of the false branch will be taken, it can't be sure `x` can be used after the branch and it reports an error. The compiler lets the programmer use a variable in a given program-point only if it is sure in every possible execution path the variable contains a value when the program reaches the program-point.

```
1 fun if_move(g : bool) {
2   let x : u64 = 1;
3   if (g) {
4     (move x); // The value of 'x' is moved away here.
5   } else {
6     (copy x);
7   };
8   (move x); // ERROR: Invalid usage of previously moved variable 'x'.
9 }
```

Drop - When an assignment writes a value into a non-empty memory location, the old value is overwritten with the new value. The old value is lost and we say the old value has been *dropped*. The assignment *drops* the value of the left-hand side variable. In the code below, the value `8` is dropped at line 2, when the assignment writes `9` into `x`.

```
1 let x : u64 = 8;
2 x = 9; // The value 8 is dropped.
```

Every variable, and by consequence every memory location, has a scope. The scope of a variable is the portion of the program where the variable is visible. When the scope of a location ends and the location contains a value, that value is no longer accessible, therefore a value *drop* happens. The end of a scope *drops* the values of all the variables whose visibility is bounded to that scope. In the code below, the value `8` is dropped at line 3, when the scope of `x` ends.

```
1 {
2   let x : u64 = 8;
3 };
```

As we will see later, *not every type of value can be dropped*. We say a type has the *drop* ability if its values can be dropped. When a variable has a non-droppable

[7]The Move Book: `https://move-language.github.io/move/variables.html#inference`
`https://github.com/move-language/move/blob/main/language/documentation/book/src/variables.md`

x state	x abilities	`copy x`	`move x`	`x = v`	`{ x }`
empty	-			✓	✓
empty	copy			✓	✓
empty	drop			✓	✓
empty	copy + drop			✓	✓
full	-		✓		
full	copy	✓	✓		
full	drop		✓	✓	✓
full	copy + drop	✓	✓	✓	✓

Table 1.1: The operations that can be performed on a variable x depending on its state and abilities. The operation indicated as `{ x }` is the closing of the scope to which x is bounded.

type, a value can be assigned to the variable only if its empty, and the scope to which the variable is bounded can be closed only if the variable is empty.

To summarize, in Move, at every moment at runtime a variable is either empty (doesn't have a value) of full (has a value). As shown in Table 1.1, depending on the type of the variable, and on its state (empty/full), some operations can be performed or not: for example, `(copy x)` can be performed only if the type of `x` is copyable and `x` is full, while `x = <EXPR>` only if `x` is empty or if it is droppable. The Move type checker ensures that in every execution path those constraints are respected.

1.7 Abilities

In the previous section we said that not every type of value can be copied or dropped. Wether this is possible or not depends on the *abilities* of the type. Types in Move can have a mixture of 4 different *abilities*. Each ability enables some operations. Those are the abilities and their meaning:

- **copy**: A value of a type with this ability can be copied.
- **drop**: A value of a type with this ability can be dropped.
- **key**: A value of a type with this ability can be published in the global state as a top-level object. All the fields of a struct with the *key* or *store* ability must have the *store* ability.
- **store**: A value of a type with this ability can be stored in the global state, as part of another struct with the *key* or *store* ability. A type with the *store* ability, but without the *key* ability, can't be published alone in the global state. In addition, an objects of a type with the *store* ability can be transferred between accounts outside of the module defining the type, using the function `public_transfer( <OBJ>, <DEST_ADDR> )` [8]. Objects with the *store* ability can be freely exchanged between users. As an example, the native type `sui::coin::Coin` has the *store* ability. We used this feature in Listing 2 at line 37, where we transferred the ownership of the `Coin<SUI>` value contained in the `Nft` to the sender of the transaction.

[8] `https://docs.sui.io/concepts/dynamic-fields/transfers/custom-rules`

All the primitive types, such as `bool` and `u64`, have the *copy*, *drop* and *store* ability, but not the *key* ability. So a value of primitive type can't be published alone in the global state. This is reasonable since the global state should be populated of resources (valuable assets mainly).

When the programmer defines a struct type, they must define also its abilities. The list of abilities of a struct appears after the name of the struct and it is preceded by the `has` keyword. In the snippet below we define a `Coin` type which is very similar to the native coin type of Sui. It has the *key* and *store* abilities, such that it can be published in the global state, it can be embedded in other types and it can be freely exchanged between users but it doesn't have the *copy* and *drop* abilities. Copying a `Coin` in a program would be like creating a new coin out of nothing in the physical world, which should be impossible. Dropping a `Coin` would be like physically destroying the coin (intentionally or by mistake), which is doable but undesirable.

```
1 struct Coin has key, store
2 {
3   id: UID,
4   value: u64,
5 }
```

Move poses some constraints in the way structs with different abilities can be composed. All the fields of a struct with the *store*, *drop* or *copy* ability must have the same ability. It is easy to see how these constraints are reasonable: the store/-copy/drop of a struct imply the store/copy/drop of all its fields. If we can't copy a `Coin`, we shouldn't be able to copy an `Nft` containing a `Coin` (as in the Listing 2), otherwise we would be able to copy the `Coin`. In addition, the Sui platform (not the Move language) requires that all the structs with the *key* ability include an `id` field of type `sui::object::UID`, which is the unique identifier of a struct instance in the global state. Since `UID` is a non-copyable and non-droppable type, all the structs with the *key* ability can't be copyable or droppable. It follows that the all the objects in the Sui global state are non-copyable and non-droppable structs with the *key* ability.

1.8 Errors preventable by Move's Typing

Taking the "NFT seller" examples from the previous sections (Listing 1, and Listing 2), we can show that Move is able to prevent some errors, in the management of valuable assets, that Solidity cannot.

Copy error - The `Nft` struct, both in the Solidity example and in the Move example represent an asset. A user purchases a `Nft` which should be guaranteed to be unique and which should be impossible to accidentally lose.

In Solidity, from a value of type `Nft` it is possible to create any number of copies of the same value. It is therefore possible to mistakenly assign two copies of the same asset to the same user, or to assign two copies of the same asset to different users. The function `buyNft` of the module `nftSeller` can be modified as follows without causing any error at compile time:

```
function buyNft() public payable { //...
  Nft memory nft = Nft(counter, msg.value);
  ownedNfts[msg.sender].push(nft);
  ownedNfts[msg.sender].push(nft);
  //...
}
```

Two copies of the same `Nft` struct are added to the array of assets owned by the user (`ownedNfts[msg.sender]`). While the error is obvious here, depending on the complexity of the code that follows the asset creation, it may not always be so.

This kind of error can't happen in Move because the type system prevents the copy of a value of an asset type like `Nft`, which has not the *drop* and *copy* abilities (described in Section 1.7). An attempt to publish the same asset twice, like the following, is identified by the Move compiler.

```
public entry fun buyNft(state: &mut State, money : Coin<SUI>,
    ctx: &mut TxContext) { //...
  let nft = Nft { id: object::new(ctx), x: state.counter, refound: move money };
  // ...
  transfer(move nft, sender(ctx));
  transfer(move nft, sender(ctx)); // <- compile error
}
```

The error reported by the compiler at line 6 is "*Invalid usage of previously moved variable 'nft'*", because the value of `nft` has been moved at line 5 and so `nft` is empty at line 6. Note that there is no way to make this code compile, even if we try to use the `copy` operator on `nft` instead of the `move` operator. The `copy` operator can't be used on variables of non-copyable type (such as `Nft`), and the compiler will report the error "*Invalid 'copy' of value without the 'copy' ability*" if we try to do so.

Drop error 1 - In Solidity, it is possible to silently and accidentally lose a value (an asset) of type `Nft`, causing in that way a loss for its owner. Assume, for example, in the `buyNft` function the programmer wants to increment the counter by 1 when the counter is less than 10, and by 2 otherwise. By doing this, he forgets to store the newly created `Nft` resource somewhere, when the counter is less than 10:

```
function buyNft() public payable { //...
  Nft memory nft = Nft(counter, msg.value);
  if (counter < 10) {
    counter ++;
  } else {
    counter += 2;
    ownedNfts[msg.sender].push(nft);
  }
}
```

The `Nft` asset in the `nft` variable is lost when the `counter` is less than 10 and `nft` goes out of scope. In this case the buyer does not receive the asset. As before, the error

is obvious here, but depending on the complexity of the function's logic it may not be so.

In Move, a value of an asset type can't be accidentally lost; it can only be explicitly unpacked, published or passed to another function, that in turn must do one of these three things with the received value. If we try to replicate the same error in Move, the compiler will identify it:

```
1  public entry fun buyNft(state: &mut State, money : Coin<SUI>,
2    ctx: &mut TxContext) { //...
3    let nft = Nft { id: object::new(ctx), x: state.counter, refound: move money };
4    if (state.counter < 10) {
5      state.counter = state.counter + 1;
6    } else {
7      state.counter = state.counter + 2;
8      transfer(move nft, sender(ctx));
9    }
10 } // <- compile error
```

The code above produces the following compile error: "*The local variable 'nft' might still contain a value. The value does not have the 'drop' ability and must be consumed before the function returns*". When `state.counter` is less than 10, the value of the `nft` variable would be dropped at the end of the function, and this is an error (as explained in Section 1.7).

Note that in the Move version of `buyNft` the `money` parameter, which is an asset of type `Coin<SUI>`, is moved (with the `move` operator) to the `refound` field of the `Nft` struct (line 3). The body of the function is forced to use somehow the `money` asset (in one of the three ways described above) before the function returns, otherwise the compiler will report a drop error.

Drop error 2 - In Solidity an asset can also be lost when a variable pointing to the asset is overwritten like in the following example:

```
1  Nft memory n1 = Nft(1, 0);
2  Nft memory n2 = Nft(4, 0);
3  n1 = n2; // <- n1 is overwritten and the asset Nft(1, 0) is lost
```

The Move equivalent doesn't compile:

```
1  let n1 = Nft { id: object::new(ctx), x: 1, refound: coin::zero(ctx) };
2  let n2 = Nft { id: object::new(ctx), x: 4, refound: coin::zero(ctx) };
3  n1 = move n2; // <- compile error
4  n1 = copy n2; // <- compile error
```

The error is "*Invalid assignment to variable 'n1'. The variable contains a value that does not have the 'drop' ability and must be used before you assign to this variable again*".

1.9 Substructural Type System

Move has a substructural type system that permits to define *normal*, *linear*, *affine* and *relevant* struct types using abilities [21]. Each category of structs has different constraints on the way its values can be consumed. We first explain what does it mean to *consume a value* and then we will explain the 4 categories. There are only two ways in which a struct value can be consume: it can be unpacked or it can be published.

Unpack - A value is unpacked using `let <STRUCT DEF> = <STRUCT>`, like in the following example which is extracted from Listing 2. The `returnNft` function receives a `Nft` as input and unpacks it at line 9. After the unpacking the variable `nft` is empty. The value retrieved from the variable is deconstructed by the `let`: the `Nft` value is deleted but its fields, themselves values, are moved into newly created variables and stay alive. In this case, the `id` and the `refound` fields (on the left-hand side of the semicolon `:`) are moved respectively into the `id` and `refound` variables (right-hand side of the semicolon `:`), while the `x` field is dropped using the `_` - underscore name. The underscore can be used only on droppable fields, like `x` that is a droppable `u64`. After the unpack of `nft`, there exists no more a value of type `Nft` in the function. Note how the fields of the struct being unpacked, if non-droppable, must be consumed by the following code. Unpack consumes a struct value but it doesn't consume the value of its fields.

```
1  //...
2    struct Nft has key, store {
3      id: UID,
4      x: u64,
5      refound: coin::Coin<SUI>,
6    }
7  // ...
8    public entry fun returnNft(nft: Nft, ctx: &mut TxContext) {
9      let Nft { id: id, x: _, refound: refound } = move nft;
10     // ... do something with 'id' and 'refound'
11   }
12 // ...
```

Publish - As said before, in Sui, a value with the *key* ability can be published in the global state as an owned, shared or immutable object. A function is available in the module `sui::transfer` for each kind of publication: `transfer(<OBJ>, <RECEIVER>)` (and `public_transfer(<OBJ>, <RECEIVER>)`), `share_object(<OBJ>)` and `freeze_object(<OBJ>)`. All the functions returns nothing and their parameters are self-explanatory.

When a struct is published, it is moved into the global state taking all its fields with it. The published value is moved to the global state and so disappears from the program and it can't be manipulated anymore in the current transaction. Unlike the unpack, the publish consumes the struct value and all its fields (recursively). In the snippet below, again extracted from Listing 2, the `share_object(counter)` call consumes the `State` value and the nested `UID` and `u64` values.

```
//...
  struct State has key, store {
    id: UID,
    counter: u64,
  }
// ...
  fun init(ctx: &mut TxContext) {
    let state = State { id: object::new(ctx), counter: 1, };
    share_object(move state);
  }
// ...
```

We can now explain the 4 categories of struct types. In Table 1.2 we show for each combination of *copy* and *drop* abilities a struct can have, the category it is and the constraints on the consumption of its values.

Move Abilities	Kind of Type	Value consumption
`copy drop`	*Normal*	Arbitrarily
`drop`	*Affine*	At most once
`copy`	*Relevant*	At least once
-	*Linear*	Exactly once

Table 1.2:

A struct copyable and droppable is a *Normal* type, like a Solidity or C struct. A value of a *normal* type can be consumed arbitrarily many times: it can be unpacked and published multiple times (included in other structs with the *key* ability) because multiple copies of the same value can be created.

A struct non-copyable and non-droppable is a *Linear* type. The type system guarantees each *linear* value is consumed exactly once in the program. As a consequence, since all the Sui objects passed as input to a transaction are *linear* values, in every valid transaction, each Sui object in input is consumed in one of three ways: it is explicitly unpacked, it is published as a top-level object, or it is published as a field of another object. The same holds for a *linear* value packed inside the transaction: it must be consumed in exactly one of the three ways. Linear types are a good way to represent valuable resources.

An *affine* value can be consumed at most once while a *relevant* value must be consumed at least once. The utility of these types is more marginal in the blockchain environment.

Chapter 2

Formal Move FM

2.1 Introduction

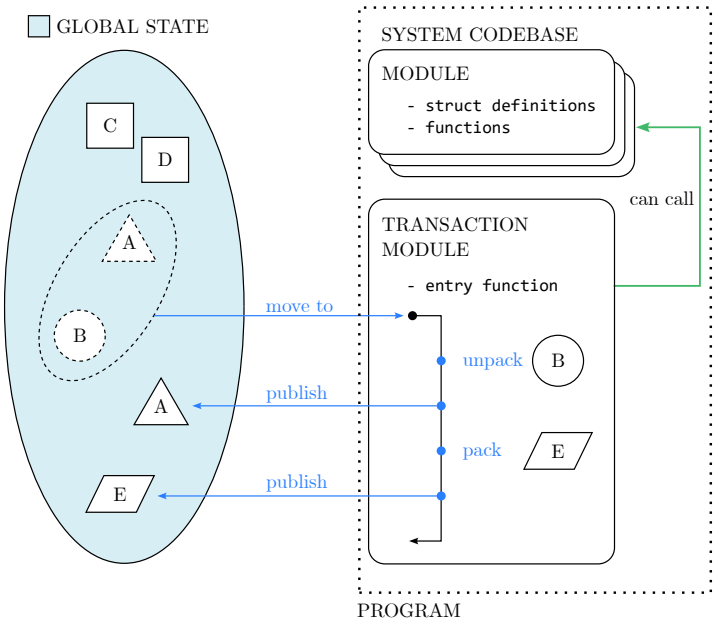

Figure 2.1: A diagram of a tranasction execution in the simplified blockchain model. When the main function of the transaction module is called, resources A and B are moved from the global state to the transaction. The transaction destroys resource B unpacking it, republishes resource A and creates and publishes the new resource E.

Inspired by the Move language, and by its use in Sui and Aptos blockchains, we define a small core language for smart contracts with linear and normal types. We call it FM which stands for Formal Move.

Follows an explanation of the blockchain system in which we imagine the language applied. However, we specify that the scope of this explanation is only to give the reader additional context, to better understand the language and its use. Many mechanisms mentioned below, such as the global state update and the execution of multiple transactions in sequence, will not be explored in depth.

Figure 2.1 shows the system model. The state of the system (the *global state*) is a

set of resources (linear values). Resources are not associated to accounts, and access control mechanisms on resources are not considered. The global state is modified executing transactions. When a transaction is invoked, some resources are moved from the global state to the transaction, so that the transaction can use those resources. The transaction, during execution, can publish resources that become part of the global state. As a net result, a transaction removes some resources from the global state (only during its invocation), and adds some resources to it.

This is similar to the Sui blockchain : a transaction can only manipulate resources that have been provided to it, and when the transaction publishes a resource, it loses control over the resource.

The system contains a fixed set of modules: the *system codebase*. Each module contains struct and function definitions. We don't model the possibility of adding modules or updating existing ones. The system codebase can include any type of smart contract or library.

A transaction is executed calling the *main function* of a *transaction module* passing to it a list of resources taken from the global state, and a list of normal values. The transaction module can be changed on each transaction execution, and it contains only the main function definition (the main function plays the role of the Programmable Transaction Block in Sui [1], or the Move Script in Aptos [2]). The main function can call functions defined in the system codebase with no restriction. When a transaction terminates, the global state is updated, the transaction module is replaced (if needed) and the next transaction can start.

Our language is used to write the *system codebase* and the *transaction module*. The transaction execution happens on-chain and the system codebase is stored on-chain.

[1] `https://docs.sui.io/concepts/transactions/prog-txn-blocks`

[2] `https://aptos.dev/move/move-on-aptos/move-scripts/`

2.2 Syntax

Term t	::=	v	value
	|	x	variable
	|	$x.j$	select j-th field of x
	|	`let` $x = t_1$ `in` t_2	let binding
	|	`call` $\mathrm{M.F}\,[\,\overline{t}\,]$	function call
	|	`pack` $\mathrm{M.S}\,[\,\overline{t}\,]$	constructor
	|	`unpack` $\{\overline{x}\} = t_1$ `in` t_2	deconstructor
	|	`if` t_1 `then` t_2 `else` t_3	
	|	`pub` t	publish a resource
	|	$\underline{\texttt{exec}\ \mathrm{M}\ t}$	function body
	|	$\underline{v.j}$	select j-th field of v

Value v	::=	n	integer
	|	$\underline{\texttt{struct}\ \{k\}\,\mathrm{M.S}\,[\,\overline{v}\,]}$	struct value $k \in K$

Figure 2.2: FM Terms

Terms - The language is a core functional language with integers, structs and functions. We say it is functional in the sense that there are no mutable variables, but the language doesn't have functions as first class values. Adding abstractions could be an interesting aspect to explore in the future.

In Figure 2.2 we show all the terms of the language. The terms that can be written by the programmer are a subset of the language terms, in particular, terms with the underline can appear only at runtime and can't be used by the programmer.

A value can be either an integer n or a struct `struct` $\{k\}\,\mathrm{M.S}\,[\,\overline{v}\,]$ which is an array of values $\overline{v}$ together with the fully qualified name of a struct type M.S and an identifier k taken from a set of unique identifiers K. In M.S, M is the name of a module and S the name of a struct defined in that module. The identifier k plays a role in the dynamic semantics: it is used to trace the identity of a resource, which will be useful to state and prove the resource preservation theorem (RPT) : Theorem 2.

Struct fields don't have names, they are identified by their position in the tuple. A field can be selected using $x.j$, where x is a variable and j is the index of the field. The term on which the selection is performed must be a variable. This is needed to ensure resource preservation (Theorem 2). Although it may seem a limitation for

$$
\begin{array}{rcll}
\text{Type T} & ::= & \text{Int} & \text{integer} \\
& | & \text{M.S} & \text{struct name}
\end{array}
$$

Figure 2.3: FM Types

$$
\begin{array}{rcll}
\text{FD} & ::= & \texttt{fun}\ \text{F}\,(\,\overline{x} : \overline{\text{T}}\,) : \text{T}_r\,\{t_b\} & \text{function definition} \\
\text{SD} & ::= & \texttt{str}\ \text{S}\,\{\,\top\,,\ \overline{\text{T}}\,\} \ \mid\ \texttt{str}\ \text{S}\,\{\,\bot\,,\ \overline{\text{T}}\,\} & \text{struct definition} \\
\text{MD} & ::= & \text{M}\,\{\,\overline{\text{SD}}\,,\ \overline{\text{FD}}\,\} & \text{module definition} \\
\text{P} & ::= & \overline{\text{MD}} & \text{program}
\end{array}
$$

Figure 2.4: FM Program

the language expressiveness, it is not. The selection of a field of a generic term t can be written equivalently as the term $\texttt{let}\, x = t\, \texttt{in}\, x.j$.

The let term $\texttt{let}\, x = t_1\, \texttt{in}\, t_2$ binds t_1 to the name x such that x can appear in t_2.

Using $\texttt{call}\, \text{M.F}\,[\,\overline{t}\,]$, the programmer can invoke a function defined in any module. M.F is the fully qualified name of the function, while $\overline{t}$ is the list of arguments. A function can call itself recursively.

A struct of type M.S can be created using $\texttt{pack}\, \text{M.S}\,[\,\overline{t}\,]$, where $\overline{t}$ is the list of terms that, after being reduced to values, are assigned to the struct fields.

A struct t_1 can be deconstructed (or unpacked) using $\texttt{unpack}\,\{\overline{x}\} = t_1\, \texttt{in}\, t_2$. t_1 is a term that reduces to a struct value, $\overline{x}$ is a vector of variable names: one for each field of the struct, and t_2 is the body of the unpack (the rest of the program). During unpack, the struct t_1 is eliminated and each field of the struct is bound to a variable whose scope is the body t_2.

The term $\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$ behaves as usual: if t_1 reduces to the integer 0 then t_3 is evaluated, otherwise t_2 is evaluated.

The programmer can move a resource t to the global state using $\texttt{pub}\ t$. When a resource is published, it is removed from the transaction and added to the global state.

Types - Figure 2.3 shows the types of the language. FM has a single base type, the integer. Other base types, like booleans, unit, etc. can be added easily. Since they don't add any interesting complexity or challenge to the language, we omit them for simplicity. Every fully qualified struct name is also a type.

Programs - In Figure 2.4 we show the structure of a program and its components, which are modules, structs and functions.

A function $\texttt{fun}\ \text{F}\,(\,\overline{x} : \overline{\text{T}}\,) : \text{Int}\,\{t_b\}$ is defined giving a name F, a list of parameter names $\overline{x}$ with their types $\overline{\text{T}}$, a return type T_r, and a term that is the body of the

function t_b. The function parameters may appear free in the body, but no other variable is allowed to appear free in the body.

A struct definition $\texttt{str}\ \mathrm{S}\,\{\,\top\,,\ \overline{\mathrm{T}}\,\}$ contains a boolean flag ($\top$ or $\bot$) and a list of types $\overline{\mathrm{T}}$, which are the types of the struct fields. The flag is used to say if the struct is linear or not. True ($\top$) means that the struct is linear (a resource), false ($\bot$) means that the struct is normal. The programmer assigns the flag to each struct they define.

A module $\mathrm{M}\,\{\,\overline{\mathrm{SD}}\ ,\ \overline{\mathrm{FD}}\,\}$ wraps a list of struct definitions and a list of function definitions. We assume all the struct and function names to be unique inside a module. The module has a name M, so the fully qualified name of a function is M.F and the fully qualified name of a struct is M.S. As we will see later, similarly to what happens in Sui Move, values of type M.S can be packed, unpacked and inspected only by functions defined in the same module M.

In our simplified model, the *system codebase* combined with the *transaction module* forms a set of modules that we call a program P. The transaction module is a special module `tm` that contains only the *main function* `main` as defined below:

$$\begin{array}{rcll} \mathrm{FD}_0 & = & \texttt{fun main}\,(\,\overline{x} : \overline{\mathrm{T}}\,) : \mathrm{Int}\,\{t_b\} & \text{main function} \\ \mathrm{MD}_0 & = & \texttt{tm}\,\{\,\overline{0}_{\mathrm{SD}}\ ,\ \mathrm{FD}_0\,\} & \text{transaction module} \end{array}$$

With $\overline{0}_{\mathrm{SD}}$ we denote the empty list of struct definitions. More in general, with $\overline{0}_{\Omega}$ we denote the empty list of elements of type Ω.

The main function must return an integer [3], while main function's parameters can be both linear or normal. *The transaction is executed calling the main function with some resources moved from the global state, and some custom normal values.* In Section 2.2.2 we will see some examples of transaction modules and how they are executed.

As said before, underlined terms emerge only at runtime. The term $\underline{\texttt{exec}\,\mathrm{M}\ t}$ encodes a running function call : the function body t executing in the module M. The term $\underline{v.j}$ appears when the variable x of a selection $x.j$ is substituted with the value v by a substitution. Finally, a struct $\underline{\texttt{struct}\ \{k\}\,\mathrm{M.S}\,[\,\overline{v}\,]}$ can only be created at runtime executing a $\texttt{pack}\,\mathrm{M.S}\,[\,\overline{v}\,]$, and can't be written directly in the program. This serves two purposes: first, it allows us to assign a unique identifier k to each struct instance at the moment of its creation, and second, it allows us to check that the module creating the struct is the same module defining the struct.

Linear & Normal Types - A type can be linear or normal (non-linear). The function IsLinear(T), shown below, defines which are the linear types. Base types are normal, while struct types are linear if they have the linear flag set to true in their definition. For a more compact notation, we will write IsLinear(T) for IsLinear(T) = $\top$ and $\neg$IsLinear(T) for IsLinear(T) = $\bot$.

[3] When we force the main function to return an integer, we don't have to decide what to do if a transaction returns a resource. All the resources passed to (moved to) the transaction must be explicitly published or unpacked inside the transaction.

$$
\begin{array}{lcll}
\multicolumn{4}{l}{\text{IsLinear} : \text{Type} \to \text{Bool}} \\
\text{IsLinear}(\text{Int}) & = & \bot & \\
\text{IsLinear}(\text{M.S}) & = & \top & \text{when } \texttt{str}\ \text{S}\,\{\,\top\,,\ \overline{\text{T}}\,\} \in \text{M} \\
\text{IsLinear}(\text{M.S}) & = & \bot & \text{when } \texttt{str}\ \text{S}\,\{\,\bot\,,\ \overline{\text{T}}\,\} \in \text{M}
\end{array}
$$

We define the set of free variables of a term t in the conventional way.

$$
\begin{array}{rcl}
\text{Free variables} & : & \text{Term} \to \text{Set} \subseteq X \\
fv(n) & = & \varnothing \\
fv(x) & = & \{x\} \\
fv(x.j) & = & x \\
fv(v.j) & = & fv(v) \qquad (= \varnothing) \\
fv(\texttt{pub}\ t) & = & fv(t) \\
fv(\texttt{let}\ x = t_1\ \texttt{in}\ t_2) & = & fv(t_1) \cup (fv(t_2) \backslash \{x\}) \\
fv(\texttt{call}\ \mathrm{M.F}\,[\,\overline{t}\,]) & = & fv(\overline{t}) \\
fv(\texttt{pack}\ \mathrm{M.S}\,[\,\overline{t}\,]) & = & fv(\overline{t}) \\
fv(\texttt{struct}\ \{k\}\ \mathrm{M.S}\,[\,\overline{v}\,]) & = & fv(\overline{v}) \qquad (= \varnothing) \\
fv(\texttt{unpack}\ \{\overline{x}\} = t_1\ \texttt{in}\ t_2) & = & fv(t_1) \cup (fv(t_2) \backslash \{\overline{x}\}) \\
fv(\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3) & = & fv(t_1) \cup fv(t_2) \cup fv(t_3) \\
fv(\texttt{exec}\ \mathrm{M}\ t) & = & fv(t) \\
fv(\overline{t}) & = & \bigcup_i fv(t_i)
\end{array}
$$

The definition of substitution is standard. All the occurrences of the variable y in t are substituted with the value v.

$$
\begin{array}{rcl}
\text{Substitution} & : & \text{Term} \to \text{Term} \\
n\{y := v\} & = & n \\
x\{y := v\} & = & v \quad \text{when } x = y \\
x\{y := v\} & = & x \quad \text{when } x \neq y \\
x.j\{y := v\} & = & v.j \quad \text{when } x = y \\
x.j\{y := v\} & = & x.j \quad \text{when } x \neq y \\
v_1.j\{y := v\} & = & (v_1\{y := v\}).j \qquad (= v_1.j) \\
\texttt{pub}\ t\{y := v\} & = & \texttt{pub}\ (t\{y := v\}) \\
\texttt{let}\ x = t_1\ \texttt{in}\ t_2\{y := v\} & = & \texttt{let}\ x = t_1\{y := v\}\ \texttt{in}\ t_2\{y := v\} \\
 & & y \neq x \\
\texttt{call}\ \mathrm{M.F}\,[\,\overline{t}\,]\{y := v\} & = & \texttt{call}\ \mathrm{M.F}\,[\,\overline{t}\ \{y := v\}\,] \\
\texttt{pack}\ \mathrm{M.S}\,[\,\overline{t}\,]\{y := v\} & = & \texttt{pack}\ \mathrm{M.S}\,[\,\overline{t}\ \{y := v\}\,] \\
\texttt{struct}\ \{k\}\ \mathrm{M.S}\,[\,\overline{v}_1\,]\{y := v\} & = & \texttt{struct}\ \{k\}\ \mathrm{M.S}\,[\,\overline{v}_1\ \{y := v\}\,] \\
\texttt{unpack}\ \{\overline{x}\} = t_1\ \texttt{in}\ t_2\{y := v\} & = & \texttt{unpack}\ \{\overline{x}\} = (t_1\{y := v\})\ \texttt{in}\ (t_2\{y := v\}) \\
 & & \forall i.y \neq x_i \\
\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3\{y := v\} & = & \texttt{if}\ t_1\{y := v\}\ \texttt{then}\ t_2\{y := v\}\ \texttt{else}\ t_3\{y := v\} \\
\texttt{exec}\ \mathrm{M}\ t\{y := v\} & = & \texttt{exec}\ \mathrm{M}\ (t\{y := v\})
\end{array}
$$

2.2.1 Syntax Example

In Listing 4 we show a toy example of an FM module implementing the "NFT seller" contract already show in the introductory sections of Ethereum and Sui (Listing 1 and Listing 2). To give a little bif of context, in Listing 5 we also show a portion of an hypothetical `modCoin` module that implements a currency. To make the code more readable, we added names to struct fields in struct definitions and we used those names in selections. Furthermore, we used unqualified names for structs and functions instead of fully qualified names when appropriate. Since we used unique names for structs and functions there is no ambiguity.

In the example we assume the existence of three binary operators for integers (arguments of type `Tint` only) we have not defined in the syntax : the logic and `&&`, the comparator `>=` and the sum `+`. They behave as expected : `a >= b` returns 1 if `a` is greater than or equal to `b`, 0 otherwise; `a + b` returns the sum of the two integer operands; `a && b` returns 1 if both operands are different than zero, 0 otherwise. In what follows, we may also assume the existence of other standard operators for integers, such as `<`, `>`, `-`, `*`, `/`, etc. Those operators are not interesting and can be easily added to the language, so we omit them in the formalization for simplicity.

Still to facilitate the understanding of the example we used the syntactic sugar t_1 `;` t_2 to sequence the terms t_1 and t_2 instead of using a let binding. The term t_1 `;` t_2 is equivalent to `let` $x = t_1$ `in` t_2 with x chosen such that it is not free in t_2.

The FM module `modSeller` in Listing 4 is similar to the Move counterpart (Listing 2). The `init` function packs a `State` initializing its `counter` field to `1`, and then publishes it.

The `buyNft` function, when invoked by the blockchain runtime receives the state of the module (`state`) and an amount of money (`money`). If the money is not enough (line 16), the function republishes the resource it received and returns the error code `-1` (line 24-25). Otherwise it creates a new `Nft` resource moving inside it the money (line 18). To update the `counter` field of the `state` struct, it unpacks the `state` (line 19), increments the `counter` field, and then packs a new updated `State` (line 20). This is the only way to change a struct field in FM: the programmer must unpack the struct, change the field, and then pack the struct again. Finally, the function publishes the new `State` and the new `Nft` (line 21-22).

As in Move, the FM type system will prevent resource duplication and resource loss errors. Forgetting to publish the new module state, removing for example the `pub newState` at line 21, will result in a type error (a drop error). Republishing two times the same amount of money, adding for example another `pub money` at line 25, will result in a type error (a copy error).

When a user calls `returnNft` passing a `Nft` resource taken from the global state as input, the `nft` is destroyed and the money within it is published (line 30).

As can be seen from the example, we have not defined and formalized an access control mechanism in our model. Every user is allowed to use any resource from the global state without restrictions. The study of a simple and flexible access control mechanism and of its relationship with the language's type system is left as future work.

```
modSeller {
  str State { T,
    counter : Tint
  }
  str Nft  { T,
    number : Tint,
    refound : Coin
  }

  fun init () : Tint {
    let counter = pack Counter [1] In
    pub counter
  }

  fun buyNft(state : State, money : Coin) : Tint {
    if (call modCoin.getValue [money] >= state.counter)
    then
      let nft = pack Nft [state.counter, money] In
        unpack { counter } = state In
          let newState = pack State [counter + 1] In
            pub newState;
            pub nft
    else
      pub counter;
      pub money; -1
  }

  fun returnNft(nft: Nft) : Tint {
    unpack { number, refound } = nft In
      pub refound
  }
}
```

Listing 4: A module that sells numbers as assets, written in FM.

```
modCoin {
  str Coin { T,
    amount : Tint
  }

  fun getValue (coin : Coin) : Tint { coin.amount }
  // ...
}
```

Listing 5: Fragment of an hypothetical coin module `modCoin`, written in FM.

2.2.2 Transaction Examples

Assuming the system codebase of our system is composed by the `modSeller` and `modCoin` modules seen in Section 2.2.1, in Table 2.1 and Table 2.2 we show two examples of transaction modules that can be executed. Every transaction module defines a single function (`main`) that receives as input a list of resources taken from the global state and a list of normal values, and returns an integer value. The transaction is executed calling the `main` function of the transaction module. The tables show the transaction module code (written in FM), the global state values used in the transaction, and the term that is reduced according to the FM operational semantics (detailed in Section 2.3) to execute the transaction.

In Table 2.1 we show a transaction module a user can request to be executed to buy a new number-asset. The main function receives a `State` resource and a `Coin` asset from the global state and calls the `buyNft` function of the `modSeller` module forwarding to it the `State` and the `Coin`. In the box "Global state input values" we show two examples of values that may live in the global state and that can be used in the transaction. The box "Transaction term" shows the term that is reduced to execute the transaction. The box "Return value" shows the value returned by the transaction and the box "Global state output values" shows the values that are published in the global state after the transaction is executed.

In Table 2.2 we show a transaction module a user can request to be executed to delete a number-asset and redeem the money inside it. The main function receives a `Nft` asset from the global state and calls the `returnNft` function of the `modSeller` module forwarding to it the `Nft`.

Transaction module

```
tm {
  fun main(state : State, money : Coin) : Tint {
    call modSeller.buyNft [state, money]
  }
}
```

Global state input values

```
- v1 = struct State [4]
- v2 = struct Coin [7]
```

Transaction term

```
call tm.main [ v1, v2 ]
```

Return value

0

Global state output values

```
- struct Nft [
  4,               // number
  struct Coin [7]  // refound
]
```

Table 2.1: Example of a transaction module a user can request to be executed to buy a new number-asset.

Transaction module

```
tm {
  fun main(nft : Nft) : Tint {
    call modSeller.returnNft [nft]
  }
}
```

Global state input values

```
- v3 = struct Nft [
  4,               // number
  struct Coin [7]  // refound
]
```

Transaction term

```
call tm.main [ v3 ]
```

Return value

0

Global state output values

```
- struct Coin [7]
```

Table 2.2: Example of a transaction module a user can request to be executed to delete a number-asset and redeem the money inside it.

2.3 Operational Semantics

The operational semantics uses a call-by-value evaluation strategy, rather standard and straightforward. The evaluation judgment $\mathrm{M} \ni t \ \rightarrow \ t'$ is read: "In module M, t can do a step and go to t'". There are terms that can do a step only in a specific module. In particular, only the code of the module defining a struct can perform operations on instances of that struct: only M can pack, unpack the struct M.S and select its fields.

As previously said, a program is the set of modules formed by the fixed *system codebase* and the *transaction module* (`tm`). The transaction module defines the *main function* called `main`. The program is executed reducing the term $\texttt{call main}\,[\,\overline{v}\,]$ in the transaction module `tm`. The vector $\overline{v}$ can contain resources taken from the global state and normal values. The output of the execution is an integer value which may indicate a success or a failure (but this is not required).

$$\frac{\mathrm{M} \ni t_1 \ \rightarrow \ t_1'}{\mathrm{M} \ni \texttt{let}\, x = t_1 \,\texttt{in}\, t_2 \ \rightarrow \ \texttt{let}\, x = t_1' \,\texttt{in}\, t_2} \ \textsc{E-let}$$

$$\frac{}{\mathrm{M} \ni \texttt{let}\, x = v \,\texttt{in}\, t_2 \ \rightarrow \ t_2\{x := v\}} \ \textsc{E-let2}$$

We start with the semantics of the let binding (E-LET and E-LET2 below). The term t_1 is first reduced to a value v and then the variable x is substituted with v in t_2. Note that, in the rule E-LET, the module M in which $\texttt{let}\, x = t_1 \,\texttt{in}\, t_2$ can do a step is the same module in which t_1 can do a step. In general, in rules saying that a term can do a step when one of its subterms can, the module in which the term can do a step is the same in which the subterm can. There is a single rule we will see later: E-EXEC, that let a term do a step in a module when its subterm can do a step in another module. More simply, in all rules but E-EXEC, if an evaluation judgment appears in the premises, it has the same module of the conclusion.

$$\frac{\mathrm{M} \ni t_i \ \rightarrow \ t_i'}{\mathrm{M} \ni \texttt{pack}\,\mathrm{M.S}\,[\,\overline{v}, t_i, \overline{t}\,] \ \rightarrow \ \texttt{pack}\,\mathrm{M.S}\,[\,\overline{v}, t_i', \overline{t}\,]} \ \textsc{E-pack}$$

$$\frac{k \in K \text{ is fresh}}{\mathrm{M} \ni \texttt{pack}\,\mathrm{M.S}\,[\,\overline{v}\,] \ \rightarrow \ \texttt{struct}\ \{k\}\,\mathrm{M.S}\,[\,\overline{v}\,]} \ \textsc{E-packed}$$

In $\texttt{pack}\,\mathrm{M.S}\,[\,\overline{t}\,]$, the vector of terms $\overline{t}$ is first evaluated to a vector of values $\overline{v}$ by E-PACK. After this, the vector of values is packed into a struct term by E-PACKED. The new struct is given a fresh identifier k taken from K. When we consider the execution sequence $\mathrm{M} \ni t \ \rightarrow^\star \ t'$ (see Definition 1), each time E-PACKED is executed, k is taken such that it is different from the identifiers initially present in t and from the identifiers assigned by previous E-PACKED executions. Note how E-PACK and E-PACKED can be executed only by the module M in which the struct M.S is defined.

$$\frac{\mathrm{M} \ni t_1 \;\rightarrow\; t_1'}{\mathrm{M} \ni \mathtt{unpack}\,\{\overline{x}\} = t_1 \,\mathtt{in}\, t_2 \;\rightarrow\; \mathtt{unpack}\,\{\overline{x}\} = t_1' \,\mathtt{in}\, t_2}\ \textsc{E-unpack}$$

$$\frac{\mathtt{str}\ \mathrm{S}\,\{\,b\,,\ \overline{\mathrm{T}}\,\} \in \mathrm{M} \qquad |\overline{x}| = |\overline{v}| = |\overline{\mathrm{T}}|}{\mathrm{M} \ni \mathtt{unpack}\,\{\overline{x}\} = \mathtt{struct}\ \{k\}\,\mathrm{M.S}\,[\,\overline{v}\,]\,\mathtt{in}\, t_2 \;\rightarrow\; t_2\{\overline{x} := \overline{v}\}}\ \textsc{E-unpacked}$$

The term $\mathtt{unpack}\,\{\overline{x}\} = t_1\,\mathtt{in}\,t_2$ decomposes the struct t_1 into its components. It is a binder like $\mathtt{let}\,x = t_1\,\mathtt{in}\,t_2$, but it binds multiple variables at once. All the variables $\overline{x}$ appearing in the unpack body t_2 are bounded by the unpack. The first variable of $\overline{x}$ will be substituted with the first field of the struct, the second variable with the second field, and so on. The programmer choses the names for the fields. The term to be unpacked is first evaluated to a struct value by E-UNPACK. Then, E-UNPACKED deletes somehow the struct value and substitutes in the unpack's body the names given to the struct fields ($\overline{x}$) with the value of those fields ($\overline{v}$).

$$\frac{\mathrm{M} \ni t_i \;\rightarrow\; t_i'}{\mathrm{M} \ni \mathtt{call}\,\mathrm{M}_2.\mathrm{F}\,[\,\overline{v}, t_i, \overline{t}\,] \;\rightarrow\; \mathtt{call}\,\mathrm{M}_2.\mathrm{F}\,[\,\overline{v}, t_i', \overline{t}\,]}\ \textsc{E-call}$$

$$\frac{\mathtt{fun}\ \mathrm{F}\,(\,\overline{x} : \overline{\mathrm{T}}\,) : \mathrm{T}_r\,\{t_b\} \in \mathrm{M}_2}{\mathrm{M} \ni \mathtt{call}\,\mathrm{M}_2.\mathrm{F}\,[\,\overline{v}\,] \;\rightarrow\; \mathtt{exec}\,\mathrm{M}_2\ t_b\{\overline{x} := \overline{v}\}}\ \textsc{E-called}$$

$$\frac{\mathrm{M}_2 \ni t \;\rightarrow\; t'}{\mathrm{M} \ni \mathtt{exec}\,\mathrm{M}_2\ t \;\rightarrow\; \mathtt{exec}\,\mathrm{M}_2\ t'}\ \textsc{E-exec} \qquad \frac{}{\mathrm{M} \ni \mathtt{exec}\,\mathrm{M}_2\ v \;\rightarrow\; v}\ \textsc{E-executed}$$

In E-CALL and E-CALLED we can see how a term executing in a module M, can call a function defined in any module M_2.F. When a function is called, the term $\mathtt{call}\,\mathrm{M}_2.\mathrm{F}\,[\,\overline{v}\,]$ is substituted with $\mathtt{exec}\,\mathrm{M}_2\ t_b\{\overline{x} := \overline{v}\}$, where t_b is the body of the function M_2.F. The term $\mathtt{exec}\,\mathrm{M}_2\ t_b\{\overline{x} := \overline{v}\}$ remembers the module M_2 in which the function is defined, so that the function body can be evaluated in the correct module.

The rule E-EXEC is used to do a step in the function body. When the body can do a step in the module in which the function is defined (M_2), we can do a step in the module which is calling the function (M). Finally, when the function body has become a value, E-EXECUTED is used to return the value to the caller. Note how a sequence of nested function calls results (at some point of the computation) in a term with nested `exec` terms. We can say `exec` is used to build up a stack of modules, where E-CALLED pushes a new module on top of the stack, while E-EXECUTED pops the topmost module.

$$\frac{\mathrm{M} \ni t \;\rightarrow\; t'}{\mathrm{M} \ni \mathtt{pub}\ t \;\rightarrow\; \mathtt{pub}\ t'}\ \textsc{E-publish} \qquad \frac{}{\mathrm{M} \ni \mathtt{pub}\ v \;\rightarrow\; 0}\ \textsc{E-published}$$

When a value is published, E-PUBLISHED simply removes that value from the program, returning zero. The meaning of this apparently useless rule becomes clearer when we will talk about the resource preservation theorem (RPT): Theorem 2. Values

published during the execution will be tracked by the R_U function (see Definition 5). We remind that we do not model the global state for simplicity.

$$\frac{\mathrm{M} \ni t_1 \ \rightarrow \ t_1'}{\mathrm{M} \ni \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \ \rightarrow \ \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3}\ \text{E-IF}$$

$$\frac{n \neq 0}{\mathrm{M} \ni \texttt{if}\ n\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \ \rightarrow \ t_2}\ \text{E-TRUE}$$

$$\frac{}{\mathrm{M} \ni \texttt{if}\ 0\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \ \rightarrow \ t_3}\ \text{E-FALSE}$$

The semantics if the branching term is standard. The only thing to mention is that, since we don't have a boolean type, the guard is an integer. A guard equal to zero is considered false, while a guard different from zero is considered true.

$$\frac{}{\mathrm{M} \ni \texttt{struct}\ \{k\}\,\mathrm{M.S}\,[\,\overline{v}\,].j \ \rightarrow \ v_j}\ \text{E-SELECT}$$

To conclude we have the rule E-SELECT that selects the j-th field of a struct. The value v_j is the j-th field of the vector $\overline{v} = v_1, .., v_j, .., v_n$.

A table resuming of all the operational semantics rules is given in Appendix B.

Now that we have a complete definition of the single-step evaluation judgment, we can define the multi-step evaluation judgment. The judgment $\mathrm{M} \ni t \ \rightarrow^\star \ t'$ is read: "In module M, t can do a sequence of steps and go to t'".

Definition 1 (Multi-step evaluation judgment)**.**

$$\frac{}{\mathrm{M} \ni t \ \rightarrow^\star \ t}$$

$$\frac{\mathrm{M} \ni t_1 \ \rightarrow^\star \ t_2 \qquad \mathrm{M} \ni t_2 \ \rightarrow \ t_3}{\mathrm{M} \ni t_1 \ \rightarrow^\star \ t_3}$$

Note that all the steps of a sequence are done in the same module M. With M fixed, $\rightarrow$ defines an homogeneous binary relation on the set of terms. Hence $\rightarrow^\star$ is the reflexive and transitive closure of $\rightarrow$.

2.3.1 Operational Semantics Examples

In this section we apply the operational semantics rules to show the execution of some example terms. When we need it, we will use the `modCoin` and `modSeller` modules defined in Section 2.2.1, that we rename respectively as M_C and M_S for brevity. So for example, with M_C.`Coin` we refer to the struct `Coin` defined in the module `modCoin`.

The first term we consider is:

$$\begin{aligned} \mathrm{M}_C \ni \ & \texttt{pack}\,\mathrm{M}_C.\texttt{Coin}\,[\,5\,] \qquad\qquad \text{E-PACKED} \\ \rightarrow \ & \texttt{struct}\ \{k_1\}\,\mathrm{M}_C.\texttt{Coin}\,[\,5\,] \end{aligned}$$

It is a pack term $\texttt{pack}\,\mathrm{M}_C.\texttt{Coin}\,[\,5\,]$ that creates a new `Coin` struct with an `amount` of 5. We evaluate this term in the module M_C (the module containing the definition of the `Coin` type), as highlighted at the beginning of the reduction sequence ($\mathrm{M}_C \ni ..$). Since 5 is already a value, the term evaluates to $\texttt{struct}\,\{k_1\}\,\mathrm{M}_C.\texttt{Coin}\,[\,5\,]$ according to the rule E-PACKED. The key k_1 is fresh, which in this context means that it is different from all the keys present in the starting term $\texttt{pack}\,\mathrm{M}_C.\texttt{Coin}\,[\,5\,]$ and different from all the keys generated by previous executions of the E-PACKED rule. Since there are no keys in $\texttt{pack}\,\mathrm{M}_C.\texttt{Coin}\,[\,5\,]$, and no previous executions of the E-PACKED rule, there is no restriction on the choice of k_1.

Note that this evaluation step can't be performed in a module different than M_C, since the E-PACKED requires the module in which the step is executed to be the same module in which the struct being packed is defined.

The second term we consider is:

$$
\begin{array}{rll}
\mathrm{M}_C \ni & \texttt{unpack}\,\{a\} = \texttt{pack}\,\mathrm{M}_C.\texttt{Coin}\,[\,1\,]\,\texttt{in}\,a & \text{E-UNPACK} \\
\rightarrow & \texttt{unpack}\,\{a\} = \texttt{struct}\,\{k_1\}\,\mathrm{M}_C.\texttt{Coin}\,[\,1\,]\,\texttt{in}\,a & \text{E-UNPACKED} \\
\rightarrow & 1 &
\end{array}
$$

It is an unpack term $\texttt{unpack}\,\{a\} = \texttt{pack}\,\mathrm{M}_C.\texttt{Coin}\,[\,1\,]\,\texttt{in}\,a$ that unpacks a `Coin` struct. The `Coin` struct is created in place by the term $\texttt{pack}\,\mathrm{M}_C.\texttt{Coin}\,[\,1\,]$, and then immediately unpacked. The body of the unpack is the variable a, which is bound to the value of the `amount` field of the struct. The rule E-UNPACK can be applied because the module in which the term is evaluated (M_C) is the same module in which the `Coin` struct is defined.

The next term we consider is:

$$
\begin{array}{rll}
\mathrm{M}_S \ni & \texttt{let}\,x = \texttt{struct}\,\{k_1\}\,\mathrm{M}_S.\texttt{Nft}\,[\,4, \texttt{struct}\,\{k_2\}\,\mathrm{M}_C.\texttt{Coin}\,[\,7\,]\,]\,\texttt{in}\,x.0 & \text{E-LET2} \\
\rightarrow & (\texttt{struct}\,\{k_1\}\,\mathrm{M}_S.\texttt{Nft}\,[\,4, \texttt{struct}\,\{k_2\}\,\mathrm{M}_C.\texttt{Coin}\,[\,7\,]\,]).0 & \text{E-SELECT} \\
\rightarrow & 4 &
\end{array}
$$

A value of type `Nft` is bounded to the variable x and its first field (the 0^{th} field), which corresponds to the `number` field, is selected. We will see later that this term is not well typed, because the linear value of type `Nft` is not explicitly consumed; but for now we are only interested in the evaluation of the term. The rule E-LET2 is applied because the term being bound to the variable x is a value. The rule E-SELECT is applied because the term being selected is a struct with the requested field. The rule E-SELECT can be applied because the term is evaluated in the module M_S, which is the module in which the `Nft` struct is defined.

We conclude with a function call:

$$
\begin{aligned}
\mathrm{M} \ni\ & \texttt{call}\ \mathrm{M}_S.\texttt{returnNft}\,[\,\texttt{struct}\ \{k_1\}\ \mathrm{M}_S.\texttt{Nft}\,[\,4, \texttt{struct}\ \{k_2\}\ \mathrm{M}_C.\texttt{Coin}\,[\,7\,]\,]\,] \\
& \text{E-CALLED} \\
\to\ & \texttt{exec}\ \mathrm{M}_S\ \texttt{unpack}\ \{n, c\} = \texttt{struct}\ \{k_1\}\ \mathrm{M}_S.\texttt{Nft}\,[\,4, \texttt{struct}\ \{k_2\}\ \mathrm{M}_C.\texttt{Coin}\,[\,7\,]\,]\ \texttt{in}\ \texttt{pub}\ c \\
& \text{E-EXEC} \\
\to\ & \texttt{exec}\ \mathrm{M}_S\ \texttt{pub}\ \texttt{struct}\ \{k_2\}\ \mathrm{M}_C.\texttt{Coin}\,[\,7\,] \\
& \text{E-EXEC} \\
\to\ & \texttt{exec}\ \mathrm{M}_S\ 0 \\
& \text{E-EXECUTED} \\
\to\ & 0
\end{aligned}
$$

The function $\mathrm{M}_S.\texttt{returnNft}$ is called with an `Nft` struct as argument. When the rule E-CALLED is applied, the term evolves to the body of the function in which the function parameter `nft` is substituted with the argument of the function, wrapped in a $\texttt{exec}\ \mathrm{M}_S\ \_$ term. In the body of the function, we changed the names given by the unpack to the fields of the struct, from `number` and `refound`, to n and c respectively, for brevity. The variable n will be bound to the integer of the `Nft`, while c will be bound to the `Coin` of the `Nft`.

Note that, in the second step, in module M we are able to execute the unpack of the `Nft` struct, which is defined in module M_S, thanks to the fact that the unpack is wrapped by an $\texttt{exec}\ \mathrm{M}_S\ \_$ term. The term $\texttt{exec}\ \mathrm{M}_S\ t$ tells us the provenience of the term t: the term t comes from module M_S. Since the term t comes from module M_S, it is allowed to execute an unpack of a struct defined in M_S.

The third step publishes the coin value $\texttt{struct}\ \{k_2\}\ \mathrm{M}_C.\texttt{Coin}\,[\,7\,]$ in the global state and returns 0. Next, since 0 is a value, the rule E-EXECUTED can be applied, and the term evaluates to 0.

```
M {
  str Coin { T,
   amount : Tint
  }

  fun doCopy(c : Coin) : Tint
  {
    let x = pub c In
      pub c // <- error
  }
}
```

```
module M {
  struct Coin has key {
    id : UID,
    amount : u64
  }

  fun doCopy(c : Coin, ctx: &mut TxContext)
  {
    transfer(move c, sender(ctx));
    transfer(move c, sender(ctx)); // <- error
  }
}
```

Figure 2.5: A resource copy error in FM (left) and in Move (right).

2.4 Typing

2.4.1 Introductory examples

In the following, we explain two examples of errors our types system should prevent. The first is a copy error while the second is a drop error.

In the first FM example on the left of Figure 2.5 we have the function `doCopy` that takes as argument a linear value `c` of type `Coin`. The function publishes in the global state the value associated to `c` twice (Here we used let bindings instead of the sequence operator `_;_` we introduced before, but note that the variable name `x` is arbitrary and doesn't appear in its scope). This is an error because the value of `c` is linear and it can't be copied. Publishing the same linear value twice would create two copies of the value in the global state. On the right of Figure 2.5 we have the same kind of error in Move: the programmer is trying to transfer twice the `Coin` value of `c` to the sender of the transaction, calling `transfer(move c, sender(ctx))` twice.

In the second FM example on the left of Figure 2.6 we have the function `doDrop` with a parameter of type `Coin`, which is linear. The function does nothing with the parameter and returns the value 0. This is an error because the value of `c` is linear and it can't be dropped. Within the function, the received `Coin` value must be published, explicitly deconstructed with an unpack or passed to another function. On the right of Figure 2.6 the equivalent Move error: at the end of the `doDrop` function the scope of `c` ends and its value is dropped. This is not allowed because the `Coin` type doesn't have the *drop* ability.

In the next section we will see how the FM type system prevents these errors.

2.4.2 Linear Typing overview

To understand the mechanism underlying a linear type system we think useful the following example. Let's consider the term `let` $x = t_x$ `in` t_b. In a conventional type system the variable x can be used any number of times in t_b. As a first approximation, we can say that using a variable in a term means making the variable appear

```
M {
  str Coin { T,
   amount : Tint
  }

  fun doDrop(c : Coin) : Tint {
      0
  } // drop error
}
```

```
module M {
  struct Coin has key {
    id : UID,
    amount : u64
  }

  fun doDrop(c : Coin) : u64 {
    0
  } // drop error
}
```

Figure 2.6: A resource drop error in FM (left) and in Move (right).

in the term. So, in a conventional type system, there is no restriction on the number of times a variable is used. In a type system with linear types, instead, the variable x, if linear, must be used exactly once in t_b. *The type system enforces that every linear variable is used exactly once. As a consequence, linear values at runtime aren't copied, aren't lost but are used exactly once.*

To do that, from the typing judgment of a term must be possible to deduce what are the linear variables used by the term. As an example, in a standard type system, if we know that $\Gamma \vdash t : \mathrm{T}$ is a valid judgment, we can say nothing about the variables used by t. The term t may use all the variables in Γ multiple times, or none of them. The typing judgment of a type system with linear types must be more informative: it must tell us what are the linear variables used by the term being typed. In that way, it can prevent the typing of a term like $t_1 \,\mathtt{op}\, t_2$ when a linear variable is used by both t_1 and t_2, and can prevent the typing of a term like $\mathtt{let}\, x = t_x \,\mathtt{in}\, t_b$ when x is linear and it is not used in t_b.

To formalize a linear type systems there are mainly two approaches:

Context splitting - The first approach is based on context splitting. In this kind of formalization the typing judgment has the classical form $\Gamma \vdash t : \mathrm{T}$, but it includes the information about the linear variables used by t: if $\Gamma \vdash t : \mathrm{T}$ is derivable, then all the linear variables appearing in Γ are used exactly once in t. This means that, for example, $x : \mathrm{T} \vdash 5 : \mathrm{Int}$ is not derivable when T is linear, because the term 5 doesn't use x, but it is derivable when T is normal.

The typing rule for a generic binary operator $t_1 \,\mathtt{op}\, t_2$ would be something like:

$$\frac{\Gamma_1 \vdash t_1 : \mathrm{T}_1 \qquad \Gamma_2 \vdash t_2 : \mathrm{T}_2 \qquad \Gamma_1 \oplus \Gamma_2}{\Gamma_1 \cup \Gamma_2 \vdash t_1 \,\mathtt{op}\, t_2 : \mathrm{T}_3}\ \text{T-OP}$$

The types T_1, T_2 and T_3 are not relevant. The relevant part is the relation $\oplus$ appearing in the premises, and the union of the contexts in the conclusion $\Gamma_1 \cup \Gamma_2$ (we assume contexts are sets of pairs $\{x : \mathrm{T}\}$). Two contexts Γ_1 and Γ_2 belong to the relation $\oplus$ if they satisfy the following two conditions:

- Γ_1 and Γ_2 contain the same normal variables, with the same types.
- The linear variables of Γ_1 and Γ_2 are disjoint.

The relation $\oplus$ checks the linear variables used by t_1 are not used by t_2, and vice versa. The union of the contexts in the conclusion ($\Gamma_1 \cup \Gamma_2$) collects the linear variables used by t_1 and t_2, because those are the variables used by $t_1 \,\mathtt{op}\, t_2$.

The strategy is called context splitting because, in many rules, the linear variables of the context in the conclusion are split among the contexts of the premises, like in the T-OP rule above: the linear variables of $\Gamma_1 \cup \Gamma_2$ are split between Γ_1 and Γ_2. This approach is more compact and elegant compared to the second one, but a type checking algorithm can't be easily derived from the rules [21, p. 11], and the rules are not easy to implement in Agda [1], which is what we do in Chapter 4 to have the computer verify the properties we have proved. For this reason, we followed the second approach.

Input and output contexts - The second approach is based on input and output contexts [21, p. 12] [1] [28]. The typing judgment has the form $\Gamma_{in} \vdash t : \mathrm{T} \rhd \Gamma_{out}$, where Γ_{in} and Γ_{out} are called respectively *input context* and *output context*. The difference between the two contexts tells us what are the linear variables used by t. There are different ways in which the input context can be modified in an output context to encode the variables used by t. One way is to remove from Γ_{in} the linear variable used by t to obtain Γ_{out}. Another way is to obtain Γ_{out} *marking* somehow the linear variables of Γ_{in} used by t.

In both cases, the typing rule of a generic binary operator $t_1 \,\mathtt{op}\, t_2$ would be something like:

$$\frac{\Gamma_1 \vdash t_1 : \mathrm{T}_1 \rhd \Gamma_2 \qquad \Gamma_2 \vdash t_2 : \mathrm{T}_2 \rhd \Gamma_3}{\Gamma_1 \vdash t_1 \,\mathtt{op}\, t_2 : \mathrm{T}_3 \rhd \Gamma_3}\ \text{T-OP}$$

The premises are chained together using input and output contexts. The output context Γ_2 of the first premise is used as input context for the second premise, such that the variables used by t_1 can't be used by t_2. Each subterm uses some of the linear variables of the context and leaves back what remains.

In our formalization we use input and output contexts and we mark the used linear variables.

2.4.3 FM Typing

In FM, a typing judgement has the following form:

$$\underbrace{\mathrm{M}}_{\text{Module}} \ni \underbrace{\Gamma_{in}}_{\text{Input Env.}} \vdash t : \mathrm{T} \rhd \underbrace{\Gamma_{out}}_{\text{Output Env.}}$$

M is a module name, Γ_{in} is the input context, t is the term we are typing with type T and Γ_{out} is the output context. We can read the judgment like this : "In module M, with input environment Γ_{in}, the term t has type T with output environment Γ_{out}".

Role of the module - Whether a term is well typed or not depends also on the module in which we are typing it. There are terms that are well typed in a module but not in another. For example, the term $\mathtt{pack}\,\mathrm{M.S}\,[\,\bar{t}\,]$ is well typed only in M, the

module in which the struct S is defined. This constraint enforces the fact that a struct can be created, deconstructed and inspected (selecting its fields) only in the module in which it is defined. Outside the defining module, the struct is opaque and can only be moved around. The same constraints are imposed by Move as we have seen in Section 1.4.

Context of usages - Input and output contexts are partial functions from variable names to *usages*. A usage can be thought of as a pair of a type and a *usability mark*. The *usability mark* can be in one of two states : *usable* ($\circ$) or *stale* ($\bullet$).

$$\text{Usage U} ::= \text{T}^{\circ} \mid \text{T}^{\bullet}$$

$$\Gamma ::= \varnothing \mid \Gamma, x : \text{U}$$

The usability mark of a variable in the input context tells us if the variable can be used in the term. When a variable is *usable* in the input context, it can be used in the term we are typing. If a linear variable is *usable* $\circ$ in input and it is used in the term, the same variable will be marked *stale* $\bullet$ in the output context. When a variable is *stale* $\bullet$ in input, it can't appear in the term we are typing.

The output context of a typing judgment is always a copy of the input context but with the linear variables used by the term marked as *stale* $\bullet$. This marking is performed by the T-VARL rule, as we see in Section 2.4.4, which is the only rule that actually performs a modification of the context.

2.4.4 Typing rules

$$\frac{}{\text{M} \ni \Gamma \vdash n : \text{Int} \ \rhd \ \Gamma}\ \text{T-NUM}$$

We start from the simplest rule which is the rule for typing numerical constants T-NUM. A constant uses no variable, therefore the output context is the same as the input context. Note that the input context may contain both *usable* and *stale* variables. Here some example of derivable judgments:

$$\begin{gathered}
\text{M} \ni \varnothing \vdash 5 : \text{Int} \ \rhd \ \varnothing \\
\text{M} \ni x : \text{T}^{\circ} \vdash 5 : \text{Int} \ \rhd \ x : \text{T}^{\circ} \\
\text{M} \ni x : \text{T}^{\bullet} \vdash 5 : \text{Int} \ \rhd \ x : \text{T}^{\bullet} \\
\text{M} \ni x : \text{T}^{\bullet}, y : \text{T}_2^{\circ} \vdash 5 : \text{Int} \ \rhd \ x : \text{T}^{\bullet}, y : \text{T}_2^{\circ}
\end{gathered}$$

$$\frac{\Gamma(x) = \text{T}^{\circ} \qquad \neg\text{IsLinear(T)}}{\text{M} \ni \Gamma \vdash x : \text{T} \ \rhd \ \Gamma}\ \text{T-VAR}$$

$$\frac{\Gamma(x) = \text{T}^{\circ} \qquad \text{IsLinear(T)}}{\text{M} \ni \Gamma \vdash x : \text{T} \ \rhd \ \Gamma\{x \mapsto \text{T}^{\bullet}\}}\ \text{T-VARL}$$

There are two rules that allow to type a variable. The first T-VAR types a normal variables (a variable with normal type). The variable must be *usable* $\circ$ in input,

and it stays *usable* $\circ$ in output; the environment is not changed by the rule. The second rule T-VARL types linear variables. The variable must be *usable* $\circ$ in input, and it is marked *stale* $\bullet$ in output. This is the only rule that explicitly modifies the environment.

Note that we can never say a variable, linear or not, is well typed, when the variable is *stale* $\bullet$ in input. Here some example of derivable judgments:

$$\begin{array}{rl}
\mathrm{M} \ni x : \mathrm{T}^{\circ} \vdash x : \mathrm{T} \rhd x : \mathrm{T}^{\circ} & \texttt{when } \mathrm{IsLinear(T)} \\
\mathrm{M} \ni x : \mathrm{T}^{\circ}, y : \mathrm{T}_2^{\circ} \vdash x : \mathrm{T} \rhd x : \mathrm{T}^{\circ}, y : \mathrm{T}_2^{\circ} & \texttt{when } \mathrm{IsLinear(T)} \\
\mathrm{M} \ni x : \mathrm{T}^{\circ} \vdash x : \mathrm{T} \rhd x : \mathrm{T}^{\bullet} & \texttt{when } \neg\mathrm{IsLinear(T)} \\
\mathrm{M} \ni x : \mathrm{T}^{\circ}, y : \mathrm{T}_2^{\bullet} \vdash x : \mathrm{T} \rhd x : \mathrm{T}^{\bullet}, y : \mathrm{T}_2^{\bullet} & \texttt{when } \neg\mathrm{IsLinear(T)}
\end{array}$$

$$\frac{\mathrm{M} \ni \Gamma_1 \vdash t_1 : \mathrm{T}_1 \rhd \Gamma_2 \qquad \mathrm{M} \ni \Gamma_2, x : \mathrm{T}_1^{\circ} \vdash t_2 : \mathrm{T}_2 \rhd \Gamma_3, x : \mathrm{T}_1^{\downarrow}}{\mathrm{M} \ni \Gamma_1 \vdash \texttt{let}\, x = t_1 \,\texttt{in}\, t_2 : \mathrm{T}_2 \rhd \Gamma_3} \text{ T-LET}$$

T-LET is the first rule we see that gives a type to a term ($\texttt{let}\, x = t_1 \,\texttt{in}\, t_2$) with multiple subterms (t_1 and t_2). The judgments in the premises are chained as mentioned in the introductory Section 2.4.2: the output context Γ_2 of the subterm t_1 is used as input context to type t_2. If a linear variable was used by t_1, it would be marked *stale* $\bullet$ in Γ_2, and so the same variable couldn't be used in t_2.

The let binding introduces a new variable x with scope t_2. If the type T_1 of the variable is linear, the rule checks the variable is used exactly once in t_2. To do so, the rule asks the body t_2 to be well typed with an input context Γ_2 expanded with $x : \mathrm{T}_1^{\circ}$, and with an output context containing $x : \mathrm{T}_1^{\downarrow}$. The symbol $^{\downarrow}$ is a function (from Type to Usage) that marks a type T as *usable* (T°) if the type is normal, and *stale* ($\mathrm{T}^{\bullet}$) if the type is linear:

Definition 2.

$$\begin{array}{rcll}
\text{mark} & : & \text{Type} \to \text{Usage} & \\
\mathrm{T}^{\downarrow} & = & \mathrm{T}^{\circ} & \text{when } \neg\mathrm{IsLinear(T)} \\
\mathrm{T}^{\downarrow} & = & \mathrm{T}^{\bullet} & \text{when } \mathrm{IsLinear(T)}
\end{array}$$

- When T_1 is linear, the typing judgement about t_2 in the premises of the rule assumes the form:

  $$\mathrm{M} \ni \Gamma_2, x : \mathrm{T}_1^{\circ} \vdash t_2 : \mathrm{T}_1 \rhd \Gamma_3, x : \mathrm{T}_1^{\bullet}$$

  The rule is asking the variable x to be used exactly once in t_2, since that's the only case in which a judgment with this form can be derived.
- When T_1 is normal, the typing judgement about t_2 assumes the form:

  $$\mathrm{M} \ni \Gamma_2, x : \mathrm{T}_1^{\circ} \vdash t_2 : \mathrm{T}_1 \rhd \Gamma_3, x : \mathrm{T}_1^{\circ}$$

  Since the *usability mark* of a normal variables is never changed by rules, the rule is asking the usability mark of x to be coherent, that is: if x is usable in input, x stays usable in output.

$$\frac{\texttt{str}\,\mathrm{S}\,\{\,b\,,\ \overline{\mathrm{T}}\,\}\in \mathrm{M} \qquad \mathrm{M}\ni\Gamma_1\vdash \bar{t}:\overline{\mathrm{T}}\ \rhd\ \Gamma_2}{\mathrm{M}\ni\Gamma_1\vdash \texttt{pack}\,\mathrm{M.S}\,[\,\bar{t}\,]:\mathrm{M.S}\ \rhd\ \Gamma_2}\ \textsc{T-pack}$$

$$\frac{\texttt{fun}\,\mathrm{F}\,(\,\overline{x}:\overline{\mathrm{T}}\,):\mathrm{T}_r\,\{t_b\}\in \mathrm{M}_2 \qquad \mathrm{M}\ni\Gamma_1\vdash \bar{t}:\overline{\mathrm{T}}\ \rhd\ \Gamma_2}{\mathrm{M}\ni\Gamma_1\vdash \texttt{call}\,\mathrm{M}_2\mathrm{.F}\,[\,\bar{t}\,]:\mathrm{T}_r\ \rhd\ \Gamma_2}\ \textsc{T-call}$$

$$\frac{}{\mathrm{M}\ni\Gamma\vdash \overline{0}_t:\overline{0}_T\ \rhd\ \Gamma}\ \textsc{T-vecz}$$

$$\frac{\mathrm{M}\ni\Gamma_1\vdash \bar{t}:\overline{\mathrm{T}}\ \rhd\ \Gamma_2 \qquad \mathrm{M}\ni\Gamma_2\vdash t':\mathrm{T}'\ \rhd\ \Gamma_3}{\mathrm{M}\ni\Gamma_1\vdash \bar{t},t':\overline{\mathrm{T}},\mathrm{T}'\ \rhd\ \Gamma_3}\ \textsc{T-vec}$$

T-PACK and T-CALL are similar because in both the rules a vector of terms $\bar{t}$ is typed in the premises. *While a term t has a type* T*, a vector of terms $\bar{t}$ has a vector of types* $\overline{\mathrm{T}}$. In T-PACK the vector of terms must have a vector of types that matches the types of the struct fields (see $\overline{\mathrm{T}}$ in T-PACK). In T-CALL the vector of terms must have a vector of types that matches the types of the function parameters (see $\overline{\mathrm{T}}$ in T-CALL).

Note that in T-CALL, the module M_2 in which the function F is defined can be different from the module M in which $\texttt{call}\,\mathrm{M}_2\mathrm{.F}\,[\,\bar{t}\,]$ is typed. Any function can be called from any module. Contrary, in T-PACK the module M in which the struct S is defined must be the same module in which $\texttt{pack}\,\mathrm{M.S}\,[\,\bar{t}\,]$ is typed. A struct can be packed only in the module in which it is defined.

The typing judgment for a vector of terms is constructed using T-VECZ and T-VEC. Using T-VECZ we can say $\overline{0}_t$ (the empty vector of terms) has types $\overline{0}_T$ (the empty vector of types) when input and output contexts are the same. Using T-VEC, if we have a well typed vector of terms $\bar{t}$ with types $\overline{\mathrm{T}}$ and using its output context Γ_2 we can type another term t' with type T', then we can type the vector $[\,\bar{t},t'\,]$ with types $[\,\overline{\mathrm{T}},\mathrm{T}'\,]$, using as input the input context of $\bar{t}$, and as output the output context of t'. The type judgment for a vector is essentially a chain of type judgments. The output context of the previous term is used as input context for the next term. As a consequence, a linear variable can be used at most by one of the terms in the vector.

$$\frac{\begin{array}{c}\texttt{str}\,\mathrm{S}\,\{\,b\,,\ \overline{\mathrm{T}}\,\}\in \mathrm{M}\\ \mathrm{M}\ni\Gamma_1\vdash t_1:\mathrm{M.S}\ \rhd\ \Gamma_2 \qquad \mathrm{M}\ni\Gamma_2,\overline{x}:\overline{\mathrm{T}}^{\circ}\vdash t_2:\mathrm{T}_2\ \rhd\ \Gamma_3,\overline{x}:\overline{\mathrm{T}}^{\downarrow}\end{array}}{\mathrm{M}\ni\Gamma_1\vdash \texttt{unpack}\,\{\overline{x}\}=t_1\,\texttt{in}\,t_2:\mathrm{T}_2\ \rhd\ \Gamma_3}\ \textsc{T-unpack}$$

The rule T-UNPACK is similar to the rule T-LET. $\texttt{unpack}\,\{\overline{x}\}=t_1\,\texttt{in}\,t_2$ has two subterms t_1 and t_2 that are typed in chain in the premises. The term t_1 must be a struct either linear or not. The body t_2 must be well typed in the output context of t_1 (that is Γ_2) expanded with one *usable* $\circ$ variable for each field of the struct ($\overline{x}:\overline{\mathrm{T}}^{\circ}$ stands for $x_1:\mathrm{T}_1^{\circ},..,x_n:\mathrm{T}_n^{\circ}$). All the new variables with linear type must appear *stale* $\bullet$ in output ($\overline{x}:\overline{\mathrm{T}}^{\downarrow}$ stands for $x_1:\mathrm{T}_1^{\downarrow},..,x_n:\mathrm{T}_n^{\downarrow}$). As an example, when the struct being unpacked has two fields, the first normal and the second linear, we have ($\overline{\mathrm{T}}=\mathrm{T}_1,\mathrm{T}_2$), ($\overline{x}:\overline{\mathrm{T}}^{\circ}=x_1:\mathrm{T}_1^{\circ},x_2:\mathrm{T}_2^{\circ}$) and ($\overline{x}:\overline{\mathrm{T}}^{\downarrow}=x_1:\mathrm{T}_1^{\circ},x_2:\mathrm{T}_2^{\bullet}$).

$$\frac{\mathrm{M} \ni \Gamma_1 \vdash t_1 : \mathrm{Int} \ \rhd \ \Gamma_2 \qquad \begin{array}{c} \mathrm{M} \ni \Gamma_2 \vdash t_2 : \mathrm{T}_b \ \rhd \ \Gamma_3 \\ \mathrm{M} \ni \Gamma_2 \vdash t_3 : \mathrm{T}_b \ \rhd \ \Gamma_3 \end{array}}{\mathrm{M} \ni \Gamma_1 \vdash \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : \mathrm{T}_b \ \rhd \ \Gamma_3}\ \text{T-IF}$$

In T-IF the true branch and the false branch are typed with the same input context Γ_2; they must have the same type T_b, and the same output context Γ_3. Requiring both branches to have the same output context means requiring both branches to use the same variables from the input context. If $z \in \Gamma_2$ and z is used by t_2, then z must be used by t_3, and vice versa. Note that this does not prevent one branch from creating and publishing new resources without the other branch having to do the same.

$$\frac{\mathrm{M} \ni \Gamma_1 \vdash t : \mathrm{T} \ \rhd \ \Gamma_2 \qquad \mathrm{IsLinear}(\mathrm{T})}{\mathrm{M} \ni \Gamma_1 \vdash \mathtt{pub}\ t : \mathrm{Int} \ \rhd \ \Gamma_2}\ \text{T-PUB}$$

The global state of our simplified blockchain model is made of resources only, so in T-PUB we require the term to be published (t) to be linear. The publish term has always type Int because it always evaluates to 0.

$$\frac{\mathrm{M} \ni \Gamma \vdash x : \mathrm{M.S} \ \rhd \ \Gamma_2 \qquad \begin{array}{c} \mathtt{str}\ \mathrm{S}\,\{\, b\,,\ \overline{\mathrm{T}}\,\} \in \mathrm{M} \\ \neg\mathrm{IsLinear}(\mathrm{T}_j) \end{array}}{\mathrm{M} \ni \Gamma \vdash x.j : \mathrm{T}_j \ \rhd \ \Gamma}\ \text{T-SELECTX}$$

The rule T-SELECTX permits to select a filed of a struct only if the type (T_j) of the field is normal. Since selecting a field is equivalent to making a copy of the field, a linear field can't be selected. A normal field can be selected from both a linear and a normal struct.

The rule T-SELECTX is somehow special. It says $x.j$ doesn't use linear variables, even in the case x is linear. The output context Γ_2 of the premise, where x may appear *stale* $\bullet$, is not used in the conclusion. The input and output contexts of the conclusion are both Γ. The variable x appearing in the term $x.j$ is not considered a use of x.

$$\frac{\mathtt{str}\ \mathrm{S}\,\{\, b\,,\ \overline{\mathrm{T}}\,\} \in \mathrm{M}_2 \qquad \mathrm{M} \ni \Gamma \vdash \overline{v} : \overline{\mathrm{T}} \ \rhd \ \Gamma \qquad k \in K}{\mathrm{M} \ni \Gamma \vdash \mathtt{struct}\ \{k\}\,\mathrm{M}_2.\mathrm{S}\,[\,\overline{v}\,] : \mathrm{M}_2.\mathrm{S} \ \rhd \ \Gamma}\ \text{T-STRUCT}$$

$$\frac{\mathrm{M} \ni \Gamma \vdash v : \mathrm{M.S} \ \rhd \ \Gamma \qquad \begin{array}{c} \mathtt{str}\ \mathrm{S}\,\{\, b\,,\ \overline{\mathrm{T}}\,\} \in \mathrm{M} \\ \neg\mathrm{IsLinear}(\mathrm{T}_j) \end{array}}{\mathrm{M} \ni \Gamma \vdash v.j : \mathrm{T}_j \ \rhd \ \Gamma}\ \text{T-SELECTV}$$

$$\frac{\mathrm{M}_2 \ni \varnothing \vdash t : \mathrm{T} \ \rhd \ \varnothing}{\mathrm{M} \ni \Gamma \vdash \mathtt{exec}\,\mathrm{M}_2\ t : \mathrm{T} \ \rhd \ \Gamma}\ \text{T-EXEC}$$

What we have seen so far are the rules for terms that can be used by the programmer to write a program. We have also typing rules for terms that appear only at runtime:

the terms underlined in Figure 2.2. Those rules are needed to prove the Lemma 6 (Type preservation).

While $\texttt{pack}\,\mathrm{M.S}\,[\,\overline{t}\,]$ can be typed only in the module M, $\texttt{struct}\,\{k\}\,\mathrm{M}_2\mathrm{.S}\,[\,\overline{v}\,]$ can be typed in any module. This means a struct can be created only inside its module, but it can be moved around inside other modules. Note that the input and output contexts both in the premise and in the conclusion of T-STRUCT are equal to Γ. In the premises of T-STRUCT there is a typing judgment about a vector of values $\overline{v}$. Variables don't appear in values, so the output context of a derivable judgment about a value (or a vector of values) is always equal to the input context.

T-SELECTV is identical to T-SELECTX, except for the variable x that is replaced by the value v and for the context Γ_2 which is replaced by Γ (for the reason just explained).

With T-EXEC we can type $\texttt{exec}\,\mathrm{M}_2\ t$, which stands for the execution of t in M_2, with type T and with the same input and output context Γ, when t has type T in M_2 with empty input and output contexts. The term t must be closed. When we use this rule in Lemma 6 (Type preservation), t is the body of a function defined in M_2.

2.4.5 Well Formation

$$\frac{\forall\ \mathrm{T} \in \overline{\mathrm{T}}.\ \neg\mathrm{IsLinear(T)}}{\texttt{str}\ \mathrm{S}\,\{\,\bot\,,\ \overline{\mathrm{T}}\,\}}\ \text{W-NORM} \qquad \frac{}{\texttt{str}\ \mathrm{S}\,\{\,\top\,,\ \overline{\mathrm{T}}\,\}}\ \text{W-LIN}$$

$$\frac{\mathrm{M} \ni \varnothing, \overline{x} : \overline{\mathrm{T}}^{\circ} \vdash t_b : \mathrm{T}_r\ \rhd\ \varnothing, \overline{x} : \overline{\mathrm{T}}^{\downarrow}}{\mathrm{M} \vdash \texttt{fun}\ \mathrm{F}\,(\,\overline{x} : \overline{\mathrm{T}}\,) : \mathrm{T}_r\,\{t_b\}}\ \text{W-FUN}$$

$$\frac{\overline{\mathrm{M} \vdash \mathrm{FD}} \qquad \overline{\mathrm{SD}}}{\vdash \mathrm{M}\,\{\,\overline{\mathrm{SD}}\,,\ \overline{\mathrm{FD}}\,\}}\ \text{W-MODULE}$$

Well formation rules state when a program, that is a set of modules, is well formed. In our system, the *system codebase* together with the *transaction module* is a set of modules that must be well formed for the transaction to be accepted for execution.

A definition of a normal struct ($\texttt{str}\ \mathrm{S}\,\{\,\bot\,,\ \overline{\mathrm{T}}\,\}$) is well formed if all its fields are normal. An asset (a linear value) can't be embedded inside a normal value because an asset can't be copied and dropped, while a normal value can. If we copied a normal value with an asset inside, we would copy also the asset. In the same way, if we dropped a normal value with an asset inside, we would drop also the asset. On the other hand, a linear struct can contain both linear and normal fields.

A function definition is well formed if the function body t_b is well typed in the module defining the function. The body must be well typed in a context containing only the function parameters marked *usable* $\circ$ in input and marked $^{\downarrow}$ in output according to the definition of the mark function (Definition 2). The linear parameters of a function must be used in the function body.

A module is well formed if are so all its struct and function definitions. Finally, a set of modules (like a program) is well formed if all its modules are well formed.

2.4.6 Typing Examples

We see now some examples of well typed and ill typed terms. We can start showing the two functions we have seen in the introductory section (Section 2.4.1) are ill typed. To save space, we will use T_c to denote the type M.`Coin`. According to the definition, a function `fun` $\mathrm{F}\,(\,\overline{x}\,:\,\overline{\mathrm{T}}\,)\,:\,\mathrm{T}_r\,\{t_b\}$ defined in module M is well formed when the following judgement is derivable:

$$\mathrm{M} \ni \varnothing, \overline{x} : \overline{\mathrm{T}}^{\circ} \vdash t_b : \mathrm{T}_r \ \rhd\ \varnothing, \overline{x} : \overline{\mathrm{T}}^{\downarrow}$$

To prove the `doCopy` function is well typed, we have to prove:

$$\mathrm{M} \ni c : \mathrm{T}_c^{\circ} \vdash \texttt{let}\, x = \texttt{pub}\, c\, \texttt{in}\, \texttt{pub}\, c : \mathrm{Int} \ \rhd\ c : \mathrm{T}_c^{\bullet}$$

The first premise of the rule T-LET asks us to prove $\mathrm{M} \ni c : \mathrm{T}_c^{\circ} \vdash \texttt{pub}\, c : \mathrm{T} \ \rhd\ \Gamma_2$ for some type T and context Γ_2. To obtain a judgement of that form we can only use the rule T-PUB, which poses the constraint $\mathrm{T} = \mathrm{Int}$. In turn, For the premise of T-PUB we have to prove $\mathrm{M} \ni c : \mathrm{T}_c^{\circ} \vdash c : \mathrm{T}_2 \ \rhd\ \Gamma_2$ for some linear type T_2. We apply the rule T-VARL, which is the only rule that can be applied in this case, and we obtain $\mathrm{T}_2 = \mathrm{T}_c$ and $\Gamma_2 = c : \mathrm{T}_c^{\bullet}$. So, for the first premise of the rule T-LET form which we started, we have derived $\mathrm{M} \ni c : \mathrm{T}_c^{\circ} \vdash \texttt{pub}\, c : \mathrm{Int} \ \rhd\ c : \mathrm{T}_c^{\bullet}$, and no other compatible judgement can be derived.

Now that we have an output context for the first premise, we can proceed to the second premise of the rule T-LET. For the second premise we have to prove $\mathrm{M} \ni c : \mathrm{T}_c^{\bullet}, x : \mathrm{Int}^{\circ} \vdash \texttt{pub}\, c : \mathrm{T}_3 \ \rhd\ c : \mathrm{T}_c^{\bullet}, x : \mathrm{Int}^{\circ}$ for some type T_3. We can only use the rule T-PUB, which poses the constraint $\mathrm{T}_3 = \mathrm{Int}$. For the premise of T-PUB we have to prove $\mathrm{M} \ni c : \mathrm{T}_c^{\bullet}, x : \mathrm{Int}^{\circ} \vdash c : \mathrm{T}_4 \ \rhd\ c : \mathrm{T}_c^{\bullet}, x : \mathrm{Int}^{\circ}$ for some linear type T_4. This judgement can't be derived, neither by using the rule T-VARL nor the rule T-VAR, because the variable c, to which we have to assign a type, appears *stale* $\bullet$ in the input context.

Follows the derivation tree. We use the symbol $\times$ in the premises of a judgement to indicate it is impossible to derive the judgement.

$$\dfrac{\dfrac{\dfrac{\mathrm{IsLinear}(\mathrm{T}_c)}{\mathrm{M} \ni c : \mathrm{T}_c^{\circ} \vdash c : \mathrm{T}_c \ \rhd\ c : \mathrm{T}_c^{\bullet}}\,\text{T-VARL}}{\mathrm{M} \ni c : \mathrm{T}_c^{\circ} \vdash \texttt{pub}\, c : \mathrm{Int} \ \rhd\ c : \mathrm{T}_c^{\bullet}}\,\text{T-PUB} \qquad \dfrac{\dfrac{\times}{\mathrm{M} \ni c : \mathrm{T}_c^{\bullet}, x : \mathrm{Int}^{\circ} \vdash c : \mathrm{T}_4 \ \rhd\ c : \mathrm{T}_c^{\bullet}, x : \mathrm{Int}^{\circ}}\,\text{T-PUB}}{\mathrm{M} \ni c : \mathrm{T}_c^{\bullet}, x : \mathrm{Int}^{\circ} \vdash \texttt{pub}\, c : \mathrm{Int} \ \rhd\ c : \mathrm{T}_c^{\bullet}, x : \mathrm{Int}^{\circ}}\,\text{T-PUB}}{\mathrm{M} \ni c : \mathrm{T}_c^{\circ} \vdash \texttt{let}\, x = \texttt{pub}\, c\, \texttt{in}\, \texttt{pub}\, c : \mathrm{Int} \ \rhd\ c : \mathrm{T}_c^{\bullet}}\,\text{T-LET}$$

Note that we can also prove the function is ill formed with another derivation tree, where we start by typing the second premise of the T-LET rule, and then we type the first premise.

$$\dfrac{\dfrac{\dfrac{\times}{\mathrm{M} \ni c : \mathrm{T}_c^{\circ} \vdash c : \mathrm{T}_c \ \rhd\ c : \mathrm{T}_c^{\circ}}}{\mathrm{M} \ni c : \mathrm{T}_c^{\circ} \vdash \texttt{pub}\, c : \mathrm{Int} \ \rhd\ c : \mathrm{T}_c^{\circ}}\,\text{T-PUB} \qquad \dfrac{\dfrac{\mathrm{IsLinear}(\mathrm{T}_c)}{\mathrm{M} \ni c : \mathrm{T}_c^{\circ}, x : \mathrm{Int}^{\circ} \vdash c : \mathrm{T}_4 \ \rhd\ c : \mathrm{T}_c^{\bullet}, x : \mathrm{Int}^{\circ}}\,\text{T-VARL}}{\mathrm{M} \ni c : \mathrm{T}_c^{\circ}, x : \mathrm{Int}^{\circ} \vdash \texttt{pub}\, c : \mathrm{Int} \ \rhd\ c : \mathrm{T}_c^{\bullet}, x : \mathrm{Int}^{\circ}}\,\text{T-PUB}}{\mathrm{M} \ni c : \mathrm{T}_c^{\circ} \vdash \texttt{let}\, x = \texttt{pub}\, c\, \texttt{in}\, \texttt{pub}\, c : \mathrm{Int} \ \rhd\ c : \mathrm{T}_c^{\bullet}}\,\text{T-LET}$$

To prove the `doDrop` function is well typed, we have to prove:

$$\mathrm{M} \ni c : \mathrm{T}_c^\circ \vdash 0 : \mathrm{Int} \ \rhd \ c : \mathrm{T}_c^\bullet$$

It is immediate to see this judgment can't be derived: the only rule that can type the term 0 is T-INT, and T-INT requires the input and output contexts to be equal.

$$\frac{\times}{\mathrm{M} \ni c : \mathrm{T}_c^\circ \vdash 0 : \mathrm{Int} \ \rhd \ c : \mathrm{T}_c^\bullet}$$

Consider the following function:

```
fun branchIll(g : Tint, c : Coin) : Tint {
  if g then
    0
  else
    pub c
}
```

We expect the function to be ill typed (in any module M), because when the true branch is taken, the linear value `c` is not consumed. When a function receives a linear value, it must consume it in all possible execution paths. The following derivation tree shows the function is ill typed. To save space, we write Γ_1 for $g : \mathrm{Int}^\circ, c : \mathrm{T}_c^\circ$ and Γ_3 for $g : \mathrm{Int}^\circ, c : \mathrm{T}_c^\bullet$.

$$\frac{\dfrac{\neg\mathrm{IsLinear}(\mathrm{Int})}{\mathrm{M} \ni \Gamma_1 \vdash g : \mathrm{Int} \ \rhd \ \Gamma_1}\ \text{T-VAR} \qquad \dfrac{\times}{\mathrm{M} \ni \Gamma_1 \vdash 0 : \mathrm{Int} \ \rhd \ \Gamma_3} \qquad \ldots}{\mathrm{M} \ni \Gamma_1 \vdash \texttt{if}\ g\ \texttt{then}\ 0\ \texttt{else pub}\ c : \mathrm{Int} \ \rhd \ \Gamma_3}\ \text{T-IF}$$

The ellipsis in the derivation tree indicates the premises of the rule T-IF we have not shown. The guard of the branch is a normal variable, so its typing gives an output context equal to the input context Γ_1. Using Γ_1 as input context, we should be able to derive a typing judgment for the true branch, where the output context is Γ_3. This is impossible since $\Gamma_1 \neq \Gamma_3$, the true branch is the constant 0, and the only rule that can type the constant 0 is T-INT, which requires the input and output contexts to be equal. There is no need to check if we are able to type the false branch, because the true branch can't be typed.

We can fix the function `branchIll` to make it well formed. We explicitly deconstruct the linear value `c` in the true branch.

```
fun branchOk(g : Tint, c : Coin) : Tint {
  if g then
    unpack {a} = c in 0
  else
    pub c
}
```

Follows the derivation tree. For space reasons, the derivation tree is divided in different named parts. The name of the derivation tree, like DT_1, is written of the left of the root of the tree.

$$\mathrm{DT}_1 = \frac{\neg\mathrm{IsLinear}(\mathrm{Int})}{\mathrm{M} \ni \Gamma_1 \vdash g : \mathrm{Int} \ \rhd \ \Gamma_1}\ \text{T-VAR}$$

$$\mathrm{DT}_2 = \frac{\dfrac{\mathrm{IsLinear}(\mathrm{T}_c)}{\mathrm{M} \ni \Gamma_1 \vdash c : \mathrm{T}_c \ \rhd \ \Gamma_3}\ \text{T-VARL} \qquad \mathrm{M} \ni \Gamma_3, a : \mathrm{Int}^\circ \vdash 0 : \mathrm{Int} \ \rhd \ \Gamma_3, a : \mathrm{Int}^\circ}{\mathrm{M} \ni \Gamma_1 \vdash \mathtt{unpack}\,\{a\} = c\,\mathtt{in}\,0 : \mathrm{Int} \ \rhd \ \Gamma_3}\ \text{T-UNPACK}$$

$$\mathrm{DT}_3 = \frac{\dfrac{\mathrm{IsLinear}(\mathrm{T}_c)}{\mathrm{M} \ni \Gamma_1 \vdash c : \mathrm{T}_c \ \rhd \ \Gamma_3}\ \text{T-VARL}}{\mathrm{M} \ni \Gamma_1 \vdash \mathtt{pub}\ c : \mathrm{Int} \ \rhd \ \Gamma_3}\ \text{T-PUB}$$

$$\frac{\mathrm{DT}_1 \qquad \mathrm{DT}_2 \qquad \mathrm{DT}_3}{\mathrm{M} \ni \Gamma_1 \vdash \mathtt{if}\ g\ \mathtt{then}\ (\mathtt{unpack}\,\{a\} = c\,\mathtt{in}\,0)\ \mathtt{else}\ \mathtt{pub}\ c : \mathrm{Int} \ \rhd \ \Gamma_3}\ \text{T-IF}$$

We conclude showing that the `returnNft` function defined in the `modSeller` module (see Listing 4 and Listing 5) is well formed. We rewrite the function here changing some variable names for convenience. In addition, to save space in the derivation tree, we will write T_n for `modSeller.Nft`, T_c for `modCoin.Coin`, M_S for `modSeller` and M_C for `modCoin`.

The rewritten function:

```
fun returnNft(w: Nft) : Tint {
  unpack { x, r } = w In
    pub r
}
```

The derivation tree proving `returnNft` is well formed:

$$\mathrm{DT}_4 = \frac{\mathrm{IsLinear}(\mathrm{T}_n)}{\mathrm{M}_S \ni w : \mathrm{T}_n^\circ \vdash n : \mathrm{T}_n \ \rhd \ w : \mathrm{T}_n^\bullet}\ \text{T-VARL}$$

$$\mathrm{DT}_5 = \frac{\dfrac{\mathrm{IsLinear}(\mathrm{T}_c)}{\mathrm{M}_S \ni w : \mathrm{T}_n^\bullet, x : \mathrm{Int}^\circ, r : \mathrm{T}_c^\circ \vdash r : \mathrm{T}_c \ \rhd \ w : \mathrm{T}_n^\bullet, x : \mathrm{Int}^\circ, r : \mathrm{T}_c^\bullet}\ \text{T-VARL}}{\mathrm{M}_S \ni w : \mathrm{T}_n^\bullet, x : \mathrm{Int}^\circ, r : \mathrm{T}_c^\circ \vdash \mathtt{pub}\ r : \mathrm{Int} \ \rhd \ w : \mathrm{T}_n^\bullet, x : \mathrm{Int}^\circ, r : \mathrm{T}_c^\bullet}\ \text{T-PUB}$$

$$\frac{\mathrm{DT}_4 \qquad \mathrm{DT}_5}{\mathrm{M}_S \ni w : \mathrm{T}_n^\circ \vdash \mathtt{unpack}\,\{x, r\} = w\,\mathtt{in}\,\mathtt{pub}\ r : \mathrm{Int} \ \rhd \ w : \mathrm{T}_n^\bullet}\ \text{T-UNPACK}$$

Chapter 3

FM Properties

3.1 Basic properties

Before speaking about peculiar properties of the language, we have to prove some basic properties (the standard properties expected from a well-formalized language). The final goal of this section is to prove a closed and well-typed term can't evolve into a stuck term at runtime.

Lemma 1 (Simple facts)**.** *If* $\mathrm{M} \ni \Gamma_1 \vdash t : \mathrm{T} \rhd \Gamma_2$ *then:*

1. $dom(\Gamma_1) = dom(\Gamma_2)$
2. $type(\Gamma_1(x)) = type(\Gamma_2(x))$
3. $\Gamma_1(x) = \mathrm{T}^\bullet \implies \Gamma_2(x) = \mathrm{T}^\bullet$
4. $\Gamma_2(x) = \mathrm{T}^\circ \implies \Gamma_1(x) = \mathrm{T}^\circ$

With $dom(\Gamma)$ we denote the set of variables in the context Γ. With $type(\mathrm{U})$ we denote the type of the Usage U; for example, T° is a Usage whose type is T. The properties 1) and 2) tell us the only thing that may change in input and output contexts is the usage marker of variables. In the output context there are always the same variables of the input context, and the type of each variable is the same. The properties 3) and 4) show there are constraints on the way the usage marker of a variable can change. If a variable is *stale* $\bullet$ in input, it must be *stale* $\bullet$ in output too. If a variable is *usable* $\circ$ in output, then it certainly was also *usable* $\circ$ in input[1].

Lemma 2 (Value type)**.** *If* $\mathrm{M} \ni \Gamma_1 \vdash v : \mathrm{T} \rhd \Gamma_2$ *then* $\Gamma_1 = \Gamma_2$ *and* $\forall\ \mathrm{M}_2$ *and* $\forall\ \Delta$ *the following judgement is derivable:*

$$\mathrm{M}_2 \ni \Delta \vdash v : \mathrm{T}_v \rhd \Delta$$

With this lemma we can change the module and the context in which we type a value. Note that in the new derived judgment, there is the same context in input and output. A value has no free variables and therefore does not modify the context.

[1] It would be possible to define a partial order relation on contexts and prove the output context is always "greater" than the input context.

Lemma 3 (Weakening)**.** *Given a* Usage U *and a variable* x *s.t.* $x \notin dom(\Gamma_1)$*, if* $\mathrm{M} \ni \Gamma_1 \vdash t : \mathrm{T} \rhd \Gamma_2$ *then:*

$$\mathrm{M} \ni \Gamma_1, x : \mathrm{U} \vdash t : \mathrm{T} \rhd \Gamma_2, x : \mathrm{U}$$

The typing judgment is preserved when we add a new variable to the input and output contexts as long as the type and the usage marker of the new variable is the same in input and output.

Lemma 4 (Strengthening)**.** *If* $\mathrm{M} \ni \Gamma_1, x : \mathrm{U}_1 \vdash t : \mathrm{T} \rhd \Gamma_2, x : \mathrm{U}_2$*, and* $x \notin fv(t)$*, then:*

$$\mathrm{M} \ni \Gamma_1 \vdash t : \mathrm{T} \rhd \Gamma_2$$

Note that in the hypothesis of the Strengthening lemma, the usage marker of x in input and output can be different. This makes the lemma more general, but actually, when x is not free in t, the usage marker of x in input and output is always the same. $x \in fv(t)$ is a necessary condition (but not sufficient) for the usage marker of x to change.

Lemma 5 (Substitution)**.** *Given* $\mathrm{M}_v \ni \Delta_1 \vdash v : \mathrm{T}_v \rhd \Delta_2$*, the following two properties hold:*

1. *If* $\mathrm{M} \ni \Gamma_1, x : \mathrm{U} \vdash t : \mathrm{T} \rhd \Gamma_2, x : \mathrm{U}$ *with* $\mathrm{U} = \mathrm{T}_v^{\circ}$ *or* $\mathrm{U} = \mathrm{T}_v^{\bullet}$
   then $\mathrm{M} \ni \Gamma_1, x : \mathrm{U} \vdash t\{x := v\} : \mathrm{T} \rhd \Gamma_2, x : \mathrm{U}$
2. *If* $\mathrm{M} \ni \Gamma_1, x : \mathrm{T}_v^{\circ} \vdash t : \mathrm{T} \rhd \Gamma_2, x : \mathrm{T}_v^{\bullet}$
   then $\mathrm{M} \ni \Gamma_1, x : \mathrm{T}_v^{\bullet} \vdash t\{x := v\} : \mathrm{T} \rhd \Gamma_2, x : \mathrm{T}_v^{\bullet}$

The substitution lemma is probably the most interesting in this section. Note that every valid typing judgment, where x belongs to the context, falls in one of the two cases. The are two ways in which x can be marked in input ($\circ$ or $\bullet$), and two ways in which it can be marked in output, for a total of four cases. Two cases are covered by 1), and another case is covered by 2). The remaining case, where x is *stale* ($\bullet$) in input and *usable* ($\circ$) in output is impossible, as stated by Lemma 1 (Simple facts 3).

1) can be applied when x is normal, when x is not free in t or when x is linear but it is not used in t (e.g. when t is a selection). 2) can be applied when x is linear and it is used in t.

Lemma 6 (Type preservation)**.** *If* $\mathrm{M} \ni \Gamma_1 \vdash t : \mathrm{T} \rhd \Gamma_2$ *and* $\mathrm{M} \ni t \to t'$ *then:*

$$\mathrm{M} \ni \Gamma_1 \vdash t' : \mathrm{T} \rhd \Gamma_2$$

Lemma 7 (Progress)**.** *If* $\mathrm{M} \ni \varnothing \vdash t : \mathrm{T} \rhd \varnothing$ *then, either* t *is a value or there exists a term* t' *such that* $\mathrm{M} \ni t \to t'$*.*

Theorem 1 (No Stuck)**.** *If* $\mathrm{M} \ni \varnothing \vdash t : \mathrm{T} \rhd \varnothing$ *and* $\mathrm{M} \ni t \to^{\star} t'$ *then, either* t' *is a value or there exists a term* t'' *such that* $\mathrm{M} \ni t' \to t''$*.*

The No Stuck theorem is the final goal of this section. A closed and well typed term can't become stuck at runtime.

3.2 Resource Safety

What we want to be able to say is that, with this language, it is not possible to use twice or accidentally lose a resource value. We have to precisely define what are the resources available for a program and what are the resources created/used by the program execution. We expect the execution to use only the resources available by the program, and once a resource is used, we expect it to be no more available. This satisfies the constraint "use a resource at most once". In addition, we expect a resource to stay available until it is used. A resource should not become unavailable without being explicitly used. This satisfies the constraint "usa a resource at least once".

Definition 3. *Given a term t, we define the resources of t as the multiset $R(t)$. The symbol $\uplus$ denotes the multiset union.*

$$
\begin{array}{rcll}
R(n) & = & \varnothing & \\
R(x) & = & \varnothing & \\
R(x.j) & = & \varnothing & \\
R(v.j) & = & \varnothing & \\
R(\texttt{pub}\ t) & = & R(t) & \\
R(\texttt{let}\ x = t_1\ \texttt{in}\ t_2) & = & R(t_1) \uplus R(t_2) & \\
R(\texttt{call}\ \mathrm{M.F}\ [\,\overline{t}\,]) & = & R(\overline{t}) & \\
R(\texttt{pack}\ \mathrm{M.S}\ [\,\overline{t}\,]) & = & R(\overline{t}) & \\
R(\texttt{struct}\ \{k\}\ \mathrm{M.S}\ [\,\overline{t}\,]) & = & \{k\} \uplus R(\overline{t}) & \text{when IsLinear(M.S)} \\
R(\texttt{struct}\ \{k\}\ \mathrm{M.S}\ [\,\overline{t}\,]) & = & R(\overline{t}) & \text{when } \neg\text{IsLinear(M.S)} \\
R(\texttt{unpack}\ \{\overline{x}\} = t_1\ \texttt{in}\ t_2) & = & R(t_1) \uplus R(t_2) & \\
R(\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3) & = & R(t_1) \uplus R(t_2) & \\
R(\texttt{exec}\ \mathrm{M}\ t) & = & R(t) & \\
R(\overline{t}) & = & \biguplus_i R(t_i) &
\end{array}
$$

Informally speaking, resources of a term are the linear-struct identifiers appearing in the term (with their multiplicity). The only term that adds to $R(t)$ is the struct term, when the struct is linear.

There are two exceptional cases in which resource identifiers appear in the term, but we don't count them. The former is the selection and the latter is the branching.

In a selection $v.j$, we don't consider resources appearing in v, because those resources can never survive the evaluation. When a select term fully evaluates, the resulting value has no resources.

The resources of an $\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$ expression are defined as the resources of the guard plus the resources of the true branch; the resources of the false branch are not considered. Only one of the two branches survives the evaluation. The other branch is discarded with all the resources it may contain. Actually, when one of the two branches is chosen, resources of the discarded branch aren't lost, because the same resources are present in the other branch. We have proved that, as long as the two branches of an if expression exist in a well formed program being executed,

they have the same resources. For this reason, we can equivalently consider only the resources of the false branch in the definition of $R(\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3)$.

Definition 4. *Given a step of evaluation $\pi = \mathrm{M} \ni t \ \rightarrow\ t'$, we define the resources introduced by π as the multiset $R_I(\pi)$.*

$$
\begin{aligned}
R_I(\text{E-PACKED}) &= \varnothing \quad \text{when } \neg\text{IsLinear(M.S)} \\
R_I(\text{E-PACKED}) &= \{k\} \quad \text{when IsLinear(M.S)} \\
R_I(\text{E-LET}) &= R_I(\mathrm{M} \ni t_1 \ \rightarrow\ t_1') \\
R_I(\text{E-PACK}) &= R_I(\mathrm{M} \ni t_i \ \rightarrow\ t_i') \\
R_I(\text{E-UNPACK}) &= R_I(\mathrm{M} \ni t_1 \ \rightarrow\ t_1') \\
R_I(\text{E-CALL}) &= R_I(\mathrm{M} \ni t_i \ \rightarrow\ t_i') \\
R_I(\text{E-EXEC}) &= R_I(\mathrm{M} \ni t \ \rightarrow\ t') \\
R_I(\text{E-IF}) &= R_I(\mathrm{M} \ni t_1 \ \rightarrow\ t_1') \\
R_I(\text{E-PUBLISH}) &= R_I(\mathrm{M} \ni t \ \rightarrow\ t') \\
R_I(*) &= \varnothing
\end{aligned}
$$

The asterisk $*$ in $R_I(*)$ stands for all the rules of evaluation we have not explicitly listed. With this definition we are saying a new resource is introduced in the program when E-PACKED is executed to create a linear struct. The resource is k, the fresh identifier given to the new struct.

Definition 5. *Given a step of evaluation $\pi = \mathrm{M} \ni t \ \rightarrow\ t'$, we define the resources used by π as the multiset $R_U(\pi)$.*

$$
\begin{aligned}
R_U(\text{E-UNPACKED}) &= \varnothing \quad \text{when } \neg\text{IsLinear(M.S)} \\
R_U(\text{E-UNPACKED}) &= \{k\} \quad \text{when IsLinear(M.S)} \\
R_U(\text{E-PUBLISHED}) &= R(v) \\
R_U(\text{E-LET}) &= R_U(\mathrm{M} \ni t_1 \ \rightarrow\ t_1') \\
R_U(\text{E-PACK}) &= R_U(\mathrm{M} \ni t_i \ \rightarrow\ t_i') \\
R_U(\text{E-UNPACK}) &= R_U(\mathrm{M} \ni t_1 \ \rightarrow\ t_1') \\
R_U(\text{E-CALL}) &= R_U(\mathrm{M} \ni t_i \ \rightarrow\ t_i') \\
R_U(\text{E-EXEC}) &= R_U(\mathrm{M} \ni t \ \rightarrow\ t') \\
R_U(\text{E-IF}) &= R_U(\mathrm{M} \ni t_1 \ \rightarrow\ t_1') \\
R_U(\text{E-PUBLISH}) &= R_U(\mathrm{M} \ni t \ \rightarrow\ t') \\
R_U(*) &= \varnothing
\end{aligned}
$$

Publishing a resource uses the resource with all of its nested resources, while unpacking a resource uses the resource being unpacked but not the nested resources. When we publish a resource, the value, with all its sub-values, is moved to the global state, and the program can no longer manipulate it. The program loses control of the resource and has no way to regain it. To modify the same resource, another program must be invoked passing the resource as an argument. When we deconstruct a resource with an unpack, we destroy the resource, but we keep the values inside

it (which may themselves be resources), and we can use those values in the body of the unpack.

In summary, the function $R(t)$ inspects the structure of the term t and tells us what are the resources still available to the term, that is the resources the term could use in subsequent computation steps. The functions $R_I(\pi)$ and $R_U(\pi)$ tell us respectively what are the resources we consider introduced and used by the computation step π. Imprecisely speaking, we will prove that when t do a step in t', the missing resources in t' are exactly the resources used by the step, and the new resources appeared in t' are exactly the resources introduced by the step.

Since we want to talk about resources preserved in a program during multiple computation steps, we extend the definition of R_I and R_U to sequences of steps.

Definition 6. *Given a sequence of steps* $\pi^\star = \mathrm{M} \ni t \ \to^\star \ t'$*, we define the resources introduced and used by* $\pi^\star$ *as follows:*

$$\begin{aligned}
R_I(\mathrm{M} \ni t \ \to^\star \ t) &= \varnothing \\
R_I(\mathrm{M} \ni t_1 \ \to^\star \ t_2, \mathrm{M} \ni t_2 \ \to \ t_3) &= R_I(\mathrm{M} \ni t_1 \ \to^\star \ t_2) \uplus R_I(\mathrm{M} \ni t_2 \ \to^\star \ t_3) \\
R_U(\mathrm{M} \ni t \ \to^\star \ t) &= \varnothing \\
R_U(\mathrm{M} \ni t_1 \ \to^\star \ t_2, \mathrm{M} \ni t_2 \ \to \ t_3) &= R_U(\mathrm{M} \ni t_1 \ \to^\star \ t_2) \uplus R_U(\mathrm{M} \ni t_2 \ \to^\star \ t_3)
\end{aligned}$$

Lemma 8 (Resources of normal values)**.** *If* $\mathrm{M}_v \ni \Delta_1 \vdash v : \mathrm{T}_v \ \rhd \ \Delta_2$ *holds, and* $\neg\mathrm{IsLinear}(\mathrm{T}_v)$*, then* $R(v) = \varnothing$*.*

We are saying: a normal value has no resources. This is guaranteed by the hypothesis of good formation for struct definitions: a normal struct can't contain linear fields.

Lemma 9 (Resource substitution)**.** *Given* $\mathrm{M}_v \ni \Delta_1 \vdash v : \mathrm{T}_v \ \rhd \ \Delta_2$*, the following properties holds:*

1. *Be* U *a use of* T_v*; If* $\mathrm{M} \ni \Gamma_1, x : \mathrm{U} \vdash t : \mathrm{T} \ \rhd \ \Gamma_2, x : \mathrm{U}$*, then*
   $R(t\{x := v\}) = R(t)$
2. *If* $\mathrm{M} \ni \Gamma_1, x : \mathrm{T}_v^\circ \vdash t : \mathrm{T} \ \rhd \ \Gamma_2, x : \mathrm{T}_v^\bullet$*, then*
   $R(t\{x := v\}) = R(t) \uplus R(v)$
3. *If* $\mathrm{M} \ni \Gamma_1, x : \mathrm{T}_v^\circ \vdash t : \mathrm{T} \ \rhd \ \Gamma_2, x : \mathrm{T}_v^\downarrow$*, then*
   $R(t\{x := v\}) = R(t) \uplus R(v)$

The property 3) follows immediately from the first two. I write it explicitly because it comes in handy in proofs. When T_v is normal, v has no resources and $\mathrm{T}_v^\downarrow = \mathrm{T}_v^\circ$, so we can use the first property. When T_v is linear, $\mathrm{T}_v^\downarrow = \mathrm{T}_v^\bullet$ and the second property applies.

The property 2) states that if we have a well typed term that uses the linear variable x, and we substitute x with a value v, the term we obtain has the same resources as the original term t plus the resources of v.

Lemma 10 (Resource preservation)**.** *If* $\mathrm{M} \ni \Gamma_1 \vdash t : \mathrm{T} \rhd \Gamma_2$ *and* $\pi = \mathrm{M} \ni t \rightarrow t'$ *then:*

$$R(t) \uplus R_I(\pi) = R(t') \uplus R_U(\pi)$$

Theorem 2 (Resource preservation)**.** *If* $\mathrm{M} \ni \Gamma_1 \vdash t : \mathrm{T} \rhd \Gamma_2$ *and* $\pi^\star = \mathrm{M} \ni t \rightarrow^\star t'$ *then:*

$$R(t) \uplus R_I(\pi^\star) = R(t') \uplus R_U(\pi^\star)$$

When a term t is well typed, and it evolves into t', every resource present in t or introduced by the computation is either still present in t' or has been explicitly used.

Chapter 4

Mechanization in Agda

4.1 Introduction

Formalizing a language and proving its properties can be a difficult task. The proofs of different lemmas and theorems are often similar to each other. When writing proofs by hand it is easy to make mistakes, and it may be difficult to find them. Agda[1] is a proof assistant based on Martin-Löf intentional type theory [18] [9] that allows to write machine verifiable formal proofs in a constructive way. In Agda, we encoded the FM language syntax, the typing rules, the operational semantics rules, and we proved the properties presented in Chapter 2. In particular, two of the most important properties we proved are the Type Preservation lemma (Lemma 6) and the Resource Preservation lemma (Lemma 10). To give an idea of how a proof looks like in Agda, we included the proofs of the two properties just mentioned respectively in Appendix D.3 and D.4. The Agda code fragments we present in this chapter and in the appendices are most of the times incomplete. Refer to the github repository of the project [23] for the complete code.

In Agda, according to Curry-Howard correspondence, we write propositions as types and proofs as values of those types. "A formula has a proof if and only if the corresponding type is inhabited" [21][p. 48]. The computer checks the correctness of the proofs by type-checking the code. This increases confidence about the correctness of the proof. Proving properties becomes a matter of writing code that type-checks.

Agda is not only a useful tool to confirm the veracity of what already discovered on paper, but it also forces to deal rigorously with every small formalization detail. On one hand, this can be tedious because it distracts and makes the programmer lose the global picture, but on the other hand gives the programmer a deeper understanding of the formalization.

We have tried to exploit the best of the two worlds (paper proofs and computer proofs). We have first written the proofs on paper, and next to it, more slowly, we have developed the Agda code. Having an outline of the proofs on paper was essential for us to speed up the development in Agda. As said before, while working in Agda, it is easy to get lost in the details, and having a paper proof to follow was very useful.

[1] https://agda.readthedocs.io/en/latest/getting-started/what-is-agda.html

For the sake of clarity, the Agda code snippets we present in this chapter may be simplified, and may not be valid Agda code. When appropriate we remove some details that would make the code harder to read and understand.

In the rest of the chapter, we will show and explain some parts of the Agda code we developed. Even though we try to convey the main ideas without going into too much detail, some familiarity with Martin-Löf type theory and Agda[2] would be helpful to the reader.

4.2 Encoding of Language Terms

The first step in the formalization of a programming language is the definition of its terms. In Listing 6 we show the definition of the terms of FM in Agda. The first line `data Term : Set where` declares the new type `Term`. The type `Term` has a constructor for each kind of expression admitted by the language. Each constructor encodes a different way to produce a value of type `Term`. Constructors are functions that take arguments and return a value of the type they are a constructor of. The constructor `num_` has type `(n : ℕ) → Term`, that is, it is a function that takes a natural number `n` and returns a `Term`. The constructor `if_then_else_` takes three terms (`t1`, `t2`, `t3`) and produces a new `Term`.

The underscore in constructor names is used to define mixfix operators[3], such that it is possible to write a conditional term in the form `if t1 then t2 else t3` in addition to the standard form `if_then_else_ t1 t2 t3`.

Here are some examples of Agda terms and their corresponding FM terms:

Agda term	FM term
`num 3`	3
`if (num 0) then (num 1) else (num 2)`	`if` 0 `then` 1 `else` 2
`var 0`	x
`Let (num 3) In (var 0)`	`let` $x = 3$ `in` x

Note that variables are represented with indices, and not with labels. To avoid having to deal with alpha-equivalence, that uselessly complicate the proofs, we use a nameless representation of terms, where variables are indexes pointing to the binder that introduced that variable, as detailed below. The constructor for the variable term `var_` takes a natural number and returns a term. Thus `var 0` and `var 3` are terms representing variables.

Since variables don't have names, in binders don't appear variable names. For example, a let binding in FM has the form `let` $x = t_1$ `in` t_2, where x is a variable name, while in Agda, it has the form `Let t1 In t2`, where there is no variable name.

Variables are indices pointing to binders. In FM we have three kinds of binders: the let term, the unpack term, and the function definition. The variable `var k` stands for "the variable introduced by the k^{th} binder". Binders are numbered from the innermost to the outermost, starting from 0. For example, the FM term `let` $x =$

[2] https://plfa.github.io/Lambda/#lambda-introduction-to-lambda-calculus

[3] https://agda.readthedocs.io/en/v2.6.4.1/language/mixfix-operators.html

```
data Term : Set where
    num_           : (n : ℕ) → Term
    var_           : (x : ℕ) → Term
    Let_In_        : (t1 t2 : Term) → Term
    call           : (fid : FunId) → (ts : Vec Term n) → Term
    if_then_else_  : (t1 t2 t3 : Term) → Term
    pack           : (sid : StrId) → (ts : Vec Term n) → Term
    unpack_In_     : (t1 t2 : Term) → Term
    _·_            : (t : Term) → (j : Fin Nsf) → Term
    pub_           : (t : Term) → Term

    struct         : {n : ℕ} → (k : K) → (sid : StrId) → (ts : Vec Term n) → Term
    exec           : (M2 : Fin Nm) → Term → Term
```

Listing 6: FM terms.

$3\,\texttt{in}\,x$ corresponds to the nameless Agda term `Let (num 3) In (var 0)`, while the term $\texttt{let}\,x = 3\,\texttt{in}\,\texttt{let}\,y = 4\,\texttt{in}\,x$ corresponds to `Let (num 3) In Let (num 4) In (var 1)`.

Terms represented in this way are called De Bruijn terms[4], and variable indices are called De Bruijn indices. The interesting feature of this representation is that the representation of a term is unique. The named term $\texttt{let}\,x = 3\,\texttt{in}\,x$ can be equivalently represented as $\texttt{let}\,y = 3\,\texttt{in}\,y$, while `Let (num 3) In (var 0)` is the only way of representing the same term with de Bruijn indices.

4.3 Syntax Constraints

In Listing 7 we show the encoding of FM function definition, struct definition, module and program in Agda. `Str`, `Fun`, `Module` and `Program` are all types, as `Term` is, but are declared using the `record` keyword[5]. Records are Agda types for grouping values together. A record has a number of fields, declared after the `field` keyword. The record `Str` has two fields: `isLin` and `fieldsT`. The first field is a boolean that tells whether the struct is linear or not. The second field is a vector of types, representing the types of the fields of the struct. All the definitions correspond to those already seen in Chapter 2.

To simplify the formalization in Agda, we imposed some constraints on the program syntax. In particular we impose that, before writing a program, the programmer must declare 5 constant numbers: `Nm`, `Ns`, `Nsf`, `Nf`, `Nfa`, and the program is then forced to respect the following constraints:

- The program must define `Nm` modules.
- Each module must define `Ns` different structs and `Nf` different functions.
- Each struct definition (of each module) must have `Nsf` fields.

[4] https://plfa.github.io/DeBruijn/#debruijn-intrinsically-typed-de-bruijn-representation

[5] https://agda.readthedocs.io/en/v2.6.4.1/language/record-types.html

```
record Str : Set where
    field
        isLin   : Bool
        fieldsT : Vec Type Nsf

record Fun : Set where
    field
        argsT : Vec Type Nfa
        retT  : Type
        body  : Term

record Module : Set where
    field
        strs : Vec Str Ns
        funs : Vec Fun Nf

record Program : Set where
    field
        mods : Vec Module Nm
```

Listing 7: Definition of an FM program and of its components.

- Each function definition (of each module) must have `Nfa` parameters.

Declaring those 5 numbers before writing a program, the language syntax can be parameterized with them. By doing so, we are able to ensure that by construction, all struct and function identifiers that appear in a program are valid: they point to actually defined structs and functions. It would be impossible to write (in Agda) a valid FM program where a function call is made to a function that does not exist, or where we pack a struct that does not exist.

However, these constraints are not a limitation for the expressiveness of the language. Any valid program that does not comply with these additional constraints can be converted into a program that does. In fact, we can always add unused fields to structs, unused parameters to functions, and unused struct and function definitions to modules.

4.4 Encoding Operational Semantics

Operational semantics rules are encoded as constructors of the dependent type `_∋_⇒_`, which is presented partially in Listing 8 (complete definition in Appendix D.1). `_∋_⇒_` is a function that accepts three arguments: a module `M` and two terms `t`, `t'`, and produces a type. The **type** `M ∋ t ⇒ t'` corresponds to the proposition:

In module `M`*, the term* `t` *can do a step in* `t'`*.*

Each constructor of `_∋_⇒_` encodes one of the evaluation rules presented in Chapter 2 and it is named according to the corresponding evaluation rule's label. An **element** of type `M ∋ t ⇒ t'` is a witness for the corresponding proposition. Therefore

```
data _∋_⇒_ : Fin Nm → Term → Term → Set
-- ... cut content

    Elet : M ∋ t1 ⇒ t1'
        --------------------
        → M ∋ (Let t1 In t2) ⇒ (Let t1' In t2)

    Elet2 : Value t1
        --------------------
        → M ∋ (Let t1 In t2) ⇒ beta-red (t1 V.:: []) t2
-- ... cut content

    Ecall : {ts ts' : Vec Term n}
        → M ∋ ts ⇒v ts'
        --------------------
        → M ∋ (call fid ts) ⇒ (call fid ts')

    Ecalled : {f : Fin Nf}
      → {ts : Vec Term n}
      → ValueV ts
      --------------------
      → M ∋ (call (fId M2 f) ts) ⇒ exec M2 (beta-red ts (gBody M2 f))
```

Listing 8: Snippet of FM operational semantics rules. Complete definition in Appendix D.1

producing an element of type `M ∋ t ⇒ t'`, using one of the constructors listed in the definition (e.g. `Elet` or `Ecall`), corresponds to proving the associated proposition, that is proving that the FM term `t` evolves to `t'` in module `M`.

The constructor `Elet` can prove `Let t1 In t2` reduces in one step to `Let t1' In t2` if we are able to provide a proof for `M ∋ t1 ⇒ t1'`. In `Elet2`, we see the use of the `Value` predicate and the `beta-red` function. `Value t1` encodes the proposition *"The term `t1` is a value"*, while `beta-red (t1 ` :: ` []) t2` replaces with `t1`, in `t2`, all the variables pointing to this let binding. `beta-red ts t2` is the equivalent of $t_2\{\overline{x} := \overline{t_s}\}$.

`Ecalled` and `Ecall` use the `ValueV` and the `_∋_⇒v_` predicate. `ValueV ts` means *"All the terms in `ts` are values"*, while `M ∋ ts ⇒v ts'` encodes the proposition *"In module `M`, the vector of terms `ts` can do a step in the vector of terms `ts'`"*. In `Ecalled`, `gBody` is a function that returns the body of the function given the module `M2` and the index `f` of the function (inside the module).

4.4.1 Using Operational Semantics

In Agda we can now prove that a term reduces in one step to another term. For example, we prove that `Let num 3 In var 0` reduces in one step to `num 3` in any module, i.e. the FM judgement $M \ni \texttt{let}\, x = 3\, \texttt{in}\, x \;\rightarrow\; 3$. We prove it by constructing an Agda term `pt3` and letting the proof assistant check it is a well typed term of type

```
{M : Fin Nm} → M ∋ Let num 3 In var 0 ⇒ num 3.
```

```
pt3 : {M : Fin Nm} → M ∋ Let num 3 In var 0 ⇒ num 3
pt3 = Elet2 Vnum
```

The prove `pt3` uses the `Elet2` constructor passing to it `Vnum`, which is a proof that `num 3` is a value. `Vnum` is a constructor for the `Value` predicate and its type is `{n : ℕ} → Value (num n)`.

Using the definition of multi-step reduction `_∋_⇒*_`, and with the aid of some utility functions, we can prove a term evaluates to another in a number of steps. The term `pt4` proves that `Let (num 0) In (if (var 0) then (num 1) else (num 2))` reduces in two steps to `(num 2)` in any module.

```
pt4 : {M : Fin Nm} →  -- For any module M
    M ∋ Let (num 0) In if (var 0) then (num 1) else (num 2) ⇒* num 2
pt4 = begin⇒
    Let (num 0) In if (var 0) then (num 1) else (num 2)   ⇒⟨ Elet2 Vnum ⟩
    if (num 0) then (num 1) else (num 2)                  ⇒⟨ Eiffalse ⟩
    num 2 ⇒∎
```

The `begin⇒_`, `_⇒⟨_⟩_` and `_⇒∎` functions are inspired by equality reasoning in Agda[6]. The proof starts with the `begin⇒` marker and ends with the `⇒∎` marker. On each line but the last there is a term on the left and a proof on the right, this latter surrounded by `⇒⟨ ... ⟩`. When we want to prove that (`M ∋ t ⇒* t'`), the term on the first line is `t` while the lonely term in the last line is `t'`. The proof on each line proves that the term on the same line evaluates in one step to the term sitting in the following line. The result is a chain of proofs of one-step reduction building up a proof for multi-step reduction.

4.5 Encoding Environments

In Listing 9 we show the definition of the context and the usage-context in Agda. A context `Env` for nameless terms is just a vector of types. The context doesn't contain bindings between variable names and types, as is usually the case, but only types. Due to the adoption of De Bruijn indexes instead of variable names, the order in which types appear in the context is important; note that this is not true for standard contexts.

A value of type `Usage T` is the type `T` decorated with a marker chosen between ○ and •. When `T1` is a type, `(T1 ○)` and `(T1 •)` are two elements of type `Usage T1`.

The usage-context `UEnv` is basically a vector of usages (types decorated with usage markers). `UEnv` is a dependent type, with two arguments: an implicit argument `l` of type `ℕ`, which is the length of the usage-context, and an argument of type `Env l`, which is a context of length `l`. Implicit parameters[7], those enclosed in curly braces, are parameters that can be inferred by the type-checker, and they are not necessary

[6] https://plfa.github.io/Equality/#chains-of-equations

[7] https://agda.readthedocs.io/en/v2.6.4.1/language/implicit-arguments.html

```
Env = Vec Type

data Usage : Type → Set where
    _∘ : (T : Type) → Usage T
    _• : (T : Type) → Usage T

data UEnv : {l : ℕ} → Env l → Set where
    []   : UEnv V.[]
    _u::_ : {l : ℕ} {T : Type} {Δ : Env l}
        → Usage T → UEnv Δ
        → UEnv (T V.:: Δ)
```

Listing 9: Context and usage-context.

to be passed explicitly. So, for example we can write `UEnv Δ` instead of `UEnv {2} Δ` when `Δ` is a context of length 2.

Given a context `Δ`, a value of type `UEnv Δ` is a vector which contains the same types of `Δ` decorated with a usage marker. Two values of type `UEnv Δ` are usage-contexts containing the same types, in tha same order, but with possibly different usage markers. In the following code snippet, `Δ` is an environment of length 2 and `Γ1` and `Γ2` are two values with the same type `UEnv Δ`. The types contained in `Γ1` and `Γ2` are the same types contained in `Δ`, but the usage makers are different.

```
Δ  : Env 2;   Δ  = Tint    :: CoinT    :: []
Γ1 : UEnv Δ;  Γ1 = Tint • u:: CoinT • u:: []
Γ2 : UEnv Δ;  Γ2 = Tint ∘ u:: CoinT • u:: []
```

The usage-context is defined in this way because in the typing rules it is always true that the input and output contexts contain the same types in the same order. By using this definition of usage-context, we can ensure that this property of the typing rules is encoded in the premises of the rules, and it is not necessary to prove it separately.

4.6 Encoding Typing Rules

In Listing 10 we present a fragment of the definition of the dependent type `HasType` (complete definition in Appendix D.2). The type `HasType M Γ1 t T Γ2` corresponds to the following proposition:

In module `M`, with input context `Γ1`, the term `t` has type `T`
and the output context is `Γ2`.

Each constructor of `HasType` encodes one of the typing rules presented in Chapter 2 and it is named according to the corresponding typing rule's label. `Tnum` is the constructor for the rule T-NUM, `Tif` for T-IF, and so on. The two constructors `Tnum` and `Tif` are functions that can be read this way:

```
data HasType  (M : Fin Nm) : UsageEnv Δ → Term → Type → UsageEnv Δ → Set where
-- ... cut content

    Tnum :
        --------------------
        HasType M Γ (num n) Tint Γ
-- ... cut content

    Tif : HasType M Γ1 t1 Tint Γ2
      →   HasType M Γ2 t2 T    Γ3
      →   HasType M Γ2 t3 T    Γ3
      --------------------
      → HasType M Γ1 (if t1 then t2 else t3) T Γ3
-- ... cut content
```

Listing 10: Snippet of FM typing rules. Complete definition in Appendix D.2.

- `Tnum` : For any module `M`, context `Γ` and natural number `n` the `Tnum` constructor can output a proof that the term `num n` has type `Tint` in `M` with input and output context `Γ`.
- `Tif` : If you provide a proof term showing the guard `t1` is well typed, and two values proving that both branches `t1` and `t2` are well typed, then the `Tif` constructor can output a proof that the whole term `(if t1 then t2 else t3)` is well typed. Note how the module is the same in all three premises and in the conclusion, the input context for the two branches is the same and it is the output context of the guard, the input context of the conclusion is the input context of the guard, and the output context of the conclusion is the output context of the two branches. The correspondence is one or to one with the rule T-IF.

4.6.1 Using Typing Rules

Using the type `HasType`, we can prove a term is well typed or prove that it is not. For example, we can prove the variable x has type Int in any module M when the input context is $x : \text{Int}^{\circ}$ (see `pt1`). We can also prove the same term can't be typed when the input context is $x : \text{T}^{\bullet}$ (see `pt2`).

```
pt1 : {M : Fin Nm} → HasType M (Tint ∘ u:: []) (var 0) Tint (Tint ∘ u:: [])
pt1 = Tvar (Xz (λ ()))

pt2 : {M : Fin Nm} {T : Type} {Γ2 : UsageEnv (T :: [])}
        → HasType M (T • u:: []) (var 0) T Γ2
        → ⊥
pt2 (Tvar ())

```

```
9  -- pt2 ht = ?          do a case split on ht
10 -- pt2 (Tvar htx) = ?  do a case split on htx
11 -- pt2 (Tvar ())
```

In `pt1` we use the `Tvar` constructor of the type `HasType`, which is the constructor corresponding to the rule T-VAR. The argument of the `Tvar` constructor (`Xz (λ ())`) proves that at the head of the input context there is a variable of type `Tint` marked as usable (marked with a ○).

In `pt2` we are proving that for any module `M`, any type `T` and output context `Γ2`, the term `var 0` cannot be well typed (notice the `→ ⊥` at the end of the type) when the input context is `T • u:: []`. The proposition $\forall\, \mathrm{M}\ \forall\, \mathrm{T}\ \forall\, \Gamma_2\ \neg(\mathrm{M} \ni x : \mathrm{T}^{\bullet} \vdash x : \mathrm{T} \rhd \Gamma_2)$ is represented in Agda with the type of the function `pt2`. To prove the proposition we have to implement the function `pt2`. The function has 4 parameters (3 implicit and 1 explicit) and returns a value of type `⊥`. For all possible values of the parameters, the function must return a value of type `⊥`. `⊥` is the empty type with no constructors, so it is impossible to produce a value of type `⊥`. What we do then, is prove that the domain of the function is empty: there exists no value of type `HasType M (T • u:: []) (var 0) T Γ2`.

The proof is obtained, as described in code comments, by doing case split on the explicit input argument of the function. Informally, with the first case split on `ht`, which is the first explicit parameter of the function, Agda realizes a value of type `HasType M (T • u:: []) (var 0) T Γ2` can only be in the form `Tvar htx`, because `Tvar` is the only constructor that gives a type to a term in the form `var n`. Then, with the case split on `htx`, Agda understands there is no constructor that can produce a value of that type. Hence Agda substitutes `htx` with the absurd pattern `()`[8] and removes the equal sign preceding the function body; there is no body to define because the domain of the function is empty.

[8] `https://agda.readthedocs.io/en/v2.6.4.1/language/function-definitions.html#absurd-patterns`

Chapter 5

Conclusions

The objective of this work was to formally study the correctness guarantees provided by state of the art typed languages for smart contracts. Although the main concepts and ideas underlying a blockchain system are quite simple, each blockchain manages smart contracts, users, access control, signature and storage in different ways. Those differences are reflected in smart contracts programming languages and frameworks in a way that complicates the programmer's work. To be productive and, more importantly, to develop correct and secure programs, the programmer must deeply understand the platform and its underlying mechanisms. The relatively narrow spectrum of use cases for blockchain programs let us think that a cross-platform blockchain programming language that hides platform specific details could be developed.

Move seemed to be a step forward in this direction declaring itself as a platform-agnostic language for writing safe smart contracts (declared in the homepage of the github project[1]). Exploring it more deeply and understanding the differences between its two main dialects: Sui Move and Aptos Move, we realized that Move is not as portable as we would have liked. Despite of this, we think linear types, as used by Move to represent resources (assets in particular), can be beneficial in the smart contracts programming context, and our work supports this claim.

We gave a clean explanation of the move semantics of the Move language, and a new viewpoint for reasoning about that. This view suggests to investigate in the use of the framework of abstract interpretation in this context. It seems feasible to think about a static analysis tool for other languages, that don't have linear types, that is able to detect the same errors in the manipulation of resources that Move linear types can detect.

We have formalized a small part of the Move language, the core language FM, and we have proved its typing and operational semantics rules guarantee a Resource Preservation property (Theorem 2) which we consider valuable for the smart contracts programmer. Furthermore, we validated our results using the Agda proof assistant. We strongly believe a programming language for the blockchain environment deserves a clean and exact formalization with machine verifiable proofs about its properties. Users of the blockchain trust the execution environment and its declared properties when they interact with smart contracts that manipulate their

[1] `https://github.com/move-language/move`

valuable assets. This trust should be supported by formal methods.

Move, in addition to be a high level language for writing smart contracts, is also an executable bytecode language with resources. In [7] Blackshear et al. provide a formalization of the Move bytecode and prove that "it enjoys resource safety, a conservation property for program values that is analogous to conservation of mass in physical world". In our work we prove that a subset of the Move source-code language, the core language FM, enjoys an equivalent property. Our work differs form [7] in that we prove the resource safety property at the source code level, rather then at the bytecode level, and we provide computer checked proofs of our results. In [8] Blackshear et al. describe the Move Borrow Checker, a static analysis tool that ensures the absence of memory violations in all Move bytecode programs, and they prove its properties. In [20] Patrignani and Blackshear define and formalize robust safety for Move programs. In [30] Zhong et al. describe the Move Prover, a tool that can automatically verify programmer-written functional correctness properties for Move procedures. The research community is active in the formalization of blockchain platforms [5] [12] [4].

The formalization in Agda was a challenging, stimulating and time consuming task. The mean mental effort per line of Agda code is high. We had to solve different challenges an more than once we found ourselves stuck, and we had to change our approach. These challenges, while interesting, are quite technical, and would require a detailed explanation that is beyond the scope of this work. We had to think in a constructive way, adapting our statements and proofs to the constructive nature of Agda. We rediscovered fundamental concepts and properties under a different light, learning how to leverage the Agda standard library.

Below we list some interesting topics that would merit further study:

1. **Linear types in other languages.** We have focused on the Move language and on the way it uses linear types, but linear types could be added to other languages. Solidity, for example, is the most used language for writing smart contracts and may benefit from the addition of linear types. It would be interesting to understand how linear types interact with Solidity language features, and to check if some difference emerge between linear types in Solidity and linear types in Move. Also the Solana platform, with its Rust based smart contracts, could be a good candidate for the addition of linear types.

2. **Imperative Move.** Move is an imperative language, where variables have an associated memory location that can be written with an assignment operator. Thanks to this, Move can implement references and mutable data structures. FM instead, is more like a functional language, where variables are bounded to values at the time of their declaration and cannot be modified. It would be interesting to understand in more detail what are the advantages and disadvantages, in the blockchain environment, of a primarily functional language. One could formalize an imperative variant of FM, including references, to compare it to the functional version.

3. **Access Control.** Access control is an important aspect of smart contracts programming. As we have see in Section 2.2.1, our simplified blockchain model

doesn't have an access control mechanism. We should think of a flexible way to manage access control to resources. In addition, one could look into whether the type system can be used to prevent certain types of errors related to access control.

4. **Linear lambda functions.** In Move and in FM there are no first class function values, and linear typing is only applied to struct types. In general, in a programming language with abstractions, like the one presented in [21, p. 8], linear typing is also applied to function types. The type of a function value can be linear or not, and the type system ensure a linear function is consumed exactly once. Consuming a linear function means calling it. It may be interesting to investigate if linear functions can be seen somehow as resources and if they can be useful in the blockchain environment for the creation of new programming patterns.

Appendix A

Smart contract Use cases in Sui

A.1 Crowdfund

Specification

The Crowdfund contract allows users to donate native cryptocurrency to fund a campaign. To create a crowdfund, one must specify:

- The recipient of the funds.
- The goal of the campaign, that is the least amount of currency that must be donated in order for the campaign to be successful.
- The deadline for the donations.

After creation, the following actions are possible:

- **donate:** anyone can transfer native cryptocurrency to the contract until the deadline.
- **withdraw:** after the deadline, the recipient can withdraw the funds stored in the contract, provided that the goal has been reached.
- **reclaim:** after the deadline, if the goal has not been reached donors can withdraw the amounts they have donated.

Code

```
module crowdfund::crowdfund {
    use sui::tx_context::{TxContext, sender};
    use sui::object::{Self, UID, ID};
    use sui::transfer::{share_object, transfer, public_transfer};
    use sui::coin::{Self, Coin};
    use sui::clock::{Clock, timestamp_ms};

    const ErrorBadTiming: u64 = 0;
    const ErrorGoalReached: u64 = 1;
```

```
10      const ErrorGoalNotReached: u64 = 2;
11      const ErrorInvalidReceipt: u64 = 3;
12
13      struct Crowdfund<phantom T> has key {
14          id: UID,
15          endDonate: u64,         // After this timestamp, no more donations are accepted
16          goal: u64,              // Amount of Coins to be raised
17          receiver: address,      // This address will receive the money
18          money: Coin<T>,
19      }
20
21      struct Receipt<phantom T> has key {
22          id: UID,
23          crowdfundId: ID,
24          amount: u64,
25      }
26
27      public entry fun create_crowdfund<T>(goal: u64, receiver: address,
28          endDonate: u64, ctx: &mut TxContext)
29      {
30          let crowdfund = Crowdfund<T> {
31              id: object::new(ctx),
32              endDonate: endDonate,
33              goal: goal,
34              receiver: receiver,
35              money: coin::zero<T>(ctx),
36          };
37          share_object(crowdfund);
38      }
39
40      public entry fun donate<T>(crowdfund: &mut Crowdfund<T>, money: Coin<T>,
41          clock: &Clock, ctx: &mut TxContext)
42      {
43          assert!(timestamp_ms(clock) <= crowdfund.endDonate, ErrorBadTiming);
44
45          let receipt = Receipt<T> {
46              id: object::new(ctx),
47              crowdfundId: object::id(crowdfund),
48              amount: coin::value(&money),
49          };
50          coin::join(&mut crowdfund.money, money);
51          transfer(receipt, sender(ctx));
52      }
53
54      public entry fun withdraw<T>(crowdfund: &mut Crowdfund<T>, clock: &Clock,
55          ctx: &mut TxContext)
```

```
56      {
57          assert!(timestamp_ms(clock) > crowdfund.endDonate, ErrorBadTiming);
58          assert!(coin::value(&crowdfund.money) >= crowdfund.goal, ErrorGoalNotReached);
59
60          let total = coin::value(&crowdfund.money);
61          let money = coin::split(&mut crowdfund.money, total, ctx);
62          public_transfer(money, crowdfund.receiver);
63      }
64
65      public entry fun reclaim<T>(crowdfund: &mut Crowdfund<T>, receipt: Receipt<T>,
66          clock: &Clock, ctx: &mut TxContext)
67      {
68          assert!(timestamp_ms(clock) > crowdfund.endDonate, ErrorBadTiming);
69          assert!(coin::value(&crowdfund.money) < crowdfund.goal, ErrorGoalReached);
70          assert!(object::id(crowdfund) == receipt.crowdfundId, ErrorInvalidReceipt);
71
72          let Receipt<T> {
73              id,
74              crowdfundId: _,
75              amount,
76          } = receipt;
77          object::delete(id);
78          let money = coin::split(&mut crowdfund.money, amount, ctx);
79          public_transfer(money, sender(ctx));
80      }
81  }
```

A.2 Auction

Specification

The Auction contract allows a seller to create an auction based on the native cryptocurrency and any participant to bid. To create the an auction, the seller must specify:

- The minimum bid of the auction.
- The duration of the auction.
- The object of the auction, which in this case it is represented as a string.

After creation, the following actions are possible:

- **start:** after the auction creation, the seller can start the auction.
- **bid:** after the auction starts, any participant can bid an amount of native cryptocurrency and transfer that amount to the contract until the duration time elapses. In the event of a raise, the contract returns the old bid to the participant.
- **withdraw:** at any time, participant can withdraw his bid if this is not the currently highest one.
- **end:** after the deadline, the seller ends the auction and withdraws the highest bid.

Code

```
1  module auction::auction {
2      use sui::tx_context::{TxContext, sender};
3      use sui::object::{Self, UID};
4      use sui::transfer::{share_object, public_transfer};
5      use sui::coin::{Self, Coin};
6      use sui::clock::{Clock, timestamp_ms};
7      use sui::dynamic_field;
8      use std::string::{String};
9
10     const StateWaitStart: u8 = 0;
11     const StateWaitClosing: u8 = 1;
12     const StateClosed: u8 = 2;
13
14     const ErrorInvalidState: u64 = 0;
15     const ErrorPermissionDenied: u64 = 1;
16     const ErrorBadTiming: u64 = 2;
17     const ErrorBidTooLow: u64 = 3;
18     const ErrorHighestBidderCantWithdraw: u64 = 4;
19
```

```
20      struct Auction<phantom T> has key {
21          id: UID,
22          state: u8,
23          seller: address,
24          thing: String,          // The thing being auctioned
25          endTime: u64,           // After this timestamp, the auction is closed
26          highestBidder: address,
27          highestBid: u64,
28      }
29
30      public entry fun create_auction<T>(thing: String, minimumBid: u64,
31          ctx: &mut TxContext)
32      {
33          let auction = Auction<T> {
34              id: object::new(ctx),
35              state: StateWaitStart,
36              thing: thing,
37              seller: sender(ctx),
38              endTime: 0,
39              highestBidder: @0x00,
40              highestBid: minimumBid,
41          };
42          share_object(auction);
43      }
44
45      public entry fun start<T>(auction: &mut Auction<T>, duration: u64, clock: &Clock,
46          ctx: &mut TxContext)
47      {
48          assert!(auction.state == StateWaitStart, ErrorInvalidState); // Auction already started
49          assert!(sender(ctx) == auction.seller, ErrorPermissionDenied);
50
51          auction.state = StateWaitClosing;
52          auction.endTime = timestamp_ms(clock) + duration;
53      }
54
55      public entry fun bid<T>(auction: &mut Auction<T>, coin: Coin<T>, clock: &Clock,
56          ctx: &mut TxContext)
57      {
58          assert!(auction.state == StateWaitClosing, ErrorInvalidState); // Auction not started
59          assert!(timestamp_ms(clock) < auction.endTime, ErrorBadTiming);
60          assert!(coin::value(&coin) > auction.highestBid, ErrorBidTooLow);
61
62          let sender = sender(ctx);
63          // if a participant makes a new bid, the previous one is automatically withdrawn
64          if (auction.highestBidder == sender) {
65              let oldMoney = dynamic_field::remove<address, Coin<T>>(&mut auction.id, sender);
```

```
66              public_transfer(oldMoney, sender);
67          };
68
69          auction.highestBidder = sender;
70          auction.highestBid = coin::value(&coin);
71          dynamic_field::add(&mut auction.id, sender, coin);
72      }
73
74      public entry fun withdraw<T>(auction: &mut Auction<T>, ctx: &mut TxContext) {
75          assert!(auction.state != StateWaitStart, ErrorInvalidState); // Auction not started
76
77          let sender = sender(ctx);
78          assert!(sender != auction.highestBidder, ErrorHighestBidderCantWithdraw);
79
80          let money = dynamic_field::remove<address, Coin<T>>(&mut auction.id, sender);
81          public_transfer(money, sender);
82      }
83
84      public entry fun end<T>(auction: &mut Auction<T>, clock: &Clock, ctx: &mut TxContext) {
85          let sender = sender(ctx);
86          assert!(sender == auction.seller, ErrorPermissionDenied);
87          assert!(auction.state == StateWaitClosing, ErrorInvalidState); // Auction not started
88          assert!(timestamp_ms(clock) >= auction.endTime, ErrorBadTiming); // Auction not ended
89
90          auction.state = StateClosed;
91          let money = dynamic_field::remove<address, Coin<T>>(&mut auction.id, auction.highestBidder);
92          public_transfer(money, sender);
93      }
94  }
```

A.3 Escrow

Specification

The Escrow contract involves a buyer and a seller. The contract acts as a trusted intermediary to protect the buyer from the possible non-delivery of the purchased goods. The buyer is expected to deposit the required amount in the contract after the contract initialization. The seller initializes the a new escrow by setting:

- The address of the buyer.
- The amount of native cryptocurrency required as a payment.

After the initialization, the contract allows one first action:

- **deposit:** with which the buyer deposits the required amount in the contract.

When the escrow is funded, one of the following two actions are possible:

- **pay:** with which the buyer can release the payment to the seller: in this case, the whole contract balance is transferred to the seller.
- **refund:** that allows the seller to accept a buyer reclaim: in this case, the contract issues a refund,transferring back the whole contract balance to the buyer.

Code

```
module escrow::escrow {
    use sui::tx_context::{TxContext, sender};
    use sui::object::{Self, UID};
    use sui::transfer;
    use sui::coin::{Self, Coin};
    use std::option::{Self, Option};

    const StateWaitDeposit: u64 = 0;
    const StateWaitRecipient: u64 = 1;
    const StateClosed: u64 = 2;

    const ErrorInsufficientAmount: u64 = 0;
    const ErrorInvalidState: u64 = 1;
    const ErrorUnauthorized: u64 = 2;

    struct Escrow<phantom T> has key {
        id: UID,
        state: u64,
        seller: address,
        buyer: address,
        amount: u64,
```

```
22          payment: Option<Coin<T>>,
23      }
24
25      public entry fun create_contract<T>(amount: u64, buyer: address, ctx: &mut TxContext) {
26          let contract = Escrow {
27              id: object::new(ctx),
28              state: StateWaitDeposit,
29              seller: sender(ctx), // The creator is the seller
30              buyer: buyer,
31              amount: amount,
32              payment: option::none<Coin<T>>(),
33          };
34          // The object is shared such that both buyer and seller can access it
35          transfer::share_object(contract);
36      }
37
38      public entry fun deposit<T>(contract: &mut Escrow<T>, money: Coin<T>,
39          ctx: &mut TxContext)
40      {
41          assert!(contract.state == StateWaitDeposit, ErrorInvalidState);
42          assert!(sender(ctx) == contract.buyer, ErrorUnauthorized);
43          assert!(coin::value<T>(&money) == contract.amount, ErrorInsufficientAmount);
44
45          option::fill(&mut contract.payment, money);
46          contract.state = StateWaitRecipient;
47      }
48
49      public entry fun pay<T>(contract: &mut Escrow<T>, ctx: &mut TxContext) {
50          assert!(contract.state == StateWaitRecipient, ErrorInvalidState);
51          assert!(sender(ctx) == contract.buyer, ErrorUnauthorized);
52
53          let money = option::extract<Coin<T>>(&mut contract.payment);
54          transfer::public_transfer(money, contract.seller);
55          contract.state = StateClosed;
56      }
57
58      public entry fun refund<T>(contract: &mut Escrow<T>, ctx: &mut TxContext) {
59          assert!(contract.state == StateWaitRecipient, ErrorInvalidState);
60          assert!(sender(ctx) == contract.seller, ErrorUnauthorized);
61
62          let money = option::extract<Coin<T>>(&mut contract.payment);
63          transfer::public_transfer(money, contract.buyer);
64          contract.state = StateClosed;
65      }
66  }
```

Appendix B

FM Operational Semantics

$$\frac{\mathrm{M} \ni t_1 \;\to\; t_1'}{\mathrm{M} \ni \mathtt{let}\, x = t_1 \,\mathtt{in}\, t_2 \;\to\; \mathtt{let}\, x = t_1' \,\mathtt{in}\, t_2} \text{ E-LET}$$

$$\frac{}{\mathrm{M} \ni \mathtt{let}\, x = v \,\mathtt{in}\, t_2 \;\to\; t_2\{x := v\}} \text{ E-LET2}$$

$$\frac{\mathrm{M} \ni t_i \;\to\; t_i'}{\mathrm{M} \ni \mathtt{pack}\, \mathrm{M.S}\,[\,\overline{v}, t_i, \overline{t}\,] \;\to\; \mathtt{pack}\, \mathrm{M.S}\,[\,\overline{v}, t_i', \overline{t}\,]} \text{ E-PACK}$$

$$\frac{k \in K \text{ is fresh}}{\mathrm{M} \ni \mathtt{pack}\, \mathrm{M.S}\,[\,\overline{v}\,] \;\to\; \mathtt{struct}\, \{k\}\, \mathrm{M.S}\,[\,\overline{v}\,]} \text{ E-PACKED}$$

$$\frac{\mathrm{M} \ni t_1 \;\to\; t_1'}{\mathrm{M} \ni \mathtt{unpack}\, \{\overline{x}\} = t_1 \,\mathtt{in}\, t_2 \;\to\; \mathtt{unpack}\, \{\overline{x}\} = t_1' \,\mathtt{in}\, t_2} \text{ E-UNPACK}$$

$$\frac{\mathtt{str}\, \mathrm{S}\, \{\, b\,,\ \overline{\mathrm{T}}\,\} \in \mathrm{M} \qquad |\overline{x}| = |\overline{v}| = |\overline{\mathrm{T}}|}{\mathrm{M} \ni \mathtt{unpack}\, \{\overline{x}\} = \mathtt{struct}\, \{k\}\, \mathrm{M.S}\,[\,\overline{v}\,] \,\mathtt{in}\, t_2 \;\to\; t_2\{\overline{x} := \overline{v}\}} \text{ E-UNPACKED}$$

$$\frac{\mathrm{M} \ni t_i \;\to\; t_i'}{\mathrm{M} \ni \mathtt{call}\, \mathrm{M}_2.\mathrm{F}\,[\,\overline{v}, t_i, \overline{t}\,] \;\to\; \mathtt{call}\, \mathrm{M}_2.\mathrm{F}\,[\,\overline{v}, t_i', \overline{t}\,]} \text{ E-CALL}$$

$$\frac{\mathtt{fun}\, \mathrm{F}\,(\,\overline{x} : \overline{\mathrm{T}}\,) : \mathrm{T}_r\, \{t_b\} \in \mathrm{M}_2}{\mathrm{M} \ni \mathtt{call}\, \mathrm{M}_2.\mathrm{F}\,[\,\overline{v}\,] \;\to\; \mathtt{exec}\, \mathrm{M}_2\; t_b\{\overline{x} := \overline{v}\}} \text{ E-CALLED}$$

$$\frac{\mathrm{M}_2 \ni t \;\to\; t'}{\mathrm{M} \ni \mathtt{exec}\, \mathrm{M}_2\; t \;\to\; \mathtt{exec}\, \mathrm{M}_2\; t'} \text{ E-EXEC} \qquad \frac{}{\mathrm{M} \ni \mathtt{exec}\, \mathrm{M}_2\; v \;\to\; v} \text{ E-EXECUTED}$$

$$\frac{}{\mathrm{M} \ni \mathtt{struct}\, \{k\}\, \mathrm{M.S}\,[\,\overline{v}\,].j \;\to\; v_j} \text{ E-SELECT}$$

$$\frac{\mathrm{M} \ni t_1 \;\to\; t_1'}{\mathrm{M} \ni \mathtt{if}\; t_1 \;\mathtt{then}\; t_2 \;\mathtt{else}\; t_3 \;\to\; \mathtt{if}\; t_1' \;\mathtt{then}\; t_2 \;\mathtt{else}\; t_3} \text{ E-IF}$$

$$\frac{n \neq 0}{\mathrm{M} \ni \mathtt{if}\; n \;\mathtt{then}\; t_2 \;\mathtt{else}\; t_3 \;\to\; t_2} \text{ E-IF-TRUE}$$

$$\frac{}{\mathrm{M} \ni \mathtt{if}\; 0 \;\mathtt{then}\; t_2 \;\mathtt{else}\; t_3 \;\to\; t_3} \text{ E-IF-FALSE}$$

$$\frac{\mathrm{M} \ni t \;\to\; t'}{\mathrm{M} \ni \mathtt{pub}\; t \;\to\; \mathtt{pub}\; t'} \text{ E-PUBLISH} \qquad \frac{}{\mathrm{M} \ni \mathtt{pub}\; v \;\to\; 0} \text{ E-PUBLISHED}$$

Appendix C

FM Typing Rules

STANDARD RULES

$$\frac{}{\mathrm{M} \ni \Gamma \vdash n : \mathrm{Int} \ \rhd \ \Gamma} \ \textsc{T-num}$$

$$\frac{\Gamma(x) = \mathrm{T}^{\circ} \qquad \neg\mathrm{IsLinear}(\mathrm{T})}{\mathrm{M} \ni \Gamma \vdash x : \mathrm{T} \ \rhd \ \Gamma} \ \textsc{T-var}$$

$$\frac{\Gamma(x) = \mathrm{T}^{\circ} \qquad \mathrm{IsLinear}(\mathrm{T})}{\mathrm{M} \ni \Gamma \vdash x : \mathrm{T} \ \rhd \ \Gamma\{x \mapsto \mathrm{T}^{\bullet}\}} \ \textsc{T-varL}$$

$$\frac{\mathrm{M} \ni \Gamma_1 \vdash t_1 : \mathrm{T}_1 \ \rhd \ \Gamma_2 \qquad \mathrm{M} \ni \Gamma_2, x : \mathrm{T}_1^{\circ} \vdash t_2 : \mathrm{T}_2 \ \rhd \ \Gamma_3, x : \mathrm{T}_1^{\downarrow}}{\mathrm{M} \ni \Gamma_1 \vdash \mathtt{let}\, x = t_1 \,\mathtt{in}\, t_2 : \mathrm{T}_2 \ \rhd \ \Gamma_3} \ \textsc{T-let}$$

$$\frac{\mathtt{str}\, \mathrm{S}\, \{\, b\,,\ \overline{\mathrm{T}}\,\} \in \mathrm{M} \qquad \mathrm{M} \ni \Gamma_1 \vdash \bar{t} : \overline{\mathrm{T}} \ \rhd \ \Gamma_2}{\mathrm{M} \ni \Gamma_1 \vdash \mathtt{pack}\, \mathrm{M.S}\, [\, \bar{t}\,] : \mathrm{M.S} \ \rhd \ \Gamma_2} \ \textsc{T-pack}$$

$$\frac{\begin{array}{c}\mathtt{str}\, \mathrm{S}\, \{\, b\,,\ \overline{\mathrm{T}}\,\} \in \mathrm{M} \\ \mathrm{M} \ni \Gamma_1 \vdash t_1 : \mathrm{M.S} \ \rhd \ \Gamma_2 \qquad \mathrm{M} \ni \Gamma_2, \overline{x} : \overline{\mathrm{T}}^{\circ} \vdash t_2 : \mathrm{T}_2 \ \rhd \ \Gamma_3, \overline{x} : \overline{\mathrm{T}}^{\downarrow}\end{array}}{\mathrm{M} \ni \Gamma_1 \vdash \mathtt{unpack}\, \{\overline{x}\} = t_1 \,\mathtt{in}\, t_2 : \mathrm{T}_2 \ \rhd \ \Gamma_3} \ \textsc{T-unpack}$$

$$\frac{\begin{array}{c}\mathtt{str}\, \mathrm{S}\, \{\, b\,,\ \overline{\mathrm{T}}\,\} \in \mathrm{M} \\ \mathrm{M} \ni \Gamma_1 \vdash x : \mathrm{M.S} \ \rhd \ \Gamma_2 \qquad \neg\mathrm{IsLinear}(\mathrm{T}_j)\end{array}}{\mathrm{M} \ni \Gamma_1 \vdash x.j : \mathrm{T}_j \ \rhd \ \Gamma_1} \ \textsc{T-selectX}$$

$$\frac{\mathtt{fun}\, \mathrm{F}\, (\, \overline{x} : \overline{\mathrm{T}}\,) : \mathrm{T}_r\, \{t_b\} \in \mathrm{M}_2 \qquad \mathrm{M} \ni \Gamma_1 \vdash \bar{t} : \overline{\mathrm{T}} \ \rhd \ \Gamma_2}{\mathrm{M} \ni \Gamma_1 \vdash \mathtt{call}\, \mathrm{M}_2.\mathrm{F}\, [\, \bar{t}\,] : \mathrm{T}_r \ \rhd \ \Gamma_2} \ \textsc{T-call}$$

$$\frac{\mathrm{M} \ni \Gamma_1 \vdash t_1 : \mathrm{Int} \ \rhd \ \Gamma_2 \qquad \begin{array}{c}\mathrm{M} \ni \Gamma_2 \vdash t_2 : \mathrm{T}_b \ \rhd \ \Gamma_3 \\ \mathrm{M} \ni \Gamma_2 \vdash t_3 : \mathrm{T}_b \ \rhd \ \Gamma_3\end{array}}{\mathrm{M} \ni \Gamma_1 \vdash \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : \mathrm{T}_b \ \rhd \ \Gamma_3} \ \textsc{T-if}$$

$$\frac{\mathrm{M} \ni \Gamma_1 \vdash t : \mathrm{T} \ \rhd \ \Gamma_2 \qquad \mathrm{IsLinear}(\mathrm{T})}{\mathrm{M} \ni \Gamma_1 \vdash \mathtt{pub}\ t : \mathrm{Int} \ \rhd \ \Gamma_2} \ \textsc{T-pub}$$

VECTOR RULES

$$\frac{}{\mathrm{M} \ni \Gamma \vdash \overline{0}_t : \overline{0}_T \ \rhd\ \Gamma}\ \textsc{T-vecz}$$

$$\frac{\mathrm{M} \ni \Gamma_1 \vdash \bar{t} : \overline{\mathrm{T}} \ \rhd\ \Gamma_2 \qquad \mathrm{M} \ni \Gamma_2 \vdash t' : \mathrm{T}' \ \rhd\ \Gamma_3}{\mathrm{M} \ni \Gamma_1 \vdash \bar{t}, t' : \overline{\mathrm{T}}, \mathrm{T}' \ \rhd\ \Gamma_3}\ \textsc{T-vec}$$

RUNTIME RULES

$$\frac{\mathsf{str}\ \mathrm{S}\,\{\, b\,,\ \overline{\mathrm{T}}\,\} \in \mathrm{M}_2 \qquad \mathrm{M} \ni \Gamma \vdash \overline{v} : \overline{\mathrm{T}} \ \rhd\ \Gamma \qquad k \in K}{\mathrm{M} \ni \Gamma \vdash \mathsf{struct}\ \{k\}\, \mathrm{M}_2.\mathrm{S}\,[\,\overline{v}\,] : \mathrm{M}_2.\mathrm{S} \ \rhd\ \Gamma}\ \textsc{T-struct}$$

$$\frac{\mathrm{M} \ni \Gamma \vdash v : \mathrm{M.S} \ \rhd\ \Gamma \qquad \begin{array}{c}\mathsf{str}\ \mathrm{S}\,\{\, b\,,\ \overline{\mathrm{T}}\,\} \in \mathrm{M} \\ \neg\mathrm{IsLinear}(\mathrm{T}_j)\end{array}}{\mathrm{M} \ni \Gamma \vdash v.j : \mathrm{T}_j \ \rhd\ \Gamma}\ \textsc{T-selectV}$$

$$\frac{\mathrm{M}_2 \ni \varnothing \vdash t : \mathrm{T} \ \rhd\ \varnothing}{\mathrm{M} \ni \Gamma \vdash \mathsf{exec}\ \mathrm{M}_2\ t : \mathrm{T} \ \rhd\ \Gamma}\ \textsc{T-exec}$$

WELL FORMATION

$$\frac{\forall\ \mathrm{T} \in \overline{\mathrm{T}}.\ \neg\mathrm{IsLinear}(\mathrm{T})}{\mathsf{str}\ \mathrm{S}\,\{\,\bot\,,\ \overline{\mathrm{T}}\,\}}\ \textsc{W-non-lin}$$

$$\frac{}{\mathsf{str}\ \mathrm{S}\,\{\,\top\,,\ \overline{\mathrm{T}}\,\}}\ \textsc{W-lin}$$

$$\frac{\mathrm{M} \ni \varnothing, \overline{x} : \overline{\mathrm{T}}^{\circ} \vdash t_b : \mathrm{T}_r \ \rhd\ \varnothing, \overline{x} : \overline{\mathrm{T}}^{\downarrow}}{\mathrm{M} \vdash \mathsf{fun}\ \mathrm{F}\,(\,\overline{x} : \overline{\mathrm{T}}\,) : \mathrm{T}_r\,\{t_b\}}\ \textsc{W-fun}$$

$$\frac{\overline{\mathrm{M} \vdash \mathrm{FD}} \qquad \overline{\mathrm{SD}}}{\vdash \mathrm{M}\,\{\,\overline{\mathrm{SD}}\,,\ \overline{\mathrm{FD}}\,\}}\ \textsc{W-module}$$

Appendix D

FM Agda Fragments

D.1 Operational Semantics

```
data _∋_⇒_ where
    Elet :
          (ev : M ∋ t1 ⇒ t1')
        --------------------
        → M ∋ (Let t1 In t2) ⇒ (Let t1' In t2)

    Elet2 :
          (v : Value t1)
        --------------------
        → M ∋ (Let t1 In t2) ⇒ beta-red (t1 V.:: V.[]) t2

    Epack :
          {s : Fin Ns}
          {ts ts' : Vec Term Nsf}
        → (ev : M ∋ ts ⇒v ts')
        --------------------
        → M ∋ (pack (sId M s) ts) ⇒ (pack (sId M s) ts')

    Epacked :
          {s : Fin Ns}
          {ts : Vec Term Nsf}
        → (k : K)
        → (vs : ValueV ts)
        --------------------
        → M ∋ (pack (sId M s) ts) ⇒ (struct k (sId M s) ts)

    Eunpack :
          (ev : M ∋ t1 ⇒ t1')
        --------------------
        → M ∋ (unpack t1 In t2) ⇒ (unpack t1' In t2)

```

```
32      Eunpacked :
33            {k : K}
34            {s : Fin Ns}
35            {ts : Vec Term Nsf}
36          → (vs : ValueV ts)
37          --------------------
38          → M ∋ (unpack (struct k (sId M s) ts) In t2) ⇒ beta-red ts t2
39
40      Ecall :
41            {ts ts' : Vec Term Nfa}
42          → (ev : M ∋ ts ⇒v ts')
43          --------------------
44          → M ∋ (call fid ts) ⇒ (call fid ts')
45
46      Ecalled :
47            {f : Fin Nf}
48          → {ts : Vec Term Nfa}
49          → (vs : ValueV ts)
50          --------------------
51          → M ∋ (call (fId M2 f) ts) ⇒ exec M2 (beta-red ts (toRun (gBody M2 f)))
52
53      Eexec :
54            (ev : M2 ∋ t ⇒ t')
55          --------------------
56          → M ∋ (exec M2 t) ⇒ (exec M2 t')
57
58      Eexecuted :
59            (v : Value t)
60          --------------------
61          → M ∋ (exec M2 t) ⇒ t
62
63      Eif :
64            (ev : M ∋ t1 ⇒ t1')
65          --------------------
66          → M ∋ (if t1 then t2 else t3) ⇒ (if t1' then t2 else t3)
67
68      Eiftrue : {g : ℕ}
69          → (nz : ¬ g ≡ 0)
70          --------------------
71          → M ∋ (if (num g) then t2 else t3) ⇒ t2
72
73      Eiffalse :
74          M ∋ (if (num 0) then t2 else t3) ⇒ t3
75
76      Eselect :
77            {k : K} {s  : Fin Ns}
```

```
78            {ts : Vec Term Nsf}
79            {j  : Fin Nsf}
80          → (vs : ValueV ts)
81          --------------------
82          → M ∋ (struct k (sId M s) ts) · j ⇒ V.lookup ts j
83
84      Epub :
85            M ∋ t ⇒ t'
86          --------------------
87          → M ∋ (pub t) ⇒ (pub t')
88
89      Epub2 :
90            (v : Value t)
91          --------------------
92          → M ∋ (pub t) ⇒ num 0
```

D.2 Typing Rules

```
data HasType M where
    Tnum :
        --------------------
        HasType M Γ (num n) Tint Γ

    Tvar :
          (htx : HasTypeX Γ1 x T Γ2)
        --------------------
        → HasType M Γ1 (var x) T Γ2

    Tlet :
          (ht  : HasType M Γ1 t1 T1 Γ2)
        → (hti : HasTypeI M Γ2 (T1 V.:: V.[]) t2 T2 Γ3)
        --------------------
        → HasType M Γ1 (Let t1 In t2) T2 Γ3

    Tpack :
          {s : Fin Ns}
        → {ts : Vec Term Nsf}
        → (htv : HasTypeV M Γ1 ts (gFieldsT M s) Γ2)
        --------------------
        → HasType M Γ1 (pack (sId M s) ts) (Tst (sId M s)) Γ2

    Tstruct :
          {k : K} {s : Fin Ns}
          {M2 : Fin Nm}
        → {ts : Vec Term Nsf}
        → (vs : ValueV ts)
        → (htv : HasTypeV M Γ ts (gFieldsT M2 s) Γ)
        --------------------
        → HasType M Γ (struct k (sId M2 s) ts) (Tst (sId M2 s)) Γ

    Tcall :
          {M2 : Fin Nm}
        → {f : Fin Nf}
        → {ts : Vec Term Nfa}
        → (htv : HasTypeV M Γ1 ts (gArgsT M2 f) Γ2)
        --------------------
        → HasType M Γ1 (call (fId M2 f) ts) (gRetT M2 f) Γ2

    Tunpack :
          {s : Fin Ns}
        → (ht  : HasType  M Γ1 t1 (Tst (sId M s)) Γ2)
        → (hti : HasTypeI M Γ2 (gFieldsT M s) t2 T2 Γ3)
```

```
45          --------------------
46          → HasType M Γ1 (unpack t1 In t2) T2 Γ3
47
48      Texec :
49            {M2 : Fin Nm}
50          → HasType M2 [] t T []
51          --------------------
52          → HasType M Γ (exec M2 t) T Γ
53
54      Tif :
55            (ht1 : HasType M Γ1 t1 Tint Γ2)
56          → (ht2 : HasType M Γ2 t2 T    Γ3)
57          → (ht3 : HasType M Γ2 t3 T    Γ3)
58          --------------------
59          → HasType M Γ1 (if t1 then t2 else t3) T Γ3
60
61      TselX :
62            {s : Fin Ns}
63            {j : Fin Nsf}
64          → (htx : HasTypeX Γ1 x (Tst (sId M s)) Γ2)
65          → (nLin : ¬ IsLinear (V.lookup (gFieldsT M s) j))
66          --------------------
67          → HasType M Γ1 ((var x) · j) (V.lookup (gFieldsT M s) j) Γ1
68
69      TselV :
70            {s : Fin Ns}
71            {j : Fin Nsf}
72          → (v  : Value t)
73          → (ht : HasType M Γ t (Tst (sId M s)) Γ)
74          → (nLin : ¬ IsLinear (V.lookup (gFieldsT M s) j))
75          --------------------
76          → HasType M Γ (t · j) (V.lookup (gFieldsT M s) j) Γ
77
78      Tpub :
79            (ht   : HasType M Γ1 t T Γ2)
80          → (yLin : IsLinear T)
81          --------------------
82          → HasType M Γ1 (pub t) Tint Γ2
```

D.3 Proof of Type Preservation Lemma

```
type-preservation :
    {Γ1 Γ2 : UEnv Δ}
    → HasType M Γ1 t T Γ2
    → M ∋ t ⇒ t'
    → HasType M Γ1 t' T Γ2

type-preservation-vec :
    {Γ1 Γ2 : UEnv Δ}
    {ts ts' : Vec Term n} {Ts : Vec Type n}
    → HasTypeV M Γ1 ts Ts Γ2
    → M ∋ ts ⇒v ts'
    → HasTypeV M Γ1 ts' Ts Γ2

type-preservation (Tlet ht hti) (Elet ev)
    = Tlet (type-preservation ht ev) hti
type-preservation (Tlet ht hti) (Elet2 v)
    rewrite htval⇒Γ1≡Γ2 v ht
    = substi-multi (ht T:: T[]) (v V:: V[]) hti
type-preservation (Tpack htv) (Epack evf)
    = Tpack (type-preservation-vec htv evf)
type-preservation (Tpack htv) (Epacked k vs)
    rewrite htval⇒Γ1≡Γ2-vec vs htv
    = Tstruct vs (value-type-vec vs htv)
type-preservation (Tunpack ht hti) (Eunpack ev)
    = Tunpack (type-preservation ht ev) hti
type-preservation (Tunpack (Tstruct vs' htv) hti) (Eunpacked vs)
    rewrite htval⇒Γ1≡Γ2-vec vs htv
    = substi-multi htv vs hti
type-preservation (Tcall htv) (Ecall eva)
    = Tcall (type-preservation-vec htv eva)

-- Here we use the hypothesis of well-formedness of the function we are calling
type-preservation (Tcall {M2 = M2} {f = f} htv) (Ecalled vs)
    rewrite htval⇒Γ1≡Γ2-vec vs htv
    = Texec (substi-multi htv vs (wellHti W M2 f))

type-preservation (Texec ht) (Eexec ev)
    = Texec (type-preservation ht ev)
type-preservation (Texec ht) (Eexecuted v)
    = value-type v ht
type-preservation (Tif ht1 ht2 ht3) (Eif ev)
    = Tif (type-preservation ht1 ev) ht2 ht3
type-preservation (Tif ht1 ht2 ht3) (Eiftrue nz)
    rewrite htval⇒Γ1≡Γ2 Vnum ht1 = ht2
```

```
type-preservation (Tif ht1 ht2 ht3) Eiffalse
    rewrite htval⇒Γ1≡Γ2 Vnum ht1 = ht3
type-preservation (TselV {j = j} _ (Tstruct vs htv) nLin) (Eselect _)
    = htvLookup vs htv j
type-preservation (Tpub ht yLin) (Epub ev)
    = Tpub (type-preservation ht ev) yLin
type-preservation (Tpub ht yLin) (Epub2 v)
    rewrite htval⇒Γ1≡Γ2 v ht = Tnum


type-preservation-vec (ht T:: htv) (E[ ev ] vs)
    rewrite htval⇒Γ1≡Γ2-vec vs htv
    = (type-preservation ht ev) T:: (value-type-vec vs htv)
type-preservation-vec (ht T:: htv) (t E:: evv)
    = ht T:: type-preservation-vec htv evv

```

D.4 Proof of Resource Preservation Lemma

```
Rsafety :
      All tIsIf⇒Rt2↭Rt3 t
    → HasType M Γ1 t T Γ2
    → (ev : M ∋ t ⇒ t')
    → RI ev L.++ R t ↭ RU ev L.++ R t'
Rsafety-vec :
    {ts ts' : Vec Term n} {Ts : Vec Type n}
    → AllV tIsIf⇒Rt2↭Rt3 ts
    → HasTypeV M Γ1 ts Ts Γ2
    → (ev : M ∋ ts ⇒v ts')
    → RIv ev L.++ Rv ts ↭ RUv ev L.++ Rv ts'

Rsafety (all-let p a1 a2) (Tlet {t1 = t1} {t2 = t2} ht hti) (Elet {t1' = t1'} ev)
    = lemma↭3 (R t2) (RI ev) (RU ev) (Rsafety a1 ht ev)
Rsafety a (Tlet {t1 = t1} {t2 = t2} ht hti) (Elet2 v)
    =
    begin
    R t1 L.++ R t2                  ↭⟨ ++-comm (R t1) (R t2) ⟩
    R t2 L.++ R t1                  ↭⟨ ++⁺ˡ (R t2) (↭-sym (++-identityʳ (R t1))) ⟩
    R t2 L.++ (R t1 L.++ L.[])      ↭⟨ ↭-sym ↭1 ⟩
    R (shift-back 0 (subst 0 t1 t2)) ∎
    where
        ↭1 = Rsubsti-multi (ht T:: T[]) (v V:: V[]) hti

Rsafety (all-pack p av) (Tpack htv) (Epack ev) = Rsafety-vec av htv ev
Rsafety a (Tpack {ts = ts} htv) (Epacked {M = M} {s = s} k x) with tyIsLin (Tst (sId M s))
... | yes yLin = refl
... | no  nLin = refl
Rsafety (all-call p av) (Tcall htv) (Ecall ev) = Rsafety-vec av htv ev

-- Here we use the fact that the function is well formed, and that the gBody
--   of the function is an LTerm (a language term).
-- LTerms are a subset of Terms
Rsafety a (Tcall {M2 = M2} {f = f} {ts = ts} htv) (Ecalled vs)
    = begin
    Rv ts               ↭⟨ refl ⟩
    L.[] L.++ Rv ts
        ↭⟨ ++⁺ʳ (Rv ts) (↭-sym (↭-reflexive (Rlterm≡[] (gBody M2 f)))) ⟩
    R (toRun (gBody M2 f)) L.++ Rv ts
        ↭⟨ ↭-sym (Rsubsti-multi htv vs (wellHti W M2 f)) ⟩
    R (beta-red ts (toRun (gBody M2 f))) ∎

Rsafety (all-unpack p a1 a2) (Tunpack {t1 = t1} {t2 = t2} ht hti) (Eunpack ev)
    = lemma↭3 (R t2) (RI ev) (RU ev) (Rsafety a1 ht ev)
```

```
Rsafety a (Tunpack (Tstruct _ htv) hti) (Eunpacked {M = M} {t2 = t2} {k = k} {s = s} {ts = ts} vs)
    with tyIsLin (Tst (sId M s))
... | yes yLin = begin
    k L.:: (Rv ts L.++ R t2)   ↭⟨ prep k (++-comm (Rv ts) (R t2)) ⟩
    k L.:: (R t2 L.++ Rv ts)   ↭⟨ prep k (↭-sym (Rsubsti-multi htv vs hti)) ⟩
    k L.:: R (beta-red ts t2)  ∎
... | no  nLin = begin
    Rv ts L.++ R t2     ↭⟨ ++-comm (Rv ts) (R t2) ⟩
    R t2 L.++ Rv ts     ↭⟨ ↭-sym (Rsubsti-multi htv vs hti) ⟩
    R (beta-red ts t2) ∎

Rsafety (all-exec p a) (Texec ht) (Eexec ev) = Rsafety a ht ev
Rsafety a (Texec ht) (Eexecuted v) = refl
Rsafety (all-if p a1 a2 a3) (Tif {t2 = t2} ht1 ht2 ht3) (Eif ev)
    = lemma↭3 (R t2) (RI ev) (RU ev) (Rsafety a1 ht1 ev)
Rsafety a (Tif ht1 ht2 ht3) (Eiftrue nz) = refl

-- Here we use the tIsIf⇒Rt2↭Rt3 property
Rsafety (all-if p a1 a2 a3) (Tif {t3 = t3} ht1 ht2 ht3) Eiffalse = p refl

Rsafety a (TselV {j = j} v (Tstruct _ htv) nLin) (Eselect vs)
    rewrite RnLin (htvLookup vs htv j) nLin (vLookup j vs) = refl
Rsafety (all-pub p a) (Tpub ht yLin) (Epub ev) = Rsafety a ht ev
Rsafety a (Tpub {t = t} ht yLin) (Epub2 v) rewrite l++[]≡l (R t) = refl

Rsafety-vec (all-vec:: a av) (_T::_ {ts = ts} ht htv) (E[ ev ] vs)
    = lemma↭3 (Rv ts) (RI ev) (RU ev) (Rsafety a ht ev)
Rsafety-vec (all-vec:: a av) (ht T:: htv) (_E::_ {ts = ts} {ts' = ts'} t evs) =
    begin
    RIv evs L.++ (R t L.++ Rv ts)   ↭⟨ ++⁺ˡ (RIv evs) (++-comm (R t) (Rv ts)) ⟩
    RIv evs L.++ (Rv ts L.++ R t)   ↭⟨ ↭-sym (++-assoc (RIv evs) (Rv ts) (R t)) ⟩
    (RIv evs L.++ Rv ts) L.++ R t   ↭⟨ ++⁺ʳ (R t) (Rsafety-vec av htv evs) ⟩
    (RUv evs L.++ Rv ts') L.++ R t  ↭⟨ ++-assoc (RUv evs) (Rv ts') (R t) ⟩
    RUv evs L.++ (Rv ts' L.++ R t)  ↭⟨ ++⁺ˡ (RUv evs) (++-comm (Rv ts') (R t)) ⟩
    RUv evs L.++ (R t L.++ Rv ts')  ∎
```

Bibliography

[1] Guillaume Allais. "Typing with leftovers: a mechanization of Intuitionistic Multiplicative-Additive Linear Logic". In: *23rd International Conference on Types for Proofs and Programs*. 2019, pp. 1–22.

[2] Diem Association. *The Diem Blockchain*. 2020. URL: https://developers.diem.com/docs/technical-papers/the-diem-blockchain-paper/.

[3] Massimo Bartoletti. "SoK: Smart Contract Languages, Lorenzo Benetollo, Michele Bugliesi, Silvia Crafa, Giacomo Dal Sasso, Roberto Pettinau, Andrea Pinna, Mattia Piras, Sabina Rossi, Stefano Salis, Alvise Spanò, Viacheslav Tkachenko, Roberto Tonelli, Roberto Zunino". In: *Not yet published* (2024).

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

[5] Massimo Bartoletti et al. "A formal model of Algorand smart contracts". In: *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I 25*. Springer. 2021, pp. 93–114.

[6] Sam Blackshear et al. "Move: A language with programmable resources". In: *Libra Assoc* (2019), p. 1.

[7] Sam Blackshear et al. "Resources: A safe language abstraction for money". In: *arXiv preprint arXiv:2004.05106* (2020).

[8] Sam Blackshear et al. "The Move Borrow Checker". In: *arXiv preprint arXiv:2205.05181* (2022).

[9] Ana Bove, Peter Dybjer, and Ulf Norell. "A brief overview of Agda–a functional language with dependent types". In: *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*. Springer. 2009, pp. 73–78.

[10] Huashan Chen et al. "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses". In: *ACM Computing Surveys (CSUR)* 53.3 (2020), pp. 1–43.

[11] Zehao Chen et al. "ChainKV: A Semantics-Aware Key-Value Store for Ethereum System". In: *Proceedings of the ACM on Management of Data* 1.4 (2023), pp. 1–23.

[12] Silvia Crafa, Matteo Di Pirro, and Elena Zucca. “Is solidity solid enough?” In: *Financial Cryptography and Data Security: FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*. Springer. 2020, pp. 138–153.

[13] Massimo Di Pierro. “What Is the Blockchain?” In: *Computing in Science & Engineering* 19.5 (2017), pp. 92–95. DOI: 10.1109/MCSE.2017.3421554.

[14] Sui Foundation. *Why We Created Sui Move*. 2022. URL: https://blog.sui.io/why-we-created-sui-move.

[15] Péter Hegedűs. “Towards analyzing the complexity landscape of solidity based ethereum smart contracts”. In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 2018, pp. 35–39.

[16] Lioba Heimbach et al. “Defi and nfts hinder blockchain scalability”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2023, pp. 291–309.

[17] Tharaka Hewa, Mika Ylianttila, and Madhusanka Liyanage. “Survey on blockchain based smart contracts: Applications, opportunities and challenges”. In: *Journal of network and computer applications* 177 (2021), p. 102857.

[18] Per Martin-Löf. “Constructive mathematics and computer programming”. In: *Studies in Logic and the Foundations of Mathematics*. Vol. 104. Elsevier, 1982, pp. 153–175.

[19] Michael Nofer et al. “Blockchain”. In: *Business & Information Systems Engineering* 59 (2017), pp. 183–187.

[20] Marco Patrignani and Sam Blackshear. “Robust safety for move”. In: *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. IEEE. 2023, pp. 308–323.

[21] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. Cambridge, MA: The MIT Press, 2005. ISBN: 0262162288.

[22] Giacomo Dal Sasso. *sui-use-cases*. 2024. URL: https://github.com/singds/sui-use-cases/tree/thesis.

[23] Giacomo Dal Sasso. *u-Agda-Move*. 2024. URL: https://github.com/singds/u-Agda-Move/tree/thesis.

[24] Move core team. *The Move Book*. https://move-language.github.io/move and https://github.com/move-language/move/tree/main/language/documentation/book.

[25] The MystenLabs Team. *The Sui Smart Contracts Platform*. 2023. URL: https://docs.sui.io/paper/sui.pdf.

[26] Yu Wang et al. “GradingShard: A new sharding protocol to improve blockchain throughput”. In: *Peer-to-Peer Networking and Applications* (2023), pp. 1–13.

[27] Adam Welc and Sam Blackshear. “Sui Move: Modern Blockchain Programming with Objects”. In: *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. 2023, pp. 53–55.

[28] Uma Zalakain and Ornela Dardha. "π with Leftovers: A Mechanisation in Agda". In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer. 2021, pp. 157–174.

[29] Zhuo Zhang et al. "Demystifying Exploitable Bugs in Smart Contracts". In: ICSE. 2023.

[30] Jingyi Emma Zhong et al. "The move prover". In: *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*. Springer. 2020, pp. 137–150.

[31] Weiqin Zou et al. "Smart contract development: Challenges and opportunities". In: *IEEE Transactions on Software Engineering* 47.10 (2019), pp. 2084–2106.