Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica Magistrale

Un confronto tra i linguaggi di programmazione per smart contract: Move e Solidity

Relatore:
Chiar.mo Prof.
Cosimo Laneve
Correlatori:
Chiar.mo Prof.
Claudio Sacerdoti Coen
Dott.ssa Adele Veschetti

Presentata da:
Marco Castiglioni

III Appello di Laurea
Anno Accademico 2018-2019

Indice

Capitolo 1

Introduzione

L'impresa statunitense Facebook Inc. ha deciso recentemente di espandersi nel campo delle criptovalute iniziando a sviluppare Libra, un sistema finanziario basato sulla tecnologia blockchain, con l'obiettivo di creare una moneta utilizzata a livello internazionale. Fin dall'annuncio, viste le problematiche relative alla privacy che hanno riguardato l'azienda americana, Libra è stata al centro di dibattiti e critiche portando con sè non pochi dubbi circa la possibilità di rivoluzionare il sistema dei pagamenti online. Certamente però, questo progetto può essere considerato una pietra miliare nell'evoluzione tecnologica della blockchain, dal momento che verrà integrato in piattaforme social utilizzate dalla maggior parte della popolazione mondiale, e, perciò, se il progetto dovesse avere successo, potrebbe effettivamente diventare la prima criptovaluta mainstream. In questa tesi, tuttavia, non ci si concentrerà sugli aspetti sociali e finanziari ma su quelli tecnici, andando ad analizzare l'infrastruttura della blockchain di Libra e, soprattutto, il nuovo linguaggio per smart contract che verrà rilasciato con essa, Move. Il progetto di Libra è ancora in fase di sviluppo, perciò non è facile reperire lavori relativi a questo argomento. La maggior parte dei documenti esterni citati in questo elaborato sono gli articoli direttamente rilasciati dalla Libra Association. Si noti infine che il codice che verrà mostrato è stato scritto con la versione di Move di inizio 2020, una versione intermedia chiamata Move

IR. Il rilascio ufficiale di Libra è stimato per fine 2020.

1.1 Svolgimento del lavoro e contributo

Il lavoro si è svolto principalmente in tre fasi:

1. Una prima fase di studio ed analisi della blockchain Libra e del linguaggio Move. La documentazione consultata è composta dagli articoli tecnici rilasciati dagli sviluppatori di Calibra, dal community forum, dove sono quotidianamente trattati i temi di sviluppo attuali, e dal GitHub ufficiale, il quale, dato che si tratta di un progetto in corso d'opera, è ancora in costante aggiornamento.

2. Una seconda fase in cui sono stati sviluppati alcuni programmi in Move lanciandoli in locale con la testnet di Libra, una versione di prova della blockchain composta da un solo nodo validatore. Successivamente è stato sviluppato un custom token in Move, chiamato PaneToken in cui si è cercato di imitare il pattern di programmazione seguito dagli sviluppatori di Libra. Non è al momento possibile testare il token in un ambiente pubblicamente condiviso, in primis perché ovviamente Libra non è stata ancora rilasciata e in secundis perchè non è stata ancora resa disponibile una metodologia di rilascio per moduli sviluppati da terzi. Si può comunque considerare un buon contributo per chi volesse cimentarsi nella scrittura di moduli Move, visto lo sviluppo di un marketplace ufficiale che verrà rilasciata prossimamente.

3. Nell'ultima fase si è confrontato il linguaggio Move con il linguaggio di programmazione per smart contract di Ethereum, Solidity. Si è successivamente cercato di costruire l'ambiente di Move con Solidity riscrivendo l'implementazione di PaneToken e sviluppando dei metaprogrammi di simulazione della struttura di dati risorsa. Siccome il linguaggio Solidity è particolarmente flessibile, ma non protegge lo sviluppatore da errori di sicurezza nello sviluppo di custom token, si è

deciso di sviluppare questo sistema in grado di descrivere un pattern per l'implementazione di asset protetti da operazioni di copia, riutilizzo e perdita non volute.

1.2 Struttura dell'elaborato

I prossimi capitoli dell'elaborato verranno strutturati in questo modo:

- Un capitolo incentrato su Libra. Verranno descritte: l'infrastruttura della blockchain, il modello logico, le strutture di dati utilizzate per rappresentare lo stato e la storia del ledger, l'implementazione delle transazioni e l'algoritmo di consenso utilizzato per approvarle e, infine, verrà effettuata un'analisi sulla Libra Coin e sui Libra Account.

- Un capitolo incentrato sul linguaggio di programmazione per smart contract proposto da Libra, Move. Questo capitolo sarà diviso in due parti: nella prima verrà descritto il sistema di tipi del linguaggio, concentrandosi principalmente sulla novità proposta, le risorse, una struttura di dati con certe proprietà di sicurezza, e su come i controlli su queste vengono attuati, attraverso il verificatore del bytecode. Nella seconda parte, invece, si effettuerà un'analisi approfondita di Move, confrontandolo con il linguaggio di smart contract attualmente più utilizzato sul mercato, Solidity.

- Un capitolo in cui si presenterà il contributo: un metaprogramma scritto in Solidity che simula il comportamento delle risorse di Move ed effettua i controlli di sicurezza per evitare che queste strutture di dati speciali vengano perse, copiate o riutilizzate.

- Un capitolo conclusivo in cui si riassumerà il contenuto dell'elaborato e si presenteranno alcuni sviluppi futuri.

Capitolo 2

Libra

La Libra blockchain [2] è un database decentralizzato sviluppato con lo scopo di creare una criptovaluta, la Libra Coin, con bassa volatilità che "potrà essere utilizzata come efficace strumento di scambio per miliardi di persone". La blockchain è di tipo *permissioned* [33], è scritta in Rust ed è stata proposta dall'azienda americana Facebook, Inc. Il progetto viene gestito da un'organizzazione non a scopo di lucro chiamata Libra Association [34] con sede a Ginevra, in Svizzera e vi partecipano come investitori diverse aziende specializzate in pagamenti online, telecomunicazioni, tecnologia, gruppi di venture capital e altre associazioni senza scopo di lucro. Questo consorzio di organizzazioni formerà i nodi validatori (*validators*) di Libra, i quali avranno il potere di decidere come modificare lo stato del *ledger* e quali transazioni potranno essere considerate valide. Ogni gruppo di investimento rappresenterà un singolo nodo validatore, con un potere di decisione pari all'1%. Il numero di nodi validatori al momento del lancio di Libra sarà infatti 100. In questo capitolo ci concentreremo sugli elementi fondamentali della Libra blockchain, descriveremo:

- Il modello logico.
- Le transazioni, come sono strutturate e come vengono emesse.
- La struttura del database replicato.

- L'algoritmo di consenso utilizzato.
- Le motivazioni per cui la Libra Coin è considerata una criptovaluta con bassa volatilità.

2.1 Modello logico di Libra

Tutti i dati nella Libra Blockchain sono immagazzinati in un singolo database. Il database utilizza un intero a 64 bit per identificare la sua versione che corrisponde al numero di transazioni totali che il sistema ha eseguito. Ad ogni versione i, il database contiene una tupla (T_i,O_i,S_i) che rappresenta la transazione, l'output della transazione e lo stato del ledger. Data una funzione determinista *Apply*, si esegue la transazione T_i su un stato del ledger S_{i-1} che produrrà un output O_i e un nuovo stato S_i: $Apply(S_{i-1},T_i) \Rightarrow \langle O_i,S_i \rangle$.

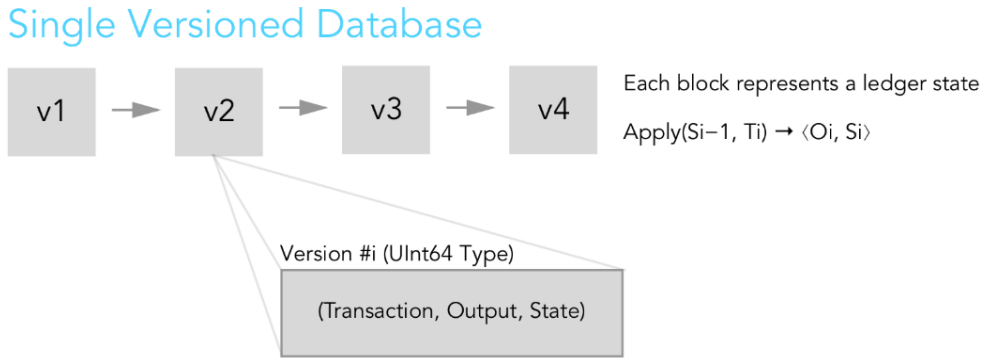

Figura 2.1: Transizioni di stato del ledger.

2.1.1 Lo stato del ledger

Lo stato del ledger rappresenta il contenuto di informazioni sulla totalità dell'ecosistema di Libra, includendo la quantità di Libra Coin e di tutte le altre risorse possedute da ciascun utente. Ogni validatore conosce lo stato del ledger e per poter eseguire una nuova transazione deve essere aggiornato all'ultima versione disponibile. Più precisamente, ogni nodo validatore può:

- Eseguire una transazione sull'ultima versione dello stato del ledger.
- Rispondere ad una query di un client, relativa ad una o più transazioni presenti nella storia del ledger. La storia può essere verificata da tutti i client, garantendo così trasparenza e tracciabilità.

Lo stato del ledger è decodificato come una mappa chiavi *account address* $\Rightarrow$ *account value*, dove per il valore si intende la collezione di moduli e risorse Move pubblicate sotto quel determinato account. Si rimanda al capitolo successivo per maggiori dettagli riguardanti il linguaggio di programmazione Move. Ogni account è un valore a 256 bit. Per creare un nuovo account, un utente genera una coppia (vk,sk), chiave di verifica e chiave di firma (*signature*). Si utilizza l'hash crittografico di una chiave pubblica di verifica vk come indirizzo di un account. $a = \mathrm{H}(vk)$. A livello fisico un account è una mappa ordinata di *access path* a diversi array di byte. Un nuovo Libra account, implementato tramite modulo di Move [7], può essere creato con la procedura create_account(a). La risorsa account è definita nel modulo LibraAccount.mvir come T.

```
module LibraAccount{
  // Ogni account di Libra ha una risorsa LibraAccount.T
  resource T {
    // La chiave di autenticazione
    // Puo' essere differente rispetto alla chiave utilizzata per
       creare l'account la prima volta
    authentication_key: bytearray,
    // La quantita' di LibraCoin posseduta dall'account
    balance: LibraCoin.T,
    // Se vera, l'autorita' per richiedere la chiave di
       autenticazione risiede da qualche altra parte
    delegated_key_rotation_capability: bool,
    // Se vera, l'autorita' per ritirare fondi da questo account
       risiede da un'altra parte
    delegated_withdrawal_capability: bool,
```

```
        // Handle per eventi ricevuti
        received_events: Self.EventHandle<Self.ReceivedPaymentEvent>,
        // Handle per eventi emessi
        sent_events: Self.EventHandle<Self.SentPaymentEvent>,
        // L'attuale numero di sequenza
        // Ogni volta che una nuova transazione viene emessa, il
           numero aumenta di 1
        sequence_number: u64,
        // Generatore di handle per eventi
        event_generator: Self.EventHandleGenerator,
    }
    ...
}
```

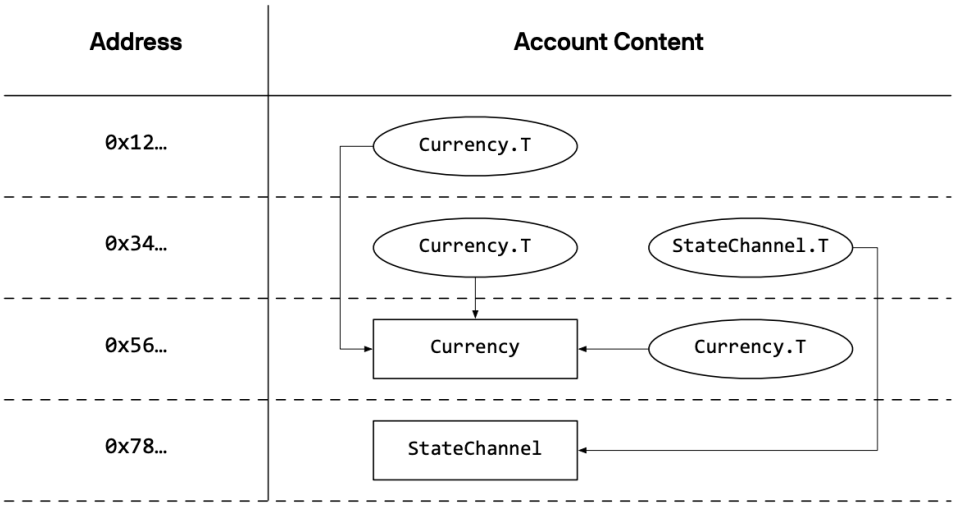

Figura 2.2: Un esempio di stato del ledger con quattro account.

Nel diagramma mostrato, gli ovali rappresentano risorse, i quadrati rappresentano moduli. La freccia che punta da una risorsa ad un modulo indica che tale risorsa è stata dichiarata nel modulo puntato. Per accedere alla risorsa 0x56.Currency.T posseduta dall'account 0x12 il client dovrà richie-

dere 0x12/resources/0x56.Currency.T. La versione di Move IR disponibile al momento costringe il programmatore ad eseguire istruzioni di basso livello, perciò per importare moduli o risorse da un account esterno si utilizza questo stesso approccio anche nel codice Move.

2.2 Transazioni

Una transazione è un wrapper autenticato attorno ad un programma Move. Questo programma è composto da una sequenza di istruzioni bytecode. Come definito in precedenza, le transazioni agiscono sullo stato del ledger corrente, producendo un nuovo stato se l'esecuzione termina con successo. Una transazione è composta dai seguenti dati:

- Indirizzo del mittente, ovvero l'indirizzo dell'account che invia la transazione. La macchina virtuale di Move legge il numero di sequenza dell'account, la sua chiave di autenticazione e il bilancio di Libra Coin dalla risorsa LibraAccount.T presente all'indirizzo specificato.
- La chiave pubblica del mittente, che dovrà corrispondere alla chiave privata utilizzata per firmare la transazione, in modo tale da poter autenticare l'account.
- Il programma, che viene scritto come *transaction script* in codice Move e successivamente compilato in una sequenza di istruzioni bytecode.
- Un costo in gas che il mittente è disposto a spendere con questa transazione.
- Il massimo costo in gas che il mittente può spendere in totale.
- Un numero di sequenza che deve essere uguale a quello presente nella risorsa LibraAccount.T.

L'esecuzione di una transazione è composta da sei fasi distinte all'interno della macchina virtuale ed è separata dall'aggiornamento dello stato del ledger. Le fasi sono le seguenti:

1. Controllo della firma.
2. Lancio della la procedura *prologue()*.
3. Verifica di transaction script e dei moduli.
4. Pubblicazione del modulo.
5. Lancio del transaction script.
6. Lancio della procedura *epilogue()*.

La prima fase controlla che la firma sulla transazione soddisfi, faccia *matching*, la chiave pubblica del mittente. Nella seconda fase, la funzione prologue() autentica il mittente della transazione e controlla che possieda una quantità di Libra Coin sufficiente per pagare il massimo numero di gas specificato sulla transazione. Successivamente, controlla che la transazione non sia una replica di un'altra transazione eseguita precedentemente, attraverso il numero di sequenza. Tutti questi controlli sono implementati in Move, tramite la funzione *prologue()*, definita nel modulo Libra_account. Non viene consumato gas durante l'esecuzione di questa procedura. Più specificamente:

- Controlla che l'hash della chiave pubblica del mittente sia uguale alla chiave di autenticazione salvata nella risorsa LibraAccount.T.
- Controlla con la riga di codice: max_transaction_fee = move(gas_price) * move(gas_units) che il mittente possieda una quantità di gas sufficiente. Lo scopo è il cercare di evitare che la macchina virtuale esegua un halt durante l'esecuzione della transazione perché il gas è terminato.
- Controlla che il numero di sequenza della transazione sia uguale a quello dell'account del mittente. Senza questo controllo, si potrebbe verificare un meccanismo di doppia spesa dove un utente malevolo potrebbe appropriarsi illegalmente di Libra Coin replicando vecchie transazioni.

Una volta conclusa l'esecuzione della procedura prologue(), la macchina virtuale performa controlli di sicurezza sul codice Move tramite il verificatore del

bytecode. Questi controlli verranno analizzati più nel dettaglio nel capitolo 3 di questo elaborato. Nella quarta fase, i moduli importati da una transazione verranno pubblicati sotto l'account del mittente. Non è possibile importare moduli con nomi identici fra loro. Nella quinta fase, la macchina virtuale esegue lo script, modificando lo stato globale. Infine, l'ultima fase esegue la funzione *epilogue()*, anch'essa presente nel modulo Libra_account.mvir. Questa funzione ritira una quantità di Libra Coin dall'account del mittente pari al gas consumato e aggiorna il numero di sequenza aumentandone il valore di uno. Esattamente come per il prologo, non verrà ritirato gas aggiuntivo a quello precedentemente consumato durante l'esecuzione della transazione.

2.3 Strutture di dati del database replicato

Dopo aver eseguito una transazione, un validatore propone una nuova versione della struttura di dati autenticata utilizzata per rappresentare il database. La generazione di questa struttura di dati è un'operazione deterministica. Di seguito viene mostrato uno schema raffigurante l'intera infrastruttura del database di Libra.

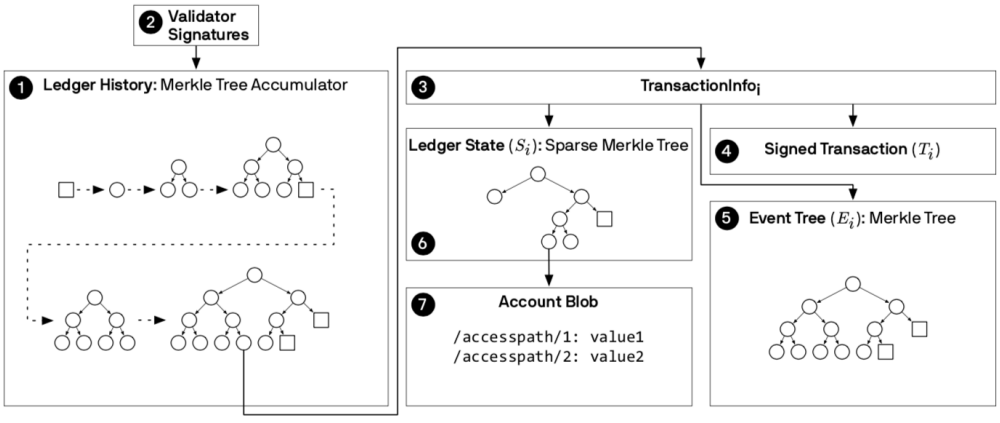

Figura 2.3: Protocollo Libra

1. La radice della storia del ledger è il verificatore di autenticità dell'intero stato del sistema. La maggior parte delle blockchain, fra cui anche il Bitcoin [9], utilizzano una lista concatenata composta da blocchi di transazioni su cui il protocollo di consenso ha raggiunto un accordo. Questa struttura di dati è inefficiente quando un certo client, fidandosi di un certo blocco B, vuole verificare delle informazioni su un certo blocco antenato B'. Per poter eseguire tali operazioni deve processare tutti gli antenati intermedi fra i due blocchi. Il protocollo di Libra, per risolvere questo problema, utilizza un unico albero Merkle per rappresentare la storia del ledger.

2. Un quorum di nodi validatori firma 1).

3. Ogni foglia della storia del ledger permette di accedere ad una specifica transazione.

4. Transazione firmata.

5. La lista degli eventi emessi durante l'esecuzione della transazione è memorizzata tramite un albero Merkle.

6. Lo stato del ledger è memorizzato tramite un albero Merkle sparso di dimensione 2^{256}. La figura seguente mostra alcune ottimizzazioni che vengono eseguite su un albero naive iniziale.

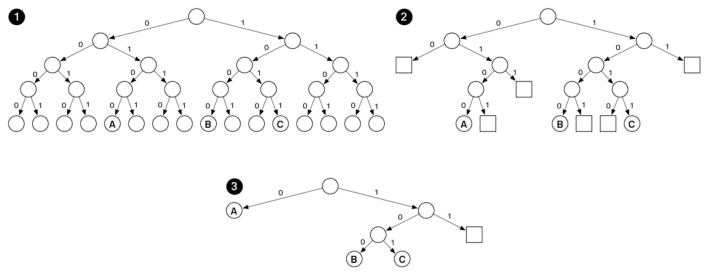

Figura 2.4: Ottimizzazioni su un Albero Merkle con D ={0100_2: A, 1000_2: B, 1011_2: C}

La prima ottimizzazione sostituisce ogni sottoalbero vuoto con una variabile metasintattica seguendo il metodo descritto in [38]. La seconda ottimizzazione sostituisce tutti i sottoalberi composti da una singola foglia con un nodo singolo. Queste ottimizzazioni riducono il numero di hash da calcolare per poter effettuare operazioni sull'albero. La profondità di ogni elemento rimane *O(logn)* dove n è il numero di nodi presenti nell'albero.

7. Ad ogni foglia troviamo il sistema di accesso ai vari account del sistema (coppia access path e valore) in una particolare versione.

Anche se l'albero Merkle garantisce un sistema per accedere ad una determinata transazione in minor tempo rispetto alla lista, lo spazio di archiviazione richiesto è maggiore. Questo spazio richiesto con il passare del tempo potrebbe diventare elevato. Gli sviluppatori di Libra hanno specificato che è in fase di ricerca un sistema per poter ovviare a questo problema a lungo termine.

2.4 Il protocollo di consenso

Libra, diversamente da Bitcoin ed Ethereum, non ricorre al protocollo *Proof of Work*, bensì al protocollo *Proof of Authority*, ovvero non ci sono miners e il potere di approvazione di nuove transazioni viene concesso ad un piccolo e predeterminato gruppo di nodi, chiamati validatori. Il protocollo Proof of Authority possiede una maggiore velocità nell'approvazione di nuove transazioni rispetto al Proof of Work, perché i validatori non devono eseguire operazioni dai costi computazionali elevati. Si perde però la definizione di decentralizzazione, dal momento che un semplice utente non può approvare blocchi, ma solo i validatori definiti sopra. Questa decisione è tutt'oggi oggetto di critica sul web, perché le aziende, se malevoli, potrebbero organizzarsi autonomamente tra di loro su come modificare lo stato del ledger. Viene comunque specificato che questa situazione sarà solamente temporanea ed è

già stata comunicata l'intenzione di spostarsi in cinque anni ad un sistema *permissionless* [35] con protocollo *Proof of Stake*, oggetto che, al momento, è ancora in fase di ricerca. Il protocollo di consenso è perciò fondamentale per garantire il corretto funzionamento della blockchain ed è composto da un algoritmo apposito per regolare il meccanismo di validazione delle transazioni. Il protocollo replica le transazioni emesse su tutti i nodi validatori, esegue le potenziali transazioni sulla versione del database corrente e cerca un accordo per permettere che tutti i nodi validatori possiedano una copia identica della versione del database. Come precedentemente accennato, nella fase embrionale di Libra gli attori dell'algoritmo non saranno dei miners generici, ma per l'appunto dei validators, nodi validatori, composti dalle aziende che hanno investito nella Libra Association. Tutti i validatori potranno controllare il loro nodo in locale o attraverso un sistema cloud. Si stima che il costo di gestione di un nodo validatore si aggiri sui 280.000$ [39]. Tutti i gestori di un nodo dovranno garantire un investimento iniziale di un valore di Libra Coin pari a 10.000.000$, ad eccezione degli enti senza scopo di lucro, che però dovranno comunque mantenere autonomamente i costi di gestione.

2.4.1 L'algoritmo LibraBFT

L'algoritmo di consenso utilizzato in Libra è chiamato LibraBFT (Libra Byzantine Fault Tolerant) [40] ed è ampiamente ispirato dall'algoritmo HotStuff [41]. LibraBFT è un modello basato sui fallimenti bizantini, a sua volta ispirato dal problema dei generali bizantini [42]. L'algoritmo assume che un insieme di $3f+1$ voti sia distribuito su un insieme di validatori che potrebbero essere onesti oppure bizantini. LibraBFT rimane sicuro, prevenendo attacchi di double-spending o fork quando al massimo f nodi sono controllati da validatori bizantini (ovvero al massimo un terzo). L'algoritmo rimane attivo finché esiste un tempo globale di stabilizzazione (GST) dopo del quale tutti i messaggi fra nodi onesti devono essere recepiti da altri nodi onesti con un ritardo massimo δ. I validatori ricevono transazione dai client e le condividono fra loro attraverso un protocollo *mempool*. Mempool è un

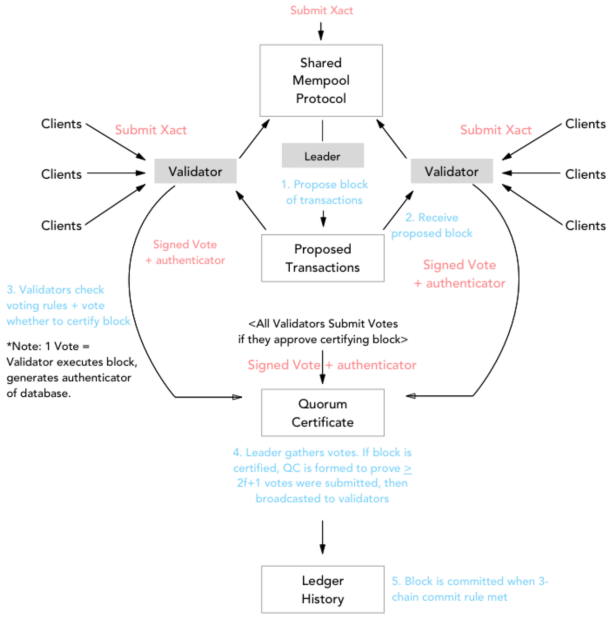

Figura 2.5: Schema dell'algoritmo LibraBFT

buffer di memoria che mantiene le transazioni in attesa di essere eseguite. A questo punto il protocollo LibraBFT agisce tramite una sequenza di round. Ad ogni round, un nodo validatore assume il ruolo di *leader* e propone un blocco di transazioni per estendere una sequenza di blocchi certificata che contiene tutta la precedente storia di transazioni. Gli altri validatori ricevono il blocco e devono decidere se votarlo oppure no. Se il validatore intende votare a favore del blocco, esegue le transazioni del blocco in locale senza causare effetti esterni. In questo modo viene computato un verificatore di autenticità (authenticator) per il database ottenuto con l'esecuzione di tale blocco. A questo punto, il validatore invia il blocco firmato e il validatore di autenticità computato per tale database al leader. Il leader riceve questi voti da tutti gli altri nodi e, se il numero di voti è superiore a $2f+1$, crea un certificato di quorum (QC) e lo invia a tutti i nodi validatori. Un blocco, per essere confermato sulla blockchain, deve superare una catena di 3 round. Un blocco al round k è confermato se possiede un certificato di quorum e se è confermato da altri due blocchi, possedenti anch'essi certificati di quorum per

i round k+1 e k+2. Quando una sequenza di blocchi è confermata, lo stato globale risultante dall'esecuzione delle transazioni presente su tali blocchi è reso ufficiale e viene condiviso fra tutti i nodi validatori della blockchain.

2.5 Stabilità della Libra Coin

Libra Coin, definita nel modulo *Libra_coin.mvir*, è la moneta ufficiale della Libra blockchain. Libra Coin è costruita come una moneta stabile, meno vulnerabile alle fluttuazioni speculative tipiche delle famose criptovalute BitCoin ed ETH, il cui valore è totalmente influenzato dal valore del network sul mercato. Per garantire questa proprietà, la Libra Association gestisce una riserva monetaria chiamata Libra Reserve [43]. Nuova Libra Coin può essere generata (tramite la procedura *mint()*) solamente dalla Libra Association una volta ricevuto il pagamento in moneta reale da parte di un client. La moneta utilizzata per l'acquisto viene tenuta in riserva o convertita in investimenti a basso rischio come obbligazioni bancarie e titoli di stato.

```
public mint(value: u64, capability: &Self.MintCapability): Self.T
    acquires MarketCap {
   let market_cap_ref: &mut Self.MarketCap;
   // LibraCoin totale presente nel mercato
        let market_cap_total_value: u64;

        _ = move(capability);

   // TODO: Controllo utilizzato solamente nella versione di test
      disponibile al momento per evitare si raggiunga il massimo
      valore di Libra imposto
   // Con la versione ufficiale questo controllo non servira' dal
      momento che la LibraCoin sara' sostenuta dalla riserva e le
      fresh mint saranno comunque rare
        assert(copy(value) <= 1000000000 * 1000000, 11); // *
            1000000 perche' l'unita' e' microlibra
```

```
        // Si aggiorna il marketcap
    // L'indirizzo hardcoded 0xA550C18 e' l'indirizzo fisico della
        Libra Association
        market_cap_ref = borrow_global_mut<MarketCap>(0xA550C18);
        market_cap_total_value = *&copy(market_cap_ref).total_value;
        *(&mut move(market_cap_ref).total_value) =
            move(market_cap_total_value) + copy(value);

        return T{value: move(value)};
    }
```

Gli sviluppatori di Libra, stanno lavorando anche ad un wallet, Calibra [43], che verrà distribuito come applicazione singola ed integrato nelle piattaforme Facebook e WhatsApp.

Capitolo 3

Move

Move [6] è un nuovo linguaggio di programmazione per l'implementazione di *smart contract* e transazioni customizzate nell'ambiente di Libra. La versione finale è ancora in fase di sviluppo ed al momento è stata distribuita una sua versione intermedia, chiamata Move IR. Il compilatore del linguaggio è scritto in Rust, compila un file Move, comunemente denominato con l'estensione *.mvir*, e restituisce una sequenza di istruzioni bytecode che verranno poi eseguite dalla macchina virtuale di Move (Move Virtual Machine). Il compilatore non esegue ottimizzazioni e controlli semantici, rimandando tale processo al verificatore del bytecode,*bytecode verifier*, di cui si parlerà più dettagliatamente nelle prossime sezioni. Le tre funzionalità principali proposte da Move, sono:

- I moduli.
- I transaction scripts.
- Le risorse First Class.

I moduli sono l'equivalente degli smart contract di Ethereum. All'interno di un modulo vengono dichiarati tipi di dato struttura, specialmente risorse, che sono delle strutture di genere (*kind*) *resource* e non *unrestricted*, e procedure. Le procedure possono manipolare le risorse eseguendo azioni come *Unpack*

(distruzione), *Pack* (creazione) o il semplice accesso. Queste operazioni possono essere utilizzate solamente all'interno del modulo in cui la risorsa viene dichiarata. I moduli vengono pubblicati sotto gli account ed ogni account può contenere un unico modulo con lo stesso nome. Tutti i moduli sono riutilizzabili. I transaction scripts sono delle transazioni programmabili che specificano delle determinate azioni da eseguire sullo stato globale.

- Interagiscono con i moduli, chiamando le procedure da loro dichiarate. Per poterle chiamare è necessario importare il modulo con l'istruzione *import* specificando l'indirizzo dell'account in cui il modulo è stato pubblicato.

- Non vengono salvati nel global state e non vengono perciò pubblicati sotto nessun account. Vengono solamente compilati ed eseguiti.

- Sono monouso.

Le risorse infine sono il fiore all'occhiello del linguaggio Move. Sono un tipo di dato strutturato ispirato dalla logica lineare. Una risorsa non potrà mai essere copiata o *discarded*, persa, durante l'esecuzione. Potrà essere creata o distrutta solamente da procedure utilizzate dal modulo in cui la risorsa stessa è stata dichiarata. Questi meccanismi di sicurezza sono esercitati staticamente dal sistema di tipi di Move, e, più precisamente, dal verificatore del bytecode. Pur avendo queste protezioni la risorsa rimane semplicemente un caso particolare di struttura. Le risorse vengono pubblicate sugli account, il quale può possedere un'unica risorsa con un determinato nome. L'utilizzo delle risorse è alla base dello sviluppo di ogni modulo o script sviluppato con il linguaggio Move. Grazie a questo particolare tipo di dato strutturato introdotto, si riescono a programmare nuovi token, senza doversi preoccupare di scrivere del codice che controlli a runtime che l'asset non venga perso, copiato o riutilizzato più volte, processo che invece è necessario eseguire nello sviluppo di token in Solidity, che vedremo più avanti. La Libra Coin, la moneta ufficiale della Libra Blockchain, è anch'essa una risorsa e non ha alcuna carat-

teristica “speciale” rispetto alle altre risorse implementabili nell’ecosistema, al contrario quindi dell’Ether nella blockchain di Ethereum.

```
module myToken {
   // Definiamo due risorse
   // Si definisce in modo standard con il nome T la risorsa
       principale del modulo
   resource T {
      valore: u64,
   }
   resource altraRisorsa {
      indirizzo: address,
      token: Self.T,
   }
   ...
}
```

Le risorse possono contenere altre risorse mentre semplici strutture *unrestricted* non possono contenere risorse per evitare problemi durante la fase di controllo di genere,*kind checking*, da parte del verificatore del bytecode.

3.1 Sistema di Tipi di Move

In questa sezione si descriverà il sistema di tipi di Move. Per la stesura si fa riferimento all’articolo tecnico ufficiale [1], al codice sorgente pubblico disponibile a [3] e a vari topic del community forum [4], dove gli sviluppatori della Libra Association illustrano aspetti assenti nella documentazione attualmente disponibile, o aggiunti solamente negli ultimi mesi. Si noti che il progetto è attualmente in fase di sviluppo, le seguenti informazioni sono corrispondenti a febbraio/marzo 2020. Move rende disponibili come tipi primitivi:

- Booleani, definiti con la keyword *bool*.

- Interi senza segno a 64 bit, *u64*.
- Indirizzi di account a 256 bit, *address*.
- Array di lunghezza fissata.

Mentre come tipo di dato strutturato rende disponibili le:

- Strutture, tipo di dato strutturato con genere *unrestricted*.
- Risorse, tipo di dato strutturato con genere *resource*.

Tutti i tipi che non sono risorse sono considerati unrestricted. Infine sono disponibili come in Rust i tipi riferimento, *reference*, che possono essere mutabili, definiti come "&*mut*", o immutabili, definiti come "&". I controlli di tipo verranno eseguiti dal verificatore del bytecode, prima che l'interprete della macchina virtuale ne esegua il codice.

3.1.1 Risorse

Si è già accennato le caratteristiche peculiari delle risorse in Move ma in questa sezione si approfondirà:

- Quali proprietà degli asset le risorse riescono a soddisfare.
- Come vengono implementate nella pratica a codice e quali esempi di codice errati vengono rifiutati dal verificatore del bytecode.
- Come il verificatore del bytecode impone questi controlli.

In ogni blockchain la scelta su come implementare asset digitali è diversa, ma le proprietà da rispettare sono le stesse ovvero:

- Scarsità (*Scarcity*).
- Controllo di accesso (*Access Control*).

La quantità di asset in un sistema deve essere controllata. Duplicare asset (token, monete...) è un'azione proibita mentre crearne nuovi da zero (mint) è un'azione privilegiata. Nel caso della risorsa LibraCoin, il mint di una certa quantità di nuova moneta può essere eseguito solamente da un account possedente un indirizzo specifico codificato, quello della Libra Association. Questi privilegi comunque devono essere specificati dal programmatore. Il fattore determinante è che procedure esterne non possono duplicare, far scomparire o riutilizzare risorse arbitrariamente. Per controllo di accesso, si intende il modo in cui un membro di un sistema può proteggere i suoi asset con particolari politiche. Dal momento che non è possibile pubblicare risorse sotto un altro account diverso dal proprio, si nota immediatamente come si riesce a soddisfare perciò entrambe le proprietà degli asset rappresentandoli sotto forma di risorse. Si mostrerà ora un esempio di codice Move corretto e tre esempi che verranno invece rifiutati dal verificatore del bytecode.

```
//transazione corretta
import 0x0.LibraAccount;
import 0x0.LibraCoin;

main(payee: address, amount: u64) {
   let coin: LibraCoin.T;
   coin = LibraAccount.withdraw_from_sender(move(amount));
   LibraAccount.deposit(move(payee), move(coin));
   return;
}
```

Questo è un esempio di transaction script corretto, una semplice transazione in cui si ritira della quantità di LibraCoin dal chiamante e la si deposita ad un indirizzo arbitrario *payee* utilizzando la procedura *withdraw_from_sender*, che a sua volta ritira una quantità di LibraCoin e la restituisce alla variabile coin. Successivamente tale quantità viene ridepositata nell'account payee. La risorsa non viene persa, duplicata o riutilizzata quindi il codice è corretto.

```
//perdita di risorse
import 0x0.LibraAccount;
import 0x0.LibraCoin;
main(payee: address, amount: u64) {
   let coin: LibraCoin.T;
   coin = LibraAccount.withdraw_from_sender(move(amount));
   return;
}
```

In questo caso la risorsa viene ritirata e salvata in *coin*, ma la transazione termina senza che la risorsa disponibile nella variabile locale venga utilizzata. Avviene perdita di risorse e quindi il verificatore del bytecode rifiuterà questa transazione.

```
//riutilizzo di risorse
import 0x0.LibraAccount;
import 0x0.LibraCoin;
main(payee: address, amount: u64) {
   let coin: LibraCoin.T;
   let coin2: LibraCoin.T;
   coin = LibraAccount.withdraw_from_sender(move(amount));
   LibraAccount.deposit(copy(payee), move(coin));
   LibraAccount.deposit(move(payee), move(coin));
   return;
}
```

In questo caso la risorsa è riutilizzata due volte, ma dopo la prima istruzione *move* la risorsa non può essere più utilizzata, perciò il codice non è valido.

```
//duplicazione di risorse
import 0x0.LibraAccount;
import 0x0.LibraCoin;
main(payee: address, amount: u64) {
   let coin: LibraCoin.T;
```

```
    let coin2: LibraCoin.T;
    coin = LibraAccount.withdraw_from_sender(move(amount));
    LibraAccount.deposit(copy(payee), copy(coin));
    LibraAccount.deposit(move(payee), move(coin));
    return;
}
```

Infine, in quest'ultimo esempio si sta cercando di eseguire l'istruzione *copy* su una risorsa, che però è un'operazione proibita (non è possibile copiare risorse). Il verificatore del bytecode rifiuterà perciò anche questo terzo esempio di transaction script. Notare che invece copiare l'indirizzo è un'azione possibile, a patto che non sia stata utilizzata precedentemente una move su tale valore.

3.1.2 Move VM: verificatore ed interprete del bytecode

Il compilatore di Move [5], scritto in Rust, traduce codice Move in una sequenza di istruzioni bytecode. L'insieme delle istruzioni bytecode è disponibile in [1]. Questo codice bytecode verrà eseguito dall'interprete del bytecode, *bytecode interpreter*, disponibile nella macchina virtuale di Move, dopo essere passato per il verificatore del bytecode, *bytecode verifier*. L'interprete è costruito tramite una struttura di dati a pila, ispirata dalla macchina virtuale di Java.

- Un'istruzione bytecode consuma operandi dalla pila e *pusha* successivamente i risultati.
- Le istruzioni possono effettuare move e copy di valori da o verso variabili locali della procedura corrente.
- Permette chiamate di procedura.

Come in Solidity, anche in Move è introdotta la nozione di *gas*. La computazione della transazione è dosata da un costo in gas associato ad ogni

istruzione bytecode. Non è stato ancora definito il costo specifico di ogni istruzione bytecode, gli sviluppatori hanno comunicato che tale informazione verrà comunicata più avanti quando il progetto sarà in uno stato più avanzato. Move supporta sei macro categorie di istruzioni bytecode:

1. *CopyLoc*/*MoveLoc* per copiare/muovere dati dalle variabili locali alla pila. *StoreLoc* per fare l'operazione contraria.

2. *Push*/*Pop* dei valori sulla pila ed operazioni aritmetiche/logiche su tali valori.

3. *Module builtins*, istruzioni riservate ai moduli: *Pack*/*UnPack*, per creare/distruggere i tipi dichiarati dal modulo. *MoveToSender*/*MoveFrom* per pubblicare o togliere tipi da un account. *BorrowField* per acquisire un riferimento ad un campo di uno o più tipi di un modulo. Notare che non esiste il MoveTo generico, dal momento che non è possibile pubblicare risorse e moduli sotto altri account, ma solo nel proprio [18].

4. Istruzioni relative ai riferimenti. *ReadRef*, *WriteRef*, *ReleaseRef* e *FreezeRef* per, rispettivamente, leggere, scrivere, distruggere un riferimento e convertire un riferimento mutabile in un riferimento immutabile (il contrario non è possibile).

5. Istruzioni di controllo di flusso.

6. Operazioni specifiche della blockchain come *get_txn_sender()* che ritorna l'indirizzo del mittente. Equivalente alla *msg.value()* di Solidity.

Quando si utilizza BorrowField, MoveToSender/MoveFrom o si chiama un'altra procedura che adopera queste istruzioni è necessario includere la parola chiave *acquires* nella dichiarazione. Questa annotazione riferisce al verificatore del bytecode quale tipo la procedura potrebbe acquisire dallo stato globale della blockchain. Per esempio, in withdraw_from_sender, disponibile nel modulo Libra_account.mvir [7], si utilizza acquires facendo riferimento al tipo LibraAccount.T, su cui verrà eseguito borrow_global_mut, per ottenere

un riferimento ad una risorsa presente nello stato globale. Nel caso si omettesse acquires, si otterrebbe l'errore "MISSING_ACQUIRES_RESOURCE_ANNOTATION" dal verificatore del bytecode, perciò non in fase di compilazione.

```
public withdraw_from_sender(amount: u64): LibraCoin.T acquires T {

        let sender_account: &mut Self.T;

        sender_account = borrow_global_mut<T>(get_txn_sender());

        ...
}
```

Il verificatore del bytecode[8] impone le regole di sicurezza staticamente per ogni modulo che si vuole pubblicare e per ogni transazione che si vuole eseguire. Questa verifica viene eseguita solo dopo la compilazione, perciò è possibile che vengano compilati alcuni programmi semanticamente errati. Tuttavia, non verrà pubblicato nessun modulo o eseguito nessuno script dalla macchina virtuale prima di essere passati per il verificatore del bytecode. L'analisi semantica e il linking si suddividono in sei fasi distinte:

1. Costruzione di un grafo per il controllo di flusso.
2. Controllo di bilanciamento della pila.
3. Controllo di tipi (*type checking*).
4. Controllo di genere, per le risorse (*kind checking*).
5. Controlli per i riferimenti (*reference checking*).
6. Linking con lo stato globale.

Il verificatore costruisce un grafo per il controllo di flusso composto da blocchi (nulla a che vedere con i blocchi di transazioni della blockchain), ognuno

composto da una sequenza di istruzioni bytecode. Tutte le istruzioni da eseguire verranno divise in vari blocchi. Questa suddivisione avverrà in presenza di un'istruzione *jump* o *return*. La fase di controllo di bilanciamento della pila assicura che il chiamato non possa accedere a locazioni della pila riservate al chiamante. Quando una procedura comincia l'esecuzione, la lunghezza della pila è n. Il bytecode, affinché sia valido, deve soddisfare al termine dell'esecuzione il seguente invariante:

- Quando si raggiunge la fine di un blocco, l'altezza della pila deve rimanere n.
- L'ultimo blocco, quello terminante con l'istruzione return, deve avere un'altezza di $n+m$, avendo in più rispetto agli altri blocchi i parametri di ritorno della procedura.

La fase di controllo di tipo assicura che ogni istruzione e procedura sia invocata con argomenti del tipo appropriato. Per controllare i tipi si fa utilizzo di una pila ausiliaria. La fase di controllo di genere comprende l'imposizione dei vincoli di sicurezza delle risorse, ovvero i controlli addizionali che vietano la duplicazione, il riutilizzo o la perdita delle risorse nel codice Move. Questi controlli sono suddivisi in tre categorie:

- Controlli per evitare la duplicazione di risorsa. Non si può utilizzare *CopyLoc* (pushare il valore sullo stack) su una struttura di genere risorsa. Non si può utilizzare *ReadRef* (pop del riferimento e push del valore puntato) su un valore della pila il cui tipo è un riferimento ad un valore di genere risorsa.
- Controlli per evitare la distruzione di risorsa. Non si può utilizzare *PopUnrestricted* su una struttura di genere risorsa sulla pila, *StoreLoc* su una variabile locale che contiene una risorsa e *WriteRef* su un riferimento ad una risorsa.
- Controlli per evitare la perdita di risorsa. Quando una procedura esegue return, nessuna variabile locale può contenere una risorsa.

Ci sono dei casi in cui questi controlli possono essere schivati e perciò devono essere trattati adeguatamente. Se una transazione causa un *halt* (la revert di Solidity) e perciò termina con un errore, non verrà eseguita nessuna modifica allo stato globale della blockchain. Questo è garantito da una copia dello stato globale che il validatore mantiene durante l'esecuzione di una transazione, come descritto in [2]. Verrà comunque pagato il quantitativo di gas in LibraCoin fino all'istruzione che causa l'errore. La nozione di gas evita inoltre l'esistenza di programmi che non terminano, i quali potrebbero rendere una risorsa irraggiungibile. Prima o poi il gas si esaurirà, facendo terminare la computazione con un *halt*, tornando perciò alla situazione descritta in precedenza. La fase di controllo dei riferimenti assicura che non ci siano *dangling reference* e che non vengano eseguite *read* o *write* senza permessi. La fase finale di linking controlla che le procedure e risorse importate dallo stato globale (procedure e risorse di moduli appartenenti ad account esterni), siano correttamente indicate dagli identificatori utilizzati nel codice.

3.2 Move vs Solidity

Se si discute riguardo a blockchain e criptovalute, Bitcoin [9] ed Ethereum [10] sono sicuramente le più note, con il valore più alto sul mercato [11] e su cui si concentra la maggior parte degli studi scientifici. Entrambe hanno linguaggi che permettono di eseguire azioni più complesse rispetto al semplice passaggio di denaro. Nell'ambiente Bitcoin si utilizza Bitcoin Script [12], o semplicemente Script, un linguaggio non Turing completo e perciò con un'espressività fortemente limitata. Più interessante, per effettuare un confronto diretto con Move e Libra, è invece Solidity [20], un linguaggio di programmazione ad oggetti, utilizzato per la programmazione di *smart contract* sulla piattaforma Ethereum. Ethereum è la prima blockchain che ha introdotto la definizione di smart contract [13], una collezione di codice e dati che risiedono ad un indirizzo, un account specifico. Solidity è un linguaggio a tipizzazione statica ed è stato sviluppato con l'intento di essere poi compilato in bytecode eseguibile sulla macchina virtuale di Ethereum, (*EVM, Ethereum Virtual Machine*) [14]. Sia Move che Solidity sono linguaggi a tipizzazione statica e turing-completi. Entrambi i linguaggi vengono compilati in codice bytecode eseguibile nelle rispettive macchine virtuali. In questa sezione si confronterà Move con Solidity, mostrando i punti di forza e debolezza di un linguaggio rispetto all'altro. Gli stati globali non sono molto diversi tra loro, i contratti e i moduli sono pubblicati sotto un determinato account, così come gli Ether posseduti e le risorse.

1. Solidity propone una varietà di tipi primitivi maggiore rispetto a Move. Comprende i tipi di funzione [15], quindi fornisce la possibilità di scrivere funzioni di ordine superiore, mentre in Move ciò non è possibile. I tipi di funzione si dividono in: *internal*, utilizzabili solo all'interno di un contratto, ed *external*, che possiedono un indirizzo e una firma (*signature*) e possono essere passati esternamente.

2. Supporta polimorfismo parametrico e gli eventi (*events*). I generici (*generics*) non sono al momento supportati da Move IR, ma è stato

annunciato che sono sulla roadmap di sviluppo e verranno implementati in futuro.

3. Un contratto può ereditare un altro contratto, in Move non c'è ereditarietà fra moduli. Se da un lato Solidity vanta una maggiore espressività dall'altro rimane vulnerabile ad un attacco re-entrancy [3.2.1], al quale invece Move è immune.

4. Solidity propone la parola chiave *payable* per indicare funzioni di un contratto che hanno un costo in Ether (aggiuntivo al gas) per essere eseguite. Il client che chiama una funzione payable decide a priori qual è la massima quantità di Ether che è disposto a spendere e non deve leggere la totalità del codice per essere sicuro che non venga ritirata moneta extra. Questa parola chiave è utilizzabile solo per l'Ether, non per altri token. In Move, la risorsa LibraCoin, è sviluppata esattamente come un qualsiasi altro token. Non esiste una parola chiave che indichi che verrà ritirata moneta. Pertanto, è necessario leggere il codice di ogni procedura di un modulo che si intende chiamare per essere sicuri che non venga eseguita una withdraw_from_sender senza il nostro permesso. In questo caso Move è più fragile di Solidity.

5. Move supporta il tipo di dato strutturato risorsa, Solidity no. Solamente l'Ether è implementato con una semantica simile e possiede la proprietà di *scarcity*. Sviluppare un nuovo token in Move è più semplice e sicuro visto che si è protetti dalla definizione di risorsa. In Solidity lo sviluppatore di un nuovo token deve scrivere autonomamente tutti i meccanismi di sicurezza, stando attento a non introdurre bug che permettano la duplicazione, perdita o riutilizzo non voluta del token. Maggior libertà per Solidity, ma meno sicurezza.

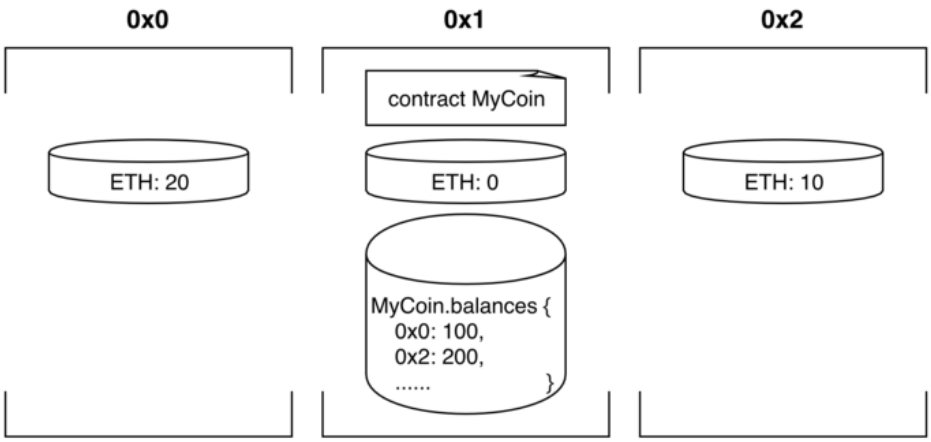

Figura 3.1: Stato globale di Ethereum

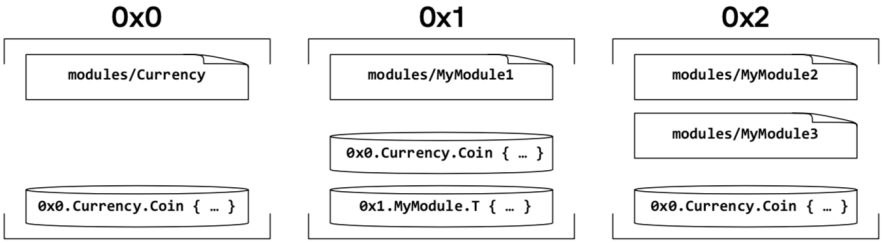

Figura 3.2: Stato globale di Move

Descriviamo più nel dettaglio le differenze 3, 4 e 5.

3.2.1 Vulnerabilità a re-entrancy

L'ereditarietà (*inheritance*) fra moduli richiederebbere l'utilizzo del dynamic dispatch. In Move si è scelto di limitare l'espressività per evitare di insorgere in problemi di re-entrancy (codice rientrante) [1, pag.14]. Un problema di vulnerabilità di questo tipo avviene quando un client può chiamare una funzione di un tuo contratto e potenzialmente rientrare nel contratto

prima che la funzione sia terminata. Il problema è piuttosto grave nel caso in cui ci siano di mezzo funzioni payable. Un problema di questo tipo in Move potrebbe violare alcuni vincoli di sicurezza delle risorse. Perciò, gli sviluppatori hanno risolto il problema in questo modo:

- Una procedura è identificata dal nome del suo modulo e dalla sua firma, questo ID è unico per ogni procedura.
- L'istruzione bytecode *Call* richiede un'ID come input, che è unico, quindi è possibile determinare staticamente tutte le chiamate di procedura.
- Un modulo può dipendere unicamente da moduli definiti in precedenza, ovvero presenti da più tempo nella storia del ledger della blockchain. Perciò, non potendo tornare indietro, il grafo di dipendenza dei moduli è aciclico.

La combinazione del grafo aciclico con l'assenza del dynamic dispatch garantisce l'invariante che tutti gli *stack frame* delle procedure di un modulo siano contigui, dunque non si può verificare un caso di re-entrancy. Maggior sicurezza in cambio di una minor espressività.

```
// esempio di codice Solidity che non puo' essere scritto in Move
contract A {
   uint256 public n;
   function set() public {
      n = 1;
   }
}

contract B is A {
   function set() public {
      n = 2;
   }
}
```

```
contract C is A,B {
   function test (uint n) public {
      A a;
      if (n == 3) {
         a = new A();
      }
      else {
         a = B(new B());
      }
      a.set();
   }
}
```

	Move IR	Solidity
Blockchain	Libra	Ethereum
Tipato	Staticamente	Staticamente
Turing completo?	Sì	Sì
Compilato?	Sì, in codice bytecode eseguibile dalla Move VM	Sì, in codice bytecode eseguibile dalla EVM
Tipi Primitivi	Booleani, interi senza segno a 64-bit, indirizzi, array di byte, riferimenti	Booleani, interi, indirizzi, array di byte, stringhe, enum, funzioni
Tipi Strutturati	Strutture e risorse	Strutture, mappe, array dinamici
Currency Principale	Libra Coin	Ether
Dynamic Dispatch	Non supportato	Supportato

3.2.2 Funzioni payable

Si è descritto in precedenza che, in Solidity, quando c'è un trasferimento di Ether tramite la procedura di un contratto, è obbligatorio inserire il modifier *payable*[16] nella dichiarazione di tale procedura.

```
function () public payable {}
```

In Solidity, inoltre, è possibile solamente depositare Ether, generalmente con una funzione denominata transfer, mentre non è possibile ritirare Ether da un altro account. In Move invece, non esiste nessuna keyword simile a pa-

yable ed è possibile dichiarare un qualsiasi modulo contentente una withdraw_from_sender che ritiri una certa quantità di Libra Coin dal chiamante caricandola ad un altro determinato indirizzo (che deve possedere almeno un'unità di Libra). Mostriamone un esempio.

```
module happy{

   import 0x0.LibraAccount;
   import 0x0.LibraCoin;

   public trap(){
       let amount: u64;
       let coin: LibraCoin.T;
       let thief: address;
       thief = 0x0000000000000;
       amount = 10;
       // do some stuff
       coin = LibraAccount.withdraw_from_sender(move(amount));
       LibraAccount.deposit(move(thief), move(coin));
       // do some other stuff
       return;
   }
}
```

Questa problematica è stata confermata dagli sviluppatori di Libra e giustificata dal fatto che, essendo certe operazioni permesse solamente dal modulo che definisce la risorsa, è sufficiente leggere il modulo per verificare che sia tutto sicuro. Il problema si verificherà soprattutto in moduli particolarmente verbosi. Rimane perciò uno svantaggio rispetto a Solidity dove è sufficiente controllare la parola chiave nella dichiarazione della funzione. Anche in questo caso perciò, la Libra Coin si comporta come un qualunque token implementabile da terze parti. Più precisamente, la fragilità è data dall'istruzione BorrowGlobal<Type>, utilizzata anche nell'implementazione di

withdraw_from_sender [17], che permette di ottenere un riferimento ad una risorsa posseduta da un indirizzo diverso dal mittente. La risposta degli sviluppatori [18] è la seguente: "*The idea is that before choosing to publish a resource T under your account, you should look over the code for the module and make sure you are ok with the API that the declaring module of T exposes.*". Infine, è interessante specificare che, a differenza di Ethereum, una piattaforma totalmente aperta, in Libra i moduli di terze parti prima di essere pubblicati dovranno essere approvati dalla Libra Association tramite dei tool automatici ancora da sviluppare e poi, quelli ritenuti "non malevoli", verranno inseriti in un marketplace. Dettagli riguardanti tool e marketplace non sono ancora stati rilasciati.

3.2.3 Implementazione di un custom token: PaneToken

Tenendo in considerazione le differenze fra i due linguaggi descritte precedentemente, in questa sezione si presenterà l'implementazione di PaneToken, un custom token scritto sia in Move che in Solidity. Si seguirà lo standard di programmazione ERC20 [19] per la versione scritta in Solidity, prendendo ispirazione da [21] e [22]. L'interfaccia da implementare è la seguente:

```
contract ERC20 {
  function totalSupply() constant returns (uint theTotalSupply);
  function balanceOf(address _owner) constant returns (uint
      balance);
  function transfer(address _to, uint _value) returns (bool
      success);
  function transferFrom(address _from, address _to, uint _value)
      returns (bool success);
  function approve(address _spender, uint _value) returns (bool
      success);
  function allowance(address _owner, address _spender) constant
      returns (uint remaining);
```

```
    event Transfer(address indexed _from, address indexed _to, uint
        _value);
    event Approval(address indexed _owner, address indexed _spender,
        uint _value);
}
```

Per la versione di PaneToken scritta in Move si seguirà invece un pattern di programmazione ispirato ai moduli attualmente disponibili sul Github. In Move l'implementazione del token sarà simile a quello della Libra Coin, la currency ufficiale di Libra, ogni account potrà importare il modulo, possedere una risorsa di tipo PaneToken.T (con una certa quantità) ed eseguire le procedure sulla propria risorsa e su quella di altri account. Invece il token in Ethereum sarà centralizzato e l'intero stato globale dovrà essere salvato nel contratto.

```
contract Pane{

    // SafeMath [23] e' una libreria che fa da wrapper a istruzioni
        aritmetiche per evitare vulnerabilita' dovute a overflow
    using SafeMath for uint256;
    mapping (address => uint256) private quantita;
    // un account da il permesso ad un altro account di poter
    // ritirare una certa quantita'
    mapping (address => mapping (address => uint256)) private
        permesso;
    // max quantita' disponibile sul mercato
    uint256 private _MaxPaneMercato;
    ...
}
```

Si utilizza un mapping address ($\Rightarrow$ uint256) per tenere traccia di quanti token possiede ogni singolo account, *quantita*. In Move invece ogni singolo account tiene traccia del proprio balance singolarmente.

```
module Pane {
   // risorsa che rappresenta il nostro token Pane
   resource T {
      // quantita di pane
      quantita: u64,
   }
   // chi possiede questa risorsa e' in grado di sfornare nuovo pane
   resource LicenzaFornaio {}
   resource MaxPaneMercato {
      // la somma massima di pane disponibile sul mercato
      totale_pane: u64,
   }
   ...
}
```

Mostreremo qui di seguito a confronto la funzione *transfer* per la versione in Solidity con la funzione *dai* (concettualmente simile alla deposit di LibraCoin) e con la funzione *prendi* (withdraw).

```
// Solidity
// trasferisce token dal mittente all'address destinatario
function transfer(address to, uint256 value) public returns (bool)
    {
   require(value <= quantita[msg.sender]);
   require(to != address(0));
   quantita[msg.sender] = quantita[msg.sender].sub(value);
   quantita[to] = quantita[to].add(value);
   emit Transfer(msg.sender, to, value);
   return true;
}
```

Si modifica semplicemente la quantità di pane di entrambi gli account e si emette l'evento relativo al trasferimento.

```
// Divide il pane in due parti
```

```
// Il pane originale avra valore = valore iniziale - valore da
    ritirare
// Il nuovo pane avra valore = valore da ritirare
public prendi(pane_ref: &mut Self.T, quantita_da_ritirare: u64):
    Self.T {
   let quantita_tmp: u64;
   quantita_tmp = *(&mut copy(pane_ref).quantita);
   assert(copy(quantita_tmp) >= copy(quantita_da_ritirare), 10);
   *(&mut move(pane_ref).quantita) = move(quantita_tmp) -
       copy(quantita_da_ritirare);
   return T{quantita: move(quantita_da_ritirare)};
}
```

Si modifica la quantità di pane puntata da pane_ref. Notare che siamo dentro al modulo in cui viene definita la risorsa, quindi abbiamo piena libertà nel poter modificare i valori. Non sarebbe possibile eseguire questo procedimento al di fuori del modulo. L'ultima istruzione, è una Pack, si ritorna una nuova risorsa pane con un valore pari alla quantità ritirata.

```
// aggiungi a *pane_ref il contenuto di pane_da_dare
// pane_da_dare viene consumato
public dai(pane_ref: &mut Self.T, pane_da_dare: Self.T) {
   let quantita_tmp: u64;
   // salva il totale del pane
   let quantita_da_dare: u64;
   quantita_tmp = *(&mut copy(pane_ref).quantita);
   // fa unpack di pane_da_dare e salva la sua quantita in
       quantita_da_dare
   // se avessi scritto solamente T { quantita } avrei fatto unpack
       ma senza salvare il valore
   T { quantita: quantita_da_dare } = move(pane_da_dare);
   *(&mut move(pane_ref).quantita) = move(quantita_tmp) +
       move(quantita_da_dare);
   return;
```

```
}
```

Di seguito mostriamo tre esempi di transaction scripts. Si utilizzano alcune procedure presenti nel modulo PaneToken che non sono state mostrate nella sezione precedente, perciò per una più completa analisi si può consultare [24]. Si può notare uno degli aspetti più interessanti di Move: il poter gestire risorse in locale, sapendo di essere protetti dal sistema di tipi.

```
//! new-transaction
import {{default}}.Pane;
main() {
   let p: Pane.T;
   Pane.initialize();
   // Pane.zero ritorna una risorsa Pane.T con quantita 0
   p = Pane.zero();
   Pane.destroy_zero(move(p));
   return;
}
```

```
//! new-transaction
import {{default}}.Pane;
main() {
   let paneGino: Pane.T;
   let paneLuigi: Pane.T;
   // sforna_senza_licenza e' una mint, ritorna risorsa Pane
   paneGino = Pane.sforna_senza_licenza(5);
   paneLuigi = Pane.sforna_senza_licenza(1);
   paneLuigi = Pane.unisci(move(paneLuigi), move(paneGino));
   Pane.distruggi(move(paneLuigi));
   return;
}

//! new-transaction
import {{default}}.Pane;
```

```
main() {
    let paneGino: Pane.T;
    let paneLuigi: Pane.T;
    paneGino = Pane.sforna_senza_licenza(5);
    //Gino si tiene 3, Luigi 2
    paneGino, paneLuigi = Pane.spezza(move(paneGino), 2);
    Pane.distruggi(move(paneLuigi));
    Pane.distruggi(move(paneGino));
    return;
}
```

Per testare il token in Solidity si è utilizzato Remix [25], un'IDE disponibile anche online. Per testare moduli e transazioni in Move invece non esiste ancora un'IDE, ma ci sono due procedimenti possibili da poter eseguire:

- Si possono compilare moduli o script inserendoli nella directory "testsuite" sotto "language" e lanciare da terminale *cargo test "nome modulo/script"*.
- Si possono seguire le istruzioni disponibili in [26].

La documentazione disponibile a [26] è comunque risultata incompleta al momento della stesura di questo documento, perciò dopo aver scaricato i file indicati per poter lanciare una testnet in locale, si consiglia successivamente di utilizzare le seguenti istruzioni:

1. cd libra
2. ./scripts/dev_setup.sh (solo la prima volta)
3. source .../.cargo/env
4. cargo build
5. ulimit -n 8192 (come suggerito nel github issue [27])
6. cargo run -p libra-swarm

Capitolo 4

Simulazione delle risorse in Solidity

In questa sezione si presenterà una tecnica per illustrare lo sviluppo di un sistema di protezione degli asset in Solidity simile a quello implementato in Move, ovvero implementando le risorse, delle strutture che non si possono copiare, riutilizzare o perdere. Si è scelto come sperimentazione di utilizzare in questo sistema le stesse operazioni implementate dalla LibraCoin ufficiale di Libra, in modo da poter imitare il più possibile il pattern standard di sviluppo di moduli Move utilizzato dagli sviluppatori della Libra Association. Si è utilizzata la libreria SafeMath per poter eseguire operazioni aritmetiche sicure sugli uint256. Per poter dichiarare funzioni che ritornano strutture, si è attivato *ABIEncoderV2*. L'interfaccia di funzioni principali che sono state implementate è la seguente:

```
// inizio dell'esecuzione sicura
function init() internal;

// fine dell'esecuzione sicura
function end() internal;

// mint di nuova risorsa per un certo indirizzo
```

```
function mint(address address_to_mint, uint256 value_to_create)
    internal;

// distrugge una risorsa
function destroy(resource memory res) internal;

// denominata anche withdraw, ritira una risorsa dal global state
function get(address address_from, uint256 value_to_withdraw)
    internal returns (resource memory);

// denominata anche deposit, deposita una risorsa da una
    variabile locale al global state
function put(resource memory res, address receiver) internal;

// prende in input una risorsa, la divide e ritorna una nuova
    coppia di risorse
function split(resource memory res, uint256 value_to_split)
    internal pure returns (resource memory, resource memory);

// prende in input due risorse, le unisce e ritorna una nuova
    risorsa
function merge(resource memory A, resource memory B) internal
    returns (resource memory);

// funzione di esecuzione, simula l'esecuzione di un transaction
    script di Move
function exec() public;
```

La funzione *exec()*, che è pubblica nel contratto, esegue una sequenza qualunque delle operazioni interne definite in precedenza in modo sicuro. Tale esecuzione dovrà essere delimitata da *init()* (come prima operazione) ed *end()* (come ultima operazione).

4.1 Stato globale e last owner

Si deve riflettere ora su come poter rappresentare i dati. Dal momento che stiamo simulando le risorse, una struttura speciale (struttura di genere *resource*) nell'ambiente di Move, la prima intuizione è stata quella di simulare il suo stato globale, dove ogni risorsa è collegata ad un determinato account. Solidity ci permette l'utilizzo del tipo di dato *struct* per l'implementazione di tipi di dato custom, che è perfetto per il compito che vogliamo portare a termine. Si è utilizzato perciò una mapping (address ⇒ resource), dichiarata come *globalState*, dove resource è di tipo struct.

```
struct resource {
   // last_owner tiene traccia di quale account ha posseduto la
      risorsa prima del get dal global state
   address last_owner;
   uint256 value;
}
   mapping (address => resource) globalState;
   mapping (address => resource) copiaGlobalState;
   // lista degli address del sistema
   uint256[] keys;
   // semplice booleano per controllare che sia stata lanciata
      init() prima delle altre operazioni
   bool safe_run\incorso;
```

In questa implementazione il campo last_owner si rivela fondamentale per andare ad aggiornare copiaGlobalState quando si ritira una risorsa da un determinato account e la si deposita poi successivamente su un terzo account, oppure quando si esegue una merge(). Notare che non è necessario mantenere la sequenza degli address che hanno posseduto la risorsa, perché ci interessa solamente controllare il meccanismo entrata ed uscita dallo stato globale. Purtroppo, l'assenza dei generici in Solidity, non ci permette di poter definire una mapping che utilizzi come chiave un address e come valore una lista di

strutture generiche. La mancanza di questa funzionalità è molto limitante: ci costringe a definire il metaprogramma ogni volta come un caso specifico e non è possibile definire una versione generica corretta per un qualunque numero di risorse, con un qualsiasi numero di campi. Nonostante ciò, possiamo arginare il problema in un modo un po' meccanico, implementando uno stato globale differente per ciascuna risorsa del nostro contratto. A questo punto dobbiamo controllare che le risorse non vadano perse, copiate o riutilizzate. Come primo approccio si è utilizzata un'altra mapping (address ⇒ resource), *copiaGlobalState*, inizializzata come una copia del globalState. Ogni operazione a quel punto aggiornerà di volta in volta una o entrambe le mapping. La mint(), per esempio, è una operazione speciale che crea risorsa "dal nulla", perciò aggiorna entrambi gli state. La withdraw() invece, quando viene eseguita ritira la risorsa solamente dal globalState, senza modificare copiaGlobalState. A questo punto, dopo la withdraw(), si possono verificare alcuni casi differenti:

- Se l'esecuzione termina senza altre operazioni (perdita di risorse), ovvero viene chiamata end(), copiaGlobalState sarà diversa dal globalState e verrà eseguita una *revert*.
- Se si esegue deposit() sulla risorsa ritirata, la quantità di globalState verrà ristabilita. Se l'account che possiede la risorsa cambia (es. risorsa A passa da account 1 ad account 2), il meccanismo del last_owner ci permette di aggiornare anche copiaGlobalState, affinchè sia uguale al globalState attuale.
- Se si esegue destroy() sulla risorsa, essa viene tolta dalla copiaGlobalState.
- Se si cerca di eseguire deposit() più volte sulla stessa risorsa (riutilizzo), avremo una differenza fra i due state e perciò anche qui verrà eseguita una *revert*.

Nel caso in cui ci siano più risorse, è possibile utilizzare split() e merge(), per poterle manipolare in locale.

- split(), è definita con la parola chiave *pure*, perché non deve modificare nulla sullo stato globale, deve solamente aggiustare il campo last_owner (A viene splittata in B e C, B e C avranno lo stesso last_owner di A).
- Per merge() (A e B, B viene unita ad A) invece, è necessario modificare copiaGlobalState perché una risorsa viene distrutta ed aggiunta alla prima: si aggiunge al valore di A il valore di B e si esegue destroy() su B.

Di seguito si mostra come sono state implementate deposit(), split(), destroy() ed end(). il resto del codice è possibile consultarlo in [28]:

```
function deposit(resource memory res, address receiver) internal {
   require (safe_run_incorso);
   if (globalState[receiver].value == 0){
      //nuovo address creato
      keys.push(receiver);
   }
   if (res.last_owner != address(0)) {
      copiaGlobalState[res.last_owner].value =
          copiaGlobalState[res.last_owner].value.sub(res.value);
      copiaGlobalState[receiver].value =
          copiaGlobalState[receiver].value.add(res.value);
      // sta tornando sul global state
      res.last_owner = address(0);
   }
   globalState[receiver].value =
       globalState[receiver].value.add(res.value);
}

function split(resource memory res, uint256 valore_da_splittare)
    internal pure returns (resource memory, resource memory){
   require(res.value > valore_da_splittare);
   // creo una nuova risorsa che ritornero' dopo aver spezzato res
   resource memory A = resource(address(0),0);
```

```
    A.value = valore_da_splittare;
    // riduco il valore di res pari al valore che assegnero' ad A
    res.value = res.value.sub(valore_da_splittare);
    A.last_owner = res.last_owner;
    // ritorno la coppia di risorse
    return (res, A);
}

function destroy(resource memory res) internal {
    require (safe_run_incorso);
    if (res.last_owner != address(0)){
        // significa che e' stata ritirata, ora a questo punto la
            voglio distruggere ma senza violare la safety
        // quindi modifico copiaGlobalState
        copiaGlobalState[res.last_owner].value =
            copiaGlobalState[res.last_owner].value.sub(res.value);
    }
}

function end() internal {
    require (safe_run_incorso);
    // controllo che copiaGlobalState sia uguale a globalState
    for(uint256 i; i < keys.length; i++) {
        require(copiaGlobalState[keys[i]].value ==
            globalState[keys[i]].value);
    }
    // fine esecuzione sicura
    safe_run_incorso = false;
}
```

4.2 Lista di operazioni

Questo primo approccio presentato possiede una grossa problematica: il costo in gas. Controllare ogni singola chiave di una mapping è un pattern non consigliato nello sviluppo di contratti in Solidity, specialmente se queste mapping tendono a diventare particolarmente corpose. Siccome stiamo sviluppando una simulazione di un sistema in cui si presume che gli account saranno un numero piuttosto elevato, allora è più conveniente cercare di utilizzare un'altra strategia per poter garantire la sicurezza delle risorse senza spendere un patrimonio in Ether. Analizzando il sistema appena presentato, si può notare come la maggior parte delle operazioni in realtà sia duale, ovvero che ad ogni withdraw dallo stato globale dovrà in qualche modo seguire una deposit o una destroy.

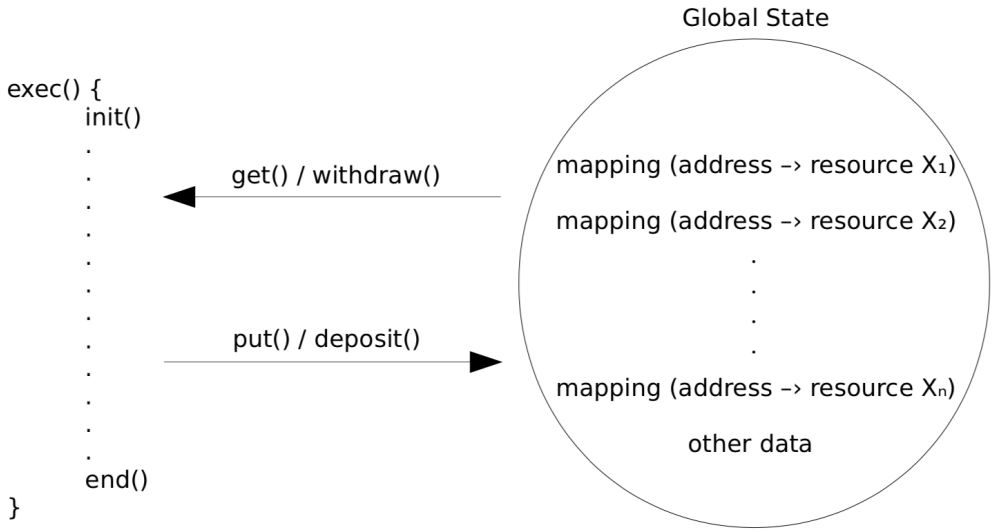

Figura 4.1: Esecuzione

Al posto della copia dello stato globale, si è pensato di utilizzare una lista di operazioni, in cui verranno *pushate* tutte le mosse eseguite su una risorsa che modificano lo stato. Solidity non permette l'implementazione di una lista di tuple, perciò la lista (*check_state_list*) è stata implementata tramite

un array dinamico di strutture, dove la struttura sarà composta da due interi: un campo action (per indicare la get o la put) e un campo value.

```
struct resource {
   uint256 value;
}
//dati per la lista che controlla le modifiche allo state
struct data {
   //0 per get/withdraw, 1 per put/deposit
   uint256 action;
   uint256 value;
}
//operazionList per tracciare il controllo risorse senza
    utilizzare una copia dello state
data[] check_state_list;
// tiene traccia di quanta risorsa e stata ritirata dallo state
// get aumenta il valore, put lo riduce (mai sotto 0, non si puo
    fare put senza prima una get)
// a fine esecuzione dovra essere 0
uint256 _borrowedfromglobalstate;
bool safe_run_incorso;
```

In questo approccio la init() dovrà semplicemente svuotare la lista e inizializzare il booleano safe_run_incorso a true. L'operazione di withdraw() aggiornerà il globalState pari al valore ritirato e pusherà sulla lista di operazioni una coppia (action, value), con action uguale a 0 (get).

```
function withdraw(address address_from, address value_to_withdraw)
    internal returns (resource memory) {
   require (safe_run_incorso);
   resource memory ret;
   data memory tmp;
   require(value_to_withdraw <= globalState[address_from].value);
   globalState[address_from].value =
```

```
        globalState[address_from].value.sub(value_to_withdraw);
    ret.value = value_to_withdraw;
    //push action sulla lista
    tmp.action = 0;
    tmp.value = value_to_withdraw;
    check_state_list.push(tmp);
    return ret;
}
```

Mentre la deposit() esegue l'operazione inversa:

```
function deposit(resource memory res, address receiver) internal {
    require (safe_run_incorso);
    data memory tmp;
    if (globalState[receiver].value == 0){
          //nuovo address creato
          keys.push(receiver);
         }
    globalState[receiver].value =
        globalState[receiver].value.add(res.value);
    //push action sulla lista
    tmp.action = 1;
    tmp.value = res.value;
    check_state_list.push(tmp);
}
```

Il resto delle operazioni si può consultare a [29]. Più particolare è invece la end(), che dovrà assicurare le proprietà delle risorse, andando a svuotare la lista e controllando che ad ogni operazione ne sia stata eseguita una duale.

```
function end() internal {
    data memory tmp;
    require (safe_run_incorso);
    for (uint256 j; j < check_state_list.length; j++){
       tmp = check_state_list[j];
```

```
        //get
        if (tmp.action == 0) {
            _borrowedfromglobalstate =
                _borrowedfromglobalstate.add(tmp.value);
        }
        //put
        else if (tmp.action == 1){
            // partendo dall'inizio della lista, ci saranno sempre
                withdraw prima di deposit, dal momento che la mint non
                ritorna risorsa in locale ma aumenta solamente il
                valore nel global state
            require (_borrowedfromglobalstate >= tmp.value);
            // controllo riutilizzo di risorsa
            _borrowedfromglobalstate =
                _borrowedfromglobalstate.sub(tmp.value);
        }
    }
    //controllo che il bilanciamento sia 0, controllo perdita di
        risorsa
    require (_borrowedfromglobalstate == 0);
    delete check_state_list;
    safe_run_incorso = false;
}
```

Questa implementazione, garantisce un costo in gas molto minore nel caso in cui il numero di account sia elevato, dal momento che vengono controllate solamente le varie operazioni eseguite sullo stato globale. Con n = numero di account ed m = numero di operazioni, operationList è perciò più performante di copiaGlobalState se m $>$ n. L'implementazione copiaGlobalState è più efficiente solo nel caso in cui siano presenti pochi account nel sistema e le operazioni che questi account eseguono sono un numero elevato. Mostriamo la differenza di costo in gas in cui la exec() è la medesima per entrambe le implementazioni. Il primo caso è con 9 account, 4 operazioni. Il secondo caso

gas	3000000 gas
transaction cost	511788 gas
execution cost	663316 gas

Figura 4.3: Esempio costo in gas operationList con n > m

gas	3000000 gas
transaction cost	225378 gas
execution cost	223306 gas

Figura 4.4: Esempio costo in gas CopiaGlobalState con m > n

è con 1 account, 18 operazioni.

gas	3000000 gas
transaction cost	723288 gas
execution cost	721216 gas

Figura 4.2: Esempio costo in gas CopiaGlobalState con n > m

Mostriamo infine degli esempi pratici su come poter scrivere la funzione exec(): quali tipologie di script sono corretti e quali invece causano una *revert*, perché violano le proprietà di sicurezza delle risorse.

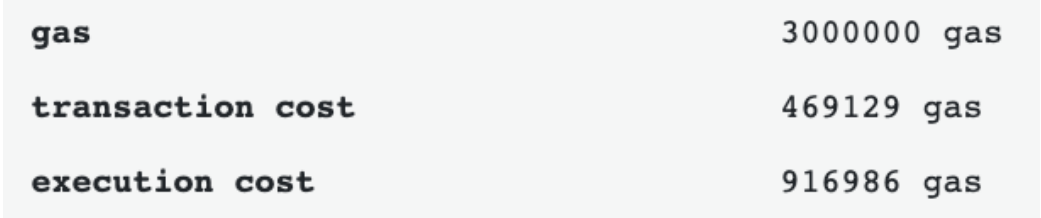

Figura 4.5: Esempio costo in gas operationList con m > n

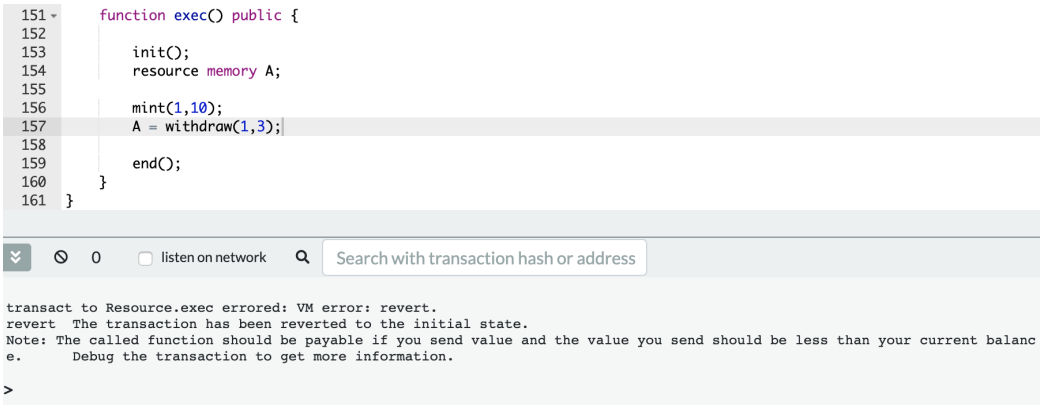

Figura 4.6: Esempio di transaction script in cui avviene perdita di risorsa

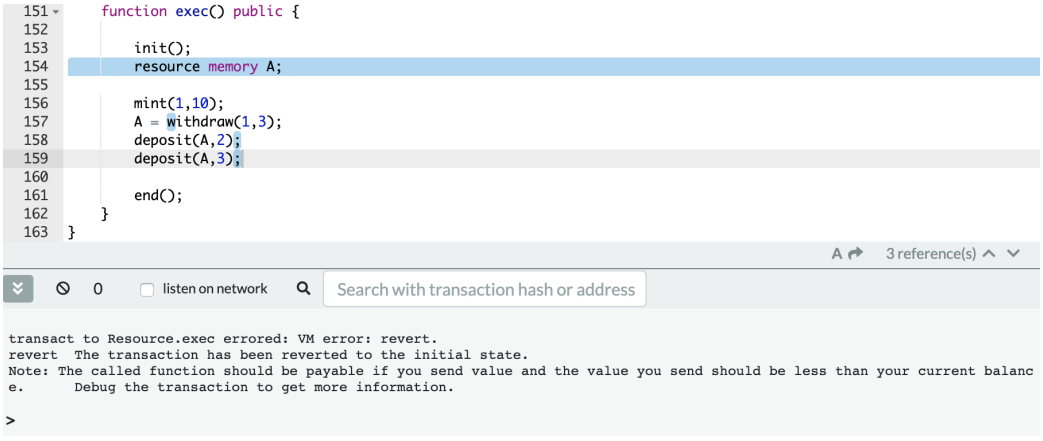

Figura 4.7: Esempio di transaction script in cui si cerca di riutilizzare una risorsa

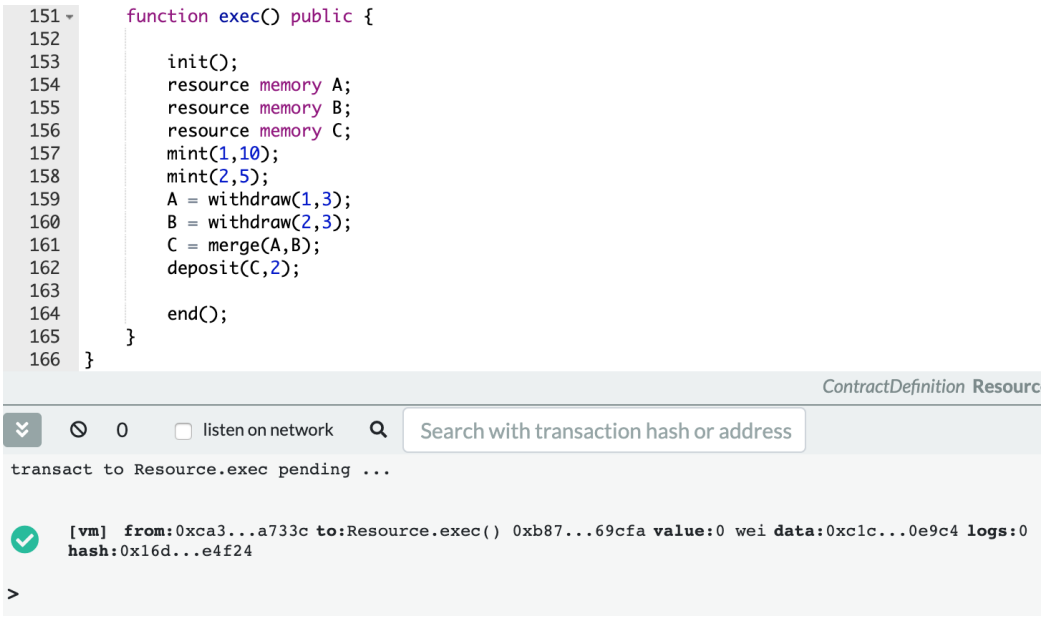

Figura 4.8: Esempio di transaction script corretto

4.3 Gestione di più risorse

Si mostra, in questa sezione, il caso del metaprogramma in cui sono presenti nello stesso contratto più risorse differenti tra loro e come è possibile scrivere operazioni per permettere l'interazione tra di esse. I casi possono essere molteplici, ne mostreremo uno in cui verrà definita una risorsa Coin, con lo stesso pattern della LibraCoin, ed una risorsa ColoredCoin, ovvero una risorsa simile alla Coin ma che possiede anche un colore. Come già accennato nelle sezioni precedenti, si definisce uno stato globale ed una lista di operazioni per ciascuna risorsa.

```
struct resourceCoin {
   uint256 value;
}
struct resourceColoredCoin {
   // 0 blue, 1 red, 2 yellow
   uint256 color;
```

```
    uint256 value;
}
// ambiente per la risorsa Coin
mapping (address => resourceCoin) coinGlobalState;
coinData[] checkCoinStateList;
uint256 _borrowedfromcoinglobalstate;

// ambiente per la risorsa ColoredCoin
mapping (address => resourceColoredCoin) coloredCoinGlobalState;
coloredCoinData[] checkColoredCoinStateList;
// enum come key in mapping non supportato
// 0 per red, 1 per blue, 2 per yellow
// ogni colore ha il suo bilanciamento
mapping (uint256 => uint256) _borrowedfromcoloredcoinglobalstate;
```

L'assenza dei generici ci costringe a scrivere tutta l'interfaccia relativa alle operazioni per ogni risorsa, dal momento che non è possibile passare in input una struttura generica. Quindi avremo per ogni singola operazione, due procedure distinte (coin_mint/colored_mint, coin_withdraw/colored_withdraw e così via).

```
function coin_withdraw(address address_from, uint256
    value_to_withdraw) internal returns (resourceCoin memory) {
   require (safe_run_incorso);
   resourceCoin memory ret;
   coinData memory tmp;
   require(value_to_withdraw <=
       coinGlobalState[address_from].value);
   coinGlobalState[address_from].value =
       coinGlobalState[address_from].value.sub(value_to_withdraw);
   ret.value = value_to_withdraw;
   //push action sulla lista
   tmp.action = 0;
   tmp.value = value_to_withdraw;
```

```
    checkCoinStateList.push(tmp);
    return ret;
}

function coloredcoin_withdraw(address address_from, uint256 color,
     uint256 value_to_withdraw) internal returns
     (resourceColoredCoin memory) {
    require (safe_run_incorso);
    resourceColoredCoin memory ret;
    coloredCoinData memory tmp;
    require(value_to_withdraw <=
        coloredCoinGlobalState[address_from].value);
    coloredCoinGlobalState[address_from].value =
        coloredCoinGlobalState[address_from].value.sub(value_to_withdraw);
    coloredCoinGlobalState[address_from].color = color;
    ret.value = value_to_withdraw;
    ret.color = color;
         //push action sulla lista
    tmp.action = 0;
    tmp.value = value_to_withdraw;
    tmp.color = color;
    checkColoredCoinStateList.push(tmp);
    return ret;
}
```

Le procedure init() ed end() rimangono uniche ma dovranno effettuare controlli su entrambe le liste delle risorse.

```
function init() internal {
    // init Coin state
    _borrowedfromcoinglobalstate = 0;
    delete checkCoinStateList;
    //init Colored Coin state
    for (uint256 i; i < 3; i++) {
```

```
        _borrowedfromcoloredcoinglobalstate[i] = 0;
    }
    delete checkColoredCoinStateList;
    safe_run_incorso = true;
}

function end() internal {
    coinData memory tmp;
    coloredCoinData memory tmp2;
    require (safe_run_incorso);
    for (uint256 j; j < checkCoinStateList.length; j++){
        tmp = checkCoinStateList[j];
        // controllo get, Global State relativo alla risorsa Coin
        if (tmp.action == 0) {
        _borrowedfromcoinglobalstate =
            _borrowedfromcoinglobalstate.add(tmp.value);
        }
        // controllo put, Global State relativo alla risorsa Coin
        else if (tmp.action == 1){
            //partendo dall'inizio, ci saranno sempre withdraw prima
                del deposit
            //dobbiamo evitare riutilizzo di risorsa
            require (_borrowedfromcoinglobalstate >= tmp.value);
            _borrowedfromcoinglobalstate =
                _borrowedfromcoinglobalstate.sub(tmp.value);
        }
    }
    for (uint256 j; j < checkColoredCoinStateList.length; j++){
        tmp2 = checkColoredCoinStateList[j];
        // controllo get, Global State relativo alla risorsa Colored
            Coin
        if (tmp2.action == 0) {
            _borrowedfromcoloredcoinglobalstate[tmp2.color] =
```

```
            _borrowedfromcoloredcoinglobalstate[tmp2.color].add(tmp2.value);
        }
        // controllo put, Global State relativo alla risorsa Colored
            Coin
        else if (tmp2.action == 1){
            //partendo dall'inizio, ci saranno sempre withdraw prima
                del deposit
            //dobbiamo evitare riutilizzo di risorsa
            require (_borrowedfromcoloredcoinglobalstate[tmp2.color]
                >= tmp2.value);
            _borrowedfromcoloredcoinglobalstate[tmp2.color] =
                _borrowedfromcoloredcoinglobalstate[tmp2.color].sub(tmp2.value);
        }
    }

    //controllo che il bilanciamento sia 0, dobbiamo evitare perdita
        di risorsa
    require (_borrowedfromcoinglobalstate == 0);
    require (_borrowedfromcoloredcoinglobalstate[tmp2.color] == 0);
    delete checkCoinStateList;
    delete checkColoredCoinStateList;
    safe_run_incorso = false;
}
```

Si può consultare l'implementazione di tutto il resto delle procedure a [30]. Mostriamo infine come è stata implementata un'operazione particolare di questo caso multi risorsa: color_a_coin, che prende in input una risorsa Coin e ne ritorna una Colored.

```
function color_a_coin (resourceCoin memory res, uint256 color)
    internal returns (resourceColoredCoin memory) {
    resourceColoredCoin memory newColoredCoin;
    coloredCoinData memory tmp;
    require (safe_run_incorso);
```

```
    require (color < 3);
    newColoredCoin.value = res.value;
    newColoredCoin.color = color;
    // push action withdraw sulla lista per evitare si perda la coin
        colorata
    tmp.action = 0;
    tmp.value = res.value;
    tmp.color = color;
    checkColoredCoinStateList.push(tmp);
    // distruggo la coin
    coin_destroy(res);
    return newColoredCoin;
}
```

E di seguito si può infine osservare un esempio di esecuzione in cui si utilizza color_a_coin e non si viola nessuna proprietà di sicurezza delle risorse.

```
function exec() public {
    init();
    resourceCoin memory A;
    resourceColoredCoin memory BlueCoin;
    coin_mint(address(1),8);
    coin_mint(address(2),4);
    coloredcoin_mint(address(1),1,4);
    A = coin_withdraw(address(1),3);
    //coloro la Coin A di blue
    BlueCoin = color_a_coin(A,1);
    coloredcoin_deposit(BlueCoin,address(2));
    end();
}
```

Capitolo 5

Conclusioni

Lo scopo di questa tesi è stato quello di analizzare la blockchain Libra e Move allo stato attuale dell'arte e di confrontare Move con Solidity. Dal momento che Solidity è un linguaggio estremamente flessibile che lascia grande libertà agli sviluppatori (libertà che può anche sfociare in vulnerabilità, specialmente nell'implementazione di custom token), è stata proposta un'implementazione di un metaprogramma che simula la definizione di risorsa, la struttura di dati utilizzata in Move per garantire dei meccanismi di protezione che sono per l'appunto assenti in Solidity. Si noti che Libra e Move sono attualmente in una situazione di "lavori in corso", il progetto di tesi è stato svolto perciò prendendo in considerazione l'attuale versione denominata Move IR e il GitHub di sviluppo su cui si sta implementando la futura versione ufficiale. Alcuni spunti, commenti ed analisi potrebbero perciò non essere più validi quando verrà rilasciata la versione ufficiale del linguaggio, mentre certi pattern ed implementazioni mostrati potrebbero essere aggiornati in futuro. Per quanto riguarda l'effettivo successo di Libra, il mio parere personale dopo aver analizzato nel dettaglio le implementazioni tecniche, è che effettivamente il progetto è da tenere sotto osservazione. Le aziende potrebbero essere più motivate ad investire in una criptovaluta stabile e sponsorizzata da aziende tecnologiche che possiedono una reputazione importante all'interno del mercato globale, rispetto a criptovalute come Ethereum e Bitcoin,

che sono comunque limitate dalla loro nozione di essere totalmente decentralizzate e dal loro valore troppo fluttuante. Inoltre non è da sottovalutare l'enorme utenza di cui dispone Facebook Inc. con le varie piattaforme social che controlla. Un eventuale wallet incorporato in tali applicazioni, con una semplice interfaccia grafica alla portata di tutti, potrebbe davvero spingere l'utente comune all'acquisto di Libra Coin e ad effettuare operazioni di compravendita sempre all'interno di queste piattaforme. Ethereum e Bitcoin, invece, sono libere, ma vengono utilizzate solamente dai più "appassionati" e dai più a scopo speculativo visto il loro andamento estremamente variabile. Infine, affinché che il progetto di Libra ottenga effettivamente un successo globale, per la Libra Association sarà importante che i governi cerchino di regolare questa economia virtuale senza opporre ostruzione (difficile). Chissà cosa succederà.

5.1 Sviluppi futuri

Gli sviluppi futuri sono ovviamente tantissimi, dal momento che stiamo parlando di un ecosistema che deve ancora essere rilasciato ufficialmente. Si possono nominare alcune idee applicabili perlopiù ai contributi proposti. Per esempio, la versione del custom token PaneToken in Move potrebbe essere aggiornata con l'inclusione degli eventi, che al momento sono ancora in fase di testing. Se si tiene in considerazione invece il metaprogramma proposto, allora una tipologia di sviluppo futuro interessante potrebbe essere la simulazione di risorse che hanno un'altra risorsa come campo (risorse di risorse). Questo tipo di struttura non è stata proposta nel metaprogramma, ma è una tipologia di implementazione perfettamente lecita su Move. Per questo tipo di miglioramento, sarebbe particolarmente utile l'inclusione dei generici che garantirebbero una maggior flessibilità nel poter definire gli stati globali e il numero di campi in una risorsa. Quando verrà confermato il costo in gas per le varie operazioni bytecode di Move, si potrebbero svolgere alcuni test, cercando di confrontare la quantità di gas spesa con un transaction script

in Move con la quantità spesa invece con l'esecuzione della funzione exec() in Solidity. Sicuramente sarà più alto in Solidity, ma sarebbe interessante scoprire in che misura. Ancora non è stato definito il momento in cui il costo in gas verrà inserito in Move, ma sicuramente sarà entro la fine del 2020. Per quanto riguarda Ethereum invece, la roadmap [31] di Solidity non ha definito un progetto riguardante i generici, seppur sia già stato postato in *issues* [32], quindi non si può negare un suo eventuale sviluppo prossimo. Nel caso questa eventualità avvenga, il metaprogramma proposto potrebbe essere migliorato in una versione più generica e più facilmente utilizzabile dall'esterno del contratto.

Bibliografia

[1] The Libra Association, "Move: A Language With Programmable Resources". https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources/2019-09-26.pdf

[2] Z. Amsden et al., "The Libra Blockchain" https://developers.libra.org/docs/assets/papers/the-libra-blockchain/2019-09-26.pdf

[3] "Libra Official Github" https://github.com/libra/libra

[4] "Libra Community Forum" https://community.libra.org/

[5] "Move Compiler" https://github.com/libra/libra/tree/master/language/compiler

[6] The Libra Association, "Move Overview" https://developers.libra.org/docs/move-overview

[7] "LibraAccount Module". https://github.com/libra/libra/blob/master/language/stdlib/modules/libra_account.mvir

[8] "Bytecode Verifier". https://github.com/libra/libra/tree/master/language/bytecode-verifier

[9] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System" https://bitcoin.org/bitcoin.pdf

[10] ethereum.org https://ethereum.org/

[11] coinmarketcap.com https://coinmarketcap.com/all/views/all/

[12] en.bitcoin.it https://en.bitcoin.it/wiki/Script

[13] "Introduction to Smart Contracts" https://solidity.readthedocs.io/en/v0.5.3/introduction-to-smart-contracts.html

[14] "Ethereum Virtual Machine Opcodes" https://ethervm.io/

[15] "Solidity Function Types" https://solidity.readthedocs.io/en/latest/types.html#function-types

[16] "Solidity Function Modifiers" https://solidity.readthedocs.io/en/develop/contracts.html#function-modifiers

[17] "LibraCoin Module" https://github.com/libra/libra/blob/master/language/stdlib/modules/libra_coin.mvir

[18] community.libra.org https://community.libra.org/t/move-to-sender-but-with-a-specified-address/201

[19] Fabian Vogelsteller, Vitalik Buterin, "ERC20 Standard". https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

[20] Solidity team, "Solidity" https://solidity.readthedocs.io/en/v0.6.2/

[21] Zeppelin, "OpenZeppelin" https://github.com/OpenZeppelin/openzeppelin-contracts/blob/9b3710465583284b8c4c5d2245749246bb2e0094/contracts/token/ERC20/ERC2

[22] "EIP20 Token Standard" https://github.com/ConsenSys/Tokens/blob/fdf687c69d998266a95f15216b1955a4965a0a6d/contracts/eip20/EIP20.sol

[23] "SafeMath Library" https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol

[24] "PaneToken" https://github.com/adeleveschetti/InnovationLab/tree/master/Codice%20Tesi%20Marco%20Castiglioni/PaneToken

[25] "Remix IDE" https://remix.ethereum.org/

[26] Libra Association, "My First Transaction" https://developers.libra.org/docs/my-first-transaction

[27] github.com/libra/libra/issues https://github.com/libra/libra/issues/225

[28] "Metaprogramma Risorsa" http://bit.ly/2xhqgQG

[29] "Metaprogramma Risorsa e Lista di Operazioni" http://bit.ly/39ODse8

[30] "Metaprogramma con più Risorse e Lista di Operazioni" http://bit.ly/2VXhlhD

[31] github.com/ethereum/solidity/projects https://github.com/ethereum/solidity/projects

[32] github.com/ethereum/solidity/issues https://github.com/ethereum/solidity/issues/869

[33] "Permissioned Blockchain" https://en.wikipedia.org/wiki/Blockchain#Permissioned_(private)_blockchain

[34] "Libra Association" https://libra.org/en-US/association/

[35] Libra Association, "Moving toward permissionless consensus" https://libra.org/permissionless-blockchain

[36] John Yang, "A Technical Dissection of the Libra Blockchain" https://john-b-yang.github.io/libra/

[37] Scott A. Crosby, Dan S. Wallach "Efficient data structures for tamper-evident logging" https://static.usenix.org/event/sec09/tech/full_papers/crosby.pdf

[38] B. Laurie, A. Langley e E. Kasper, "Certificate transparency," RFC, vol. 6962, pp. 1–27, 2013.

[39] John Dantoni, "How to become a founding member of the Libra Association" https://yhoo.it/39kmh45

[40] S. Bano et al., "State machine replication in the Libra Blockchain" https://developers.libra.org/docs/state-machine-replication-paper

[41] M. Yin et al., "Hotstuff: BFT consensus in the lens of blockchain" https://arxiv.org/abs/1803.05069

[42] Leslie Lamport, Robert Shostak, Marshall Pease, "The Byzantine Generals Problem" http://lamport.azurewebsites.net/pubs/byz.pdf

[43] "Libra Reserve" https://libra.org/en-US/about-currency-reserve/#the_reserve

[44] "Calibra, Digital Wallet for Libra Cryptocurrency" https://www.calibra.com/

[45] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in USENIX Symposium on Operating Systems Design and Implementation (OSDI), 1999, pp. 173–186.