Move: A Language With Programmable Resources

Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, Runtian Zhou *

Note to readers: This report was published before the Association released White Paper v2.0, which includes a number of key updates to the Libra payment system. Outdated links have been removed, but otherwise, this report has not been modified to incorporate the updates and should be read in that context.

Abstract. We present *Move*, a safe and flexible programming language for the Libra Blockchain [1][2]. Move is an executable bytecode language used to implement custom transactions and smart contracts. The key feature of Move is the ability to define custom *resource types* with semantics inspired by linear logic [3]: a resource can never be copied or implicitly discarded, only moved between program storage locations. These safety guarantees are enforced statically by Move's type system. Despite these special protections, resources are ordinary program values — they can be stored in data structures, passed as arguments to procedures, and so on. First-class resources are a very general concept that programmers can use not only to implement safe digital assets but also to write correct business logic for wrapping assets and enforcing access control policies. The safety and expressivity of Move have enabled us to implement significant parts of the Libra protocol in Move, including Libra coin, transaction processing, and validator management.

1. Introduction

The advent of the internet and mobile broadband has connected billions of people globally, providing access to knowledge, free communications, and a wide range of lower-cost, more convenient services. This connectivity has also enabled more people to access the financial ecosystem. Yet, despite this progress, access to financial services is still limited for those who need it most.

The mission of Libra is to change this state of affairs [1]. In this paper, we present Move, a new programming language for implementing custom transaction logic and smart contracts in the Libra protocol [2]. To introduce Move, we:

1. Describe the challenges of representing digital assets on a blockchain (Section 2).
2. Explain how our design for Move addresses these challenges (Section 3).
3. Give an example-oriented overview of Move's key features and programming model (Section 4).
4. Dig into the technical details of the language and virtual machine design (Section 5, Section 6, and Appendix A).

Revised September 25th, 2019

5. Conclude by summarizing the progress we have made on Move, describing our plans for language evolution, and outlining our roadmap for supporting third-party Move code on the Libra Blockchain (Section 7).

Audience. This paper is intended for two different audiences:

- Programming language researchers who may not be familiar with blockchain systems. We encourage this audience to read the paper from cover-to-cover, but we warn that we may sometimes refer to blockchain concepts without providing enough context for an unfamiliar reader. Reading [2] before diving into this paper will help, but it is not necessary.
- Blockchain developers who may not be familiar with programming languages research but are interested in learning about the Move language. We encourage this audience to begin with Section 3. We caution that Section 5, Section 6, and Appendix A contain some programming language terminology and formalization that may be unfamiliar.

2. Managing Digital Assets on a Blockchain

We will begin by briefly explaining a blockchain at an abstract level to help the reader understand the role played by a "blockchain programming language" like Move. This discussion intentionally omits many important details of a blockchain system in order to focus on the features that are relevant from a language perspective.

2.1. An Abstract View of a Blockchain

A blockchain is a replicated state machine [4][5]. Replicators in the system are known as *validators*. Users of the system send *transactions* to validators. Each validator understands how to *execute* a transaction to transition its internal state machine from the current state to a new state.

Validators leverage their shared understanding of transaction execution to follow a *consensus protocol* for collectively defining and maintaining the replicated state. If

- the validators start from the same initial state, and
- the validators agree on what the next transaction should be, and
- executing a transaction produces a deterministic state transition,

then the validators will also agree on what the next state is. Repeatedly applying this scheme allows the validators to process transactions while continuing to agree on the current state.

Note that the consensus protocol and the state transition components are not sensitive to each other's implementation details. As long as the consensus protocol ensures a total order among transactions and the state transition scheme is deterministic, the components can interact in harmony.

2.2. Encoding Digital Assets in an Open System

The role of a blockchain programming language like Move is to decide how transitions and state are represented. To support a rich financial infrastructure, the state of the Libra Blockchain must be able to encode the owners of digital assets at a given point in time. Additionally, state transitions should allow the transfer of assets.

There is one other consideration that must inform the design of a blockchain programming language. Like other public blockchains, the Libra Blockchain is an *open* system. Anyone can view the current blockchain state or submit transactions to a validator (i.e., propose state transitions). Traditionally, software for managing digital assets (e.g., banking software) operates in a closed system with special administrative controls. In a public blockchain, all participants are on equal footing. A participant can propose any state transition she likes, yet not all state transitions should be allowed by the system. For example, Alice is free to propose a state transition that transfers assets owned by Bob. The state transition function must be able to recognize that this state transition is invalid and reject it.

It is challenging to choose a representation of transitions and state that encodes ownership of digital assets in an open software system. In particular, there are two properties of physical assets that are difficult to encode in digital assets:

- **Scarcity.** The supply of assets in the system should be controlled. Duplicating existing assets should be prohibited, and creating new assets should be a privileged operation.
- **Access control.** A participant in the system should be able to protect her assets with access control policies.

To build intuition, we will see how these problems arise during a series of strawman proposals for the representation of state transitions. We will assume a blockchain that tracks a single digital asset called a `StrawCoin`. The blockchain state G is structured as a key-value store that maps user identities (represented with cryptographic public keys) to natural number values that encode the `StrawCoin` held by each user. A proposal consists of a *transaction script* that will be evaluated using the given evaluation rule, producing an update to apply to the global state. We will write $\mathrm{G}[K] := n$ to denote updating the natural number stored at key K in the global blockchain state with the value n.

The goal of each proposal is to design a system that is expressive enough to allow Alice to send `StrawCoin` to Bob, yet constrained enough to prevent any user from violating the scarcity or access control properties. The proposals do not attempt to address security issues such as replay attacks [6] that are important, but unrelated to our discussion about scarcity and access control.

Scarcity. The simplest possible proposal is to directly encode the update to the state in the transaction script:

Transaction Script Format	Evaluation Rule
$\langle K, n \rangle$	$\mathrm{G}[K] := n$

This representation can encode sending `StrawCoin` from Alice to Bob. But it has several serious problems. For one, this proposal does not enforce the scarcity of `StrawCoin`. Alice can give herself as many `StrawCoin` as she pleases "out of thin air" by sending the transaction $\langle$Alice, 100$\rangle$. Thus, the `StrawCoin` that Alice sends to Bob are effectively worthless because Bob could just as easily have created those coins for himself.

Scarcity is an important property of valuable physical assets. A rare metal like gold is naturally scarce,

but there is no inherent physical scarcity in digital assets. A digital asset encoded as some sequence of bytes, such as G[Alice] $\rightarrow$ 10, is no physically harder to produce or copy than another sequence of bytes, such as G[Alice] $\rightarrow$ 100. Instead, the evaluation rule must enforce scarcity programmatically.

Let's consider a second proposal that takes scarcity into account:

Transaction Script Format	**Evaluation Rule**
$\langle K_a, n, K_b \rangle$	if G[K_a] $\geq n$ then G[K_a] := G[K_a] - n G[K_b] := G[K_b] + n

Under this scheme, the transaction script specifies both the public key K_a of the sender, Alice, and the public key K_b of the recipient, Bob. The evaluation rule now checks that the number of `StrawCoin` stored under K_a is at least n before performing any update. If the check succeeds, the evaluation rule subtracts n from the `StrawCoin` stored under the sender's key and adds n[1] to the `StrawCoin` stored under the recipient's key. Under this scheme, executing a valid transaction script enforces scarcity by conserving the number of `StrawCoin` in the system. Alice can no longer create `StrawCoin` out of thin air — she can only give Bob `StrawCoin` debited from her account.

Access control. Though the second proposal addresses the scarcity issue, it still has a problem: Bob can send transactions that spend `StrawCoin` belonging to Alice. For example, nothing in the evaluation rule will stop Bob from sending the transaction ⟨Alice, 100, Bob⟩. We can address this by adding an access control mechanism based on digital signatures:

Transaction Script Format	**Evaluation Rule**
$S_{K_a}(\langle K_a, n, K_b \rangle)$	if `verify_sig`($S_{K_a}(\langle K_a, n, K_b \rangle)$) && G[$K_a$] $\geq n$ then G[K_a] := G[K_a] - n G[K_b] := G[K_b] + n

This scheme requires Alice to sign the transaction script with her private key. We write $S_K(m)$ for signing the message m using the private key paired with public key K. The evaluation rule uses the `verify_sig` function to check the signature against Alice's public key K_a. If the signature does not verify, no update is performed. This new rule solves the problem with the previous proposal by using the unforgeability of digital signatures to prevent Alice from debiting `StrawCoin` from any account other than her own.

As an aside, notice that there was effectively no need for an evaluation rule in the first strawman proposal — the proposed state update was applied directly to the key-value store. But as we progressed through the proposals, a clear separation between the preconditions for performing the update and the update itself has emerged. The evaluation rule decides both whether to perform an update and what update to perform by evaluating the script. This separation is fundamental because enforcing access control and scarcity policies inevitably requires some form of evaluation — the user proposes a state change, and computation must be performed to determine whether the state change conforms to the policy. In an open system, the participants cannot be trusted to enforce the policies off-chain and submit direct updates to the state (as in the first proposal). Instead, the access control policies must be enforced on-chain by the evaluation rule.

[1] For simplicity, we will ignore the possibility of integer overflow here.

2.3. Existing Blockchain Languages

`StrawCoin` is a toy language, but it attempts to capture the essence of the Bitcoin Script [7][8] and Ethereum Virtual Machine bytecode [9] languages (particularly the latter). Though these languages are more sophisticated than `StrawCoin`, they face many of the same problems:

1. **Indirect representation of assets.** An asset is encoded using an integer, but an integer value is not the same thing as an asset. *In fact, there is no type or value that represents Bitcoin/Ether/*`StrawCoin`*!* This makes it awkward and error-prone to write programs that use assets. Patterns such as passing assets into/out of procedures or storing assets in data structures require special language support.

2. **Scarcity is not extensible.** The language only represents *one* scarce asset. In addition, the scarcity protections are hardcoded directly in the language semantics. A programmer that wishes to create a custom asset must carefully reimplement scarcity with no support from the language.

3. **Access control is not flexible.** The only access control policy the model enforces is the signature scheme based on the public key. Like the scarcity protections, the access control policy is deeply embedded in the language semantics. It is not obvious how to extend the language to allow programmers to define custom access control policies.

Bitcoin Script. Bitcoin Script has a simple and elegant design that focuses on expressing custom access control policies for spending Bitcoin. The global state consists of a set of unspent transaction output (UTXOs). A Bitcoin Script program provides inputs (e.g., digital signatures) that satisfy the access control policies for the old UTXOs it consumes and specifies custom access control policies for the new UTXOs it creates. Because Bitcoin Script includes powerful instructions for digital signature checking (including multisignature [10] support), programmers can encode a rich variety of access control policies.

However, the expressivity of Bitcoin Script is fundamentally limited. Programmers cannot define custom datatypes (and, consequently, custom assets) or procedures, and the language is not Turing-complete. It is possible for cooperating parties to perform some richer computation via complex multi-transaction protocols [11] or informally define custom assets via "colored coins" [12][13]. However, these schemes work by pushing complexity outside the language and, thus, do not enable true extensibility.

Ethereum Virtual Machine bytecode. Ethereum is a ground-breaking system that demonstrates how to use blockchain systems for more than just payments. Ethereum Virtual Machine (EVM) bytecode programmers can publish *smart contracts* [14] that interact with assets such as Ether and define new assets using a Turing-complete language. The EVM supports many features that Bitcoin Script does not, such as user-defined procedures, virtual calls, loops, and data structures.

However, the expressivity of the EVM has opened the door to expensive programming mistakes. Like `StrawCoin`, the Ether currency has a special status in the language and is implemented in a way that enforces scarcity. But implementers of custom assets (e.g., via the ERC20 [15] standard) do not inherit these protections (as described in (2)) — they must be careful not to introduce bugs that allow duplication, reuse, or loss of assets. This is challenging due to the combination of the indirect representation problem described in (1) and the highly dynamic behavior of the EVM. In particular, transferring Ether to a smart contract involves dynamic dispatch, which has led to a new class of bugs known as *re-entrancy vulnerabilities* [16]. High-profile exploits, such as the DAO attack [17] and the Parity Wallet hack [18], have allowed attackers to steal millions of dollars worth of cryptocurrency.

3. Move Design Goals

The Libra mission is to enable a *simple global currency and financial infrastructure that empowers billions of people* [1]. The Move language is designed to provide a safe, programmable foundation upon which this vision can be built. Move must be able to express the Libra currency and governance rules in a precise, understandable, and verifiable manner. In the longer term, Move must be capable of encoding the rich variety of assets and corresponding business logic that make up a financial infrastructure.

To satisfy these requirements, we designed Move with four key goals in mind: first-class assets, flexibility, safety, and verifiability.

3.1. First-Class Resources

Blockchain systems let users write programs that directly interact with digital assets. As we discussed in Section 2.2, digital assets have special characteristics that distinguish them from the values traditionally used in programming, such as booleans, integers, and strings. A robust and elegant approach to programming with assets requires a representation that preserves these characteristics.

The key feature of Move is the ability to define custom *resource types* with semantics inspired by linear logic [3]: a resource can never be copied or implicitly discarded, only moved between program storage locations. These safety guarantees are enforced statically by Move's type system. Despite these special protections, resources are ordinary program values — they can be stored in data structures, passed as arguments to procedures, and so on. First-class resources are a very general concept that programmers can use not only to implement safe digital assets but also to write correct business logic for wrapping assets and enforcing access control policies.

Libra coin itself is implemented as an ordinary Move resource with no special status in the language. Since a Libra coin represents real-world assets managed by the Libra reserve [19], Move must allow resources to be created (e.g., when new real-world assets enter the Libra reserve), modified (e.g., when the digital asset changes ownership), and destroyed (e.g., when the physical assets backing the digital asset are sold). Move programmers can protect access to these critical operations with *modules*. Move modules are similar to smart contracts in other blockchain languages. A module declares resource types and procedures that encode the rules for creating, destroying, and updating its declared resources. Modules can invoke procedures declared by other modules and use types declared by other modules. However, modules enforce strong *data abstraction* — a type is transparent inside its declaring module and opaque outside of it. Furthermore, critical operations on a resource type `T` may only be performed inside the module that defines `T`.

3.2. Flexibility

Move adds flexibility to Libra via *transaction scripts*. Each Libra transaction includes a transaction script that is effectively the `main` procedure of the transaction. A transaction script is a single procedure that contains arbitrary Move code, which allows customizable transactions. A script can invoke multiple procedures of modules published in the blockchain and perform local computation on the results. This means that scripts can perform either expressive one-off behaviors (such as paying a specific set of recipients) or reusable behaviors (by invoking a single procedure that encapsulates the reusable logic)[2].

[2] By contrast, Ethereum transactions only support the second use-case — they are constrained to invoking a single smart contract method.

Move modules enable a different kind of flexibility via safe, yet flexible code composition. At a high level, the relationship between modules/resources/procedures in Move is similar to the relationship between classes/objects/methods in object-oriented programming. However, there are important differences — a Move module can declare multiple resource types (or zero resource types), and Move procedures have no notion of a `self` or `this` value. Move modules are most similar to a limited version of ML-style modules [20].

3.3. Safety

Move must reject programs that do not satisfy key properties, such as resource safety, type safety, and memory safety. How can we choose an executable representation that will ensure that every program executed on the blockchain satisfies these properties? Two possible approaches are: (a) use a high-level programming language with a compiler that checks these properties, or (b) use low-level untyped assembly and perform these safety checks at runtime.

Move takes an approach between these two extremes. The executable format of Move is a typed bytecode that is higher-level than assembly yet lower-level than a source language. The bytecode is checked on-chain for resource, type, and memory safety by a *bytecode verifier*[3] and then executed directly by a bytecode interpreter. This choice allows Move to provide safety guarantees typically associated with a source language, but without adding the source compiler to the trusted computing base or the cost of compilation to the critical path for transaction execution.

3.4. Verifiability

Ideally, we would check every safety property of Move programs via on-chain bytecode analysis or runtime checks. Unfortunately, this is not feasible. We must carefully weigh the importance and generality of a safety guarantee against the computational cost and added protocol complexity of enforcing the guarantee with on-chain verification.

Our approach is to perform as much lightweight on-chain verification of key safety properties as possible, but design the Move language to support advanced off-chain static verification tools. We have made several design decisions that make Move more amenable to static verification than most general-purpose languages:

1. **No dynamic dispatch.** The target of each call site can be statically determined. This makes it easy for verification tools to reason precisely about the effects of a procedure call without performing a complex call graph construction analysis.
2. **Limited mutability.** Every mutation to a Move value occurs through a reference. References are temporary values that must be created and destroyed within the confines of a single transaction script. Move's bytecode verifier uses a "borrow checking" scheme similar to Rust to ensure that at most one mutable reference to a value exists at any point in time. In addition, the language ensures that global storage is always a tree instead of an arbitrary graph. This allows verification tools to modularize reasoning about the effects of a write operation.
3. **Modularity.** Move modules enforce data abstraction and localize critical operations on resources. The encapsulation enabled by a module combined with the protections enforced by the Move type system ensures that the properties established for a module's types cannot be

[3] This design is similar to the load-time bytecode verification performed by the Java Virtual Machine [21] and Common Language Runtime [22].

violated by code outside the module. We expect this design to enable exhaustive functional verification of important module invariants by looking at a module in isolation without considering its clients.

Static verification tools can leverage these properties of Move to accurately and efficiently check both for the absence of runtime failures (e.g., integer overflow) and for important program-specific functional correctness properties (e.g., the resources locked in a payment channel can eventually be claimed by a participant). We share more detail about our plans for functional verification in Section 7.

4. Move Overview

We introduce the basics of Move by walking through the transaction script and module involved in a simple peer-to-peer payment. The module is a simplified version of the actual Libra coin implementation. The example transaction script demonstrates that a malicious or careless programmer outside the module cannot violate the key safety invariants of the module's resources. The example module shows how to implement a resource that leverages strong data abstraction to establish and maintain these invariants.

The code snippets in this section are written in a variant of the Move intermediate representation (IR). Move IR is high-level enough to write human-readable code, yet low-level enough to have a direct translation to Move bytecode. We present code in the IR because the stack-based Move bytecode would be more difficult to read, and we are currently designing a Move source language (see Section 7). We note that all of the safety guarantees provided by the Move type system are checked at the bytecode level before executing the code.

4.1. Peer-to-Peer Payment Transaction Script

```
public main(payee: address, amount: u64) {
  let coin: 0x0.Currency.Coin = 0x0.Currency.withdraw_from_sender(copy(amount));
  0x0.Currency.deposit(copy(payee), move(coin));
}
```

This script takes two inputs: the account address of the payment's recipient and an unsigned integer that represents the number of coins to be transferred to the recipient. The effect of executing this script is straightforward: `amount` coins will be transferred from the transaction sender to `payee`. This happens in two steps. In the first step, the sender invokes a procedure named `withdraw_from_sender` from the module stored at `0x0.Currency`. As we will explain in Section 4.2, `0x0` is the account address[4] where the module is stored and `Currency` is the name of the module. The value `coin` returned by this procedure is a resource value whose type is `0x0.Currency.Coin`. In the second step, the sender transfers the funds to `payee` by *moving* the `coin` resource value into the `0x0.Currency` module's `deposit` procedure.

This example is interesting because it is quite delicate. Move's type system will reject small variants of the same code that would lead to bad behavior. In particular, the type system ensures that resources can never be duplicated, reused, or lost. For example, the following three changes to the script would be rejected by the type system:

[4] Addresses are 256-bit values that we abbreviate as `0x0`, `0x1`, etc., for convenience.

Duplicating currency by changing `move(coin)` to `copy(coin)`. Note that each usage of a variable in the example is wrapped in either `copy()` or `move()`. Move, following Rust and C++, implements move semantics. Each read of a Move variable `x` must specify whether the usage moves `x`'s value out of the variable (rendering `x` unavailable) or copies the value (leaving `x` available for continued use). Unrestricted values like `u64` and `address` can be both copied and moved. But resource values can *only* be moved. Attempting to duplicate a resource value (e.g., using `copy(coin)` in the example above) will cause an error at bytecode verification time.

Reusing currency by writing `move(coin)` twice. Adding the line `0x0.Currency.deposit(copy(some_other_payee), move(coin))` to the example above would let the sender "spend" `coin` twice — the first time with `payee` and the second with `some_other_payee`. This undesirable behavior would not be possible with a physical asset. Fortunately, Move will reject this program. The variable `coin` becomes unavailable after the first `move`, and the second `move` will trigger a bytecode verification error.

Losing currency by neglecting to `move(coin)`. The Move language implements *linear* [3][23] resources that must be moved **exactly once**[5]. Failing to move a resource (e.g., by deleting the line that contains `move(coin)` in the example above) will trigger a bytecode verification error. This protects Move programmers from accidentally — or intentionally — losing track of the resource. These guarantees go beyond what is possible for physical assets like paper currency.

We use the term *resource safety* to describe the guarantees that Move resources can never be copied, reused, or lost. These guarantees are quite powerful because Move programmers can implement custom resources that also enjoy these protections. As we mentioned in Section 3.1, even the Libra currency is implemented as a custom resource with no special status in the Move language.

4.2. `Currency` Module

In this section, we will show how the implementation of the `Currency` module used in the example above leverages resource safety to implement a secure fungible asset. We will begin by explaining a bit about the blockchain environment in which Move code runs.

Primer: Move execution model. As we explained in Section 3.2, Move has two different kinds of programs: transaction scripts, like the example outlined in Section 4.1 and modules, such as the `Currency` module that we will present shortly. Transaction scripts like the example are included in each user-submitted transaction and invoke procedures of a module to update the global state. Executing a transaction script is all-or-nothing — either execution completes successfully, and all of the writes performed by the script are committed to global storage, or execution terminates with an error (e.g., due to a failed assertion or out-of-gas error), and nothing is committed. A transaction script is a single-use piece of code — after its execution, it cannot be invoked again by other transaction scripts or modules.

By contrast, a module is a long-lived piece of code published in the global state. The module name `0x0.Currency` used in the example above contains the account address `0x0` where the module code is published. The global state is structured as a map from account addresses to accounts.

Each account can contain zero or more modules (depicted as rectangles) and one or more resource values (depicted as cylinders). For example, the account at address `0x0` contains a module `0x0.Currency` and a resource value of type `0x0.Currency.Coin`. The account at address `0x1` has two resources and one module; the account at address `0x2` has two modules and a single resource value.

[5] This is similar to Rust, which implements *affine* resources that can be moved **at most once**.

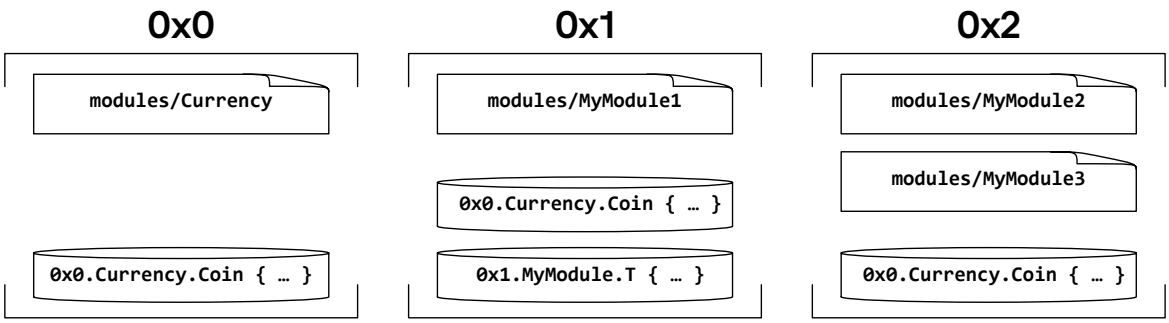

Figure 1: A example global state with three accounts.

Accounts can contain at most one resource value of a given type and at most one module with a given name. The account at address `0x0` would not be allowed to contain an additional `0x0.Currency.Coin` resource or another module named `Currency`. However, the account at address `0x1` could add a module named `Currency`. In that case, `0x0` could also hold a resource of type `0x1.Currency.Coin`. `0x0.Currency.Coin` and `0x1.Currency.Coin` are distinct types that cannot be used interchangeably; the address of the declaring module is part of the type.

Note that allowing at most a single resource of a given type in an account is not restrictive. This design provides a predictable storage schema for top-level account values. Programmers can still hold multiple instances of a given resource type in an account by defining a custom wrapper resource (e.g., `resource TwoCoins { c1: 0x0.Currency.Coin, c2: 0x0.Currency.Coin }`).

Declaring the `Coin` resource. Having explained how modules fit into the Move execution model, we are finally ready to look inside the `Currency` module:

```
module Currency {
  resource Coin { value: u64 }
  // ...
}
```

This code declares a module named `Currency` and a resource type named `Coin` that is managed by the module. A `Coin` is a struct type with a single field value of type `u64` (a 64-bit unsigned integer). The structure of `Coin` is opaque outside of the `Currency` module. Other modules and transaction scripts can only write or reference the `value` field via the public procedures exposed by the module. Similarly, only the procedures of the `Currency` module can create or destroy values of type `Coin`. This scheme enables strong data abstraction — module authors have complete control over the access, creation, and destruction of their declared resources. Outside of the API exposed by the `Currency` module, the only operation another module can perform on a `Coin` is a `move`. Resource safety prohibits other modules from copying, destroying, or double-moving resources.

Implementing `deposit`. Let's investigate how the `Currency.deposit` procedure invoked by the transaction script in the previous section works:

```
public deposit(payee: address, to_deposit: Coin) {
  let to_deposit_value: u64 = Unpack<Coin>(move(to_deposit));
  let coin_ref: &mut Coin = BorrowGlobal<Coin>(move(payee));
  let coin_value_ref: &mut u64 = &mut move(coin_ref).value;
  let coin_value: u64 = *move(coin_value_ref);
  *move(coin_value_ref) = move(coin_value) + move(to_deposit_value);
}
```

At a high level, this procedure takes a `Coin` resource as input and combines it with the `Coin` resource stored in the `payee`'s account. It accomplishes this by:

1. Destroying the input `Coin` and recording its value.
2. Acquiring a reference to the unique `Coin` resource stored under the `payee`'s account.
3. Incrementing the value of payee's `Coin` by the value of the `Coin` passed to the procedure.

There are some aspects of the low-level mechanics of this procedure that are worth explaining. The `Coin` resource bound to `to_deposit` is owned by the `deposit` procedure. To invoke the procedure, the caller needs to move the `Coin` bound to `to_deposit` into the callee (which will prevent the caller from reusing it).

The `Unpack` procedure invoked at the first line is one of several module *builtins* for operating on the types declared by a module. `Unpack<T>` is the only way to delete a resource of type `T`. It takes a resource of type `T` as input, destroys it, and returns the values bound to the fields of the resource. Module builtins like `Unpack` can only be used on the resources declared in the current module. In the case of `Unpack`, this constraint prevents other code from destroying a `Coin`, which, in turn, allows the `Currency` module to set a custom precondition on the destruction of `Coin` resources (e.g., it could choose only to allow the destruction of zero-valued `Coin`s).

The `BorrowGlobal` procedure invoked on the third line is also a module builtin. `BorrowGlobal<T>` takes an address as input and returns a reference to the unique instance of `T` published under that address[6]. This means that the type of `coin_ref` in the code above is `&mut Coin` — a mutable *reference* to a `Coin` resource, not `Coin` — which is an owned `Coin` resource. The next line moves the reference value bound to `coin_ref` in order to acquire a reference `coin_value_ref` to the `Coin`'s `value` field. The final lines of the procedure read the previous value of the payee's `Coin` resource and mutate `coin_value_ref` to reflect the amount deposited[7].

We note that the Move type system cannot catch all implementation mistakes inside the module. For example, the type system will not ensure that the total value of all `Coin`s in existence is preserved by a call to `deposit`. If the programmer made the mistake of writing `*move(coin_value_ref) = 1 + move(coin_value) + move(to_deposit_value)` on the final line, the type system would accept the code without question. This suggests a clear division of responsibilities: it is the programmer's job to establish proper safety invariants for `Coin` inside the confines of the module, and it is the type system's job to ensure that clients of `Coin` outside the module cannot violate these invariants.

Implementing `withdraw_from_sender`. In the implementation above, depositing funds via the `deposit` procedure does not require any authorization — `deposit` can be called by anyone. By contrast, withdrawing from an account must be protected by an access control policy that grants exclusive privileges to the owner of the `Currency` resource. Let us see how the `withdraw_from_sender` procedure called by our peer-to-peer payment transaction script implements this authorization:

```
public withdraw_from_sender(amount: u64): Coin {
  let transaction_sender_address: address = GetTxnSenderAddress();
  let coin_ref: &mut Coin = BorrowGlobal<Coin>(move(transaction_sender_address));
  let coin_value_ref: &mut u64 = &mut move(coin_ref).value;
  let coin_value: u64 = *move(coin_value_ref);
  RejectUnless(copy(coin_value) >= copy(amount));
  *move(coin_value_ref) = move(coin_value) - copy(amount);
  let new_coin: Coin = Pack<Coin>(move(amount));
```

[6] If no instance of `T` exists under the given address, then the builtin will fail.
[7] If the addition on the final line results in an integer overflow, it will trigger a runtime error.

```
  return move(new_coin);
}
```

This procedure is almost the inverse of `deposit`, but not quite. It:

1. Acquires a reference to the unique resource of type `Coin` published under the sender's account.
2. Decreases the value of the referenced `Coin` by the input `amount`.
3. Creates and returns a new `Coin` with value `amount`.

The access control check performed by this procedure is somewhat subtle. The `deposit` procedure allows the caller to specify the address passed to `BorrowGlobal`, but `withdraw_from_sender` can only pass the address returned by `GetTxnSenderAddress`. This procedure is one of several *transaction builtins* that allow Move code to read data from the transaction that is currently being executed. The Move virtual machine authenticates the sender address before the transaction is executed. Using the `BorrowGlobal` builtin in this way ensures that that the sender of a transaction can only withdraw funds from her own `Coin` resource.

Like all module builtins, `BorrowGlobal<Coin>` can only be invoked inside the module that declares `Coin`. If the `Currency` module does not expose a procedure that returns the result of `BorrowGlobal`, there is no way for code outside of the `Currency` module to get a reference to a `Coin` resource published in global storage.

Before decreasing the value of the transaction sender's `Coin` resource, the procedure asserts that the value of the coin is greater than or equal to the input formal `amount` using the `RejectUnless` instruction. This ensures that the sender cannot withdraw more than she has. If this check fails, execution of the current transaction script halts and none of the operations it performed will be applied to the global state.

Finally, the procedure decreases the value of the sender's `Coin` by `amount` and creates a new `Coin` resource using the inverse of `Unpack` — the `Pack` module builtin. `Pack<T>` creates a new resource of type `T`. Like `Unpack<T>`, `Pack<T>` can only be invoked inside the declaring module of resource `T`. Here, `Pack` is used to create a resource `new_coin` of type `Coin` and move it to the caller. The caller now owns this `Coin` resource and can move it wherever she likes. In our example transaction script in Section 4.1, the caller chooses to deposit the `Coin` into the `payee`'s account.

5. The Move Language

In this section, we present a semi-formal description of the Move language, bytecode verifier, and virtual machine. Appendix A lays out all of these components in full detail, but without any accompanying prose. Our discussion here will use excerpts from the appendix and occasionally refer to symbols defined there.

Global state.

$\Sigma \in \mathsf{GlobalState} =$	$\mathsf{AccountAddress} \rightharpoonup \mathsf{Account}$
$\mathsf{Account} =$	$(\mathsf{StructID} \rightharpoonup \mathsf{Resource}) \times (\mathsf{ModuleName} \rightharpoonup \mathsf{Module})$

The goal of Move is to enable programmers to define global blockchain state and securely implement operations for updating global state. As we explained in Section 4.2, the global state is organized as a partial map from addresses to accounts. Accounts contain both resource data values and module

code values. Different resources in an account must have distinct identifiers. Different modules in an account must have distinct names.

Modules.

Module =	ModuleName × (StructName ⇀ StructDecl) × (ProcedureName ⇀ ProcedureDecl)
ModuleID =	AccountAddress × ModuleName
StructID =	ModuleID × StructName
StructDecl =	Kind × (FieldName ⇀ NonReferenceType)

A module consists of a name, struct declarations (including resources, as we will explain shortly), and procedure declarations. Code can refer to a published module using a unique identifier consisting of the module's account address and the module's name. The module identifier serves as a namespace that qualifies the identifiers of its struct types and procedures for code outside of the module.

Move modules enable strong data abstraction. The procedures of a module encode rules for creating, writing, and destroying the types declared by the module. Types are transparent inside their declaring module and opaque outside. Move modules can also enforce preconditions for publishing a resource under an account via the `MoveToSender` instruction, acquiring a reference to a resource under an account via the `BorrowGlobal` instruction, and removing a resource from an account via the `MoveFrom` instruction.

Modules give Move programmers the flexibility to define rich access control policies for resources. For example, a module can define a resource type that can only be destroyed when its `f` field is zero, or a resource that can only be published under certain account addresses.

Types.

PrimitiveType =	AccountAddress ∪ Bool ∪ UnsignedInt64 ∪ Bytes
StructType =	StructID × Kind
$\mathcal{T} \subseteq$ NonReferenceType =	StructType ∪ PrimitiveType
Type ::=	$\mathcal{T}$ | `&mut` $\mathcal{T}$ | `&` $\mathcal{T}$

Move supports primitive types, including booleans, 64-bit unsigned integers, 256-bit account addresses, and fixed-size byte arrays. A struct is a user-defined type declared by a module. A struct type is designated as a resource by tagging it with a resource *kind*. All other types, including non-resource struct types and primitive types, are called *unrestricted* types.

A variable of resource type is a resource variable; a variable of unrestricted type is an unrestricted variable. The bytecode verifier enforces restrictions on resource variables and struct fields of type resource. A resource variable cannot be copied and must always be moved. Both a resource variable and a struct field of resource type cannot be reassigned — doing so would destroy the resource value previously held in the storage location. In addition, a reference to a resource type cannot be dereferenced, since this would produce a copy of the underlying resource. By contrast, unrestricted types can be copied, reassigned, and dereferenced.

Finally, an unrestricted struct type may not contain a field with a resource type. This restriction ensures that (a) copying an unrestricted struct does not result in the copying of a nested resource, and (b) reassigning an unrestricted struct does not result in the destruction of a nested resource.

A reference type may either be mutable or immutable; writes through immutable references are

disallowed. The bytecode verifier performs reference safety checks that enforce these rules along with the restrictions on resource types (see Section 5.2).

Values.

$\mathsf{Resource} =$	$\mathsf{FieldName} \rightharpoonup \mathsf{Value}$
$\mathsf{Struct} =$	$\mathsf{FieldName} \rightharpoonup \mathsf{UnrestrictedValue}$
$\mathsf{UnrestrictedValue} =$	$\mathsf{Struct} \cup \mathsf{PrimitiveValue}$
$v \in \mathsf{Value} =$	$\mathsf{Resource} \cup \mathsf{UnrestrictedValue}$
$g \in \mathsf{GlobalResourceKey} =$	$\mathsf{AccountAddress} \times \mathsf{StructID}$
$ap \in \mathsf{AccessPath} ::=$	$x \mid g \mid ap\ .\ f$
$r \in \mathsf{RuntimeValue} ::=$	$v \mid \texttt{ref}\ ap$

In addition to structs and primitive values, Move also supports reference values. References are different from other Move values because they are transient. The bytecode verifier does not allow fields of reference type. This means that a reference must be created during the execution of a transaction script and released before the end of that transaction script.

The restriction on the shape of struct values ensures the global state is always a tree instead of an arbitrary graph. Each storage location in the state tree can be canonically represented using its *access path* [24] — a path from a root in the storage tree (either a local variable x or global resource key g) to a descendant node marked by a sequence of field identifiers f.

The language allows references to primitive values and structs, but not to other references. Move programmers can acquire references to local variables with the `BorrowLoc` instruction, to fields of structs with the `BorrowField` instruction, and to resources published under an account with the `BorrowGlobal` instruction. The latter two constructs can only be used on struct types declared inside the current module.

Procedures and transaction scripts.

$\mathsf{ProcedureSig} =$	$\mathsf{Visibility} \times (\mathsf{VarName} \rightharpoonup \mathsf{Type}) \times \mathsf{Type}^*$
$\mathsf{ProcedureDecl} =$	$\mathsf{ProcedureSig} \times (\mathsf{VarName} \rightharpoonup \mathsf{Type}) \times [\mathsf{Instr}_\ell]_{\ell=0}^{\ell=i}$
$\mathsf{Visibility} ::=$	$\texttt{public} \mid \texttt{internal}$
$\ell \in \mathsf{InstrIndex} =$	$\mathsf{UnsignedInt64}$
$\mathsf{TransactionScript} =$	$\mathsf{ProcedureDecl}$
$\mathsf{ProcedureID} =$	$\mathsf{ModuleID} \times \mathsf{ProcedureSig}$

A procedure signature consists of visibility, typed formal parameters, and return types. A procedure declaration contains a signature, typed local variables, and an array of bytecode instructions. Procedure visibility may be either `public` or `internal`. Internal procedures can only be invoked by other procedures in the same module. Public procedures can be invoked by any module or transaction script.

The blockchain state is updated by a transaction script that can invoke public procedures of any module that is currently published under an account. A transaction script is simply a procedure declaration with no associated module.

A procedure can be uniquely identified by its module identifier and its signature. The `Call` bytecode instruction requires a unique procedure ID as input. This ensures that all procedure calls in Move are statically determined — there are no function pointers or virtual calls. In addition, the dependency relationship among modules is acyclic by construction. A module can only depend on modules

that were published earlier in the linear transaction history. The combination of an acyclic module dependency graph and the absence of dynamic dispatch enforces a strong execution invariant: all stack frames belonging to procedures in a module must be contiguous. Thus, there is no equivalent of the re-entrancy [16] issues of Ethereum smart contracts in Move modules.

In the rest of this section, we introduce bytecode operations and their semantics (Section 5.1) and describe the static analysis that the bytecode verifier performs before allowing module code to be executed or stored (Section 5.2).

5.1. Bytecode Interpreter

$\sigma \in \mathsf{InterpreterState} =$	$\mathsf{ValueStack} \times \mathsf{CallStack} \times \mathsf{GlobalRefCount} \times \mathsf{GasUnits}$
$vstk \in \mathsf{ValueStack} ::=$	$[] \mid r :: vstk$
$cstk \in \mathsf{CallStack} ::=$	$[] \mid c :: cstk$
$c \in \mathsf{CallStackFrame} =$	$\mathsf{Locals} \times \mathsf{ProcedureID} \times \mathsf{InstrIndex}$
$\mathsf{Locals} =$	$\mathsf{VarName} \rightharpoonup \mathsf{RuntimeValue}$

Move bytecode instructions are executed by a stack-based interpreter similar to the Common Language Runtime [22] and Java Virtual Machine [21]. An instruction consumes operands from the stack and pushes results onto the stack. Instructions may also move and copy values to/from the local variables of the current procedure (including formal parameters).

The bytecode interpreter supports procedure calls. Input values passed to the callee and output values returned to the caller are also communicated via the stack. First, the caller pushes the arguments to a procedure onto the stack. Next, the caller invokes the `Call` instruction, which creates a new call stack frame for the callee and loads the pushed values into the callee's local variables. Finally, the bytecode interpreter begins executing the bytecode instructions of the callee procedure.

The execution of bytecode proceeds by executing operations in sequence unless there is a branch operation that causes a jump to a statically determined offset in the current procedure. When the callee wishes to return, it pushes the return values onto the stack and invokes the `Return` instruction. Control is then returned to the caller, which finds the output values on the stack.

Execution of Move programs is *metered* in a manner similar to the EVM [9]. Each bytecode instruction has an associated *gas* unit cost, and any transaction to be executed must include a gas unit budget. The interpreter tracks the gas units remaining during execution and halts with an error if the amount remaining reaches zero.

We considered both a register-based and a stack-based bytecode interpreter and found that a stack machine with typed locals is a very natural fit for the resource semantics of Move. The low-level mechanics of moving values back and forth between local variables, the stack, and caller/callee pairs closely mirror the high-level intention of a Move program. A stack machine with no locals would be much more verbose, and a register machine would make it more complex to move resources across procedure boundaries.

Instructions. Move supports six broad categories of bytecode instructions:

- Operations such as `CopyLoc`/`MoveLoc` for copying/moving data from local variables to the stack, and `StoreLoc` for moving data from the stack to local variables.
- Operations on typed stack values such as pushing constants onto the stack, and arithmetic/logical operations on stack operands.
- Module builtins such as `Pack` and `Unpack` for creating/destroying the module's declared types, `MoveToSender`/`MoveFrom` for publishing/unpublishing the module's types under an account, and `BorrowField` for acquiring a reference to a field of one of the module's types.
- Reference-related instructions such as `ReadRef` for reading references, `WriteRef` for writing references, `ReleaseRef` for destroying a reference, and `FreezeRef` for converting a mutable reference into an immutable reference.
- Control-flow operations such as conditional branching and calling/returning from a procedure.

- Blockchain-specific builtin operations such as getting the address of the sender of a transaction script and creating a new account.

Appendix A gives a complete list of Move bytecode instructions. Move also provides cryptographic primitives such as `sha3`, but these are implemented as modules in the standard library rather than as bytecode instructions. In these standard library modules, the procedures are declared as `native`, and the procedure bodies are provided by the Move VM. Only the VM can define new native procedures, which means that these cryptographic primitives could instead be implemented as ordinary bytecode instructions. However, native procedures are convenient because the VM can rely on the existing mechanisms for invoking a procedure instead of reimplementing the calling convention for each cryptographic primitive.

5.2. Bytecode Verifier

$C \in \mathsf{Code} =$	$\mathsf{TransactionScript} \cup \mathsf{Module}$
$z \in \mathsf{VerificationResult} ::=$	`ok` | `stack_err` | `type_err` | `reference_err` | ...
$\boxed{C \rightsquigarrow z}$	bytecode verification

The goal of the bytecode verifier is to statically enforce safety properties for any module submitted for publication and any transaction script submitted for execution. No Move program can be published or executed without passing through the bytecode verifier.

The bytecode verifier enforces general safety properties that must hold for any well-formed Move program. We aim to develop a separate offline verifier for program-specific properties in future work (see Section 7).

The binary format of a Move module or transaction script encodes a collection of tables of entities, such as constants, type signatures, struct definitions, and procedure definitions. The checks performed by the verifier fall into three categories:

- Structural checks to ensure that the bytecode tables are well-formed. These checks discover errors such as illegal table indices, duplicate table entries, and illegal type signatures such as a reference to a reference.
- Semantic checks on procedure bodies. These checks detect errors such as incorrect procedure arguments, dangling references, and duplicating a resource.
- Linking uses of struct types and procedure signatures against their declaring modules. These checks detect errors such as illegally invoking an internal procedure and using a procedure identifier that does not match its declaration.

In the rest of this section, we will describe the phases of semantic verification and linking.

Control-flow graph construction. The verifier constructs a control-flow graph by decomposing the instruction sequence into a collection of basic blocks (note that these are unrelated to the "blocks" of transactions in a blockchain). Each basic block contains a contiguous sequence of instructions; the set of all instructions is partitioned among the blocks. Each basic block ends with a branch or return instruction. The decomposition guarantees that branch targets land only at the beginning of some basic block. The decomposition also attempts to ensure that the generated blocks are maximal. However, the soundness of the bytecode verifier does not depend on maximality.

Stack balance checking. Stack balance checking ensures that a callee cannot access stack locations that belong to callers. The execution of a basic block happens in the context of an array of local

variables and a stack. The parameters of the procedure are a prefix of the array of local variables. Passing arguments and return values across procedure calls is done via the stack. When a procedure starts executing, its arguments are already loaded into its parameters. Suppose the stack height is n when a procedure starts executing. Valid bytecode must satisfy the invariant that when execution reaches the end of a basic block, the stack height is n. The verifier ensures this by analyzing each basic block separately and calculating the effect of each instruction on the stack height. It checks that the height does not go below n, and is n at the basic block exit. The one exception is a block that ends in a `Return` instruction, where the height must be $n+m$ (where m is the number of values returned by the procedure).

Type checking. The second phase of the verifier checks that each instruction and procedure (including both builtin procedures and user-defined procedures) is invoked with arguments of appropriate types. The operands of an instruction are values located either in a local variable or on the stack. The types of local variables of a procedure are already provided in the bytecode. However, the types of stack values are inferred. This inference and the type checking of each operation is done separately for each basic block. Since the stack height at the beginning of each basic block is n and does not go below n during the execution of the block, we only need to model the suffix of the stack starting at n for type checking the block instructions. We model this suffix using a stack of types on which types are pushed and popped as the instruction sequence in a basic block is processed. The type stack and the statically known types of local variables are sufficient to type check each bytecode instruction.

Kind checking. The verifier enforces resource safety via the following additional checks during the type checking phase:

- Resources cannot be duplicated: `CopyLoc` is not used on a local variable of kind `resource`, and `ReadRef` is not used on a stack value whose type is a reference to a value of kind `resource`.
- Resources cannot be destroyed: `PopUnrestricted` is not used on a stack location of kind `resource`, `StoreLoc` is not used on a local variable that already holds a resource, and `WriteRef` is not performed on a reference to a value of kind `resource`.
- Resources must be used: When a procedure returns, no local variables may hold a resource value, and the callee's segment of the evaluation stack must only hold the return values of the procedure.

A non-resource struct type cannot have a field of kind `resource`, so these checks cannot be subverted by (e.g.) copying a non-resource struct with a resource field.

Resources cannot be destroyed by a program execution that halts with an error. As we explained in Section 4.2, no state changes produced by partial execution of a transaction script will ever be committed to the global state. This means that resources sitting on the stack or in local variables at the time of a runtime failure will (effectively) return to wherever they were before execution of the transaction began.

In principle, a resource could be made unreachable by a nonterminating program execution. However, the gas metering scheme described in Section 5.1 ensures that execution of a Move program always terminates. An execution that runs out of gas halts with an error, which will not result in the loss of a resource (as we explained above).

Reference checking. The safety of references is checked using a combination of static and dynamic analyses. The static analysis uses borrow checking in a manner similar to the Rust type system, but performed at the bytecode level instead of at the source code level. These reference checks ensure two strong properties:

- All references point to allocated storage (i.e., there are no dangling references).

- All references have safe read and write access. References are either shared (with no write access and liberal read access) or exclusive (with limited read and write access).

To ensure that these properties hold for references into global storage created via `BorrowGlobal`, the bytecode interpreter performs lightweight dynamic reference counting. The interpreter tracks the number of outstanding references to each published resource. It uses this information to halt with an error if a global resource is borrowed or moved while references to that resource still exist.

This reference checking scheme has many novel features and will be a topic of a separate paper.

Linking with global state.

$D \in \mathsf{Dependencies} =$	$\mathsf{StructType}^* \times \mathsf{ProcedureID}^*$
$\mathsf{deps} \in \mathsf{Code} \to \mathsf{Dependencies}$	computing dependencies
$l \in \mathsf{LinkingResult} ::=$	`success` | `fail`
$\boxed{\langle D, \Sigma\rangle \hookrightarrow l}$	linking dependencies with global state

During bytecode verification, the verifier assumes that the external struct types and procedure ID's used by the current code unit exist and are represented faithfully. The linking step checks this assumption by reading the struct and procedure declarations from the global state Σ and ensuring that the declarations match their usage. Specifically, the linker checks that the following declarations in the global state match their usage in the current code unit:

- Struct declarations (name and kind).
- Procedure signatures (name, visibility, formal parameter types, and return types).

6. Move Virtual Machine: Putting It All Together

$T \in \mathsf{Transaction} =$	$\mathsf{TransactionScript} \times \mathsf{PrimitiveValue}^* \times \mathsf{Module}^*$ $\times \mathsf{AccountAddress} \times \mathsf{GasUnits} \ldots$
$B \in \mathsf{Block} =$	$\mathsf{Transaction}^* \times \ldots$
$E \in \mathsf{TransactionEffect} =$	$\mathsf{AccountAddress} \rightharpoonup \mathsf{Account}$
$\mathsf{apply} \in (\mathsf{GlobalState} \times \mathsf{TransactionEffect})$ $\to \mathsf{GlobalState}$	updating global state
$\boxed{\langle T, E, \Sigma\rangle \Downarrow E'}$	transaction evaluation
$\boxed{\langle B, \Sigma\rangle \Downarrow E}$	block evaluation

The role of the Move virtual machine is to execute a block B of transactions from a global state Σ and produce a transaction effect E representing modifications to the global state. The effect E can then be applied to Σ to generate the state Σ' resulting from the execution of B. Separating the effects from the actual state update allows the VM to implement transactional semantics in the case of execution failures.

Intuitively, the transaction effect denotes the update to the global state at a subset of the accounts. A transaction effect has the same structure as the global state: it is a partial map from account addresses to accounts, which contain canonically-serialized representations of Move modules and resource values. The canonical serialization implements a language-independent 1-1 function from a Move module or resource to a byte array.

To execute the block B from state Σ_{i-1}, the VM fetches a transaction T_i from B, processes it to produce an effect E_i, then applies E_i to Σ_{i-1} to produce a state Σ_i to use as the initial state for the next transaction in the block. The effect of the entire block is the ordered composition of the effects of each transaction in the block.

Each transaction is processed according to a workflow that includes steps such as verifying the bytecode in the transaction and checking the signature of the transaction sender. The entire workflow for executing a single transaction is explained in more detail in [2].

Today, transactions in a block are executed sequentially by the VM, but the Move language has been designed to support parallel execution. In principle, executing a transaction could produce a set of reads as well as a set of write effects E. Each transaction in the block could be speculatively executed in parallel and re-executed only if its read/write set conflicts with another transaction in the block. Checking for conflicts is straightforward because Move's tree memory model allows us to uniquely identify a global memory cell using its access path. We will explore speculative execution schemes in the future if virtual machine performance becomes a bottleneck for the Libra Blockchain.

7. What's Next for Move

So far, we have designed and implemented the following components of Move:

- A programming model suitable for blockchain execution.
- A bytecode language that fits this programmable model.
- A module system for implementing libraries with both strong data abstraction and access control.
- A virtual machine consisting of a serializer/deserializer, bytecode verifier, and bytecode interpreter.

Despite this progress, there is a long road ahead. We conclude by discussing some immediate next steps and longer-term plans for Move.

Implementing core Libra Blockchain functionality. We will use Move to implement the core functionality in the Libra Blockchain: accounts, Libra coin, Libra reserve management, validator node addition and removal, collecting and distributing transaction fees, cold wallets, etc. This work is already in progress.

New language features. We will add parametric polymorphism (generics), collections, and events to the Move language. Parametric polymorphism will not undermine Move's existing safety and verifiability guarantees. Our design adds type parameters with kind (i.e, `resource` or `unrestricted`) constraints to procedure and structs in a manner similar to [25].

In addition, we will develop a trusted mechanism for versioning and updating Move modules, transaction scripts, and published resources.

Improved developer experience. The Move IR was developed as a testing tool for the Move bytecode verifier and virtual machine. To exercise these components, the IR compiler must intentionally produce bad bytecode that will be (e.g.) be rejected by the bytecode verifier. This means that although the IR is suitable for prototyping Move programs, it is not particularly user-friendly. To make Move more attractive for third-party development, we will both improve the IR and work toward developing an ergonomic Move source language.

Formal specification and verification. We will create a logical specification language and automated formal verification tool that leverage Move's verification-friendly design (see Section 3.4). The verification toolchain will check program-specific functional correctness properties that go beyond the safety guarantees enforced by the Move bytecode verifier (Section 5.2). Our initial focus is to specify and verify the modules that implement the core functionality of the Libra Blockchain.

Our longer-term goal is to promote a culture of correctness in which users will look to the formal specification of a module to understand its functionality. Ideally, no Move programmer will be willing to interact with a module unless it has a comprehensive formal specification and has been verified to meet to that specification. However, achieving this goal will present several technical and social challenges. Verification tools should be precise and intuitive. Specifications must be modular and reusable, yet readable enough to serve as useful documentation of the module's behavior.

Support third-party Move modules. We will develop a path to third-party module publishing. Creating a good experience for both Libra users and third-party developers is a significant challenge. First, opening the door to general applications must not affect the usability of the system for core payment scenarios and associated financial applications. Second, we want to avoid the reputational risk that scams, speculation, and buggy software present. Building an open system while encouraging high software quality is a difficult problem. Steps such as creating a marketplace for high-assurance modules and providing effective tools for verifying Move code will help.

Acknowledgments

We thank Tarun Chitra, Sophia Drossopoulou, Susan Eisenbach, Maurice Herlihy, John Mitchell, James Prestwich, and Ilya Sergey for their thoughtful feedback on this paper.

A. Move Language Reference

In this appendix, we present the structure of programs and state in the Move bytecode language.

Identifiers
$n \in$ StructName
$f \in$ FieldName
$x \in$ VarName
ProcedureName
ModuleName

Types and Kinds	
$a \in$ AccountAddress	
$b \in$ Bool	
$u \in$ UnsignedInt64	
$\vec{b} \in$ Bytes	
Kind ::=	`resource` | `unrestricted`
ModuleID =	AccountAddress $\times$ ModuleName
StructID =	ModuleID $\times$ StructName
StructType =	StructID $\times$ Kind
PrimitiveType =	AccountAddress $\cup$ Bool $\cup$ UnsignedInt64 $\cup$ Bytes
$\mathcal{T} \subseteq$ NonReferenceType =	StructType $\cup$ PrimitiveType
Type ::=	$\mathcal{T}$ | `&mut` $\mathcal{T}$ | `&` $\mathcal{T}$

Values	
Resource =	FieldName $\rightharpoonup$ Value
Struct =	FieldName $\rightharpoonup$ UnrestrictedValue
PrimitiveValue ::=	a | b | u | $\vec{b}$
UnrestrictedValue =	Struct $\cup$ PrimitiveValue
$v \in$ Value =	Resource $\cup$ UnrestrictedValue
$g \in$ GlobalResourceKey =	AccountAddress $\times$ StructID
$ap \in$ AccessPath ::=	x | g | ap `.` f
$r \in$ RuntimeValue ::=	v | `ref` ap

Global State	
$\Sigma \in$ GlobalState =	AccountAddress $\rightharpoonup$ Account
Account =	(StructID $\rightharpoonup$ Resource) $\times$ (ModuleName $\rightharpoonup$ Module)

Modules and Transaction Scripts	
Module =	ModuleName × (StructName ⇀ StructDecl) × (ProcedureName ⇀ ProcedureDecl)
TransactionScript =	ProcedureDecl
StructDecl =	Kind × (FieldName ⇀ NonReferenceType)
ProcedureSig =	Visibility × (VarName ⇀ Type) × Type*
ProcedureDecl =	ProcedureSig × (VarName ⇀ Type) × $[\text{Instr}_\ell]_{\ell=0}^{\ell=i}$
Visibility ::=	`public` | `internal`
$\ell \in$ InstrIndex =	UnsignedInt64

Interpreter State	
$\sigma \in$ InterpreterState =	ValueStack × CallStack × GlobalRefCount × GasUnits
$vstk \in$ ValueStack ::=	`[]` | r `::` $vstk$
$cstk \in$ CallStack ::=	`[]` | c `::` $cstk$
$c \in$ CallStackFrame =	Locals × ProcedureID × InstrIndex
Locals =	VarName ⇀ RuntimeValue
$p \in$ ProcedureID =	ModuleID × ProcedureSig
GlobalRefCount =	GlobalResourceKey ⇀ UnsignedInt64
GasUnits =	UnsignedInt64

Evaluation	
$T \in$ Transaction =	TransactionScript × PrimitiveValue* × Module* × AccountAddress × GasUnits ...
$B \in$ Block =	Transaction* × ...
$E \in$ TransactionEffect =	AccountAddress ⇀ Account
apply ∈ (GlobalState × TransactionEffect) → GlobalState	updating global state
$\boxed{\langle B, \Sigma\rangle \Downarrow E}$	block evaluation
$\boxed{\langle T, E, \Sigma\rangle \Downarrow E'}$	transaction evaluation
$\boxed{\langle \sigma, E, \Sigma\rangle \Downarrow \sigma', E'}$	interpreter state evaluation

Verification	
$C \in$ Code =	TransactionScript ∪ Module
$z \in$ VerificationResult ::=	`ok` | `stack_err` | `type_err` | `reference_err` | ...
$\boxed{C \rightsquigarrow z}$	bytecode verification
$D \in$ Dependencies =	StructType* × ProcedureID*
deps ∈ Code → Dependencies	computing dependencies
$l \in$ LinkingResult ::=	`success` | `fail`
$\boxed{\langle D, \Sigma\rangle \hookrightarrow l}$	linking dependencies with global state

Instructions	† indicates an instruction whose execution may fail at runtime
LocalInstr ::=	
`MoveLoc`$< x >$	Push value stored in x on the stack. x is now unavailable.
| `CopyLoc`$< x >$	Push value stored in x on the stack.
| `StoreLoc`$< x >$	Pop the stack and store the result in x. x is now available.
| `BorrowLoc`$< x >$	Create a reference to the value stored in x and push it on the stack.
ReferenceInstr ::=	
`ReadRef`	Pop r and push $*r$ on the stack.
| `WriteRef`	Pop two values v and r, perform the write $*r = v$.
| `ReleaseRef`	Pop r and decrement the appropriate refcount if r is a global reference.
| `FreezeRef`	Pop mutable reference r, push immutable reference to the same value.
CallInstr ::=	
`Call`$< p >$	Pop arguments r^*, load into p's formals x^*, transfer control to p.
| `Return`	Return control to the previous frame in the call stack.
ModuleBuiltinInstr ::=	
`Pack`$< n >$	Pop arguments v^*, create struct of type n with f_i: v_i, push it on the stack.
| `Unpack`$< n >$	Pop struct of type n from the stack and push its field values v^* on the stack.
| `BorrowField`$< f >$	Pop reference to a struct and push a reference to field f of the struct.
| `MoveToSender`$< n >^\dagger$	Pop resource of type n and publish it under the sender's address.
| `MoveFrom`$< n >^\dagger$	Pop address a, remove resource of type n from a, push it.
| `BorrowGlobal`$< n >^\dagger$	Pop address a, push a reference to the resource of type n under a.
| `Exists`$< n >$	Pop address a, push bool encoding "a resource of type n exists under a".
TxnBuiltinInstr ::=	
`GetGasRemaining`	Push u64 representing remaining gas unit budget.
| `GetTxnSequenceNumber`	Push u64 encoding the transaction's sequence number.
| `GetTxnPublicKey`	Push byte array encoding the transaction sender's public key.
| `GetTxnSenderAddress`	Push address encoding the sender of the transaction.
| `GetTxnMaxGasUnits`	Push u64 representing the initial gas unit budget.
| `GetTxnGasUnitPrice`	Push u64 representing the Libra coin per gas unit price.
SpecialInstr ::=	
`PopUnrestricted`	Pop a non-resource value.
| `RejectUnless`†	Pop bool b and u64 u, fail with error code u if b is false.
| `CreateAccount`†	Pop address a, create a `LibraAccount.T` under a.
ConstantInstr ::=	
`LoadTrue`	Push true on the stack.
| `LoadFalse`	Push false on the stack.
| `LoadU64`$< u >$	Push the u64 u on the stack.
| `LoadAddress`$< a >$	Push the address a on the stack.
| `LoadBytes`$< \vec{b} >$	Push the byte array $\vec{b}$ on the stack.
StackInstr ::=	
`Not`	Pop boolean b and push $\neg b$.
| `Add`†	Pop two u64's u_1 and u_2 and push $u_1 + u_2$. Fail on overflow.
| `Sub`†	Pop two u64's u_1 and u_2 and push u_1 - u_2. Fail on underflow.
| `Mul`†	Pop two u64's u_1 and u_2 and push $u_1 \times u_2$. Fail on overflow.
| `Div`†	Pop two u64's u_1 and u_2 and push $u_1 \div u_2$. Fail if u_2 is zero.
| `Mod`†	Pop two u64's u_1 and u_2 and push u_1 mod u_2. Fail if u_2 is zero.
| `BitOr`	Pop two u64's u_1 and u_2 and push $u_1 \mid u_2$.
| `BitAnd`	Pop two u64's u_1 and u_2 and push u_1 & u_2.
| `Xor`	Pop two u64's u_1 and u_2 and push $u_1 \oplus u_2$.
| `Lt`	Pop two u64's u_1 and u_2 and push $u_1 < u_2$.

Instructions	† indicates an instruction whose execution may fail at runtime
| `Gt`	Pop two u64's u_1 and u_2 and push $u_1 > u_2$.
| `Le`	Pop two u64's u_1 and u_2 and push $u_1 \leq u_2$.
| `Ge`	Pop two u64's u_1 and u_2 and push $u_1 \geq u_2$.
| `Or`	Pop two booleans b_1 and b_2 and push $b_1 \vee b_2$.
| `And`	Pop two booleans b_1 and b_2 and push $b_1 \wedge b_2$.
| `Eq`	Pop two values r_1 and r_2 and push $r_1 = r_2$.
| `Neq`	Pop two values r_1 and r_2 and push $r_1 \neq r_2$.
ControlFlowInstr ::=	
`Branch`$< \ell >$	Jump to instruction index ℓ in the current procedure.
| `BranchIfTrue`$< \ell >$	Pop boolean, jump to instruction index ℓ in the current procedure if true.
| `BranchIfFalse`$< \ell >$	Pop boolean, jump to instruction index ℓ in the current procedure if false.
Instr =	
LocalInstr	
$\cup$ ReferenceInstr	
$\cup$ CallInstr	
$\cup$ ModuleBuiltinInstr	
$\cup$ TxnBuiltinInstr	
$\cup$ SpecialInstr	
$\cup$ ConstantInstr	
$\cup$ StackInstr	
$\cup$ ControlFlowInstr	

References

[1] The Libra Association, "An Introduction to Libra." https://libra.org/en-us/whitepaper.

[2] Z. Amsden *et al.*, "The Libra Blockchain." https://developers.libra.org/docs/the-libra-blockchain-paper.

[3] J. Girard, "Linear logic," *Theor. Comput. Sci.*, 1987.

[4] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," *ACM Trans. Program. Lang. Syst.*, 1984.

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

[6] M. Saad *et al.*, "Exploring the attack surface of blockchain: A systematic overview," *CoRR*, 2019.

[7] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. http://bitcoin.org/bitcoin.pdf

[8] bitcoin.org, "Bitcoin script.". https://en.bitcoin.it/wiki/Script

[9] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014. https://ethereum.github.io/yellowpaper/paper.pdf

[10] bitcoin.org, "Multisignature.". https://en.bitcoin.it/wiki/Multisignature

[11] M. Bartoletti and R. Zunino, "BitML: A calculus for Bitcoin smart contracts," in *Proceedings of the ACM SIGSAC conference on computer and communications security*, 2018.

[12] bitcoin.org, "Colored coin.". https://en.bitcoin.it/wiki/Colored_Coin

[13] K. Crary and M. J. Sullivan, "Peer-to-peer affine commitment using bitcoin," in *Proceedings of the ACM SIGPLAN conference on programming language design and implementation*, 2015.

[14] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997. https://ojphi.org/ojs/index.php/fm/article/view/548

[15] V. Buterin and F. Vogelsteller, "ERC20 token standard." 2015. https://theethereum.wiki/w/index.php/ERC20_Token_Standard

[16] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts," in *Financial cryptography and data security*, 2017.

[17] V. Buterin, "Critical update re DAO." 2016. https://ethereum.github.io/blog/2016/06/17/critical-update-re-dao-vulnerability

[18] "The Parity wallet hack explained." 2017. https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7

[19] C. Catalini, O. Gratry, J. M. Hou, S. Parasuraman, and N. Wernerfelt, "The Libra reserve." 2019.

[20] R. Milner, M. Tofte, and R. Harper, "Definition of standard ML," 1990.

[21] T. Lindholm and F. Yellin, *The Java virtual machine specification*. Addison-Wesley, 1997.

[22] E. Meijer, R. Wa, and J. Gough, "Technical overview of the common language runtime." 2000.

[23] D. Walker, "Substructural type systems," in *Advanced topics in types and programming languages*, The MIT Press, 2004.

[24] N. D. Jones and S. S. Muchnick, "Flow analysis and optimization of LISP-like structures," in *POPL*, 1979.

[25] K. Mazurak, J. Zhao, and S. Zdancewic, "Lightweight linear types in System F," in *Proceedings of the ACM SIGPLAN international workshop on types in languages design and implementation*, 2010.