Move: A Language With Programmable Resources(中英混合版)

> Move: A Language With Programmable Resources(中文版)

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.
Move是为Libra设计的一种安全且灵活的编程语言。Move 是一种可执行的字节码语言，用于实现自定义事务和智能合约。 Move 的关键特征是能够用线性语义定义资源类型[3]：一个资源永远不能被复制或隐式丢弃，只能在程序存储位置之间移动。 Move是通过静态数据类型来保证的。尽管有这些特殊的保护，但资源是普通的程序值——它们可以存储在数据结构中，作为参数传递给程序，等等。 一流资源是一个非常通用的概念，程序员不仅可以用来实现安全的数字资产，还可以用来编写正确的业务逻辑，用于包装资产和执行访问控制策略。 移动的安全性和表现力使我们能够在移动中实现 Libra 协议的重要部分，包括 Libra 币、交易处理和验证器管理。

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:

互联网和移动宽带的出现已经连接了全球数十亿人，提供了获取知识、免费通信和一系列更低成本、更便捷的服务。这种连接也使更多人能够进入金融生态系统。然而，尽管取得了这一进展，对最需要金融服务的人来说，获得金融服务的机会仍然有限。Libra的使命就是改变这种状态[1]。在本文中，我们提出了Move，一种新的编程语言，用于在 Libra 协议[2]中实现自定义事务逻辑和智能合约。为了介绍Move，我们:

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).
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).

1. 描述在区块链上代表数字资产的挑战（第2节）。
2. 解释我们的Move设计是如何解决这些挑战的(第3节)。
3. 给出一个实例展示Move的关键特性和编程模型概述(第4节)。
4. 深入研究语言和虚拟机设计的技术细节（第5节，第6节和附录 A ）。
5. 总结我们在Move方面取得的进展，描述我们的Move演进计划，并概述我们在Libra的区块链上支持第三方移动代码的规划(第7节)。

Audience. This paper is intended for two different audiences:
观众。本文面向两种不同的观众:

- 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.
- 可能不熟悉区块链系统的编程语言研究人员。我们鼓励这位受众从封面到封面阅读这篇论文，但我们警告说，我们有时可能会参考区块链概念，而没有为一个陌生的读者提供足够的上下文。在潜入这篇论文之前阅读[2]会有帮助，但没有必要。

- 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 withSection 3. We caution that Section 5, Section 6, and Appendix A contain some programming language terminology and formalization that may be unfamiliar.
- 区块链开发人员可能不熟悉编程语言研究，但对学习移动语言感兴趣。我们鼓励这些受众从第3节开始。我们警告第5节、第6节和附录 A 包含一些可能不熟悉的编程语言术语和形式化。

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.
我们将首先从抽象的层面简要解释一个区块链，以帮助读者理解像Move这样的“区块链编程语言”所发挥的作用。这种讨论有意省略了区块链系统的许多重要细节，以便从语言的角度关注相关的特征。

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.
区块链是一个复制状态机[4][5]，系统中的复制器被称为验证器，系统的用户向验证器发送交易，每个验证器都了解如何执行交易，以将其内部的状态机从当前状态过渡到新状态。

Validators leverage their shared understanding of transaction execution to follow a consensus protocol for collectively defining and maintaining the replicated state. If
验证人员利用他们对交易执行的共同理解，遵循共识协议，共同定义和维护复制状态。

- 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,
- 验证器从相同的初始状态开始，并且
- 验证者同意下一个事务应该是什么，
- 执行交易产生确定性的状态转换

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.
然后验证者也将同意下一个状态是什么。重复应用该方案允许验证者在继续同意当前状态的同时处理交易。

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.
请注意，共识协议和状态转换组件对彼此的实现细节不敏感。只要共识协议确保交易之间的总顺序，状态转换方案是确定性的，组件就可以和谐地交互。

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.
像Move这样的区块链编程语言的作用是决定如何表示过渡和状态。为了支持丰富的金融基础设施，Libra Blockchain 的状态必须能够在给定的时间点编码数字资产的所有者。此外，状态过渡应该允许资产的转移。

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.
还有一个其他的考虑必须为区块链编程语言的设计提供信息。 和其他公共区块链一样， Libra Blockchain 是一个开放的系统。 任何人都可以查看当前区块链状态或向验证器提交交易(即提议状态过渡)。 传统上，管理数字资产的软件(如银行软件)在一个具有特殊行政控制的封闭系统中运行。 在一个公共区块链中，所有参与者都是平等的。 参与者可以提出她喜欢的任何状态转换，但不是所有的状态转换都应该被系统允许。 例如，爱丽丝可以自由提出转移鲍勃拥有的资产给自己。 状态转移函数必须能够识别这个状态跃迁是无效的，并拒绝它。

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:
在开放的软件系统中选择转换和编码数字资产所有权的表征是有挑战性的。特别是，在数字资产中，有两种物理资产的属性是难以编码的:

- **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.
- **稀缺性。**系统中的资产供应应该受到控制。应该禁止重复现有资产，创造新的资产应该是一种特权操作。

- **Access control.** A participant in the system should be able to protect her assets with access control policies.
- **访问控制。**系统中的参与者应该能够使用访问控制策略保护她的资产。

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 G[☒] := ☒ to denote updating the natural number stored at key ☒ in the global blockchain state with the value ☒.
为了建立直觉，我们将看到斯特劳曼提出的一系列建议，用来标识状态的转移。 我们将假设一个区块链跟踪一个名为 Straw Coin 的单一数字资产。 区块链状态 G 被抽象为一个键值存储在用户标识的Map(用密码公共密钥表示)中，它将用户身份(用密码公共密钥表示)映射到编码每个用户持有的 Straw Coin 的自然数值。 一个提案由一个交易脚本，该脚本将使用给定的评估规则进行评估，产生一个适用于全局状态的更新。 我们将写G[☒] := ☒，表示更新G[☒]的值为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.
每个提案的目标是设计一个足够表达的系统，允许爱丽丝将 StrawCoin 发送给鲍勃，但足够限制，防止任何用户违反稀缺性或访问控制属性。这些提案没有试图解决重放攻击等安全问题，[6]这些问题很重要，但与我们关于稀缺性和访问控制的讨论无关。

Scarcity. The simplest possible proposal is to directly encode the update to the state in the transaction script:
稀缺性. 最简单的建议是直接将状态更新编码到交易脚本中：

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 ⟨Alice, 100⟩. Thus, the StrawCoin that Alice sends to Bob are effectively worthless because Bob could just as easily have created those coins for himself.
这个表示可以编码从爱丽丝发送 Straw Coin 到鲍勃。但是它有几个严重的问题。首先，这个提议没有强制执行 Straw Coin 的稀缺性。爱丽丝可以通过发送交易<Alice, 100>，“凭空”给自己增加Straw Coin 。因此，爱丽丝发送给鲍勃的 Straw Coin 实际上是没有价值的，因为鲍勃可以很容易地为自己创建这些硬币。

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] → 10, is no physically harder to produce or copy than another sequence of bytes, such as G[Alice] → 100. Instead, the evaluation rule must enforce scarcity programmatically.
稀缺性是有价值的物理资产的重要属性，像黄金这样的稀有金属天然稀缺，但数字资产中不存在固有的物理稀缺性，一个编码为一些字节序列的数字资产，如 G [Alice]→10，在物理上并不比另一个字节序列，如 G [Alice]→100难生产或复制，相反，评估规则必须以编程方式强制稀缺性。

Let’s consider a second proposal that takes scarcity into account:
让我们考虑等而建议来保证账户的稀缺性：

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

Under this scheme, the transaction script specifies both the public key ⊠⊠ of the sender, Alice, and the public key ⊠⊠ of the recipient, Bob. The evaluation rule now checks that the number of StrawCoin stored under ⊠⊠ is at least ⊠ before performing any update. If the check succeeds, the evaluation rule subtracts ⊠ from the StrawCoin stored under the sender’s key and adds ⊠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.
在该方案下，事务脚本既指定了发送方 Alice 的公钥 ⊠⊠ 和接收方 Bob 的公钥⊠⊠。 评估规则现在检查在执行任何更新之前，在⊠⊠账户下的 StrawCoin的数量至少大于⊠ 。如果检查成功，评估规则从发送人密钥下存储的 Straw Coin 中减去 ⊠，并在收件人密钥下存储的 Straw Coin 中添加n。通过在系统中保存 Straw Coin 的数量，执行有效的交易脚本来实现稀缺性。 Alice不再凭空创造 Straw Coin ——她只能从自己的账户给鲍勃 Straw Coin。

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:
访问控制。尽管第二个建议解决了稀缺性问题，但它仍然有一个问题:鲍勃可以发送属于爱丽丝的 StrawCoin 的交易。例如，评估规则中的任何东西都不会阻止鲍勃发送交易⟨Alice, 100, Bob⟩。我们可以通过添加基于数字签名的访问控制机制来解决这个问题:

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

This scheme requires Alice to sign the transaction script with her private key. We write ⊠⊠(⊠) for signing the message ⊠ using the private key paired with public key ⊠. The evaluation rule uses the verify_sig function to check the signature against Alice’ s public key ⊠⊠. 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.
这个方案需要 Alice 用她的私钥来签署事务脚本，我们使用私钥与公钥配对的私钥来签署消息，我们使用私钥与公钥配对的私钥来签署协议，评估规则使用verify_sig函数来对照 Alice 的公钥来检查签名，如果签名没有验证，则不执行更新，这个新规则通过使用数字签名的不可遗忘性来解决了之前建议的问题，以防止 Alice 从除自己之外的任何账户借记 Straw Coin 。

As an aside, notice that there was effectively no need for an evaluation rule in the first strawman pro- posal — 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.
另一方面，注意到在第一个斯特劳曼提议中实际上不需要评估规则——提议的状态更新直接应用于键值存储。 但是随着我们在提案中的进展，执行更新的先决条件和更新本身之间已经出现了明显的分离。 评估规则通过评估脚本来决定是否执行更新和执行什么更新。 这种分离是基本的，因为执行访问控制和稀缺策略不可避免地需要某种形式的评估-用户提出状态变化，必须执行计算来确定状态变化是否符合策略。 在开放系统中，不能信任参与者在链外执行策略，并向状态提交直接更新(如第一项建议)。 相反，访问控制策略必须通过评估规则在链上执行。

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:
StrawCoin 是一种玩具语言，但它试图捕捉比特币脚本[7][8]和以太坊虚拟机字节码[9]语言(特别是后者)的本质。尽管这些语言比 Straw Coin 更复杂，但它们面临着许多相同的问题:

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.
资产的间接表示。资产使用整数编码，但整数值和资产不是一回事。事实上，没有一种类型或值代表比特币/ Ether / Straw Coin！ 这使得编写使用资产的程序变得尴尬和容易出错。将资产传入/传出程序或在数据结构中存储资产等模式需要特殊的语言支持。

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.
稀缺是不可扩展的。语言只代表一种稀缺资产。此外，稀缺保护直接用语言语义硬编码。希望创建自定义资产的程序员必须小心地在语言不支持的情况下重新实现稀缺。

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.
访问控制并不灵活。模型执行的唯一访问控制策略是基于公钥的签名方案。与稀缺性保护一样，访问控制策略深深嵌入了语言语义。如何扩展语言以允许程序员定义自定义访问控制策略并不明显。

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 multi signature [10] support), programmers can encode a rich variety of access control policies.
比特币脚本. Bitcoin Script 具有简洁优雅的设计，专注于表达用于花费比特币的自定义访问控制策略.全局状态由一组未花费的事务输出(UTXOs)组成.一个 Bitcoin Script 程序提供了满足其消耗的旧 UTXOs 的访问控制策略的输入(例如，数字签名)，并为其创建的新 UTXOs 指定自定义访问控制策略.由于 Bitcoin Script 包含了强大的数字签名检查指令(包括多签名[10]支持)，程序员可以编码丰富多样的访问控制策略。

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]. How- ever, these schemes work by pushing complexity outside the language and, thus, do not enable true extensibility.
然而，比特币脚本的表达性从根本上受到限制。程序员无法定义自定义数据类型(以及自定义资产)或程序，语言也不是图灵完备的。合作各方有可能通过复杂的多交易协议执行一些更丰富的计算[11]，或通过“彩色硬币”[12][13]非正式地定义自定义资产。然而，这些方案通过将复杂性推向语言之外的方式工作，因此，无法实现真正的可扩展性。

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.

以太坊虚拟机字节码。以太坊是一个突破性的系统，它演示了如何使用区块链系统不仅仅用于支付。以太坊虚拟机（ EVM ）字节码程序员可以发布与以太等资产交互的智能合约[14]，并使用图灵完备的语言定义新资产。 EVM 支持比特币 Script 没有的许多功能，如用户定义的过程、虚拟调用、循环和数据结构。

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.
然而， EVM 的表现力为昂贵的编程错误打开了大门。 与 StrawCoin 一样，以太币在语言中具有特殊地位，并以一种强化稀缺性的方式实现。 但是自定义资产的实施者(例如，通过 ERC20[15]标准)不继承这些保护(如(2)所述)——他们必须小心不要引入允许重复、重复使用或资产损失的错误。 由于(1)中描述的间接表示问题和 EVM 的高度动态行为相结合，这是具有挑战性的。 特别是，将 Ether 转移到智能合约涉及动态调度，这导致了一类新的 bug 被称为重新进入漏洞[16]。 高调的利用，如 DAO 攻击[17]和 Parity Wallet 攻击[18]，让攻击者窃取了价值数百万美元的加密货币。

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.
Libra的使命是使一个简单的全球货币和金融基础设施能够为数十亿人赋能[1]。Move语言旨在提供一个安全、可编程的基础，在此基础上建立这一愿景。Move必须能够以精确、可理解和可验证的方式表达Libra货币和治理规则。从长远来看，Move必须能够编码构成金融基础设施的丰富的资产种类和相应的业务逻辑。

To satisfy these requirements, we designed Move with four key goals in mind: first-class assets, flexibility, safety, and verifiability.
为了满足这些要求，我们设计了四个关键目标:一流的资产、灵活性、安全性和可验证性。

3.1. First-ClassResources

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 tradi- tionally 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.
区块链系统让用户编写与数字资产直接交互的程序。正如我们在第2.2节中讨论的那样，数字资产具有特殊的特征，这些特征将它们与传统编程中使用的值相区别，如 booleans 、整数和字符串。用资产编程的稳健和优雅的方法需要一种保留这些特征的表示形式。

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.
Move 的关键特征是能够用线性语义定义自定义资源类型[3]：一个资源永远不能被复制或隐式丢弃，只能在程序存储位置之间移动。 这些安全保障是由 Move 的静态类型系统执行的。 尽管有这些特殊的保护，但资源是普通的程序值——它们可以存储在数据结构中，作为参数传递给程序，等等。 一流资源是一个非常通用的概念，程序员不仅可以用来实现安全的数字资产，还可以用来编写正确的业务逻辑，用于包装资产和执行访问控制策略。

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.

Libra本身是一种普通资源，在语言中没有特殊地位。 由于 Libra 币代表 Librac储备管理的现实世界资产[19]， Move 必须允许资源被创建(例如，当新的现实世界资产进入 Libra储备)、被修改(例如，当数字资产改变所有权)和被销毁(例如，当支持数字资产的实物资产被出售)。 Move程序员可以用模块保护对这些关键操作的访问。 Move 模块类似于其他区块链语言中的智能合约。 模块声明资源类型和程序，这些类型和程序编码创建、销毁和更新其声明资源的规则。 模块可以调用其他模块声明的程序，使用其他模块声明的类型。 然而，模块强制执行强大的数据抽象——一种类型在声明模块内部是透明的，在它外部是不透明的。 此外，对资源类型 T 的关键操作只能在定义 T 的模块内执行。

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).
Move 通过交易脚本为 Libra 增加了灵活性。每个 Libra 交易都包括一个交易脚本，它实际上是交易的主要程序。交易脚本是一个包含任意 Move 代码的单个程序，它允许自定义交易。一个脚本可以调用在区块链中发布的模块的多个程序，并对结果执行本地计算。这意味着脚本可以执行表达式一次性行为(例如支付特定的一组收件人)或可重用行为(通过调用封装可重用逻辑的单个程序)。

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].
Move 模块通过安全但灵活的代码构成实现了不同类型的灵活性。在高层次上， Move 中模块/资源/过程之间的关系类似于面向对象编程中的类/对象/方法之间的关系。然而，有重要的区别----一个 Move 模块可以声明多个资源类型(或零资源类型)，而 Move 程序没有自我或这个值的概念。 Move 模块最类似于 ML 风格模块的有限版本[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 必须拒绝不满足关键属性的程序，如资源安全、类型安全和内存安全。我们如何选择一个可执行的表示形式，确保在区块链上执行的每个程序都满足这些属性？两种可能的方法是:(a)使用高级编程语言，配一个检查这些属性的编译器，或者(b)使用低级无类型程序集，并在运行时执行这些安全检查。

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 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.
Move 采取了介于这两个极端之间的方法。Move的可执行格式是一个类型字节码，它比汇编高级别，但比源语言低级别。字节码在链上被一个字节码验证器验证资源、类型和内存安全，然后直接由字节码解释器执行。这种选择允许移动提供通常与源语言相关联的安全保障，但不需要将源代码编译器添加到可信计算基础上，也不需要将编译器编译到事务执行的关键路径上。

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.
理想情况下，我们会通过链上字节码分析或运行时检查Move程序的每一个安全特性。不幸的是，这是不可行的。我们必须仔细权衡安全保证的重要性和通用性，以及通过链上验证执行保证的计算成本和增加的协议复杂性。

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:
我们的方法是对关键安全属性进行尽可能多的轻量级链上验证，但设计Move语言以支持先进的链外静态验证工具。我们已经做出了几个设计决策，使Move比大多数通用语言更适合静态验证:

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.

1. **没有动态调度**。可以静态确定每个调用站点的目标。这使得验证工具很容易准确地解释过程调用的影响，而不需要进行复杂的调用图构造分析。

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 transac- tion 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.
2. **有限的可变性**。Move值的每一次突变都通过引用发生。引用是临时值，必须在单个交易脚本的范围内创建和销毁。Move的字节码验证器使用类似于 Rust 的“借用检查”方案，以确保在任何时间点最多存在一个值的可变引用。此外，该语言确保全局存储永远是树而不是任意图。这允许验证工具模块化关于写操作效果的推理。

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 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.
3. **模块化**。Move模块强制执行数据抽象，并在重源上定位关键操作。模块启用的封装与移动类型系统强制执行的保护相结合，确保模块类型建立的属性不会被模块外部的代码破坏。我们期望这种设计能够通过孤立地观察模块而不考虑其客户端而实现重要模块不变量的详尽的功能验证。

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 Sec- tion 7.
静态验证工具可以利用Move的这些特性来准确和有效地检查运行时是否存在故障(例如整数溢出)和重要的程序特定功能正确性属性(例如，锁定在支付通道中的资源最终可以由参与者要求)。我们在第7节中分享关于功能验证计划的更多细节。

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

implementa- tion. 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.
我们通过简单的点对点支付中涉及的交易脚本和模块来介绍Move的基础知识。该模块是实际 Libra coin 实现的简化版本。示例交易脚本演示了模块之外的恶意或粗心的程序员不能违反模块资源的关键安全不变量。示例模块展示了如何实现利用强数据抽象来建立和维护这些不变量的资源。

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 (seeSection 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.
本节中的代码片段是以Move中间表示(IR)的变体编写的。Move IR 足够高级别，可以编写人类可读的代码，但足够低级别，可以直接翻译到移动字节码。我们在 IR 中展示代码，因为基于堆栈的移动字节码更难读取，我们目前正在设计一种Move源语言(见第7节)。在执行代码之前, 我们注意到移动类型系统提供的所有安全保证都被在字节码级别检查。

4.1. Peer-to-Peer Payment Transaction Script

```
1 public main(payee: address, amount: u64) {
2   let coin: 0x0.Currency.Coin = 0x0.Currency.withdraw_from_sender(copy(amount));
3   0x0.Currency.deposit(copy(payee), move(coin));
4 }
```

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_senderfrom the module stored at 0x0.Currency. As we will explain in Section 4.2, 0x0 is the account address 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.

这个脚本包含两个输入：支付收款方的账户地址和一个表示要转账给收款方的硬币数量的无符号整数。执行这个脚本的效果很简单：金额将从交易发送者转移到收款人。 这是分两步进行的。 在第一步中，发送者从存储在0x0的模块中调用一个名为withdraw_from_senderfrom的过程。货币正如我们将在第4.2节中解释的那样，0x0是存储模块的帐户地址，货币是模块的名称。 此过程返回的值为类型为0x0.Currency.Coin的资源值。在第二步中，发送者通过将硬币资源移动到0x0.Currency(钱币的存放程序。)中，将资金转移给收款人。

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:
这个例子很有趣，因为它非常微妙。 Move 的类型系统将拒绝相同代码的小变体，这些变体会导致不良行为。特别是，类型系统确保资源永远不能重复、重用或丢失。例如，对脚本的以下三个修改将被类型系统拒绝:

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 movesx’ 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.
通过改变移动(硬币)到复制(硬币)来复制货币。 请注意，示例中变量的每个用法都包在 copy ()或 move ()中。 Move遵循 Rust 和 C ++实现了Move语义。 Move变量 x 的每次读取都必须指定使用是将变量的值移动出去(呈现 x 不可用)还是复制该值(留下 x 可用以继续使用)。 U64和地址这样的无限制值既可以复制也可以移动。 但资源价值只能移动。 试图复制资源值（例如，在上面的例子中使用拷贝（硬币））会在字节码验证时间产生错误。

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.
通过写移动(coin)来重复使用货币。向上面的例子添加行 0x0.Currency.deposit(copy(some_other_payee), move(coin)) ，会让发送者两次“消费”硬币-第一

次用收款人，第二次用某些收款人 。这种不良行为将不可能用物理资产。幸运的是，Move会拒绝这个程序。第一次移动后变量硬币变得不可用，第二次移动将触发字节码验证错误。

Losing currency by neglecting to move(coin). The Move language implements linear [3][23] resources that must be moved exactly once. 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.
由于忽略移动(硬币)而丢失货币。移动语言实现了线性[3][23]资源，这些资源必须移动一次。未能移动资源(例如，删除上面例子中包含移动(硬币)的行)将触发字节码验证错误。这保护了Move程序员不小心或故意丢失资源的轨迹。这些保证超出了纸币等实物资产的可能。

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.
我们使用资源安全一词来描述Move资源永远不能被复制、重用或丢失的保证。这些保证相当强大，因为Move程序员可以实现同样享受这些保护的自定义资源。正如我们在第3.1节中提到的，即使是 Libra 货币也作为自定义资源实现，在移动语言中没有特殊地位。

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.
在本节中，我们将展示上面示例中使用的货币模块的实现如何利用资源安全来实现安全的可替代资产。我们将首先解释一点关于移动Move运行的区块链环境。

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 theCurrency 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.
入门：移动执行模型。 正如我们在第3.2节中解释的那样，Move有两种不同的程序:交易脚本，如第4.1节中概述的例子，以及模块，如我们即将介绍的货币模块。 像示例这样的事务脚本包含在每个用户提交的事务中，并调用模块的程序来更新全局状态。 执行事务脚本是成功或失败----要么执行成功完成，脚本执行的所有写入都被承诺到全局存储，要么执行以错误结束(例如，由于失败的断言或出格错误)，任何事情都没有被承诺。 交易脚本是一个单一用途的代码-在执行后，它不能被其他事务脚本或模块再次调用。

By contrast, a module is a long-lived piece of code published in the global state. The module name0x0.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.
相比之下，一个模块是在全局状态下发布的长生命的代码。上面示例中使用的模块名称0x0.Currency包含了发布模块代码的账户地址0x0。全局状态被构造为从账户地址到账户的映射。

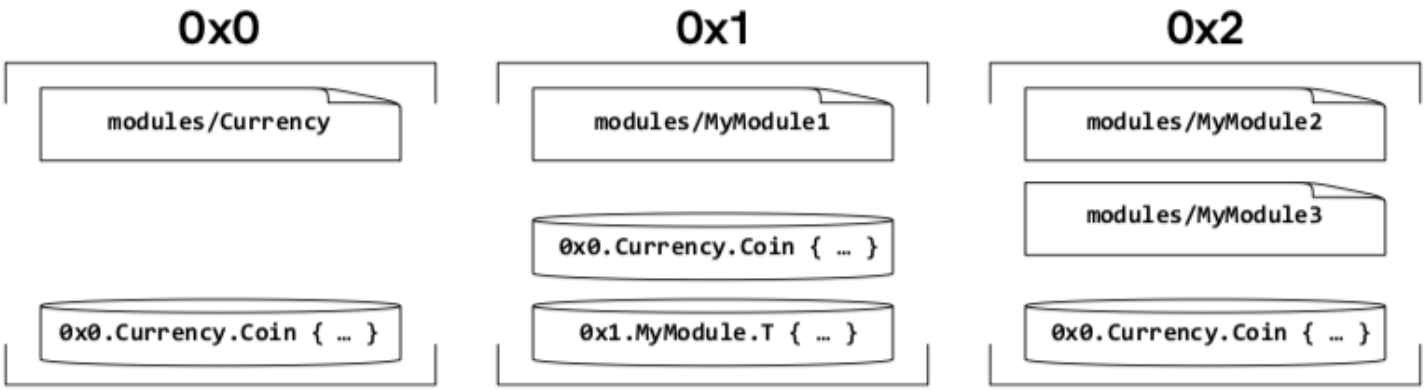

Figure 1: A example global state with three accounts.

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.Currencyand 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.
每个帐户可以包含零或更多的模块(描述为矩形)和一个或多个资源值(描述为圆柱形)。例如，地址0x0的帐户包含一个模块0x0.Currency 和一个类型为0x0.Currency.Coin 的资源值。地址0x1的帐户有两个资源和一个模块；地址0x2的帐户有两个模块和一个单一资源值。

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.Coinresource 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.
帐户最多可以包含一个给定类型的资源值，最多可以包含一个给定名称的模块。地址0x0的帐户将不允许包含一个额外的0x0.Currency.Coin 资源或另一个名为 Currency 的模块。然而，地址0x1的帐户可以添加一个名为 Currency 的模块。在那种情况下，0x0也可以持有0x1.Currency.Coin.0x0.Currency.Coin 和 0x1.Currency.Coin 是不同的类型，不能互换使用；声明模块的地址是类型的一部分。

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 }).
请注意，在帐户中最多允许一个给定类型的单一资源是没有限制性的。这种设计为顶级帐户值提供了一个可预测的存储模式。程序员仍然可以通过定义自定义包装资源(例如，资源 TwoCoins { c1:0x0.Currency.Coin ，c2: 0x0.Currency.Coin }) 在帐户中保存给定资源类型的多个实例。

Declaring the Coin **resource**. Having explained how modules fit into the Move execution model, we are finally ready to look inside the Currency module:
声明硬**币资源**。在解释了模块如何融入Move执行模型后，我们最终准备好查看货币模块:

```
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.

该代码声明了一个名为“货币”的模块和一个由该模块管理的名为“硬币”的资源类型。 Coin 是一种结构类型，单个字段值为 u64(64位无符号整数)。 硬币的结构在货币模块之外是不透明的。 其他模块和事务脚本只能通过模块暴露的公共程序编写或引用值字段。 同样，只有货币模块的过程可以创建或破坏类型 Coin 的值。 该方案支持强大的数据抽象模块——模块作者完全控制其声明资源的访问、创建和销毁。 在货币模块暴露的 API 之外，另一个模块可以在硬币上执行的唯一操作是移动。 资源安全禁止其他模块复制、销毁或重复移动资源。

Implementing deposit. Let’s investigate how the Currency.deposit procedure invoked by the transaction script in the previous section works:
实现存款。让我们来调查上一节交易脚本调用的 Currency.deposit 程序是如何工作的:

```
1 public deposit(payee: address, to_deposit: Coin) {
2   let to_deposit_value: u64 = Unpack<Coin>(move(to_deposit));
3   let coin_ref: &mut Coin = BorrowGlobal<Coin>(move(payee));
4   let coin_value_ref: &mut u64 = &mut move(coin_ref).value;
5   let coin_value: u64 = *move(coin_value_ref);
6   *move(coin_value_ref) = move(coin_value) + move(to_deposit_value);
7 }
```

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:
在高层次上，该过程将硬币资源作为输入，并将其与存储在收款人账户中的硬币资源相结合。它通过以下方法实现这一目标:

1. Destroying the input Coin and recording its value.
1. 销毁输入的 Coin 并记录其价值。

2. Acquiring a reference to the unique Coin resource stored under the payee’s account.
2. 获取对收款人账户下存储的唯一硬币资源的引用。

3. Incrementing the value of payee’s Coin by the value of the Coin passed to the procedure.
3. 通过传递给程序的硬币的价值增加收款人硬币的价值。

There are some aspects of the low-level mechanics of this procedure that are worth explaining. TheCoin 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).
这个过程的低级机制有一些方面值得解释，绑定到to_deposit的 Coin 资源属于存款过程，调用者需要将绑定到 to_deposit 的 Coin 移动到 callee 中（这将阻止调用者重用它）。

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 theCurrency 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 Coins).
在第一行调用的 Unpack 过程是几个模块内置程序之一，用于在模块声明的类型上操作。 Unpack < T > 是删除 T 类型资源的唯一方法。 它将 T 型资源作为输入，破坏它，并返回与资源字段绑定的值。 像 Unpack 这样的模块构建只能在当前模块中声明的资源上使用。 在 Unpack 的情况下，这种约束可以阻止其他代码破坏硬币，这反过来又允许货币模块设置一个关于破坏硬币资源的自定义前提条件(例如，它可以选择只允许销毁零值硬币)。

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 address6. This means that the type of coin_ref in the code above is &mut Coin — a mutablereference 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.
第三行调用的BorrowGlobal程序也是一个模块构建。 BorrowGlobal < T >将地址作为输入，并返回对在该地址下发布的 T 的唯一实例的引用。 这意味着上面代码中的 coin_ref 类型是& mut Coin ——对 Coin 资源的互斥引用，而不是 Coin ——这是一个拥有的 Coin 资源。 下一行移动绑定到 coin_ref 的参考值，以便获得一个参考 coin_value_ref 到 Coin 的值字段。 该程序的最后一行读出收款人硬币资源的前值，并将coin_value_ref 进行修改，以反映存入的金额。

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 Coins 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.
我们注意到， Move 类型系统无法捕捉模块内的所有实现错误。 例如，类型系统不会确保所有存在的硬币的总价值通过存款调用来保存。 如果程序员犯了在最后一行上写*move (coin_value_ref) = 1+ move (coin_value)+ move (to_banking_value)的错误，类型系统将毫无疑问地接受代码。 这表明了一个明确的责任分工:程序员的工作是在模块的范围内为 Coin 建立适当的安全不变量，类型系统的工作是确保模块之外的 Coin 客户端不能违反这些不变量。

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:
实现withdraw_from_sender。在上面的实现中，通过存款程序存入资金不需要任何授权——任何人都可以调用存款。相比之下，从账户中撤回必须受到访问控制策略的保护，该策略授予货币资源所有者独家特权。让我们看看由我们的点对点支付交易脚本调用的撤回withdraw_from_sender程序是如何实现这种授权的:

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

This procedure is almost the inverse of deposit, but not quite. It:
这个程序几乎是存款的逆向，但不是完全。它:

1. Acquires a reference to the unique resource of type Coin published under the sender’ s account.
1.获得对发送者账户下发布的类型 Coin 的唯一资源的引用。

2. Decreases the value of the referenced Coin by the input amount.
2.将引用的 Coin 的值减去输入量。

3. Creates and returns a new Coin with value amount.
3.创建并返回一个价值金额的新硬币。

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 theBorrowGlobal builtin in this way ensures that that the sender of a transaction can only withdraw funds from her own Coin resource.
这个过程执行的访问控制检查有些微妙。存款程序允许调用者指定传递给 BorrowGlobal 的地址，但 withdraw_from_sender 只能传递由 GetTxnSenderAddress 返回的地址。这个过程是允许Move代码从当前正在执行的交易中读取数据的几个交易内置程序之一。Move虚拟机在交易执行前验证发送方地址。通过这种方式使用 BorrowGlobal 内置程序可以确保交易的发送方只能从她自己的 Coin 资源中提取资金。

Like all module builtins, BorrowGlobal<Coin> can only be invoked inside the module that declaresCoin. 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.
像所有模块内置程序一样， BorrowGlobal < Coin >只能在声明 Coin 的模块内调用。如果货币模块没有暴露返回 BorrowGlobal 结果的过程，那么货币模块之外的代码就没有办法获得对在全局存储中发布的 Coin 资源的引用。

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 RejectUnlessinstruction. 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.
在减少交易发送者的硬币资源值之前，该过程使用“拒绝无偿指令”断言硬币的价值大于或等于输入的正式金额。这确保了发送者不能比她提取的更多。如果此检查失败，当前交易脚本的执行将停止，它执行的所有操作都不会应用于全局状态。

Finally, the procedure decreases the value of the sender’ s Coin by amount and creates a new Coinresource 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.
最后，该过程按金额减少发送者的硬币值，并使用 Unpack 的逆创建一个新的硬币资源—— Pack 模块内置。 Pack < T >创建了一个新的 T 型资源，像 Unpack < T >一样， Pack < T >只能在资源 T 的声明模块中调用。这里， Pack 被用来创建一个类型硬币的资源 new_coin，并将其移动到调用者。调用者现在拥有这个硬币资源，并可以将其移动到她喜欢的任何地方。在我们的示例交易脚本中，第4.1节，调用者选择将硬币存入收款人的账户。

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.
在本节中，我们对Move语言、字节码验证器和虚拟机进行了半形式的描述。附录 A 详细地列出了所有这些组件，但没有任何自适应的散文。我们在这里的讨论将使用附录中的摘录，偶尔也会提到在那里定义的符号。

Global state.

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

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.
Move 的目标是使程序员能够定义全局区块链状态，并安全地实现更新全局状态的操作，正如我们在第4.2节中解释的那样，全局状态被组织成从地址到账户的部分映射。账户既包含资源数据值，也包含模块代码值。账户中的不同资源必须有不同的标识符。账户中的不同模块必须有不同的名称。

Modules.

$$
\begin{array}{ll}
\text{Module} = & \text{ModuleName} \times (\text{StructName} \rightharpoonup \text{StructDecl}) \\
& \quad \times (\text{ProcedureName} \rightharpoonup \text{ProcedureDecl}) \\
\text{ModuleID} = & \text{AccountAddress} \times \text{ModuleName} \\
\text{StructID} = & \text{ModuleID} \times \text{StructName} \\
\text{StructDecl} = & \text{Kind} \times (\text{FieldName} \rightharpoonup \text{NonReferenceType})
\end{array}
$$

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.
模块由名称、结构声明(包括资源，我们将在稍后解释)和程序声明组成。代码可以使用由模块的帐户地址和模块名称组成的唯一标识符来指一个已发布的模块。模块标识符作为命名空间，对其结构类型的标识符和模块之外的代码程序进行限定。

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 Move From instruction.
Move模块启用强大的数据抽象。模块的程序编码创建、编写和销毁模块声明的类型的规则。类型在声明模块内部是透明的，外部是不透明的。Move模块还可以强制执行通过 MoveToSender 指令在账户下发布资源的先决条件，通过借出全球指令获取对账户下资源的引用，并通过移动冻结机制从账户中删除资源。

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.
模块使移动程序员能够灵活地为资源定义丰富的访问控制策略，例如，一个模块可以定义一个资源类型，该资源类型只能在其 f 字段为零时被销毁，或者一个资源只能在某些帐户地址下发布。

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.
Move 支持原始类型，包括 booleans 、64位无符号整数、256位地址和固定大小的字节数组。结构是由模块声明的用户定义的类型。结构类型通过用资源类型标记被指定为资源。所有其他类型，包括非资源结构类型和原始类型，都被称为无限制类型。

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.
资源类型的变量是资源变量；非受限类型的变量是非受限变量。 字节码验证器强制限制资源变量并构造类型资源的字段。 资源变量不能被复制，必须始终被移动。 资源变量和资源类型的结构字段都不能重新分配--这样做将破坏先前在存储位置中保存的资源值。 此外，不能取消对资源类型的引用，因为这将产生基础资源的副本。 相比之下，不受限制的类型可以被复制、重新分配和取消引用。

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).
最后，无限制结构类型可能不包含具有资源类型的字段。这种限制确保(a)复制无限制结构不会导致复制嵌套资源，(b)重新分配无限制结构不会导致破坏嵌套资源。
引用类型可以是可变的，也可以是不可变的；不允许通过不可变引用写入。字节码验证器执行引用安全检查，执行这些规则以及对资源类型的限制(见第5.2节)。

Values.

Resource =	FieldName ⇀ Value
Struct =	FieldName ⇀ UnrestrictedValue
UnrestrictedValue =	Struct ∪ PrimitiveValue
$v \in$ Value =	Resource ∪ UnrestrictedValue
$g \in$ GlobalResourceKey =	AccountAddress × StructID
$ap \in$ AccessPath ::=	$x \mid g \mid ap \ . \ f$
$r \in$ 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.
除了结构和原始值之外，Move还支持引用值。引用与其他Move值不同，因为它们是瞬态的。字节码验证器不允许引用类型的字段。这意味着必须在执行交易脚本期间创建引用，并在该事务脚本结束前发布。

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 ☒ or global resource key ☒) to a descendant node marked by a sequence of field identifiers ☒.
结构值形状的限制确保全局状态永远是树而不是任意图。状态树中的每个存储位置都可以使用其访问路径[24]进行规范表示--从存储树中的根(本地变量 ☒ 或全局资源密钥 ☒)到由字段标识符序列 ☒ 标记的后代节点的路径。

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 theBorrowGlobal instruction. The latter two constructs can only be used on struct types declared inside the current module.
该语言允许对原始值和结构的引用，但不允许对其他引用。Move程序员可以用 BorrowLoc 指令获取对局部变量的引用，用 BorrowField 指令获取对结构字段的引用，以及用 BorrowGlobal 指令获取在账户下发布的资源。后两种结构只能用于当前模块中声明的结构类型。

Procedures and transaction scripts.

ProcedureSig =	Visibility × (VarName ⇀ Type) × Type*
ProcedureDecl =	ProcedureSig × (VarName ⇀ Type) × $[\text{Instr}_\ell]_{\ell=0}^{\ell=i}$
Visibility ::=	`public` | `internal`
$\ell \in$ InstrIndex =	UnsignedInt64
TransactionScript =	ProcedureDecl
ProcedureID =	ModuleID × 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. Proce- dure 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.
过程签名由可见性、定义的形式参数和返回类型组成。过程声明包含签名、定义的局部变量和字节码指令。过程可见性可以是公开的也可以是内部的。内部过程只能由同一模块中的其他过程调用。公共过程可以由任何模块或事务脚本调用。

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.
区块链状态由一个交易脚本更新，该脚本可以调用当前在账户下发布的任何模块的公共程序。交易脚本只是一个没有关联模块的程序声明。

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.
一个过程可以通过其模块标识符和签名来唯一识别。 调用字节码指令需要一个唯一的过程 ID 作为输入。 这确保了移动中的所有过程调用都是静态确定的——没有函数指针或虚拟调用。 此外，模块之间的依赖关系是无环结构的。 模块只能依赖线性交易历史记录中早先发布的模块。 无环模块依赖图与动态

调度的缺失相结合，使得执行不变性很强：属于模块中程序的所有堆栈框架必须是连续的。 因此，在Move模块中不存在等价的以太坊智能合约中的重入[16]问题。

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).
在本节的其余部分，我们介绍字节码操作及其语义(第5.1节)，并描述字节码验证器在允许执行或存储模块代码之前执行的静态分析(第5.2节)。

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).
Move字节码指令由基于堆栈的解释器执行，类似于常见的运行时语言 [22]和 Java 虚拟机[21]。指令从堆栈消耗操作数并将结果推送到堆栈。指令也可以移动和复制值到/从当前过程的局部变量(包括形式参数)。

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.
字节码解释器支持过程调用。传递给被调用方的输入值和返回给调用者的输出值也通过堆栈通信。首先，调用者将参数推送到堆栈上。接下来，调用者调用 Call 指令，为被调方创建一个新的调用堆栈框架，并将推送的值加载到被调方的局部变量中。最后，字节码解释器开始执行被调方过程的字节码指令。

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.
字节码的执行是通过按顺序执行操作来进行的，除非在当前过程中有一个分支操作导致跳转到静态确定的偏移。当被调方希望返回时，它将返回值推送到堆栈上，并调用返回指令。然后将控制权返回给调用程序，调用程序在堆栈上找到输出值。

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.
Move程序的执行以类似 EVM 的方式计量[9]。每个字节码指令都有一个相关的气体单元成本，任何要执行的交易都必须包括气体单元预算。解释器跟踪执行过程中剩余的气体单元，如果剩余的数量达到零，就会以错误停止。

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.
我们考虑了基于寄存器和基于堆栈的字节码解释器，发现带有类型化局部的堆栈机非常适合Move的资源语义。在局部变量、堆栈和调用者/被调用者之间来回移动值的低级机制反映了Move程序的高级意图。没有局部的堆栈机制会更冗长，而寄存器机制会使跨越过程边界移动资源变得更复杂。

Instructions. Move supports six broad categories of bytecode instructions:
指令。Move支持六个大类别的字节码指令:

- Operations such as CopyLoc/MoveLoc for copying/moving data from local variables to the stack,
- 如CopyLoc / MoveLoc 等操作，将数据从本地变量复制/移动到堆栈，

- and StoreLoc for moving data from the stack to local variables.
- 和 StoreLoc 用于将数据从堆栈移动到本地变量。

- Operations on typed stack values such as pushing constants onto the stack, and arithmetic/logical operations on stack operands.
- 对类型堆栈值的操作，例如将常量推送到堆栈上，以及算术/逻辑的堆栈操作。

- 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.
- 模块内置程序，如creating/destroying 模块声明类型的 Pack 和 Unpack ，在帐户下 publishing/unpublishing模块类型的 MoveToSender/MoveFrom ，以及获取模块类型中一个字段的引用的 BorrowField 。

- 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.
- 引用相关的指令，如读取引用的 ReadRef ，写入引用的 WriteRef ，销毁引用的 ReleaseRef ，以及将可变引用转换为不可变引用的 FreezeRef 。

- Control-flow operations such as conditional branching and calling/returning from a procedure.
- 控制流操作，如条件分枝和从过程中调用/返回。

- Blockchain-specific builtin operations such as getting the address of the sender of a transaction script and creating a new account.
- 区块链特有的构建操作，如获取交易脚本的发送方地址和创建新账户。

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 pro- cedures, 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.
附录 A 给出了Move字节码指令的完整列表。 Move 还提供了像 sha3这样的密码基元，但这些基元是作为标准库中的模块而不是字节码指令实现的。 在这些标准库模块中，程序被声明为本地程序，程序主体由Move的VM 提供。 只有虚拟机可以定义新的原生程序，这意味着这些密码基元可以作为普通字节码指

令实现。 然而，本地程序是方便的，因为 VM 可以依赖现有的机制来调用程序，而不是重新实现每个密码原始程序的调用公约。

5.2. Bytecode Verifier

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

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.
字节码验证器的目标是静态地执行提交发布的任何模块和提交执行的任何事务脚本的安全属性。没有通过字节码验证器，任何移动程序都不能发布或执行。

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).
字节码验证器强制执行任何良好形成的Move程序必须具有的一般安全属性。我们的目标是在未来的工作中为特定程序的属性开发一个单独的离线验证器(见第7节)。

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:
Move模块或交易脚本的二进制格式编码实体表的集合，如常量、类型签名、结构定义和过程定义。验证者执行的检查分为三类:

- 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.
- 结构检查以确保字节码表是良好的格式。这些检查发现错误，如非法表索引、重复表条目和非法类型签名，如引用引用。

- Semantic checks on procedure bodies. These checks detect errors such as incorrect procedure arguments, dangling references, and duplicating a resource.
- 对过程体进行语义检查。这些检查检测错误，如不正确的过程参数、悬挂引用和复制资源。

- 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.
- 将结构类型和程序签名与其声明模块联系起来。这些检查检测错误，如非法调用内部程序和使用与其声明不匹配的程序标识符。

In the rest of this section, we will describe the phases of semantic verification and linking.
在本节的其余部分，我们将描述语义验证和链接的阶段。

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.
控制流图构造。 验证者通过将指令序列分解为基本块的集合（注意，这些与区块链中交易的"块"无关），来构建一个控制流图。 每个基本块都包含一个连续的指令序列；所有指令的集合在块之间进行分区。 每个基本块以分支或返回指令结束。 分解保证了分支目标只在一些基本块的开始落地。 分解还试图确保生成的块是最大的。 然而，字节码验证器的健全度并不取决于最大值。

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).
堆栈平衡检查。 堆栈平衡检查可确保调用者无法访问属于调用者的堆栈位置。 基本块的执行发生在一个局部变量数组和一个堆栈的上下文中。 过程的参数是局部变量数组的前缀。 通过堆栈传递过程调用中的参数和返回值。 当一个过程开始执行时，它的参数已经加载到它的参数中。 假设当一个过程开始执行时，堆栈高度是n。 有效的字节码必须满足不变量，即当执行到达基本块的末端时，堆栈高度为 n 。 验证器通过分别分析每个基本块并计算每个指令对堆栈高度的影响来确保这一点。 它检查高度不低于 n ，并在基本块出口处为 n 。 一个例外是以返回指令结束的块，其中高度必须是 n + m (其中 m 是过程返回的值的个数)。

Type checking. The second phase of the verifier checks that each instruction and procedure (includ- ing 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.
类型检查。 验证器的第二阶段检查每个指令和过程（包括构建过程和用户定义过程）是否被调用了适当类型的参数。 指令的操作数是位于局部变量或堆栈中的值。 字节码中已经提供了过程的局部变量类型。 但是，推断出堆栈值的类型。 这个推论和每个操作的类型检查是针对每个基本块单独进行的。 由于每个基本块开头的堆栈高度是 n ，在块执行过程中不会低于 n ，因此我们只需要对从 n 开始的堆栈后缀进行建模，以进行块指令的类型检查。 我们使用类型堆栈建模这个后缀，在处理基本块中的指令序列时，类型被推送和弹出。 类型堆栈和静态已知的局部变量类型足以键入检查每个字节码指令。

Kind checking. The verifier enforces resource safety via the following additional checks during the type checking phase:
种类检查。验证者在类型检查阶段通过以下额外检查来确保资源安全:

- Resources cannot be duplicated: CopyLoc is not used on a local variable of kind resource, andReadRef is not used on a stack value whose type is a reference to a value of kind resource.
- 资源不能重复： CopyLoc 不能使用在种类资源的本地变量上，而 ReadRef 不能使用在栈值上，栈值的类型是对种类资源的值的引用。

- 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 WriteRefis not performed on a reference to a value of kind resource.
- 资源不能被销毁: PopUnlimited 不用于种类资源的堆栈位置， StoreLoc 不用于已经持有资源的本地变量，而 WriteRef 不用于对种类资源的值的引用。

- 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.
- 必须使用资源：当一个过程返回时，任何局部变量都不能保存一个资源值，而评估栈的被调者部分只能保存这个过程的返回值。

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.
非资源结构类型不可能有一个种类资源的字段，因此这些检查不能通过(例如)用资源字段复制非资源结构来颠覆。

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.
资源不能被错误停止的程序执行破坏。正如我们在第4.2节中解释的那样，交易脚本的部分执行不会产生任何状态变化，这意味着运行时故障时堆栈上或本地变量中的资源将(实际上)返回到事务开始执行之前的任何地方。

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).
原则上，非终止程序执行可能使资源无法到达。然而，第5.1节描述的气体计量方案确保了移动程序的执行总是终止。耗尽气体的执行会随着错误而停止，这不会导致资源的损失(正如我们上文所解释的)。

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:

引用检查。引用的安全性使用静态和动态分析的结合来检查。静态分析以类似于 Rust 类型系统的方式使用借用检查，但在字节码级别而不是在源代码级别执行。这些引用检查确保了两个强大的特性:

- All references point to allocated storage (i.e., there are no dangling references).
- 所有引用都指向分配的存储(即没有悬挂引用)。

- 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).
- 所有引用都有安全的读写访问。引用要么是共享的(没有写入访问和自由读取访问)，要么是排他性的(有有限的读写访问)。

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.
为了确保这些属性为通过 BorrowGlobal 创建的全局存储提供参考，字节码解释器执行轻量级的动态参考计数。解释器跟踪每个已发布资源的未完成引用的数量。如果全局资源被借用或移动，而该资源的引用仍然存在，它使用此信息停止错误。

This reference checking scheme has many novel features and will be a topic of a separate paper.
该参考资料核对方案有许多新的特点，将作为单独一篇论文的主题。

Linking with global state.

$D \in \text{Dependencies} =$	$\text{StructType}^* \times \text{ProcedureID}^*$
$\text{deps} \in \text{Code} \rightarrow \text{Dependencies}$	computing dependencies
$l \in \text{LinkingResult} ::=$	`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:
在字节码验证过程中，验证器假设当前代码单元使用的外部结构类型和过程 ID 存在并被忠实地表示。链接步骤通过读取来自全局状态的结构Σ和过程声明并确保声明符合它们的用途来验证这一假设。具体来说，链接器检查全局状态下的下列声明是否符合它们在当前代码单元中的用途:

- Struct declarations (name and kind).
- 结构化声明(名称和种类)。

- Procedure signatures (name, visibility, formal parameter types, and return types).
- 程序签名（名称、可见性、正式参数类型和返回类型）。

6. Move Virtual Machine: Putting It All Together

$T \in \mathsf{Transaction} =$	$\mathsf{TransactionScript} \times \mathsf{PrimitiveValue}^* \times \mathsf{Module}^*$ $\times \mathsf{AccountAddress} \times \mathsf{GasUnits} \ldots$
$B \in \mathsf{Block} =$	$\mathsf{Transaction}^* \times \ldots$
$E \in \mathsf{TransactionEffect} =$	$\mathsf{AccountAddress} \rightharpoonup \mathsf{Account}$
$\mathrm{apply} \in (\mathsf{GlobalState} \times \mathsf{TransactionEffect})$ $\rightarrow \mathsf{GlobalState}$	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 ☒ of transactions from a global state Σ and produce a transaction effect ☒ representing modifications to the global state. The effect ☒ can then be applied to Σ to generate the state Σ′ resulting from the execution of ☒. Separating the effects from the actual state update allows the VM to implement transactional semantics in the case of execution failures.
Move 虚拟机的作用是，从全局状态 Σ 执行块 ☒ 的交易，并产生代表对全局状态 Σ 修改的交易影响☒。然后可以将效果 ☒ 应用到 全局状态 Σ 上，生成执行☒所产生的状态Σ′，将效果与实际状态更新分离，允许 VM 在执行失败的情况下实现事务语义。

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.
直观地说，交易效果表示在帐户的子集上更新到全局状态。交易效果具有与全局状态相同的结构:它是一个从帐户地址到帐户的部分映射，其中包含对Move模块和资源值的标准序列化表示。标准序列化实现了从Move模块或资源到字节数组的语言无关的1-1函数。

To execute the block □ from state Σ□−1, the VM fetches a transaction □□ from □, processes it to produce an effect □□, then applies □□ to Σ□−1 to produce a state Σ□ 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.
要从状态 Σ□−1中执行该区块， VM 会从该区块 □ 中获取一个事务 □□ ，将其处理为产生一个效果 □□，然后将该效果应用到 Σ□−1 中，产生一个状态 Σ□ 作为该区块中下一个事务的初始状态，整个区块的效果是该区块中每个事务的效果的有序组成。

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].
每个事务都根据一个工作流程进行处理，该工作流程包括验证事务中的字节码和检查事务发送方的签名等步骤。执行单个事务的整个工作流程在[2]中得到了更详细的解释。

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 □. 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.
今天，块中的交易由 VM 按顺序执行，但 Move 语言被设计成支持并行执行。 原则上，执行交易可以产生一组读以及一组写效果□。块中的交易是并行执行，只有当它的读/写与块中的另一个交易发生冲突时，才会重新执行。由于 Move 的树内存模型允许我们使用其访问路径唯一地识别全局内存单元，因此对冲突的要求非常简单。我们会在未来探索投机性执行方案，如果虚拟机性能成为 Libra Blockchain 的瓶颈。

7. What's Next for Move

So far, we have designed and implemented the following components of Move:
到目前为止，我们已经设计和实施了以下Move的组成部分:

- A programming model suitable for blockchain execution.
- 适合区块链执行的编程模型。

- A bytecode language that fits this programmable model.
- 适合这个可编程模型的字节码语言。

- A module system for implementing libraries with both strong data abstraction and access control.
- 一种实现具有强大数据抽象和访问控制的库的模块系统。

- A virtual machine consisting of a serializer/deserializer, bytecode verifier, and bytecode interpreter.
- 由序列化器/反序列化器、字节码验证器和字节码介面器组成的虚拟机。

Despite this progress, there is a long road ahead. We conclude by discussing some immediate next steps and longer-term plans for Move.
尽管取得了这些进展，但仍有很长的路要走。最后，我们讨论了一些近期的下一步行动和较长远的计划。

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.
实现核心的 Libra Blockchain 功能我们将使用 Move 来实现 Libra Blockchain 中的核心功能：账户、Libra 币、 Libra 备用金管理、验证器节点增减、收取和分配交易手续费、冷钱包等，这项工作已经在进行中。

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].
新的语言特性。我们将向Move语言添加参数多态性(泛型)、集合和事件。参数多态性不会破坏移动现有的安全性和可验证性保证。我们的设计以类似[25]的方式向过程和结构添加了具有种类(即资源或无限制)约束的类型参数。

In addition, we will develop a trusted mechanism for versioning and updating Move modules, trans- action scripts, and published resources.
此外，我们将开发一个可信的机制，用于版本和更新移动模块、转载脚本和已发布的资源。

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.
改进的开发人员体验。Move IR 是作为移动字节码验证器和虚拟机的测试工具开发的。为了执行这些组件， IR 编译器必须故意产生不良字节码，这些字节码验证器将拒绝(例如)。这意味着尽管 IR 适合原型Move程序，但它不是特别友好的用户。为了使Move对第三方开发更有吸引力，我们将改进 IR 并致力于开发符合人体工程学的Move源语言。

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.
正式规范和验证。我们将创建一个逻辑规范语言和自动匹配的正式验证工具，利用Move的验证友好设计(见第3.4节)。验证工具链将检查特定程序的功能正确性属性，这些属性超出了移动字节码验证器强制执行的安全保障(第5.2节)。我们的最初重点是指定和验证实现 Libra Blockchain 核心功能的模块。

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.
我们的长期目标是促进一种正确的文化，在这种文化中，用户将期待模块的正式规范来理解它的功能。理想情况下，没有Move程序员愿意与模块交互，除非它有一个全面的正式规范，并且已经被验证符合该规范。然而，实现这一目标将带来一些技术和社会挑战。验证工具应该是精确和直观的。规范必须是模块化和可重用的，但有足够的可读性，可以作为模块行为的有用文档。

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.
支持第三方移动模块。我们将开发第三方模块发布的路径。为 Libra 用户和第三方开发者创造良好的体验是一个重大挑战。首先，打开通用应用的大门必须不影响系统对核心支付场景和相关金融应用的可用性。其次，我们希望避免欺诈、投机和童车软件存在的声誉风险。在鼓励高软件质量的同时构建开放系统是一个难题。为高保证模块创建市场和提供验证移动代码的有效工具等步骤将有所帮助。

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 ::=	$a \mid b \mid u \mid \vec{b}$
UnrestrictedValue =	Struct $\cup$ PrimitiveValue

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

Global State

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

Modules and Transaction Scripts

$\text{Module} =$	$\text{ModuleName} \times (\text{StructName} \rightharpoonup \text{StructDecl}) \times (\text{ProcedureName} \rightharpoonup \text{ProcedureDecl})$
$\text{TransactionScript} =$	ProcedureDecl
$\text{StructDecl} =$	$\text{Kind} \times (\text{FieldName} \rightharpoonup \text{NonReferenceType})$
$\text{ProcedureSig} =$	$\text{Visibility} \times (\text{VarName} \rightharpoonup \text{Type}) \times \text{Type}^*$
$\text{ProcedureDecl} =$	$\text{ProcedureSig} \times (\text{VarName} \rightharpoonup \text{Type}) \times [\text{Instr}_\ell]_{\ell=0}^{\ell=i}$
$\text{Visibility} ::=$	$\texttt{public} \mid \texttt{internal}$
$\ell \in \text{InstrIndex} =$	UnsignedInt64

Interpreter State

$\sigma \in \text{InterpreterState} =$	$\text{ValueStack} \times \text{CallStack} \times \text{GlobalRefCount} \times \text{GasUnits}$
$vstk \in \text{ValueStack} ::=$	$[] \mid r :: vstk$
$cstk \in \text{CallStack} ::=$	$[] \mid c :: cstk$
$c \in \text{CallStackFrame} =$	$\text{Locals} \times \text{ProcedureID} \times \text{InstrIndex}$
$\text{Locals} =$	$\text{VarName} \rightharpoonup \text{RuntimeValue}$
$p \in \text{ProcedureID} =$	$\text{ModuleID} \times \text{ProcedureSig}$
$\text{GlobalRefCount} =$	$\text{GlobalResourceKey} \rightharpoonup \text{UnsignedInt64}$
$\text{GasUnits} =$	UnsignedInt64

Evaluation

$T \in \text{Transaction} =$	$\text{TransactionScript} \times \text{PrimitiveValue}^* \times \text{Module}^*$

	$\times \mathsf{AccountAddress} \times \mathsf{GasUnits} \ldots$
$B \in \mathsf{Block} =$	$\mathsf{Transaction}^* \times \ldots$
$E \in \mathsf{TransactionEffect} =$	$\mathsf{AccountAddress} \rightharpoonup \mathsf{Account}$
$\mathrm{apply} \in (\mathsf{GlobalState} \times \mathsf{TransactionEffect}) \rightarrow \mathsf{GlobalState}$	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	
$C \in \mathsf{Code} =$	$\mathsf{TransactionScript} \cup \mathsf{Module}$
$z \in \mathsf{VerificationResult} ::=$	`ok` | `stack_err` | `type_err` | `reference_err` | ...
$\boxed{C \rightsquigarrow z}$	bytecode verification
$D \in \mathsf{Dependencies} =$	$\mathsf{StructType}^* \times \mathsf{ProcedureID}^*$
$\mathrm{deps} \in \mathsf{Code} \rightarrow \mathsf{Dependencies}$	computing dependencies
$l \in \mathsf{LinkingResult} ::=$	`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"

`Exists<` R `>`	Pop address a, push bool encoding "a resource of type R 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	