AlgoMove: Typed Abstractions for Algorand Smart Contracts

Alvise Spanò[a], Lorenzo Benetollo[a,b], Michele Bugliesi[a], Silvia Crafa[c], Dalila Ressi[a], Sabina Rossi[a]

[a] *Università Ca' Foscari Venezia, Venezia, Italy*
[b] *Università degli Studi di Camerino, Macerata, Italy*
[c] *Università degli Studi di Padova, Padova, Italy*

Abstract

As Distributed Ledger Technology and smart contracts continue to grow in popularity, there is increasing interest in developing more expressive programming abstractions for digital asset management, along with verification tools that ensure safety and correctness before deployment on blockchain platforms. Addressing this challenge, we introduce AlgoMove, a framework designed to improve smart contract development on the Algorand blockchain. While Algorand is widely recognized for its high performance, scalability, and secure consensus protocol, it still lacks high-level programming abstractions and strong language-based verification mechanisms. AlgoMove brings the Move language, renowned for its robust support for secure digital asset management, to the Algorand platform, adapting its abstractions to the underlying execution model. The result is a high-level, resource-oriented programming model that preserves the core principles of Move while adapting them to Algorand's unique environment.

We present a formal specification of AlgoMove and its encoding into TEAL, Algorand's native assembly-level language, along with a proof of the soundness of this encoding. To demonstrate the practical value and expressive power of the framework, we provide a prototype implementation consisting of a Move-to-TEAL compilation system and an accompanying library for writing smart contracts. Beyond enhancing the Algorand smart contract ecosystem, AlgoMove is significant in its own right as part of a broader effort to bring advances in programming language theory and formal verification into the blockchain space. By balancing expressiveness, ease of use, and strong compile-time guarantees, we seek to meet the distinctive requirements of secure and reliable blockchain applications.

1. Introduction

Blockchain platforms have undergone a remarkable evolution over the past decade. Originally conceived as infrastructures for decentralized payments and money transfers, modern blockchains have emerged as versatile frameworks for asset management and access control, primarily through the use of smart contracts. In contrast, blockchain programming languages have only partially followed the same trajectory of progress. This discrepancy is largely due to the fact that language design has often been driven by the constraints of underlying blockchain architectures rather than by application requirements or foundational programming principles. As a result, with a few notable exceptions (cf. the related work section below), existing languages offer limited support for smart contract development. Developers are typically forced to operate at a low level of abstraction, relying on platform-specific idioms and primitives while depending on the underlying layer-1 middleware to enforce safety properties and security guarantees. In addition to raising substantial entry barriers for developers, this situation leads to the undesirable consequence that smart contracts are often deployed on-chain without undergoing prior certified verification. This poses a significant risk even in blockchains that offer middleware-level protections against faulty contracts, since rolling back or disabling a defective contract post-deployment is typically a costly and cumbersome process.

Overcoming these limitations requires a paradigm shift, bringing programming language theory and formal verification techniques into closer alignment with blockchain development. In this regard, the Move programming language [1], originally developed by the (subsequently dismissed) Libra/Diem project, represents perhaps the most significant advancement. Motivated by the idea that "providing first-class abstractions for key asset management concepts would significantly improve both the safety of smart contracts and the productivity of smart contract programmers" [2], Move is designed as a cross-platform, embedded language featuring a powerful module system (with static dispatch) to support expressive access control policies. It introduces first-class abstractions for digital assets, based on

a linear/affine type system that provides static guarantees against duplication, leakage, or unintended disposal—thus preventing common forms of asset mismanagement. Blockchain-specific features such as accounts, transactions, timestamps, and cryptographic operations are delegated to the host platform, which integrates Move and exposes those capabilities via an API. This modular architecture allows developers to focus on core logic and benefit from Move's strong type system and verification tools. The approach has proven successful, as demonstrated by the number of Move-based blockchains now under active development, including Aptos [3] and Sui [4].

While Move has set a new standard in smart contract safety and expressiveness, its adoption has been largely confined to blockchain platforms explicitly designed around it. A key challenge, therefore, lies in retrofitting these benefits into existing blockchain ecosystems, replicating the strengths of Move without requiring a full redesign of the underlying platform. Motivated by that vision, the present work focuses on the Algorand blockchain. Widely recognized as one of the most advanced blockchain platforms, offering high performance, scalability and a secure consensus mechanism, the Algorand platform still appears to lack comprehensive programming abstractions for smart contract design and effective language-based security tools. The recent release of the AlgoKit toolset [5] has partially addressed the problem of providing developers with higher-level idioms to invoke transactions and structure their smart contracts. At the same time, however, the loosely/dynamically typed nature of Python and the mostly transparent structure of the Python layer over the underlying, transaction-based middleware leave designers with little support for statically enforcing even the most basic resource management invariants and access control policies.

To address these limitations, we introduce AlgoMove, a porting of the Move programming language tailored for the Algorand ecosystem that brings the resource-centric programming model of Move to Algorand, enabling developers to write secure, high-level smart contracts with strong static guarantees. In particular, it introduces a built-in `Asset` datatype, governed by Move's linear type system, that provides a language-level abstraction mirroring the native asset infrastructure of Algorand. Move's access control and type safety mechanisms enable the enforcement of correctness properties at compile time, significantly reducing the risk of resource-related vulnerabilities. We discuss the resulting programming model and its benefits over existing Algorand toolkits, emphasizing the added safety and expressiveness gained through integration with a type-theoretic framework.

Our experiment appears valuable both as a new high-level language made available to Algorand developers and as a novel implementation of Move on one of the most prominent blockchain platforms to date. From a broader perspective, AlgoMove represents a concrete effort to bring the advances of programming language theory and formal verification into the realm of blockchain development. It aims to address the distinctive requirements of smart contract applications while striking a balance between expressiveness, ease of design, and strong compile-time guarantees for safety and security. The design principles underlying AlgoMove scale to different platforms, making the embedding for Algorand proposed in this paper an instance of a more general paradigm for porting Move to other account-based platforms.

Contributions

The present paper is a revised and substantially extended version of the preliminary work presented in [6]. We provide a formal account of the soundness of AlgoMove by defining a semantic-preserving encoding into TEAL and proving the correspondence between the state transitions of the AlgoMove interpreter and the state transitions of the TEAL counterpart. As a corollary of the operational correspondence proof, we derive a *resource preservation* result, which formally characterizes the way in which AlgoMove assets serve as language-level abstractions of Algorand's layer-1 assets, preserving their structure and behavior across the translation.

We also developed a prototype implementation of AlgoMove, realized as a tiered framework consisting of two main components: a library and a translation tool. In order to run on Algorand, Move programs must import and use the AlgoMove library, and then they must be compiled into TEAL using the translation tool.he library, which is written in Move, defines the `Asset` datatype alongside a set of additional types and utility functions specifically tailored for integration with the Algorand platform. Rather than developing a full compiler from scratch, we build on the existing Move toolchain: the Move source code is first compiled to bytecode using the Aptos compiler, which provides a reliable and actively maintained infrastructure. Our translation tool then parses the disassembled bytecode and emits equivalent TEAL code. The resulting TEAL output is compatible with the Algorand toolchain and can

be deployed directly on the blockchain. The prototype of the AlgoMove framework is available at `github.com/alvisespano/AlgoMove`.

In summary, our contributions are as follows:

- a porting of Move to Algorand, that is a framework that enables the development of Algorand smart contracts written in Move, a strongly typed and structured language with well-established safety guarantees;
- a formalization of the semantics of the TEAL language that enables a formal correctness treatment of the translation from Move to TEAL, including a proof of operational correspondence and, as a direct consequence, resource preservation;
- an advanced prototype implementation of both the library and the translation tool, supporting a substantial fragment of the Move language, including struct types, parametric polymorphism, global storage primitives, borrows and references.

Related Work

Move implementations. Several blockchain platforms have adopted Move as their primary programming language, including 0L [7] and Starcoin [8], besides the already mentioned Aptos and Sui. As previously noted, our purpose is orthogonal to these initiatives: whereas those platforms are designed as integrated Move-based blockchains built from the ground up, AlgoMove targets an existing and fully developed blockchain infrastructure. As a consequence, although AlgoMove shares certain features with these platforms—most notably the presence of an API layer for interacting with the underlying blockchain—the technical development it entails is fundamentally distinct and, in many respects, more complex. This is due to the non-trivial effort required to bridge the design discrepancies between the Move and Algorand Virtual Machines. Solana Labs [9] also initiated an experimental effort to port Move to LLVM, with the goal of executing Move modules on Solana's virtual machine via an LLVM backend [10]. However, the project's GitHub repository was later archived, indicating that the initiative was discontinued, and there is no evidence that it reached a functional state [11]. According to Solana's official documentation, a new in-house approach to Move integration is currently under development, though no further technical details have been disclosed [12].

Blockchain programming languages. The substantial body of existing research and development on blockchain programming languages is only loosely related to the objectives of our present work. Solidity [13], the first and still most widely used smart contract language, was explicitly designed for the Ethereum blockchain. However, the frequency and severity of vulnerabilities encountered since its introduction (such as the infamous reentrancy exploit and numerous other attacks [14]) have exposed critical weaknesses in its design and motivated the search for safer alternatives [15]. Among the proposed enhancements is *ASAC* [16], which compiles authorization policies written in ALFA (Abbreviated Language for Authorization) into Solidity contracts, thereby enforcing fine-grained access control. Another recent contribution, *Solmover* [17], leverages large language models (LLMs) to translate existing Solidity codebases into equivalent Move programs, offering improved safety and verifiability.

Rust [18], a general-purpose language released in 2011, is widely considered a safer alternative due to its ownership model and rich type system, which enable strong guarantees around memory safety and resource management. Though originally unrelated to blockchain, Rust was later adopted by Solana as the main language for writing smart contracts. Move, in turn, builds on Rust's foundation, inheriting many of its benefits while offering a simpler programming model and enhanced safety guarantees through static verification and a more disciplined resource-handling model. As discussed in [19], Move provides significantly stronger support for composability and formal reasoning than Rust.

In the context of Algorand, Algorand Python [20] is a recent addition to the development toolchain. Designed as an evolution of PyTeal [21], it allows developers to express contracts in Python syntax while generating TEAL bytecode under the hood. While it improves developer ergonomics, it ultimately retains the low-level, assembly-like abstraction of TEAL and inherits its limitations in terms of safety, expressiveness, and data manipulation. Other high-level languages for Algorand smart contracts include TEALish, PyTeal, and Reach. TEALish [22] is a domain-specific procedural language for writing Algorand contracts more naturally. Although it has been used to write large production contracts, it remains experimental and is not yet regarded as stable for general-purpose development. PyTeal provides

a convenient syntax and static checks to generate TEAL using Python scripts. It continues to receive regular updates, making it the de facto standard for a native Python experience for Algorand smart contract development. Reach [23] is a cross-chain domain-specific language for writing smart contracts, including Algorand. However, the core Reach-lang compiler is no longer maintained, suggesting that active development and uptake in the Algorand ecosystem have slowed.

Paper Plan

In Section 2 we introduce the background material on Move and Algorand required to make the paper self-contained. Section 3 gives an overview of the AlgoMove programming model and shows its effectiveness for smart contract development on Algorand. Sections 4 and 5 define the operational semantics of AlgoMove and TEAL, respectively. Section 6 details the encoding of AlgoMove into TEAL and proves the main properties of operational correspondence and resource preservation. Section 7 illustrates the structure of our prototype implementation. Finally, Section 9 reports closing remarks and a brief discussion of our plans for future work. In the interest of clarity and improved readability, most of the technical details are collected in a series of separate Appendices.

2. Background

As anticipated, AlgoMove is a porting of the Move programming language tailored for the Algorand blockchain . It brings Move's high-level, resource-oriented programming model to the Algorand ecosystem, enabling developers to write expressive and verifiably safe smart contracts while interfacing directly with Algorand's low-level infrastructure. To provide context, we provide a background on the key features of Move and the Algorand blockchain.

Resource-based programming in Move

The Move programming model is centered around a few simple principles. Smart contracts are modules consisting of C-like `struct` definitions and functions: the former serve as the fundamental building blocks for representing data, while the latter provide the interface through which module clients can create, access, and/or modify the module's data structures. Scope rules ensure that module functions are the only entry point to the module's data types. Moreover, these functions are always executed on behalf of an authenticated blockchain account, represented in Move as the account's `signer`. The combination of these *access control* mechanisms forms one of the two fundamental components of Move's robust support for resource management. The second component is the language's static typing system, based on *linear types*, which enforces *scarcity*: once created (by a privileged module function), the type system treats module `struct`s as first-class resources that cannot be copied or implicitly discarded, only moved between program storage locations or passed around between functions.

The linear restrictions imposed by the type system are lifted for `struct`s endowed with specific *abilities*, which qualify them as *values* that can be duplicated (via the `copy` ability) and/or discarded (via the `drop` ability), i.e., not necessarily transferred, and thus effectively destroyed as execution proceeds through the program. Two additional abilities, `key` and `store`, enable `struct`s to be stored persistently on the underlying blockchain and, respectively, to be nested within other `struct`s held in persistent storage. The Move standard library provides a `Coin` resource, i.e. a non-copiable and non-droppable datatype, representing a certain amount of fungible assets. The `Coin` type is parametric over an asset or currency type, allowing programmers to specify valuable entities being manipulated in a strongly typed manner. A simple asset transfer looks like:

```
use std::coin;

struct Dollar {}  // a dummy type representing a given currency

public fun transferDollars(from: &signer, to: address, amt: u64) {
    let coins: Coin<Dollar> = coin::withdraw<Dollar>(from, amt);
    coin::deposit(to, coins);
}
```

Different Move implementations provide different, blockchain-specific representations of persistent storage. For our purposes, we assume the original Diem-Move (sometimes referred to as *Core Move*) representation, which is

defined in terms of the *global storage* abstraction, a table of *(acct, struct)* pairs, keyed by *acct*, the addresses of the accounts holding the resources encoded by *struct*. A resource can be stored on the global storage only by the account's signer, through the invocation of the `move_to` primitive. This is a *write* operation that also releases the ownership of the linear resource at the type level. Resources stored in the global storage can be accessed by requesting (i.e., `borrow`-ing) a reference to the resource using the address of the account under which it is stored. Finally, the `move_from` primitive acts as the inverse operation of the `move_to`, retrieving a resource from the global storage. This is a *read-and-delete* operation that transfers the linear ownership to the caller's scope. These primitives are the distinctive features of the Move language.

The Algorand blockchain

Algorand is a decentralized platform designed to support scalable, secure, and efficient transactions. It relies on a unique Pure Proof-of-Stake (PPoS) consensus mechanism [24], which ensures rapid transaction finality and mitigates the risk of forks.

A key component of Algorand is its Transaction Execution Approval Language (TEAL), an assembly-level programming language tailored for smart contract execution on the Algorand Virtual Machine (AVM). Initially conceived as simple TEAL scripts encoding authorization policies for micro-payments and other transfer protocols involving Algos (Algorand's native currency), Algorand Smart Contracts now support fully general digital-asset management applications. The Algorand programming model centers on two components: TEAL or PyTEAL [21] (more recently, Algorand Python [20]) scripts that encode the application logic, and an API that provides access to the underlying blockchain transactions and Algorand Standard Assets (ASAs). Unlike Move resources, which are first-class values, Algorand assets are layer-1 entities that can only be manipulated through platform-provided transactions. These transactions are made accessible for invocation within scripts via the API.

The Algorand storage model comprises two areas of persistent state. The first, the *local state*, is structured around accounts, much like in Move. However, the key difference lies in how access to an account's state is managed: in Move, access is granted to contract functions signed by the account owner, whereas in Algorand, the account itself grants access to the contract through an opt-in transaction[1]. The second area, called the *global state*, is associated with smart contracts to make them stateful, similar to contracts in Solidity, and has no direct counterpart in Move. To avoid confusion among the different notions of state, we adopt the following terminology: *global/local state* when referring to Algorand; *global storage* when referring to Move.

TEAL and Algorand smart contracts

Algorand supports smart contracts through a stack-based virtual machine and a low-level language called *Transaction Execution Approval Language* (TEAL). TEAL is designed to be simple, deterministic, and verifiable on-chain, providing the necessary primitives to manipulate accounts, assets, and application state while ensuring high security and predictable gas costs. TEAL programs operate on a stack-based architecture, executing instructions sequentially and manipulating values directly on the stack. Values can be integers or byte strings; no other data type is supported. Control flow is explicit, with assembly-like branch instructions such as `bnz` (branch if nonzero) and `bz` (branch if zero), while arithmetic and logical operations are performed using standard stack operators like `+`, `-`, `and`, `or`, and `xor`. A minimal TEAL example for checking an account balance looks like this:

```
txn Sender
asset_holding_get AssetBalance
pushint 1000
>=
bnz success
err
success:
```

The program fetches the asset balance of the transaction sender, compares it to 1000 units, and branches to the `success` label if the balance is sufficient; otherwise, it fails. Another common use case in TEAL is transferring currency using inner transactions:

[1] Though Move does not natively support opt-in, an opt-in programming pattern is required for effective execution of complex asset exchanges. This limitation motivated the design of the Sui Move platform [2] as an alternative to the Aptos architecture.

```
itxn_begin
pushbytes "pay"
itxn_field TypeEnum
pushbytes 0x12345678
itxn_field Receiver
pushint 1000000
itxn_field Amount
itxn_submit
```

This snippet represents a simple transfer of 1 algo from one account to another, revealing how transactions appear at the programmatic level. An inner transaction has to be initiated using the `itxn_begin` instruction, then the relevant fields must be populated using `itxn_field`: the type of transaction (pay), the receiver address, the amount of currency (1 million microalgos); finally, the inner transaction can be executed using the `itxn_submit` opcode.

While TEAL is powerful and expressive at the low level, it treats assets and accounts as second-class entities, requiring explicit instructions for creation, inspection, and transfer. This motivates the design of higher-level abstractions such as AlgoMove, which lift assets to first-class, typed values and provide a structured, safe interface for their manipulation, while still compiling down to efficient TEAL code.

As a final remark, it is important to clarify the terminology adopted by Algorand. In this context, the notion of a *transaction* extends well beyond monetary transfers or asset management. Transactions in Algorand closely resemble system calls in traditional operating systems: any interaction with the underlying platform is expressed as a transaction. This includes, for instance, retrieving the caller address, accessing the arguments passed to a contract, or querying the assets available to an application. To perform any of these operations, a transaction must be constructed and executed. Transactions may be issued off-chain, while *inner transactions* are generated and executed programmatically by TEAL code during contract execution.

3. Overview of AlgoMove

AlgoMove is a framework for compiling Move smart contracts to the Algorand platform. Move programs must import and use the AlgoMove library, which reimplements the primitives and abstractions of the standard Move library, preserving the conventional Move programming model while adapting it to the characteristics of the target platform. Since Algorand features an execution architecture that significantly differs from existing Move-based systems, mapping Move constructs onto it requires careful design and platform-specific handling.

- Accesses to the Move *global storage* must be converted into accesses to something equivalent in Algorand. The best candidate is what Algorand calls *local storage*, or *local state*, which is a memory space accessible to an account and to the application (smart contract) it interacts with. In AlgoMove, datatypes tagged with the `key` ability are stored in the Algorand *local state*.
- Algorand does not support user-defined datatypes; struct types in Move must therefore be marshaled and converted into strings of bytes according to some serialization format.
- While in most Move embeddings balance management is implemented in Move itself using data structures located on the global storage, in Algorand assets are not first-class entities, and asset manipulation is performed by special instructions called *transactions*. Our system addresses this design discrepancy by providing a custom library on the Move side, where operations such as asset transfer trigger the required transactions on the Algorand side. To mimic the original Move flavour, we provide an `Asset` linear datatype replacing the classic `Coin` type and serving the same purpose, while adhering to the Algorand terminology.
- Certain Algorand features (e.g., hashing primitives, accessing memory boxes and transactions performing platform-specific operations) require special-purpose TEAL instructions. However, the AlgoMove translation tool would never emit such instructions, as they do not have a Move counterpart and simply do not appear in the source Move program. As a bonus feature, to allow AlgoMove programmers to take full advantage of Algorand-specific features, our library provides a low-level API that allows the invocation of all special TEAL instructions from the Move code.

A sample smart contract

We illustrate AlgoMove in action with a smart contract[2] that implements a simple auction (cf. Figure 1 below). As in standard Move, the smart contract is implemented as a module. Specifically, the `auction` module defines two datatypes, `Auction` and `Bid`, and three entry functions: one to start the auction, one to place bids, and one to close the auction. The datatype definitions all adhere to the typing discipline distinctive of Move. `Auction` represents the auction state, holding the item being auctioned (a kind of on-chain NFT) along with other administrative information. The current highest bid amount is stored separately using the `Bid` type, which acts as an envelope holding the bid in the form of assets. In both the `Auction` and `Bid` types, the `key` ability allows values of these types to be stored on-chain and ensures that they cannot be copied or dropped. Finally, the `Asset` type models a notion of currency as a collection of fungible tokens, mimicking Move's `Coin` type. All functions and datatypes make extensive use of parametric polymorphism (a.k.a. generics), as suggested by the Move programming guidelines, allowing the reuse of the same contract with multiple types.

The contract flow proceeds as follows. The auctioneer begins by invoking `start_auction` to initialize the auction state and store the base bid under the auctioneer's account. Participants then join by invoking the `bid` function with their bid amount, passing in the auctioneer's address to obtain a reference to the auction state and update it as needed. The top bid is updated by first retrieving the previous bid from the blockchain (via `move_from`) and then storing the new bid under the new bidder's account using `move_to`. The participant whose bid is outbid is reimbursed through an invocation of `deposit`. Updates occur only if the new bid exceeds the current top bid, as enforced by an assert: if the assertion fails, the transaction executing the function fails as well, and all its side effects are rolled back. Finally, the auctioneer may close the auction by invoking `finalize_auction`, and the winner may retrieve the auctioned item by calling `retrieve_item`.

Why static typing matters

The example in Figure 1 clearly illustrates AlgoMove's distinctive high-level model of programmable resources, as well as the added value provided by the framework's support for static typing.

Unlike currently available development kits for Algorand, which are based on Python, AlgoMove provides assets as first-class datatypes, treated as resources subject to the static typing discipline inherited from Move. This enables static, type-based enforcement for rich access-control policies, capabilities that are beyond the reach of any Python-based extension of TEAL and must otherwise be handled by Algorand's native layer-1 mechanisms. These mechanisms remain available in AlgoMove, as the primitives for manipulating the programmable assets (e.g., the `deposit` function) are compiled into the corresponding Algorand transactions that operate on the assets' layer-1 representations. However, Move's typing discipline provides static protection against misuse of such assets, ensuring that the transactions implementing AlgoMove primitives satisfy the safety properties enforced by the blockchain runtime. In principle, Algorand layer-1 security mechanisms could be omitted for any deployed AlgoMove contract, but doing so would compromise the safe integration of the contract with the broader Algorand/TEAL codebase.

A further advantage of Move's typing system arises from its support for parametric polymorphism (a.k.a. generics). In this regard, all datatype and function definitions in the auction module are generic over various type parameters. This design allows the module to support the concurrent execution of multiple auctions for different items and currencies without additional programming effort to maintain consistency across auctions. This capability directly results from the use of type parameters, which ensure that each new bid is accepted and associated with the intended auction, provided it instantiates the `AssetType` and `ItemType` parameters consistently with the instantiation chosen by the auctioneer who opened the bids. While implementing the same logic in an untyped language is possible, it would require cumbersome runtime housekeeping that Move's static typing system handles automatically.

The `Auction` contract introduced in Figure 1 shows Move's linear types in action and provides a practical example of their benefits when manipulating assets. Each bid is represented as a linear `Bid` resource, and the auction item itself

[2] The implementation builds on an extension of the language formalized in Section 4, incorporating a library that provides an API to the services of the Algorand Virtual Machine, as described in Section 7. The resulting code largely retains the look and feel of Aptos Move, including distinctive primitives such as `move_to`, `move_from`, and `borrow_global` [25, 26].

```
module auction
use algomove::asset;
use algomove::utils;

struct Auction<ItemType> has key {
    item: ItemType,
    auctioneer: address,
    top_bidder: address,
    expired: bool
}

struct Bid<AssetType> has key {
    assets: Asset<AssetType>
}

public fun start_auction<AssetType, ItemType>(acc: &signer, base: Asset<AssetType>, item: ItemType) {
    let auctioneer = utils::address_of_signer(acc);
    let auction = Auction<ItemType> { item, auctioneer, top_bidder: auctioneer, expired: false };
    move_to(acc, auction);
    move_to(acc, Bid { assets: base });
}

public fun bid<AssetType, ItemType>(acc: &signer, auctioneer: address, assets: Asset<AssetType>){
    let auction = borrow_global_mut<Auction<ItemType>>(auctioneer);
    let Bid { assets: top_bid } = move_from<Bid<AssetType>>(auction.top_bidder);
    assert!(!auction.expired && asset::value(&assets) > asset::value(&top_bid));
    asset::deposit(auction.top_bidder, top_bid);
    auction.top_bidder = utils::address_of_signer(acc);
    move_to(acc, Bid { assets });
}

public fun finalize_auction<AssetType, ItemType>(acc: &signer) {
    let auctioneer = utils::address_of_signer(acc);
    let auction = borrow_global_mut<Auction<ItemType>>(auctioneer);
    assert!(auctioneer == auction.auctioneer);
    auction.expired = true;
    let Bid { assets: top_bid } = move_from<Bid<AssetType>>(auction.top_bidder);
    asset::deposit(auctioneer, top_bid);
}

public fun retrieve_item<AssetType, ItemType>(acc: &signer, auctioneer: address): ItemType {
    let Auction { item, auctioneer: addr,
                  top_bidder, expired } = move_from<Auction<ItemType>>(auctioneer);
    assert!(expired && auctioneer == addr);
    let self = utils::address_of_signer(acc);
    assert!(self == top_bidder);
    item
```

Figure 1: An auction contract written in ALGOMOVE. Like existing Move embeddings (e.g., Aptos, Sui and others), ALGOMOVE provides the standard primitives for payment and for accessing the account state. While a few function and type names may differ to align with Algorand terminology (e.g., `Asset` instead of `Coin`), their underlying logic remains the same. The `utils` module imported at the top provides a handful of miscellaneous utility functions, such as `address_of_signer`, which extracts the address of a given signer account.

is also a linear resource. Move's type system guarantees at compile time that these resources cannot be duplicated or lost. In a direct PyTEAL implementation on Algorand, the same logic would require manually managing:

- the current top bid and bidder;
- refunding the previous top bidder;
- transferring the auction item to the winner.

Focusing on the bidding alone, a PyTEAL implementation of the `bid` function could be:

```
bid = Seq([
    Assert(App.globalGet(Bytes("expired")) == Int(0)),
    top_bid = App.globalGet(Bytes("top_bid")),
    Assert(Txn.application_args[1] > top_bid),

    # Refund previous top bidder
    InnerTxnBuilder.Begin(),
```

```
    InnerTxnBuilder.SetFields({
        TxnField.type_enum: TxnType.AssetTransfer,
        TxnField.asset_receiver: App.globalGet(Bytes("top_bidder")),
        TxnField.asset_amount: top_bid,
        TxnField.xfer_asset: App.globalGet(Bytes("asset_id"))
    }),
    InnerTxnBuilder.Submit(),

    # Update top bidder and bid amount
    App.globalPut(Bytes("top_bid"), Txn.application_args[1]),
    App.globalPut(Bytes("top_bidder"), Txn.sender()),

    Return(Int(1))
])
```

If any branch fails to execute the refund or update steps correctly, tokens may become trapped or lost. There is no static guarantee that every bid is properly conserved. In contrast, the Move implementation of the `bid` function ensures that:

- every `Bid` is either moved, deposited, or explicitly consumed;
- the auction item is never duplicated or lost;
- the compiler rejects any code path where a resource is ignored or dropped.

Move statically enforces asset conservation, eliminating a class of defects involving asset manipulation that is present when programming directly in PyTEAL. When compiling Move to TEAL, our compiler preserves linearity guarantees at the Move level and ensures that every resource consumption is translated into exactly one ledger-level asset operation. Consequently, safety derives from the source language type system rather than from post hoc auditing of TEAL code.

Security benefits

The Move language provides strong safety guarantees through its resource-oriented design and static type system, which prevent several classes of vulnerabilities, such as double-spending, asset duplication, or accidental asset loss [26]. Notably, reentrancy, a well-known source of exploits in other smart contract platforms [27, 28], is impossible by design in Move, as modules cannot mutually invoke each other's internal functions [29]. Despite these protections, Move contracts are not immune to all vulnerabilities. Analyses of deployed Move code show that most issues arise from higher-level programming errors, such as incorrect capability usage, violations of invariants, or improper access control [30, 31], rather than from low-level execution flaws.

In contrast, Algorand smart contracts are exposed to a broader spectrum of vulnerabilities due to their low-level, stack-based execution model and the absence of a resource-aware type system [32]. Sun et al. [33] identify several classes of contract-level vulnerabilities and report a substantial number of affected applications deployed on-chain. Most of the vulnerabilities discussed therein stem from issues that do not naturally arise within the programming model enforced by Move, and would therefore not be directly addressed by adopting AlgoMove. Nevertheless, a subset of vulnerabilities known in the literature could in principle be prevented at compile time, whereas in Algorand they are typically detected only at runtime.

Table 1 summarizes these vulnerability classes according to the SWC[3] taxonomy [34] and the corresponding CWE entries [35]. The table illustrates how Move prevents or mitigates such issues at compile time, either through its type system or through specific language design choices, briefly describing what underlying mechanism participates in the mitigation. The table highlights the benefits of porting Move's resource-oriented programming model to the Algorand blockchain: although it does not eliminate all possible risks, it effectively prevents a significant class of defects, thereby improving the overall safety of smart contract development.

[3]The SWC taxonomy was originally introduced for the Solidity language and the Ethereum blockchain. In the absence of a dedicated taxonomy for Algorand or Move, it is adopted here as a reference.

SWC	CWE	Vulnerability Description	Mitigation	Underlying Mechanism
SWC-104	CWE-252	Unchecked Return Value	✿	static typing and bytecode verifier
SWC-105	CWE-284	Unprotected Withdrawal / Improper Access Control	✰	modules enforce controlled access
SWC-107	CWE-841	Reentrancy / Improper Enforcement of Behavioral Workflow	✰	mutual inter-module calls unsupported
SWC-132	–	Unexpected Balance	✿	linear types prevent balance loss
–	CWE-664	Improper Control of a Resource Through its Lifetime	✿	linear types and borrow checker
–	CWE-843	Type Confusion	✿	strong typing and no cast support

Table 1: Vulnerability classes that Algorand handles at runtime and that Move, thus AlgoMove, mitigates at compile time. When applicable, the corresponding entries in the SWC and CWE taxonomies are reported. Column *Mitigation* indicates whether the mitigation in Move arises from its type system (✿) or from language design choices (✰).

4. AlgoMove Language Semantics

As in existing Move presentations, we formalize the semantics of AlgoMove directly in terms of the state transition system of its bytecode instructions. Our formalization adapts the notation from [36, 26] to fit our specific choices and extends it to accommodate AlgoMove-specific features such as asset management operations. Throughout this and the following sections, we refer to source and bytecode instructions interchangeably, simply as Move instructions.

The bytecode is assumed to have been either generated by the Move compiler or verified by the bytecode verifier, and is therefore type-safe. This implies, for example, that no `ReadRef` instruction appears on a non-reference value, no `Unpack` instruction is applied to a non-record value, and `Call` instructions always receive the expected arguments on the stack. Regarding the distinctive AlgoMove primitives that deal with assets, these are presented here as special bytecode instructions (**Create**, **Withdraw**, **Deposit**). The standard Move bytecode does not include such opcodes; in the actual output from a real Move compiler, such as Aptos', these operations instead appear as **Call** instructions invoking functions defined in our library.

Syntax. A program is organized into modules, each containing datatype and function declarations. The instruction set, given below, includes the standard Move instructions—to operate on local variables, references, records, and global storage; to call and return from functions; to branch conditionally and unconditionally; and to manipulate the stack, as well as the AlgoMove 's library instructions for operating with assets.

Local vars	**MoveLoc** $\langle n \rangle$ | **CopyLoc** $\langle n \rangle$ | **StLoc** $\langle n \rangle$ | **BorrowLoc** $\langle n \rangle$
References	**ReadRef** | **WriteRef**
Records	**Pack** $\langle s \rangle$ | **Unpack** $\langle s \rangle$ | **BorrowField** $\langle s.x \rangle$
Global storage	**MoveTo** $\langle s \rangle$ | **MoveFrom** $\langle s \rangle$ | **BorrowGlobal** $\langle s \rangle$ | **Exists** $\langle s \rangle$
Calls and jumps	**Call** $\langle f \rangle$ | **Ret** | **Branch** $\langle \ell \rangle$ | **BrCond** $\langle \ell \rangle$
Stack ops	**Pop** | **LdConst** $\langle v \rangle$ | ***Op***
Assets	**Create** $\langle s \rangle$ | **Withdraw** $\langle s \rangle$ | **Deposit** $\langle s \rangle$

Functions are called by name (f), while instructions are referenced by labels (ℓ). Types are limited to struct typenames (s). Values (v) include identifiers x (variable or record field names), records, and references. Records are sets of (identifier, value) pairs ($\{x_1 : v_1, \dots, x_n : v_n\}$) representing instances of struct datatypes. References consist of a memory location plus a path (`ref` $\langle c, p \rangle$), which encodes a possibly empty sequence of field selections.

Computation states. A program operates on a computation state of the form $\langle M, L, G, S, B \rangle$, which includes a memory area referenced to from the environment of local variables and from the global storage, the call stack, and the asset balance.

Memory	$M : c \mapsto v$	a map from memory cells (or locations) to values
Local Storage	$L : n \mapsto c$	a map from numeric indexes to memory locations
Global Storage	$G : (a, s) \mapsto c$	a map from pairs (address, type name) to memory locations
Call Stack	$S : i :: S$	containing program points ℓ , values v, and locals L.
Asset Balance	$B : (a, s) \mapsto n$	a map from pairs (address, type name) to amount (a non-negative integer)

State Transition Semantics. The semantics is given as a small-step transition system with judgments of the form $\mu \xrightarrow{I} \mu'$, where μ is the input state, I is an instruction and μ' is the output state. The rules follow those in [36, 26], with only minor, primarily notational, modifications. Most rules are standard and thus omitted here (see Appendix A for the complete system). In the remainder of this section, we present the most representative ones: a selection of core Move rules for resource management and the rules for handling Algorand-style assets.

AlgoMove state transitions (Core Move excerpts)

M-MoveLoc
$$\frac{L(n) = c \qquad c \in \texttt{dom}(M)}{\langle M, L, G, S, B\rangle \xrightarrow{\textbf{MoveLoc}\ \langle n\rangle} \langle M \setminus c, L \setminus n, G, M(c) :: S, B\rangle}$$

M-StLoc
$$\frac{c \notin \texttt{dom}(M)}{\langle M, L, G, v :: S, B\rangle \xrightarrow{\textbf{StLoc}\ \langle n\rangle} \langle M[c \mapsto v], L[n \mapsto c], G, S, B\rangle}$$

M-BorrowField
$$\frac{P(s) = \texttt{struct}\ s\,\{\ldots, x : \tau, \ldots\} \qquad M(c)[p] = \{(\ldots, (x, v), \ldots)\,\}}{\langle M, L, G, \texttt{ref}\ \langle c, p\rangle :: S, B\rangle \xrightarrow{\textbf{BorrowField}\ \langle s.x\rangle} \langle M, L, G, \texttt{ref}\ \langle c, p :: x\rangle :: S, B\rangle}$$

M-MoveTo
$$\frac{\langle a, s\rangle \notin \texttt{dom}(G) \qquad c \notin \texttt{dom}(M)}{\langle M, L, G, a :: v :: S, B\rangle \xrightarrow{\textbf{MoveTo}\ \langle s\rangle} \langle M[c \mapsto v], L, G[\langle a, s\rangle \mapsto c], S, B\rangle}$$

M-MoveFrom
$$\frac{G(a, s) = c \qquad M(c) = v}{\langle M, L, G, a :: S, B\rangle \xrightarrow{\textbf{MoveFrom}\ \langle s\rangle} \langle M \setminus c, L, G \setminus \langle a, s\rangle, v :: S, B\rangle}$$

Rule (M-MoveLoc) performs a destructive read of a local variable by removing it from L and placing its value on the stack: this is the so-called *move* semantics that is distinctive of Move. Rule (M-StLoc) allocates a new local variable with the value found on the top of the stack. Rule (M-BorrowField) extends the path component p of the reference on the top of the stack by appending the selected field x. Rule (M-MoveTo) writes a value v into the global storage G, while rule (M-MoveFrom), similarly to (M-MoveLoc), retrieves a value from global storage and removes it.

AlgoMove state transitions (rules for Algorand-style asset management)

M-Deposit
$$\frac{v = \{\, (\mathtt{amount}, m) \,\} \qquad n = B(a,s)}{\langle M, L, G, a :: v :: S, B\rangle \xrightarrow{\textbf{Deposit}\ \langle s\rangle} \langle M, L, G, S, B[\langle a, s\rangle \mapsto n+m]\rangle}$$

M-Withdraw
$$\frac{v = \{\, (\mathtt{amount}, m) \,\} \qquad n = B(a,s) \qquad n \geq m}{\langle M, L, G, a :: m :: S, B\rangle \xrightarrow{\textbf{Withdraw}\ \langle s\rangle} \langle M, L, G, v :: S, B[\langle a, s\rangle \mapsto n-m]\rangle}$$

M-Create
$$\frac{v = \{\, (\mathtt{amount}, m) \,\}}{\langle M, L, G, a :: m :: S, B\rangle \xrightarrow{\textbf{Create}\ \langle s\rangle} \langle M, L, G, v :: S, B[\langle a, s\rangle \mapsto 0]\rangle}$$

Rule (M-Deposit) increases the amount of assets of type s held by address a of the given amount m. Operand v is a record value consisting of a single field `amount` and represents our `Asset` datatype. Although the implementation (cf. Section 7) includes additional fields required by Algorand, in the semantic rules of AlgoMove we abstract away such details and retain only the minimal structure necessary for the formalization. Rule (M-Withdraw) is the counterpart of (M-Deposit); it decreases the asset balance of address a by the amount m being withdrawn. Rule (M-Create) captures the minting of m assets of type s and the corresponding insertion of a new binding into the B map.

Resource Safety. AlgoMove, i.e., Move code using the AlgoMove library, inherits the resource safety guarantees established in [26] for Move almost for free. In fact, an inspection of the state transitions associated with the three asset-management primitives readily shows that they involve no loss or duplication of any resource.

5. TEAL Language Semantics

Syntax. Programs are sequences of labeled and unlabeled instructions J. Values υ are restricted to 64-bit unsigned integers and byte arrays $[b_1, \ldots, b_n]$. No additional datatypes are supported by the platform; for instance, addresses are represented as fixed-size 32-byte arrays. The following excerpt from the instruction set J includes only the instructions relevant to the translation and is not intended as a complete description of TEAL (the full instruction set is provided in Appendix B).

Scratch space	**load** $\langle n\rangle$ | **store** $\langle n\rangle$ | **loads**
Data manipulation	**extract** $\langle n, m\rangle$ | **extract3** | **replace2** $\langle n\rangle$ | **itob** | **btoi** | **concat**
Operand stack	**pop** | **dupn** $\langle n\rangle$ | **swap** | **pushbytes** $\langle n_1 \,..\, n_m\rangle$ | **pushint** $\langle n\rangle$
Call stack	**frame_dig** $\langle n\rangle$ | **frame_bury** $\langle n\rangle$ | **cover** $\langle n\rangle$ | **uncover** $\langle n\rangle$
Calls and jumps	**proto** $\langle n, m\rangle$ | **callsub** $\langle \ell\rangle$ | **retsub** | **b** $\langle \ell\rangle$ | **bz** $\langle \ell\rangle$ | **bnz** $\langle \ell\rangle$
Local state	**app_local_put** | **app_local_get** | **app_local_del**
Transactions	**txn** $\langle x\rangle$ | **itxn_begin** | **itxn_field** $\langle x\rangle$ | **itxn_submit**

Computation states. As for AlgoMove, we formalize the operational semantics of TEAL in terms of a notion of computation state σ, defined as the tuple $\langle C, V, \Delta, \Omega, A, \Phi\rangle$ where

Call Stack	$C : i :: C$	program points ℓ or tuples of values $\langle \upsilon_1, .., \upsilon_n\rangle$
Operand Stack	$V : v :: V$	values υ
Scratch Space	$\Delta : n \mapsto \upsilon$	a map from numeric indexes to values
Local State	$\Omega : (a, k) \mapsto \upsilon$	a map from $(addr, key)$ pairs to values
Asset Balance	$A : (a, n) \mapsto m$	a map from $(addr, asset\text{-}id)$ to the amount held by the address
Current Transaction	$\Phi : x \mapsto \upsilon$	a map from field names into values

The Call and Operand stacks are typically structured as a single entity; however, our presentation remains faithful to the Algorand implementation, which maintains them as separate components. TEAL computation states also include two additional components — *boxes* and *global state* — which we omit from our formalization, as they do not play a role in the AlgoMove encoding introduced in Section 6 and are therefore irrelevant to our development. Briefly, boxes serve as a memory encapsulation mechanism with no counterpart in AlgoMove, while TEAL's global state supports persistent contract state, a feature absent from AlgoMove 's stateless contract model.

State Transition Semantics. The semantics is presented as a small-step transition system with judgments of the form $\sigma \xrightarrow{J} \sigma'$. For the most part, the rules follow standard formulations and are therefore omitted here (cf. Appendix B for the complete system). In the remainder of this section, we present a representative subset of the rules, focusing on the manipulation of the operand stack and the local state (the global state is omitted, as it is not involved in the translation of stateless Move contracts), along with a selection of transaction rules responsible for creating and managing assets at the blockchain's layer-1.

TEAL state transitions for stack and local state (excerpts)

A-Load
$$\frac{\Delta(n) = v}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{load}\ \langle n\rangle} \langle C, v :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-Store
$$\frac{}{\langle C, v :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{store}\ \langle n\rangle} \langle C, V, \Delta[n \mapsto v], \Omega, A, \Phi\rangle}$$

A-Dup2
$$\frac{}{\langle C, v_1 :: v_2 :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{dup2}} \langle C, v_1 :: v_2 :: v_1 :: v_2 :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-Cover
$$\frac{V = v :: v_1 :: .. :: v_n :: V'}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{cover}\ \langle n\rangle} \langle C, v_1 :: .. :: v_n :: v :: V', \Delta, \Omega, A, \Phi\rangle}$$

A-PushBytes
$$\frac{}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{pushbytes}\ \langle b_1\ ..\ b_n\rangle} \langle C, [b_1,\ ..\ , b_n] :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-AppLocalGet
$$\frac{v = (a, k) \in \mathtt{dom}(\Omega)\ ?\ \Omega(a, k)\ :\ 0}{\langle C, k :: a :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{app_local_get}} \langle C, v :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-AppLocalPut
$$\frac{v = \Delta(n)}{\langle C, v :: k :: a :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{app_local_put}\ \langle n\rangle} \langle C, V, \Delta, \Omega[\langle a, k\rangle \mapsto v], A, \Phi\rangle}$$

A-AppLocalDel
$$\frac{}{\langle C, k :: a :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{app_local_del}} \langle C, V, \Delta, \Omega \setminus \langle a, k\rangle, A, \Phi\rangle}$$

Rules (A-Load) and (A-Store) are self-explanatory. Rule (A-Dup2) replicates both the top of the stack and the value underneath, while rule (A-Cover) pushes the top of the stack down by n positions. Rule (A-AppLocalGet) returns the value associated with the (address, key) pair, if a value exists, otherwise it returns 0. Rule (A-AppLocalPut) creates a new local state binding $\langle a, k\rangle \mapsto v$, from data retrieved from the operand stack: here a is expected to be a 32-byte array representing an address and k is an arbitrary byte array. Dually, rule (A-AppLocalDel) discards biding, if it exists.

TEAL state transitions for transactions (excerpts)

A-Itxn-Begin

$$\frac{}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{itxn_begin}} \langle C, V, \Delta, \Omega, A, \varnothing\rangle}$$

A-Itxn-Field

$$\frac{}{\langle C, v :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{itxn_field}\ \langle x\rangle} \langle C, V, \Delta, \Omega, A, \Phi[x \mapsto v]\rangle}$$

A-Itxn-Submit-Xfer

$$\frac{\begin{array}{l}\Phi[\texttt{Type}] = \texttt{"axfer"} \\ a_s = \Phi[\texttt{AssetSender}] \quad n = \Phi[\texttt{XferAsset}] \quad a_r = \Phi[\texttt{AssetReceiver}] \\ m = \Phi[\texttt{AssetAmount}] \quad m_s = A(a_s, n) \quad m_r = A(a_r, n) \quad m_s \geq m\end{array}}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{itxn_submit}} \langle C, V, \Delta, \Omega, A[\langle a_s, n\rangle \mapsto m_s - m][\langle a_r, n\rangle \mapsto m_r + m], \varnothing\rangle}$$

A-Itxn-Submit-Create

$$\frac{\Phi[\texttt{Type}] = \texttt{"acfg"} \quad a = \Phi[\texttt{Sender}] \quad m = \Phi[\texttt{Total}] \quad (n\ \texttt{fresh})}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{itxn_submit}} \langle C, V, \Delta, \Omega, A[\langle a, n\rangle \mapsto m], \varnothing\rangle}$$

Rule (A-Itxn-Begin) initializes an inner transaction by clearing the Φ component of the state tuple. Rule (A-Itxn-Field) updates the transaction by adding a new binding for the field name x into Φ, where x ranges over the transaction field identifiers, such as `Type`, `AssetSender`, `AssetReceiver`, and others. The (A-Itxn-Submit) rules commit the current transaction and dispatch according to the `Type` field. In particular, rule (A-Itxn-Submit-Xfer) handles asset transfers by decrementing the balance of the sender a_s and incrementing the balance of the receiver a_r by the amount m being transferred. Rule (A-Itxn-Submit-Create) handles asset creation, introducing a new binding into A that records the amount m of minted units associated with the address a. The asset identifier n is generated by Algorand and treated as a fresh numeric index.

6. Encoding AlgoMove into TEAL

The encoding of AlgoMove into TEAL is primarily a bytecode-to-bytecode translation, where each Move bytecode instruction is mapped to a corresponding TEAL opcode. There are a few exceptions to this general scheme, notably for Move instructions that manage global resources and for AlgoMove's asset primitives. Additional technical considerations arise from the need to align the different calling conventions of the respective virtual machines, as well as to bridge the gap between the datatype abstractions of the two bytecodes: structures and references in Move versus integers and byte arrays in TEAL. In particular, encoding references is technically subtle, as it requires an ad-hoc serialization mechanism along with dedicated TEAL subroutines, provided as part of a runtime AVM library, to capture the behavior of Move's **ReadRef** and **WriteRef** instructions.

The translation operator, denoted by $[\![\cdot]\!]_P$, is parametric in a program P and relies on the serialization mechanism, denoted by $\|\cdot\|$. The complete set of translation clauses, along with the details of the serialization mechanism, is provided in Appendix C. In the following, we discuss some representative rules.

The first block illustrates the encoding of a selection of core Move instructions. The cases of **MoveLoc** $\langle n\rangle$ and **StLoc** $\langle n\rangle$ are straightforward. In rule T-Pack, the meta-notation ***itob?nop*** $\langle\tau\rangle$ stands for either an **itob** instruction or (the TEAL equivalent of) a no-op, depending on whether the type τ is a `struct` type name. Rule (T-MoveTo) emits an `app_local_put` instruction, which takes the top of the stack as the main argument (i.e., the data to be stored) and uses the one-byte representation of the type name s as the key. In Move, only structs can be stored globally; therefore, whatever precedes the TEAL instructions emitted by this rule is guaranteed to be a byte array, in compliance with the type constraints enforced by `app_local_put`. Rule (T-MoveFrom) reproduces the semantics of (M-MoveFrom) by deleting the entry from Algorand's local state immediately after the `app_local_get`. This behavior cannot lead to error states in TEAL, such as attempts to access deleted keys, since the Move type system and borrow checker prevent such scenarios.

Encoding excerpts: a selection of Move instructions

(T-MoveLoc)

$[\![\textbf{MoveLoc}\ \langle n \rangle]\!]_P \quad = \quad \textbf{load}\ \langle n \rangle$

(T-Pack: $P(s) = \texttt{struct}\ s\ \{x_1 : \tau_1 \ldots x_n : \tau_n\}$)

$$[\![\textbf{Pack}\ \langle s \rangle]\!]_P \quad = \quad \left.\begin{array}{l}\textbf{uncover}\ \langle n-1 \rangle \\ \textit{itob?nop}\ \langle \tau_1 \rangle \\ \vdots \\ \textbf{uncover}\ \langle n-1 \rangle \\ \textit{itob?nop}\ \langle \tau_n \rangle \end{array}\right\}\ n\ \texttt{times} \quad \left.\begin{array}{l}\textbf{concat} \\ \vdots \\ \textbf{concat}\end{array}\right\}\ n\ \texttt{times}$$

(T-StoreLoc)

$[\![\textbf{StLoc}\ \langle n \rangle]\!]_P \quad = \quad \textbf{store}\ \langle n \rangle$

(T-MoveTo)

$$[\![\textbf{MoveTo}\ \langle s \rangle]\!]_P \quad = \quad \begin{array}{l}\textbf{pushbytes}\ \langle [\![s]\!]_1 \rangle \\ \textbf{swap} \\ \textbf{app_local_put}\end{array}$$

(T-MoveFrom)

$$[\![\textbf{MoveFrom}\ \langle s \rangle]\!]_P \quad = \quad \begin{array}{l}\textbf{pushbytes}\ \langle [\![s]\!]_1 \rangle \\ \textbf{dup2} \\ \textbf{app_local_get} \\ \textbf{cover}\ \langle 2 \rangle \\ \textbf{app_local_del}\end{array}$$

The next block of encoding clauses handles the translation of AlgoMove assets into standard Algorand assets. The three opcodes therein defined are actually virtual instructions defined for the sake of our formalization: an actual implementation would rather detect `Call` instructions to native functions representing the `deposit`, `withdraw` and `create` primitives.

Encoding excerpts: AlgoMove asset manipulation instructions

(T-Deposit)

$[\![$**Deposit** $\langle s\rangle]\!]_P$ =
  swap
  $[\![$**Unpack** ⟨`Asset`⟩$]\!]_P$
  itxn_begin
  pushbytes ⟨`"axfer"`⟩
  itxn_field ⟨`Type`⟩
  itxn_field ⟨`XferAsset`⟩
  itxn_field ⟨`AssetAmount`⟩
  itxn_field ⟨`Sender`⟩
  itxn_field ⟨`AssetReceiver`⟩
  itxn_submit

(T-Create)

$[\![$**Create** $\langle s\rangle]\!]_P$ =
  pop
  global ⟨`CurrentApplicationAddress`⟩
  dup2
  itxn_begin
  pushbytes ⟨`"acfg"`⟩
  itxn_field ⟨`Type`⟩
  itxn_field ⟨`Sender`⟩
  itxn_field ⟨`Total`⟩
  pushbytes $\langle \lfloor\!\lfloor s \rfloor\!\rfloor \rangle$
  itxn_field ⟨`Name`⟩
  itxn_submit
  swap
  itxn ⟨`CreatedAssetID`⟩
  $[\![$**Pack** ⟨`Asset`⟩$]\!]_P$

(T-Withdraw)

$[\![$**Withdraw** $\langle s\rangle]\!]_P$ =
  itxn_begin
  itxn_field ⟨`Sender`⟩
  pushbytes ⟨`"axfer"`⟩
  itxn_field ⟨`Type`⟩
  pushbytes $\langle \lfloor\!\lfloor s \rfloor\!\rfloor \rangle$
  callsub ⟨`retrieve_id_by_name`⟩
  dup2
  itxn_field ⟨`XferAsset`⟩
  itxn_field ⟨`AssetAmount`⟩
  global ⟨`CurrentApplicationAddress`⟩
  itxn_field ⟨`AssetReceiver`⟩
  itxn_submit
  global ⟨`CurrentApplicationAddress`⟩
  cover ⟨2⟩
  $[\![$**Pack** ⟨`Asset`⟩$]\!]_P$

Rule (T-Deposit) sets up an inner transaction of type `axfer` (asset transfer) and populates the relevant fields. Since the second stack-argument is a structured datatype `Asset`, an unpacking step is required to extract its fields — the asset-id, the amount, and the owner. Notably, the type argument s is unused in the translation. Rule (T-Withdraw) performs the inverse operation and reconstructs an `Asset` from the same fields. In this case, however, the asset-id is not directly available and must be retrieved via an auxiliary subroutine, which we omit for brevity. Given the type name s converted into a byte string, the subroutine searches for the asset by name using Algorand's special transactions that access the contract's opted-in asset list, eventually returning the corresponding ID. Rule (T-Create) sets up an inner transaction of type `acfg` (asset configuration) to mint a specified amount of fresh assets. Newly created assets are assigned the name s, allowing subsequent retrieval through the `retrieve_id_by_name` subroutine mentioned above. Once the inner transaction has been submitted, the asset-id generated by the platform is retrieved and packed into the returned struct.

Properties of the encoding

We conclude the presentation of the encoding by establishing two main properties. The first is a standard operational correspondence result, which states that the AlgoMove state transitions of any program are faithfully mapped to the

corresponding TEAL state transitions for the encoded (TEAL) program. This correspondence is illustrated by the following commuting diagram in Figure 2.

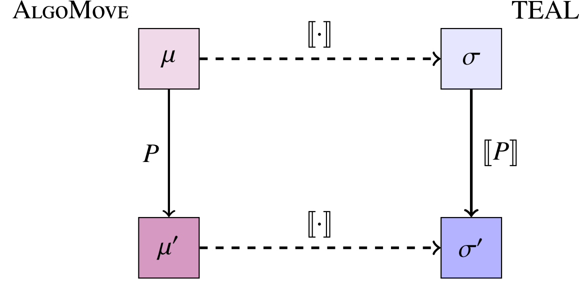

Figure 2: The diagram illustrates the ALGOMOVE translation system as a commutative square. In the top-left corner, we have an ALGOMOVE program P in an initial state μ. In the top-right corner, its translation $[\![P]\!]$ along with the corresponding TEAL state σ, which encodes μ. The left vertical arrow represents the execution semantics of P, transitioning from μ to a final state μ'. The right vertical arrow depicts the analogous transition on the TEAL side. The bottom horizontal arrow is the property we aim to prove: that the final TEAL state σ' encodes the final ALGOMOVE state μ'. The thicker lines highlight the novel contributions of this paper, with particular emphasis on the bottom horizontal arrow.

As a direct corollary, we derive the second relevant property, namely that the encoding is *resource safe*. This implies that the language-level management of assets embodied by ALGOMOVE is sound with respect to the layer-1 mechanisms provided by Algorand. In turn, this enables the seamless integration between ALGOMOVE and TEAL asset management via smart contracts.

It is worth noting, however, that this safety result applies only to contracts deployed on the Algorand blockchain that interact with other contracts adhering to the same invariants enforced by the ALGOMOVE verification toolchain. In other words, while the operational correspondence result ensures that the encoding is safe, it does not guarantee robust safety [36]. Also, safety guarantees apply only to contracts that rely on the ALGOMOVE library for asset management. As discussed in Section 7, the framework also permits the inlining of low-level TEAL instructions; if used without discipline, this capability may break the intended security guarantees.

Operational correspondence

The operational correspondence proof relies on several auxiliary definitions that formalize the encoding of values, stacks, and states. For conciseness, we introduce these definitions informally here and refer the reader to Section D for the complete formal treatment.

Encoding Values. The encoding of values is defined case-by-case, according to the various ALGOMOVE datatypes. Short integers and booleans admit a direct representation in TEAL. All other ALGOMOVE datatypes — including 64-byte integers, references, strings, and (nested) struct fields — are encoded in TEAL through their byte-level serialization. We denote by $\lfloor\!\lfloor v \rfloor\!\rfloor$ the encoding of an ALGOMOVE value v (omitting the subscript indicating the byte length; see Appendix D for details).

Encoding States. The encoding of states proceeds component-wise, translating ALGOMOVE state tuples $\langle M, L, G, S, B\rangle$ into the corresponding components of TEAL state tuples $\langle C, V, \Delta, \Omega, A, \Phi\rangle$. Specifically:

- C and V encode the program points and operand stack entries from the Move stack S;
- Δ is the map $n \mapsto \lfloor\!\lfloor M(L(n)) \rfloor\!\rfloor$, corresponding to the ALGOMOVE local memory L;
- Ω is the map $(\lfloor\!\lfloor a \rfloor\!\rfloor, \lfloor\!\lfloor s \rfloor\!\rfloor) \mapsto \lfloor\!\lfloor M(G(a, s)) \rfloor\!\rfloor$, corresponding to the ALGOMOVE global state G;
- A is the map $(\lfloor\!\lfloor a \rfloor\!\rfloor, \lfloor\!\lfloor s \rfloor\!\rfloor) \mapsto \lfloor\!\lfloor B(a, s) \rfloor\!\rfloor$, corresponding to the ALGOMOVE balance B;
- Φ is set to $\varnothing$, as it has no counterpart in ALGOMOVE states.

The encoding is largely straightforward; as previously noted, we disregard TEAL boxes and global state, since these are not involved in the translation. As a further remark, observe that $\mathtt{dom}(L) = \mathtt{dom}(\Delta)$, meaning that local variables in Move and scratch space slots in TEAL share the same index space in any given context. This correspondence follows directly from the translation rules (T-MoveLoc), and (T-StoreLoc), which do not alter the index parameter n when emitting a **load** or **store** instruction.

The following theorem is the core property of the translation.

Theorem 6.1 (Operational Correspondence)**.** *Let P be an* AlgoMove *program, I an instruction of P and* $[\![I]\!]_P = J_1 \cdots .. J_n$, $(n \geq 1)$ *be the TEAL instruction sequence encoding I. Then, for any* AlgoMove *transition* $\mu \xrightarrow{I} \mu'$*, there exist TEAL states* $\sigma_1, \ldots, \sigma_{n-1}$ *such that* $[\![\mu]\!] \xrightarrow{J_1} \sigma_1 \cdots \xrightarrow{J_i} \cdots \sigma_{n-1} \xrightarrow{J_n} [\![\mu']\!]$.

Resource and Asset Preservation

By a straightforward inductive argument, Theorem 6.1 extends directly to any sequences of AlgoMove instructions. To simplify notation, we write $\sigma \xRightarrow{J_1 \ldots J_k} \sigma'$ to denote the sequence of TEAL state transitions $\sigma \xrightarrow{J_1} \ldots \xrightarrow{J_k} \sigma'$. Now let $\mu_0 \xrightarrow{I_1} \mu_1 \xrightarrow{I_2} \cdots \xrightarrow{I_k} \mu_k \xrightarrow{I_{k+1}} \cdots \xrightarrow{I_n} \mu_n$ be a sequence of AlgoMove transitions. Then, there exists a corresponding sequence of TEAL transitions $\sigma_0 \xRightarrow{[\![I_1]\!]} \sigma_1 \xRightarrow{[\![I_2]\!]} \cdots \xRightarrow{[\![I_k]\!]} \sigma_k \xRightarrow{[\![I_{k+1}]\!]} \cdots \xRightarrow{[\![I_n]\!]} \sigma_n$, such that $\sigma_i = [\![\mu_i]\!]$ for $i \in 1..n$. From this, it immediately follows that the encoding provides a sound representation of Algorand layer-1 assets in terms of their language-level AlgoMove counterpart. Indeed, focusing on the asset components $A_i \in \sigma_i$ and $B_i \in \mu_i$, the equality $\sigma_i = [\![\mu_i]\!]$ implies $A_i = [\![B_1]\!]$.

7. Implementation

We can interpret the encoding presented in the previous section as an abstract implementation, capturing the core semantics of TEAL programs within the AlgoMove framework. This abstraction demonstrates how TEAL's computational model can be systematically derived from the resource-aware semantics of a high-level language like Move.

In practical scenarios, however, a concrete implementation should support a more flexible execution model—one that allows developers to selectively exploit Move's expressive power and safety guarantees, while retaining the option to interact directly with native TEAL features when needed. Such a hybrid approach balances safety and control: it enables developers to write performance-critical or low-level components directly in TEAL, while relying on Move's strong static typing, ownership discipline, and formal verification support for managing resources and assets reliably.

Our prototype implementation adopts this hybrid strategy. It is composed of two distinct components:

- A **library**, implemented in Move, provides an API for writing smart contracts to be deployed on Algorand. It includes the portable subset of the Move standard library, along with a collection of additional modules offering services specifically tailored to the Algorand environment.
- A **translator** converts Move programs into TEAL. Instead of building a complete compiler from scratch, we leverage an existing Move compiler to generate Move bytecode, which is then translated into TEAL via a *transpiler*. For the prototype presented in this paper, we use the Aptos compiler, chosen for its active maintenance and its bundled bytecode disassembler. Our transpiler operates as a backend, parsing the disassembled bytecode and emitting TEAL code, which is subsequently passed to the Algorand toolchain for deployment.

Relying on the Aptos toolchain is primarily a prototyping choice. Our transpiler currently depends on the instruction syntax produced by the Aptos disassembler and on the meta-information embedded in it (e.g., certain type annotations appearing in the pretty-printed bytecode are exploited during translation). A production-ready AlgoMove ecosystem would instead provide a standalone compiler together with a dedicated toolchain. Likewise, a fully fledged framework would implement a much larger portion of the Aptos standard library, whereas the modules currently included in the prototype represent only a core subset of the most relevant data types and functions.

The Library

Much like other existing Move embeddings, such as those for Aptos, StarCoin, Libra/Diem, and Sui, the AlgoMove library extends core Move with an API that enables interaction with the underlying blockchain—in this case, Algorand.

The library is organized as a three-layered stack (cf. Figure 3). While AlgoMove smart contracts typically import modules from the topmost layer, the architecture is fully transparent: developers can also access the lower layers when needed. This allows for the direct invocation of standard transactions and offers fine-grained control over Algorand-specific features, including cryptographic primitives, memory boxes, and custom transaction construction.

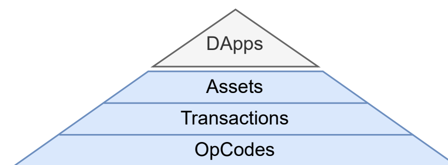

Figure 3: The AlgoMove library is structured as a three-layered stack, where each layer builds upon the abstractions of the one below. Smart contracts typically import the topmost layer, which provides high-level primitives for manipulating entities such as assets. However, all layers remain accessible, allowing developers to bypass higher abstractions when necessary. This enables direct use of lower-level APIs to interact with platform-specific features of Algorand, including cryptographic operations, access to global state, and custom transaction construction.

The OpCode Layer

The lowest layer of the library consists of a Move module declaring a set of function prototypes marked with the *native* keyword. An excerpt is shown in the table below. Each native function acts as a placeholder for a corresponding TEAL opcode bearing the same name. These functions expose only their type signatures in Move and lack any implementation; accordingly, our translation tool converts calls to such functions directly into the appropriate TEAL instructions.

As mentioned earlier, this *opcodes* layer forms the foundation of the library stack but remains accessible for direct use in AlgoMove smart contracts. This is particularly useful for accessing Algorand-specific features not abstracted by higher layers. For example, TEAL's global state model has no direct counterpart in AlgoMove. If a contract needs to be leveraged it, developers can do so by invoking the relevant native stubs provided at this level.

The AlgoMove Library – Excerpts from the *Opcode* layer

```
module algomove::opcode

// transactions
native public fun txn_Sender(): address;
native public fun txn_CreatedAssetID(): u64;
// inner transactions
native public fun itxn_begin();
native public fun itxn_submit();
native public fun itxn_field_Fee(x: u64);
native public fun itxn_field_Type(x: String);
native public fun itxn_field_Amount(x: u64);
// global instruction
native public fun global_MinBalance(): u64;
native public fun global_LatestTimestamp(): u64;
// local state
native public fun app_local_put<T: key>(addr: address, k: vector<u8>, data: T);
native public fun app_local_get<T: key>(addr: address, k: vector<u8>): T;
native public fun app_local_del(addr: address, k: vector<u8>);
// serialization
native public fun itob(data: u64): vector<u8>;
native public fun btoi(data: vector<u8>): u64;
```

Every TEAL opcode is represented by a Move native function mimicking its name. Native functions do not provide an implementation and exhibit only a signature. When called, the Move compiler emits a special `Call-Native` instruction that our translator recognizes and converts into the homonymous TEAL opcode. This trick basically allows for inlining low-level TEAL code into Move programs: the AlgoMove library itself is built upon this, providing higher-level functions that are implemented through a series of calls to the native functions defined here.

Invoking the OpCode layer directly is analogous to inlining assembly instructions in conventional programming languages. As anticipated in Section 6, this bypasses the guarantees normally enforced by the compiler and toolchain, transferring responsibility to the programmer. Although opcode stubs expose strongly typed signatures, they provide only limited type safety when invoked directly. An implementation of the AlgoMove ecosystem could alternatively restrict public access to the OpCode layer and instead expose higher-level wrappers for selected Algorand-specific operations.

The Transaction Layer

The middle layer provides a higher-level API for working with transactions, inner transactions, and transaction groups, abstracting away the underlying low-level opcodes. Internally, most of the functionality in this layer is implemented as wrappers around calls to the underlying *opcodes* module. This design promotes code reuse and readability, allowing developers to express complex transaction logic using familiar, structured abstractions. An excerpt of the API is presented in the following table.

The AlgoMove Library – Excerpts from the *Transaction* layer

```
module algomove::transaction
use algomove::opcodes;

const DEFAULT_FEE: u64 = 100;

// transaction initialization

public fun init_header(sender: address, fee: u64, ty: String) {
    opcode::itxn_begin();
    opcode::itxn_field_Fee(fee);
    opcode::itxn_field_Type(ty);
    opcode::itxn_field_Sender(sender);
}

public fun init_asset_config(sender: address, total: u64, decimals: u64, default_frozen: bool,
                             name: String, short_name: String) {
    init_header(sender, DEFAULT_FEE, utf8(b"acfg"));
    opcode::itxn_field_Total(total);
    opcode::itxn_field_Decimals(decimals);
    opcode::itxn_field_DefaultFrozen(default_frozen);
    opcode::itxn_field_Name(name);
    opcode::itxn_field_UnitName(short_name);
}

public fun init_asset_transfer(sender: address, id: u64, receiver: address) {
    init_header(DEFAULT_FEE, utf8(b"axfer"), sender);
    opcode::itxn_field_XferAsset(id);
    opcode::itxn_field_AssetReceiver(receiver);
}

// shortcuts to common transactions

public fun asset_transfer(sender: address, id: u64, amount: u64, receiver: address) {
    init_asset_transfer(sender, id, receiver);
    opcode::itxn_field_AssetSender(@0x0);
    opcode::itxn_field_AssetAmount(amount);
    opcode::itxn_submit();
}

public fun asset_config(sender: address, total: u64, decimals: u64, default_frozen: bool,
                        name: String, short_name: String) {
    init_asset_config(sender, total, decimals, default_frozen, name, short_name);
    opcode::itxn_submit();
}
```

This layer provides an easy-to-use API for performing transactions and inner transactions, such as asset creation and asset transfers (payments). The (simplified) implementations shown here rely on the underlying OpCode layer, populating transaction fields incrementally by using the `itxn_field` instruction stub. A few handy wrappers are included too, offering a mid-level abstraction for creating custom transactions, which is sometimes desirable in Algorand for performing uncommon tasks.

Invoking the Transaction layer directly does not necessarily violate safety guarantees such as the resource safety introduced in Section 6. However, if used without discipline, its API may enable the construction of ill-formed transactions (e.g., with missing mandatory fields), potentially producing TEAL code that is ineffective or fails at execution time. More importantly, manually issuing payment transactions (i.e., asset transfers) bypasses the Move resource discipline: since such operations are not visible to the Move type checker, the linear type system cannot enforce its guarantees on resource usage, including the prevention of unintended copies or drops.

The Asset Layer

The topmost layer of the library introduces the primary AlgoMove abstractions for asset management. This layer offers a strongly typed API enabling developers to create and manipulate assets through functions analogous to the `withdraw` and `deposit` operations of the Move standard library. In AlgoMove, the `Asset` type replaces Move's `Coin` type to represent fungible tokens. The linear type constraints associated with `Asset` enforce strict ownership semantics, effectively preventing unintended copies or drops. This provides compile-time guarantees against double-spending and other forms of asset mismanagement.

The AlgoMove Library – Excerpts from the *Asset* layer

```
module algomove::asset
use algomove::opcode;
use algomove::transaction;
use algomove::utils;

struct Asset<phantom AssetType> {
    id: u64,
    amount: u64,
    owner: address
}

public fun create<AssetType>(acc: &signer, total: u64, decimals: u64, default_frozen: bool,
    name: String, short_name: String): Asset<AssetType> {
    let sender = op::global_CurrentApplicationAddress();
    let name = opcode::name_of<AssetType>();
    transaction::asset_config(sender, total, decimals, default_frozen, name, short_name);
    Asset<AssetType> { id: opcode::txn_CreatedAssetID(), amount: total, owner: sender }
}

public fun deposit<AssetType>(receiver: address, assets: Asset<AssetType>) {
    let Asset { id, amount, owner } = assets;
    transaction::asset_transfer(owner, id, amount, receiver)
}

public fun withdraw<AssetType>(acc: &signer, amount: u64): Asset<AssetType> {
    let id = transaction::retrieve_asset_id<AssetType>();
    let sender = utils::address_of_signer(acc);
    assert!(amount <= opcode::asset_holding_get_AssetBalance(sender, id));
    let escrow = opcode::global_CurrentApplicationAddress();
    transaction::asset_transfer(sender, id, amount, escrow);
    Asset<AssetType> { id: id, amount, owner: escrow }
}
```

In TEAL, assets are second-class entities that require a dedicated subset of instructions. The AlgoMove abstraction elevates them to first-class, structured values that represent a wallet of created assets, including ownership information and the amount held. Here, the Move type system plays a central role: once a value of type `Asset` is constructed via the `create` function, it becomes a linear resource, meaning it can only be moved, never copied or dropped. Most notably, invoking the `deposit` function acts as a sink for a value of type `Asset`, effectively consuming it and transferring ownership, which is enforced at the type level. Conversely, the `withdraw` function serves as a constructor for `Asset`, returning a linear type that must be consumed or transferred, but cannot be duplicated or discarded.

Move's strict visibility rules further reinforce this discipline by preventing external access to the fields of a struct, even disallowing pattern matching. As a result, once an `Asset` has been withdrawn or minted via `create`, the only valid operation is to deposit it, or the program will fail to compile. This entire mechanism, which embodies Move's resource-oriented semantics, is faithfully reproduced on Algorand through the combined design of the AlgoMove library and its accompanying bytecode translator.

The implementation combines calls to the Transaction layer with invocations of native stubs from the OpCode layer, as certain operations require low-level handling. A notable example is the `withdraw` function, which must validate the user's current balance before proceeding. This is achieved through an explicit call to the native stub `asset_holding_get_AssetBalance`, which the translator converts directly into the corresponding TEAL instruction at code generation time.

The Translator

The entire AlgoMove architecture builds upon a systematic translation of standard Move instructions into their TEAL counterparts, while allowing explicit opcode invocations for operations that require direct access to Algorand's low-level infrastructure. Our translation tool works as a *transpiler* performing a bytecode-to-bytecode conversion that turns Move bytecode into TEAL. When encountering a native opcode stub, the translator bypasses the generation of a `callsub` instruction and emits the corresponding TEAL opcode directly. This mechanism is reminiscent of how intrinsics are handled in conventional compilers, and it enables seamless integration between idiomatic Move code and Algorand-specific operations—a defining feature of the AlgoMove framework.

The translator operates as a backend on disassembled Move bytecode, either produced by the Aptos compiler used as a frontend or validated by the bytecode checker. This guarantees that translation applies only to well-typed, type-safe modules. Moreover, the disassembled format provides valuable metadata, such as user-defined types and type annotations, that support accurate translation and ensure correct data serialization and deserialization. As a result, the generated TEAL code runs natively on the Algorand Virtual Machine while preserving both the functional behavior and the semantic properties of the original Move program. The tool directly implements the translation rules formalized in Section 6 and a prototype implementation is available at `github.com/alvisespano/AlgoMove`.

Figure 4 shows an end-to-end translation from Move to the output produced by our implementation, which departs in several respects from the theoretical model. The sample shows how the `deposit` function, which belongs to the `asset` layer of the AlgoMove library and is presented in Section 7, is actually translated into TEAL. The left side illustrates the Move bytecode generated by the Aptos compiler. The result is a mix of regular Move code and stubs supplied by the `opcode` layer. Each stub invocation is compiled into a `Call` instruction, which is subsequently translated by our transpiler into the corresponding TEAL opcode (on the right).

We now provide a detailed account of the internals of the translation process. Major challenges are the encoding of struct datatypes, nested field access, the handling of opaque references to both local variables and the global storage, and the reification of type arguments to enable parametric polymorphism.

Struct serialization and deserialization

Algorand does not natively support structured data types; as a result, the `Pack` and `Unpack` instructions are translated into non-trivial sequences of TEAL instructions that perform serialization and deserialization over byte arrays. The serialization format is defined as the recursive binary concatenation of all struct fields. Algorand provides only two primitive data types: integral values (`uint64`) and byte arrays. Byte-array fields require no additional processing, whereas fields of type `uint64` must be explicitly converted to and from their byte-array representation using the TEAL instructions `itob` and `btoi`, respectively, as formalized by rules (T-Pack) and (T-Unpack) in Table C.4 in the Appendix. Partial deserialization is also performed when accessing individual struct fields, as this requires extracting and decoding the corresponding segment of the serialized byte array.

The formalization of AlgoMove presented in this paper, particularly its translation subsystem, assumes a monomorphic setting in which the types of struct fields are statically known. Under this assumption, no additional metadata is embedded in the serialized representation to track field offsets or lengths. In contrast, our prototype implementa-

Move Bytecode	TEAL Bytecode

```
deposit(to: address, assets: Asset)
  0: MoveLoc[1]
  1: Unpack<Asset>
  2: Call opcode::itxn_begin
  3: LdConst([5, 97, 120, 102, 101, 114])
  4: Call opcode::itxn_field_Type
  5: Call opcode::itxn_field_Sender
  6: StLoc[2]
  7: Call opcode::itxn_field_XferAsset
  8: MoveLoc[0]
  9: Call opcode::itxn_field_AssetReceiver
 10: MoveLoc[2]
 11: Call opcode::itxn_field_AssetAmount
 12: Call opcode::itxn_submit
 13: Ret
```

```
deposit:
   proto 2 0
   load 2
0: frame_dig -2
1: dup   // Unpack start
   extract 0 8
   btoi
   swap
   dup
   extract 8 8
   btoi
   swap
   dup
   extract 16 32
   swap
   pop   // Unpack end
2: itxn_begin
3: pushbytes "axfer"
4: itxn_field Type
5: itxn_field Sender
6: store 2
7: itxn_field XferAsset
8: frame_dig -1
9: itxn_field AssetReceiver
10:load 2
11:itxn_field AssetAmount
12:itxn_submit
13:store 2
   retsub
```

Figure 4: A sample translation performed by our prototype implementation. The snippet shows a simplified version of the `deposit` function included in the `asset` module of the AlgoMove library and introduced in Section 7. Calls to the `transaction` layer have been unfolded and type parameters have been erased.

tion supports parametric polymorphism and therefore introduces additional complexity in the serialization format to accommodate struct fields whose types are generic.

References and paths

Algorand does not support memory addresses, whereas Move does. Encoding pointers/references on a platform that lacks addressable memory represents one of the main challenges of the proposed system. References in Move can target two entities: local variables and data in the global storage, respectively produced by the `BorrowLocal` and `BorrowGlobal` instructions. Local variables may have arbitrary types, whereas only structs can be persisted in the global storage. Furthermore, references are implicitly created when accessing struct fields via paths, at any level of nesting, by the `BorrowField` instruction. The example below illustrates all possible reference scenarios and their combinations.

```
struct S1 { a: u64, b: S2 }
struct S2 { c: u64, d: bool }

let n:  u64   = 23;                            // local integer
let rn: &u64  = &n;                            // reference to a local integer
let s2: S2    = S2 { c: 11, d: true };         // local struct
let r1: &bool = s2.d;                          // reference to field d of S2
let rg: &S1   = borrow_global<S1>(some_address); // reference to data in the global storage
let r2: &u64  = rg.b.c;                        // reference to nested field of S1 (path .b.c)
```

AlgoMove defines a custom binary format for encoding Move references as byte arrays on Algorand, which is exercised by the translation rules (T-BorrowLow), (T-BorrowGlobal) and (T-BorrowField) in Tables C.4 and C.5 in the Appendix. The first byte specifies the *kind* of reference: `0x00` denotes references to local variables, while `0x01` identifies references to values stored in the global storage. The second byte encodes either the scratch-space slot identifier when the reference targets a local variable, or a one-byte hash of the struct type name when it targets the global storage. In the latter case, the hash serves as a key for accessing Algorand's local state, which corresponds to

Move's global storage in our mapping.

The remainder of the byte sequence can vary in length and encodes an optional access path. Each field selection in the path is represented by a fixed-size pair (*offset*, *length*) occupying four bytes in total. Paths consisting of multiple field selections are encoded by concatenating these pairs, resulting in a linear growth of the overall representation. The *offset* component is a 16-bit word whose most significant bit acts as a flag indicating whether the selected field is a nested struct (0) or a primitive type (1). This information is exploited by the serialization and deserialization subsystem for structured values. The remaining 15 bits encode the actual offset as an unsigned value, allowing offsets of up to 32 KB, while the *length* component is encoded as an unsigned 16-bit value supporting field sizes of up to 64 KB. The following table shows the encoding of the references appearing in the Move snippet above.

Var	**Kind**	**Slot/Name**	**Path**	**Binary String** (hex)	**Size** (B)	**Remark**
`rn`	local	3		`00 03`	2	referenced variable in slot #3
`r1`	local	4	`.d`	`00 04 8008 0008`	6	referenced variable in slot #4
`rg`	global	`"S1"`		`01 AB`	2	`hash("S1")= 0xAB`
`r2`	global	`"S1"`	`.b.c`	`01 AB 0008 0010 8000 0008`	12	

Dereferencing

References support two fundamental operations: reading and writing. The translation rules (T-ReadRef) and (T-WriteRef) in Table C.4 emit a `callsub` instruction targeting two subroutines provided by the AlgoMove runtime library, which is implemented in pure TEAL and appended to the end of the output program generated by the transpiler. For brevity, these subroutines are not reproduced in this paper, as they are relatively long; however, their full implementations are available in the repository accompanying our prototype. At a high level, the subroutines decode the custom binary format used to represent references and then perform the corresponding operation. In essence, the reference format serves as a binary-level descriptor for driving the appropriate dereference operation. When the reference kind is 00, the operation directly reads from or writes to the appropriate scratch-space slot. When the kind is 01, the operation accesses or updates the corresponding struct stored in Algorand local state.

While scratch-space access maps directly to `load` and `store` instructions, manipulating local state requires more elaborate operations, such as `app_local_get` and `app_local_put`. In addition, handling access paths entails extensive bitwise operations and byte-level manipulation to deserialize, modify, and re-serialize the affected portion of a struct encoded as a byte array. This combination of factors makes reference access particularly complex and computationally expensive.

Parametric polymorphism

The formal system presented in this paper does not model Move's parametric polymorphism (i.e., generics), although full support for generics is provided by our prototype transpiler. Supporting parametric polymorphism introduces additional complexity, both for functions with type parameters and for parametric struct types. To handle generics on a target platform that lacks native type abstraction, our implementation adopts a *type-witness*-based approach in which selected static type information is reified into explicit runtime artifacts [37].

Generic functions are translated into TEAL subroutines that accept one additional argument per type parameter. At each invocation, the caller passes the corresponding type witnesses, which serve as runtime type descriptors encoding the concrete types supplied at the beginning of the call chain, where non-generic type arguments are first instantiated. The following snippet shows a simple scenario with a couple of generic function invocations; on the right, the pseudo-Move code shows the extra arguments and how type witnesses are carried along.

Move with Generics	Reification of Type Witnesses

<code>fun g<T>(x: u64, y: T) { ... }</code>
 	
<code>fun f<T>(x: u64, y: T) {</code>
<code> g(x, y);</code>
<code>}</code>
 	
<code>fun main() {</code>
<code> f<u64>(3, 7);</code>
<code>}</code>
	<code>fun g(t: type-descriptor, x: u64, y: T) { ... }</code>
	
	<code>fun f(t: type-descriptor, x: u64, y: T) {</code>
	<code> g(t, x, y); // this is just a forward</code>
	<code>}</code>
	
	<code>fun main() {</code>
	<code> // type argument u64 becomes type descriptor</code>
	<code> f(u64-type-descriptor, 3, 7);</code>
	<code>}</code>

For parametric struct types, when fields are instantiated with generic types, additional metadata is carried alongside serialized values to enable type-directed operations such as packing, unpacking, and field access. This metadata acts as a lightweight descriptor of the concrete type instance and is consumed by both the translation and runtime subsystems to compute correct layouts and to perform sound serialization and deserialization. By making type information explicit at runtime, the compiler avoids full monomorphization while preserving the safety guarantees of the Move type system. Consequently, type-dependent operations in the generated TEAL code are driven by these explicit descriptors rather than by ad-hoc assumptions about value layouts, enabling dynamic manipulation of generic data structures.

Experimental Evaluation

The objective of this evaluation is to quantify and explain the overhead introduced by the proposed system in terms of both gas consumption and code size, compared to existing Algorand programming platforms. PyTEAL is adopted as the baseline, and contracts written in AlgoMove are evaluated against functionally equivalent implementations developed in PyTEAL. In Algorand, each TEAL instruction incurs a unitary gas cost; therefore, the length of the generated code closely approximates the total gas cost, disregarding control-flow effects. Since typical smart contracts rarely rely on loops or complex branching structures, these two metrics can be considered essentially equivalent.

Our evaluation considers two scenarios. The first consists of a minimal contract exposing three entry points for asset manipulation (`deposit`, `withdraw`, and `transfer`), which are commonly used in smart contracts and provide insight into how gas costs are distributed across these operations. The second scenario is the auction contract previously introduced in Figure 1 and described in Section 3, which exercises a broad subset of language features, including structured state, on-chain state updates, borrowing and dereferencing.

Crucially, although the Move and PyTEAL contracts are semantically equivalent, as they both implement the same logic and support the same high-level interactions, they are not and *cannot be* identical. The difference is inherent to the paradigmatic gap between the two languages and their execution models: fundamental operations such as payments, asset transfers, and state access are expressed and managed differently in the two implementations. Such a gap remains unavoidable, as each contract must conform to the abstractions, conventions and idioms of its respective language and underlying platform.

Gas costs are evaluated comparing two TEAL programs, one generated by PyTEAL and the other by AlgoMove, and by measuring the total code length, which approximates the gas cost. Figure 5 summarizes the results. For each contract entry point, the cost of the PyTEAL implementation is used as a baseline, against which we report the overhead introduced by our system. We distinguish between user code and library code, as the TEAL output produced by our transpiler relies extensively on auxiliary functions provided by our runtime library. We first compare the cost of the user code generated by AlgoMove against the code generated by PyTEAL, which, notably, consists entirely of user code and invokes no auxiliary libraries. We then report the cost overhead introduced by the AlgoMove runtime library and, as the rightmost column, the overall cost ratio of AlgoMove with respect to PyTEAL.

Additionally, we report gas consumption after applying a set of optimizations to the auction contract, which significantly reduces the overall overhead. These optimizations are discussed in detail below.

Discussion

		AlgoMove Gas Consumption		
Contract	**Entry Point**	**User Code vs. PyTEAL**	**Library Overhead**	**Total**
Asset Management	`deposit`	-13%	0%	**-13%**
	`withdraw`	-11%	17%	**+18%**
	`transfer`	-39%	35%	**-4%**
Auction	`start_auction`	+14%	109%	**+123%**
	`bid`	-43%	225%	**+268%**
	`finalize_auction`	-40%	409%	**+369%**
Optimized Auction	`start_auction`	+28%	0%	**+28%**
	`bid`	-37%	53%	**+16%**
	`finalize_auction`	-33%	91%	**+58%**

Figure 5: Gas consumption overhead introduced by AlgoMove with respect to PyTEAL. Two scenarios are compared: a contract comprising basic asset management primitives and an auction. The auction costs appear twice: as translated by the baseline prototype implementation, and as processed by a series of optimizations.

Most of the observed overhead is attributable to the AlgoMove library, understood here as the combination of the three-layer library architecture introduced in this paper and the runtime support injected by the transpiler. By contrast, the translation of user code introduces only limited overhead, indicating that the core translation process is already fairly efficient, even at this early prototype stage.

The dominant source of gas overhead lies in the mismatch between the programming paradigms of PyTEAL and Move. Supporting Move's high-level abstractions, such as struct manipulation and references, requires non-trivial runtime machinery, largely concentrated in the library components that implement these abstractions. PyTEAL, on the other hand, acts as a thin Pythonic wrapper over TEAL, exposing its low-level execution model and enabling developers to write compact, gas-efficient code by directly leveraging TEAL idioms. Considering the radical paradigm shift, the additional gas cost incurred by Move's typed discipline and structured programming model ranges from moderate to heavy in the current unoptimized implementation.

- In the best case, programs that perform transactions without struct manipulation or borrowing exhibit virtually no overhead.
- In the worst case, small programs dominated by packing/unpacking, serialization, and dereferencing incur a significant overhead.
- In more realistic mixed scenarios, involving larger contracts and a balanced combination of abstraction-heavy and general-purpose logic, the overhead is expected to be moderate even without any optimization.
- Applying the optimizations discussed below, the overhead dramatically reduces in contracts involving struct manipulation and other heavy tasks.

Optimizations

The most gas-intensive operations in AlgoMove are undoubtedly two: struct packing/unpacking and dereferencing. To reduce the gap between PyTEAL and AlgoMove, we adopt a set of optimization strategies aimed primarily at lowering the gas cost of the supporting library code.

- **Resource flattening**. Move structs annotated with the `key` or `store` capability are compiled into a flat collection of entries directly stored in the Algorand local state. This layout eliminates most serialization and deserialization overhead: accessing a field no longer requires unpacking the entire struct, but only a single `app_local_get` with a statically computed key. In practice, this reduces field-access costs substantially.
- **Static path reconstruction**. Since Move structs may be nested, field-access paths can be arbitrarily deep. The current implementation resolves dereferences dynamically by progressively descending into nested structures

and deserializing increasingly smaller substrings of the original binary representation. A compiler-aware approach can instead statically analyze field paths and compute the final offset and length of the required slice, greatly reducing dereferencing costs.

- **Inlining and unfolding**. Inlining small operations and unfolding short subroutine calls reduces the number of executed `branch` and `callsub` instructions, while also avoiding repeated subroutine prologues and epilogues. Although the savings per occurrence are modest, the cumulative effect can be significant in larger contracts.
- **Asset caching**. When an asset type is not statically known, AlgoMove resolves it by inspecting existing Algorand assets available to the application, an operation with linear complexity in the number of registered assets. Caching resolved asset identifiers eliminates repeated lookups for commonly used assets, a frequent pattern in smart contracts, and substantially reduces gas consumption.

8. Limitations

Despite its expressive power, the current AlgoMove prototype presents a number of limitations. The first concerns library coverage. The current prototype implements only a subset of the Aptos standard library, mainly focusing on the abstractions required for asset-oriented programming and for the examples discussed in this paper. A complete framework would need to provide a significantly broader portion of the standard libraries and additional utilities for general-purpose smart contract development.

From a performance perspective, encoding Move abstractions on top of TEAL introduces a certain gas overhead. In particular, operations such as struct packing/unpacking and reference dereferencing must be compiled into non-trivial sequences of TEAL instructions that manipulate serialized data and reconstruct access paths at runtime. While our evaluation shows that this overhead remains reasonable in realistic scenarios and can be mitigated through optimization strategies, it reflects the inherent cost of preserving Move's structured programming model on a low-level execution environment such as Algorand.

Finally, the framework exposes low-level interfaces allowing programmers to invoke Algorand primitives directly. While these escape hatches are useful for advanced use cases, they define the boundaries of the system's safety guarantees. Invoking the transaction layer manually or performing asset transfers outside the Move library bypasses the resource discipline, delegating responsibility to the programmer. More broadly, our formal demonstrations of safety apply only to closed AlgoMove programs: they guarantee that a program respects resource safety when executed in isolation, but do not provide robust safety in the presence of other concurrently executing contracts on the Algorand blockchain.

9. Conclusions

The growing influence and worldwide adoption of DLT applications across various critical economic and societal sectors call for urgent action to provide developers with principled tools and programming language techniques.

The endeavor we have described in the present paper aligns with this call to action, and represents a first attempt to shed new insight into what such tools and techniques should provide and how. A further step towards making AlgoMove an effective platform for secure DApp design on the Algorand blockchain will strengthen the current implementation to ensure what is commonly referred to a *robust safety*, that is the ability to preserve the invariants of the AlgoMove typing system in an *open-world* setting, and specifically when the compiled code deployed on the Algorand blockchain gets to interact with smart contracts for which the static invariants may not be assumed to hold. An interesting step in that direction has recently been taken by [38] with a robustly safe Move implementation on Ethereum. The approach appears fully compatible with the current AlgoMove implementation, and we are currently considering work in that direction.

The significance of AlgoMove extends beyond its impact as Move implementation platform for Algorand. Equally important is its role in forging a conceptual link between two well-developed, but previously unrelated approaches to smart contract development. Gaining a full understanding of the essence of digital assets and of the principles for their management on distributed ledger platforms will require more effort both in theoretical research and in engineering

work. This represents a further direction of our plans for future work, which include both engineering experiments (of Move embeddings) with other platforms (e.g. Sui, Solana, Ethereum, UTXO frameworks), and foundational research to assess the formal properties of the embeddings and to contribute to the construction of a solid semantic and type-theoretic framework for digital assets.

Acknowledgements

This study was carried out within the PE0000014 - Security and Rights in the CyberSpace (SERICS) and received funding from the European Union Next-GenerationEU - National Recovery and Resilience Plan (NRRP) – MISSION 4, COMPONENT 2, INVESTIMENT 1.3 – CUP N. H73C22000890001. This work has been also partially supported by the Research Project PRIN 2020 - CUP N. 20202FCJMH "NiRvAna - Noninterference and Reversibility Analysis in Private Blockchains" and by the National Recovery and Resilience Plan (NRRP) Project "Securing sOft- ware Platforms - SOP", CUP H73C22000890001". This manuscript reflects only the authors' views and opinions, neither the European Union northe European Commission can be considered responsible for them.

Author Contributions

Alvise Spanò: Conceptualization, Methodology, Investigation, Formal analysis, Software, Writing – original draft, Writing – review and editing. **Lorenzo Benetollo:** Methodology, Investigation, Software, Writing – original draft. **Michele Bugliesi:** Conceptualization, Supervision, Methodology, Writing – review and editing. **Silvia Crafa:** Validation, Writing – review and editing. **Dalila Ressi:** Validation, Writing – review and editing. **Sabina Rossi:** Project administration, Supervision, Validation, Writing – review and editing.

References

[1] S. Blackshear, E. Cheng, D. L. Dill, V. G. , B. Maurer, T. Nowacki, A. Pott, S. Qadeer, Rain, D. Russi, S. Sezer, T. Zakian, R. Zhou, Move: A Language with Programmable Resources, Available at: https://www.bgp4.com/wp-content/uploads/2019/06/libra-move-a-language-with-programmable-resources.pdf.
[2] Sui Foundation, Why we created sui move, Available at: https://blog.sui.io/why-we-created-sui-move/ (2023).
[3] Aptos Foundation, The aptos white paper, Available at: https://aptos.dev/aptos-white-paper (2023).
[4] The MystenLab Team, The Sui Smart Contracts Platform, Available at: https://github.com/MystenLabs/sui/blob/main/doc/paper/sui.pdf (2023).
[5] Algorand Foundation, Algokit, https://algorand.co/algokit (2024).
[6] L. Benetollo, M. Bugliesi, S. Crafa, S. Rossi, A. Spanò, Algomove - a move embedding for algorand, in: 2023 IEEE International Conference on Blockchain (Blockchain), IEEE, 2023, pp. 62–67.
[7] The 0L Network Community, Open, transparent & community driven, Available at: https://0l.network/ (2022).
[8] Starcoin Foundation, Introduction to Starcoin, Available at: https://starcoin.org/downloads/Starcoin_Whitepaper.pdf (2022).
[9] A. Yekovenko, Solana: A new architecture for a high performance blockchain v0.8.13, Available at: https://solana.com/solana-whitepaper.pdf (2022).
[10] B. Anderson, Writing an llvm backend for the move language in rust, https://brson.github.io/2023/03/12/move-on-llvm#writing-an-llvm-backend-with-rust (2023).
[11] Solana Labs Inc., Move to solana, https://github.com/solana-labs/move/tree/llvm-sys/language/tools/move-mv-llvm-compiler (2023).
[12] Anza Technologies Inc., Embedding the move language, https://docs.anza.xyz/proposals/embedding-move (2024).
[13] C. Dannen, Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners, 1st Edition, Apress, USA, 2017.
[14] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on Ethereum smart contracts (SoK), in: Principles of Security and Trust (POST), Vol. 10204 of LNCS, Springer, 2017, pp. 164–186.
[15] D. Ressi, A. Spanò, L. Benetollo, C. Piazza, M. Bugliesi, S. Rossi, Vulnerability detection in ethereum smart contracts via machine learning: A qualitative analysis, arXiv preprint arXiv:2407.18639.
[16] B. Bellaj, A. Ouaddah, N. Crespi, A. Mezrioui, E. Bertin, A transpilation-based approach to writing secure access control smart contracts, in: 2023 5th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS), IEEE, 2023, pp. 1–7.
[17] R. Karanjai, L. Xu, W. Shi, Solmover: Smart contract code translation based on concepts, in: Proceedings of the 1st ACM International Conference on AI-Powered Software, 2024, pp. 112–121.
[18] N. D. Matsakis, F. S. Klock, The rust language, ACM SIGAda Ada Letters 34 (3) (2014) 103–104.
[19] K. Klas, Smart contract development: Move vs rust, Available at: https://medium.com/@kklas/smart-contract-development-move-vs-rust-4d8f84754a8f (2022).
[20] Algorand Foundation, Algorand python, Available at: https://algorandfoundation.github.io/puya/ (2024).
[21] Algorand Foundation, Pyteal: Algorand smart contracts in python, Available at: https://pyteal.readthedocs.io/en/stable/ (2022).
[22] Tinyman Corp., Tealish, https://tealish.tinyman.org/en/latest/ (2024).

[23] Reach Platform Inc., The reach language, `https://github.com/reach-sh/reach-lang` (2023).
[24] J. Chen, S. Gorbunov, S. Micali, G. Vlachos, Algorand agreement: Super fast and partition resilient byzantine agreement, Available at: `https://eprint.iacr.org/2018/377` (2018).
[25] D. Shamanaev, The move book, Available at: `https://move-book.com/`.
[26] S. Blackshear, D. L. Dill, S. Qadeer, C. W. Barrett, J. C. Mitchell, O. Padon, Y. Zohar, Resources: A safe language abstraction for money, CoRR abs/2004.05106.
URL `https://arxiv.org/abs/2004.05106`
[27] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), ACM, 2016, pp. 254–269. doi:10.1145/2976749.2978309.
[28] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on ethereum smart contracts, International Conference on Principles of Security and Trust (POST).
[29] Aptos Labs, Reentrancy attacks, `https://move-developers-dao.gitbook.io/aptos-move-by-example/move-vs-solidity/reentrancy-attacks`, accessed: 2026-03-11 (2023).
[30] D. Esterhuizen, S. Blackshear, D. L. Dill, S. Qadeer, S. Tasiran, The move prover, in: International Conference on Computer Aided Verification (CAV), Springer, 2022.
[31] C. Torres, et al., Security issues in move smart contracts, arXiv preprint arXiv:2302.07359.
[32] M. Bartoletti, L. Benetollo, M. Bugliesi, S. Crafa, G. Dal Sasso, R. Pettinau, A. Pinna, M. Piras, S. Rossi, S. Salis, et al., Smart Contract Languages: a Comparative Analysis, Future Generation Computer Systems (2024) 107563.
[33] Z. Sun, X. Luo, Y. Zhang, Panda: Security analysis of algorand smart contracts, in: 32nd USENIX Security Symposium (USENIX Security), 2023, pp. 1811–1828.
[34] SWC Foundation, Swc registry: Smart contract weakness classification, `https://swcregistry.io/`, accessed: 2026-03-11 (2023).
[35] MITRE Corporation, Common weakness enumeration (cwe), `https://cwe.mitre.org/`, accessed: 2026-03-13 (2024).
[36] M. Patrignani, S. Blackshear, Robust safety for move, in: 2023 IEEE 36th Computer Security Foundations Symposium (CSF), IEEE, 2023, pp. 308–323.
[37] J. P. Magalhães, A. Löh, A formal comparison of approaches to datatype-generic programming, arXiv:1202.2920.
URL `https://arxiv.org/abs/1202.2920`
[38] L. Benetollo, A. Lackner, M. Maffei, M. Scherer, Move to EVM, Personal Communication (2025).
[39] Algorand Developer Portal, Algorand developer docs, Available at: `https://developer.algorand.org/docs/`.

Appendix

In the Appendix, we provide a detailed account of the AlgoMove framework, including the full semantics of our extension of Move and TEAL, the complete bytecode-to-bytecode translation rules, and the corresponding proofs of soundness. Some of the material presented here has already appeared throughout the paper in various excerpts; for completeness, we replicate this material alongside the additional content to serve as a comprehensive reference.

A. Semantics of the AlgoMove Language

We now provide a full formalization of what we refer to as the AlgoMove language in this paper, that is, the original Move bytecode extended with three additional virtual opcodes representing the basic primitives offered by the Asset layer of the accompanying library.

Syntax

Programs P consist of a possibly empty sequence of datatype definitions T, followed by one or more functions F, whose bodies contain instructions I. Each instruction is annotated with a unique label ℓ marking its program location. Types τ are limited to a few built-in types, references, and user-defined type names s. Notably, only type names s appear in the bytecode, while full types τ are used only in type definitions T and function definitions F. Values v include basic literals, records, and references. Records are sets of (identifier, value) pairs representing instances of struct datatypes. References consist of a memory location c, which is an opaque memory address, together with a path p, which encodes a possibly empty sequence of record field selections.

Instructions I cover the most relevant Move opcodes, including the primitives for accessing the global storage, such as the `move_to`, which were part of the original design[4] of the language [25]. Instructions that are not relevant with respect to our translation system have been omitted; for example, **FreezeRef** does not appear, as it does not affect the semantics but is only relevant for static bytecode verification. Similarly, the immutable variants of the borrow instructions, such as **ImmBorrowLoc**,

[4] At the time of writing, Sui Move [4] has removed such primitives from the language, although Aptos and other Move embeddings still support them.

ImmBorrowField, etc., are not represented here for the same reason. Most instructions are parametric over a single immediate argument, except for **BorrowField**, which is parametric over two: a struct name s and a field name x. The meta instruction *Op* represents any binary operator, such as arithmetic and logical operators, whose semantics and translation are straightforward. The three instructions dealing with assets are not actual Move opcodes, but rather virtual instructions that extend the original language with primitives for asset management. In a real-world setting, a Move compiler would emit **Call** instructions to invoke the corresponding native functions provided by the AlgoMove library.

$$
\begin{array}{lcll}
P & ::= & T^* \; F^+ & \text{programs} \\
T & ::= & \texttt{struct}\; s \; \{ \; x_1 : \tau_1 \; .. \; x_n : \tau_n \; \} & \text{user-defined datatypes} \\
F & ::= & f(x_1 : \tau_1 \; .. \; x_n : \tau_n) : \tau_0 \; \{ \; (\ell : \; I)^+ \; \} & \text{function definitions} \\
\\
\tau & ::= & & \text{types} \\
 & & \texttt{u64} \mid \texttt{bool} \mid \texttt{address} & \text{built-in types} \\
 & \mid & s & \text{struct type names} \\
 & \mid & \&\tau & \text{references} \\
\\
I & ::= & & \text{instructions} \\
 & & \textbf{MoveLoc}\,\langle n \rangle \mid \textbf{CopyLoc}\,\langle n \rangle \mid \textbf{StLoc}\,\langle n \rangle \mid \textbf{BorrowLoc}\,\langle n \rangle & \text{local variables} \\
 & \mid & \textbf{ReadRef} \mid \textbf{WriteRef} & \text{references} \\
 & \mid & \textbf{Pack}\,\langle s \rangle \mid \textbf{Unpack}\,\langle s \rangle \mid \textbf{BorrowField}\,\langle s.x \rangle & \text{records} \\
 & \mid & \textbf{MoveTo}\,\langle s \rangle \mid \textbf{MoveFrom}\,\langle s \rangle \mid \textbf{BorrowGlobal}\,\langle s \rangle \mid \textbf{Exists}\,\langle s \rangle & \text{global storage} \\
 & \mid & \textbf{Call}\,\langle f \rangle \mid \textbf{Ret} \mid \textbf{Branch}\,\langle \ell \rangle \mid \textbf{BrCond}\,\langle \ell \rangle & \text{calls and jumps} \\
 & \mid & \textbf{Pop} \mid \textbf{LdConst}\,\langle v \rangle \mid \boldsymbol{Op} & \text{stack operations} \\
 & \mid & \textbf{Create}\,\langle s \rangle \mid \textbf{Withdraw}\,\langle s \rangle \mid \textbf{Deposit}\,\langle s \rangle & \text{assets} \\
\\
v & ::= & & \text{values} \\
 & & n & \text{unsigned integers} \\
 & \mid & b & \text{booleans} \\
 & \mid & a & \text{addresses} \\
 & \mid & \{ \; (x_1, v_1) \; .. \; (x_n, v_n) \; \} & \text{records} \\
 & \mid & \texttt{ref}\,\langle c, p \rangle & \text{references} \\
\\
p & ::= & & \text{field paths} \\
 & & [] & \text{empty path} \\
 & \mid & p :: x & \text{field append} \\
\\
 & & n, \ell \in \mathbb{N} & \text{immediates and labels} \\
 & & x, f, s & \text{identifiers} \\
 & & b \in \{ \; \texttt{true}, \texttt{false} \; \} & \text{booleans} \\
 & & c & \text{memory locations}
\end{array}
$$

State Transition Semantics

The operational semantics of Move is presented in the form of syntax-directed inference rules. Many of these are adapted from [36, 26], although certain aspects have been tailored for our system. We assume that any program P has either been produced by the Move compiler or successfully verified by the bytecode verifier. In other words, our semantic rules operate under the assumption that programs are type-correct. For example, no `ReadRef` instruction may appear on a non-reference value, nor may an `Unpack` instruction be applied to a non-record value. Similarly, a compiled program ensures that a `Call` instruction finds the correct number of arguments on the stack, and that a `CopyLoc` never occurs on a non-copyable datatype, thanks to the linearity check — allowing certain runtime checks to be safely omitted.

We assume that all instructions are annotated with distinct labels. Labels ℓ are positive integers representing unique program locations. Types, denoted by τ, include primitive types, struct names, and references. We do not distinguish between mutable and immutable references in the semantics, as this distinction exists only at the type level and does not affect the translation.

Before presenting the semantics of Move programs, we briefly introduce the notion of stacks. Stacks are treated as syntactic lists of items, using the append notation $i :: S$, where i is the top item and S is the remainder of the stack. The empty stack is denoted by [].

Computation states

Our semantics relies on state configurations, i.e., tuples of information representing the state of a computation. In Move, the state configuration (or simply state) includes a stack, memory, an environment of local variables, the Move global storage, and the asset balance.

Definition A.1 (Move State Configuration)**.** *The state configuration for evaluating a Move program is represented by a tuple* $\langle M, L, G, S, B\rangle$ *where:*

- $M : c \mapsto v$ *represents memory as a map from memory locations to values;*
- $L : n \mapsto c$ *represents local variables as a map from numeric indices to memory locations;*
- $G : (a, s) \mapsto c$ *represents global storage as a map from pairs (address, type name) to memory locations;*
- S *is a stack whose elements may be values* v*, program points* ℓ*, or bundles of locals* L*;*
- $B : (a, s) \mapsto n$ *represents asset balances as a map from pairs (address, type name) to numeric amounts.*

We use μ *to range over state configurations. When extended with a label* ℓ*, the pair* $\langle \ell, \langle M, L, G, S, B\rangle\rangle$ *is called a* labeled *state configuration.*

Branching and subroutines

Clauses are logical formulas of the form $\mu \xrightarrow{I} \mu'$, where μ is the input state configuration, I is an instruction, and μ' is the output state configuration. Only clauses for branch instructions take the form $\langle \ell, \mu\rangle \xrightarrow{I} \langle \ell', \mu'\rangle$, manipulating labeled state configurations to handle instructions that modify the program counter. The program counter is represented by the label ℓ, indicating the current program point. Incrementing a label, for example $\ell + 1$, corresponds to advancing the program counter to the next instruction. Rule (M-Step) acts as a generic wrapper to derive rules operating on unlabeled states. All such rules involve instructions that do not modify the program counter and are obtained as instances of (M-Step). Rules handling branch instructions update the program counter to the new program location ℓ'. Rule (M-Call) behaves like a branch to a location ℓ_1, corresponding to the first instruction of the called function f. Additionally, it pushes the current local variables L and the return program point $\ell + 1$ onto the stack. A new local variable space L' is created, and the corresponding memory locations are allocated in M'. Symmetrically, rule (M-Ret) restores the caller's program counter and local variables by popping them from the stack.

As a notation remark, $P(f)$ denotes the lookup of function identifier f in the program P, yielding the function definition F associated with f. For brevity, the program P is omitted from the premises of inference rules, as it is treated as a global parameter.

$$\textsc{M-Step}\quad \frac{\mu \xrightarrow{I} \mu'}{\langle \ell, \mu\rangle \xrightarrow{I} \langle \ell + 1, \mu'\rangle}$$

$$\textsc{M-Call}\quad \frac{P(f) = f(x_1 : \tau_1 \,..\, x_n : \tau_n) : \tau_0 \; \{ \; \ell_1 : I_1 \,..\, \ell_m : I_m \; \} \qquad L' = \langle c_1, .., c_n\rangle \qquad M' = M[c_1 \mapsto v_1]..[c_n \mapsto v_n] \qquad c_1 \,..\, c_n \notin \texttt{dom}(M)}{\langle \ell, \langle M, L, G, v_1 :: \;..\; :: v_n :: S, B\rangle\rangle \xrightarrow{\textbf{Call}\ \langle f\rangle} \langle \ell_1, \langle M', L', G, \ell + 1 :: L :: S, B\rangle\rangle}$$

$$\textsc{M-Ret}\quad \frac{}{\langle \ell, \langle M, L, G, \ell' :: L' :: S, B\rangle\rangle \xrightarrow{\textbf{Ret}} \langle \ell', \langle M, L', G, S, B\rangle\rangle}$$

$$\textsc{M-Branch}\quad \frac{}{\langle \ell, \langle M, L, G, S, B\rangle\rangle \xrightarrow{\textbf{Branch}\ \langle \ell'\rangle} \langle \ell', \langle M, L, G, S, B\rangle\rangle}$$

$$\textsc{M-BrCond-True}\quad \frac{}{\langle \ell, \langle M, L, G, \texttt{true} :: S, B\rangle\rangle \xrightarrow{\textbf{BrCond}\ \langle \ell'\rangle} \langle \ell', \langle M, L, G, S, B\rangle\rangle}$$

$$\textsc{M-BrCond-False}\quad \frac{}{\langle \ell, \langle M, L, G, \texttt{false} :: S, B\rangle\rangle \xrightarrow{\textbf{BrCond}\ \langle \ell'\rangle} \langle \ell + 1, \langle M, L, G, S, B\rangle\rangle}$$

Basic Move Instructions

The characteristic `MoveLoc` instruction is modeled by rule (M-MoveLoc), which removes the value from both memory and locals, reflecting the *must-move* semantics enforced by Move's linear type system at compile time. Rule (M-CopyLoc) behaves similarly, except that it does not perform any removal. Rule (M-StLoc) stores a value v into a local variable n by allocating a fresh memory

location c, updating memory with the binding $M[c \mapsto v]$, and mapping n to c in locals. Rule (M-BinOp) acts as a generic case for all binary operators, including arithmetic (e.g., addition, subtraction, multiplication) and logical operations.

M-MoveLoc
$$\frac{L(n) = c \qquad c \in \texttt{dom}(M)}{\langle M, L, G, S, B\rangle \xrightarrow{\textbf{MoveLoc}\ \langle n\rangle} \langle M \setminus c, L \setminus n, G, M(c) :: S, B\rangle}$$

M-CopyLoc
$$\frac{L(n) = c \qquad c \in \texttt{dom}(M)}{\langle M, L, G, S, B\rangle \xrightarrow{\textbf{CopyLoc}\ \langle n\rangle} \langle M, L, G, M(c) :: S, B\rangle}$$

M-StLoc
$$\frac{c \notin \texttt{dom}(M)}{\langle M, L, G, v :: S, B\rangle \xrightarrow{\textbf{StLoc}\ \langle n\rangle} \langle M[c \mapsto v], L[n \mapsto c], G, S, B\rangle}$$

M-Pop
$$\frac{}{\langle M, L, G, v :: S, B\rangle \xrightarrow{\textbf{Pop}} \langle M, L, G, S, B\rangle}$$

M-LdConst
$$\frac{}{\langle M, L, G, S, B\rangle \xrightarrow{\textbf{LdConst}\ \langle v\rangle} \langle M, L, G, v :: S, B\rangle}$$

M-BinOp
$$\frac{v = \llbracket v_1\ \boldsymbol{Op}\ v_2 \rrbracket}{\langle M, L, G, v_1 :: v_2 :: S, B\rangle \xrightarrow{\boldsymbol{Op}} \langle M, L, G, v :: S, B\rangle}$$

Borrows, References, Packing/Unpacking

Rule (M-BorrowLoc) produces a reference to a local variable. Rule (M-BorrowField) extends the path component p of the reference on top of the stack by appending the selected field x. Note that there is no need to check whether $x \in \{x_1, \ldots, x_n\}$, since the source program is assumed to have been successfully compiled or verified. Rule (M-Pack) constructs a record value consisting of n fields taken from the top of the stack. The notation $P(s)$ looks up the user-defined datatype s in the program P, yielding the corresponding struct definition T. Rule (M-Unpack) deconstructs a record value and pushes its components—i.e., all field values, in order of appearance—onto the stack. Rule (M-ReadRef) dereferences a reference and reads its value, whereas rule (M-WriteRef) updates the referenced value by modifying it in memory. The notation $v[p]$ denotes the value obtained by accessing v through multiple field selections according to the path p.

Definition A.2 (Field-path Read Access)**.** *Given a record value $v = \{\ (x_1, v_1)\ ..\ (x_n, v_n)\ \}$ and a path p, the read access operator $v[p]$ is defined as follows:*

$$\begin{array}{rcll} v[[]] & = & v & \texttt{empty path} \\ v[x_i :: p] & = & v_i[p] & i \in [1, n] \end{array}$$

The notation $v[p := v']$ updates the record value v by writing the new value v' into the field selected by the path p. Since the program is well-typed, each successive field selection along the path p is guaranteed to operate on a valid record.

Definition A.3 (Field-path Write Access)**.** *Given a record value $v = \{\ (x_1, v_1)\ ..\ (x_n, v_n)\ \}$ and a path p, the write access operator $v[p := v']$ is defined as follows:*

$$\begin{array}{rcll} v[[] := v'] & = & v' & \texttt{empty path} \\ v[x_i :: p := v'] & = & v_i[p := v'] & i \in [1, n] \end{array}$$

M-BorrowLoc

$$\frac{L(n) = c}{\langle M, L, G, S, B\rangle \xrightarrow{\textbf{BorrowLoc}\ \langle n\rangle} \langle M, L, G, \texttt{ref}\ \langle c, []\rangle :: S, B\rangle}$$

M-BorrowField

$$\frac{v = \texttt{ref}\ \langle c, p\rangle \qquad c \in \texttt{dom}(M) \qquad M(c)[p] = \{(x_1, v_1),\ ..\ (x_n, v_n)\ \}}{\langle M, L, G, v :: S, B\rangle \xrightarrow{\textbf{BorrowField}\ \langle s.x\rangle} \langle M, L, G, \texttt{ref}\ \langle c, p :: x\rangle :: S, B\rangle}$$

M-Pack

$$\frac{P(s) = \texttt{struct}\ s\ \{\ x_1 : \tau_1\ ..\ x_n : \tau_n\ \} \qquad v = \{\ (x_1, v_1),\ ..,\ (x_n, v_n)\ \}}{\langle M, L, G, v_1 ::\ ..\ :: v_n :: S, B\rangle \xrightarrow{\textbf{Pack}\ \langle s\rangle} \langle M, L, G, v :: S, B\rangle}$$

M-Unpack

$$\frac{v = \{\ (x_1, v_1),\ ..\ , (x_n, v_n)\ \}}{\langle M, L, G, v :: S, B\rangle \xrightarrow{\textbf{Unpack}\ \langle s\rangle} \langle M, L, G, v_1 ::\ ..\ :: v_n :: S, B\rangle}$$

M-ReadRef

$$\frac{v = \texttt{ref}\ \langle c, p\rangle \qquad c \in \texttt{dom}(M) \qquad M(c)[p] = v}{\langle M, L, G, v :: S, B\rangle \xrightarrow{\textbf{ReadRef}} \langle M, L, G, v :: S, B\rangle}$$

M-WriteRef

$$\frac{v_2 = \texttt{ref}\ \langle c, p\rangle \qquad v = M(c)}{\langle M, L, v_1 :: v_2 :: S, B\rangle \xrightarrow{\textbf{WriteRef}} \langle M[c \mapsto v[p := v_1]], L, G, S, B\rangle}$$

Global Storage

Rule (M-MoveTo) writes a value v into the global storage G, binding it to a pair (address, type name). Rule (M-MoveFrom) behaves like (M-MoveLoc): it retrieves the stored value from the global storage and then removes it. Rule (M-BorrowGlobal) produces a reference to a value stored in the global storage, similarly to (M-BorrowLoc). Rule (M-Exists) simply checks whether the given (address, type name) pair exists in the global storage.

M-MoveTo

$$\frac{\langle a, s\rangle \notin \texttt{dom}(G) \qquad c \notin \texttt{dom}(M)}{\langle M, L, G, a :: v :: S, B\rangle \xrightarrow{\textbf{MoveTo}\ \langle s\rangle} \langle M[c \mapsto v], L, G[\langle a, s\rangle \mapsto c], S, B\rangle}$$

M-MoveFrom

$$\frac{G(a, s) = c \qquad M(c) = v}{\langle M, L, G, a :: S, B\rangle \xrightarrow{\textbf{MoveFrom}\ \langle s\rangle} \langle M \setminus c, L, G \setminus \langle a, s\rangle, v :: S, B\rangle}$$

M-BorrowGlobal

$$\frac{G(a, s) = c}{\langle M, L, G, a :: S, B\rangle \xrightarrow{\textbf{BorrowGlobal}\ \langle s\rangle} \langle M, L, G, \texttt{ref}\ \langle c, []\rangle\rangle :: S, B\rangle}$$

M-Exists

$$\frac{b \iff \langle a, s\rangle \in \texttt{dom}(G)}{\langle M, L, G, a :: S, B\rangle \xrightarrow{\textbf{Exists}\ \langle s\rangle} \langle M, L, G, b :: S, B\rangle}$$

Assets

Direct calls to the functions responsible for asset management are captured by providing a high-level description of their intended semantics. The notation $\llbracket s \rrbracket_8$ denotes a 64-bit unsigned integer uniquely associated with type name s.

Rather than specifying the granular semantics of low-level native calls that interface with TEAL instructions, we adopt a top-down approach that directly describes the intended behavior of the high-level primitives for asset creation and transfer. This choice reflects the fact that such primitives are ubiquitous across Move-based platforms and have become a distinctive hallmark of the language. Our framework reformulates these primitives for Algorand while preserving both their original signatures and intended behavior, thereby making them an ideal entry point for formalizing asset dynamics. Rule (M-Deposit) specifically captures `Call` instructions invoking the `asset::deposit()` function. Its semantics is straightforward: it increases the amount of assets of type s held by address a by the amount m. In our framework, assets are represented by an `Asset` datatype, which is parametric over a phantom type `AssetType`. This type parameter, corresponding to s, identifies the fungible token as an opaque type name. The rule assumes v is a record value comprising a single field and representing an instance of the `Asset` datatype. In our implementation

(cf. Section 7), such datatype encapsulates the asset owner, the amount, and the asset identifier, as this information is required by Algorand. On the Move side, however, the state transition semantics do not require all these fields, and some of them are therefore omitted. Rule (M-Withdraw) is the counterpart of (M-Deposit), capturing the semantics of the `asset::withdraw()` function. It decreases the asset balance of address a by the amount m being withdrawn for asset type s. Rule (M-Create) handles calls to the `asset::create()` function, minting m units of the asset type s and introducing a new binding into the B map.

M-Deposit
$$\frac{v = \{\, (\texttt{amount}, m) \,\} \qquad n = B(a, s)}{\langle M, L, G, a :: v :: S, B\rangle \xrightarrow{\textbf{Deposit}\ \langle s\rangle} \langle M, L, G, S, B[\langle a, s\rangle \mapsto n + m]\rangle}$$

M-Withdraw
$$\frac{v = \{\, (\texttt{amount}, m) \,\} \qquad n = B(a, s) \qquad n \geq m}{\langle M, L, G, a :: m :: S, B\rangle \xrightarrow{\textbf{Withdraw}\ \langle s\rangle} \langle M, L, G, v :: S, B[\langle a, s\rangle \mapsto n - m]\rangle}$$

M-Create
$$\frac{v = \{\, (\texttt{amount}, m) \,\}}{\langle M, L, G, a :: m :: S, B\rangle \xrightarrow{\textbf{Create}\ \langle s\rangle} \langle M, L, G, v :: S, B[\langle a, s\rangle \mapsto 0]\rangle}$$

B. Semantics of the TEAL Language

Here we provide a formalization of the subset of TEAL involved in our translation process. This includes all general-purpose instructions plus a number of special-purpose opcodes for accessing the local state and for dealing with transactions. Programs Q are flat sequences of either labeled or unlabeled instructions J, since a single Move instruction may be translated into a block of TEAL code marked by a single label. Values υ include 64-bit unsigned integers and byte arrays; no additional datatypes are supported by the platform, with addresses represented as fixed-size 32-byte arrays. The non-terminal J includes only those opcodes that are relevant to the translation and is not intended to fully characterize the TEAL language. For instance, cryptographic operations such as `keccak256` and `sha256`, along with other special-purpose instructions, are omitted. All general-purpose instructions are included, together with the subset of opcodes pertaining to the Algorand transaction system, which is responsible for asset creation and transfer—both central to the semantics of AlgoMove.

That said, the framework implementation provides programmers with a low-level library (the OpCode layer introduced in Section 7) that allows the translator to emit arbitrary TEAL opcodes. Consequently, a contract written in Move through our framework can, if desired, access the entire TEAL instruction set, enabling custom transactions, invocation of built-in cryptographic primitives, and any other operation supported by the Algorand platform. Such usage, however, is left to the discretion and responsibility of the developer. In this paper, we focus on formalizing the subset of TEAL that falls within the scope of the AlgoMove framework and translation system, and for which soundness can be formally established.

Q	::=	$(\ell : J^+)^+$	programs
J	::=		instructions
		load $\langle n\rangle$ | **store** $\langle n\rangle$ | **loads**	scratch space
	|	**extract** $\langle n, m\rangle$ | **extract3** | **replace2** $\langle n\rangle$ | **itob** | **btoi** | **concat**	data manipulation
	|	**pop** | **dupn** $\langle n\rangle$ | **swap**	operand stack manipulation
	|	**frame_dig** $\langle n\rangle$ | **frame_bury** $\langle n\rangle$ | **cover** $\langle n\rangle$ | **uncover** $\langle n\rangle$	call stack manipulation
	|	**proto** $\langle n, m\rangle$ | **callsub** $\langle \ell\rangle$ | **retsub** | **b** $\langle \ell\rangle$ | **bz** $\langle \ell\rangle$ | **bnz** $\langle \ell\rangle$	calls and jumps
	|	**app_local_put** | **app_local_get** | **app_local_del**	access to the local state
	|	**pushbytes** $\langle n_1 \,..\, n_m\rangle$ | **pushint** $\langle n\rangle$	other instructions
	|	**txn** $\langle x\rangle$ | **itxn_begin** | **itxn_field** $\langle x\rangle$ | **itxn_submit**	transactions and inner transactions
υ	::=		values
		n	unsigned integers
	|	$[b_1, \,..\, , b_m]$	byte arrays
		$n, m, b, \ell \in \mathbb{N}$	numeric immediates, bytes and labels
		x	field name identifiers

Computation states

We now introduce the definition of the TEAL state configuration, in analogy with Definition A.1.

Definition B.1 (TEAL State Configuration)**.** *The state configuration for the evaluation of a TEAL program is represented by a tuple where:*

- *C is the call stack, where stored items can either be program points ℓ or tuples of values $\langle \upsilon_1, .., \upsilon_n\rangle$;*
- *V is the operand stack, where stored items can only be values υ;*
- *$\Delta : n \mapsto \upsilon$ represents the scratch space as a map from numeric indexes to values;*
- *$\Omega : (a, k) \mapsto \upsilon$ represents the Algorand local state as a map from $(address, key)$ pairs to values, where k is a byte array of any size.*
- *$A : (a, n) \mapsto m$ is the asset balance, which maps (address, asset-id) pairs to the amount held by that address.*
- *$\Phi : x \mapsto \upsilon$ represents the transaction currently under construction as a dictionary mapping field names into values.*

We use σ to range over TEAL state configurations. When extended with a label ℓ, we call $\langle \ell, \langle C, V, \Delta, \Omega, A, \Phi\rangle\rangle$ a labeled state configuration.

State Transition Semantics

We present the operational semantics of TEAL through syntax-directed inference rules.

Branching and subroutines

Clauses are logical formulas of the form $\sigma \xrightarrow{J} \sigma'$, where σ is the input state configuration, J is an instruction, and σ' is the output state configuration. Clauses for branch instructions have the form $\langle \ell, \sigma\rangle \xrightarrow{J} \langle \ell', \sigma'\rangle$, manipulating labeled state configurations to handle instructions that modify the program counter. As in Move, the program counter is represented by the label ℓ. Rule (A-Step) serves as a meta-rule, allowing the derivation of all rules that operate on unlabeled states, that is, instructions that do not affect the program counter. Rules that involve branching set the program counter to the new program location ℓ'. In particular, rule (A-Callsub) behaves as a branch to location ℓ', while additionally pushing onto the call stack C the return program point $\ell + 1$. Symmetrically, rule (A-Retsub) restores the program counter and pops from the call stack the tuple containing the return address.

$$\text{A-STEP}\quad \frac{\sigma \xrightarrow{J} \sigma'}{\langle \ell, \sigma \rangle \xrightarrow{J} \langle \ell + 1, \sigma' \rangle}$$

$$\text{A-BRANCH}\quad \frac{}{\langle \ell, \sigma \rangle \xrightarrow{\textbf{b}\ \langle \ell' \rangle} \langle \ell', \sigma \rangle}$$

$$\text{A-BRANCH-IF-ZERO}\quad \frac{n = 0}{\langle \ell, \langle C, n :: V, \Delta, \Omega, A, \Phi \rangle \rangle \xrightarrow{\textbf{bz}\ \langle \ell' \rangle} \langle \ell', \langle C, V, \Delta, \Omega, A, \Phi \rangle \rangle}$$

$$\text{A-BRANCH-IF-NOT-ZERO}\quad \frac{n \neq 0}{\langle \ell, \langle C, n :: V, \Delta, \Omega, A, \Phi \rangle \rangle \xrightarrow{\textbf{bnz}\ \langle \ell' \rangle} \langle \ell', \langle C, V, \Delta, \Omega, A, \Phi \rangle \rangle}$$

$$\text{A-CALLSUB}\quad \frac{}{\langle \ell, \langle C, V, \Delta, \Omega, A, \Phi \rangle \rangle \xrightarrow{\textbf{callsub}\ \langle \ell' \rangle} \langle \ell', \langle \ell + 1 :: C, V, \Delta, \Omega, A, \Phi \rangle \rangle}$$

$$\text{A-RETSUB}\quad \frac{C = \langle v_1, .., v_n \rangle :: \ell' :: C'}{\langle \ell, \langle C, V, \Delta, \Omega, A, \Phi \rangle \rangle \xrightarrow{\textbf{retsub}} \langle \ell', \langle C', V, \Delta, \Omega, A, \Phi \rangle \rangle}$$

Call stack instructions

The asymmetric behavior between rules (A-CALLSUB) and (A-RETSUB) above stems from the way TEAL manages the call stack frame via the `proto` instruction, which must occur as the first instruction of any subroutine. Rule (A-PROTO) completes the setup of the call stack: n input arguments are popped from the operand stack V and packed into a tuple, which is then pushed onto the call stack C. This mechanism allows all arguments to be stored into a single stack slot, simplifying their removal by rule (A-RETSUB). Rules (A-FRAMEBURY-) and (A-FRAMEDIG-) replicate the exact behavior of the TEAL interpreter implementation.[5] In contrast to what is described in the official Algorand documentation [39], these instructions exhibit two distinct behaviors: when given a negative offset, they access the stack frame (i.e., subroutine arguments); when given a positive offset, they access the general operand stack. This requires introducing two pairs of semantic rules to distinguish accesses to the frame and to the operand stack, respectively.

$$\text{A-PROTO}\quad \frac{V = v_1 :: .. :: v_n :: V' \qquad C' = \langle v_1, .., v_n \rangle :: C}{\langle C, V, \Delta, \Omega, A, \Phi \rangle \xrightarrow{\textbf{proto}\ \langle n,m \rangle} \langle C', V', \Delta, \Omega, A, \Phi \rangle}$$

$$\text{A-FRAMEBURY-ARG}\quad \frac{C = \langle v_1 :: .. :: v_n :: .. :: v_m \rangle :: C' \qquad C'' = \langle v_1 :: .. :: v :: .. :: v_m \rangle :: C' \qquad 1 \leq n \leq m}{\langle C, v :: V, \Delta, \Omega, A, \Phi \rangle \xrightarrow{\textbf{frame_bury}\ \langle -n \rangle} \langle C'', V, \Delta, \Omega, A, \Phi \rangle}$$

$$\text{A-FRAMEBURY-LOC}\quad \frac{V = v_0 :: v_1 :: .. :: v_n :: V' \qquad V'' = v_0 :: v :: .. :: v_n :: V' \qquad n \geq 0}{\langle C, v :: V, \Delta, \Omega, A, \Phi \rangle \xrightarrow{\textbf{frame_bury}\ \langle n \rangle} \langle C, v_n :: V, \Delta, \Omega, A, \Phi \rangle}$$

$$\text{A-FRAMEDIG-ARG}\quad \frac{C = \langle v_1 :: .. :: v_n :: .. :: v_m \rangle :: C' \qquad 1 \leq n \leq m}{\langle C, V, \Delta, \Omega, A, \Phi \rangle \xrightarrow{\textbf{frame_dig}\ \langle -n \rangle} \langle C, v_n :: V, \Delta, \Omega, A, \Phi \rangle}$$

$$\text{A-FRAMEDIG-LOC}\quad \frac{V = v_0 :: v_1 :: .. :: v_n :: V' \qquad n \geq 0}{\langle C, V, \Delta, \Omega, A, \Phi \rangle \xrightarrow{\textbf{frame_dig}\ \langle n \rangle} \langle C, v_n :: V, \Delta, \Omega, A, \Phi \rangle}$$

Basic TEAL instructions

The semantic rules for TEAL basic instructions, including those for accessing the scratch space and manipulating the operand stack, are presented below. Rules (A-LOAD) and (A-STORE) are straightforward, while (A-LOADS) is a variant of (A-LOAD) that obtains the slot number from the stack rather than as an immediate argument. Rule (A-DUPN) replicates the top element of the stack n times, where n is given as an immediate. Rule (A-DUP2) duplicates both the top of the stack and the element immediately beneath it, once each. Rule (A-SWAP) swaps the top element of the stack with the one below it. Rule (A-COVER) pushes the top of the stack down by n positions, whereas rule (A-UNCOVER) lifts a buried value to the top of the stack by reversing this operation.

[5] The TEAL interpreter official implementation of the stack frame instructions is available at: `https://github.com/algorand/go-algorand/blob/master/data/transactions/logic/frames.go`

A-Load

$$\frac{\Delta(n) = \upsilon}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{load}\ \langle n\rangle} \langle C, \upsilon :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-Loads

$$\frac{\Delta(n) = \upsilon}{\langle C, n :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{loads}} \langle C, \upsilon :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-Store

$$\frac{}{\langle C, \upsilon :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{store}\ \langle n\rangle} \langle C, V, \Delta[n \mapsto \upsilon], \Omega, A, \Phi\rangle}$$

A-Pop

$$\frac{}{\langle C, \upsilon :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{pop}} \langle C, V, \Delta, \Omega, A, \Phi\rangle}$$

A-Dupn

$$\frac{}{\langle C, \upsilon :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{dupn}\ \langle n\rangle} \langle C, \underbrace{\upsilon :: .. :: \upsilon}_{n \text{ times}} :: \upsilon :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-Dup2

$$\frac{}{\langle C, \upsilon_1 :: \upsilon_2 :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{dup2}} \langle C, \upsilon_1 :: \upsilon_2 :: \upsilon_1 :: \upsilon_2 :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-Swap

$$\frac{}{\langle C, \upsilon_1 :: \upsilon_2 :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{swap}} \langle C, \upsilon_2 :: \upsilon_1 :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-Cover

$$\frac{V = \upsilon :: \upsilon_1 :: .. :: \upsilon_n :: V'}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{cover}\ \langle n\rangle} \langle C, \upsilon_1 :: .. :: \upsilon_n :: \upsilon :: V', \Delta, \Omega, A, \Phi\rangle}$$

A-Uncover

$$\frac{V = \upsilon_1 :: .. :: \upsilon_n :: \upsilon :: V'}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{uncover}\ \langle n\rangle} \langle C, \upsilon :: \upsilon_1 :: .. :: \upsilon_n :: V', \Delta, \Omega, A, \Phi\rangle}$$

Datatype conversion, manipulation, and concatenation

The semantic rules for instructions handling datatype conversion, manipulation, and concatenation are grouped here. Rule (A-Extract) extracts the byte interval $[n, n+m]$ from the byte array on top of the stack; when the immediate $m = 0$, rule (A-Extract-0) applies, setting the upper bound of the interval to the end of the array of length k. Rule (A-Extract3) behaves like (A-Extract) but takes all arguments from the stack. Rule (A-Replace2) overwrites the first argument byte array υ by blitting the second argument υ' starting at position n. Rules (A-PushBytes) and (A-PushInt) respectively push a byte array and a 64-bit unsigned integer constant onto the stack. Rules (A-Btoi) and (A-Itob) convert between an 8-byte array and a 64-bit unsigned integer, in each direction. Finally, rule (A-Concat) pops two byte arrays from the stack and pushes their concatenation.

A-Extract
$$\frac{\upsilon = [b_1,\ ..\ , b_k] \qquad n < k \qquad n + m < k \qquad \upsilon' = [b_n,\ ..\ , b_{n+m}]}{\langle C, \upsilon :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{extract}\ \langle n,m\rangle} \langle C, \upsilon' :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-Extract-0
$$\frac{\upsilon = [b_1,\ ..\ , b_k] \qquad n < k \qquad \upsilon' = [b_n,\ ..\ , b_k]}{\langle C, \upsilon :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{extract}\ \langle n,0\rangle} \langle C, \upsilon' :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-Extract3
$$\frac{\upsilon = [b_1,\ ..\ , b_k] \qquad n < k \qquad n + m < k \qquad \upsilon' = [b_n,\ ..\ , b_{n+m}]}{\langle C, \upsilon :: n :: m :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{extract3}} \langle C, \upsilon' :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-Replace2
$$\frac{\upsilon = [b_1,\ ..\ , b_n,\ ..\ , b_m] \qquad \upsilon' = [b'_1,\ ..\ , b'_k] \qquad k \leq m \qquad \upsilon'' = [b_1,\ ..\ , b_n, b'_1,\ ..\ , b'_k,\ ..\ b_m]}{\langle C, \upsilon :: \upsilon' :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{replace2}\ \langle n\rangle} \langle C, \upsilon'' :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-PushBytes
$$\frac{\upsilon = [b_1,\ ..\ , b_n]}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{pushbytes}\ \langle b_1\ ..\ b_n\rangle} \langle C, \upsilon :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-PushInt
$$\frac{}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{pushint}\ \langle n\rangle} \langle C, n :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-Btoi
$$\frac{n = [b_8,\ ..\ , b_1]}{\langle C, [b_8,\ ..\ , b_1] :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{btoi}} \langle C, n :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-Itob
$$\frac{[b_8,\ ..\ , b_1] \equiv n}{\langle C, n :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{itob}} \langle C, [b_8,\ ..\ , b_1] :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-Concat
$$\frac{\upsilon = [b_1,\ ..\ , b_n] \qquad \upsilon' = [b'_1,\ ..\ , b'_m] \qquad \upsilon'' = [b_1,\ ..\ , b_n, b'_1,\ ..\ , b'_m]}{\langle C, \upsilon :: \upsilon' :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{concat}} \langle C, \upsilon'' :: V, \Delta, \Omega, A, \Phi\rangle}$$

Local state

The semantics of instructions operating on the Algorand local state are given below. Rule (A-AppLocalGet) has two variants depending on whether the given (a, k) pair exists in Ω: If the pair exists, (A-AppLocalGet) returns the associated value. Otherwise, (A-AppLocalGet-$\notin$) returns zero. Rule (A-AppLocalPut) inserts or updates a binding $\langle a, k\rangle \mapsto \upsilon$ in Ω, where a is a 32-byte array representing an address, and k is a byte array of arbitrary length.

A-AppLocalGet
$$\frac{(a, k) \in \texttt{dom}(\Omega) \qquad \upsilon = \Omega(a, k)}{\langle C, k :: a :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{app_local_get}} \langle C, \upsilon :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-AppLocalGet-$\notin$
$$\frac{(a, k) \notin \texttt{dom}(\Omega)}{\langle C, k :: a :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{app_local_get}} \langle C, 0 :: V, \Delta, \Omega, A, \Phi\rangle}$$

A-AppLocalPut
$$\frac{\upsilon = \Delta(n)}{\langle C, \upsilon :: k :: a :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{app_local_put}\ \langle n\rangle} \langle C, V, \Delta, \Omega[\langle a, k\rangle \mapsto \upsilon], A, \Phi\rangle}$$

A-AppLocalDel
$$\frac{}{\langle C, k :: a :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{app_local_del}} \langle C, V, \Delta, \Omega \setminus \langle a, k\rangle, A, \Phi\rangle}$$

Inner transactions

The final rules address opcodes that manipulate transactions. Rule (A-Itxn-Begin) initializes an inner transaction by clearing the current Φ component of the state tuple. Rule (A-Itxn-Field) sets a transaction field named x by adding a new binding into Φ. The (A-Itxn-Submit-*) family of rules commits the currently constructed transaction, and their behavior depends on the value of the `Type` field in the transaction. Rule (A-Itxn-Submit-Xfer) manages asset transfers by decrementing the balance of the sender a_1 and incrementing the balance of the receiver a_2 by the amount m of assets identified by asset-id n. Rule (A-Itxn-Submit-Create) handles asset creation by adding a new binding to A, which records the amount m of newly minted assets associated with address

a. The asset identifier n is generated by Algorand and treated as a fresh opaque number. Finally, rule (A-Itxn-CreatedAssetID) returns the asset identifier n of the most recently created asset.

$$\textsc{A-Itxn-Begin}\quad \frac{}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{itxn_begin}} \langle C, V, \Delta, \Omega, A, \varnothing\rangle}$$

$$\textsc{A-Itxn-Field}\quad \frac{}{\langle C, v :: V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{itxn_field}\ \langle x\rangle} \langle C, V, \Delta, \Omega, A, \Phi[x \mapsto v]\rangle}$$

$$\textsc{A-Itxn-Submit-Xfer}\quad \frac{\begin{array}{c}\Phi[\texttt{Type}] = \texttt{"axfer"} \qquad a = \Phi[\texttt{AssetSender}] \qquad n = \Phi[\texttt{XferAsset}] \\ a' = \Phi[\texttt{AssetReceiver}] \qquad m = \Phi[\texttt{AssetAmount}] \qquad n_1 = A(a,n) \qquad n_2 = A(a',n) \qquad n_1 \geq m\end{array}}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{itxn_submit}} \langle C, V, \Delta, \Omega, A[\langle a,n\rangle \mapsto n_1 - m][\langle a',n\rangle \mapsto n_2 + m], \varnothing\rangle}$$

$$\textsc{A-Itxn-Submit-Create}\quad \frac{\Phi[\texttt{Type}] = \texttt{"acfg"} \qquad a = \Phi[\texttt{Sender}] \qquad m = \Phi[\texttt{Total}] \qquad (n\ \texttt{fresh})}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{itxn_submit}} \langle C, V, \Delta, \Omega, A[\langle a,n\rangle \mapsto m], \varnothing\rangle}$$

$$\textsc{A-Itxn-CreatedAssetID}\quad \frac{A = A'[\langle a,n\rangle \mapsto m]}{\langle C, V, \Delta, \Omega, A, \Phi\rangle \xrightarrow{\textbf{itxn}\ \langle\texttt{CreatedAssetID}\rangle} \langle C, n :: V, \Delta, \Omega, A, \Phi\rangle}$$

C. Encoding

We introduce a translation operator $[\![\cdot]\!]_P$ that, given a Move instruction or fragment of code, produces its corresponding TEAL encoding.

Definition C.1 (Translation Operator)**.** *Let $[\![\cdot]\!]_P : (I \cup F \cup \varnothing) \times P \to Q$ be an operator, parametric over a program P, that maps either an instruction $I \in P$, a function definition $F \in P$, or the empty element $\varnothing$, into a corresponding block of TEAL instructions Q. When applied to the empty element, the entire program P is translated; in this case, we write $[\![P]\!]$ instead of $[\![\varnothing]\!]_P$.*

In Tables C.2 to C.5, we present the translation rules from Move to TEAL, grouped by category. Table C.2 collects rules that operate on entire programs and control-flow instructions. Rule (T-Program) specifies how a Move program P is translated by concatenating the translations of all its function definitions $F_1, \ldots, F_m$. Datatype definitions T are ignored, as they do not contribute any executable code. Since TEAL programs are flat sequences of instructions that start execution from the first line, we emit a `callsub` to the first function f_1 to simulate an initial function call, thus avoiding starting directly with a `proto`. The program terminates by returning 1 on the stack, signaling successful approval of the transaction. In practice, a real-world implementation would prepend a more sophisticated preamble, typically parsing the *ApplicationCall* transaction arguments used to invoke the contract from off-chain.

Rule (T-Fun) handles the translation of a single function definition by first emitting a `proto` instruction followed by a preamble of variable length. This is followed by a central part that concatenates the TEAL code generated for each instruction $I_1, \ldots, I_m$ within the function body. Finally, an epilogue is appended before the `retsub` instruction.

The preamble serves two main purposes:

- It saves on the stack the contents of scratch space slots 0 through M, where M is the highest local variable index used in the Move function;
- It retrieves n function arguments from the call stack and stores them into scratch space slots 0 through $n - 1$.

Since Algorand's scratch space is a flat set of 256 registers without scoping, we simulate the local variable scoping of the Move VM by employing a standard compiler technique: saving the registers that the callee may overwrite and restoring them afterward. Additionally, because Move function arguments occupy the initial local variable slots, replicating this convention in TEAL simplifies the mapping from local variable indices to scratch space slots.

The epilogue restores the saved caller slots and stores the result in the call stack through a `frame_bury` before returning. Rule (T-Call) emits a `callsub` to label ℓ_1, which marks the first instruction I_1 of the callee. Thanks to label preservation during translation, this and other jump-related rules remain straightforward. As illustrated by (T-Fun), the label ℓ_1 corresponds to the `proto` instruction—the entry point for subroutine calls in TEAL. The remaining rules in Table C.2 are self-explanatory.

$$
\text{(T-PROGRAM)} \quad [\![P]\!] = \left|\begin{array}{l} \textbf{callsub}\ \langle \ell_1 \rangle \\ \textbf{pushint}\ \langle 1 \rangle \\ \textbf{return} \\ [\![F_1]\!]_P \\ \vdots \\ [\![F_m]\!]_P \end{array}\right. \qquad \begin{array}{rcl} P & = & T_1..T_n\ F_1..F_m \\ F_1 & = & f_1(x_1 : \tau_1,\ ..\ , x_n : \tau_n)\{\ \ell_1 : I_1\ ..\ \} \end{array}
$$

$$
\text{(T-FUN)} \quad [\![F]\!]_P = \left|\begin{array}{l} \ell_1 : \textbf{proto}\ \langle n, 1 \rangle \\ \\ \textbf{load}\ \langle 0 \rangle \\ \vdots \\ \textbf{load}\ \langle M \rangle \\ \\ \textbf{frame_dig}\ \langle -1 \rangle \\ \textbf{store}\ \langle 0 \rangle \\ \vdots \\ \textbf{frame_dig}\ \langle -n \rangle \\ \textbf{store}\ \langle n-1 \rangle \\ \\ [\![I_1]\!]_P \\ \ell_2 : [\![I_2]\!]_P \\ \vdots \\ \ell_m : [\![I_m]\!]_P \\ \\ \textbf{frame_bury}\ \langle 0 \rangle \\ \textbf{store}\ \langle M \rangle \\ \vdots \\ \textbf{store}\ \langle 0 \rangle \\ \\ \textbf{retsub} \end{array}\right. \qquad \begin{array}{rcl} F & = & f(x_1 : \tau_1,\ ..\ , x_n : \tau_n) \\ & & \{\ \ell_1 : I_1\ ..\ \ell_m : I_m\ \} \\ M & = & \texttt{maxlocal}(I_1\ ..\ I_m) \end{array}
$$

$$
\begin{array}{lrcl} \text{(T-CALL)} & [\![\textbf{Call}\ \langle f \rangle]\!]_P & = & \textbf{callsub}\ \langle \ell_1 \rangle \qquad P(f) = f(x_1 : \tau_1\ ..\ x_n : \tau_n)\ \{\ \ell_1 : I_1\ ..\ \ell_m : I_m\ \} \\ \text{(T-RET)} & [\![\textbf{Ret}]\!]_P & = & \textbf{retsub} \\ \text{(T-BRANCH)} & [\![\textbf{Branch}\ \langle \ell \rangle]\!]_P & = & \textbf{b}\ \langle \ell \rangle \\ \text{(T-BRTRUE)} & [\![\textbf{BrTrue}\ \langle \ell \rangle]\!]_P & = & \textbf{bnz}\ \langle \ell \rangle \\ \text{(T-BRFALSE)} & [\![\textbf{BrFalse}\ \langle \ell \rangle]\!]_P & = & \textbf{bz}\ \langle \ell \rangle \end{array}
$$

Table C.2: Translation rules for Move programs and control-flow instructions into TEAL. Elements in blue correspond to Move constructs, while those in orange represent the target TEAL code. When a single Move instruction translates into multiple TEAL instructions, the resulting block is indicated by a vertical bar. Vertical dots denote the repeated application of one or more instructions over a range of immediate arguments. The function `maxlocal` computes the highest local variable index referenced within a given block of Move instructions.

Table C.3 presents the translation rules for basic Move instructions, including those that manipulate local variables and perform arithmetic operations. These rules are mostly straightforward. Notably, both `CopyLoc` and `MoveLoc` instructions are translated simply as `load` operations in TEAL. The meta opcode *Op* represents any unary or binary arithmetic or logical operator. Although TEAL operator names differ, the mapping is direct and intuitive. For example, `Add` translates to `+`, `Sub` to `-`, `Eq` to `==`, `Not` to `!`,

and so forth.

(T-CopyLoc)	$[\![$**CopyLoc** $\langle n \rangle]\!]_P$	=	**load** $\langle n \rangle$
(T-MoveLoc)	$[\![$**MoveLoc** $\langle n \rangle]\!]_P$	=	**load** $\langle n \rangle$
(T-StoreLoc)	$[\![$**StLoc** $\langle n \rangle]\!]_P$	=	**store** $\langle n \rangle$
(T-LoadConst)	$[\![$**LdConst** $\langle n \rangle]\!]_P$	=	**pushint** $\langle n \rangle$
(T-Pop)	$[\![$**Pop**$]\!]_P$	=	**pop**
(T-BinOp)	$[\![$***Op***$]\!]_P$	=	***Op***

Table C.3: Translation rules for basic Move instructions involving local variables, stack manipulation, and arithmetic operations. The *Op* pseudo-instruction denotes any unary or binary operator, such as addition, multiplication, or logical connectives. Its translation is straightforward, though operator names differ slightly across the two languages; for instance, the Move instruction `Add` corresponds to `+` in TEAL.

Table C.4 presents the translation rules for manipulating references, borrows, and struct packing/unpacking. The binary encoding of references is detailed in Section D, and formally introduced by Definition D.1. To construct a reference, the generated TEAL code must synthesize a binary string formatted according to the reference kind and its content. Rule (T-BorrowLoc) emits a 2-byte string representing a reference to a scratch space slot. Rules (T-ReadRef) and (T-WriteRef) simply call corresponding *runtime library* subroutines of AlgoMove deployed on Algorand; the full TEAL implementations of these subroutines are provided in our prototype implementation. Rules (T-BorrowField) and (T-BorrowField-S) handle field selection within struct datatypes. The former applies when the field type x is not a struct, while the latter applies when it is. Both extend the reference at the top of the stack by concatenating an additional offset-length pair (each a 16-bit word) as a byte array. Our reference format encodes the path as a variable-length sequence of offset-length pairs appended at the end of the binary string. This design allows easy extension of the reference path by appending a new 32-bit pair whenever a `BorrowField` instruction is processed. A subtle difference between the two rules is that (T-BorrowField) sets the 15th bit of the offset (the highest 16-bit word) to 1. This bit acts as a flag triggering special serialization/deserialization for the last field when decoding the reference. Rule (T-Pack) emits code to serialize a struct field-by-field in the original field order. The meta-instruction ***itob?nop*** $\langle \tau \rangle$ emits either an **itob** or a **nop** depending on whether the field's type τ is non-struct or struct, respectively. After this, the byte arrays are concatenated n times. Conversely, rule (T-Unpack) deserializes a packed struct from the top of the stack by splitting it into n chunks, one per field. Fields of non-struct types are further processed with a `btoi` instruction to convert from bytes to integer; this is controlled by the meta-instruction ***btoi?nop*** $\langle \tau \rangle$, which emits **btoi** for non-structs and **nop** otherwise.

In Table C.5, instructions for accessing the global storage are translated. These correspond to the characteristic Move primitives. Rule (T-MoveTo) emits an `app_local_put` instruction, taking the top of the stack as argument (i.e., the data to be stored), and using the 1-byte representation of the type name s as key. In Move, only struct datatypes can be stored globally; therefore, whatever precedes the TEAL instructions emitted by this rule is guaranteed to be a serialized byte array that satisfies the type constraints enforced by `app_local_put`. Rule (T-MoveFrom) reproduces the original semantics of (M-MoveFrom) by deleting the entry from the Algorand local state immediately after the `app_local_get`. This behavior cannot lead to error states in TEAL, such as accessing a deleted key, since the Move type system and borrow checker statically prevent such scenarios. Rule (T-Exists) uses the `app_local_get_ex` instruction to check whether a given key exists in the local state. In TEAL, this requires an (address, application_id, key) triple on the operand stack; the `txn` instruction is used to obtain the current application ID[6]. Since in Move the `exists` primitive can only access the global storage of the current application, the translated TEAL code similarly restricts access to the local state of the current application. No foreign application is involved, and thus no opt-in check is required. Finally, rule (T-BorrowGlobal) operates similarly to (T-BorrowLoc) in Table C.4, but synthesizes a reference of kind 1.

In Table C.6, virtual instructions representing the Asset layer of the AlgoMove library are presented. These are treated as native primitives, each translating into a custom block of TEAL code, although, as already anticipated, an actual implementation would detect a `Call` instruction to the native functions defined by the library. For instance, a **Deposit** $\langle s \rangle$ would appear in the disassembled bytecode as a **Call** ⟨`deposit<`s`>(address, Asset)`⟩. Rule (T-Deposit) sets up an inner transaction of type `axfer`

[6]We do not provide a rule for the semantics of the `txn` instruction in TEAL, as our translation system only uses it once with the `ApplicationID` field, whose behavior is trivial.

$$(\text{T-BorrowLoc}) \quad [\![\textbf{BorrowLoc}\ \langle n \rangle]\!]_P = \textbf{pushbytes}\ \langle \texttt{0x00}\ n \rangle$$

$$(\text{T-ReadRef}) \quad [\![\textbf{ReadRef}]\!]_P = \textbf{callsub}\ \langle \texttt{read_ref} \rangle$$

$$(\text{T-WriteRef}) \quad [\![\textbf{WriteRef}]\!]_P = \textbf{callsub}\ \langle \texttt{write_ref} \rangle$$

$$(\text{T-BorrowField}) \quad [\![\textbf{BorrowField}\ \langle s.x \rangle]\!]_P = \left|\ \begin{array}{l} \textbf{pushbytes}\ \langle d\ l \rangle \\ \textbf{concat} \end{array}\right. \qquad \begin{array}{l} P(s) = \texttt{struct}\ s\ \{\ ..\ x : \tau\ ..\ \} \\ \tau \neq \texttt{struct} \\ d = \texttt{off}(s.x)\ \|\ \texttt{0x8000} \\ l = \texttt{len}(s.x) \end{array}$$

$$(\text{T-BorrowField-S}) \quad [\![\textbf{BorrowField}\ \langle s.x \rangle]\!]_P = \left|\ \begin{array}{l} \textbf{pushbytes}\ \langle d\ l \rangle \\ \textbf{concat} \end{array}\right. \qquad \begin{array}{l} P(s) = \texttt{struct}\ s\ \{\ ..\ x : \tau\ ..\ \} \\ \tau = \texttt{struct} \\ d = \texttt{off}(s.x) \\ l = \texttt{len}(s.x) \end{array}$$

$$(\text{T-Pack}) \quad [\![\textbf{Pack}\ \langle s \rangle]\!]_P = \left|\ \begin{array}{l} \textbf{uncover}\ \langle n - 1 \rangle \\ \textit{\textbf{itob?nop}}\ \langle \tau_1 \rangle \\ \vdots \\ \textbf{uncover}\ \langle n - 1 \rangle \\ \textit{\textbf{itob?nop}}\ \langle \tau_n \rangle \\ \\ \left.\begin{array}{l} \textbf{concat} \\ \vdots \\ \textbf{concat} \end{array}\right\}\ n\ \texttt{times} \end{array}\right. \qquad P(s) = \texttt{struct}\ s\ \{\ x_1 : \tau_1\ ..\ x_n : \tau_n\ \}$$

$$(\text{T-Unpack}) \quad [\![\textbf{Unpack}\ \langle s \rangle]\!]_P = \left|\ \begin{array}{l} \textbf{dup} \\ \textbf{extract}\ \langle d_1, l_1 \rangle \\ \textit{\textbf{btoi?nop}}\ \langle \tau_1 \rangle \\ \textbf{swap} \\ \vdots \\ \textbf{dup} \\ \textbf{extract}\ \langle d_n, l_n \rangle \\ \textit{\textbf{btoi?nop}}\ \langle \tau_n \rangle \\ \textbf{swap} \\ \\ \textbf{pop} \end{array}\right. \qquad \begin{array}{l} P(s) = \texttt{struct}\ s\ \{\ x_1 : \tau_1\ ..\ x_n : \tau_n\ \} \\ \left.\begin{array}{l} d_i = \texttt{off}(s.x_i) \\ l_i = \texttt{len}(s.x_i) \end{array}\right.\ \texttt{with}\ i \in [1, n] \end{array}$$

Table C.4: Translation rules for Move instructions dealing with references and structs. The meta instructions ***itob?nop*** and ***btoi?nop*** appearing in the rules for `Pack` and `Unpack` indicate that an **itob** or **btoi** instruction must be emitted when the corresponding type is a non-struct; otherwise, a **nop** instruction is generated. The large curly brace notation specifies the number of `concat` instructions to emit, with n denoting the number of fields in the struct s

(asset transfer) and populates the relevant transaction fields. Since the second argument to the `deposit()` function is a structured datatype `Asset`, an unpacking step is required to extract its fields, that is, the asset-id, the amount, and the owner. The type argument s is unused in the translation, as it represents the phantom type argument of the generic type `Asset` and the `Unpack` instruction takes into account only the raw type without any type argument, hence `Asset` alone. Rule (T-Withdraw) performs the inverse operation, constructing an `Asset` from these same fields. The asset-id, however, is not directly available and must be retrieved via an auxiliary `retrieve_id_by_name` subroutine, which we omit for brevity. Given the type name s converted into a byte string, the subroutine queries Algorand's asset list, accessible through the contract's opted-in assets, eventually returning the

Rule			
(T-MoveTo)	$[\![\textbf{MoveTo}\ \langle s \rangle]\!]_P$	=	**pushbytes** $\langle \lfloor\!\lfloor s \rfloor\!\rfloor_1 \rangle$ **swap** **app_local_put**
(T-MoveFrom)	$[\![\textbf{MoveFrom}\ \langle s \rangle]\!]_P$	=	**pushbytes** $\langle \lfloor\!\lfloor s \rfloor\!\rfloor_1 \rangle$ **dup2** **app_local_get** **cover** $\langle 2 \rangle$ **app_local_del**
(T-Exists)	$[\![\textbf{Exists}\ \langle s \rangle]\!]_P$	=	**txn** $\langle$`ApplicationID`$\rangle$ **pushbytes** $\langle \lfloor\!\lfloor s \rfloor\!\rfloor_1 \rangle$ **app_local_get_ex** **uncover** $\langle 1 \rangle$ **pop**
(T-BorrowGlobal)	$[\![\textbf{BorrowGlobal}\ \langle s \rangle]\!]_P$	=	**pushbytes** $\langle$`0x01` $\lfloor\!\lfloor s \rfloor\!\rfloor_1 \rangle$ **swap** **concat**

Table C.5: Translation rules for Move instructions dealing with global storage. These instructions correspond to the characteristic Move primitives for resource management. The Algorand *local state* is used as the counterpart of the Move global storage. The notation $\lfloor\!\lfloor s \rfloor\!\rfloor_1$ denotes the 1-byte representation of the struct type name s.

corresponding ID. Rule (T-Create) sets up an inner transaction of type `acfg` (asset configuration) that mints a fresh amount of assets. The newly created asset is assigned the name s to allow for subsequent retrieval via the `retrieve_id_by_name` subroutine mentioned above. After the inner transaction is submitted, the freshly generated asset-id produced by the platform is queried and packed into the returned struct.

D. Properties of the Encoding

In this section, we introduce the formal framework underpinning the operational correspondence result stated in Theorem 6.1. This theorem asserts that the state transitions of any AlgoMove program are faithfully reflected by the corresponding TEAL state transitions of its encoded counterpart.

We introduce the notation $\lfloor\!\lfloor n \rfloor\!\rfloor$, where n is a natural number, to denote the 8-byte serialization of a 64-bit unsigned integer in big-endian format. When a subscript is present, it constrains the length (in bytes) of the serialization; for example, $\lfloor\!\lfloor n \rfloor\!\rfloor_1$ denotes the 1-byte serialization of an unsigned integer n. In the absence of a subscript, the serialization length depends on the value being encoded. Addresses in TEAL are represented as byte arrays, hence $\lfloor\!\lfloor a \rfloor\!\rfloor_{32}$ denotes the 32-byte serialization of an address. Strings are also treated as byte arrays; thus, $\lfloor\!\lfloor s \rfloor\!\rfloor$ represents the variable-length serialization of a string s, which may also denote a struct type name. Serializing field paths, written as $\lfloor\!\lfloor p \rfloor\!\rfloor$, yields a variable-length byte array composed of $(\mathit{offset}, \mathit{length})$ pairs, one for each field in the path p. Each offset and length is encoded as a 2-byte value, resulting in a total of 4 bytes per field. If $p = []$, the serialization yields an empty byte array. The operator $+\!\!+$ denotes byte-array concatenation: given two byte-array operands, it produces a new byte array representing their sequential composition.

Definition D.1 (Value Encoding)**.** *Given a Move value v, its encoding $[\![v]\!]_{L,G}$ is a TEAL value υ, defined by a case analysis on the*

$$
\text{(T-DEPOSIT)} \qquad [\![\textbf{Deposit}\ \langle s \rangle]\!]_P \;=\; \left|\begin{array}{l}
\textbf{swap} \\
[\![\textbf{Unpack}\ \langle \texttt{Asset} \rangle]\!]_P \\
\textbf{itxn_begin} \\
\textbf{pushbytes}\ \langle \texttt{"axfer"} \rangle \\
\textbf{itxn_field}\ \langle \texttt{Type} \rangle \\
\textbf{itxn_field}\ \langle \texttt{XferAsset} \rangle \\
\textbf{itxn_field}\ \langle \texttt{AssetAmount} \rangle \\
\textbf{itxn_field}\ \langle \texttt{Sender} \rangle \\
\textbf{itxn_field}\ \langle \texttt{AssetReceiver} \rangle \\
\textbf{itxn_submit}
\end{array}\right.
$$

$$
\text{(T-WITHDRAW)} \qquad [\![\textbf{Withdraw}\ \langle s \rangle]\!]_P \;=\; \left|\begin{array}{l}
\textbf{itxn_begin} \\
\textbf{itxn_field}\ \langle \texttt{Sender} \rangle \\
\textbf{pushbytes}\ \langle \texttt{"axfer"} \rangle \\
\textbf{itxn_field}\ \langle \texttt{Type} \rangle \\
\textbf{pushbytes}\ \langle \lfloor\!\lfloor s \rfloor\!\rfloor \rangle \\
\textbf{callsub}\ \langle \texttt{retrieve_id_by_name} \rangle \\
\textbf{dup2} \\
\textbf{itxn_field}\ \langle \texttt{XferAsset} \rangle \\
\textbf{itxn_field}\ \langle \texttt{AssetAmount} \rangle \\
\textbf{global}\ \langle \texttt{CurrentApplicationAddress} \rangle \\
\textbf{itxn_field}\ \langle \texttt{AssetReceiver} \rangle \\
\textbf{itxn_submit} \\
\textbf{global}\ \langle \texttt{CurrentApplicationAddress} \rangle \\
\textbf{cover}\ \langle 2 \rangle \\
[\![\textbf{Pack}\ \langle \texttt{Asset} \rangle]\!]_P
\end{array}\right.
$$

$$
\text{(T-CREATE)} \qquad [\![\textbf{Create}\ \langle s \rangle]\!]_P \;=\; \left|\begin{array}{l}
\textbf{pop} \\
\textbf{global}\ \langle \texttt{CurrentApplicationAddress} \rangle \\
\textbf{dup2} \\
\textbf{itxn_begin} \\
\textbf{pushbytes}\ \langle \texttt{"acfg"} \rangle \\
\textbf{itxn_field}\ \langle \texttt{Type} \rangle \\
\textbf{itxn_field}\ \langle \texttt{Sender} \rangle \\
\textbf{itxn_field}\ \langle \texttt{Total} \rangle \\
\textbf{pushbytes}\ \langle \lfloor\!\lfloor s \rfloor\!\rfloor \rangle \\
\textbf{itxn_field}\ \langle \texttt{Name} \rangle \\
\textbf{itxn_submit} \\
\textbf{swap} \\
\textbf{itxn}\ \langle \texttt{CreatedAssetID} \rangle \\
[\![\textbf{Pack}\ \langle \texttt{Asset} \rangle]\!]_P
\end{array}\right.
$$

Table C.6: Translation rules for Move instructions dealing with the Asset layer. Calls to the most relevant functions dealing with asset management are included here. The $\lfloor\!\lfloor s \rfloor\!\rfloor$ notation stands for the string representation of the type name s.

structure of v, where L denotes the local variable store and G the global storage:

$$
\begin{array}{rcll}
[\![n]\!]_{L,G} & = & n & \\
[\![\texttt{true}]\!]_{L,G} & = & 1 & \\
[\![\texttt{false}]\!]_{L,G} & = & 0 & \\
[\![a]\!]_{L,G} & = & \lfloor\!\lfloor a \rfloor\!\rfloor_{32} & \\
[\![\{\ (x_1, v_1) \mathinner{..} (x_n, v_n)\ \}]\!]_{L,G} & = & \lfloor\!\lfloor [\![v_1]\!]_{L,G} \rfloor\!\rfloor \mathbin{+\!\!+} \mathinner{..} \mathbin{+\!\!+} \lfloor\!\lfloor [\![v_n]\!]_{L,G} \rfloor\!\rfloor & \\
[\![\texttt{ref}\ \langle c, p \rangle]\!]_{L,G} & = & \lfloor\!\lfloor 0 \rfloor\!\rfloor_1 \mathbin{+\!\!+} \lfloor\!\lfloor n \rfloor\!\rfloor_1 \mathbin{+\!\!+} \lfloor\!\lfloor p \rfloor\!\rfloor & (L(n) = c) \\
[\![\texttt{ref}\ \langle c, p \rangle]\!]_{L,G} & = & \lfloor\!\lfloor 1 \rfloor\!\rfloor_1 \mathbin{+\!\!+} \lfloor\!\lfloor s \rfloor\!\rfloor_1 \mathbin{+\!\!+} \lfloor\!\lfloor a \rfloor\!\rfloor_{32} \mathbin{+\!\!+} \lfloor\!\lfloor p \rfloor\!\rfloor & (G(a, s) = c)
\end{array}
$$

The treatment of references breaks syntax-directedness: in practice, a concrete implementation should introduce a discriminating tag to distinguish between local and global references. However, to remain faithful to the formalization in [36], we adopt a unified representation of references and rely on existential checks to determine their kind. This simplification is sound, as local and global memory locations are disjoint by definition; formally, $\texttt{codom}(L) \cap \texttt{codom}(G) = \varnothing$. For brevity, the parameters L and G may be omitted when their values are clear from context.

In the final case of the encoding function, we encounter a byte representation that is not strictly tied to a value: $\lfloor\!\lfloor s \rfloor\!\rfloor_1$ denotes the serialization of the user-defined type name s into a single byte. This byte corresponds to the ordinal position of the struct definition for s among the type declarations $T_1, \ldots, T_n$ in the program P. More formally, if $T_i \equiv \texttt{struct}\ s \ldots$ for some $i \in [1, n]$, then $\lfloor\!\lfloor s \rfloor\!\rfloor_1 = i$.

Multiple kinds of stack structures are involved in the proposed translation system. In Move, a single unified stack is used to store all relevant data, including operands, call arguments, and program points (also referred to as labels). In contrast, TEAL distinguishes between two separate stacks: a *call stack*, which holds call arguments and program points, and an *operand stack*, dedicated to computation operands. Accordingly, our translation system must coherently encode the contents of the Move stack into the appropriate components of the TEAL execution environment, ensuring a faithful correspondence between the two operational models.

Definition D.2 (Stack Encoding). *Let S be a Move stack. Its encoding, denoted by $[\![S]\!]_{M,L,G} = C, V$, converts S into a pair of TEAL stacks: the call stack C and the operand stack V. This encoding is defined relative to a memory M, a local variable environment L, and a global storage G. The construction of $[\![S]\!]_{M,L,G}$ proceeds inductively as follows:*

$$
\begin{array}{rcll}
[\![[]]\!]_{M,L,G} & = & [], [] & \\
[\![\ell :: S]\!]_{M,L,G} & = & \ell :: C, V & \texttt{where}\ \ [\![S]\!]_{M,L,G} = C, V \\
[\![v :: S]\!]_{M,L,G} & = & C, [\![v]\!]_{L,G} :: V & \texttt{where}\ \ [\![S]\!]_{M,L,G} = C, V \\
[\![L :: S]\!]_{M,L,G} & = & \langle [\![v_0]\!]_{L,G}, .., [\![v_{n-1}]\!]_{L,G} \rangle :: C, V & \texttt{where}\ \ [\![S]\!]_{M,L,G} = C, V \\
 & & & \quad\ \wedge\ \ \forall i \in \mathit{dom}(L).\ v_i = M(L(i)) \\
 & & & \quad\ \wedge\ \ |\mathit{dom}(L)| = n
\end{array}
$$

The first case represents the base cases of the recursion. The latter cases are recursive cases, where the result of the recursion is bound in the side condition on the right. The last case converts the map of Move local variables L into a tuple of encoded values that fits in a single slot of the TEAL call stack. From the above definition, it follows directly that if $[\![S]\!]_{M,L,G} = C, V$, then $[\![\ell :: v :: S]\!]_{M,L,G} = \ell :: C, [\![v]\!]_{L,G} :: V$. The relation holds even if we swap ℓ and v in the left-hand. This implies that $C \supseteq \{\ \ell \in S\ \}$ and $V = \{\ [\![v]\!]_{L,G} \mid v \in S\ \}$.

We now formalize the translation of state configurations from AlgoMove to TEAL. This definition plays a central role in our system, as it underpins the operational correspondence result stated in Theorem D.1.

Definition D.3 (State Configuration Encoding). *Let the Move state be $\mu = \langle M, L, G, S, B \rangle$. Then, its encoding is defined as $[\![\mu]\!] = \langle C, V, \Delta, \Omega, A, \Phi \rangle$, where:*

- $C, V = [\![S]\!]_{M,L,G}$
- $\Delta = \{\ n \mapsto [\![M(L(n))]\!]_{L,G}) \mid n \in \mathit{dom}(L)\ \}$
- $\Omega = \{\ \langle \lfloor\!\lfloor a \rfloor\!\rfloor_{32}, \lfloor\!\lfloor s \rfloor\!\rfloor_1 \rangle \mapsto [\![M(G(a, s))]\!]_{L,G} \mid (a, s) \in \mathit{dom}(G)\ \}$
- $A = \{\ \langle \lfloor\!\lfloor a \rfloor\!\rfloor_{32}, \lfloor\!\lfloor s \rfloor\!\rfloor_8 \rangle \mapsto B(a, s)\ \}$

The last component, Φ, is set to $\varnothing$ since it has no counterpart in Move and is irrelevant for the encoding.

The second bullet of the above definition essentially states that $\texttt{dom}(L) = \texttt{dom}(\Delta)$, meaning that local variables in Move and scratch space slots in Algorand share the same indexes within any given context. This correspondence directly follows from the translation rules (T-CopyLoc), (T-MoveLoc), and (T-StLoc), which do not modify the numeric parameter n when emitting a **load** or **store** instruction. In other words, variable indexes are preserved by the translation.

The following theorem establishes that the behavior of a Move instruction is preserved by the translation, as the corresponding TEAL code reproduces the same effect on the resulting state configuration. Since a single Move instruction I may be translated into a sequence of TEAL instructions Q, the resulting TEAL state is obtained through a chained derivation involving multiple steps.

Theorem D.1 (Operational Correspondence of Instructions). *Let P be a Move program with $I \in P$ and $Q = [\![I]\!]_P$ with $Q = J_1\ ..\ J_n$ for $n \geq 1$. Given a state $\mu = \langle M, L, G, S, B \rangle$ such that $\mu \xrightarrow{I} \mu'$ and given $\sigma_1 = [\![\mu]\!]$, there exist n TEAL states σ_i such that $\sigma_i \xrightarrow{J_i} \sigma_{i+1}$ for $i \in [1, n]$, with $\sigma_{n+1} = [\![\mu']\!]$.*

Proof. The proof is provided only for a few relevant instructions.

When I = **CopyLoc** $\langle n\rangle$ the translated TEAL code Q consists in a single instruction J = **load** $\langle n\rangle$ according to rule (T-CopyLoc), with $n \geq 0$. Rule (M-CopyLoc) has an input state $\mu = \langle M, L, G, S, B\rangle$ and an output state $\mu' = \langle M, L, G, M(L(n)) :: S, B\rangle$. Rule (A-Load) exhibits an input state $\sigma = \langle C, V, \Delta, \Omega, A, \Phi\rangle$ and an output state $\sigma' = \langle C, \Delta(n) :: V, \Delta, \Omega, A, \Phi\rangle$. We want to prove that $[\![\mu']\!] = \sigma'$. By hypothesis we know that $[\![\mu]\!] = \sigma$ and from Definition D.3 comes that $[\![\langle M, L, G, v :: S, B\rangle]\!] = \langle C', V', \Delta', \Omega', A', \Phi'\rangle$ such that:

- since $M(L(n)) = v$, $[\![v :: S]\!]_{M,L,G} = C', \upsilon :: V'$, with $[\![S]\!]_{M,L,G} = C', V'$ is part of the hypothesis, then $C = C'$ and $V = \upsilon :: V'$ with $\upsilon = [\![M(L(n))]\!]_{L,G}$
- the scratch space, the local state, the asset balance and the transaction fields are left untouched, thus $\Delta = \Delta'$, $\Omega = \Omega'$, $A = A'$ and $\Phi = \Phi'$.

When I = **Call** $\langle f\rangle$ the translated TEAL code Q is a single instruction J = **callsub** $\langle \ell_1\rangle$ according to rule (T-Call), such that ℓ_1 is the first label in the translated body of function f. Rule (M-Call) deals with an input labeled state $\langle \ell, \mu\rangle$ and an output $\langle \ell_1, \mu'\rangle$, where $\mu = \langle M, L, G, v_1 :: .. :: v_n :: S, B\rangle$ and $\mu' = \langle M[c_1 \mapsto v_1]..[c_n \mapsto v_n], \langle c_1, .., c_n\rangle, G, \ell + 1 :: L :: S, B\rangle$. Rule (T-Fun) shows that the first instruction of a function body in TEAL is always a **proto** followed by a preamble whose size depends on the number of locals used M and the number of arguments n. By evaluating the preamble using rules (A-Callsub), (A-Proto), (A-Load), (A-FrameDig) and (A-Store) interleaved by (A-Step), we reach the state $\langle \ell_1 + k, \sigma'\rangle$ with k being the length of the preamble and $\sigma' = \langle \upsilon_1 :: .. :: \upsilon_n :: 0 :: \ell + 1 :: C, \upsilon'_1 :: .. :: \upsilon'_M :: V, \Delta[0 \mapsto \upsilon_1]..[n-1 \mapsto \upsilon_n], \Omega, A, \Phi\rangle$. The full derivation is omitted for brevity. We want to show that $[\![\mu']\!] = \sigma'$ and that the program point ℓ_1 in Move corresponds to $\ell_1 + k$ in TEAL. The latter is trivial and directly comes from rule (T-Fun): ℓ_1 marks the **proto** instruction at the beginning of the preamble, whereas $\ell_1 + k$ marks Q_1, which is the actual beginning of the translated function body. The former can be proved as follows. By hypothesis $[\![\mu]\!] = \langle C, \upsilon_1 :: .. :: \upsilon_n :: V, \Delta, \Omega, A, \Phi\rangle$ with $\upsilon_i = [\![v_i]\!]_{M,L,G}$ for all $i \in [1, n]$. From Definition D.3 comes that $[\![\mu']\!] = \langle C', V', \Delta', \Omega', A', \Phi'\rangle$ such that:

- $[\![v_1 :: .. :: v_n :: S]\!]_{M,L,G} = C, \upsilon_1 :: .. :: \upsilon_n :: V$ is part of the hypothesis, thus $[\![\ell + 1 :: L :: S]\!]_{M,L,G} = \upsilon_1 :: .. :: \upsilon_n :: 0 :: \ell + 1 :: C, \upsilon'_1 :: .. :: \upsilon'_M :: V$. Now, $\ell + 1$ appears in both hands and $\upsilon'_1..\upsilon'_M$ come from the chunk of **load** instructions in the preamble and are equivalent to the locals L in the caller scope, since $\forall i \in [1, M]$. $[\![M(L(i))]\!]_{M,L,G} = \upsilon'_i$. The remaining components are auxiliary and have no effect on execution, including the `0` value in the call stack, which serves merely as a placeholder for the function's return value.
- given μ', let $M' = M[c_1 \mapsto v_1]\ldots[c_n \mapsto v_n]$, $L' = \langle c_1, \ldots, c_n\rangle$, and $\Delta' = \Delta[0 \mapsto \upsilon_1]\ldots[n-1 \mapsto \upsilon_n]$. Then, for each $i \in \mathtt{dom}(L') = \mathtt{dom}(\Delta')$, we have $[\![M'(L'(i))]\!]_{M,L,G} = \Delta'(i)$, since all $\upsilon_i = [\![v_i]\!]_{M,L,G}$.
- the global storage G remains unchanged, thus $\Omega = \Omega'$. Similarly, the balance B and the transaction fields are unaffected, so $A = A'$ and $\Phi = \Phi'$.

When I = **Deposit** $\langle s\rangle$, the translated code $Q = J_1\ Q_2\ J_3\ J_4\ J_5\ J_6\ J_7\ J_8\ J_9\ J_{10}$ is defined by rule (T-Deposit), with J_1 = **swap**, Q_2 the sub-block produced by rule (T-Unpack), J_3 = **itxn_begin**, J_4 = **pushbytes** $\langle$`"axfer"`$\rangle$, J_5 = **itxn_field** $\langle$`Type`$\rangle$, J_6 = **itxn_field** $\langle$`XferAsset`$\rangle$, J_7 = **itxn_field** $\langle$`AssetAmount`$\rangle$, J_8 = **itxn_field** $\langle$`Sender`$\rangle$, J_9 = **itxn_field** $\langle$`AssetReceiver`$\rangle$, J_{10} = **itxn_submit**. Rule (M-Deposit) is applied to an unlabeled input state $\mu = \langle M, L, G, a :: v :: S, B\rangle$, where $v = \{\ (\mathtt{amount}, m)\ \}$ and $n = [\![s]\!]_8$. The corresponding output state is $\mu' = \langle M, L, G, S, B'\rangle$ with $B' = B[\langle a, s\rangle \mapsto B(a, s) + m]$. By deriving Q from $\sigma = [\![\mu]\!]$, we obtain the following execution trace[7]:

[7]For readability, we omit applications of rule (A-Step), which increment the program counter and are interleaved between each instruction.

$$\frac{\sigma = \langle C, \lfloor a \rfloor\!\rfloor_{32} :: [\![v]\!]_{L,G} :: V, \Delta, \Omega, A, \Phi \rangle \qquad \sigma_1 = \langle C, [\![v]\!]_{L,G} :: \lfloor a \rfloor\!\rfloor_{32} :: V, \Delta, \Omega, A, \Phi \rangle}{\sigma \xrightarrow{\textbf{swap}} \sigma_1}$$

$$\frac{\sigma_2 = \langle C, n :: m :: \lfloor \overline{a} \rfloor\!\rfloor_{32} :: \lfloor a \rfloor\!\rfloor_{32} :: V, \Delta, \Omega, A, \Phi \rangle}{\sigma_1 \xrightarrow{Q_2} \sigma_2}$$

$$\frac{\sigma_3 = \langle C, n :: m :: \lfloor \overline{a} \rfloor\!\rfloor_{32} :: \lfloor a \rfloor\!\rfloor_{32} :: V, \Delta, \Omega, A, \varnothing \rangle}{\sigma_2 \xrightarrow{\textbf{itxn_begin}} \sigma_3}$$

$$\frac{\sigma_4 = \langle C, \lfloor \texttt{"axfer"} \rfloor\!\rfloor :: n :: m :: \lfloor \overline{a} \rfloor\!\rfloor_{32} :: \lfloor a \rfloor\!\rfloor_{32} :: V, \Delta, \Omega, A, \varnothing \rangle}{\sigma_3 \xrightarrow{\textbf{pushbytes}\ \langle \texttt{"axfer"} \rangle} \sigma_4}$$

$$\frac{\sigma_5 = \langle C, n :: m :: \lfloor \overline{a} \rfloor\!\rfloor_{32} :: \lfloor a \rfloor\!\rfloor_{32} :: V, \Delta, \Omega, A, \varnothing[\texttt{Type} \mapsto \texttt{"axfer"}] \rangle}{\sigma_4 \xrightarrow{\textbf{itxn_field}\ \langle \texttt{Type} \rangle} \sigma_5}$$

$$\frac{\sigma_6 = \langle C, m :: \lfloor \overline{a} \rfloor\!\rfloor_{32} :: \lfloor a \rfloor\!\rfloor_{32} :: V, \Delta, \Omega, A, \varnothing[\texttt{Type} \mapsto \texttt{"axfer"}][\texttt{XferAsset} \mapsto n] \rangle}{\sigma_5 \xrightarrow{\textbf{itxn_field}\ \langle \texttt{XferAsset} \rangle} \sigma_6}$$

$$\frac{\sigma_7 = \langle C, \lfloor \overline{a} \rfloor\!\rfloor_{32} :: \lfloor a \rfloor\!\rfloor_{32} :: V, \Delta, \Omega, A, \varnothing[\texttt{Type} \mapsto \texttt{"axfer"}][\texttt{XferAsset} \mapsto n][\texttt{AssetAmount} \mapsto m] \rangle}{\sigma_6 \xrightarrow{\textbf{itxn_field}\ \langle \texttt{AssetAmount} \rangle} \sigma_7}$$

$$\frac{\sigma_8 = \langle C, \lfloor a \rfloor\!\rfloor_{32} :: V, \Delta, \Omega, A, \varnothing[\texttt{Type} \mapsto \texttt{"axfer"}][\texttt{XferAsset} \mapsto n][\texttt{AssetAmount} \mapsto m][\texttt{Sender} \mapsto \overline{a}] \rangle}{\sigma_7 \xrightarrow{\textbf{itxn_field}\ \langle \texttt{Sender} \rangle} \sigma_8}$$

$$\frac{\sigma_9 = \langle C, V, \Delta, \Omega, A, \varnothing[\texttt{Type} \mapsto \texttt{"axfer"}][\texttt{XferAsset} \mapsto n][\texttt{AssetAmount} \mapsto m][\texttt{Sender} \mapsto \overline{a}][\texttt{AssetReceiver} \mapsto a] \rangle}{\sigma_8 \xrightarrow{\textbf{itxn_field}\ \langle \texttt{AssetReceiver} \rangle} \sigma_9}$$

$$\frac{\sigma' = \langle C, V, \Delta, \Omega, A[\langle a, n \rangle \mapsto A(a, n) + m][\langle \overline{a}, n \rangle \mapsto A(\overline{a}, n) - m], \varnothing \rangle}{\sigma_9 \xrightarrow{\textbf{itxn_submit}} \sigma'}$$

We omit the full derivation of Q_2 for brevity, assuming it unpacks the fields of the record v onto the stack, as prescribed by rule (T-Unpack). To ease the reader, we also omit the encoding of addresses $\lfloor a \rfloor\!\rfloor_{32}$ on the Algorand side. Our goal is to prove that $[\![\mu']\!] = \sigma'$. From Definition D.3, we obtain the following:

- $[\![\mu']\!] = \langle C, V, \Delta, \Omega, A', \varnothing \rangle$, where $A' = A[\langle \overline{a}, n \rangle \mapsto A(\overline{a}, n) - m][\langle a, n \rangle \mapsto A(a, n) + m]$, where n is the asset-id bound to field `id` of the unpacked `Asset` struct, such that $n = \lfloor s \rfloor\!\rfloor_8$; and address $\overline{a}$ is the content of the `owner` field. The latter is granted to be set to the current application address by rules (T-Withdraw) and (T-Create), which are by design the only constructors of `Asset` (cf. Table C.6).
- The stacks remain unchanged between μ' and σ', hence $[\![S]\!]_{M,L,G} = C, V$. Similarly, Δ and Ω are preserved.
- The only affected component is the Move balance $B' \in \mu'$, and its Algorand counterpart $A' \in \sigma'$. Clearly, $[\![B[\langle a, s \rangle \mapsto B(a, s) + m]]\!] = \overline{A}[\langle a, n \rangle \mapsto A(a, n) + m]$, for some $\overline{A} = A[\langle \overline{a}, n \rangle \mapsto A(\overline{a}, n) - m]$ such that $[\![B]\!] = \overline{A}$. Since Move linear types forbids a deposit without a corresponding withdraw/create, then $[\langle \overline{a}, n \rangle \mapsto m] \in A$ so that $A(\overline{a}, n) - m = 0$, thus $\overline{A} = A$ and $[\![\mu']\!] = \sigma'$.

When $I = \textbf{MoveTo}\ \langle s \rangle$, the translated code is $Q = J_1\ J_2\ J_3$ with $J_1 = \textbf{pushbytes}$, $J_2 = \textbf{swap}$, $J_3 = \textbf{app_local_put}$, according to rule (T-MoveTo). Rule (M-MoveTo) operates over an unlabeled input state $\mu = \langle M, L, G, a :: v :: S, B \rangle$ producing an output state $\mu' = \langle M[c \mapsto v], L, G[\langle a, s \rangle \mapsto c], S, B \rangle$. By executing Q from the initial configuration $\sigma = [\![\mu]\!]$, we derive the following state transitions:

$$\frac{\sigma = \langle C, \lfloor a \rfloor\!\rfloor_{32} :: [\![v]\!]_{L,G} :: V, \Delta, \Omega, A, \Phi \rangle \qquad \sigma_1 = \langle C, \lfloor s \rfloor\!\rfloor_1 :: \lfloor a \rfloor\!\rfloor_{32} :: [\![v]\!]_{L,G} :: V, \Delta, \Omega, A, \Phi \rangle}{\sigma \xrightarrow{\textbf{pushbytes}} \sigma_1}$$

$$\frac{\sigma_2 = \langle C, \lfloor a \rfloor\!\rfloor_{32} :: \lfloor s \rfloor\!\rfloor_1 :: [\![v]\!]_{L,G} :: V, \Delta, \Omega, A, \Phi \rangle}{\sigma_1 \xrightarrow{\textbf{swap}} \sigma_2}$$

$$\frac{\sigma' = \langle C, V, \Delta, \Omega[\langle \lfloor a \rfloor\!\rfloor_{32}, \lfloor s \rfloor\!\rfloor_1 \rangle \mapsto [\![v]\!]_{L,G}], A, \Phi \rangle}{\sigma_2 \xrightarrow{\textbf{app_local_put}} \sigma'}$$

We want to prove that $[\![\mu']\!] = \sigma'$. From Definition D.3 we have that:

- $\llbracket\mu'\rrbracket = \langle C, V, \Delta, \Omega[\langle a, k\rangle \mapsto \upsilon], A, \Phi\rangle$, where $\upsilon = \llbracket v\rrbracket_{L,G}$ and v is the value from the original Move stack in state μ.
- the stacks are untouched in μ' and σ', therefore $\llbracket S\rrbracket_{M,L,G} = C, V$.
- L and B are untouched, so are Δ, A and Φ.
- The only affected component is the Move global storage G, whose update corresponds to an update in the Algorand local state Ω. In particular, the new binding in σ' is $\Omega[\langle\llbracket a\rrbracket_{32}, \llbracket s\rrbracket_1\rangle \mapsto \llbracket v\rrbracket_{L,G}]$ which is equivalent to $\Omega[\langle a, k\rangle \mapsto \upsilon]$ in σ_2, for $k = \llbracket s\rrbracket_1$ and $\upsilon = \llbracket v\rrbracket_{L,G}$. Since this is the only newly added binding and the rest of the state is preserved by hypothesis, all conditions in Definition D.3 are satisfied.

$\square$

We now establish the operational correspondence for entire programs, which follows as a consequence of Theorem D.1. Ignoring datatype definitions, which are not relevant in this context, a Move program P consists of a sequence of function definitions $F_1 \dots F_n$, any of which may serve as an entry point. Conversely, a TEAL program Q is a flat sequence of instructions executed sequentially from the first instruction. We assume the first function $F_1 = f_1() \{ \ell_1 : I_1 \dots \}$ takes no arguments and returns no value, making it suitable as an entry point. The execution of P is initiated by a synthetic **Call** $\langle f_1\rangle$ instruction that jumps to label ℓ_1, while on the TEAL side, execution of Q begins with a **callsub** $\langle \ell_1\rangle$ instruction that jumps to the first instruction of Q. The following result is a corollary of Theorem D.1.

Corollary D.2 (Operational Correspondence of Programs)**.** *Let P be a Move program such that $f_1() \{ \ell_1 : I_1 .. \} \in P$, and let $Q = \llbracket P\rrbracket$. Starting from an initial empty state $\mu = \langle\varnothing, \varnothing, \varnothing, []\rangle$, consider its encoding $\sigma = \llbracket\mu\rrbracket$, and a derivation $\langle\ell, \mu\rangle \xrightarrow{\textbf{Call}\ \langle f_1\rangle} \langle\ell + 1, \mu'\rangle$, where $\mu' = \langle M', \varnothing, G', [], B'\rangle$ and ℓ is a dummy program location. Then, the corresponding derivation on the TEAL side $\langle\ell, \sigma\rangle \xrightarrow{\textbf{callsub}\ \langle \ell_1\rangle} \langle\ell + 1, \sigma'\rangle$ holds, with $\sigma' = \llbracket\mu'\rrbracket$.*

Proof. The proof is straightforward, as it directly follows from Theorem D.1 applied to I = **Call** with the initial state $\mu = \langle\varnothing, \varnothing, \varnothing, []\rangle$. Since ℓ is a dummy label, the term $\ell + 1$ is irrelevant and can be disregarded. The only relevant label is ℓ_1, which identifies the first instruction I_1 of f_1, as well as the start of the preamble in its translation, given that labels are preserved by rule (T-Fun). Moreover, since Q is the concatenation of the translations of all functions $f_1..f_n \in P$, as described by rule (T-Prg), each function segment begins with a **proto** instruction and ends with a **retsub**. This explains the presence of the **callsub** $\langle \ell_1\rangle$ instruction used to initiate the TEAL program. Finally, because f_1 is a function without arguments and return values, both μ and μ', as well as their respective encodings σ and σ', have empty stacks.

$\square$

The following lemma states that, at the end of the execution of a Move program P, all data allocated in memory during the computation is cleared, except for the resources stored in the global storage. Intuitively, whenever a program creates a resource using the **Pack** instruction, the resulting value can either be passed to a **MoveTo** instruction or stored in a local variable via **StLoc**. In the latter scenario, however, a corresponding **MoveLoc** must eventually occur; otherwise, a type error arises. Since we consider only well-typed programs, we establish that after the complete execution of a program, both the local variables and the stack are empty, and the memory contains no residual data except for the resources.

Lemma D.3 (Memory Leftovers)**.** *Let P be a program and $\mu \xrightarrow{P} \mu'$ its derivation, then $\mathtt{dom}(M') = \mathtt{codom}(G')$ with $M', G' \in \mu'$.*

Proof. The final state is $\mu' = \langle M', \varnothing, G', [], B'\rangle$ as per Theorem D.2. Now, let $\langle M_1, L_1, G_1, S_1, B_1\rangle$ be the output state of a (M-Pack) rule at some point in a derivation of program P. If a (M-MoveTo) follows, the output state is $\langle M_1[c \mapsto v], L_1, G_1[(a, s) \mapsto c], S_1, B_1\rangle$, where c is a fresh memory location. If instead a (M-StLoc) follows, the output state is $\langle M_1[c \mapsto v], L_1[n \mapsto c], G_1, S_1, B_1\rangle$, where c is a fresh memory location and n is the index of the local variable. In the latter case, a (M-MoveTo) must eventually follow in the derivation. Assuming other rules occur in between, the input state of (M-MoveTo) is $\langle M_1[c \mapsto v] \cup M_2, L_1[n \mapsto c] \cup L_2, G_1 \cup G_2, S_1 \cup S_2, B_1 \cup B_2\rangle$, where $\cup$ here denotes the general extension of sets, maps, or stacks. Notably, M_2, L_2, G_2, S_2, and B_2 may be empty or not. The output of (M-MoveTo) is then $\langle M_1[c \mapsto v] \cup M_2 \setminus c, L_1[n \mapsto c] \cup L_2 \setminus n, G_1 \cup G_2, S_1 \cup S_2, B_1 \cup B_2\rangle = \langle M_1 \cup M_2, L_1 \cup L_2, G_1 \cup G_2, S_1 \cup S_2, B_1 \cup B_2\rangle$, which shows that the effect of the (M-StLoc) rule has been reverted. When the full derivation of P completes, the output state must be $\mu' = \langle M', \varnothing, G', [], B'\rangle$, since (M-Ret) ends the derivation and restores the caller's L after popping them from the stack, yielding the original empty locals as in the initial state μ. The only remaining case is that a memory location c could be enclosed inside a reference `ref` $\langle c, p\rangle$ via (M-BorrowLoc) or (M-BorrowGlobal). However, the Move borrow checker forbids references to resources from escaping their scope or being stored inside data structures. Since we only derive successfully compiled programs, this implies $\nexists c \in M' \mid c \notin \mathtt{codom}(G')$. In other words, M' contains only the memory locations referenced by G', thus $\mathtt{dom}(M') = \mathtt{codom}(G')$. $\square$

We now state the following theorem, which establishes the correspondence between the Move and TEAL program states after execution: the Algorand local state contains exactly the resources present in the Move global storage, and asset balances are preserved accordingly.

Theorem D.4 (Resource Preservation)**.** *Let P be a Move program and let $Q = [\![P]\!]$ be its compiled TEAL counterpart. Given a derivation $\mu \xrightarrow{P} \mu'$ and a derivation $\sigma \xrightarrow{Q} \sigma'$ such that $\sigma = [\![\mu]\!]$ and $\sigma' = [\![\mu']\!]$, then:*

1. *For all $(\langle a, s\rangle \mapsto v) \in G'$, there exists $(\langle a', k\rangle \mapsto \upsilon) \in \Omega'$ such that $\langle a', k\rangle = [\![\langle a, s\rangle]\!]$ and $\upsilon = [\![v]\!]$*
2. *For all $(\langle a, s\rangle \mapsto m) \in B'$, there exists $(\langle a', n\rangle \mapsto m) \in A'$ such that $\langle a', n\rangle = [\![\langle a, s\rangle]\!]$*

for some encoding $[\![\cdot]\!]$, and assuming $\Omega', A' \in \sigma'$.

Proof. By Theorem D.2, the initial and final Move states are $\mu = \langle \varnothing, \varnothing, \varnothing, [], B\rangle$ and $\mu' = \langle M', \varnothing, G', [], B'\rangle$. Their encodings are, respectively, $\sigma = [\![\mu]\!] = \langle [], [], \varnothing, \varnothing, A, \varnothing\rangle$ and $\sigma' = [\![\mu']\!]_{M',\varnothing,G'} = \langle [], [], \varnothing, \Omega', A', \varnothing\rangle$. By Definition D.3, the local state satisfies: $\Omega' = \{\, (\lfloor\!\lfloor a\rfloor\!\rfloor_{32}, \lfloor\!\lfloor s\rfloor\!\rfloor_1) \mapsto [\![M'(G'(a, s))]\!]_{\varnothing,G'} \mid (a, s) \in \texttt{dom}(G') \,\}$, which, together with Theorem D.3, establishes point (1). Moreover, Definition D.3 also defines: $A' = \{\, (\lfloor\!\lfloor a\rfloor\!\rfloor_{32}, \lfloor\!\lfloor s\rfloor\!\rfloor_8) \mapsto B'(a, s) \,\}$, which directly implies point (2).

□