Ca' Foscari
University
of Venice

Master Degree
in **Computer Science and Information Technology**

Final Thesis

Bridging Smart Contract Languages
A Move-to-Aiken Compiler

Supervisor
Ch. Prof. Michele Bugliesi

Assistant supervisor
Dr. Alvise Spanò

Graduand
Andrea Roveroni
Matriculation Number 880092

Academic Year
2025 / 2026

Abstract

Smart contracts enable the development of complex logic and Decentralized Applications, DApps, on blockchain platforms. Ongoing research has focused on language-specific features such as usability, programming paradigms, safety, and security. Among existing smart contract languages, Move provides strong safety guarantees for resource management through its linear type system. However, Move is designed for account-based blockchains and follows an imperative programming paradigm, which limits its adoption to architectures supporting this model, such as Aptos and Sui.

In contrast, blockchains like Bitcoin and Cardano provide different security and performance characteristics based on the UTxO model, which is incompatible with the Move language and is often perceived as having a steeper learning curve. In this work, we design and implement a compiler that enables Cardano developers to write smart contracts using a subset of the Move language, generating code in Aiken, a functional language used for Cardano smart contracts.

The compiler manages global state through UTxOs, maps non-native Move assets into native Cardano assets, and translates imperative constructs into functional logic compatible with Aiken validators. Although not fully implemented at the time of writing, the system is also designed to generate the corresponding off-chain code required by the Cardano ecosystem. This work aims to simplify smart contract and DApp development on Cardano by introducing an abstraction layer over the UTxO model and functional programming paradigm, while preserving transaction purity, formal correctness, and extending asset handling with linear type safety guarantees.

Contents

Chapter 1

Introduction and motivation

Smart contracts have become a cornerstone of modern blockchain ecosystems, enabling the development of complex decentralized applications (DApps) that operate without centralized control. Since the introduction of Bitcoin and later Ethereum, research and development efforts have increasingly focused on improving the expressiveness, safety, and usability of smart contract languages. Today, developers can choose among a wide range of languages, spanning from domain-specific languages (DSLs) such as Solidity to general-purpose languages (GPLs) like Rust or C++, each offering different trade-offs in terms of abstraction, security guarantees, and ease of use.

Among modern smart contract languages, Move stands out for its strong safety guarantees, particularly in the management of digital assets. Its linear type system enforces strict rules that prevent duplication or unintended loss of resources at compile time, significantly reducing the risk of financial vulnerabilities. However, Move is tightly coupled with account-based architectures and follows an imperative programming paradigm, limiting its applicability to blockchains that support such a model.

On the other hand, Aiken is a functional language designed for Cardano's UTxO model. While it benefits from the inherent properties of the underlying architecture, it exposes developers to a lower level of abstraction, particularly when handling transactions and assets. This often results in verbose and error-prone code, as developers must manually encode state transitions and asset manipulations.

These differences highlight a broader challenge in blockchain development: the lack of high-level abstractions that bridge programming paradigms and blockchain models. Prior studies (Bartoletti et al. (2025)) emphasize the importance of such abstractions to improve both security and developer productivity, especially in UTxO-based environments where the programming model can be unintuitive. At the same time, they underline the effectiveness of strong type systems, such as Move's linear types, in enforcing correct asset handling.

In this context, the motivation for developing a compiler from Move to Aiken emerges naturally. The goal is to combine the strengths of both ecosystems: the safety guarantees and familiar procedural style of Move, and the robustness and parallelism of Cardano's UTxO model. By enabling developers to write smart contracts in a subset of Move and automatically translate them into Aiken, the compiler introduces a higher-level abstraction layer over the UTxO model. This approach reduces the cognitive burden associated with functional and approval-based programming, while preserving key properties such as transaction purity and formal correctness.

Ultimately, this work aims to answer an open research question: whether it is possible to reconcile the procedural programming style typical of account-based systems with the UTxO

model, without sacrificing its advantages. The proposed compiler represents a step in this direction, offering a practical solution to improve accessibility, safety, and cross-paradigm interoperability in smart contract development.

Declaration on the use of AI tools During the preparation of this work, the authors used an LLM model, specifically ChatGPT, to improve the readability and language of the manuscript. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the published work. The design and implementation of the compiler, including its architecture and translation logic, were carried out entirely by the authors. In some cases, an AI-based code generation tool, specifically GitHub Copilot, was used during development to ease the writing of automated tests.

Chapter 2

Background

2.1 Blockchain paradigms

2.1.1 Account-Based vs UTxO-Based Models

Blockchain systems differ fundamentally in how they represent ownership and state evolution. Two dominant paradigms have emerged: the *account-based model*, adopted by platforms such as Ethereum and Aptos, and the *Unspent Transaction Output (UTxO) model*, originally introduced by Bitcoin and later extended in Cardano as the Extended UTxO (eUTxO) model. Understanding their differences is essential, as the computational model directly constrains smart contract design, execution semantics, and verification strategies.

Account-based models Account-based blockchains are a type of distributed ledger technology that tracks user balances directly, similar to a traditional bank account. The state of the entire blockchain is thus represented as a mapping between an address and any data related to it:

$$\text{State} := \text{Address} \rightarrow \text{Account Data}$$

Where each Address represents an account that has some balance and possibly executes code.

Each transaction specifies:

- A sender account
- One or more recipient accounts
- A state transition function

Validation proceeds by executing the transaction against the current global state. If the execution succeeds, the state is updated in place. Formally, if S_t denotes the global state at time t and T is a transaction, executing the transaction computes:

$$S_{t+1} = T(S_t)$$

where T is interpreted as a state transformation.

Notice that concurrent transactions implicitly compete for accessing the shared state. Since multiple transactions may attempt to modify the same account data, their execution order becomes semantically significant.

UTxO-based models In contrast, the UTxO model represents the state as a set:

$$\text{State} := \text{Set<UTxO>}$$

where each element is an immutable output of a transaction, that can either be consumed or remain available (unspent).

Each transaction specifies:

1. A set of unspent transaction outputs to spend as inputs,
2. A set of transaction outputs as output

Let the state at time t be the set S_t. A transaction removes consumed outputs and inserts newly created ones:

$$S_{t+1} = (S_t \setminus I) \cup O$$

where I is the set of inputs and O the generated outputs.

Each output contains:

- A value
- A locking condition (script or validator),
- Optional attached data.[1]

Because outputs are immutable, no transaction can partially modify existing state in-place. Instead, state evolution is modeled as data consumption followed by recreation.

Relationship with programming paradigms

While account-based systems evolve by *mutating shared state*, UTxO-based systems evolve by *consuming parts of the state and replacing it with new outputs*. This distinction propagates to all higher-level programming abstractions. In fact, the account-based paradigm naturally supports imperative programming models, since contracts resemble procedures that mutate persistent storage. On the other hand, the UTxO-based model closely resembles functional programming semantics.

Determinism

In account-based systems, transaction validity depends on the entire global state at execution time. Since contracts may read arbitrary storage locations, the exact execution result may change if unrelated transactions are executed beforehand. Therefore, predicting execution requires knowledge of global ordering.

By contrast, a UTxO transaction depends only on the outputs it consumes. Validation requires inspecting only:

- Referenced inputs
- The transaction itself

This allows this particular model to provide stronger determinism guarantees, which simplifies formal verification and static analysis.

[1] In Cardano, this additional data is called a *datum*, enabling richer state encoding without introducing mutable storage.

Concurrency Implications

In account-based systems, conflicts emerge when transactions access the same account. Because the shared state is mutable, serialization is required.

In UTxO systems, conflicts occur only when two transactions attempt to consume the same output. This means that if two transactions T_1 and T_2 are independent if and only if:

$$Inputs(T_1) \cap Inputs(T_2) = \emptyset$$

and can therefore be processed in parallel.

Implications for Smart Contract Design

The ledger abstraction determines how application state is encoded.

In account-based environments, contracts maintain internal storage that persists across executions. Functions directly update variables, and execution resembles traditional imperative programming.

In UTXO-based systems, contracts do not mutate state. Instead, validators verify whether a proposed state transition — represented by the consumption and recreation of outputs — satisfies predefined rules. Notice that contracts approve transitions rather than performing them.

As before, this implies that developers must explicitly reconstruct updated state inside newly produced outputs. The responsibility for state evolution shifts partially from contract code to transaction construction.

2.1.2 On-chain and Off-chain Computation

In addition to differences in state representation, blockchain systems can also be classified according to where computation and state transitions are executed. In particular, it is useful to distinguish between *on-chain* and *off-chain* execution models, which define whether computation is performed directly on the blockchain layer or delegated to external systems.

On-chain computation On-chain computation refers to any logic that is executed and validated directly by the blockchain network as part of transaction processing. In this model, every node participating in consensus independently re-executes the computation to verify its correctness before accepting a state transition. As a result, all on-chain execution is transparent and fully replicated across the network.

Formally, an on-chain computation can be seen as part of the state transition function:

$$S_{t+1} = T(S_t, tx)$$

where T is executed by all validating nodes under consensus rules.

This model guarantees strong consistency and trust minimization, but it comes with inherent scalability limitations, since all computations must be replicated across all nodes.

Off-chain computation Off-chain computation refers to any processing that occurs outside the blockchain consensus layer. This includes computations performed by users, external servers, or dedicated off-chain protocols before interacting with the chain.

In this setting, the blockchain does not directly execute the computation, but instead verifies proofs, commitments, or resulting state updates submitted by external actors. Formally, off-chain computation can be modeled as:

$$\text{compute}(S_t) \to S' \quad \text{(off-chain)}, \qquad \text{verify}(S_t, S') \to \mathit{bool} \quad \text{(on-chain)}$$

This separation allows for improved scalability and flexibility, since heavy computation can be moved outside the consensus layer. However, it introduces additional assumptions, such as the correctness of the off-chain environment or the validity of submitted proofs.

Interaction between on-chain and off-chain systems Modern blockchain applications typically combine both paradigms. On-chain components enforce critical invariants and ownership rules, while off-chain systems handle computation, optimization, or interaction logic. This hybrid design is particularly relevant in smart contract ecosystems, where full computation on-chain would be too expensive or inefficient.

The distinction between on-chain and off-chain execution is orthogonal to the state model (account-based or UTxO-based), but it strongly influences system architecture, scalability properties, and trust assumptions.

2.2 Programming paradigms

In this chapter we provide some background on particular aspects of programming languages that will be encountered later.

2.2.1 Imperative vs Functional Programming

Programming languages can be broadly classified according to the computational paradigm they adopt, among which imperative and functional programming represent two fundamentally different approaches.

Imperative programming models computation as a sequence of commands that manipulate program state. Programs are expressed through assignments, mutable variables, loops, and control-flow constructs that describe *how* a computation proceeds step by step. State mutation is therefore central to the imperative paradigm, where execution is viewed as the evolution of memory through time.

Functional programming, by contrast, models computation as the evaluation of expressions and the composition of functions. Rather than emphasizing mutable state and sequences of commands, it focuses on describing *what* should be computed. Immutability, higher-order functions, and recursion replace many of the mechanisms commonly used in imperative programming, leading to a declarative style where computations are expressed as transformations of values.

The main distinction between the two paradigms therefore lies in their treatment of state and control flow: imperative programs rely on mutable state and explicit sequencing, whereas functional programs favor immutable data and composition. While these differences lead to distinct reasoning models and implementation techniques, they also pose significant challenges when translating programs from one paradigm into the other.

2.2.2 Continuation-Passing Style

A central challenge in translating imperative programs into functional programs lies in reconciling two fundamentally different models of computation. Imperative languages are built around mutable state, sequential execution, and control flow mechanisms such as assignments, loops, and early returns. Functional languages, on the other hand, emphasize immutability, expression-oriented computation, and explicit data transformations. Bridging these paradigms therefore requires techniques capable of encoding imperative behavior within a purely functional setting.

One established approach to this problem is *Continuation-passing Style* (CPS), a program transformation technique in which control flow is made explicit by representing "the rest of the computation" as a first-class function, called a continuation. Rather than returning results directly, computations pass their results to continuations, which determine how execution proceeds.

CPS has long been studied both as a theoretical model of computation and as a practical intermediate representation in compilers, where it is used to make sequencing, control transfer, and evaluation order explicit. These properties make CPS particularly suitable for translating imperative constructs into functional code, since features such as mutable updates, structured control flow, and even side effects can be reformulated as transformations over explicitly threaded computational contexts.

In the context of this work, CPS is adopted as a mechanism to encode imperative behavior in a target language that enforces immutability. In particular, it provides a foundation for modeling assignments and state updates in a purely functional form, enabling the preservation of imperative semantics while remaining within the constraints of the functional paradigm.

2.2.3 Linear Type System

A fundamental problem in blockchain programming is guaranteeing that digital assets cannot be accidentally duplicated or unintentionally lost. Unlike ordinary data, assets such as coins, tokens, or other on-chain resources represent value, and any bug that allows copying a resource without authorization or discarding it silently may lead to severe security and correctness violations. Preventing these two classes of errors, asset duplication and asset loss, is therefore a core requirement in the design of smart contract languages.

Traditional programming languages provide limited guarantees in this regard, as values can generally be copied or dropped freely unless additional mechanisms are introduced. For asset-oriented programming, stronger static guarantees are needed, so that resource safety is enforced by construction rather than delegated entirely to runtime checks or programmer discipline.

An approach to achieving these guarantees is the use of a *linear type system*. Originating from linear logic, linear type systems treat certain values as resources that must be used exactly once: they cannot be duplicated arbitrarily, nor can they be discarded implicitly. Every resource must instead be explicitly consumed, transferred, or returned, allowing ownership and resource usage to be verified statically.

These properties make linear typing particularly suitable for blockchain languages, where preserving asset integrity is a foundational requirement. In languages such as Move, this model is used to encode digital assets as linear resources, ensuring at the type level that they cannot be forged or lost.

In the context of this work, these guarantees are not only a feature of the source language, but also a property that must be preserved through translation.

2.3 The Move Language

2.3.1 Overview and Design Goals

Move is a statically typed smart contract programming language originally designed for the Diem project and later adopted and extended by ecosystems such as *Aptos* and *Sui*. It was conceived specifically for asset-oriented programming, integrating correctness and safety guarantees directly into the language semantics.

Unlike general-purpose smart contract languages that adapt conventional programming models to blockchain settings, Move introduces abstractions designed around ownership, resources, and verifiable state transitions. It is designed considering resource safety as a priority, digital assets are in fact modeled as linear resources that cannot be implicitly copied or discarded; strong static typing and ownership-like constraints provide further safety by preventing a broad classes of vulnerabilities at compile time; in addition, the language is designed to support formal reasoning via tools such as the Move Prover.

One of Move's defining features is the *resource* abstraction. Resources are special structs that satisfy linearity constraints: they cannot be copied or dropped unless explicitly permitted; if needed, these constraints can be relaxed by providing *abilities* to a resource upon declaration, namely:

- *copy*: allows a resource value to be copied
- *drop*: allows a resource value to be discarded
- *store*: allows a resource to be stored inside the global storage, under other resources
- *key*: similar to *store*, but the resource is allowed to exist at the top level of the global storage, and can therefore be accessed by using the operators on the global storage

Abilities can be viewed as permissions attached to types and play a central role in enforcing resource safety.

As a further safety constraint, Move programs are organized into *modules*, which act not only as a way to separate different parts of the program but also to enforce logical ownership of the resources they declare. In fact, only the module that declares a certain resource can create and destroy it; other modules need to use any ad-hoc function exported by the owner module instead; this provides a powerful mechanism to prevent unexpected manipulations of resource values outside the declaring module.

Other characteristics of the Move language is that it adopts an imperative programming style, allowing mutability both in variables ans on the global storage; references are also supported, providing a convenient way to access a value and optionally modifying it.

The security guarantees provided by the Move type checker do not apply only at compile lever; in fact, Move source code compiles to verified bytecode, and execution is guarded by a bytecode verifier enforcing safety properties even beyond source-level typing.

2.3.2 The Aptos Move Ecosystem

While the previous concepts characterize Move at the language level, Aptos extends them into a full execution and programming environment.

Because Aptos follows the account-based model, it exposes a notion of global mutable blockchain state called *global storage*; unlike a simple mapping from addresses to balances,

global storage separates resources from executable code, making an explicit distinction between persistent resource data and published executable modules:

$$\text{GlobalStorage} := \begin{cases} \text{resources} & : & \text{Address} \times \text{ResourceType} \to \text{ResourceValue} \\ \text{modules} & : & \text{Address} \times \text{ModuleName} \to \text{ModuleBytecode} \end{cases}$$

Modules interact with this storage in-place, making Move naturally aligned with account-based state mutation.

Unlike Cardano, where assets are native to the ledger model, Aptos provides standard libraries for implementing assets through Move resources; a representative example is the *Coin* module, which provides a generic resource for *fungible tokens*:

$$\text{Coin<CoinType>} := \begin{cases} \text{value} & : & \text{u64}, \\ \text{name} & : & \text{String}, \\ \text{symbol} & : & \text{String}, \\ \ldots & & \end{cases}$$

Together with operations designed both for administrative tasks, such as initializing a coin and minting or burning some quantities, as well as standard operations, including transferring, splitting, and merging coins.

Assets are therefore not primitive blockchain objects, but resource abstractions implemented through the language itself.

A Move project consists not only in modules but also in *transaction scripts*, historically, scripts served as transaction entry points and defined a single `main` function invoked off-chain. Modern Aptos emphasizes public *entry functions* inside modules, which play the same role.

A transaction therefore is started off-chain by invoking a script or an entry function; this invocation than triggers code execution, which usually modifies the global storage and therefore the state of the blockchain.

It is evident how Move places contract logic primarily on-chain through module execution, unlike in UTxO systems where much logic is shifted off-chain into transaction construction.

Regarding the tooling offered by the Aptos Move ecosystem, we have already mentioned the Move compiler and the Move Prover along with the bytecode verifier, but developers can also leverage a custom package manager, testing framework and a wide variety of ready-to-use modules provided by a solid Move standard framework.

The combination of imperative programming constructs, which allow for an easier programming experience, and strong safety guarantees toward resources makes Move an interesting source language for studying translation toward more functional smart contract paradigms.

2.3.3 Move Syntax in EBNF

To support later formalization and translation, we provide an abstract grammar of Move in *Extended Backus–Naur Form* (EBNF), consisting in a simplified version of (Spanò et al. (2026)).

Programs P consist of a possibly empty sequence of datatype definitions T and one or more function definitions F, whose bodies are composed of expressions E. Types τ include simple types t, references and tuples. Simple types comprise Move built-in primitive types, such as unsigned integers and booleans, together with user-defined struct names s. References may either be immutable or mutable through the `mut` qualifier.

Datatype declarations are shown for *named structs*, but we clarify that *positional structs*, whose fields are accessed by position only, are supported too in Move; however, since they

basically act as a syntactic sugar for named structs, even when it comes to unpacking, we prefer to omit them from the EBNF syntax to avoid redundancy.

Expressions include standard imperative constructs such as variable bindings, assignments, sequencing, conditionals and loops, together with Move-specific operations such as struct packing/unpacking and explicit borrowing and dereferencing operations. Struct values can therefore be manipulated either through field names or positional destructuring patterns.

Function definitions may additionally be polymorphic through type parameters s^*, and expressions support tuple construction as first-class values. Literals k represent primitive runtime values, including integers, booleans, addresses and the unit value $()$.

As a brief note, the grammar reported below allows the generation of certain types that do not result in a Move code accepted by the compiler; specifically, the Move type checker rejects type applications in which references or tuples are used as type arguments. This restriction was likely introduced to ensure that Move resources cannot contain references among their fields and can therefore be handled safely by the program. Since this constraint is not enforced during parsing but rather at a later type-checking stage, we have chosen to keep the grammar simple and mention this special case here instead.

P	$\coloneqq$	$T^*\ F^+$	**program modules**
T	$\coloneqq$	`struct` $s\ \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$	**user-defined structs**
F	$\coloneqq$	$f\langle s^* \rangle (x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau_0\ \{E^*\}$	**function definitions**
τ	$\coloneqq$		**types**
		t	simple types
	$\mid$	`&[mut]` t	reference types
	$\mid$	$(\tau_1, \ldots, \tau_n)$	tuple types
	$\mid$	$s\langle \tau_1, .., \tau_n \rangle$	type application
t	$\coloneqq$		**simple types**
		`u8 | u16 | u32 | u64 | u128 | u256`	unsigned integers
	$\mid$	`bool`	booleans
	$\mid$	`address`	addresses
	$\mid$	`signer`	signers
	$\mid$	`unit`	unit type
	$\mid$	s	type names
E	$\coloneqq$		**expressions**
		k	literals
	$\mid$	x	variables
	$\mid$	`let` $x = E_1$ `in` E_2	let bindings
	$\mid$	$s\ \{x_1 = E_1, \ldots, x_n = E_n\}$	packing
	$\mid$	`let` $s\ \{x_1, \ldots, x_n\} = E_1$ `in` E_2	unpacking
	$\mid$	$E_1 = E_2$	assignments
	$\mid$	E_1; E_2	sequences
	$\mid$	`if` E_1 `then` E_2 `else` E_3	conditionals
	$\mid$	`while` E_1 `do` E_2	while loops
	$\mid$	`&[mut]` E	references
	$\mid$	$*E$	dereferences
	$\mid$	$E.x$	field selections
	$\mid$	$f(E_1, \ldots, E_n)$	function calls
	$\mid$	$E_1\ bop\ E_2$	binary operations
	$\mid$	$uop\ E$	unary operations
	$\mid$	$(E_1, \ldots, E_n)$	tuples
k	$\coloneqq$		**literals**
		$()$	unit literal
	$\mid$	n	unsigned integers
	$\mid$	`true` | `false`	booleans
	$\mid$	a	addresses

Table 2.1: Move EBNF syntax

2.4 The Aiken Language

2.4.1 Overview and Design Goals

Aiken is a statically typed functional programming language designed specifically for writing smart contracts for the Cardano blockchain. Unlike general-purpose languages later adapted for blockchain programming, Aiken was designed directly around the execution model and constraints of the eUTxO ledger.

Its design reflects the fact that, in Cardano, smart contracts do not actively perform state transitions, but rather validate whether proposed transitions are allowed; being designed for this blockchain, it is natural to expect that the Aiken language closely follows the functional and deterministic nature of Cardano validator nodes, emphasizing strong typing, pure execution and predictable runtime behavior. When compiled, Aiken code produces Untyped Plutus Code, a representation that very closely resembles *Lambda Calculus* and is then interpreted and executed by the Cardano VM on the validator nodes.

Being Aiken a functional programming language, it provides numerous features familiar for this programming paradigm, such as enforcing state immutability, no side effects, deterministic evaluation, and an explicit data flow from function inputs to outputs. Note how this approach closely matched the eUTxO models, where contracts validate state transition rather then performing them, making Aiken particularly suited for writing smart contracts on Cardano.

The central abstraction in Aiken is the *validator*; a validator is fundamentally a predicate over transactions.

At a high level:

$$validate : \text{Datum} \times \text{Redeemer} \times \text{Context} \rightarrow \text{Bool}$$

Rather than describing sequences of actions, Aiken programs specify conditions under which a transaction is valid.

As a common characteristic of functional languages, Aiken uses a strong static type system that supports, among others, *Algebraic Data Types* ADTs, pattern matching and parametric polymorphism, although the latter being relatively simple, as it will be show in later chapters. In the case where some introduction to these concepts is needed, we suggest to look at resources on functional programming paradigm, such as Bird and Wadler (1988).

Unlike account-based smart contract languages, Aiken has no notion of persistent mutable contract storage; state evolution is instead encoded explicitly through the consumption and recreation of UTxOs; this property is fundamental rather than incidental, and shapes the entire programming model.

2.4.2 The Aiken Ecosystem

Aiken exists within the broader Cardano smart contract ecosystem and is tightly coupled to the eUTxO execution model. Aiken contracts operate over the concepts of eUTxOs, datums, redeemers and transaction contexts; rather than mutating shared state, they validate whether a proposed transaction preserves contract invariants, and the state is inherently modified when input UTxOs are spent to produce new output UTxOs.

Because Cardano supports native assets at the ledger level, Aiken can reason about multi-asset values directly, without implementing token logic from scratch through contracts.

Values are naturally represented as multi-asset bundles:

$$\text{Value} := \{(\text{policy\_id}, \text{asset\_name}, \text{quantity})\}$$

This allows contracts to manipulate tokens as native data rather than external abstractions.

As said, computation is usually performed by spending eUTxOs, however Aiken validators support other *purposes* for other use cases, including:

- **spend**: validates spending of script-locked outputs
- **mint**: validates minting and burning policies
- **withdraw**: validates reward withdrawals
- **publish**: handles delegation certificates
- **vote**: validates governance voting
- **propose**: validates governance proposals

Each purpose requires different parameters and triggers the execution of the corresponding *handler*, if present. Each transaction specified which purpose to execute, along with the required arguments.

For completeness, the *spend* purpose is defined as follows:

$$validator_{spend} : \text{Datum} \times \text{Redeemer} \times \text{OutputReference} \times \text{Transaction} \to \text{Bool}$$

Its role is to validate spending conditions for all input eUTxOs locked by the current validator.

Minting policies are used to govern asset issuance and destruction, and have the form:

$$validator_{mint} : \text{Redeemer} \times \text{PolicyId} \times \text{Transaction} \to \text{Bool}$$

Similarly to the Aptos Move ecosystem, Aiken also provides a set of tools useful for developing smart contracts, such as the Aiken compiler along with a property-based testing framework, a package manager, and standard library.

All the properties that we have stated for Aiken demonstrate its suitablity for writing smart contracts on the Cardano blockchain, although its functional paradigm and the UTxO model surely represent a steeper learning curve than Aptos Move. This makes Aiken a natural target language for studying translations from imperative smart contract models.

2.4.3 Aiken Syntax in EBNF

To support later formalization and translation, an abstract grammar of Aiken can be introduced using Extended Backus–Naur Form (EBNF).

Programs P consist of a possibly empty sequence of datatype definitions T and function definitions F, whose bodies are composed of expressions E. Types τ include both primitive built-in types, such as integers and booleans, and user-defined algebraic datatype names s. Composite types additionally support parameterized collections, such as lists, together with generic type names s^* used in polymorphic functions.

Datatype declarations support both *named constructors*, whose fields are explicitly labeled, and *positional constructors*, whose arguments are identified solely by position; similarly to the

Move syntax, positional constructors basically act as syntactic sugar for named constructors, so we prefer to omit them from the EBNF syntax to prevent redundancy; as a side note however, note that the Aiken compiler treats both constructors the other way around, meaning that the resulting Untyped Plutus Core only refers to arguments by their position and not by label.

Expressions provide a predominantly functional programming model, including immutable variable bindings, constructor applications, function calls, conditional expressions and pattern matching through `when` expressions. Destructuring bindings allow values to be unpacked directly through patterns P, enabling concise manipulation of algebraic datatypes.

The language also includes higher-level constructs such as expression blocks, tuples and field access operations. Control flow is expression-oriented: conditional branches and pattern matches evaluate to values rather than statements. Special expressions such as `todo` and `fail` are used to represent incomplete or intentionally failing computations during development or validation.

Function definitions may be polymorphic through type parameters s^*, and values are represented through literals k, constructor applications and tuples. Primitive literals include integers, booleans, strings, byte arrays and the unit value `Void`.

Aiken supports pattern matching, allowing to test an expression E against patterns M, resulting in a corresponding expression E. For simplicity, patterns are not shown in this syntax.

P	$\coloneqq$	$T^*\ F^*$	programs
T	$\coloneqq$	`type` $s\ \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$	**user-defined structs**
F	$\coloneqq$	`fn` $f\langle s^* \rangle (x_1 : \tau_1, \ldots, x_n : \tau_n) \rightarrow \tau\ \{E\}$	**function definitions**
τ	$\coloneqq$		**types**
		`Int`	integers
	$\mid$	`Bool`	booleans
	$\mid$	`ByteArray`	byte arrays
	$\mid$	`Void`	unit type
	$\mid$	`Data`	plutus data
	$\mid$	`List`$\langle \tau \rangle$	lists
	$\mid$	s	custom type names
E	$\coloneqq$		**expressions**
		k	literals
	$\mid$	x	variables
	$\mid$	`let` $x = E_1$ `in` E_2	let bindings
	$\mid$	$s\ \{x_1 = E_1, \ldots, x_n = E_n\}$	packing
	$\mid$	`let` $s\ \{x_1, \ldots, x_n\} = E_1$ `in` E_2	unpacking
	$\mid$	$f(E_1, \ldots, E_n)$	function calls
	$\mid$	$E_1\ bop\ E_2$	binary operations
	$\mid$	$uop\ E$	unary operations
	$\mid$	`if` $E_1\ \{E_2\}$ `else` $\{E_3\}$	conditional expressions
	$\mid$	`when` E `is` $\{R_1 \ldots R_n\}$	pattern matching
	$\mid$	$E.x$	field access
	$\mid$	$(E_1, \ldots, E_n)$	tuples
	$\mid$	$\{E_1 \ldots E_n\}$	expression blocks
	$\mid$	`todo`	incomplete expressions
	$\mid$	`fail`	failing expressions
R	$\coloneqq$	$M \rightarrow E$	match rules
k	$\coloneqq$		**literals**
		n	integer literals
	$\mid$	`True` $\mid$ `False`	boolean literals
	$\mid$	`@"..."`	string literals
	$\mid$	$\#[b_1, \ldots, b_n]$	bytearray literals
	$\mid$	`Void`	unit literal

Table 2.2: Aiken EBNF syntax

Chapter 3

Overview of the translation

In this chapter we present an high-level overview of how a Move smart contract is translated into an Aiken validator. We introduce how the global storage is represented, how the computation of a Move transaction is mapped into a Cardano transaction, and additionally provide a way to represent the *Coin* module via Cardano native assets.

3.1 State representation

3.1.1 State in UTxOs

As previously stated, the Aptos blockchain follows an account-based model and therefore has the concept of a global state of the blockchain that is mutated by each transaction.

Cardano does not have this concept, however it is possible to leverage the eUTxO model, and particularly the *datum* field, to store arbitrary data.

The naive implementation is to store the entire state of a smart contract into one or more outputs, such that the state can be reconstructed by computing the union of all the datums:

$$\text{State} = \bigcup_{u \in \text{eUtxO}} \text{datum}(u)$$

Naturally, this approach does not scale as the state grows, since it requires to prove all the unspent outputs to the transaction and generating an entire new set as output, causing large transaction fee and possibly hitting the maximum transaction size allowed by Cardano.

Other alternatives have been proposed, such as the work of Fanton et al. (2025), that evolve the current de-facto standard known as State Thread Token (Aiken (2026)), and Bartoletti et al. (2026) that propose a hybrid UTxO approach. For simplicity and demonstration purposes however, the current implementation of this compiler uses the naive approach, but it can be easily adapted so to leverage better alternatives.

3.1.2 Implementation in Aiken

In particular, we are interested in capturing the state related to resources, previously formalized as:

$$\text{resources} := \text{Address} \times \text{ResourceType} \to \text{ResourceValue}$$

In the compiled Aiken code, we have chosen to represent it as a nested dictionary for implementation convenience. The outer dictionary maps an address to an inner dictionary that

associates a resource type to its actual value:

$$\text{resources} := \text{Address} \to \text{ResourceType} \to \text{ResourceValue}$$

The two versions are equivalent by currying. In particular, each entry $(address, type) \mapsto value$ in the initial formalization corresponds to the entry $address \mapsto (type \mapsto value)$ in the Aiken representation, and vice versa.

In addition, it is important to present a consideration regarding the inner dictionary. It can be noticed that it uses a type as key for the mapping, while the actual implementation should expect a runtime value. This causes the problem of knowing the runtime value of types passed as type arguments, which will be examined in a later chapter.

Global storage operators

As seen in the previous chapter, the Move language works on the concept of a global storage, providing the following operators to manipulate it:

- `move_to<T>(&signer, T):()` : Publishes `T` under `signer`'s address, aborting if the signer already holds `T`
- `move_from<T>(address):T` : Removes `T` from `address` and returns it, aborting if the address does not hold `T`
- `borrow_global_mut<T>(address):&mut T` : Returns a mutable reference of the `T` stored under `address`, aborting if the address does not hold `T`
- `borrow_global<T>(address):&T` : Same behavior of `borrow_global_mut` but returns an immutable reference
- `exists<T>(address):bool` : Returns `true` if `address` holds `T`. It never aborts

These operators have been implemented in a custom library, with a version for the off-chain part and a second one for the on-chain. Any invocation of the Move operators is therefore replaced by the invocation of the corresponding custom operator.

The provided library has been designed to behave exactly as the original Move operators, including in the abort conditions, but works on the alternative implementation of the state as nested dictionaries.

A note on time complexity

We want to consider the impacts of implementing the state via means of dictionaries. In particular, transaction fees on Cardano are linearly influenced by both *CPU steps*, evaluated as discrete units of work, and *memory cost* of a validation execution.

In the Aiken standard library (stdlib), dictionaries are ordered sets of key-value pairs. This means that, without any optimization, accessing a dictionary corresponds to a linear scan of the elements. As the state grows, so does the number of elements in the set, thus increasing the total CPU steps needed to perform operations on the global storage and the related fees. On the worst case, this could result in hitting the maximum budget for CPU steps, causing the blockchain to reject the transaction.

At the time of writing, profiling of the generated Aiken validators is under planning, in order to better understand the impacts of this approach on the transaction fees and explore possibilities to optimize it.

3.2 Move to Aiken project

In this section, we present the approach adopted to translate a Move project into a single validator written in Aiken, alongside multiple off-chain scripts.

3.2.1 Off-chain computation

As discussed in the previous chapter, Move modules must be deployed on-chain so that transaction scripts can reference and invoke them during execution.

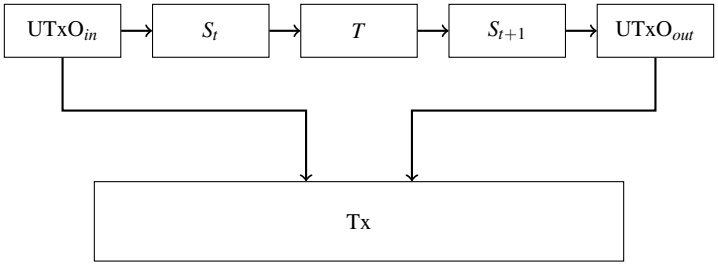
Cardano, however, follows a different execution model. Since validators act only as predicate functions, the main computation must take place off-chain. As a consequence, the off-chain logic has to reproduce the same computation that Move would normally execute on-chain, and then submit the result to the validator, which is responsible for verifying its correctness and, consequently, validating the transaction itself.

For this reason, each Move script and module is translated into a corresponding off-chain program. Because Cardano does not provide a notion of global storage, the generated off-chain code must first retrieve the current state, execute the same computations as the original Move program, and finally produce the updated state.

As explained in the previous section, the blockchain state is represented through datums stored across one or more eUTxOs. Therefore, the off-chain code must access all relevant datums, reconstruct the global state into a convenient in-memory representation, perform the required computations, and eventually serialize the updated state back into one or more output UTxOs.

In this way, the off-chain logic can emulate the behavior originally implemented by Move on-chain modules. Notice also that state evolution follows a functional model: instead of mutating the existing state, the computation produces a new output state derived from the previous one.

Once the computation is complete, the off-chain code constructs a transaction containing the input UTxOs, the newly generated output UTxOs, the validator script, and any additional required metadata, before finally submitting the transaction to the blockchain.



3.2.2 On-chain computation

The validator included in the transaction is responsible for verifying the correctness of the computation. More precisely, it must ensure that the transaction outputs correspond to the expected result produced from the given inputs.

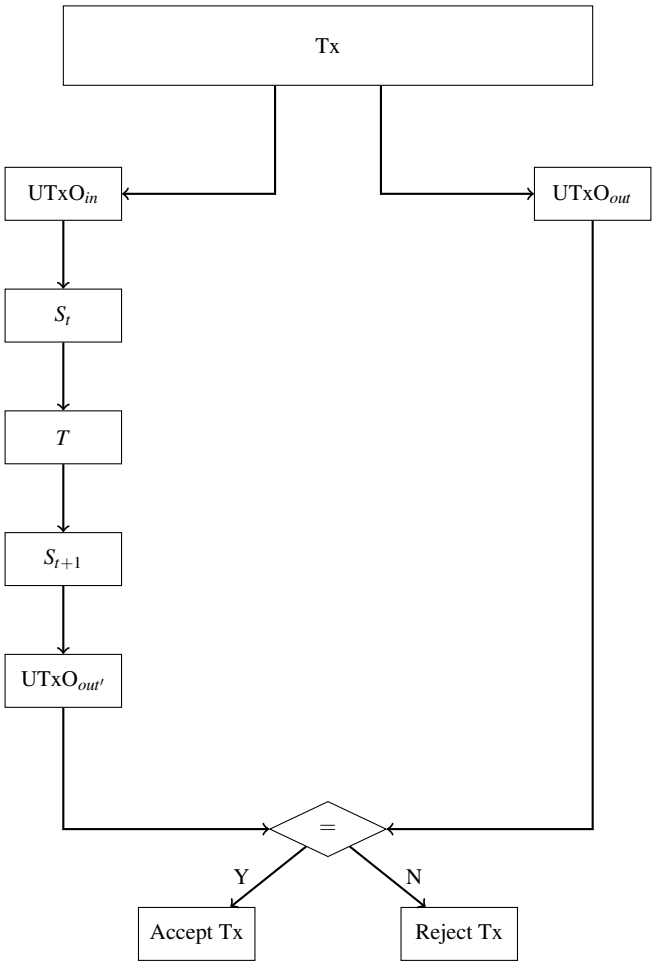
To achieve this, the validator reproduces the same computation previously executed off-chain. It first reconstructs the initial state from the input UTxOs and then applies the same transition logic used by the off-chain code. This process produces a computed final state, which can then be compared against the output UTxOs contained in the transaction.

The validator verifies two conditions:

- The computed non-asset state must match the non-asset data contained in the transaction outputs
- The computed asset balances must match the assets included in the transaction outputs

This distinction is necessary because native assets in Cardano are stored separately from the datum field of a UTxO. While the datum only contains arbitrary data with no intrinsic monetary value, assets are represented directly in the UTxO value field.

Together, these checks guarantee that the state transition executed on Cardano produces the same final state that would have resulted from executing the original transaction on the Aptos blockchain.



Move project translation

We now describe how a Move project is translated into an Aiken project while preserving all the computations that can originally be performed in Move.

To identify the set of possible computations in a Move project, recall that each Aptos transaction specifies a code entry point that initiates the on-chain execution. The set of all entry points of a Move project corresponds to the set of main functions defined in its script files:

$$\text{Entry points} = \bigcup_{s \in \text{scripts\_in\_project}} \text{main\_function}(s)$$

Identifying the main function of a script is straightforward because, by construction in Move, each script contains exactly one function declaration.

For each entry point, it is also necessary to collect its function signature, since the correct arguments must later be supplied during invocation.

This information can be represented as a set containing a unique identifier for each main function together with its corresponding function type:

$$\text{Entry points} := [(\text{function\_identifier}, \tau_1 \times \ldots \times \tau_n \to \tau_r)]$$

Now that the notion of Move entry points has been defined, recall the existence of the *redeemer* field in Cardano transactions. The redeemer is an arbitrary piece of data attached to the transaction and made available to the validator during execution.

We use the redeemer to specify which Move entry point is being invoked, allowing the validator to execute the corresponding validation logic. The redeemer can therefore be represented as an *Algebraic Data Type* (ADT), where each variant corresponds to an entry point together with its associated parameters:

$$\text{Redeemer} = \sum_{s \in \text{scripts\_in\_project}} \text{function\_identifier}(s)(\text{function\_params(s)})$$

Notice that the return type is irrelevant in this representation. Indeed, the Move type system enforces that an entry point cannot return any value; equivalently, its return type can always be considered the unit type.

As an example, consider a Move project containing the following entry points:

$$\text{Entry points} = [(\text{Entry1}, \tau_1 \times \tau_2 \to \tau_3), (\text{Entry2}, \tau_4 \to \tau_5), (\text{Entry3}, \tau_6 \times \tau_7 \to \tau_8)]$$

The generated redeemer type is therefore:

$$\text{Redeemer} = \text{Entry1}(T_1, T_2) + \text{Entry2}(T_4) + \text{Entry3}(T_6, T_7)$$

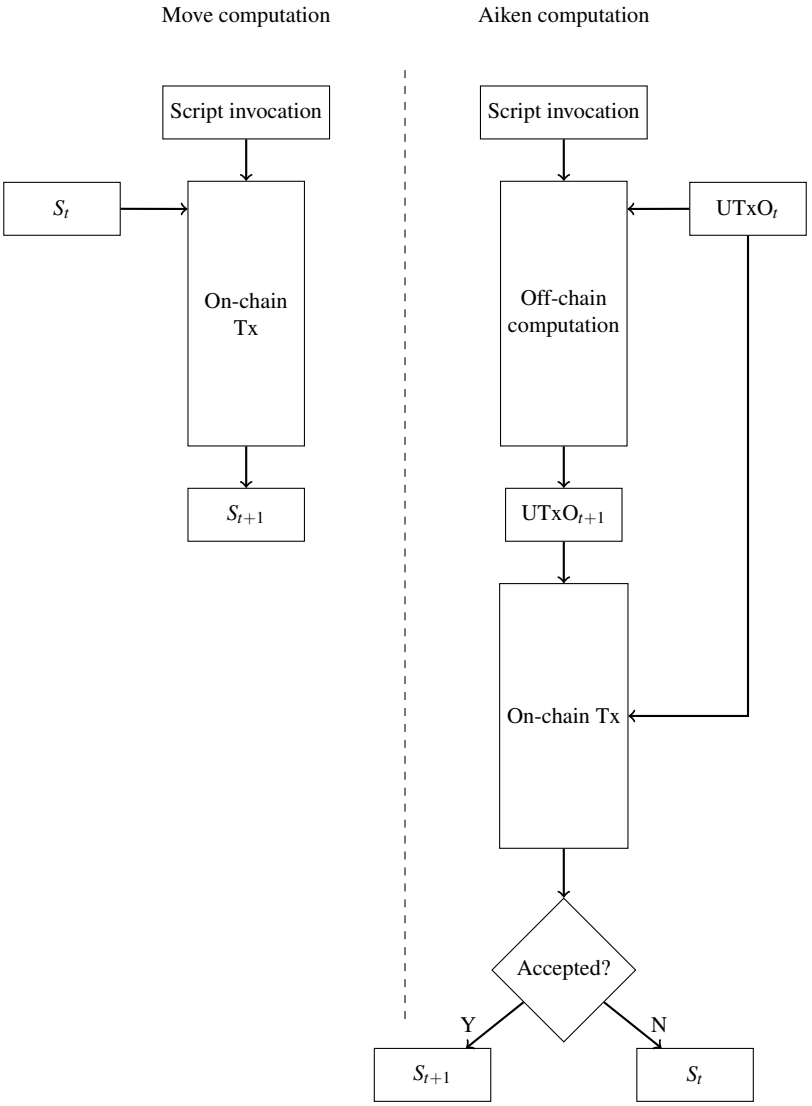
The validator body then contains a branch in the spend-purpose logic that pattern matches on the redeemer in order to invoke the appropriate entry point, passing the arguments stored in the corresponding constructor. Continuing the previous example using Aiken syntax:

```
...
when redeemer is {
    Entry1(arg0, arg1) -> entry1(arg0, arg1)
    Entry2(arg0) -> entry2(arg0)
    Entry3(arg0, arg1) -> entry3(arg0, arg1)
}
...
```

The off-chain code is therefore responsible for constructing the correct redeemer variant and supplying the corresponding arguments.

With this translation scheme, invoking a Move script becomes equivalent to invoking an off-chain program that performs the corresponding computation, constructs a transaction, includes both the input state and the expected output state, attaches the compiled validator, includes the appropriate redeemer variant, and finally submits the transaction to the Cardano blockchain. The validator then reproduces the same computation and verifies that the resulting state matches

the expected one. If the verification succeeds, the transaction is accepted and added to the blockchain.



3.3 Aptos Coin module

As mentioned, Aptos provides asset management through modules included in the Move standard library. One of these is the Coin module, which enables the definition of custom fungible tokens and provides methods to operate on them. These methods support both administrative tasks, such as initializing, minting, and burning coins, and standard operations, including transferring, splitting, and merging coins.

When translating assets to Cardano, however, representing them as resources within the global storage would effectively treat them as ordinary data without intrinsic monetary value. Instead, they must be converted into native Cardano assets embedded in UTxO values, allowing them to be recognized and exchanged like any other currency on the blockchain.

At the time of writing, the translation has been defined and implemented for the Coin module, and the same approach can be extended to other Move modules, such as Fungible Asset and Digital Asset.

The invariant we maintain is the following: whenever a given Coin is owned by a specific account on the Aptos blockchain, the corresponding Cardano asset must also be owned by the same address.

This implies that, for every Coin present in the Aptos global storage at any point in time, there must exist an eUTxO containing the corresponding asset and locked by the same address:

$$\begin{aligned} &\forall \mathit{coin} \in \text{global\_storage},\ \text{aptos\_owner}(\mathit{coin}) = a \\ &\Rightarrow \exists u \in \text{eUTxOs} : \text{cardano\_asset}(\mathit{coin}) \in \text{values}(u) \wedge \text{cardano\_owner}(u) = a \end{aligned}$$

Since Move Coins are generalized as `Coin<CoinType>`, it is also necessary to uniquely identify the specific `CoinType` being stored. This can be achieved by leveraging the *asset_name* property of Cardano assets, which, as recalled, are identified by the triplet:

$$\text{Asset} := (\text{policy\_id}, \text{asset\_name}, \text{quantity})$$

Our compiler therefore includes dedicated logic that translates all operations on the *Coin* module into corresponding operations on eUTxO assets. This is achieved through an ad-hoc library that provides both off-chain and on-chain components.

The Aiken validator retrieves the initial assets from the input eUTxOs, performs the required computation, and finally verifies that the resulting assets match those contained in the transaction output eUTxOs.

Chapter 4

The Compiler: Design and Implementation

4.1 Translation steps

Now that we have provided an high-level overview of how each aspect of the problem is mapped from the Aptos blockchain to Cardano, we proceed by describing how the actual compilation happens from a Move project to an Aiken project.

Specifically, the compilation has been divided into discrete steps. Each of them will be analyzed more in detail in its own section:

- **Lexing and parsing**: the compilation starts by taking as input the Move source code, performing the lexing and parsing steps normally present in compilers, and produces a set of *Abstract Syntax Trees* (AST) to represent the whole project
- **AST rewrital** steps: They progressively replace the imperative parts of the Move programs into functional constructs, producing *Intermediate Representation* ASTs that will be further mapped to Aiken code. Designing and implementing these steps required the majority of the efforts and time regarding the practical part of this project. As it will be seen later, they are composed by multiple steps each solving a particular problem of the compilation. Although the compilation steps are logically independent, the overall process is order-dependent, as the output of each step affects the applicability or behavior of subsequent steps. At the end, these result into an IR that is ready to be translated into an Aiken validator
- **Aiken generation**: Takes as input each Intermediate Representation AST and produces a corresponding module in the Aiken Language. This code generation happens mostly by a one-to-one mapping for each node type, since nearly all of the abstractions have been handled by the rewrital steps
- **Validator generation** and file writing: The final part consists in generating the main Aiken validator for the project and writing the actual files on the file system. This step collects all the translation flows seen in the previous chapter, such as state and asset management, redeemer creation via ADTs, custom libraries imports, in order to produce an Aiken project that is ready to be compiled and invoked by off-chain scripts

At the time of writing, the generation of the off-chain code is under design and not currently implemented. In any case, the proposed destination language is Typescript, leveraging a library named MeshJS as interface for the Cardano blockchain.

4.2 Lexing and parsing

As for any compiler, starting from the source code requires converting the *program text* into a representation that is more suited to be further worked on. This representation is called *Abstract Syntax Tree* (AST) and is a way to represent the syntactic structure of any program as a tree, where each node represents a language construct, such as an expression, declaration, statement or operator, and all its children represent nested constructs.

Lexer

The first step in order to build such ASTs requires the use of a *Lexer*, alternatively called *Scanner*. A lexer has the role of analyzing a textual content representing the program, that can be obtained in any way such as reading a source file or the standard input of an interpreter, and converting it into a sequence of *tokens*, that represent the base terms and keywords of a language. Example of tokens are the symbols for algebraic and boolean operators such as +, -, < and so on, but also keywords for function declarations (usually *function*, *func* or even *fn*), bindings (commonly used *var*, *let*, *const*), assignments and type annotations.

In addition, the Lexer removes tokens and symbols that have no particular meaning and can be ignored, such as white-spaces and line termination characters.

Since lexing is a particularly common and well established practice, various tools exist that allow to define a parsed in a declarative way and take care of generating the actual code for the lexer. In our work, we have leveraged the Haskell's *Alex* library for lexer generation (Alex Developers (2026)), that provides a friendly interface to define tokens and lexing rules.

The Alex generator is thus invoked during the compilation step of the compiler, resulting in a ready to use lexer that is invoked as one of the first steps for the actual translation (the initial ones being recognizing the project contents and reading the files).

Creating the definitions for performing the lexing of Move source text has been mainly straightforward task, and has mostly followed what done by the Move compiler in order to maintain the same final results.

Parser

Along with the lexer, a *Parser* is usually present. The parser has the more complex task of reading the sequence of tokens produced by the lexer and interpreting them, in order to represent the actual semantic of the original source code and building the AST. The main difficulties arising from creating a parser is that writing the grammar rules for a language might generate ambiguities, where the same program text can be derived in multiple valid ways. Additionally, writing a grammar for an already existing language means that the precedence order of the language constructs and operators has to be respected. As a final aspect, finding a good and flexible definition for the AST nodes is crucial for easing the steps that will follow.

Handling the complexities related to the grammar can be made easier by leveraging parser generators libraries, that allow to write the grammar files in a specific, declarative way and automatically alert when ambiguities are found. In this project, we used the Haskell's *Happy* library (Happy Developers (2026)) to generate a parser from the custom grammar rules for the Move language.

Writing the grammar for the Move language required multiple iterations and trials since different issues have been encountered. To ensure that each progressive extension of the parser still maintained the same results of the Move language, a test-based approach has been adopted during the development. Multiple tests have been added to the project, running automatically

and consisting in multiple test cases that assert that a Move program text is parsed accordingly to the original Move compiler.

There have been found multiple interesting cases during this step, but for simplicity and easier reading they have been omitted from this document. We suggest to look at the compiler's source code for more details (CardaMove Contributors (2026)).

At the end of this process, the parser supports most of the grammar for the Move 1.0 language, with some functionalities of the 2.0 version.

The structure of the AST has been defined so to mostly resemble the ASTs generated by the Move compiler, with exceptions regarding intermediate representations and type representations.

Once the Move project files are parsed and represented via ASTs, the rewrital process takes place. This step has been designed in multiple, logically independent steps, each tackling a specific problem of the compilation process.

4.3 Traversal helper and type deduction

At an high level, each step performs one or more post order traversals of an AST. Therefore, we have implemented a post order traversal helper that allows to traverse a generic AST while collecting some useful information along the paths, such as the the current scope the program, and provides interfaces for replacing nodes and carrying around a custom state.

To implement this, we have leveraged the Haskell's *Uniplate* library (Mitchell (2026)). Its documentation describes it as a generic library, meaning that it allows to writes functions that operate on a certain data structure without the need to handle all the aspects of the data structure. Particularly, given the fact that the traversal has to work on expression nodes, and such expressions are defined with numerous constructors, the Uniplate library has allowed to reduce the boilerplate code to pattern match all the possible variants and their recursive expression nodes.

As an example, we have leveraged the *descend* function and its monadic variant *descendM* to recursively descend into *Expr* types:

```
-- |
-- Perform a transformation on all the immediate children,
-- then combine them back
descendM :: Monad m => (on -> m on) -> on -> m on
```

In our case, the *descendM* works on *Expr* nodes, applying a transformation function on all its immediate *Expr* children and combining the resulting node. By passing a transformation function that calls itself recursively, it is possible to implement a post order traversal of an *Expr* node.

An important information that is needed in almost all rewriting steps is the ability to know the type of a certain expression, in order to handle it accordingly. In order to do so, we have implemented a simple type deduction function that accepts the expression to type, the scope containing all the available variables at that point of the Move program, and tries to deduct the type of the expression:

```
-- |
-- Given any expression and the accumulated scopes,
-- tries to infer the type of the expression
inferExprType :: Expr -> [Scope] -> Type
```

For clarification, the implemented type deduction is not complete, meaning that can still fail for some very specific cases, but it has been extended and tested in such a way that it is able to handle the majority of the cases encountered during the development, such as references and dereferences of either variables or literal values, record accesses, function invocations, structs construction, and so on. If needed, more cases can be easily added in a second time.

Knowing the type of an expression is useful also for deducing the type of a variable when it is declared without an explicit type annotation. This is particularly helpful in the steps that deal with references.

The inductive rules for type deduction are provided in the appendix A.

4.4 Loops

The first step of the compilation handles the rewrital of certain constructs present in the Move language that allow for iteration. Since Move follows an imperative programming style, it naturally provides explicit iterative constructs such as while loops and other control-flow mechanisms, that rely on state mutation and instruction execution. These constructs allow developers to express repetition in a direct and familiar way, closely aligned with traditional imperative programming practices.

In contrast, the functional nature of the Aiken language deliberately omits native support for imperative looping constructs such as while or for loops. This design choice is motivated by the need for greater predictability and analyzability of the validators, where unbounded or poorly controlled loops may introduce risks related to resource consumption and termination.

As a consequence, when translating code from Move to Aiken, iterative behaviors expressed through loops must be systematically reformulated. The approach adopted in this work consists in encoding such constructs using recursive functions, which provide an equivalent mechanism for iteration while remaining consistent with Aiken's functional paradigm. Through recursion, it is possible to model repetition in a structurally controlled manner, often making termination conditions more explicit and better suited for formal verification.

The Move language provides three constructs for looping:

- `while`: Like in most imperative languages, this construct repeats its body until a condition evaluates to `false`. In addition, the familiar `break` and `continue` expressions can be used respectively to exit the loop and to skip to the next iteration
- `for-in`: It iterates over an integer range, executing its body for each element in the range. It also supports the `break` and `continue` expressions
- `loop`: It continuously repeats its body without checking any condition, until a `break` expression is found. Without a `break`, execution will continue forever
- Starting from version `2.1` of the Move language, loops can also specify labels that can be referred by `break` and `continue` to indicate the exact instruction to jump next

In our work, we focused on translating the `while` and `loop` construct, leaving the `loop-in` for a later implementation, which we suppose will be relatively straightforward and reuse most of the existing translation logic. In addition, we focused on features of the Move language prior to version `2.0` and a limited support for the `2.0`, therefore the support for labels is left for a future version of the compiler.

An important consideration to make is that this step is the only one that does not produce *intermediate representation* ASTs, but rather entirely leverages features of the Move language

to rewrite loops in an alternative way. The resulting code will therefore be a perfectly valid Move code that can type check and compile.

We start by describing the compilation logic for the `while` construct. Further on, we will see a very simple mapping of the `loop` construct to this same problem.

At an high level, the translation leverages the traversal helper to perform a post order traversal of each AST, creating additional nodes in the process that correspond to the recursive functions needed to represent each looping construct. Each looping construct is then replaced with an invocation of the corresponding function. At the end of the traversal, all generated functions are lifted to the top level of the enclosing module or script and appended to the existing declarations. The final result is a transformed program in which all while loops have been eliminated and replaced by structurally recursive functions, making the control flow explicit and compatible with a functional execution model.

In order to maintain the same semantics of the original construct, this translation logic should consider a few points:

- The invocation of a recursive function should maintain the same behavior of the original looping construct
- Both the condition and the body of the looping construct can refer to variables outside their local scope. If the outer nodes are not considered, occurrences of those variables can be interpreted as *free variables*. When nodes are extracted from their current position on the AST and lifted to the top level, it is important to allow them to still refer to any of their free variables, that must therefore present in the scope of the program, otherwise the produced code would be semantically incorrect since it would refer to undeclared variables.

To consider the second point, the compiler performs an initial scan of both the condition and body nodes of each `while` construct, in order to identify their free variables, meaning variables whose binding does not happen inside the while construct but rather in an outer scope. The identifiers and type of those free variables are collected and mapped as parameters for the function that will later correspond to the while construct itself. In this way, free variables inside the condition and body of the while construct will still refer to existing bindings, precisely the ones introduced by the function parameters. The occurrence of the while construct itself is then replaced by the invocation of the corresponding function, passing any of the referenced variables as arguments.

Since Move allows to perform mutation on variables, it is needed to consider the case where those free variables translated as function parameters can be reassigned, and their new value should be preserved both when performing subsequent iterations and when exiting from the loop. To allow for this, the function parameters are added not following their actual type, but rather wrapped as mutable references, a feature of Move language that allows a variable to point to another one and both access its value and update it, in a syntax similar to C-like pointers. This naturally means that any operation on those variables inside the function body needs to be rewritten as the same operation on references.

Inductively, substituting an occurrence of `while` with a function call can be expressed as follows:

$$\frac{\forall v \in Fv(c) \cup Fv(e).\Gamma \vdash v : \tau_i \qquad \mathscr{W}_i \notin \Delta}{\Gamma \vdash \texttt{while}(c)\ e \longrightarrow \mathscr{W}_i(\texttt{\&mut}\ v_1, \ldots, \texttt{\&mut}\ v_n)}$$

Where $Fv(e)$ indicates the set of all the free variables in the expression e, Δ is the set of all the newly defined functions, and $\mathscr{W}_i$ is the name given to the resulting function, chosen so that

it is a new symbol among Δ. Its role is to allow defining a function whose name does not clash with existing ones, as well as tracking all the already defined functions.

Therefore, suppose the following code in the Move language:

```
// Move source code
let a: u64 = ...;
while (a > 0){
    a = a - 1;
    ...
}
a == 0; // true
```

Since `a` is a free variable in the body of the loop, it has to be passed as function argument as a mutable reference. Its high level translation is therefore the following. Recall how this translation produces Move code that still type-checks and compiles rather than introducing intermediate representations:

```
// Move code
let a: u64 = ...;

// With while_0 : &mut Int -> ()
mapped_while_0(&mut a);
a == 0; // true
```

We now proceed by describing how the `while` construct itself is translated as a function. First of all, the looping condition is mapped into a conditional check at the start of the function body. By continuing the above compilation, it is possible to see the initial condition checked, and if it evaluates to true, a single iteration of the loop is performed, then the function calls itself recursively in order to proceed with the next iteration:

```
// Move code
fun mapped_while_0(&mut u64){
    if(*a > 0){
        *a = *a - 1;
        ...
        mapped_while_0(a)
    }
}
```

Now, to consider `break` and `continue`, their behavior is that both of them cause to skip any subsequent instruction, until reaching the end of the loop body. At that point, a `break` simply exits the loop, while a `continue` causes the execution to start again.

We have implemented both constructs with two boolean flags that track if either one of them has been encountered during the execution. Every expression that follows a break or continue is then wrapped inside an if statement, in order to skip them if needed. Suppose the following loop body:

```
// Move source code
while(...){
    ...
    break;
```

```
    ...
}
```

As said, everything that follows the `break` should be wrapped in a conditional expression. Moreover, at the end of the loop body the function invokes itself recursively only if a break has never been encountered:

```
// Move code
fun mapped_while_0(...){
    if(...){
        // Flags
        let break_hit = false;
        let continue_hit = false;
        ...
        // A break is encountered
        break_hit = true;
        // Wrapping subsequent expressions
        if(!break_hit && !continue_hit){
            ...
        }

        // Recursive call
        if(!break_hit) mapped_while_0(...)
    }
}
```

The above logic works for multiple occurrences of `break` and `continue`, with the output code resulting slightly more verbose in case they occur in nested code blocks.

The translation logic we have described allows to compile any occurrence of `while` constructs into recursive calls, additionally handling any variable that can be accessed or mutated inside the construct itself and possible occurrences of `break` and `continue` expressions that alter the execution flow.

We now consider the case of the `loop` construct. As said, this construct simply repeats a certain block of code forever, or until a `break` is encountered. Its behavior is therefore equivalent to a `while` loop with a condition that always evaluates to true, and therefore it is mapped as such, delegating the compilation to the logic seen before:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{loop}\ e \longrightarrow \texttt{while(true)}\ e}$$

4.5 Type witness

We have already mentioned in a previous chapter how the Aptos blockchain stores both resources and modules under the global storage; for easier reading, we recall the definition that we have given for the global storage, considering only the part relevant for storing resource data:

$$\text{GlobalStorage} := \begin{cases} \text{resources} & : & \text{Address} \times \text{ResourceType} \to \text{ResourceValue} \\ \text{modules} & : & \dots \end{cases}$$

This creates a particular problem when trying to translate the same behavior in Cardano. In fact, it requires to know at runtime the actual type of the resource that we want to access.

This is evident when considering the `move_from` operator, which obtains the information about the type of the resource that it is requested by just means of the type parameter T:

$$\texttt{move\_from<T>}(address) : T$$

In literature, two approaches are commonly used to carry runtime information about types: template instantiation, also known as monomorphization, and type witnesses.

Monomorphization is known particularly for being implemented in the `C++` language. In `C++`, the compiler generates monomorphic versions of a polymorphic function or class for each possible type that it is instantiated with. This has the effect of producing code optimized for each concrete type and no runtime overhead, at the expense of a larger binary output and longer compilation times.

For our case, the biggest downside of implementing monomorphization comes from the fact that a larger output code would result in a bigger Aiken validator and therefore in a larger transaction size, thus increasing the fees that have to be paid and possibly hitting the maximum transaction size allowed by Cardano.

The second approach, type witnesses, consists instead in keeping a single version of the function, and extending it with additional parameters, each acting as a witness for the corresponding type parameter. Upon function invocation, the compiler therefore has to pass additional arguments respecting the type arguments that are passed to the function.

Implementing type witnesses comes with the downside of a slight runtime overhead to maintain type information as variables, while also constructing and passing those information to nested function calls. However, the complexity to handle type witnesses at runtime is lower and, mainly, it does not affect the size of the output code by much, apart for few parameter definitions. With this explained, we have chosen to track generic functions by leveraging the type witness approach.

Specifically, consider a parametric polymorphic function accepting k type arguments:

$$\forall \alpha_1, \ldots, \alpha_k.\ (\tau_1, \ldots, \tau_n) \to \tau_r$$

The compiler produces a corresponding function that accepts k more parameters, typed with an *Intermediate Representation* type ω:

$$\forall \alpha_1, \ldots, \alpha_k.\ (\tau_1, \ldots, \tau_n, \underbrace{\omega, \ldots, \omega}_{k \text{ times}}) \to \tau_r$$

Consider now the possible cases that can happen when a function is invoked with a type argument. That type argument can either be a type constructor, eventually followed by nested type arguments, or a type parameter itself, as in the following example in Move:

```
// Move source code
// Type constructor
foo<MyResource>(...);

// Type parameter
foo<T>(...);

// Type constructor with type arguments
foo<AnotherResource<T, MyResource>>(...);
```

Therefore, we have extended the Move syntax with a non terminal W that is used to express type witnesses; it is composed by two variants, one for type constructors and the other for type parameters:

τ	$:=$		**types**
	$\mid$	$\ldots$	
	$\mid$	ω	type witness types
E	$:=$		**expressions**
	$\mid$	$\ldots$	
	$\mid$	W	type witness expressions
W	$:=$		**type witness expressions**
	$\mid$	$\text{tw\_c}\langle s, W^* \rangle$	type constructors
	$\mid$	$\text{tw\_i}\langle s \rangle$	type parameters

Table 4.1: Move syntax extended with IR type witness

The former acting as witness for concrete types and the latter for type parameters.

When a function invocation is encountered, depending on the case of each type argument, the corresponding type witness is instantiated. Formally, we want to translate the type argument α to the type witness W:

$$\Gamma \vdash \alpha \longrightarrow W$$

We can therefore define two inductive rules for the translation:

$$\frac{x \in \text{TypeVars}(\Gamma)}{\Gamma \vdash x \longrightarrow \text{tw\_i}\langle x \rangle}$$

$$\frac{\Gamma \vdash \alpha_1 \longrightarrow w_1 \quad \ldots \quad \Gamma \vdash \alpha_k \longrightarrow w_k}{\Gamma \vdash C\langle \alpha_1, \ldots, \alpha_k \rangle \longrightarrow \text{tw\_c}\langle C, [w_1, \ldots, w_k] \rangle}$$

The first one translates a type variable x present in the translation context Γ, while the second is used to translate a type constructor, along with its own type arguments that might recursively be one of the two cases.

Some examples involving type witness will be shown in later sections.

4.6 Global storage and Program scope

4.6.1 Continuation-passing Style

An important aspect to consider when translating imperative code into functional is immutability. Functional programming paradigm is based on the concept that there is no state that can be mutated in-place, but rather any modification has to consume the old value and produce a new one.

Being Aiken a functional programming language, it enforces immutability;his creates an obstacle not only when dealing with mutations on the Move global storage, but also when the Move program performs assignments or in-place mutations via the usage of references.

Consider for example the following code in Move, where multiple lines can cause the state to be mutated, either explicitly by an assignment, or implicitly by passing a mutable reference to a function:

```
...
let a: u64 = ...;
a = a + 1;

let r: &mut u64 = &mut a;
update_value(r);
...
```

In literature, a possible solution to implement state mutation in an immutable context is by leveraging *Continuation-passing Style* (CPS). In CPS, the control flow is passed explicitly in the form of continuations, meaning functions that are passed to high order functions and are manually invoked in order for the computation to proceed.

A function written in Continuation-passing style accepts therefore an additional argument, the continuation, which is a function itself that accepts a single argument:

$$f_{CPS} : A \to \underbrace{(B \to R)}_{continuation} \to R$$

When the CPS function has computed its resulting value, it can return it by calling the continuation with that value as the argument.

For our case, we need to track mutations on both the global storage and on program variables, therefore we have defined the type of the continuation parameter as a tuple:

$$\text{CPS} := \text{PS} \times \text{GS}$$

Where *PS* represents the scope available at any point in the program, and *GS* is the global storage of the blockchain as seen in the previous chapter. Their actual types will be shown later in this section.

4.6.2 Program scope and operators

Regarding the program scope, consider that in Move, like in any imperative programming language, it behaves like a stack where each block of code and function invocation adds a new scope on top of the existing ones, populating it with local variables, and that scope is later popped when exiting the code block that pushed it onto the stack, since its variables are no longer accessed.

In order to correctly compile the program and leaving unaltered its behavior, it is needed to respect the stack behavior of the Move scope. For example, reusing a variable name for a variable declaration in an inner code block causes the inner binding to shadow the outer one, as normally expected, until the control flow exits the inner code block and therefore removes the shadowing variable from it.

For these reasons, we have designed and implemented the program scope as a list of individual scopes:

$$\text{PS} \coloneqq [\text{Scope}]$$

While also implemented two operators that allow pushing a new scope to it and popping the topmost one. Being PS an opaque type for the compiler, those operators work instead on the whole CPS type, returning the updated version:

$$\textit{push\_scope} : \text{CPS} \to \text{CPS}$$
$$\textit{pop\_scope} : \text{CPS} \to \text{CPS}$$

The motivation for making those operators work on the CPS type rather than PS is mainly for lowering the complexity of the output Aiken program when dealing with both the global storage and the program scope.

Each block of code, that the Move compiler internally calls a Sequence, will therefore perform a *push_scope* as its first operation and then a *pop_scope* as the last one.

Since each Scope should associate any variable name, also called *Identifier*, with its corresponding value, an initial definition for the Scope could be a mapping between an identifier and a single value. However, an important consideration is that the Move language allows to rebind variables that already exist in the current scope. This acts like a shadowing, where further occurrences of that variable name always refer to the latter binding. However, the shadowed binding can still be accessed by the use of references, like for traditional shadowing behavior.

This means that the Scope should associate each identifier with a list of values, where each element of the list is a single bind occurrence.

$$\text{Scope} \coloneqq \text{Identifier} \to [\top]$$

However, it is important to mention a particular constraint that the Aiken type checker enforces on container data structures, such as a list $[\tau]$; those data structures, in fact, can contain only values that can be serialized; the motivation for this constraint is to allow any container data structure to be encoded inside a transaction body.

Therefore, the Aiken language provides a particular type, Data, that is used to represent any serializable value. We can leverage this Data type to formalize a typing rule that restricts elements of a list to be serializable:

$$\frac{T \leq \text{Data}}{\Gamma \vdash \text{nil} : \text{List}\langle T \rangle} \qquad \frac{\Gamma \vdash v : T \quad T \leq \text{Data} \quad \Gamma \vdash l : \text{List}\langle T \rangle}{\Gamma \vdash \text{cons}(v, l) : \text{List}\langle T \rangle}$$

Therefore, we are restricted to only insert serializable values in the scope, making it necessary to rewrite the definition of the Scope in terms of the supertype Data:

$$\text{Scope} \coloneqq \text{Identifier} \to [\text{Data}]$$

While at first this might not seem a particularly impactful aspect, its presence caused some issues later on when designing a translation logic for references, which will be explained in a later section.

Scope operators Now that we have given a definition for the program scope, we proceed by explaining which operations have to be translated as manipulation of it. Recall that all operators are defined to work over the whole CPS.

We will start by assuming that all variables present in the Move program need to be inserted into the program scope in the Aiken translation; later on, we will provide an optimization to this logic.

When a variable declaration is encountered, its identifier should be pushed onto the topmost scope. We therefore define a $\mathscr{S}_{\text{POST}}$ operation on the scope, that accepts the name of the variable along with an optional initialization expression:

$$\mathscr{S}_{\text{POST}} : \text{Identifier} \times \text{Maybe Expr}^{1} \times \text{CPS} \rightarrow \text{CPS}$$

The $\mathscr{S}_{\text{POST}}$ operation should additionally preserve the shadowing behavior of Move rebinds, meaning that if a variable with the same name already exists in the same scope, it should be shadowed instead of reinitialized or replaced by the new one.

A `let` binding is therefore translated as a $\mathscr{S}_{\text{POST}}$ operation on the program scope:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{let } x = e \longrightarrow \texttt{let cps = } \mathscr{S}_{\texttt{POST}}(\texttt{"x"},\ e,\ \texttt{cps})}$$

Performing an assignment requires instead that a binding occurrence of that identifier exists in the program scope and if so, the topmost and most recent occurrence should be updated. We have called this expression $\mathscr{S}_{\text{PUT}}$:

$$\mathscr{S}_{\text{PUT}} : \text{Identifier} \times \text{Maybe Expr} \times [\text{Field}] \times \text{CPS} \rightarrow \text{CPS}$$

An additional parameter of opaque type [Field] can be observed, whose presence has not been explained yet. The motivation is that an assignment can be performed not only on a variable, but also on a nested field of a struct, and is therefore needed to know which fields should be accessed. The actual definition for the Field type has been determined mainly by restrictions imposed by the Aiken language on struct manipulation, and will be explained later for completeness.

An assignment is therefore translated as a $\mathscr{S}_{\text{PUT}}$ operation on the scope:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash x = e \longrightarrow \texttt{let cps = } \mathscr{S}_{\texttt{PUT}}(\texttt{"x"},\ e,\ [],\ \texttt{cps})}$$

$$\frac{\Gamma \vdash x : s\langle\vec{\tau}\rangle \qquad \Gamma \vdash e : \tau \qquad \Gamma, x : s\langle\vec{\tau}\rangle \vdash f_i \rightarrow f_i' : \text{Field}}{\Gamma \vdash x[.f_1, \ldots, .f_n] = e \longrightarrow \texttt{let cps = } \mathscr{S}_{\texttt{PUT}}(\texttt{"x"},\ e,\ [f_1', \ldots, f_n'],\ \texttt{cps})}$$

Note the writing regarding the access of nested fields f_i. We say that each field f_i, which is basically an identifier, has to be translated into a certain f_i' of type Field in order for the translation rule to be correct. The motivation for this is mainly technical, and will be explained later along the definition for the Field type.

Since variables are inserted in the program scope, any occurrence to them has to be rewritten as a $\mathscr{S}_{\text{GET}}$ operation from the CPS. Since this operator does not alter the program scope, there is also no need to return it, so to lower the verbosity of the produced Aiken code:

$$\mathscr{S}_{\text{GET}} : \text{Identifier} \times \text{CPS} \rightarrow \text{Data}$$

It is important to notice the return type of the $\mathscr{S}_{\text{GET}}$ operator as the serializable supertype rather than the actual type of the variable. The reason for this is for a constraint imposed

[1] The *Maybe e* is the Haskell monad used to represent the possibility of some value of type *e* or the absence of it

by the type definition of the Scope: being it a mapping between an identifier and a list of possible values of any type Data, accessing a variable from the scope must naturally produce a value of type Data. However, the Aiken code needs the correct subtype in order to type check successfully.

This can be easily achieved by leveraging the type deduction helper to retrieve the expected type of a variable occurrence, and then performing a downcast to that type:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \longrightarrow \mathscr{S}_{\text{GET}}(\texttt{"x", cps}) \texttt{ as } \tau}$$

We have defined the base operators that act on the program scope in order to replace some functionalities of the Move language that are not present in the functional Aiken language, mainly variable and field mutations. Nevertheless, Move also supports references, that can be either immutable or mutable, depending if the Move type system allows mutating the referenced value.

The description of how references can affect both the program scope and the global storage is explained in a dedicated section, as well as how we have designed and implemented specific operators to handle them and their effects.

As said previously, we just assumed that all variables present in the Move source code need to be inserted into the program scope. However, there exist certain conditions for which a certain variable might be represented by a simple binding in the Aiken language without the need of adding it to the program scope, therefore lowering the verbosity of the generated code and improving its runtime performances. This optimization is possible only if the variable respects three conditions:

- It is never mutated in any possible execution path of the program
- It has no references to it, neither immutable
- Is not a reference itself

If those conditions hold, then that variable is used basically in an immutable and functional setting, and therefore can be translated by a simple `let` binding in the Aiken code. Our compiler checks for this optimization during translation, therefore simplifying the output Aiken code whenever possible.

An interesting result given by this optimization is that allows a Move program, when written applying the concept of immutable variables and forbidding references, to be compiled to an Aiken code that requires little access to the program scope, therefore being objectively less verbose and more performant. Since transaction fees in Cardano depend also on the size of the validator and execution steps performed, this optimization is also useful to reduce them, and writing the source Move program with a functional mindset helps lowering the fees even more.

We have previously introduced the Field type that has the role of tracking any nested fields that are being assigned in the Move program, and later on we will show other cases in which it is needed. Here, we want to clarify how tracking the fields have been achieved in the compiled Aiken code.

Our first approach was to track the name of the fields, thus defining the Field type as an alias for a list of identifiers. Suppose an assignment to a nested field of a variable:

$$\mathscr{S}_{\text{PUT}}(\texttt{"x"}, e, [f_1], \texttt{cps})$$

The $\mathscr{S}_{\text{PUT}}$ operator would first have to access the `"x"` variable from the CPS, then extract the field named f_1, assign it the new value e, and finally return the updated program state.

However, we encountered a problem where, where since `"x"` comes from the program scope, it is actually typed as Data. In order to access its fields by name, it is needed to first downcast that value to the correct type. This would require to either know the correct type at compile time, or to perform a runtime casting. The first approach is not feasible, for the same reasoning that caused the need of the type witness mechanism, and the second approach, while allowed in other languages such as Java via means of reflection or dynamic types, is not supported in Aiken. Therefore, we found that accessing a nested field of a Data type by name was not feasible. Instead, the Aiken language provides a library that allows to manipulate data constructors by accessing their fields by index rather than name. Specifically, the builtin `unconstr_fields`:

$$unconstr\_fields : \text{Data} -> [\text{Data}]$$

Takes any serializable value, interprets it as a data constructor and returns its fields, following the order they have been declared.

Tracking the indices of the fields at compile time is feasible, even if slightly more complex then tracking their name. What it requires is that the traversal needs to know how each struct is defined, interpreting it as a data constructor with ordered fields, and retrieve the correct index for each field that is accessed in any point of the Move program. This is exactly the motivation of saying that a field named f_i has to be translated into a certain f_i' of type Field.

Regarding the Field type itself, it can be therefore defined as an alias for an integer, indicating not the name of the accessed field but rather its index in the declaration of the corresponding struct:

$$\text{Field} := [\text{Int}]$$

4.6.3 Direct-style CPS

Now that we have given a definition for the type of the continuation parameter CPS, we will explore how the CPS can be implemented in Aiken without the need to rewrite the entire code as a series of continuation passing.

Consider two functions written in CPS:

$$f : A \to (\text{CPS} \to B) \to B$$
$$g : X \to (\text{CPS} \to B) \to B$$

Invoked inside an expression:

$$\lambda k.f\ a\ (\lambda x.g\ x\ k)$$

This can be interpreted as an invocation to f that produces, other than an updated CPS state, a new variable named x that can be used as a bounded variable in the continuation. In a direct-style control flow, this can be rewritten as a binding occurrence of x initialized as the resulting value from f. Then, the computation proceeds with g being able to use this new variable:

```
let x = f(a) in
g(x)
```

This means that if a programming language supports *let in* bindings, it is possible to implement CPS without the need to explicitly pass continuations, something that would make the code less readable and possibly harder to understand. In fact, Aiken supports *let in* bindings,

so we have leveraged this feature to produce a compilation that still resembles an imperative flow, while leveraging CPS to propagate along with the computation any update to the global storage and program scope.

Suppose the following invocation:

```
// Move source code
let a = my_function(...);
```

The compiler adds a final CPS argument, but also has to handle the returned CPS, therefore the binding now becomes:

```
let (a, cps) = my_function(..., cps);
```

In this way, the updated CPS can be forwarded further in the program code.

However, we have encountered an additional complexity since function invocations can also be specified inside expression nodes, like in the following example:

```
// Move source code
let a = 1 + my_function(...);
```

Handling this case requires to extract the function invocation in a temporary binding, capturing its return value in a temporary variable and substituting the original function call with that variable:

```
let (temp, cps) = my_function(..., cps);
let a = 1 + temp;
```

Also notice that there is no need to insert `temp` into the program scope, since it is just an helper variable that is used exactly once and never mutated.

An important consideration is to ensure that this extraction would result in the same computation order of the original Move program. In fact, changing the execution order for temporary bindings could update the CPS in a different order that the original code, therefore diverging from the intended execution.

However, the correctness of the translation is ensured by how the AST is constructed and traversed. In the AST, an inner node expression represents an expression that will be evaluated before the parent node, either being it determined by operator precedence or by parenthesis. Additionally, multiple children of the same node are assumed to be evaluated from left to right. This order is exactly the post order traversal performed by the compiler, meaning that each temporary binding will be added respecting the computation order that the Move compiler would have performed.

Now that we have described how functions are translated, we apply a similar reasoning to sequences. In fact, modifications of either the global storage or the program scope can not only happen when invoking functions, but also inside nested code blocks.

We have therefore designed this compilation step to also analyze code blocks. If an intermediate operation that modifies either the global storage or the program scope is found inside a sequence, the entire sequence is marked as (possibly) mutating the state. A marked sequence if therefore handled in the same way of a function invocation, with its return value handled in a temporary binding. For example, in the following code we show an inlined code block that performs an update to the program scope:

```
a + {
    ...
    IRPutScope ...
    ...
} + c
```

The compiler therefore extracts the sequence in a temporary binding, replacing its occurrence with the temporary variable:

```
let (temp, cps) = {
    ...
    IRPutScope ...
    ...
}
a + temp + c
```

This still follows the rules explained previously for how functions are handled. However, a small consideration has to be performed for specific language constructs such as the if-then-else. In that construct, it is the result of the if guard that determines which branch will be executed, while extracting both of them, as in the following example, would cause both branches to be executed, naturally diverging from the intended computation:

```
// Wrong: only one branch should be computed
let (if_result, cps) = {...};
let (else_result, cps) = {...};

let res = if (cond) if_result else else_result;
...
```

Instead, a specific handling for the if-then-else has been implemented. Specifically, it wraps the entire construct in a temporary binding, allowing for just one of the branches to have effects on the CPS:

```
let (res, cps) = if (cond) {
    ...
    (if_result, cps)
}
else {
    ...
    (else_result, cps)
}
...
```

4.6.4 Global storage and operators

Similarly to how certain operations such as bindings and assignment perform mutation on the program scope, we have seen that Move provides few operators to modify the global storage.

Recall how we have defined the global storage as a mapping between an address and the type of the resource and its actual value:

$$\text{GS} := \text{Address} \times \text{ResourceType} \rightarrow \text{ResourceValue}$$

Concretely, the compiled data structure has to work on the type witness to handle the type of the resource and can store values of any serializable type; the definition therefore becomes:

$$\text{GS} := \text{Address} \times \omega \rightarrow \text{Data}$$

Following a reasoning similar to the program scope, we have defined and implemented a set of IR nodes with corresponding custom Aiken library to handle the global storage operators. To lower the complexity of the output Aiken program, these operators work on the entire CPS rather than on the GS, similarly to the operators defined for the program scope; additionally, each operator has to leverage the type witness in order to know the type of the resource at runtime and work on the CPS, and in order to respect the behavior of the original Move operators they should also return both the original value and the CPS.

Consider the *move_to* operator used to publish a resource under a certain address:

```
move_to<T>(&signer, T): ()
```

We have implemented a corresponding Aiken operator that along with the CPS returns its original `Unit` type. This operator behaves exactly like the original one, also regarding the abort condition in the case the resource already exists under that address:

$$\mathscr{G}_{\text{move\_to}} : \text{Ref<Signer>} \times \omega \times \text{Expr} \times \text{CPS} \rightarrow (\text{Unit}, \text{CPS})$$

Notice how the operator is not parametric polymorphic, but rather accepts any Expr as resource value, that in the library code is simply an alias to the supertype Data. A detailed explanation on the motivations that required rewriting the parametric polymorphism is given in a later section.

An occurrence of *move_to* is therefore translated with an intermediate representation node as follows:

$$\frac{\Gamma \vdash s : \text{Ref<signer>} \qquad \Gamma \vdash v : \tau \qquad \Gamma \vdash T' \longrightarrow T'}{\Gamma \vdash \texttt{move\_to<T>}(s,\ v) \longrightarrow \mathscr{G}_{\text{move\_to}}(s,\ T',\ v,\ \texttt{cps})}$$

Where `T'` is the type witness resulting from the compilation of the type argument `T`.

The operator for the *move_from* follows a similar reasoning, with the exception that it can return a value of any type, due to the definition of the GS:

$$\mathscr{G}_{\text{move\_from}} : \text{Address} \times \omega \times \text{CPS} \rightarrow (\text{Data}, \text{CPS})$$

The corresponding compilation rule additionally needs a casting to the correct resource type, similar to the $\mathscr{S}_{\text{GET}}$ operator seen previously:

$$\frac{\Gamma \vdash a : \mathit{address} \qquad \Gamma \vdash T \longrightarrow T'}{\Gamma \vdash \texttt{move\_from<T>}(a) \longrightarrow \mathscr{G}_{\text{move\_from}}(a,\ T',\ \texttt{cps})}$$

Regarding borrowing resources from the global storage, the distinction between a mutable and an immutable borrow is relevant only for the Move type checker, meaning that we can compile both cases to a single operator. Additionally, since this operator does not modify the global storage, we can simplify the generated Aiken code by simply returning the requested reference instead of wrapping it into a CPS tuple:

$$\mathscr{G}_{\text{borrow\_global}} : \text{Address} \times \omega \rightarrow \text{Ref<}\tau\text{>}$$

Notice two things: first, this operator does not accept nor return the CPS parameter, second, the returned value is parametric polymorphic rather than constrained as a reference to the Data supertype, which might appear unexpected due to how we have defined the GS, but intuitively comes more useful since it avoids the need of an explicit casting. The reason why we can make the reference polymorphic will be shown later when describing how references are implemented, but in order to give an intuition, we have implemented references in Aiken in such a way that they do not actually need to access the resource they are pointing unless dereferenced, and therefore the Aiken type checker does not require them to be typed as Ref<Data>. For the same reason, this operator does not need to know the CPS at all, therefore we do not include it in its definition in order to lower the verbosity of the compiled Aiken code. The compilation then is relatively straightforward and identical both for mutable and immutable borrows:

$$\frac{\Gamma \vdash a : \mathit{address} \qquad \Gamma \vdash T \longrightarrow T'}{\Gamma \vdash \texttt{borrow\_global<T>(}a\texttt{)} \longrightarrow \mathscr{G}_{\text{borrow\_global}}(a,\ T')}$$

$$\frac{\Gamma \vdash a : \mathit{address} \qquad \Gamma \vdash T \longrightarrow T'}{\Gamma \vdash \texttt{borrow\_global\_mut<T>(}a\texttt{)} \longrightarrow \mathscr{G}_{\text{borrow\_global}}(a,\ T')}$$

The last operator is the *exists*, which returns a boolean value if a certain address owns a specific resource or not; intuitively, it requires the CPS parameter, but since it does not perform any mutation on it, there is no need for it to be returned:

$$\mathscr{G}_{\text{exists}} : \text{Address} \times \omega \times \text{CPS} \to \text{Bool}$$

Again, its translation is straightforward:

$$\frac{\Gamma \vdash a : \mathit{address} \qquad \Gamma \vdash T \longrightarrow T'}{\Gamma \vdash \texttt{exists<T>(}a\texttt{)} \longrightarrow \mathscr{G}_{\text{exists}}(a,\ T')}$$

4.6.5 References

In order to complete the compilation step regarding global storage and program scope, it is needed to handle one last functionality provided by the Move language, which are the references.

In Move, a value of any type `T` can have one or more references that point to it; references can be defined either as immutable or mutable depending on what actions are allowed on them by the type checker; in fact, an immutable reference `&T` supports the dereference operator `*` when appearing as a right value in order to return the value of type `T` of the referenced variable, while mutable references `&mut T` extend them by also allowing to assign a new value to the referenced variable, as long as the type of the right value remains `T`. The syntax for managing references in Move is similar to C-like pointers:

```
// Move source code
let a: u64 = 1;
let r: &u64 = &a;

let b: u64 = 2;
let r_mutable: &mut u64 = &mut b;

*r_mutable = *r_mutable + 1;
*r + *r_mutable
```

While being commonly present in many imperative programming languages, references are often purposely limited in functional languages, particularly regarding their mutability, and specifically Aiken does not support them at all, creating the need for translating Move references in some alternative way.

An initial attempt was tried to implement references through closures, meaning functions that refer to variables declared in an outer scope and can therefore access them in a second time. For example, the following code in Aiken accesses the outer variable `a`, making the final predicate hold:

```
test closure_test() {
  let a: Int = 12

  let closure = fn(b: Int) { a + b }

  closure(1) == 13 // True
}
```

However, we have encountered few problems when examining this solution; the most evident is that while closures work for returning the value of a variable, and thus for representing dereferences happening on the right value, there is no clear way to leverage them to assign a new variable to the captured variable. Even performing a rebind of the *a* variable would in fact simply shadow the previous occurrence, leaving the closure unaltered.

The second problem comes from the fact that reference variables can themselves be mutated, therefore needing to be inserted into the program scope along every other variable; while this is not a problem by itself, recall how the Aiken language imposes that container data structures can only hold serializable values; this poses a problem since closures, and more generally functions, are not serializable in Aiken, meaning that representing a reference by means of closures would prevent it from being inserted in the program scope, thus requiring another approach to represent mutations on that reference.

We have therefore discarded the above approach, preferring instead to represent a reference via a custom type with all the necessary information to correctly point a variable, or a value, on the CPS. Specifically, two types of references can be identified: references to program variables, created by using the `&` operator, and references to resources in the global storage, returned by the operators `borrow_global` and `borrow_global_mut`; therefore, we have extended the Aiken syntax so to express these two kinds of references:

$$
\begin{aligned}
\text{Ref<}\tau\text{>} &:= \text{RefKind} \times [\text{Field}] \\
\\
\text{RefKind} &:= \text{GSRef Address } \omega \\
&\mid \text{PSRef ScopePtr}
\end{aligned}
$$

Here, we define that a reference can either point to a resource published under the global storage, `GSRef`, in which case it is needed to know the address of the owner and the type of the resource, by means of type witness as already seen, or otherwise points to a variable in the program scope, `PSRef`, in which case we track some specific data, that mainly depends on the data structure chosen for the scope and therefore we have not reported for simplicity.

In both cases, a reference can point to inner fields of a struct, so there is the need to track them via the [Field] information. Note how instead there is no need to distinguish if a reference

is mutable or not, since this distinction is only useful for the Move type checker in order to ensure that a reference can mutate a value only if explicitly allowed, but for compilation purposes it is not a meaningful information, since we assume that any input Move program type checks correctly.

We then identify three possible operations that involve references. The first one is, naturally, creating a reference to a variable of any type; we name if $\mathscr{R}_{\text{make}}$:

$$\mathscr{R}_{\text{make}} : \text{Identifier} \times [\text{Field}] \times \text{CPS} \rightarrow \text{Ref<}\tau\text{>}$$

The concrete type for the reference is determined at compile time by leveraging the type deduction system, and depends on the type of the variable that is being referenced:

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash \texttt{\&[mut]}\ x \longrightarrow \mathscr{R}_{\text{make}}(\texttt{"x"},\ [],\ \texttt{cps})\ :\ \texttt{Ref<}\tau\texttt{>}}$$

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma, x : \tau \vdash f_i \rightarrow f_i' : \text{Field}}{\Gamma \vdash \texttt{\&[mut]}\ x.f_1[.f_2, \ldots, .f_n] \longrightarrow \mathscr{R}_{\text{make}}(\texttt{"x"},\ [f_1', \ldots, f_n'],\ \texttt{cps})\ :\ \texttt{Ref<}\tau\texttt{>}}$$

Regarding the dereference operator, first of all we consider dereferences on the right value; in this case, consider that any Ref<τ> points either to a variable on the program scope or a resource on the global storage, both of which are typed as Data; therefore the dereference operator has to return a value of type Data:

$$\mathscr{R}_{\text{GET}} : \text{Ref<}\tau\text{>} \times \text{CPS} \rightarrow \text{Data}$$

This implies that the corresponding translation must include a casting to the correct type, which can be deduced at compile time:

$$\frac{\Gamma \vdash x : \texttt{Ref<}\tau\texttt{>}}{\Gamma \vdash *x \longrightarrow \mathscr{R}_{\text{GET}}(\texttt{"x", cps})\ \texttt{as}\ \tau}$$

The final operation is when a dereference occurs on the left value, corresponding to assigning a value to either the referenced value or one of its nested fields. Since the corresponding operation for normal variables has been named $\mathscr{S}_{\text{PUT}}$, we have defined the $\mathscr{R}_{\text{PUT}}$ operation:

$$\mathscr{R}_{\text{PUT}} : \text{Ref<}\tau\text{>} \times \text{Data} \times [\text{Field}] \times \text{CPS} \rightarrow \text{CPS}$$

The translation rules are similar to those of the $\mathscr{S}_{\text{PUT}}$ operator, and basically consist in leveraging the CPS to carry on the updated program scope and global storage, one of which has been mutated by the operation:

$$\frac{\Gamma \vdash x : \text{Ref<}\tau\text{>} \qquad \Gamma \vdash e : \tau}{\Gamma \vdash *x = e \longrightarrow \texttt{let cps =}\ \mathscr{R}_{\text{PUT}}(x,\ e,\ [],\ \texttt{cps})}$$

$$\frac{\Gamma \vdash x : \text{Ref<}s\langle\vec{\tau}\rangle\text{>} \qquad \Gamma \vdash e : \tau \qquad \Gamma, x : \text{Ref<}s\langle\vec{\tau}\rangle\text{>} \vdash f_i \rightarrow f_i' : \text{Field}}{\Gamma \vdash x[.f_1, \ldots, .f_n] = e \longrightarrow \texttt{let cps =}\ \mathscr{R}_{\text{PUT}}(x,\ e,\ [f_1', \ldots f_n'],\ \texttt{cps})}$$

4.7 Polymorphism

We have seen how the usage of type witness solves the problem of knowing at runtime the type arguments that are passed to any given function; the motivation for this need is that the global

storage is defined in terms of the type of each resource and therefore needs it at runtime so to access the resource.

However, parametric polymorphism in the Move source code causes few other problems when compiling it to Aiken code; in particular, the Aiken type checker does not support at all any casting involving parametric types, in any direction:

```
// Aiken code
// With as_data builtin function of type Data -> Data

// Casting to a parametric type is not allowed
expect a: t = ...
expect a: MyStruct<t> = ...
// Neither for the serializable supertype
expect a: t = as_data(...)

let some_t: t = ...
// Casting from a parametric type is not allowed
expect a: u64 = some_t
expect a: MyStruct<u64> = some_t
// Neither for the serializable supertype
expect a: Data = some_t
```

We suspect that the conversion toward a generic type is not supported for a soundness reason, in fact, a type system allowing to cast a concrete type into a generic type T would probably not be able to guarantee that the concrete value would be correct for any T, therefore possibly causing a runtime error in the case the resolved type is not compatible with the actual value, as in the following pseudocode:

```
// Pseudocode
let a: Int = 42;
let b: T = a as T;
let c: String = b as String; // Error
```

On the opposite direction for the casting, conversion from a generic type is instead not supported since, as stated by the Aiken developers, the type checker needs to know the full type in order to perform all the structural checks that are needed (KtorZ (2026)).

While these might appear as non-impactful cases, the problem arises mainly when dealing with both the global storage and the program scope; in fact, recall how both of them are defined in terms of the serializable supertype Data, as required by the constraint that the Aiken type checker imposes on container data structure:

$$\text{Scope} \coloneqq \text{Identifier} \to [\text{Data}]$$

$$\text{GS} \coloneqq \text{Address} \times \omega \to [\text{Data}]$$

This means that all the operators that work on them must not be type parametric, otherwise they would incur in the casting errors just seen; for example, suppose a generic implementation of the *move_to* operator in Aiken pseudocode:

```
pub fn move_to(signer: Ref<Signer>, resource_type: IRTypeWitnessT,
    resource_value: t, cps: CPS) -> (Unit, CPS){
    let addr: address = get_address(signer)

    // Type error: expected Data, found t
    let cps = insert_key_value((address, resource_type),
        resource_value, cps)

    cps
}
```

This requires the operators to be defined themselves on the Data supertype rather then being type parametric, however, this naturally delegates the problem to the caller function rather than solving it; the solution we have implemented is therefore to compile parametric polymorphism into subtype polymorphism, specifically by replacing each occurrence of type parameters into the supertype Data, both on function signature and body, and to commit to a concrete type only when actually needed.

The supertype Data is replaced for any occurrence of a type parameter, also when occurring as type argument itself:

$$\frac{\tau \in \text{TypeParameters}(\Gamma)}{\Gamma \vdash \tau \longrightarrow \text{Data}} \qquad \frac{\tau \notin \text{TypeParameters}(\Gamma)}{\Gamma \vdash \tau \longrightarrow \tau} \qquad \frac{\Gamma \vdash \tau_1 \longrightarrow \tau_1' \quad \ldots \quad \Gamma \vdash \tau_n \longrightarrow \tau_n'}{\Gamma \vdash C\langle \tau_1, \ldots, \tau_n \rangle \longrightarrow C\langle \tau_1', \ldots, \tau_n' \rangle}$$

Meaning that any function signature is rewritten accordingly, taking into consideration that introduces itself new type names s^*:

$$\frac{\Gamma \cup s^* \vdash \vec{\tau} \longrightarrow \vec{\tau}' \qquad \Gamma \cup s^* \vdash \tau_r \longrightarrow \tau_r'}{\Gamma \vdash s*.\ (\vec{\tau} \to \tau_r) \longrightarrow (\vec{\tau}' \to \tau_r')}$$

To recap, we have converted the parametric polymorphism of the Move language into subtype polymorphism, mainly due to restrictions imposed by Aiken both on container data structures and casting on generic types. In this way, we can be sure that a portion of code that invokes operators on global storage and program scope does not cause issues for the Aiken type checker.

We have one last issue to resolve, which is still caused by the Aiken type checker. This time, casting involving the Data supertype as type argument are not allowed by Aiken, even if their presence is perfectly sound:

```
// With as_data builtin function of type Data -> Data

// Downcasting A<Data> into A<concreteType> is not allowed
expect b: A<Int> = A{b: as_data(12)}

// Upcasting A<concreteType> into A<Data> is not allowed
expect b: A<Data> = A{b: 12}
```

Aiken developers report that this has been a design choice so to lower the complexity of the type checker and solve multiple pitfalls that would have occurred otherwise. Moreover, the solution is particularly simple since the entire right value can be upcasted to Data and later downcasted again, operation that does not cause any runtime overhead since it only involves the type checker:

```
expect b: A<Int> = as_data(A{b: as_data(12)})

expect b: A<Data> = as_data(A{b: 12})
```

Now that we have shown how functions can be rewritten, we provide an example to how a function invocation is translated, so to give a visual indication of the produced Aiken code; suppose a generic Move function that is parametric on a generic type *T*, used both in a parameter and in the return type, and its corresponding invocation:

```
// Move source code
// With my_func : T => Foo<T> -> ... -> Bar<T>

let foo: Foo<u64> = ...;
let bar: Bar<u64> = my_func<u64>(foo, ...);
```

When compiled to Aiken, type witness are added along with the CPS, as seen in previous sections, and additionally it is translated to subtype polymorphism using Data. The produced Aiken code is undoubtedly more verbose, so a step-by-step explanation is provided along with the example:

```
// Aiken output code
// With my_func : Foo<Data> -> ... -> IRTypeWitnessT ->
                    CPS -> (Bar<Data>, CPS)

let foo: Foo<Int> = ...;

// The CPS is included in the function call
let (bar, cps): (Bar<Int>, CPS) = {

    // Being CPS opaque, each element of the tuple has to be
    // casted separately
    let (el_0, el_1): (Bar<Data>, CPS) = my_func({
        // The argument is casted to the expected type
        expect val: Foo<Data> = as_data(foo)
        val
    }, ..., IRTypeWitnessC("Int", []), cps) // Type witness

    // Return the tuple with correct type
    ({
        expect val: Bar<Int> = as_data(el_0)
        val
    }, el_1)
}
```

As one last mention, see the annotation regarding CPS being an *opaque type*. With opaque type we mean a type that hides its implementation detail in order to maintain some invariants. Specifically, CPS inherits its opacity from the Dict type defined the Aiken's stdlib, used as the data structure to represent both the GS and PS; the specific invariant maintained by a Dict is that a key can only be present once in the dictionary.

An opaque type has the constraint that it can not be downcasted from Data since it is not generally possible to know what invariants apply to that type; instead, to manipulate the opaque type it is needed to use the functions explicitly exported by the Aiken module that defines it.

The direct implication of this limitation is that the CPS can not be represented as Data and later downcasted, something that imposes a slight workaround to handle functions returning a tuple with the CPS as last parameter, specifically, in those cases the tuple needs to be unconstructed and each of its elements have to be downcasted separately, CPS excluded, and finally it is possible to construct again a new tuple and return it, as in the above example.

4.8 Post processing and Aiken generation

We have described the main steps that are needed in order to translate a Move project into a set of ASTs; those final ASTs have all the imperative functionalities and characteristics of the Move language rewritten for the Aiken language, and therefore each node corresponds to an Aiken code text in a nearly 1:1 mapping.

Before being able to generate actual Aiken files, the compiler performs few simpler rewriting that we have grouped in a single post processing step, whose purpose is to add the last details so to allow a clean and simple code generation.

Address resolution Move allows to define *named addresses*, that are basically placeholders for actual address values and can be used for naming modules, and subsequently referring to symbols in those modules. At compile time, the Move compiler loads the `Move.toml` file looking for associations between each named address and its actual value, replacing all occurrences of named addresses during the compilation.

The post processing step we have implemented performs the same work, loading the associations present in the `Move.toml` and performing a simple traversal of every AST in parallel by leveraging the `transformBi` function provided by the Haskell's *Uniplate* library, which provides a simple and performant way for traversing tree-like structures such as ASTs.

Library imports In addition to resolving named address, we briefly mention that this step also prepares and adds the necessary imports for the custom Aiken modules that our compiler provides, such as the modules that define the operators working on the global storage and program scope, as well as some other utility modules to handle shared types and helper functions.

Aiken generation When the compilation reaches this point, each AST is fully rewritten so to generate Aiken code that can be later compiled down to Untyped Plutus Core.

Producing the Aiken code from an AST is a matter of providing a simple mapping between each AST node and its textual representation; in order to provide better readability in the compiler source code, we have preferred not to deal manually with string interpolation but rather to leverage the Haskell's *Quasiquotation* functionality (contributors (2026b)) and the *NeatInterpolation* library (contributors (2026a)) that provides an easy abstraction layer on top of it.

Quasiquotation is a mechanism that allows programmers to use a custom, domain-specific syntax to construct fragments of a program; the *NeatInterpolation* library leverages this mechanism in order to provide and easy-to-use string interpolation functionality that allows to define a base *Text* template with placeholder variables that will be populated at runtime.

We have therefore leveraged the *NeatInterpolation* library to write the Aiken code for each AST node in a clear and concise way, abstracting away the need to manually handle string

concatenation and, main advantage, correct indentation of the output code. The actual mapping between each node and the textual Aiken code is omitted from this document for simplicity, if needed, it can be found on the repository of the project.

Validator generation We now describe the final step that is needed in order to generate a complete on-chain validator in the Aiken language. So far, the compiler has managed to translate ASTs corresponding to Move scripts and modules into Aiken modules, converting the imperative style of Move programs into the functional style of Aiken, and representing the account-based aspects of the Aptos blockchain into onchain eUTxO logic.

What it is needed then is a way to group together those Aiken modules in a single validator that can be used to spend UTxOs; as we have seen in a previous chapter, each validator has a `spend` purpose that is invoked when a transaction tries to spend an output of a previous transaction, and corresponds to a predicate whose evaluation determines if the output can be spent or not.

Moreover, the `spend` purpose allows to pass a custom data payload as additional parameter, called *Redeemer*; In a previous chapter, we have shown how the redeemer is designed as a Algebraic Data Type where each data variant is the identifier of a main function of a script, along with its parameters, so that the redeemer can then be pattern matched in order to perform the same computation of the original Move script.

At an high level, the validator is composed by the following parts:

- Define the type for the Redeemer
- Define a spend purpose where:
    1. Retrieve the input global storage GS from the input eUTxOs' datums
    2. Extend the input global storage by adding the assets from the input eUTsOs' values
    3. Retrieve and extend the expected output global storage GS'_E in the same way
    4. Prepare the CPS data structure by grouping the input global storage and an empty program scope $CPS = (\text{Empty}, GS)$
    5. Perform a pattern match on the Redeemer to invoke the correct entry point. Pass any redeemer argument and the CPS, collecting the resulting CPS CPS'. Additionally, retrieve and forward to the entry point any signer of the transaction, if needed.
    6. From the resulting CPS, extract the output global storage GS'
    7. Compare the expected output GS with the output GS, both for assets and non-asset data $GS'_E == GS'$
    8. Return the result of the comparison as the verdict
- Define a fallback handler to forbid any other transaction purpose

We have explored how the Redeemer is created and how the entire CPS computation happens; the initial and final steps are achieved by an Aiken library that we have designed, with the role to map the interface of eUTxOs into the familiar representation for the CPS.

An high level overview of a generated Aiken validation is therefore the following:

```
// Imports

pub type Redeemer {
```

```
    // redeemer variants
    case_i args_i
}

validator <name> {
    spend(
        datum: Option<SerializedGS>,
        redeemer: Redeemer,
        utxo: OutputReference,
        self: Transaction
    ){
        // Steps 1-2
        let gs = gs_from_eutxos(self.inputs)

        // Step 3
        let gs_e' = gs_from_eutxos(self.outputs)

        // Step 4
        let cps = (empty, gs)

        // Step 5
        let cps' = when redeemer is {
            // cases
            case_i(args_i) -> main_i(args_i, cps)
        }

        // Step 6
        let gs' = extract_gs(cps')

        // Step 7
        and {
            extract_non_assets(gs_e') == extract_non_assets(gs'),
            extract_assets(gs_e') == extract_assets(gs')
        }
    }

    else(_){
        false
    }
}
```

In this chapter, we have completed the compilation pipeline by addressing the final transformations required to bridge Move and Aiken. The post-processing phase refines the intermediate representation through lightweight yet essential updates, such as address resolution and import preparation, ensuring that the resulting ASTs are both consistent and ready for code emission. Building on this, the Aiken generation step demonstrates how a near 1:1 structural correspondence between AST nodes and target code can be achieved, with Quasiquotation enabling a clear and maintainable implementation.

Finally, the construction of the validator encapsulates the full semantic translation, integrating the rewritten modules into a coherent on-chain execution model; by mapping Move's execution paradigm into the eUTxO-based validation logic and CPS representation, the compiler produces validators that faithfully preserve the original program behavior. Overall, this stage consolidates the transformation process, delivering complete and executable Aiken code from the initial Move source.

Chapter 5

Case Study: The Betting Contract

We now illustrate the behavior of the compiler through a concrete case study: the *Betting Smart Contract*, whose specification is described below. The contract logic is inspired by the example available in the Rosetta Smart Contracts repository (Blockchain UNICA (2026)).

In the betting protocol, three parties are involved: two players and an oracle. The oracle is the entity responsible for starting a betting round by defining the main parameters of the game, such as the addresses of the participating players, the type and amount of the asset being wagered, and a deadline timestamp.

Once the round has been initialized, each player may join by depositing the required amount of tokens, provided that their address matches one of the participants authorized by the oracle. After both players have joined, the oracle privately determines the winner, for example through an off-chain randomization process. The winning player then receives both their original stake and the amount deposited by the other participant.

To prevent funds from remaining locked indefinitely, every round is associated with a deadline. If the oracle fails to declare a winner before the deadline expires, each player is allowed to reclaim their deposited funds.

From an analytical perspective, each round is therefore characterized by the following information:

- Identifiers, meaning addresses, of the two players and the oracle
- Type of the asset to deposit, along with the required amount
- Deadline timestamp

In the Move language, a round is represented through the following Oracle type:

```
 struct Oracle<phantom CoinType> has key {
     player1: address,
     player2: address,
     oracle: address,
     stake: u64,
     deadline: u64
}
```

The protocol supports the following actions:

- **init**: Used by the oracle to initialize a round
- **join**: Used by a player to join an initialized round

- **win**: Invoked by the oracle to announce the winner and assign the reward
- **timeout**: Invoked by a player to reclaim its funds once the deadline is passed without a winner

In Move, each action is implemented as a dedicated entry function, defined in a separate script and accepting the required parameters:

```
// init_round.move
fun init_round<CoinType>(
    bookmaker: &signer,
    player1: address,
    player2: address,
    oracle: address,
    stake: u64,
    deadline: u64
){...}

// join_round.move
fun join_round<CoinType>(
    partecipant: &signer,
    bookmaker: address
){...}

// win_round.move
fun win_round<CoinType>(
    oracle: &signer,
    winner: address,
    bookmaker: address
){...}

// timeout_round.move
fun timeout_round<CoinType>(
    bookmaker: address
){...}
```

The compiler detects all these entry functions and automatically generates an Algebraic Data Type for the Redeemer:

```
// main.ak
pub type Redeemer {
    WinRound(Addr, Addr, TWitness)
    TimeoutRound(Addr, TWitness)
    JoinRound(Addr, TWitness)
    InitRound(Addr, Addr, Addr, Int, Int, TWitness)
}
```

Notice that signer parameters are omitted, since they are automatically extracted from the transaction context by the validator and therefore do not need to be explicitly included in the Redeemer.

Later in the validator logic, the Redeemer is pattern matched to execute the appropriate computation:

```
// main.ak

// The pattern match is performed on the Redeemer.
// After the pattern match, retrieve the computed program state
let (_, program_state): CPS = when redeemer is {
    WinRound(arg1, arg2, arg3) -> {
        // The first parameter of the init_round is a Ref<Signer>,
        // so extract it from the function signatories
        expect Some(arg0) = list.at(self.extra_signatories, 0)
        let arg0 = Signer{address: arg0}

        // The signer is POST-ed on the scope so that it can referenced
        // in the next line
        let cps = scope_utils.post_scope("arg0", arg0, cps)

        let (_, cps) = win_round.win_round(
            reference_utils.make_ref("arg0", [], cps),
            arg1, arg2, arg3, cps)
        cps
    }
    TimeoutRound(arg0, arg1) -> {
        // The timeout_round does not need any signer
        let (_, cps) = timeout_round.timeout_round(arg0, arg1, cps)
        cps
    }
    JoinRound(arg1, arg2) -> {
        // Similar reasoning, omitted for readability
    }
    InitRound(arg1, arg2, arg3, arg4, arg5, arg6) -> {
        // Similar reasoning, omitted for readability
    }
}
```

Now that we have shown the high level flow for the validator, we pick two specific actions to show in more detail the Aiken code that the compilation produces, taking into consideration all the translation steps needed to handle the differences between the imperative style in the Move language and the functional approach of Aiken.

At first, consider the simple case of the **init** handler, whose only effect is to publish a new Round resource on the global storage:

```
// bet.move
public entry fun init<CoinType>(
    bookmaker: &signer,
    player1: address,
    player2: address,
    oracle: address,
    stake: u64,
    deadline: u64
): () {
```

```
    let bet = Round<CoinType> { player1, player2,
        oracle, stake, deadline };
    move_to<Round<CoinType>>(bookmaker, bet);
}
```

We now provide the Aiken code that is generated by the above Move function, extending it with some inline comment to better explain it:

```
// bet.ak
pub fn init(
    bookmaker: Ref<Signer>,
    player1: Addr,
    player2: Addr,
    oracle: Addr,
    stake: Int,
    deadline: Int,
    tw_coin_type: TWitness,
    cps_state: CPS) -> (Void, CPS) {
        // At the start of any function, a new scope is pushed on top
        // of the existing ones
        let cps_state: CPS = scope_utils.push_scope(cps_state)

        let bet = Round {
            player1: player1,
            player2: player2,
            oracle: oracle,
            stake: stake,
            deadline: deadline }

        // Custom operator for the move_to
        // also notice the usage of type witness
        let (temp_63, cps_state): (Void, CPS) =
            global_storage_utils.move_to(
                bookmaker, TypeWitnessC("Round", [tw_coin_type]),
                bet,
                cps_state)

        // This line is a byproduct of the translation logic
        // it has no impact on the code and can be removed
        // from the generation in a later improvement
        temp_63

        // At the end of any function, pop the current scope.
        // Notice how the Aiken code returns Void
        // for the original unit return type
        (Void, scope_utils.pop_scope(cps_state))
}
```

It can be noticed the additional lines to handle the CPS, both for adding a new scope to the stack and to handle mutations on the global storage when invoking other functions, as well

as the usage of type witness, required in order for the *move_to* operator to access the correct resource.

The **join** action is instead slightly more complex since in its source code we can find operations involving references and invocations to the Aptos Coin module, in addition to updates to the global storage:

```
// bet.move
public fun join<CoinType>(
    partecipant: &signer,
    bookmaker: address): () acquires Round {
    let round = borrow_global_mut<Round<CoinType>>(bookmaker);

    // Asserts are not supported at the time of writing,
    // therefore we present the code with the corresponding
    // lines commented

    // assert!(
    //     address_of(partecipant) == round.player1
    //         || address_of(partecipant) == round.player2,
    //     0
    // );

    // Invocation of the Aptos Coin module to withdraw funds
    let bet = coin::withdraw<CoinType>(partecipant, round.stake);

    // assert!(coin::value<CoinType>(&bet) == round.stake, 0);

    // Update on the global storage
    let bet = Bet<CoinType> { value: bet };
    move_to<Bet<CoinType>>(partecipant, bet);
}
```

The resulting Aiken code is therefore more verbose, since it has to handle all the above complexity, and additionally it has to take into consideration the difference in polymorphic code. We suggest to look in particular at the lines corresponding to the coin withdrawal:

```
// bet.ak
pub fn join(
    partecipant: Ref<Signer>,
    bookmaker: Addr,
    tw_coin_type: TWitness,
    cps_state: CPS) -> (Void, CPS) {
        let cps_state: CPS = scope_utils.push_scope(cps_state)

        // Extract the round from the global storage
        let round = global_storage_utils.borrow_global(
            bookmaker,
            TypeWitnessC("Round", [tw_coin_type]),
            cps_state)
```

```
        // The following line is an example of the inevitable verbosity
        // of the generated code
        // Its main purpose is to invoke the ad-hoc Aiken library
        // for the Aptos Coin module,
        // however, the entire result of the invocation
        // has to be wrapped to handle polymorphism.
        // In a similar way, the ‘round.state‘ record access needs
        // to be first downcasted from the Data supertype
        let (temp_64, cps_state): (coin.Coin<Data>, CPS) = {
            let (el_0, el_1): (coin.Coin<Data>, CPS) = coin.withdraw(
                partecipant,
                {expect val: Round<Data> = reference_utils
                                                .deref(round, cps_state)
                    val
                }.stake,
                tw_coin_type,
                cps_state)

        // The result of the function invocation has to
        // respect the CPS
        ({
            expect val: coin.Coin<Data> = as_data(el_0)
            val
        }, el_1)
  }

  // This line is used to provide the original variable name
  // from the previous computation
  // Notice the optimization that did not require to POST
  // this variable on the program scope
  let bet = temp_64
  let bet = Bet { value: bet }

  // Operation of the global storage and CPS return, as already seen
  let (temp_65, cps_state): (Void, CPS) = global_storage_utils.move_to(
    partecipant,
    TypeWitnessC("Bet", [tw_coin_type]),
    bet,
    cps_state)
  temp_65
  (Void, scope_utils.pop_scope(cps_state))
}
```

The previous code highlights one of the main drawbacks of the generated Aiken output, namely its increased verbosity. As a consequence of the transformations required to guarantee a correct translation, the resulting code is inevitably less human readable.

Nevertheless, the generated Aiken code should be regarded purely as the result of a compilation process rather than as an alternative codebase intended for manual development. For

this reason, any modification to the contract logic should be performed directly on the original Move source code instead of the generated validator.

This approach allows developers to implement smart contracts using the more familiar imperative programming style of Move, while still benefiting from the security guarantees provided by its linear type system. Whenever deployment on the Cardano blockchain is required, the compiler can then be invoked to automatically produce a ready-to-use Aiken validator.

Chapter 6

Conclusions

In this work, we have designed and implemented a translation flow that works on two different aspects: on a programming language perspective, we have a tool capable of translating code written in imperative style into a function style, managing complexities such as mutability and references to variables typical of an imperative approach and expressing them in a functional way via Continuation-passing style, while also providing translation for parametric polymorphism and type witness. On a blockchain perspective, this work shows how a smart contract written for an account-based blockchain, and therefore designed around the concept of a global storage, can be translated into a set of off-chain and on-chain scripts that, combined together, achieve the same behavior on a UTxO-based blockchain, managing the very different computation style of a state being the union of transaction outputs.

On a third and purely technical perspective, the compiler has been implemented in the Haskell language, whose type system and strict functional approach greatly improved the robustness of the code, and to further ensure its correctness we have written numerous test cases as the implementation proceeded. All of this has produced a tool that, in our opinion, is interesting both on a Programming Language side and for blockchain developers, and our objective is to keep expanding it in the belief that other people will find it useful.

On a personal note, the author of this work wants to spend a few words on his journey across the design and implementation of this compiler. The multiple works of understanding the characteristics and, above all, differences between Move and Aiken, as well as getting used to the UTxO-based approach, then working with two very different ecosystems, Aptos and Cardano respectively, along with designing a translation and implementing that into an unfamiliar and very strict language, which is Haskell, all of this combined together made so that a lot of effort has been needed, both regarding cognitive load and time-requiring proof of concepts, experimentation, and trial and errors. Nevertheless, the author is incredibly satisfied for what he has learned, both on blockchains and on programming languages, as well as for the tool that he has produced.

6.1 Future work

While this tool is currently able to generate code for a wide range of features of the Move language, there exist many areas where additional work is needed or welcome. Addressing a major point, the compiler currently generates on-chain code but not the off-chain part, which can not be considered a marginal part of any DApp, especially if it targets UTxO-based environments where it usually performs most of the work.

The design for the off-chain logic has already been analyzed and is documented in this

work, however the practical implementation of a compiler branch to generate it will naturally provide its own challenges and problems to solve. In our intentions, we aim to generate off-chain code for the Typescript community, mainly considering the popularity of the language and its portability. To integrate Typescript with Cardano, we have experimented with some proof of concepts scripts leveraging the MeshJS (MeshJS (2026)) library for Cardano, but we are open to consider alternatives in case they result more useful or practical to use.

Other than the off-chain code, there are other features available on Aptos Move that our compiler currently does not capture.

For example, the support for the Aptos Coin module currently assumes that the particular `Coin<CoinType>` has already been initialized and instantiated *a priori*; future work in this direction has to include the possibility to initialize new Coins at runtime and configure them, for example by providing the ability to configure minting and burning policies.

Another aspect is logging: Aptos Move allows emitting events that are logged on the blockchain, something that in Cardano, to the best of our knowledge, should work a bit differently, like passing metadata to a transaction or writing in Datums; a proper translation of logging might be useful for smart contract developers.

Another aspect is retrieving the timestamp. In Aptos, timestamps work relatively straightforward and the current time can be retrieved by the smart contract during its execution, however, timestamps in Cardano are very different, and instead rely on a validity range for the transaction, which might be specified when building a transaction and will determine it it will be accepted or not by the validator nodes. Therefore, a simple logic that work on timestamps in Aptos, such as restricting a certain action until some time has elapsed, needs a specific rework to be adapted in Cardano, and we expect it to be a non-trivial problem to solve.

On a programming language perspective, in addition, Move is an evolving language that is currently in active development and expanding with new features. As said in previous chapters, we focused on supporting features for a language prior to 2.0, therefore there exist a number of features for which the compiler can be extended, such as adding support for vectors, specific syntactic sugar for resources, and other modules from the standard library such as the Fungible Asset and Digital Asset.

6.2 Acknowledgments

As a final note, the author of this work would like to exploit his role as editor of this document to temporarily abandon academic third-person narration and thank all the people who made this project possible.

From a technical and academic perspective, thanks to Ch. Prof. Michele Bugliesi, who introduced me to the functional programming style with his master course *Advanced Programming Languages* and proposed me to work on this thesis, dedicating time to review my progress. Along with him, thanks to Dr. Alvise Spanò for helping me designing lot of difficult parts, dedicating his time to brainstorm ideas and possibilities for handling mutability, references and type witness among all. Thanks to Lorenzo Fanton, who introduced me to the project and provided numerous resources to get started with. Thanks to everyone that has worked and is working on the *blockchain-unica/rosetta-smart-contracts* repository for providing easy to use code snippets to learn and understand both Move and Aiken, as well as to Matthias Benkort, that goes online by the name of KtorZ, for his quick and exhaustive responses to some very specific questions regarding the Aiken language.

Thanks to everyone that supported me, from my parents and especially my sister Silvia,

for being such a pain to me with the "comfort zone" thing. Thanks to all my friends for their support, as well as to all the people in the Associazione Culturale Talentree, and thanks to everyone that has faked some interest when I started a conversation with "It's a compiler between two languages".

Lastly, I want to thank me, myself and I, for all the effort over the last few years. It took patience, questionable decisions, and multiple personalities, but we got here in the end.

Appendix A

Type deduction rules

The following are the induction rules for the type deduction implemented at the time of writing, including judgments for Intermediate Representation nodes.

Typing judgments have the form:

$$\Gamma \vdash e : \tau$$

where Γ is the stack of scopes, e an expression, and τ is a type.

Literals

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \quad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}} \quad \frac{}{\Gamma \vdash n : \texttt{Num}} \quad \frac{}{\Gamma \vdash a : \texttt{Address}}$$

Binary Operators

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \ \texttt{op}_{bool} \ e_2 : \texttt{Bool}} \quad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \ \texttt{op}_{num} \ e_2 : \texttt{Num}}$$

where

$$\texttt{op}_{bool} \in \{||, \wedge, ==, \neq, <, >, \leq, \geq\} \quad \texttt{op}_{num} \in \{+, -, *, /, \%, |, \&, \oplus, <<, >>\}$$

Notice how no constraint is performed on the type of the operands. In fact, the entire compilation process assumes that the source Move project type-checks correctly for the Move compiler. With such assumption, there is no need to enforce a stricter typing rule that would result in unnecessary runtime complexity to the compiler.

Assignment

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (x := e) : \texttt{Unit}}$$

Unary Operators Boolean negation:

$$\frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma \vdash \neg e : \texttt{Bool}}$$

References:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{\&}\ e : \texttt{ImmRef}\ \tau} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{\&mut}\ e : \texttt{MutRef}\ \tau}$$

Dereference:

$$\frac{\Gamma \vdash e : \texttt{MutRef}\ \tau}{\Gamma \vdash *e : \tau} \qquad \frac{\Gamma \vdash e : \texttt{ImmRef}\ \tau}{\Gamma \vdash *e : \tau}$$

Move and Copy

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash \texttt{move}\ x : \tau} \qquad \frac{\Gamma \vdash x : \tau}{\Gamma \vdash \texttt{copy}\ x : \tau}$$

Tuples Single element:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e) : \tau}$$

Multiple expressions:

$$\frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n)}$$

Name Access

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

This means accessing any symbol available on the scope, including imports from other modules

Typed Expressions

$$\frac{}{\Gamma \vdash (e : \tau) : \tau}$$

Casting

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash (e\ \texttt{as}\ \tau_2) : \tau_2}$$

Struct construction

$$\frac{\texttt{struct}\ \ s\langle s_1, \ldots, s_n \rangle \in \Gamma}{\Gamma \vdash s\langle \tau_1, \ldots, \tau_n \rangle : s\langle \tau_1, \ldots, \tau_n \rangle}$$

With $s_1, \ldots, s_n$ type parameter names. Positional structs can be interpreted as a syntactic sugar for named structs.

Field Access

$$\frac{\Gamma \vdash e : s\langle \vec{\tau} \rangle \qquad \Gamma \vdash s.f : \tau}{\Gamma \vdash e.f : \tau}$$

Accessing a field of a referenced struct is also a syntactic sugar to dereference it:

$$\frac{\Gamma \vdash e : \texttt{Ref}\ s\langle \vec{\tau} \rangle \qquad \Gamma \vdash s.f : \tau}{\Gamma \vdash e.f : \tau}$$

Function Calls

$$\frac{\Gamma \vdash f : s^*.\ (\tau_1, \ldots, \tau_n) \to \tau_r \qquad \Gamma \vdash e_i : \sigma_i}{\Gamma \vdash f\langle \vec{\tau} \rangle (e_1, \ldots, e_n) : \tau_r[s^* \mapsto \vec{\tau}]}$$

With s^* type parameters and $\vec{\tau}$ type arguments. Also notice how the concrete types of the function arguments σ_i are ignored. This is to simplify the type deduction logic, but it can be further extended so to cover the case when the function is invoked without explicitly passing the type arguments, by looking at the concrete types of the function arguments.

Sequences

$$\frac{}{\Gamma \vdash \{\} : \texttt{Unit}} \qquad \frac{}{\Gamma \vdash \{\ldots;\} : \texttt{Unit}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \{\ldots; e\} : \tau}$$

The term "sequence" is how the Move compiler internally refers to code blocks, enclosed by curly braces. The resulting type of a Sequence is the type of the last expression, identified by the lack of a final ; sign

Conditionals

$$\frac{\Gamma \vdash c : \sigma \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ e_1 : \texttt{Unit}} \qquad \frac{\Gamma \vdash c : \sigma \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

Notice how the assumption that the Move project already type checks allows to simplify the type deduction logic by not imposing any particular type of the condition, which is assumed to be boolean, nor to enforce that both branches have the same type. These checks are already performed by the Move type checker and adding them to the type deduction logic would only introduce unnecessary runtime complexity.

Loops

$$\frac{\Gamma \vdash c : \sigma \qquad \Gamma e \vdash e : \tau}{\Gamma \vdash \texttt{while}\ c\ e : \texttt{Unit}} \qquad \frac{}{\Gamma \vdash \texttt{loop}\ e : \texttt{Unknown}}$$

As a note, the Move language provides a slightly different typing rule for the `loop` construct, depending on the presence of `break` expressions in its body. For simplicity, we have omitted this logic and assumed that a `break` surely exists. This is mostly expected, since in absence of it the loop construct would continue iterating forever.

Control Flow

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}\ e : \tau}$$

$$\frac{}{\Gamma \vdash \texttt{break} : \texttt{Unknown}} \qquad \frac{}{\Gamma \vdash \texttt{continue} : \texttt{Unknown}}$$

In the Move type system, `break` and `continue` are typed accordingly in order to respect the outer loops they are encased in. In the current implementation of the type deduction, the type of these constructs is ignored.

Abort

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{abort}\ e : \tau}$$

IR - Type witness

$$\frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash \text{tw\_c}\langle n, [e_1, \ldots, e_k]\rangle : \omega}$$

$$\frac{}{\Gamma \vdash \text{tw\_i}\langle x\rangle : \omega}$$

IR - Program scope operators

$$\frac{\Gamma \vdash x : \text{Identifier} \qquad \Gamma \vdash e : \tau \qquad \Gamma \vdash \texttt{cps} : \text{CPS}}{\Gamma \vdash \mathscr{S}_{\text{POST}}(x,\ e,\ \texttt{cps}) : \text{CPS}}$$

$$\frac{\Gamma \vdash x : \text{Identifier} \qquad \Gamma \vdash \texttt{e} : \tau \qquad \Gamma \vdash \texttt{fs} : [\text{Field}] \qquad \Gamma \vdash \texttt{cps} : \text{CPS}}{\Gamma \vdash \mathscr{S}_{\text{PUT}}(x,\ e,\ \texttt{fs},\ \texttt{cps}) : \text{CPS}}$$

$$\frac{\Gamma \vdash x : \text{Identifier} \qquad \Gamma \vdash \texttt{cps} : \text{CPS}}{\Gamma \vdash \mathscr{S}_{\text{GET}}(x,\ \texttt{cps}) : \text{Data}}$$

IR - Global storage operators

$$\frac{\Gamma \vdash s : \text{Ref<signer>} \qquad \Gamma \vdash t : \omega \qquad \Gamma \vdash v : \tau \qquad \Gamma \vdash \texttt{cps} : \text{CPS}}{\Gamma \vdash \mathscr{G}_{\text{move\_to}}(s,\ t,\ v,\ \texttt{cps}) : (\text{Unit, CPS})}$$

$$\frac{\Gamma \vdash a : \text{address} \qquad \Gamma \vdash t : \omega \qquad \Gamma \vdash \texttt{cps} : \text{CPS}}{\Gamma \vdash \mathscr{G}_{\text{move\_from}}(a,\ t,\ \texttt{cps}) : (\text{Data, CPS})}$$

$$\frac{\Gamma \vdash a : \text{address} \qquad \Gamma \vdash t : \omega}{\Gamma \vdash \mathscr{G}_{\text{borrow\_global}}(a,\ t) : \text{Ref<}\tau\text{>}}$$

$$\frac{\Gamma \vdash a : \text{address} \qquad \Gamma \vdash t : \omega \qquad \Gamma \vdash \texttt{cps} : \text{CPS}}{\Gamma \vdash \mathscr{G}_{\text{exists}}(a,\ t,\ \texttt{cps}) : \text{Bool}}$$

IR - Reference operators

$$\frac{\Gamma \vdash x : \text{Identifier} \qquad \Gamma \vdash \texttt{fs} : [\text{Field}] \qquad \Gamma \vdash \texttt{cps} : \text{CPS}}{\Gamma \vdash \mathscr{R}_{\text{make}}(x,\ \texttt{fs},\ \texttt{cps}) : \text{Ref<}\tau\text{>}}$$

$$\frac{\Gamma \vdash x : \text{Ref<}\tau\text{>} \qquad \Gamma \vdash \texttt{cps} : \text{CPS}}{\Gamma \vdash \mathscr{R}_{\text{GET}}(x,\ \texttt{cps}) : \text{Data}}$$

$$\frac{\Gamma \vdash x : \text{Ref<}\tau\text{>} \qquad \Gamma \vdash e : \tau \qquad \texttt{fs} : [\text{Field}] \qquad \texttt{cps} : \text{CPS}}{\Gamma \vdash \mathscr{R}_{\text{PUT}}(xe,\ \texttt{fs},\ \texttt{cps}) : \text{Data}}$$

Bibliography

Aiken (2026). Common design patterns – state thread tokens (a.k.a stt). https://aiken-lang.org/fundamentals/common-design-patterns#state-thread-tokens-aka-stt
Accessed: 2026-05-07.

Alex Developers (2026). Alex user guide. https://haskell-alex.readthedocs.io/en/latest/
Accessed: 2026-05-07.

Bartoletti, M., Benetollo, L., Bugliesi, M., Crafa, S., Sasso, G. D., Pettinau, R., Pinna, A., Piras, M., Rossi, S., Salis, S., Spanò, A., Tkachenko, V., Tonelli, R., and Zunino, R. (2025). Smart contract languages: A comparative analysis. *Future Generation Computer Systems*, 164:107563.

Bartoletti, M., Marchesin, R., and Zunino, R. (2026). Scalable utxo smart contracts via fine-grained distributed state. *Future Generation Computer Systems*, 175:108023.

Bird, R. and Wadler, P. (1988). *An introduction to functional programming*. Prentice Hall International (UK) Ltd., GBR.

Blockchain UNICA (2026). Rosetta smart contracts. GitHub repository. https://github.com/blockchain-unica/rosetta-smart-contracts
Accessed: 2026-05-08.

CardaMove Contributors (2026). Cardamove/compiler. GitHub repository. https://github.com/CardaMove/compiler
Accessed: 2026-05-07.

contributors, H. (2026a). Neatinterpolation. https://hackage.haskell.org/package/neat-interpolation-0.5.1.4/docs/NeatInterpolation.html
Accessed: 2026-05-08.

contributors, H. (2026b). Quasiquotation. https://wiki.haskell.org/Quasiquotation
Accessed: 2026-05-08.

Fanton, L., Spanò, A., and Bugliesi, M. (2025). Secure distributed state management in eutxo blockchains. In *2025 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pages 27–33.

Happy Developers (2026). Happy documentation. https://haskell-happy.readthedocs.io/en/latest/
Accessed: 2026-05-07.

KtorZ (2026). Upcasting and downcasting data. https://discord.com/channels/946071061567529010/1479058137360371897
Accessed: 2026-05-08.

MeshJS (2026). Mesh sdk. https://meshjs.dev/
Accessed: 2026-05-08.

Mitchell, N. (2026). uniplate: Help writing simple, concise and fast generic operations. Hackage package. https://hackage.haskell.org/package/uniplate
Accessed: 2026-05-07.

Spanò, A., Benetollo, L., Bugliesi, M., Crafa, S., Ressi, D., and Rossi, S. (2026). Algomove: Typed abstractions for algorand smart contracts. *Blockchain: Research and Applications*, page 100485.