Runwai: A Refinement-Typed DSL for Certified Constraints in Zero-Knowledge Virtual Machine

HIDEAKI TAKAHASHI, Columbia University, USA
JIAYU ZHOU, Columbia University, USA
ADITYA UNNIKRISHNAN, Columbia University, USA
MINGKAI LI, Columbia University, USA
KESHAV BERIWAL, Columbia University, USA

Zero-Knowledge Proofs (ZKPs) are rapidly emerging as a foundational technology for both security- and scalability-critical applications, spanning private transactions, scalable rollups, and verifiable machine learning. Zero-Knowledge Virtual Machines (zkVMs) extend this paradigm by enabling verifiable computation over arbitrary programs. However, modern zkVMs rely heavily on the Algebraic Intermediate Representation (AIR) and lookup arguments, low-level, error-prone constraint systems where even a subtle bug can introduce silent and catastrophic vulnerabilities, often undetectable through conventional testing. Although formal verification is clearly essential, current domain-specific languages (DSLs) impose a trilemma among performance, safety, and expressiveness: one must either sacrifice performance for high-level abstraction, safety for low-level control and efficiency, or expressiveness for formal tractability.. Worse, existing formal-first languages are often limited in expressiveness and difficult to use, leaving a critical verification gap in zkVM development.

To address this challenge, we present **Runwai**, a novel refinement-typed DSL for authoring and formally certifying AIR and lookup constraints. Runwai enables developers to express efficient, low-level constraints, comparable to those in high-performance embedded domain-specific languages (eDSLs), while embedding mathematical specifications directly into a rich, intuitive type system. Implemented in the Lean 4 theorem prover, Runwai uniquely translates type-checking obligations into formal lemmas, allowing developers to interactively and rigorously prove constraint correctness. This design bridges the long-standing gap between circuit-level efficiency and machine-checked safety. We formally specify Runwai's syntax, semantics, and type system, and prove type preservation. Finally, Runwai's compiler integrates the Lean4-based certification workflow with high-performance Rust proving systems, demonstrating a practical path toward provably correct zkVMs.

1 INTRODUCTION

Zero-Knowledge (ZK) proof systems have become a robust foundation for security-critical applications, ranging from anonymous transactions in Web 3.0 to verifiable machine learning. The ZKP market is projected to surpass $10 billion by 2030, underscoring its increasing significance in modern verifiable computing [26]. Ethereum's founder, Vitalik Buterin, has further proposed transitioning the EVM to a RISC-V-based architecture to make the platform more ZK-friendly [6]. Recent advances in zero-knowledge virtual machines (zkVMs) have further expanded the applicability of ZK proofs by enabling the execution and verification of arbitrary programs succinctly written in high-level languages [20].

Despite their promise, developing ZK circuits remains a notoriously challenging task. In particular, the Algebraic Intermediate Representation (AIR) and lookup arguments, widely used constraint formats favored by zkVMs, force programmers to encode intricate constraints over an execution trace represented as a two-dimensional matrix. This low-level representation is unintuitive and prone to errors for human developers [2, 35].

Consequently, bugs in ZK circuits are both easy to introduce and potentially catastrophic. A single flawed or missing constraint can create a silent vulnerability. This would allow a malicious prover

Authors' addresses: Hideaki Takahashi, ht2673@columbia.edu, Columbia University, USA; Jiayu Zhou, jz4019@columbia.edu, Columbia University, USA; Aditya Unnikrishnan, au2325@columbia.edu, Columbia University, USA; Mingkai Li, ml5120@columbia.edu, Columbia University, USA; Keshav Beriwal, kb3492@columbia.edu, Columbia University, USA.

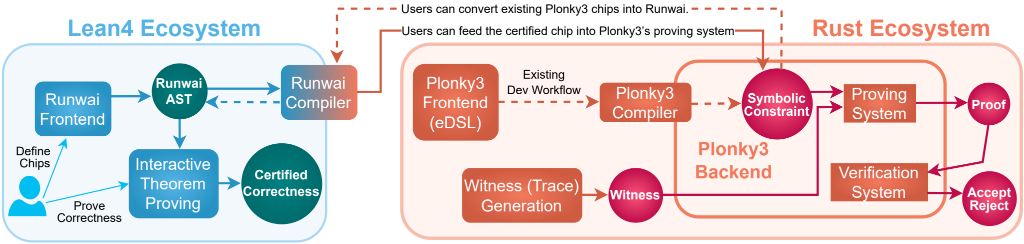

Fig. 1. Overview of Runwai Language. Runwai focuses on specifying constraints and their formal verification. Developers define and certify constraints in Runwai, which is implemented in the Lean4 Ecosystem (left). The Runwai compiler then emits symbolic constraints for a standard, high-performance Rust Ecosystem (e.g., Plonky3, right), which handles witness and proof generation."

to generate fraudulent proofs that are accepted by the verifier, for example, to create counterfeit assets in a rollup, bypass security checks, or corrupt a verifiable machine learning model [30, 33].

However, current DSLs trap developers in a fundamental trilemma, forcing a choice between efficiency, simplicity, and safety:

Low-level DSLs, such as Circom [4], Plonky3 [31], and Airscript [22], offer maximum performance by providing developers with direct access to constraints, while this path of efficiency comes at the cost of astronomical complexity, making the code non-intuitive and difficult to debug or verify.

High-level languages, such as Noir [3] and Zokrates [10], offer a path of simplicity with a familiar, imperative experience with automatic constraint generation from high-level specifications written in intuitive syntax. However, this abstraction can result in highly unoptimized circuits [23], with performance overheads that are often untenable for critical zkVM components.

Formal-first languages, such as CODA [21] and Clean [36], have successfully demonstrated a path of safety using refinement types to certify circuits formally. However, CODA is fundamentally built for Rank-1 Constraint System (R1CS) and does not support AIR or lookup arguments. While Clean supports AIR and lookup, its syntax remains significantly different from that of traditional programming languages, resulting in a steep learning curve and poor readability.

To address this gap, we present **Runwai** (**R**efinement-based **U**ltra-lightweight La**N**guage for **W**ell-typed **AI**r), a novel statically typed language for authoring zero-knowledge circuits in AIR. Runwai is designed to resolve the trilemma: it allows developers to write low-level, highly efficient constraints (like in Plonky3) while enabling formal, machine-checked guarantees of correctness (like in CODA) with native support for AIR and lookup arguments (like in Clean).

Runwai's core mechanism is a refinement type system designed to capture the precise mathematical properties of AIR and lookup constraints. Developers embed specifications directly into function types, aligning the circuit's intended behavior with its implementation. The key challenge in such a system is that verifying these properties requires reasoning about complex polynomial equations over finite fields. While existing SMT solvers do not scale to complex constraints over the finite fields used in real-world ZKP systems, Runwai overcomes this by being implemented in the Lean4 interactive theorem prover. It translates type-checking obligations into formal proof obligations within Lean, enabling developers to semi-automatically verify that their low-level constraints are correct, effectively bridging the gap between circuit-level efficiency and machine-checked safety.

Fig. 1 illustrates the Runwai ecosystem, which integrates a formal verification environment (Lean4) with a high-performance proving environment (Rust).

In the Lean4 Ecosystem (left), a developer defines their constraints, called *chips*, using the Runwai frontend. The language's integration with Lean4 enables them to interactively verify the correctness of their definitions against their specified refinement types, thereby producing a Certified Correctness guarantee for the chips written in Runwai.

In the Rust Ecosystem (right), the Runwai Compiler translates the certified constraints into a format that a standard Plonky3 backend can consume, which offers popular ZKP proving and verification algorithms, such as FRI [5]. Crucially, Runwai focuses only on the correctness of the constraints; the high-performance Witness (Trace) Generation and the final Proving and Verification System still run in the Rust environment, leveraging its optimized computation and rich cryptography libraries. This architecture enables Runwai to certify existing Plonky3 chips or directly integrate new, certified chips into Plonky3's proofing system.

In summary, we make the following contributions:

- **Runwai Language**: A novel, refinement-typed DSL for authoring and certifying low-level AIR and lookup constraints for modern zkVMs with intuitive syntax.
- **Theoretical Guarantee**: We formally prove that the well-typed Runwai programs satisfy their specifications, guaranteeing that the type checking of Runwai is equivalent to the formal verification of constraints.
- **Verification of Real-World zkVM**: We re-implement some basic components of zkVM in our Runwai and verify their correctness to demonstrate the power of Runwai.

Our implementation is publicly available at https://github.com/Koukyosyumei/Runwai.

2 BACKGROUND

A **Zero-Knowledge Proof (ZKP)** is a cryptographic protocol that enables one party (the prover) to prove to another party (the verifier) that a given statement is true, without revealing any information beyond the validity of the statement itself, while the verification process is succinct [20]. These remarkable properties of proving knowledge without disclosing secrets and succinctly verifying it have transformed ZKPs from a theoretical curiosity into a foundational technology for modern security and privacy. Its applications are rapidly expanding, most notably in the blockchain space to enable private transactions (e.g., Zcash [29]) and scalable off-chain computation (e.g., zk-rollups). Beyond Web 3.0, ZKPs are powering new forms of verifiable identity, secure machine learning, and privacy-preserving data analysis [34]. As a result, the ZKP market has seen significant investment and research, establishing it as a critical component for the next generation of trustless and verifiable digital systems.

A **Zero-Knowledge Virtual Machine (zkVM)** is a ZKP system that allows a prover to demonstrate the correct execution of a program to a verifier. The verifier can then check a succinct proof to confirm the program's integrity without needing to re-run it. At its core, proving a program's execution is equivalent to proving the validity of its "execution log." To illustrate this, consider a program that calculates a Fibonacci number. To prove this computation, we first record every step of the calculation—the initial state, each addition, and the final result—into a sequential log. This detailed, step-by-step record of a program's state at every cycle is called an *execution trace*, which is essentially a two-dimensional matrix defined over the finite field. zkVM generates a proof guaranteeing that every step recorded in this trace correctly follows the program's rules (in this case, the Fibonacci rule $F_n = F_{n-1} + F_{n-2}$), while optionally allowing the prover to keep the intermediate computational steps private.

2.1 The Multi-Table Architecture of zkVM

To prove the execution of an arbitrary program, a typical zkVM first takes the program and its input values, then executes each instruction as a standard virtual machine would. During this process, the zkVM records all runtime events, such as which instruction was executed at each step, when and where memory values were loaded or stored, and how they were modified. After execution, these logs are transformed into an execution trace table, a two-dimensional matrix defined over a finite field. Each row in this table typically represents a single execution event (e.g., the execution of an instruction or a memory load/store). At the same time, each column corresponds to a particular aspect of the system state, such as the clock, program counter, opcode, operands, or auxiliary values like the multiplicative inverse of operands in the finite field.

The zkVM also defines a set of algebraic constraints over this trace table, which are satisfied if and only if the table represents a valid execution history. One of the most commonly used formats in zkVM is Arithmetic Intermediate Representation (AIR) [14], which specifies the relationship between two consecutive rows in a table as a polynomial constraint:

Definition 1 (Arithmetic Intermediate Representation (AIR)). *An Arithmetic Intermediate Representation (AIR) $\mathcal{A}$ consists of two types of constraints over a two-dimensional matrix:*

- *a state transition function: $f : \mathbb{F}^m \times \mathbb{F}^m \longrightarrow \{0, 1\}$*
- *a list of boundary conditions: $\mathcal{B} \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{F}$*

A matrix $T \in \mathbb{F}^{n \times m}$ satisfies the AIR $\mathcal{A} = \langle f, \mathcal{B} \rangle$ if and only if the following conditions are met:

$$\forall i \in [0, \dots n-2].\ \ f(T[i,:], T[i+1,:]) = 1 \tag{1}$$

$$\forall (i, j, b) \in \mathcal{B}.\ \ T[i, j] = e \tag{2}$$

For many real-world programs, a single execution trace can become enormous and contain a wide range of operations, from arithmetic to memory access. Validating such a large, monolithic trace is complex and inefficient. To manage this, modern zkVMs employ a **multi-table architecture** [9, 32], which uses a core principle of "division of labor". Instead of one giant trace, the execution is broken down into several smaller, specialized tables of data. Each table is governed by a corresponding chip, which is the virtual object that defines the set of mathematical constraints for that table. This modular design includes specialized chips such as: *CPU chip* to constrain the main program flow and instruction sequencing; *ALU chip* to constrain different arithmetic operations like addition or multiplication; *Memory chip* to ensure that all memory reads and writes are consistent throughout the execution.

This separation simplifies the constraint design for each component, but it creates a new challenge: ensuring that all these independent tables remain consistent with one another. This cross-table communication and consistency is enforced using a cryptographic tool known as a **lookup argument** [13]. Lookup arguments are the "glue" that holds the multi-table architecture together, ensuring that the separate, specialized tables are consistent with each other. Specifically, many modern zkVMs use lookup arguments with multiplicities [9, 13] to enforce a counting membership relationship between two vectors. In this approach, one vector, U, has a multiplicity value μ_i associated with its i-th element. The constraint is satisfied if and only if, for any given element u, the total number of times u appears in the other vector, V, equals the sum of the multiplicities for all elements in U whose value is also u.

Definition 2 (Multiplicity-aware Lookup). *Let $V \in \mathbb{F}^n$ and $U \in \mathbb{F}^m$ be two tables, and let $\mu = (\mu_0, \mu_1, \dots, \mu_{m-1}) \in \mathbb{N}^m$ be a multiplicity vector associated with the elements of U.*

The multiplicity-aware lookup constraint is satisfied if and only if for every row index $i \in \{0, \dots, m-1\}$, the count of the element U_i in V equals the sum of multiplicities for all elements in U that share the same value:

$$\sum_{j=0}^{n-1} \mathbb{1}\left[V_j = U_i\right] \; = \; \sum_{k=0}^{m-1} \mu_k \cdot \mathbb{1}\left[U_k = U_i\right]. \tag{3}$$

Consider a program that consists of two instructions (see Tab. 1): an IMM32 that loads the immediate value 6 into memory address -4, followed by an ADD32 that adds the value at address -4 and an immediate 2, then stores the result 8 at address -8.

Each instruction is represented in the Program Table, where the first operand denotes the destination (or result) address. The fourth and fifth operands act as immediate flags indicating whether the second and third operands are immediate values or memory addresses. For example, the entry `ADD32 [-8, -4, 2, 0, 1]` means that the result will be stored at address -8, the first operand -4 is a memory address, and the second operand 2 is an immediate value.

The CPU Table records the per-cycle execution trace of these instructions. Each row includes read and write channels, where `read_ch_1`, `read_ch_2`, and `write_ch` are triples of the form `(used, addr, val)`. Here, `used` indicates whether the channel is active in that cycle, `addr` specifies the memory address, and `val` is the value read or written. For instance, in the second row, the CPU reads 6 from address -4, uses an immediate value 2, and writes 8 to address -8.

The AIR constraints on the CPU Table enforce control-flow correctness (e.g., `next.pc = current.pc + 4`) and address consistency between operands and channels. However, they do not check the arithmetic relation itself (`6 + 2 = 8`). To verify this, the CPU chip emits the tuple `(6, 2, 8)` to a shared communication channel (the bus), where the Add Table receives it. The AIR constraints on the Add Table then verify that each row satisfies `op_a + op_b = result`. A lookup argument ensures that every tuple emitted by the CPU exists in the Add Table, guaranteeing arithmetic correctness. The Memory Table also enforces the consistency of memory accesses across all instructions. Each row records the address, clock cycle, read/write flag, and value. The AIR constraints on the Memory Table verify that: 1) Read operations return the most recently written value to that address, and 2) Writes update the memory state deterministically across consecutive cycles. The lookup between CPU and Memory tables ensures global memory coherence between the CPU's internal trace and the external memory view.

Table 1. Example of zkVM Execution Trace Tables

Program Table

pc	opcode	operand
0	IMM32	`[-4,6,0,0,0]`
4	ADD32	`[-8,-4,2,0,1]`

CPU Table

clk	pc	opcode	read_ch_1	read_ch_2	write_ch
0	0	IMM32	(0, 0, 0)	(0, 0, 0)	(1, -4, 6)
1	4	ADD32	(1, -4, 6)	(0, 0, 2)	(1, -8, 8)

Add Table

op_a	op_b	result
6	2	8

Memory Table

addr	clk	is_read	is_write	val
-4	0	0	1	6
-4	1	1	0	6
-8	1	0	1	8

2.2 Domain Specific Language for ZKP

Developing a zero-knowledge proof (ZKP) application typically involves multiple stages: (1) designing an arithmetic circuit that encodes the computation to be proven, (2) expressing the circuit in a domain-specific language (DSL) for compilation into low-level constraints, and (3) generating and verifying proofs using a ZKP backend (e.g., Groth16 [12], PLONK [11], and DEEP-FRI [5]). Among these stages, the circuit definition step is both the most crucial and the most error-prone. To ease this process, numerous DSLs have been proposed, each trading off performance, usability, and safety in distinct ways.

At the core of these DSLs lies the constraint model used to represent computations. Traditional ZK systems rely on Rank-1 Constraint System (R1CS), which is widely supported and remains the foundation of many popular DSLs such as Circom [4], Noir [3], and ZoKrates [10].

In contrast, modern proof systems—especially those used in zkVMs—favor Algebraic Intermediate Representations (AIR) and lookup arguments. AIR expresses computations as transition constraints over execution traces, where each row represents a program state and constraints relate consecutive states. Lookup arguments, on the other hand, enable enforcing membership in pre-defined tables (e.g., for range checks or custom gates) without explicit polynomial constraints, significantly improving efficiency. These models are essential for scalable zkVM design but are harder to express safely in a general-purpose language.

In practice, users of domain-specific languages for ZK circuits are trapped in a fundamental trilemma, forced to choose between **efficiency**, **simplicity**, and **safety**, with no existing tool or language satisfactorily offering all three.

The Path of Efficiency. Low-level DSLs, such as Circom, plonky3, airscript, and Zigren, provide maximum performance by giving developers direct, fine-grained control over the arithmetic constraints defined over the finite field. However, this control comes at the cost of astronomical complexity. For instance, in a field modulo 7, the expression `1 / 2` equals four since $2 \cdot 4 \equiv 1 \bmod 7$. Such behaviors make it difficult for developers to reason about correctness using standard programming intuition. As a result, those low-level ZK DSLs often are non-intuitive and require deep domain expertise, making them difficult to write, debug, and formally verify.

The Path of Simplicity. Relatively high-level ZK languages like Noir or ZoKrates offer a much gentler learning curve. They provide a familiar, imperative programming experience, automatically compiling high-level code into constraints. The trade-off is a crippling performance overhead. This abstraction layer often generates highly unoptimized circuits, reportedly 3x to 300x slower than their manually crafted equivalents [23]. For performance-critical applications like zkVMs, this level of inefficiency is untenable.

The Path of Safety. A third category of languages attempts to provide formal safety guarantees at the language level. However, they suffer from their own critical flaws. For example, languages like CODA are built for R1CS constraints, and they fundamentally lack support for the AIR and lookup arguments that are the core of modern zkVM. Other formal languages, like Clean, do support AIR and lookups, but introduce complex semantics that deviate radically from those of standard programming languages and impose a steep learning curve on developers.

3 THE RUNWAI LANGUAGE

3.1 Motivation

To overcome the limitations of the existing ZK DSL stated above, we argue that ZK developers need a DSL that allows them to define low-level constraints with intuitive semantics while retaining the ability to verify correctness formally. This motivates the design of Runwai, a refinement-typed DSL with an automatic theorem-prover backend for authoring zero-knowledge circuits.

Compared to existing DSLs that support formal verification, Runwai provides a more practical and intuitive foundation for writing and reasoning about ZK constraints in zkVM. Compared to CODA, Runwai is built to natively support AIR and lookup arguments, the key primitives underlying modern zkVMs. Compared to Clean, where correctness is expressed as a Lean4's theorem external to the program, Runwai enables developers to express the expected behavior as part of the type itself, specifically, as the refinement predicate in the function's output type. This design aligns specifications with standard programming intuition, thereby bridging the gap between circuit-level efficiency, intuitive readability for regular code readers, and machine-checked safety.

Runwai is implemented in Lean 4, leveraging existing rich proof automation and ZKP-related libraries for constraint reasoning and formal verification. As a result, it provides a unified environment for both specifying and proving zero-knowledge constraints, significantly improving the reliability of zkVM development.

```
pub struct IsZeroOperation<T> {
    pub inverse: T;
    pub result: T;
}
impl<F: Field> IsZeroOperation<F> {
    pub fn eval<AB: SP1AirBuilder>(
        builder: &mut AB,
        a: AB::Expr,
        cols: IsZeroOperation<AB::Var>,
    ) {
    builder.assert_bool(cols.result);
    let one = AB::Expr::one();
    let inverse = cols.inverse;
    let is_zero = one.clone() - inverse * a.clone();
    builder.assert_eq(is_zero, cols.result);
    builder.when(cols.result).assert_zero(a.clone());
    }
}
```

Code 1. Plonky3 implementation

```
def main (x : Var field F) : Circuit F (Var field F) := do
  let xInv \to witness fun env =>
    if x.eval env = 0 then 0
    else (x.eval env : F)^-1
  let isZero <== 1 - x*xInv
  isZero * x === 0
  return isZero

def Spec (x : F) (output : F) : Prop :=
  output = if x = 0 then 1
           else 0
```

Code 2. Clean implementation

```
component IsZero(val: Val) {
 isZero := NondetReg(Isz(val));
 inv := NondetReg(Inv(val));
 isZero * (1 - isZero) = 0;
 val * inv = 1 - isZero;
 isZero * val = 0;
 isZero * inv = 0;
 isZero
}
```

Code 3. Zirgen implementation

```
chip IsZero
 (trace: [[Field: 3]: n], i : {v: UInt | v < n})
 -> {Unit| trace [i][1] == if trace [i][0] == Fp 0 then {Fp 1} else {Fp 0}} {
 let x = trace [i][0];
 let y = trace [i][1];
 let inv = trace [i][2];
 let u = assert_eq(y, ((Fp 0 - x) * inv) + Fp 1);
 let v = assert_eq(x*y, Fp 0); v
}
```

Code 4. Runwai implementation

Fig. 2. Comparison of the IsZero operation across different languages.

3.2 Design Principle

The design of Runwai follows four core principles: 1) a low-level yet intuitive representation of AIR and lookup constraints, 2) a strict separation between witness computation and circuit constraints, 3) explicit type annotations to distinguish finite field elements from integers, and 4) the use of refinement types to specify chip behavior.

Principle 1: Low-Level yet Intuitive Constraints. First, Runwai represents AIR and lookup constraints as a function, denoted as `chip`, that takes a trace matrix and a row index, then sequentially applies all relevant AIR and lookup constraints to that row. This low-level formulation allows developers to write highly efficient constraints. While eDSL of Plonky3 inspires design, Runwai's grammar is far more intuitive: it expresses constraints as sequential assertions over a two-dimensional matrix, rather than requiring numerous struct definitions and method calls on abstract objects.

Principle 2: Separation of Computation and Constraints. Second, Runwai focuses exclusively on constraint specification and does not handle witness computation (i.e., generating trace tables in zkVMs). Users can compute witnesses in any general-purpose language, such as Rust or Python. This separation is motivated by common sources of bugs in existing ZK DSLs, where witness computation and constraint definition are intermingled [30]. For instance, Circom often suffers from confusion between === and `assert`: while === both `assert` a condition and generates a corresponding constraint, `assert` only checks the condition during witness computation without adding a constraint. Misusing `assert` instead of === has led to critical under-constrained vulnerabilities in many real-world Circom projects.

Principle 3: Type Safety through Explicit Annotations. Third, Runwai enforces type annotations to finite field values with the `Fp` type to clearly distinguish them from regular integers, preventing subtle semantic errors.

Principle 4: Refinement Types as Specifications. Finally, the expected behavior of each chip is captured by its output type, which serves as a concise, intuitive specification embedded directly in the chip's function signature. CODA's success inspires this approach, which applies similar ideas to R1CS, a more classical constraint representation.

3.3 Runwai: Syntax

The syntax of Runwai, presented in Fig. 3, is designed to be a minimal functional language extended with constructs essential for defining AIR constraints and lookups. We basically follow CODA's notation. Its design provides a high-level, human-readable abstraction over the low-level polynomial constraints that define a ZK circuit.

Chip. A Runwai program is defined as a `chip`, which is the fundamental unit of verifiable computation. Each chip is a pure function that specifies the constraints for a single row of an execution trace. It takes two arguments:

(1) $x_{\mathtt{tr}} : [[\mathtt{Field}; m]; n]$: The entire execution trace, represented as a two-dimensional matrix of n rows and a fixed m columns over a finite field $\mathbb{F}$,
(2) $x_i : \{v : \mathtt{UInt} \mid v < n\}$: The index of the *current* row for which the constraints are being checked. Its type ensures that the index is always within the valid bounds of the trace.

The body of the chi, e, is an expression that must evaluates to a `Unit` type, refined by a post-condition ϕ_2. The constraints themselves are not specified by a return value but by the side effects within the body.

Core Functional Constructs. Runwai includes standard features from functional languages, such as variable bindings ($\mathtt{let}\ x\ =\ e\_1\ \mathtt{in}\ e\_2$), lambda abstractions ($\lambda x\ :\ \tau.e$), and conditionals (`if...then...else`). This functional core provides a structured and familiar environment for developers to express complex logic clearly and concisely. Variable bindings are immutable, which simplifies reasoning and aligns well with the static nature of circuit constraints.

$d ::= \textbf{chip}\ C(x_{\texttt{tr}} : [[\texttt{Field}; m]; n],$	**Chip declaration**
$\quad x_i : \{v : \texttt{UInt} \mid v < n\}) \longrightarrow \{v : \texttt{Unit} \mid \phi\}\{e\}$	
$e ::=$	**Expression:**
$\mid c$	constants
$\mid x$	variable
$\mid f \mid n \mid z \mid b$	arithmetic expressions
$\mid e_1 \oslash e_2$	binary relation
$\mid e_1[e_2]$	array indexing
$\mid \lambda x : \tau.\, e$	function abstraction
$\mid e_1\ e_2$	function application
$\mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$	variable binding
$\mid \texttt{if}\ e_c\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	conditional branch
$\mid \texttt{assert\_eq}(e_1, e_2)$	assert
$\mid \texttt{\#C}\ x_{\texttt{tr}}\ x_{\mathrm{i}}$	chip call
$\mid \texttt{lookup}\ C(e_1 : n_1, \ldots, e_k : n_k)$	lookup
$f ::= e_1 \circledast e_2$	**Field operations**
$n ::= e_1 \circledcirc e_2$	**Signed Integer expression**
$z ::= e_1 \ominus e_2$	**Integer expression**
$b ::= e_1 \odot e_2$	**Boolean expression**
$\circledast \in \{+, -, *, /\}$	**Field operators**
$\circledcirc \in \{+_{\mathbb{N}}, -_{\mathbb{N}}, *_{\mathbb{N}}\}$	**Signed Integer operators**
$\ominus \in \{+_{\mathbb{Z}}, -_{\mathbb{Z}}, *_{\mathbb{Z}}\}$	**Integer operators**
$\odot \in \{\wedge, \vee\}$	**Boolean operators**
$\oslash \in \{=, <, \leq\}$	**Binary Relations**

Fig. 3. Syntax of Runwai Program.

Trace Access and Arithmetic. Expressions `e_1[e_2]` allow for direct access to the execution trace. This is fundamental for defining transition constraints, which relate the state of a row `x_i` to the state of the previous row (e.g., `x_tr[x_i - 1]`). The language supports arithmetic expressions over finite fields (f), signed integers (z), and unsigned integers (n), which are essential for encoding computational logic.

Constraint Specification. The primary mechanism for defining constraints is `assert_eq(e_1, e_2)`. This expression asserts that `e_1` and `e_2` are equal. During compilation to AIR, this is translated into a polynomial constraint `e_1 - e_2 = 0`. The use of `assert_eq` makes the developer's intent explicit and localizes constraint definitions.

The `lookup C(...)` construct provides a high-level syntax for lookup arguments, a powerful technique in ZK systems for proving that a set of values in one computation trace exists within

$T ::=$	**Basic types** :
$\mid \texttt{Field}$	field element
$\mid \texttt{UInt} \mid \texttt{Int}$	unsigned / signed integer
$\mid \texttt{Bool}$	boolean
$\mid [T; n]$	fixed-size array
$\mid \texttt{Unit}$	empty
$\mid \{v : T \mid \phi\}$	refinement type
$\tau ::= T$	**Function type:**
$\mid x : \tau_1 \rightarrow \tau_2$	
$\phi ::=$	**Logical quantifiers:**
$\mid t$	refinement term
$\mid \neg\phi \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2$	boolean operators
$\mid \forall_{i \in [0,k]}.\ \phi$	bounded quantification
$\mid \exists x : \tau.\ \phi$	existential quantification
$t ::=$	**Refinement terms:**
$\mid \texttt{true} \mid \texttt{false}$	boolean constants
$\mid \mathbb{F}$	field value
$\mid e$	expression
$\mid \texttt{toN}(t) \mid \texttt{toZ}(t)$	field to integer

Fig. 4. Syntax of Runwai Types

another pre-computed table. The expression `lookup C(f_1: t_1, ..., f_k: t_k)` asserts that there exists a row in the trace of chip `C` where the `t_1`, ..., `t_k` on that row evaluate to the values of `f_1`, ..., `f_k`.

3.4 Runwai: Semantics

The operational semantics of Runwai, detailed in Fig. 5, formalize the evaluation of its expressions. The evaluation judgement is of the form $\Sigma; \sigma \vdash e \Downarrow v$, which reads *in the global context* Σ *and local environment* σ*, the expression* e *evaluates to the value* v. The evaluation environment is split into two components:

- Local Valuation σ: A standard, immutable mapping from variables to their values (e.g., $x \mapsto v$). This environment is updated within nested scopes, such as in let bindings and function applications.
- Global Chip Environment Σ: A read-only, global mapping from chip identifiers to their concrete execution traces. This environment is static and essential for the semantics of lookup arguments, as it provides the concrete data tables to be searched.

The semantics are largely standard for a functional language, but several rules are specific to the ZK domain and warrant further explanation:

E-Assert: The rule for `assert_eq(e_1, e_2)` dictates that the expression evaluates to unit if and only if `e_1` and `e_2` evaluate to the same value in the field, $f \in \mathbb{F}$. If they are not equal, the evaluation "blocks," signifying a constraint violation. In a concrete execution, this would cause the program to halt. In the context of ZK proof generation, this failure means the provided witness (i.e., the execution trace) is invalid, and the proving process would fail.

E-Lookup: This rule is the most complex and formalizes the behavior of the lookup argument construct `lookup C(f_1:t_1, ..., f_k:t_k)`. A lookup is successful and evaluates to `unit` if two distinct conditions are met:

- *Table Soundness*: The rule premises require that the constraints of the target chip C itself must hold for every row of its trace v_{tr}, given by $\Sigma(C)$. This is a crucial computation: a lookup into a table is only meaningful if the table itself represents a valid computation.
- *Row Existence*: There must exist at least one row i' in the target trace v_{tr} that simultaneously matches all the key-value pairs provided in the lookup. This is expressed by the clause $\exists i' \in [0, \mathrm{height}(v_{\mathrm{tr}} - 1)]$ such that for every lookup entry $\ell \in \{1, \ldots, k\}$, the value of f_ℓ matches the value of t_ℓ in the lookuped trace at row i'. This ensures the integrity of the cross-chip reference.

3.5 Runwai: Type System

The cornerstone of Runwai is its type system, summarized in Fig. 6, which leverages refinement types to provide formal, static guarantees about the correctness of ZK circuits. The typing judgement $\Delta; \Gamma \vdash e : \tau$ asserts that expression e has type τ under the chip signature context Δ and the variable typing context Γ. The primary goal is to prove at compile time that the specified constraints are logically consistent and that certain vulnerabilities (e.g., integer overflows, out-of-bounds access) are absent, without needing to execute the circuit or generate a proof.

Refinement Types: A type of the form $\{\nu : T \mid \phi\}$ consists of a base type T (e.g., `Int`, `Field`) and a logical predicate ϕ that constrains the possible values v of that type. For instance, $\{\nu : \mathtt{UInt} \mid \nu < 100\}$ is the type for an unsigned integer whose value is provably less than 100. These predicates are drawn from a decidable logic, enabling efficient verification via theorem provers.

TE-Assert: The typing rule for `assert_eq`(e_1, e_2) is central to the system's utility. It assigns the type $\{\nu : \mathtt{Unit} \mid e_1 = e_2\}$. While the base type is `Unit`, the refinement predicate $e_1 = e_2$ is added to the typing context for all subsequent expressions. This allows the type checker to learn and propagate equality constraints. For example, after `assert_eq`$(x, y * z)$, the type checker can freely substitute x with $y * z$ in later type derivations, which is essential for proving complex properties of a circuit.

TE-Branch: The rule for conditionals demonstrates how the type system reasons about program paths. The resulting refinement predicate is a disjunction: $(e_c \land \phi_1) \lor ((\neg e_c) \land \phi_2)$. This precisely captures the fact that if the program terminates, its state must satisfy the properties of the 'then' branch if the condition was true, or the properties of the 'else' branch if the condition was false.

TE-Lookup: This rule types the lookup construct and, like `TE-Assert`, enriches the context with new facts. The resulting type is $\{\nu : \mathtt{Unit} \mid \exists x'_{\mathtt{tr}} : \tau_{\mathtt{tr}}. \exists x'_{\mathtt{i}} : \tau_{\mathtt{i}}. \ \phi[x_{\mathtt{tr}} \mapsto x'_{\mathtt{tr}}, x_i \mapsto x'_i] \land (\forall l \in 1..k, f_l = t'_l)\}$. This predicate states two things:

(1) The post-condition ϕ of the looked-up chip C holds, with its symbolic variables substituted with the actual arguments used in the lookup. This allows the caller to assume the properties guaranteed by the lookup table.
(2) The values being looked up (f_l) are provably equal to the values at the target columns in the table (t'_l). This allows the type checker to reason about the data being cross-referenced between chips.

$$\frac{}{\Sigma; \sigma \vdash v \Downarrow v}\ (\text{E-}\textsc{Value}) \qquad \frac{\sigma(x) = v}{\Sigma; \sigma \vdash x \Downarrow v}\ (\text{E-}\textsc{Var})$$

$$\frac{\Sigma; \sigma \vdash e_1 \Downarrow v_a \quad \Sigma; \sigma \vdash e_2 \Downarrow n \quad n \in \mathbb{N} \quad n < \mathtt{len}(v_a) \quad v_a[n] = v}{\Sigma; \sigma \vdash e_1[e_2] \Downarrow v}\ (\text{E-}\textsc{ArrIdx})$$

$$\frac{\Sigma; \sigma \vdash e_1 \Downarrow f_1 \quad \Sigma; \sigma \vdash e_2 \Downarrow f_2 \quad f_1 \circledast f_2 = f}{\Sigma; \sigma \vdash e_1 \circledast e_2 \Downarrow f}\ (\text{E-FB}\textsc{inop}) \qquad \frac{\Sigma; \sigma \vdash e_1 \Downarrow n_1 \quad \Sigma; \sigma \vdash e_2 \Downarrow n_2 \quad n_1 \circledcirc n_2 = n}{\Sigma; \sigma \vdash e_1 \circledcirc e_2 \Downarrow n}\ (\text{E-UI}\textsc{nt}\text{B}\textsc{inop})$$

$$\frac{\Sigma; \sigma \vdash e_1 \Downarrow z_1 \quad \Sigma; \sigma \vdash e_2 \Downarrow z_2 \quad z_1 \ominus z_2 = z}{\Sigma; \sigma \vdash e_1 \ominus e_2 \Downarrow z}\ (\text{E-}\textsc{IntBinop}) \qquad \frac{\Sigma; \sigma \vdash e_1 \Downarrow b_1 \quad \Sigma; \sigma \vdash e_2 \Downarrow b_2 \quad b_1 \odot b_2 = b}{\Sigma; \sigma \vdash e_1 \odot e_2 \Downarrow b}\ (\text{E-}\textsc{BoolBinop})$$

$$\frac{}{\Sigma; \sigma \vdash \lambda x : \_.e \Downarrow \text{Closure}(\lambda x : \_.e, \sigma)}\ (\text{E-}\textsc{Lam}) \qquad \frac{\Sigma; \sigma \vdash e_1 \Downarrow f \quad \Sigma; \sigma \vdash e_2 \Downarrow f \quad f \in \mathbb{F}}{\Sigma; \sigma \vdash \mathtt{assert\_eq}(e_1, e_2) \Downarrow \mathtt{unit}}\ (\text{E-}\textsc{Assert})$$

$$\frac{\Sigma; \sigma \vdash e_1 \Downarrow v \quad \Sigma; \sigma \vdash e_2 \Downarrow v \quad (v \in \mathbb{F}) \lor (v \in \mathbb{Z}) \lor (v \in \mathbb{B})}{\Sigma; \sigma \vdash e_1 = e_2 \Downarrow \mathtt{true}}\ (\text{E-}\textsc{EqTrue})$$

$$\frac{\Sigma; \sigma \vdash e_1 \Downarrow v_1 \quad \Sigma; \sigma \vdash e_2 \Downarrow v_2 \quad (v_1, v_2) \notin \oslash}{\Sigma; \sigma \vdash e_1 \oslash e_2 \Downarrow \mathtt{false}}\ (\text{E-}\textsc{EqFalse})$$

$$\frac{\Sigma; \sigma \vdash e_1 \Downarrow v_1 \quad \Sigma; \sigma \vdash e_2 \Downarrow v_2 \quad v_1, v_2 \in \mathbb{Z} \quad \oslash \in \{<, \leq\} \quad (v_1, v_2) \in \oslash}{\Sigma; \sigma \vdash e_1 \oslash e_2 \Downarrow \mathtt{true}}\ (\text{E-}\textsc{IneqTrue})$$

$$\frac{\Sigma; \sigma \vdash e_1 \Downarrow v_1 \quad \Sigma; \sigma \vdash e_2 \Downarrow v_2 \quad v_1, v_2 \in \mathbb{Z} \quad \oslash \in \{<, \leq\} \quad (v_1, v_2) \notin \oslash}{\Sigma; \sigma \vdash e_1 \oslash e_2 \Downarrow \mathtt{false}}\ (\text{E-}\textsc{IneqFalse})$$

$$\frac{\Sigma; \sigma \vdash e_1 \Downarrow \text{Closure}(\lambda x : \_.e, \sigma') \quad \Sigma; \sigma \vdash e_2 \Downarrow v \quad \sigma'[x \rightarrow v] \vdash e \Downarrow v'}{\Sigma; \sigma \vdash e_1\ e_2 \Downarrow v'}\ (\text{E-}\textsc{App})$$

$$\frac{\Sigma; \sigma \vdash e_c \Downarrow \mathtt{true} \quad \Sigma; \sigma \vdash e_1 \Downarrow v_1}{\Sigma; \sigma \vdash \mathtt{if}\ e_c\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow v_1}\ (\text{E-}\textsc{IfTrue}) \qquad \frac{\Sigma; \sigma \vdash e_c \Downarrow \mathtt{false} \quad \Sigma; \sigma \vdash e_2 \Downarrow v_2}{\Sigma; \sigma \vdash \mathtt{if}\ e_c\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow v_2}\ (\text{E-}\textsc{IfFalse})$$

$$\frac{\begin{array}{c}\Delta(C) = \mathtt{chip}(x_{\mathtt{tr}} : \tau_{\mathtt{tr}}, x_i : \tau_{\mathtt{i}}) \longrightarrow \{\nu : \mathtt{Unit} \mid \phi_{\mathtt{chip}}\}\{e_{\mathtt{chip}}\} \\ \Sigma; \sigma \vdash e_{\mathtt{tr}} \Downarrow v_{\mathtt{tr}} \qquad \Sigma; \sigma \vdash e_{\mathtt{i}} \Downarrow v_{\mathtt{i}} \qquad \Sigma; \sigma[x_{\mathtt{tr}} \rightarrow v_{\mathtt{vr}}, x_{\mathtt{i}} \rightarrow v_{\mathtt{i}}] \vdash e_{\mathtt{chip}} \Downarrow v\end{array}}{\Sigma; \sigma \vdash \#C\ e_{\mathtt{tr}}\ e_{\mathtt{i}} \Downarrow v}\ (\text{E-}\textsc{Chip})$$

$$\frac{\begin{array}{c}\Delta(C) = \mathtt{chip}(x_{\mathtt{tr}} : \tau_{\mathtt{tr}}, x_i : \tau_{\mathtt{i}}) \longrightarrow \{\nu : \mathtt{Unit} \mid \phi_{\mathtt{chip}}\}\{e_{\mathtt{chip}}\} \quad \Sigma(C) = v_{\mathtt{tr}} \\ \forall v_{\mathtt{i}} \in [0, \mathtt{height}(v_{\mathtt{tr}}) - 1].\ \Sigma; \sigma \vdash \#C\ v_{\mathtt{tr}}\ v_{\mathtt{i}} \Downarrow \mathtt{unit} \\ \exists v_{\mathtt{i}} \in [0, \mathtt{height}(v_{\mathtt{tr}}) - 1].\ \forall \ell \in \{1, \ldots, k\}.\ \Sigma; \sigma \vdash f_\ell \Downarrow v_\ell \land \Sigma; \sigma \vdash v_{\mathtt{tr}}[v_{\mathtt{i}}][t_\ell] \Downarrow v_\ell \quad t_\ell \in \mathbb{N} \ \ v_\ell \in \mathbb{F}\end{array}}{\Sigma; \sigma \vdash \mathtt{lookup}\ \ C(f_1 : t_1, \ldots f_k : t_k) \Downarrow \mathtt{unit}}\ (\text{E-}\textsc{Lookup})$$

Fig. 5. Semantics Rules of Runwai Expressions

TE-Sub: This rule allows a value with a more specific type to be used where a more general type is expected. An expression e with type T' can be given type T if T' is a subtype of T ($\Gamma \models T' <: T$). The formal subtyping rule is shown in Fig. 7. The TSub-Refl and TSub-Trans establish that the relation is reflexive and transitive. For functions, TSub-FUN is contravariant in its argument type and covariant in its return type. For arrays, TSub-Array is covariant with respect to the element type. TSub-Refine is the core rule that allows us to convert the typing checking to the logical validity. Specifically, a type $\{\nu : T_1 \mid \phi_1\}$ is a subtype of $\{\nu : T_2 \mid \phi_2\}$ if two conditions are met: first the base type T_1 must be a subset of T_2, and second, the logical predicate ϕ_1 must logically imply ϕ_2 under the current typing context Γ. This implication check can be reduced to verification of the logical condition, which we can solve via the theorem-proving system.

3.6 Proof Obligations in Lean4

A key challenge in verifying ZK circuits is that they require reasoning about number-theoretic properties and polynomial equations over large prime fields, which are tasks that exceed the capabilities of standard SMT solvers. Although some SMT solvers provide domain-specific support for finite fields, they typically rely on Gröbner basis computation, which is doubly exponential in the worst case and thus impractical for complex, real-world ZK circuits. Like CODA, Runwai addresses this limitation by integrating with an interactive theorem prover. Whereas CODA is built on Coq, Runwai adopts Lean4, leveraging the recent progress and growing community effort around Lean4 in formalizing theories relevant to zero-knowledge proofs [27].

Generating Verification Conditions. The source of these proof obligations is the TE-SUB typing rule, which in turn relies on the subtyping rule TSub-Refine. This rule states that a refinement type $\{T_1 \mid \phi_1\}$ is a subtype of $\{T_2 \mid \phi_2\}$ only if the underlying logical predicates are validly entailed. This generates a logical validity query. Following the notation used in [21], we can formalize this query using semantic interpretation functions. Let $[\![\Gamma]\!]_\sigma$ be the semantic proposition that the typing environment Γ is valid under a given valuation σ. Let $[\![\phi]\!]_\sigma(v)$ be the semantic proposition that the syntactic predicate ϕ holds true for a value v under valuation σ. The TSub-Refine rule thus translates the subtyping check into the following native Lean Prop:

$$\forall \sigma, T, v, \quad [\![\Gamma]\!]_\sigma \to ([\![\phi_1]\!]_\sigma(v) \to [\![\phi_2]\!]_\sigma(v))$$

Automated Discharge via `runwai_vcg`. Discharging these obligations manually is tedious due to the verbose nature of operational semantics. Runwai automates this with a specialized tactic, `runwai_vcg`. Unlike simple rewriting scripts, `runwai_vcg` implements a custom saturation loop that significantly reduces the proof search space through three key mechanisms:

(1) *Exploiting Determinism*: In circuit constraints, the same sub-expression often appears in multiple branches or constraints. `runwai_vcg` scans the local context for distinct evaluation hypotheses of the same expression, and unifies these values automatically. This effectively implements Common Subexpression Elimination (CSE) at the proof level.
(2) *Value Canonicalization*: The tactic inspects the structure of expressions to infer the necessary constructor of their resulting values. For instance, if an expression is a field expression, `runwai_vcg` automatically proves that its result must be finite field value, eliminating impossible cases from the goal state.
(3) *Recursive Destruction*: The tactic recursively breaks down complex evaluation hypotheses into their atomic constituents (e.g., decomposing a Let evaluation into the evaluation of the bound value and the body), exposing the underlying arithmetic constraints to Lean's `simp` and `linarith` tactics.

$$\frac{\Gamma(x) = \{\nu : T \mid \phi\}}{\Delta;\Gamma \vdash x : \{\nu : T \mid \nu = x\}}\ (\textsc{TE-Var}), \qquad \frac{\Gamma(f) = x : \tau_1 \rightarrow \tau_2}{\Delta;\Gamma \vdash f : (x : \tau_1 \rightarrow \tau_2)}\ (\textsc{TE-VarFunc})$$

$$\frac{f \in \mathbb{F}}{\Delta;\Gamma \vdash f : \{\nu : \mathtt{Field} \mid \nu = f\}}\ (\textsc{TE-ConstField}), \qquad \frac{b \in \mathbb{B}}{\Delta;\Gamma \vdash b : \{\nu : \mathtt{Bool} \mid \nu = b\}}\ (\textsc{TE-ConstBool})$$

$$\frac{n \in \mathbb{N}}{\Delta;\Gamma \vdash n : \{\nu : \mathtt{UInt} \mid \nu = n\}}\ (\textsc{TE-ConstUInt}) \qquad \frac{z \in \mathbb{Z}}{\Delta;\Gamma \vdash z : \{\nu : \mathtt{Int} \mid \nu = z\}}\ (\textsc{TE-ConstInt})$$

$$\frac{\Delta; x : \tau_1, \Gamma \vdash e : \tau_2}{\Delta;\Gamma \vdash \lambda x : \tau_1 .\, e : (x : \tau_1 \rightarrow \tau_2)}\ (\textsc{TE-Abs}), \qquad \frac{\Delta;\Gamma \vdash x_1 : (s : \tau_s \rightarrow \tau_r), \quad \Delta;\Gamma \vdash x_2 : \tau_s}{\Delta;\Gamma \vdash x_1\ x_2 : \tau_r[s \mapsto x_2]}\ (\textsc{TE-App})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \{\nu : \mathtt{Field} \mid \phi_1\} \quad \Delta;\Gamma \vdash e_2 : \{\nu : \mathtt{Field} \mid \phi_2\}}{\Delta;\Gamma \vdash e_1 \circledast e_2 : \{\nu : \mathtt{Field} \mid \nu = e_1 \circledast e_2\}}\ (\textsc{TE-BinopField})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \{\nu : \mathtt{UInt} \mid \phi_1\} \quad \Delta;\Gamma \vdash e_2 : \{\nu : \mathtt{UInt} \mid \phi_2\}}{\Delta;\Gamma \vdash e_1 \circledcirc e_2 : \{\nu : \mathtt{UInt} \mid \nu = e_1 \circledcirc e_2\}}\ (\textsc{TE-BinopUInt})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \{\nu : \mathtt{Int} \mid \phi_1\} \quad \Delta;\Gamma \vdash e_2 : \{\nu : \mathtt{Int} \mid \phi_2\}}{\Delta;\Gamma \vdash e_1 \ominus e_2 : \{\nu : \mathtt{Int} \mid \nu = e_1 \ominus e_2\}}\ (\textsc{TE-BinopInt})$$

$$\frac{\Delta;\Gamma \vdash e_c : \{\nu : \mathtt{Bool} \mid \phi_c\}, \Delta;\Gamma \vdash e_1 : \{\nu : \tau \mid \phi_1\}, \quad \Delta;\Gamma \vdash e_2 : \{\nu : \tau \mid \phi_2\}}{\Delta;\Gamma \vdash \mathtt{if}\ e_c\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \{\nu : \tau \mid (e_c \wedge \phi_1) \vee ((\neg e_c) \wedge \phi_2)\}}\ (\textsc{TE-Branch})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \qquad \Delta;\Gamma : x : \tau_1 \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash \mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ e_2 : \tau_2[x \mapsto e_1]}\ (\textsc{TE-LetIn}), \qquad \frac{\Delta;\Gamma \vdash e : T', \quad \Gamma \models T' <: T}{\Delta;\Gamma \vdash e : T}\ (\textsc{TE-Sub})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \{\nu : \mathtt{Field} \mid \phi_1\}, \quad \Delta;\Gamma \vdash e_2 : \{\nu : \mathtt{Field} \mid \phi_2\}}{\Delta;\Gamma \vdash \mathtt{assert\_eq}(e_1, e_2) : \{\nu : \mathtt{Unit} \mid e_1 = e_2\}}\ (\textsc{TE-Assert}) \qquad \frac{}{\Delta;\Gamma \vdash \mathtt{unit} : \{\nu : \mathtt{Unit} \mid \mathtt{true}\}}\ (\textsc{TE-Unit})$$

$$\frac{\Delta(C) = \mathtt{chip}(x_{\mathtt{tr}} : \tau_{\mathtt{tr}}, x_i : \tau_{\mathtt{i}}) \rightarrow \{\nu : \mathtt{Unit} \mid \phi_{\mathtt{chip}}\}\{e_{\mathtt{chip}}\}}{\Delta : \Gamma \vdash \#C : (x_{\mathtt{tr}} : \tau_{\mathtt{tr}} \rightarrow x_{\mathtt{i}} : \tau_{\mathtt{i}} \rightarrow \{\nu : \mathtt{Unit} \mid \phi_{\mathtt{chip}}\})}\ (\textsc{TE-Chip})$$

$$\frac{\begin{array}{c} \Delta(C) = \mathtt{chip}(x_{\mathtt{tr}} : \tau_{\mathtt{tr}}, x_i : \tau_{\mathtt{i}}) \rightarrow \{\nu : \mathtt{Unit} \mid \phi_{\mathtt{chip}}\}\ \{e_{\mathtt{chip}}\} \\ \forall \ell \in 1 \ldots k.\ \Delta;\Gamma \vdash f_\ell : \{\nu : \mathtt{Field} \mid \phi_\ell\}, \quad t_\ell \in \mathbb{N} \\ \phi' = \exists x'_{\mathtt{tr}} : \tau_{\mathtt{tr}}.\, \exists x'_{\mathtt{i}} : \tau_{\mathtt{i}}.\ \ \phi_{\mathtt{chip}}[x_{\mathtt{tr}} \mapsto x'_{\mathtt{tr}}, x_i \mapsto x'_i] \wedge (\forall \ell \in 1 \ldots k.\ f_\ell = x'_{\mathtt{tr}}[x'_i][t_\ell]) \end{array}}{\Delta;\Gamma \vdash \mathtt{lookup}\ \ C\,(f_1 : t_1, \ldots, f_k : t_k)\ \ : \{\nu : \mathtt{Unit} \mid \phi'\}}\ (\textsc{TE-Lookup})$$

Fig. 6. Selected typing rules for Runwai.

Typing Automation. Complementing the VCG, Runwai provides the `autoTy` tactic to automate the application of typing rules. It heuristically applies subtyping lemmas, resolves environment lookups, and handles arithmetic refinements, allowing developers to focus on the verification of arithmetic properties.

$$\frac{}{\Gamma \models \tau <: \tau}\ (\textsc{TSub-Refl}), \qquad \frac{\Gamma \models \tau_1 <: \tau_2, \quad \Gamma \models \tau_2 <: \tau_3}{\Gamma \models \tau_1 <: \tau_3}\ (\textsc{TSub-Trans}) \qquad \frac{\Gamma \models T_1 <: T_2}{\Gamma \models [T_1] <: [T_2]}\ (\textsc{TSub-Array})$$

$$\frac{\begin{array}{c}\Gamma \models T_1 <: T_2, \quad P \equiv [\![\phi_1]\!] \Rightarrow [\![\phi_2]\!] \\ \forall \vec{x} \in \mathrm{dom}(\Gamma).\ \forall v.\ \mathrm{Encode}(\Gamma) \Rightarrow P\end{array}}{\Gamma \models \{v : T_1 \mid \phi_1\} <: \{v : T_2 \mid \phi_2\}}\ (\textsc{TSub-Refine}) \qquad \frac{\begin{array}{c}\Gamma \models \tau_y <: \tau_x \\ z \text{ fresh}, \quad \Gamma \models \tau_r[x \to z] <: \tau_s[y \to z]\end{array}}{\Gamma \models (x : \tau_x \to \tau_r) <: (y : \tau_y \to \tau_s)}\ (\textsc{TSub-Fun})$$

Fig. 7. Subtyping rules for Runwai

For example, consider the `IsZero` chip. It asserts the constraints `y = ((Field 0 - x) * inv) + Field 1` and `x*y = Field 0`. To prove this chip meets its goal refinement `trace[i][1] == if trace[i][0] == Field 0 then {Field 1} else {Field 0}`, the `TSub-Refine` rule generates a proof obligation equivalent to the following Lean proposition:

$$\forall (x,\ y,\ inv \in \mathbb{F}).\ (y = ((0 - x) \cdot inv) + 1) \land (x \cdot y = 0) \longrightarrow (y = \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ 0)$$

3.7 Theoretical Guarantee of Well-Types Runwai Programs

Similar to CODA [21], we establish a type preservation guarantee for a well-typed Runwai program. Specifically, if an expression e is well-typed with type τ, and e evaluates to a value v, then v has the same type τ, assuming that the valuations are consistent with their corresponding typing environments. We denote this consistency as $\Delta; \Gamma \models \sigma$, meaning that every variable in the valuation σ has the same type as in the static contexts Γ and Δ.

Theorem 1 (Expression Type Preservation of Runwai). *If 1)* $\Delta; \Gamma \vdash e : \tau$, *2)* $\Sigma; \sigma \vdash e \Downarrow v$, *and 3)* $\Delta; \Gamma \models \sigma$, *then* $\Delta; \Gamma \vdash v : \tau$

The proof follows the structure of CODA's expression type preservation theorem and is provided in the Appendix A.

4 COMPILATION TO ARITHMETIC CONSTRAINTS

This section details the Runwai compiler, which lowers Runwai programs into Algebraic Intermediate Representation (AIR) and lookup constraints. By targeting these standard constraint forms, Runwai enables users to leverage existing, highly optimized Zero-Knowledge Proof (ZKP) backends, facilitating the deployment of certified constraints in production-grade zkVM environments.

Constraint Definitions. We define the target constraint system over a finite field $\mathbb{F}$. An AIR constraint is a polynomial $p \in \mathbb{F}[\vec{x}]$ that must vanish (i.e., $p(\vec{x}) = 0$) for a valid trace. A lookup constraint is a tuple $\ell = \langle \vec{u}, \vec{t}, m \rangle$, consisting of: 1) $\vec{u}$, a vector of symbolic expressions representing the inputs (linear combinations of columns from the caller); 2) $\vec{t}$, a vector of column indices representing the table (the callee); and 3) m, a symbolic expression representing the multiplicity (how many times the lookup is invoked).

While general lookup arguments support linear combinations on the callee side, we restrict $\vec{t}$ to column indices for simplicity. This formulation is sufficient to capture the semantics of most modern zkVMs.

Compilation Rules. We formalize the compiler via a judgment based on partial evaluation. Drawing inspiration from CODA [21], we employ a symbolic valuation σ and a *path polynomial* ω. The judgment takes the form:

$$\sigma; \omega \vdash e \rightsquigarrow \langle \Phi, \Psi, v \rangle$$

$$\frac{}{\sigma;\omega \vdash u \rightsquigarrow \langle \emptyset, \emptyset, u \rangle}\ \text{(C-Value)}$$

$$\frac{\sigma;\omega \vdash e_1 \rightsquigarrow \langle \Phi_1, \Psi_1, u_1 \rangle \qquad \sigma;\omega \vdash e_2 \rightsquigarrow \langle \Phi_2, \Psi_2, u_2 \rangle \qquad u_1, u_2 \in \mathbb{F} \qquad u_3 = u_1 \otimes u_2}{\sigma;\omega \vdash e_1 \otimes e_2 \rightsquigarrow \langle \Phi_1 \cup \Phi_2, \Psi_1 \cup \Psi_2, u_3 \rangle}\ \text{(C-Reducible)}$$

$$\frac{\sigma;\omega \vdash e_1 \rightsquigarrow \langle \Phi_1, \Psi_1, u_1 \rangle \qquad \sigma;\omega \vdash e_2 \rightsquigarrow \langle \Phi_2, \Psi_2, u_2 \rangle \qquad u_1 \notin \mathbb{F} \vee u_2 \notin \mathbb{F}}{\sigma;\omega \vdash e_1 \otimes e_2 \rightsquigarrow \langle \Phi_1 \cup \Phi_2, \Psi_1 \cup \Psi_2, u_1 \otimes u_2 \rangle}\ \text{(C-Irreducible)}$$

$$\frac{\sigma;\omega \vdash e_1 \rightsquigarrow \langle \Phi_1, \Psi_1, u_1 \rangle \quad \sigma;\omega \vdash e_2 \rightsquigarrow \langle \Phi_2, \Psi_2, u_2 \rangle}{\sigma;\omega \vdash \texttt{assert\_eq}(e_1, e_2) \rightsquigarrow \langle \Phi_1 \cup \Phi_2 \cup \{\omega \cdot (u_1 - u_2) = 0\}, \Psi_1 \cup \Psi_2, \texttt{unit} \rangle}\ \text{(C-Assert)}$$

$$\frac{e_1 = \texttt{assert\_eq}(0, 0) \qquad b_1, b_2 \in \mathbb{F} \qquad \sigma;\ \omega \cdot (b_1 - b_2) \vdash e_2 \rightsquigarrow \langle \Phi, \Psi, \texttt{unit} \rangle}{\sigma;\omega \vdash \texttt{if}\ b_1 = b_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \langle \Phi, \Psi, \texttt{unit} \rangle}\ \text{(C-Branch)}$$

$$\frac{\forall \ell \in \{1 \ldots k\}.\ \sigma;\omega \vdash f_\ell \rightsquigarrow \langle \Phi_\ell, \Psi_\ell, u_\ell \rangle \quad t_\ell \in \mathbb{N}}{\sigma;\omega \vdash \texttt{lookup}\ C\,(f_1 : t_1, \ldots, f_k : t_k) \rightsquigarrow \left\langle \bigcup_\ell \Phi_\ell,\ (\bigcup_\ell \Psi_\ell) \cup \{\langle (u_1, \ldots, u_k), (t_1, \ldots, t_k), \omega \rangle\},\ \texttt{unit} \right\rangle}\ \text{(C-Lookup)}$$

Fig. 8. Selected compilation rules for Runwai. The operator $\otimes$ denotes a binary field operation. Note that in C-Branch, the path polynomial is updated to $\omega \cdot (b_1 - b_2)$ for the recursive call, effectively activating the constraints of e_2 only when $b_1 \neq b_2$.

This states that under valuation σ and path polynomial ω, the expression e evaluates to a compiled value v, generating a set of AIR constraints Φ and lookup constraints Ψ.

Here, v represents the irreducible form of e (a symbolic expression or a concrete field element). The environment σ maps variables to their compiled values. The term ω represents the algebraic condition under which the current branch is active. For example, when traversing the `else` branch of nested conditionals `if x=y` and `if y=z`, the path condition is $(x \neq y) \wedge (y \neq z)$. We encode this algebraically as the polynomial $\omega = (x - y) \cdot (y - z)$. If the branch is inactive, ω evaluates to 0, trivially satisfying any constraints multiplied by it.

Figure 8 presents key compilation rules. The rules C-Reducible and C-Irreducible distinguish between concrete operations (constant folding) and symbolic construction. C-Assert generates an AIR constraint conditioned on the current path; the constraint $\omega \cdot (u_1 - u_2) = 0$ enforces equality only when the path is active (i.e., $\omega \neq 0$). Similarly, C-Lookup generates a lookup tuple using ω as the multiplicity. This effectively enables *branch-dependent lookups*, a prevalent pattern in zkVM architectures where lookups are only enforced when specific opcodes are executed.

Note that our compilation judgment is partial; it is undefined for expressions that cannot be lowered to valid AIR or lookup constraints. Furthermore, while many zkVM proving systems enforce degree limits on AIR polynomials for efficiency, the Runwai compiler decouples this check. We allow the backend prover to report degree violations, simplifying the compiler design while retaining flexibility for provers with varying degree capacities.

5 IMPLEMENTATION

We have implemented Runwai as a two-stage compiler framework that bridges formal verification with high-performance zero-knowledge proving. The system consists of a Verified Frontend implemented in Lean 4, which handles parsing, type-checking, and constraint certification, and a High-Performance Backend implemented in Rust, which handles witness generation and proof orchestration via the Plonky3 framework.

5.1 Verified Frontend (Lean4)

The core of Runwai is developed in the Lean 4 interactive theorem prover. This component is responsible for defining the DSL syntax, enforcing refinement typing rules, and generating the Intermediate Representation (IR).

Syntax and Semantics. We define the Abstract Syntax Tree (AST) of Runwai as an inductive data type in Lean. The AST supports standard arithmetic operations over finite fields, Boolean logic, and array manipulations. The operational semantics are encoded via `PropSemantics`, which maps Runwai expressions directly to Lean's native propositions (`Prop`), enabling the system to leverage Lean's existing logic kernel for consistency checks.

Refinement Type Checker. The type system is implemented as an inductive relation `TypeJudgment`. Unlike traditional compilers that emit errors, Runwai utilizes Lean's tactic framework to discharge verification conditions automatically. We implemented custom tactics, such as `auto_trace_index` and `auto_let_assert`, which automate the resolution of boilerplate proof obligations related to array bounds and let-binding assertions. This allows the developer to prove that a circuit satisfies its specification during the type-checking phase.

Constraint Compilation. Once a program is well-typed, it is serialized into a JSON-based Intermediate Representation. This serialization handles the lowering of high-level functional constructs of Runwai into flat constraint structures that the downstream ZK proving system can easily consume.

5.2 High-Performance Backend (Rust)

To ensure the system remains practical for real-world zkVM applications, we decouple the proving engine from the formal verification logic. The backend is written in Rust and serves as the execution runtime for the JSON IR produced by the frontend.

Plonky3 Integration. We leverage Plonky3, a toolkit for polynomial IOPs (Interactive Oracle Proofs), to handle the cryptographic heavy lifting. The Runwai backend parses the JSON IR and dynamically constructs an Air implementation using the `RunwaiAir` struct. This struct implements the `BaseAir` and `Air` traits required by Plonky3, mapping Runwai's symbolic constraints to the underlying field operations and matrix evaluations.

Lookup Arguments. Runwai supports lookups (e.g., for byte-range checks) by integrating with Plonky3's lookup facilities. The backend identifies lookup constraints in the IR and instantiates the corresponding `ByteRangeAir` or auxiliary tables during the proving phase.

Proving and Verification. The backend orchestrates the full STARK proving pipeline, including trace generation, commitment via Merkle trees, and FRI (Fast Reed-Solomon Interactive Oracle Proof of Proximity) based proving. By utilizing Rust, we maintain the performance characteristics of native zkVM implementations while ensuring that the constraints being proven are certified correct by the Lean frontend.

6 CASE STUDIES

To demonstrate Runwai's expressiveness and verification capabilities, we present two representative gadgets essential to modern zkVM architectures: a fundamental arithmetic unit (`IsZero`) and a complex range-checking unit (`KoalaBearWordRangeChecker`). Due to space constraints, we present abridged implementations and proofs here; the full formalizations are available in our repository.

Arithmetic Primitives: The `IsZero` *Gadget.* The `IsZero` gadget (Code 5) is a ubiquitous primitive in ZK circuits, used to assert whether a value x is zero without branching. The constraint implementation follows the standard algebraic pattern: $y = 1 - x \cdot x^{-1}$ and $x \cdot y = 0$. Crucially, the correctness proof (Code 6) is highly automated. The `autoTy` tactic automatically resolves the typing environment, while the underlying `runwai_vcg` tactic (invoked via `isZero_typing_soundness`) discharges the arithmetic implications, verifying that the algebraic constraints logically imply the conditional behavior specified in the refinement type.

```
#runwai_register chip IsZero(trace: [[Field: 3]: n], i : {v: UInt | v < n})
  -> {Unit| trace [i][1] == if trace [i][0] == Fp 0 then {Fp 1} else {Fp 0}} {
  let x = trace [i][0];
  let y = trace [i][1];
  let inv = trace [i][2];
  let u1 = assert_eq(y, ((Fp 0 - x) * inv) + Fp 1);
  let u2 = assert_eq(x*y, Fp 0);
  u2
}
```

Code 5. Runwai implementation of IsZero

```
#runwai_prove IsZero := by {
  autoTy "u2"
  apply isZero_typing_soundness
  repeat decide
  repeat rfl
  simp[Ast.renameTy]
}
```

Code 6. Runwai proof of IsZero

Complex Composite Gadgets: 32-bit Range Checker. Real-world zkVMs, such as those using the KoalaBear field (a 31-bit prime field $p = 2^{31} - 2^{24} + 1$), require rigorous range checking to handle 32-bit operations safely. Code 7 illustrates a `KoalaBearWordRangeChecker`, which verifies that a set of four bytes forms a valid 32-bit integer strictly less than the field modulus p. This example highlights two key features of Runwai:

- *Lookup Integration*: The chip uses the lookup construct (lines 22-25) to verify that input bytes (`alpha_0`...`alpha_3`) exist in a pre-computed `u8` table. The type system ensures these lookups are sound before the values are consumed by subsequent arithmetic operations.
- *Complex Specification*: The refinement type captures the exact arithmetic invariant: the integer interpretation of the weighted sum of the bytes must be strictly less than $2,130,706,433$ (the field modulus).

Verifying this chip manually would involve reasoning about dozens of polynomial constraints and bitwise relationships from first principles. To mitigate this complexity, Runwai provides a comprehensive library of verified lemmas specifically designed for finite field arithmetic. Instead of relying solely on generic proof search, Runwai accelerates verification by applying high-level theorems that map algebraic constraints directly to logical propositions. For instance, the proof utilizes the `bit_val_to_nat` lemma, which lifts boolean constraints from the finite field domain (e.g., $x(x-1)=0$) directly into properties of natural numbers ($x \in \{0,1\}$), allowing the solver to bypass expensive algebraic re-derivations.

```
#runwai_register chip koalabearWordRangeCheckerChip(trace, i, 18)
  -> {Unit| (toN(trace [i][0]) <+> (toN(trace [i][1]) <*> 256) <+> (toN(trace [i][2]) <*> 65536) <+> (toN(trace [i][3]) <*> 16777216)) < 2130706433 }
{
  let alpha_0 = trace [i][0];
  let alpha_1 = trace [i][1];
  let alpha_2 = trace [i][2];
  let alpha_3 = trace [i][3];
  let msb_decomp_0 = trace [i][4];
  let msb_decomp_1 = trace [i][5];
  let msb_decomp_2 = trace [i][6];
  let msb_decomp_3 = trace [i][7];
  let msb_decomp_4 = trace [i][8];
  let msb_decomp_5 = trace [i][8];
  let msb_decomp_6 = trace [i][10];
  let msb_decomp_7 = trace [i][11];
  let and_msb_decomp_0_to_2 = trace [i][12];
  let and_msb_decomp_0_to_3 = trace [i][13];
  let and_msb_decomp_0_to_4 = trace [i][14];
  let and_msb_decomp_0_to_5 = trace [i][15];
  let and_msb_decomp_0_to_6 = trace [i][16];
  let and_msb_decomp_0_to_7 = trace [i][17];
  let l0 = lookup u8(alpha_0 : 0);
  let l1 = lookup u8(alpha_1 : 0);
  let l2 = lookup u8(alpha_2 : 0);
  let l3 = lookup u8(alpha_3 : 0);
  let b1 = assert_eq(msb_decomp_0 * (msb_decomp_0 - Fp 1), Fp 0);
  let b2 = assert_eq(msb_decomp_1 * (msb_decomp_1 - Fp 1), Fp 0);
  let b3 = assert_eq(msb_decomp_2 * (msb_decomp_2 - Fp 1), Fp 0);
  let b4 = assert_eq(msb_decomp_3 * (msb_decomp_3 - Fp 1), Fp 0);
  let b5 = assert_eq(msb_decomp_4 * (msb_decomp_4 - Fp 1), Fp 0);
  let b6 = assert_eq(msb_decomp_5 * (msb_decomp_5 - Fp 1), Fp 0);
  let b7 = assert_eq(msb_decomp_6 * (msb_decomp_6 - Fp 1), Fp 0);
  let b8 = assert_eq(msb_decomp_7 * (msb_decomp_7 - Fp 1), Fp 0);
  let u1 = assert_eq(((((((msb_decomp_0 + (msb_decomp_1 * 2)) + (msb_decomp_2 * 4)) + (msb_decomp_3 * 8)) + (msb_decomp_4 * 16)) + (msb_decomp_5 * 32)) + (msb_decomp_6 * 64)) + (msb_decomp_7 * 128)), value_3);
  let u2 = assert_eq(msb_decomp_7, Fp 0);
  let u3 = assert_eq(and_msb_decomp_0_to_2, msb_decomp_0 * msb_decomp_1);
  let u4 = assert_eq(and_msb_decomp_0_to_3, and_msb_decomp_0_to_2 * msb_decomp_2);
  let u5 = assert_eq(and_msb_decomp_0_to_4, and_msb_decomp_0_to_3 * msb_decomp_3);
  let u6 = assert_eq(and_msb_decomp_0_to_5, and_msb_decomp_0_to_4 * msb_decomp_4);
  let u7 = assert_eq(and_msb_decomp_0_to_6, and_msb_decomp_0_to_5 * msb_decomp_5);
  let u8 = assert_eq(and_msb_decomp_0_to_7, and_msb_decomp_0_to_6 * msb_decomp_6);
  let u9 = assert_eq(Fp 0, and_msb_decomp_0_to_7 * value_0);
  let u10 = assert_eq(Fp 0, and_msb_decomp_0_to_7 * value_1);
  let u11 = assert_eq(Fp 0, and_msb_decomp_0_to_7 * value_2);
  u1
}
```

Code 7. Runwai implementation of koalabearWordRangeChecker

7 RELATED WORK

7.1 Refinement Types

Refinement types enrich a standard type system with logical predicates, allowing developers to specify invariants that are checked at compile time [18]. This approach has been successfully applied in general-purpose languages like LiquidHaskell and F* to verify program correctness.

In the domain of zero-knowledge proofs, CODA [21] pioneered the use of refinement types to certify ZK circuits.CODA allows developers to attach semantic specifications to Rank-1 Constraint Systems (R1CS) constraints, ensuring that the circuit implementation matches its high-level mathematical definition. However, CODA is fundamentally limited to R1CS and lacks support for the Algebraic Intermediate Representation (AIR) or lookup arguments. These features are the backbone of modern, high-performance zkVMs [1, 9]. Runwai builds upon the foundation laid by CODA but extends the refinement type paradigm to natively support the complex, multi-dimensional constraints and lookup tables required by modern architectures.

7.2 DSLs for Zero-Knowledge Proofs

The landscape of Domain Specific Languages (DSLs) for Zero-Knowledge Proofs is characterized by a trade-off between performance, simplicity, and safety.

Low-Level & Optimizable. Languages like Circom, AirScript, and the Plonky3 framework [16, 22, 31] provide developers with fine-grained control over constraint generation. These tools allow for highly optimized circuits essential for zkVMs but often lack semantic safeguards. Writing constraints in these frameworks requires manual management of polynomial alignments and witness generation, making them prone to under-constrained bugs and difficult to audit [33].

Simplicity & Abstraction. High-level languages such as Noir, Leo, and ZoKrates [3, 8, 10] prioritize developer experience, offering familiar imperative syntax. However, they abstract away the underlying constraint system—typically compiling to generic R1CS or PLONK gates. This abstraction introduces significant performance overhead and prevents the manual optimizations necessary for high-performance zkVM construction.

7.3 Verification and Bug Detection

Beyond language design, several tools aim to verify or detect bugs in ZK circuits. Static analyzers such as Circomspect [24] and ZKAP [33] can efficiently flag potential issues, but they often suffer from high false-positive rates. Approaches like ConsCS [19], Picus [25], CIVER [17], and AC4 [7] use SMT-solver-based formal verification to prove correctness, but their scalability is limited by the high cost of SMT reasoning over finite fields. Meanwhile, fuzzing-based tools such as zkFuzz [30] and ARGUZZ [15] are effective at quickly uncovering bugs, but they cannot guarantee correctness.

8 CONCLUSION

In this work, we presented Runwai, a refinement-typed domain-specific language designed to address the critical verification gap in Zero-Knowledge Virtual Machine (zkVM) development. By embedding formal specifications directly into the type system, Runwai resolves the long-standing trilemma between performance, simplicity, and safety. We demonstrated that it is possible to retain the low-level control required for efficient Algebraic Intermediate Representation (AIR) and lookup arguments while simultaneously enforcing rigorous, machine-checked correctness guarantees via the Lean 4 theorem prover.

Our implementation validates this approach through a dual-stage architecture: a verified frontend that discharges proof obligations during type-checking, and a high-performance Rust backend

Table 2. Comparison of Domain Specific Languages (DSLs) for ZKP

	Low-Level Optimizable	Simplicity of Syntax	Supprt AIR	Support Lookup	Comp. /Const. Seperation	Formal Verification
Circom [4]	✓	✗	✗	✗	✗	✗
Noir [3]	◐	✓	✗	✗	✗	✗
Zokrates [10]	◐	✓	✗	✗	✗	✗
CODA [21]	✓	◕	✗	✗	✓	✓
Airscript [22]	✓	✗	✓	✗	✓	✗
Plonky3 [31]	✓	✗	✓	✓	✓	✗
Zigren [28]	✓	✗	✓	✓	✓	✗
Clean [36]	✓	◕	✓	✓	✗	✓
Runwai (Ours)	✓	✓	✓	✓	✓	✓

that integrates seamlessly with the Plonky3 proving system. This design not only catches subtle under-constrained bugs at compile time but also ensures that the final executable circuits maintain the performance characteristics necessary for production-grade zkVMs.

Looking forward, we plan to expand the Runwai ecosystem in several directions. First, we aim to enhance the automation of our tactic library, reducing the manual proof burden for complex, stateful constraints. Second, we intend to develop a comprehensive standard library of certified gadgets, such as hash functions and signature verification chips, to accelerate secure zkVM development. Finally, we plan to apply Runwai to the formal verification of full instruction set architectures, such as RISC-V, moving closer to the ultimate goal of end-to-end verified zero-knowledge computing.

REFERENCES

[1] Arasu Arun, Srinath Setty, and Justin Thaler. 2024. Jolt: Snarks for virtual machines via lookups. In Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, 3–33.

[2] Jeremy Avigad, Lior Goldberg, David Levit, Yoav Seginer, and Alon Titelman. 2022. A verified algebraic representation of cairo program execution. In Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. 153–165.

[3] Aztec Network. 2024. Noir: The Universal Language of Zero-Knowledge. https://azt3c-st.webflow.io/noir. Accessed: February 23, 2025.

[4] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. 2022. Circom: A circuit description language for building zero-knowledge applications. IEEE Transactions on Dependable and Secure Computing 20, 6 (2022), 4733–4751.

[5] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. 2019. DEEP-FRI: sampling outside the box improves soundness. arXiv preprint arXiv:1903.12243 (2019).

[6] Vitalik Buterin. 2025. Long-term L1 execution layer proposal: replace the EVM with RISC-V. https://ethereum-magicians.org/t/long-term-l1-execution-layer-proposal-replace-the-evm-with-risc-v/23617. https://ethereum-magicians.org/t/long-term-l1-execution-layer-proposal-replace-the-evm-with-risc-v/23617 Accessed: 2025-08-21.

[7] Hao Chen, Guoqiang Li, Minyu Chen, Ruibang Liu, and Sinka Gao. 2024. AC4: Algebraic Computation Checker for Circuit Constraints in ZKPs. arXiv preprint arXiv:2403.15676 (2024).

[8] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. 2021. Leo: A programming language for formally verified, zero-knowledge applications. Cryptology ePrint Archive (2021).

[9] OpenVM Contributors. 2025. OpenVM Whitepaper. Whitepaper –. OpenVM, –. https://openvm.dev/whitepaper.pdf

[10] Zokrates Contributors. 2023. ZoKrates: A toolbox for zkSNARKs on Ethereum. https://github.com/Zokrates/ZoKrates GitHub repository; accessed 2025-11-07.

[11] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. 2019. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive (2019).

[12] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In Annual international conference on the theory and applications of cryptographic techniques. Springer, 305–326.

[13] Ulrich Haböck. 2022. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive (2022).

[14] Yahya Hassanzadeh-Nazarabadi and Sanaz Taheri-Boshrooyeh. 2025. Constraint-Level Design of zkEVMs: Architectures, Trade-offs, and Evolution. arXiv preprint arXiv:2510.05376 (2025).

[15] Christoph Hochrainer, Valentin Wüstholz, and Maria Christakis. 2025. Arguzz: Testing zkVMs for Soundness and Completeness Bugs. arXiv preprint arXiv:2509.10819 (2025).

[16] iden3. 2025. Library of basic circuits for circom. https://github.com/iden3/circomlib. Accessed: 2025-04-01.

[17] Miguel Isabel, Clara Rodríguez-Núñez, and Albert Rubio. 2024. Scalable Verification of Zero-Knowledge Protocols. In 2024 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, 133–133.

[18] Ranjit Jhala, Niki Vazou, et al. 2021. Refinement types: A tutorial. Foundations and Trends® in Programming Languages 6, 3–4 (2021), 159–317.

[19] Jinan Jiang, Xinghao Peng, Jinzhao Chu, and Xiapu Luo. 2025. ConsCS: Effective and Efficient Verification of Circom Circuits. In 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). IEEE Computer Society, 737–737.

[20] Ryan Lavin, Xuekai Liu, Hardhik Mohanty, Logan Norman, Giovanni Zaarour, and Bhaskar Krishnamachari. 2024. A Survey on the Applications of Zero-Knowledge Proofs. arXiv preprint arXiv:2408.00243 (2024).

[21] Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig, and Yu Feng. 2024. Certifying zero-knowledge circuits with refinement types. In 2024 IEEE Symposium on Security and Privacy (SP). IEEE, 1741–1759.

[22] Mikerah. 2024. AirScript: Scripting language for defining zk-STARKs. https://github.com/Mikerah/AirScript GitHub repository; accessed 2025-11-07.

[23] noir lang. 2025. ZK Benchmark. https://github.com/noir-lang/zk_bench. Accessed: 2025-04-03.

[24] Trail of Bits. 2024. Circomspect: A static analyzer and linter for the Circom zero-knowledge DSL. https://github.com/trailofbits/circomspect. Accessed: 2025-02-27.

[25] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. 2023. Automated detection of under-constrained circuits in zero-knowledge proofs. Proceedings of the ACM on Programming Languages 7, PLDI (2023), 1510–1532.

[26] Protocol Labs. 2023. The Future of ZK Proofs. https://www.protocol.ai/protocol-labs-the-future-of-zk-proofs.pdf. Accessed: 2025-02-18.

[27] Reilabs. 2025. proven-zk: A support library for working with zero knowledge cryptography in Lean 4. https://github.com/reilabs/proven-zk GitHub repository; accessed 2025-11-07.

[28] risc0. 2025. Zirgen: compiler and RISC Zero circuits. https://github.com/risc0/zirgen accessed: 2025-12-04.

[29] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized anonymous payments from bitcoin. In 2014 IEEE symposium on security and privacy. IEEE, 459–474.

[30] Hideaki Takahashi, Jihwan Kim, Suman Jana, and Junfeng Yang. 2025. zkFuzz: Foundation and Framework for Effective Fuzzing of Zero-Knowledge Circuits. arXiv preprint arXiv:2504.11961 (2025).

[31] Polygon Zero Team. 2023. Plonky3: A toolkit for polynomial IOPs (PIOPs). https://github.com/Plonky3/Plonky3/ GitHub repository; accessed 2025-11-07.

[32] Morgan Thomas, Mamy Ratsimbazafy, Marcin Bugaj, Lewis Revill, Carlo Modica, Sebastian Schmidt, Ventali Tan, Daniel Lubarov, Max Gillett, and Wei Dai. 2025. Valida ISA Spec, version 1.0: A zk-Optimized Instruction Set Architecture. arXiv preprint arXiv:2505.08114 (2025).

[33] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. 2024. Practical Security Analysis of {Zero-Knowledge} Proof Circuits. In 33rd USENIX Security Symposium (USENIX Security 24). 1471–1487.

[34] Zhibo Xing, Zijian Zhang, Jiamou Liu, Ziang Zhang, Meng Li, Liehuang Zhu, and Giovanni Russello. 2023. Zero-knowledge proof meets machine learning in verifiability: A survey. arXiv preprint arXiv:2310.14848 (2023).

[35] Youwei Zhong. 2024. A Parameterized Framework for the Formal Verification of Zero-Knowledge Virtual Machines. In Companion Proceedings of the 2024 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. 22–24.

[36] zkSecurity (Verified-zkEVM grant). 2025. clean: An embedded Lean 4 DSL for writing ZK circuits. https://github.com/Verified-zkEVM/clean GitHub repository; accessed 2025-11-07.

A PROOF OF EXPRESSION TYPE PRESERVATION

DEFINITION 3 (CONSISTENT ENVIRONMENT [21]). *The valuation typing relation, written as $\Delta;\Gamma \models \sigma$ is defined as:*

$$
\begin{aligned}
&\Delta;\Gamma \models \sigma ::= \\
&\quad \mathit{dom}(\Gamma) = \mathit{dom}(\sigma) \\
&\quad \land (\forall x, v, \tau.\ \ \Gamma(x) = \{v : T \mid \phi\} \land \sigma(x) = v \Rightarrow \Delta;\Gamma \vdash v : \{v : T \mid v = x\}) \\
&\quad \land (\forall x, v, \tau.\ \ \Gamma(x) = x' : \tau_1 \to \tau_2 \land \sigma(x) = v \Rightarrow \Delta;\Gamma \vdash v : x' : \tau_1 \to \tau_2)
\end{aligned}
$$

LEMMA 2 (SUBTYPING REFINEMENT INVERSION). *If $\Gamma \models \{v : T \mid \phi\} <: \tau$, then there exists a T', ϕ such that all of the following hold:*

- $\tau = \{v : T' \mid \phi'\}$
- $\Gamma \models T <: T'$
- $\models \forall \bar{x} \in \mathit{dom}(\Gamma).\ \forall v.\ \mathit{Encode}(\Gamma) \Rightarrow [\![\phi]\!] \Rightarrow [\![\phi']\!]$

PROOF. By induction on the derivation of $\Gamma \models \{v : T \mid \phi\} <: \tau$. See Lemma 4 of [21] for the details. □

LEMMA 3 (TYPE INVERSION). *If $\Delta;\Gamma \vdash e : \tau$, then there exists a type τ' such that $\Delta;\Gamma \vdash e : \tau'$ and $\Gamma \models \tau' <: \tau$*

PROOF. By induction on the derivation of $\Delta;\Gamma \vdash e : \tau$. See Lemma 7 of [21] for the details. □

PROOF OF THEOREM 1. By the induction on the derivation of $\Sigma;\sigma \vdash e \Downarrow v$, where we prove the following property: $\forall \Gamma, \tau.\ \Delta;\Gamma \vdash e : \tau \Rightarrow \Delta;\Gamma \vdash v : \tau$. As most cases are either trivial or the same as the proof for the CODA program in [21], we show only the cases for E-LOOKUP, which is a unique evaluation rule of Runwai.

Case E-LOOKUP. From the statement of Theorem 1, we have three premises:

$$\Delta;\Gamma \vdash \mathtt{lookup}\ \ C\,(f_1 : t_1, \dots, f_k : t_k) : \tau \tag{4}$$

$$\Sigma : \sigma \vdash \mathtt{lookup}\ \ C\,(f_1 : t_1, \dots, f_k : t_k) \Downarrow \mathtt{unit} \tag{5}$$

$$\Delta;\Gamma \models \sigma \tag{6}$$

Then, our goal is to show

$$\Delta : \Gamma \vdash \text{unit} : \tau \tag{7}$$

First, by Lemma 3 on Eq. 4, we know there exists a type τ' such that

$$\Delta;\Gamma \vdash \mathtt{lookup}\ \ C\,(f_1 : t_1, \dots, f_k : t_k) : \tau' \tag{8}$$

$$\Gamma \models \tau' <: \tau \tag{9}$$

From Eq. 8 and the definition of TE-LOOKUP, we know the exact form of τ':

$$\tau' = \{v : \mathtt{Unit} \mid \phi'\} \tag{10}$$

where

$$\Delta(C) = \mathtt{chip}(x_{\mathtt{tr}} : \tau_{\mathtt{tr}},\ x_i : \tau_{\mathtt{i}}) \to \{v : \mathtt{Unit} \mid \phi\}\ e \tag{11}$$

$$\phi' = \exists x'_{\mathtt{tr}} : \tau_{\mathtt{tr}}.\ \exists x'_{\mathtt{i}} : \tau_{\mathtt{i}}.\ \ \phi[x_{\mathtt{tr}} \mapsto x'_{\mathtt{tr}}, x_i \mapsto x'_i] \land (\forall \ell \in 1 \dots k,\ \ f_\ell = x'_{\mathtt{tr}}[x'_i][t_\ell]) \tag{12}$$

From Eq. 9 and TE-SUB rule, it suffices to prove the stronger statement to prove our goal:

$$\Delta;\Gamma \vdash \mathtt{unit} : \tau' \tag{13}$$

By TE-Unit rule, we can type the value unit:

$$\Delta; \Gamma \vdash \mathtt{unit} : \{\nu : \mathtt{Unit} : \mathtt{true}\} \tag{14}$$

By TE-Sub rule, our goal, Eq. 7, is achievable if we can prove the subtyping:

$$\Gamma \models \{\nu : \mathtt{Unit} \mid \mathtt{true}\} <: \tau' \tag{15}$$

By TSub-Refine rule, this subtyping holds if the following logical formula is valid:

$$\models \forall \vec{x} \in \mathrm{dom}(\Gamma).\ \forall \nu.\ \mathrm{Encode}(\Gamma) \Rightarrow \phi' \tag{16}$$

Given Eq. 5, we have the following from the premises of E-Lookup rule:

$$\Delta(C) = \mathtt{chip}(x_{\mathtt{tr}} : \tau_{\mathtt{tr}},\ x_i : \tau_{\mathtt{i}}) \longrightarrow \{\nu : \mathtt{Unit} \mid \phi\}\{e\} \tag{17}$$

$$\Sigma(C) = v_{\mathtt{tr}} \tag{18}$$

$$\forall_{v_{\mathtt{i}} \in [0, \mathtt{height}(v_{\mathtt{tr}})-1]}.\ \Sigma; \sigma \vdash \#C\ v_{\mathtt{tr}}\ v_{\mathtt{i}} \Downarrow \mathtt{unit} \tag{19}$$

$$\exists_{v_{\mathtt{i}} \in [0, \mathtt{height}(v_{\mathtt{tr}})-1]}.\ \forall_{\ell \in \{1,\ldots,k\}}.\ \Sigma; \sigma \vdash f_\ell \Downarrow v_\ell\ \wedge\ \Sigma; \sigma \vdash v_{\mathtt{tr}}[v_{\mathtt{i}}][t_\ell] \Downarrow v_\ell \tag{20}$$

, where $\forall_{\ell \in \{1,\ldots,k\}}$, we have $t_\ell \in \mathbb{N},\ v_\ell \in \mathbb{F}$. We then apply the inductive hypothesis of Theorem 1 to Eq. 19 at the specific matching row i_{match} that satisfies Eq. 20. Given TE-APP rule, we have the following premises of inductive hypothesis:

$$\Delta; \Gamma \vdash \#C\ v_{\mathtt{tr}}\ i_{\mathrm{match}} : \{\nu : \mathtt{Unit} \mid \phi[x_{\mathtt{tr}} \mapsto v_{\mathtt{tr}}, x_i \mapsto i_{\mathrm{match}}]\} \tag{21}$$

$$\Sigma; \sigma \vdash \#C\ v_{\mathtt{tr}}\ i_{\mathrm{match}} \Downarrow \mathtt{unit} \tag{22}$$

$$\Delta; \Gamma \models \sigma \tag{23}$$

such that

$$\forall_{\ell \in \{1,\ldots,k\}}.\ \Sigma; \sigma \vdash f_\ell \Downarrow v_\ell\ \wedge\ \Sigma; \sigma \vdash v_{\mathtt{tr}}[i_{\mathrm{match}}][t_\ell] \Downarrow v_\ell \tag{24}$$

Combining Eq. 21, 22, and 23 gives the bellow conclusion of inductive hypothesis:

$$\Delta; \Gamma \vdash \mathtt{unit} : \{\nu : \mathtt{Unit} \mid \phi[x_{\mathtt{tr}} \mapsto v_{\mathtt{tr}}, x_i \mapsto i_{\mathrm{match}}]\}\} \tag{25}$$

Since it is obvious that $\Gamma \models \{\nu : \mathtt{Unit} \mid \mathtt{True}\} <: \{\nu : \mathtt{Unit} \mid \phi[x_{\mathtt{tr}} \mapsto v_{\mathtt{tr}}, x_i \mapsto i_{\mathrm{match}}]\}\}$, we have the following proposition thanks to Lemma 2:

$$\models \forall \vec{x} \in \mathrm{dom}(\Gamma).\ \forall \nu.\ \mathrm{Encode}(\Gamma) \Rightarrow \mathtt{True} \Rightarrow [\![\phi[x_{\mathtt{tr}} \mapsto v_{\mathtt{tr}}, x_i \mapsto i_{\mathrm{match}}]\}\}]\!] \tag{26}$$

We now argue that $\forall_{\ell \in \{1,\ldots,k\}}, f_\ell = v_{\mathtt{tr}}[i_{\mathrm{match}}][t_\ell]$. By Lemma 3, we have:

$$\forall_{\ell \in \{1,\ldots,k\}}.\ \ \Delta; \Gamma \vdash f_\ell : \{\nu : \mathtt{Field} \mid \nu = f_\ell\} \tag{27}$$

$$\forall_{\ell \in \{1,\ldots,k\}}.\ \ \Delta; \Gamma \vdash v_{\mathtt{tr}}[i_{\mathrm{match}}][t_\ell] : \{\nu : \mathtt{Field} \mid \nu = v_{\mathtt{tr}}[i_{\mathrm{match}}][t_\ell]\} \tag{28}$$

Applying the inductive hypothesis for f_ℓ and $v_{\mathtt{tr}}[i_{\mathrm{match}}][t_\ell]$ gives:

$$\forall_{\ell \in \{1,\ldots,k\}}.\ \ \Delta; \Gamma \vdash v_\ell : \{\nu : \mathtt{Field} \mid \nu = f_\ell\} \tag{29}$$

$$\forall_{\ell \in \{1,\ldots,k\}}.\ \ \Delta; \Gamma \vdash v_\ell : \{\nu : \mathtt{Field} \mid \nu = v_{\mathtt{tr}}[i_{\mathrm{match}}][t_\ell]\} \tag{30}$$

By Lemma 3 on the above, we have:

$$\forall_{\ell \in \{1,\ldots,k\}}.\ \ \Gamma \models \{\nu : \mathtt{Field} \mid \nu = v_\ell\} <: \{\nu : \mathtt{Field} \mid \nu = f_\ell\} \tag{31}$$

$$\forall_{\ell \in \{1,\ldots,k\}}.\ \ \Gamma \models \{\nu : \mathtt{Field} \mid \nu = v_\ell\} <: \{\nu : \mathtt{Field} \mid \nu = v_{\mathtt{tr}}[i_{\mathrm{match}}][t_\ell]\} \tag{32}$$

Using Lemma 2 on the above yeilds:

$$\forall_{\ell \in \{1,\ldots,k\}}.\ \ \models \forall \vec{x} \in \mathrm{dom}(\Gamma).\ \forall \nu.\ \mathrm{Encode}(\Gamma) \Rightarrow [\![\nu = v_\ell]\!] \Rightarrow [\![\nu = f_\ell]\!] \tag{33}$$

$$\forall_{\ell \in \{1,\ldots,k\}}.\ \ \models \forall \vec{x} \in \mathrm{dom}(\Gamma).\ \forall \nu.\ \mathrm{Encode}(\Gamma) \Rightarrow [\![\nu = v_\ell]\!] \Rightarrow [\![\nu = v_{\mathtt{tr}}[i_{\mathrm{match}}][t_\ell]]\!] \tag{34}$$

Instantiating ν in both proposition above by setting $\nu = v_\ell$ gives the following proposition:

$$\forall_{\ell\in\{1,\ldots,k\}}.\quad \models \forall\vec{x}\in\mathrm{dom}(\Gamma).\ \mathrm{Encode}(\Gamma) \Rightarrow [\![v_\ell = v_\ell]\!] \Rightarrow [\![v_\ell = f_\ell \wedge v_\ell = v_{\mathtt{tr}}[i_{\mathrm{match}}][t_\ell]]\!] \tag{35}$$

, which is equivalent to

$$\models \forall\vec{x}\in\mathrm{dom}(\Gamma).\ \mathrm{Encode}(\Gamma) \Rightarrow \forall_{\ell\in\{1,\ldots,k\}}.\ \ [\![f_\ell = v_{\mathtt{tr}}[i_{\mathrm{match}}][t_\ell]]\!] \tag{36}$$

Combining Eq. 26 and 36 yeilds

$$\models \forall\vec{x}\in\mathrm{dom}(\Gamma).\ \ \forall\nu.\ \ \mathrm{Encode}(\Gamma) \Rightarrow \tag{37}$$

$$[\![\phi[x_{\mathtt{tr}} \mapsto v_{\mathtt{tr}}, x_i \mapsto i_{\mathrm{match}}]\}\}]\!] \wedge \forall_{\ell\in\{1,\ldots,k\}}.\ \ [\![f_\ell = v_{\mathtt{tr}}[i_{\mathrm{match}}][t_\ell]]\!] \tag{38}$$

, which immediately leads to Eq. 16. □