zkFlow: Verified Compilation of Programs to Zero-Knowledge Constraints

Written and Verified in Lean

Tanner Duve

May 6, 2025

Outline

Source Language: zkFlow

Target Language: zkLean

The Compiler

Verification and Correctness

Demo

Motivation: A High-Level Language for ZK

Problem: Computations for zero-knowledge proofs must be encoded by hand as polynomial constraints over some field.

Goal: Design language to write normal high-level programs and automatically get correct ZK constraints.

Challenge: Prove compiler is sound, i.e., *preserves semantics*.

Idea - Compile from simple expression language to ZK circuit language

Define a small programming language: arithmetic, booleans, control flow, assertions.

Compile programs into circuits (using zkLean DSL as target language).

Prove in Lean that the compiled circuit faithfully represents the source program.

Outline

Source Language: zkFlow

Target Language: zkLean

The Compiler

Verification and Correctness

Demo

Language Design - Syntax

Source language terms consist of

- Variables, Literals over a field F, and Booleans
- Arithmetic: $+, *, -$
- Logic: $\&\&, ||, !,$ inSet
- Control Flow: if, let, seq, assert

An example program:

```
<{let role := 1 in
      assert (role inn {1, 2, 3}) ;
      assert !(role == 2)}>
```

```
inductive ArithBinOp where
| add | sub | mul
deriving Inhabited, BEq, Repr
inductive BoolBinOp where
| and | or
deriving Inhabited, BEq, Repr

inductive Term (f : Type u) [Field f] where
| var : String → Term f
| lit : f → Term f
| bool : Bool → Term f
| arith : ArithBinOp → Term f → Term f → Term f
| boolB : BoolBinOp  → Term f → Term f → Term f
| eq  : Term f → Term f → Term f
| not : Term f → Term f
| lett : String → Term f → Term f → Term f
| ifz : Term f → Term f → Term f → Term f
| inSet : Term f → List f → Term f
| assert : Term f → Term f
| seq : Term f → Term f → Term f
deriving Inhabited, BEq
```

Before Semantics - Environments

To evaluate programs, first assign values to variables.

```
mutual
inductive Val (f : Type) [Field f] where
| Field    : f → Val f
| Bool     : Bool → Val f

structure Env (f : Type) [Field f] where
  lookup : String → Option (Val f)
end
```

Before Semantics - Environments

A term *t* is *well-scoped* in an environment env when every free variable in *t* is mapped to either a bool or a field element:

```
def wellScoped [Field f] (t : Term f) (env : Env f) : Prop :=
  ∀ x ∈ freeVars t, ∃ v, env.lookup x = some v
```

Where

```
def freeVars {f} [Field f] (t : Term f) : Finset String
```

is defined by simple recursion on *t*.

Language Design - Semantics

We define a *big step operational semantics* using an inductive relation:

```
Eval (f : Type) [Field f] [BEq f] : Term f → Env f → Val f → Prop
```

Semantics cont'd - some rules

When t is a variable:

$$\frac{\mathsf{env.lookup}(x) = \mathsf{some}(v) \quad t = \mathsf{var}(x)}{(t, \mathsf{env}) \longrightarrow v}\ \texttt{var}$$

When t checks membership:

$$\frac{(t, \mathsf{env}) \longrightarrow x \quad x \in ts}{(t\ \texttt{inn}\ ts, \mathsf{env}) \longrightarrow \mathsf{true}}\ \texttt{inSet\_true}$$

When t is $t_1 \bullet t_2$ for a binary operation $\bullet$:

$$\frac{(t_1, \mathsf{env}) \longrightarrow n_1 \quad (t_2, \mathsf{env}) \longrightarrow n_2}{(t_1 \bullet t_2, \mathsf{env}) \longrightarrow (n_1 \bullet n_2)}\ \texttt{BinOp}$$

When t is sequencing:

$$\frac{(t_1, \mathsf{env}) \longrightarrow v_1 \quad (t_2, \mathsf{env}) \longrightarrow v_2}{(t_1;\ t_2, \mathsf{env}) \longrightarrow v_2}\ \texttt{seq}$$

Outline

zkLean: Target Language for ZK Compilation

zkLean is a small expression language for building zero-knowledge circuits.

Programs rep. as `ZKExpr f`, representing field computations.

Expressions include:

- Literals
- Witness variables
- Arithmetic: $+, -, *, \text{neg}$
- Equality constraints
- Lookup operations (for Jolt-style lookup arguments)

ZKExpr Evaluation as Circuits

Each `ZKExpr` is interpreted as a circuit constraint.

Expressions over literals and witness variables build polynomials:

- `Add`, `Mul`, `Sub`, `Neg` → algebraic constraints
- `Eq` → enforces equality between expressions

Witnesses provide variable assignments satisfying all constraints.

Lookup tables provide efficient range checks and indirect computations.

Witness Semantics and Constraint Checking

Witness values assigned by the prover at proving time.

Evaluation:

```
def semantics_zkexpr [Field f] (exprs: ZKExpr f) (witness: List f) :
    Value f
```

Constraints are enforced by evaluating expressions and checking they hold:

```
def constraints_semantics [Field f] (constraints: List (ZKExpr f))
    (witness: List f) : Bool
```

State monad `ZKBuilder`:

- Allocates witnesses
- Keeps track of constraints

Outline

The Compiler: From Programs to Circuits

Compiling from source terms (`Term f`) into ZK constraints (`ZKExpr f`).

Monadic structure (`ZKBuilder`):

- Allocate new witnesses
- Enforce constraints

Goal: Preserve program semantics through circuit satisfiability.

How Compilation Works

The compilation function `compileExpr` is defined recursively as follows:

```
def compileExpr (t : Term f) (env : Env f) : ZKBuilder f (ZKExpr f) :=
  match t with
  | .var x      => ...
  | .lit n      => ...
  | .bool b     => ...
  | .arith op  t  t => ...
  | .boolB op  t  t => ...
  ...
```

Constraints and Witnesses

- Compilation is done in the ZkBuilder monad
  - State monad keeping track of constraints + witnesses
- In compiling, use `Witnessable.witness` to allocate a new witness
- Constraints are added using `constrain` `constrainEq`, `constrainR1CS`
- Compiler is **total**, all source terms are compiled into circuits.

Example Case: Arithmetic Operation

The `Term.arith` case is handled as follows via combinators:

```
def ArithBinOp.toZKExpr {f}
(op : ArithBinOp) :
ZKExpr f → ZKExpr f → ZKExpr f :=
  match op with
  | .add => ZKExpr.Add
  | .sub => ZKExpr.Sub
  | .mul => ZKExpr.Mul
```

`ArithBinOp.toZKExpr`

```
def liftOpM {f} [Field f]
[JoltField f]
[DecidableEq f] :
    ArithBinOp → ZKExpr f → ZKExpr f →
    ZKBuilder f (ZKExpr f)
  | op, ea, eb => do
      let w ← Witnessable.witness
      constrainEq (op.toZKExpr ea eb) w
      pure w
```

`liftOpM`

```
| Term.arith op t1 t2 => do
  let a ← compileExpr t1 env
  let b ← compileExpr t2 env
  liftOpM op a b
```

`compileExpr`

Example Case: Set Membership

The “inSet” case of the recursion is as follows:

```
| Term.inSet t ts => do
-- 1) compile the inner term
  let x ← compileExpr t env
-- 2) build product P = ∏ (x - c)
  let prod ← ts.foldlM
              (fun acc c => pure (ZKExpr.Mul acc (ZKExpr.Sub x (ZKExpr.Literal c))))
              ((ZKExpr.Literal 1))
-- 3) allocate witnesses
  let b   ← Witnessable.witness      -- Boolean result
  let inv ← Witnessable.witness      -- inverse of prod when prod ≠ 0
-- 4) add constraints
  constrainEq (ZKExpr.Mul b prod) (ZKExpr.Literal 0)          -- b * P = 0
  constrainEq (ZKExpr.Mul prod inv)
             (ZKExpr.Sub (ZKExpr.Literal 1) b)              -- P * inv = 1 - b
  assertIsBool b                                            -- b ∈ {0,1}
-- 5) return Boolean indicator
  return b
```

Outline

Verification: What Is Our Correctness Claim?

Recall: Compiler should *preserve semantics* of zkFlow programs

Claim: If a term *t* is well-scoped in environment *env*, and evaluates to value *v*, then compiling *t* produces a circuit expression and a set of constraints such that:

- The constraints are satisfied by some witness assignment
- The compiled circuit expression evaluates to the same value ‘v‘ under that witness.

In Lean:

```
theorem compileExpr_correct :
  ∀ (t : Term F) (env : Env F) (v : Val F),
    wellScoped t env →
    Eval F t env v →
    ∃ (witness : List F),
      let (compiledExpr, st) := (compileExpr t env).run
    initialZKBuilderState
      constraints_semantics st.constraints witness = true ∧
      semantics_zkexpr compiledExpr witness = Val.toValue v :=
```

Towards Verification - Some Proven Lemmas

`lemma csappend` — Constraint semantics distribute over list append.

`lemma constraints_semantics_perm` — Satisfaction is preserved under permutation of constraints.

`lemma wellScoped_iff_*` — A collection of well-scopedness lemmas: a term is well-scoped iff all of its subterms are well-scoped.

`lemma homomorphism_thrm_arith` — Compilation is compositional: arithmetic structure is preserved under compilation.

`lemma semantics_zkexpr_suffix_irrelevant` - ZK expression evaluation unaffected by appending unused witnesses

`lemma constraints_suffix_irrelevant` - Constraint satisfaction is unaffected by appending unused witnesses

...and many more.

Example Proof:

lemma semantics_zkexpr_suffix_irrelevant

Appending unused witnesses doesn't affect ZK expression evaluation.

```
lemma semantics_zkexpr_suffix_irrelevant {f} [JoltField f] (c : ZKExpr f) (w w' : List f)
  (h : ∀ i, i ∈ witnessIndices c → i < w.length) :
  semantics_zkexpr c w = semantics_zkexpr c (w ++ w') := by
  induction' c with n i e₁ e₂ ih₁ ih₂ e₁ e₂ ih₁ ih₂ e ih e₁ e₂ ih₁ ih₂ e₁ e₂ ih₁ ih₂ table e₁ e₂ ih₁ ih₂
  · case Literal n =>
    simp [semantics_zkexpr, semantics_zkexpr.eval]
  · case WitnessVar i =>
    simp [semantics_zkexpr]; specialize h i
    have lem : i ∈ @witnessIndices f (ZKExpr.WitnessVar i) := by
      simp [witnessIndices]
    specialize h lem
    have lem2 : i < w.length + w'.length := by
      omega
    simp [semantics_zkexpr, semantics_zkexpr.eval, h, lem2]
  case Neg =>
    simp [semantics_zkexpr]; (specialize ih h); simp [semantics_zkexpr, semantics_zkexpr.eval] at *; simp [ih]
  all_goals {
    simp [semantics_zkexpr]; simp [witnessIndices] at h
    have h₁ : ∀ i, i ∈ witnessIndices e₁ → i < w.length := by
      intro i hi; (specialize h i (Or.inl hi)); exact h
    have h₂ : ∀ i, i ∈ witnessIndices e₂ → i < w.length := by
      intro i hi; (specialize h i (Or.inr hi)); exact h
    (specialize ih₁ h₁); specialize ih₂ h₂
    simp [semantics_zkexpr, semantics_zkexpr.eval] at *; simp [ih₁, ih₂]
  }
```

Verification: Structure of Correctness Proof

Theorem Statement:

```
theorem compileExpr_correct :
    (t : Term F) (env : Env F) (v : Val F),
    wellScoped t env →
    Eval F t env v →
      (witness : List F),
      let (compiledExpr, st) := (compileExpr t
    env).run initialZKBuilderState
       constraints_semantics st.constraints
    witness = true
       semantics_zkexpr compiledExpr witness =
    Val.toValue v
```

Proof Progress:

Approach

Induction on Eval hypothesis

Proven:

Base cases: `var`, `lit`, `bool`

Arithmetic: `add`, `sub`, `mul`

Remaining:

Logical: `and`, `or`, `not`

Control: `ifz`, `lett`, `seq`, `assert`

Membership: `inSet`

Outline