A Verified Compiler from a Small LISP to a Stack Machine in Lean 4

Jad Assaf
Youngstown State University
assaf.jad230@gmail.com

26 July 2026

Abstract

This report describes *phi*, a compiler from a six-constructor expression language to a stack-machine instruction list, together with a machine-checked proof in Lean 4 that compilation preserves meaning. The proof covers every constructor of the source language, including let-bindings compiled to stack slots and primitives of arbitrary arity compiled to a left fold. A constant folding pass is proven semantics-preserving and composes with the compiler theorem in two rewrites. The whole development is 700 lines and builds under Lean 4.11.0 with no `sorry`; the axiom check on every headline theorem returns `[propext, Quot.sound]`. The result itself is old, and the report says so up front. What is worth writing down is the set of statements that had to be found before any of the proofs would go through, and a handful of Lean 4 specifics that cost more time than the mathematics did.

1 What this is, and what it is not

Proving that a compiler from arithmetic expressions to a stack machine preserves meaning is one of the oldest results in program verification. McCarthy and Painter did it in 1967 [1]. Nipkow and Klein present essentially the same development as a textbook chapter in Isabelle [2]. At the other end of the scale, CompCert [3] and CakeML [4] are verified compilers for realistic languages, and the distance between those systems and this one is enormous. Nothing in this report is a new theorem.

I am writing it anyway for two reasons. The first is that the proofs are real and independently checkable, and a claim of that kind should come with a document that states precisely what was checked. The second is that the difficulty in a development like this is concentrated in places that the textbook presentations, quite reasonably, compress. Their compiler handles arithmetic and boolean expressions. Adding let-bindings that live on the value stack, and primitives whose arity is not fixed, changes which statements are provable, and finding the provable statement turned out to be the entire problem. Every section below that describes something hard describes a statement I had written down wrongly, not a tactic I could not find.

The scope is deliberately narrow. There is no parser: `Phi/Parser.lean` contains a single stub definition and will stay that way. A hand-rolled unverified frontend would sit on the critical path between what a user writes and what the theorem talks about, which would weaken the claim rather than strengthen it. There are no closures, no recursion, and no heap.

Two disclosures. This was a personal project, neither affiliated with nor supervised by Youngstown State University. And it is the first thing I have written in Lean 4. I mention the second because it should change how the sections below are read: some of what I describe as difficult is difficult only for someone who has not built one of these before, and where I can tell the difference I have tried to say so.

2 The two languages

The source language is an inductive type with six constructors. Atoms are variable references, `binOp` is a fixed two-argument operation, `cond` is the conditional, `letE` binds a name over a body, and `primitive` applies an operator to a list of arguments of any length.

```
inductive LispExpr where
  | atom : String → LispExpr
  | num  : Int → LispExpr
  | binOp : String → LispExpr → LispExpr → LispExpr
  | cond : LispExpr → LispExpr → LispExpr → LispExpr
  | letE : String → LispExpr → LispExpr → LispExpr
  | primitive : String → List LispExpr → LispExpr
```

An earlier version carried a seventh constructor, `list`, which evaluated and compiled to a constant zero in every function. Its case in the correctness proof closed by `rfl`, since both sides reduced to the same placeholder. That is not a proof that lists compile correctly; it is a proof that a stub agrees with itself, and its only real effect was to make the sentence "every constructor is proven" false while looking true. Giving it genuine semantics requires the value stack to become a nested type rather than `List Int`, which propagates into `runInst`, `execute`, `evalPrim`, and every existing proof. I deleted the constructor instead.

Environments are total functions `String → Int`. This is the single most consequential simplification in the development and it is worth being blunt about the cost. An unbound atom evaluates to zero rather than failing. So does an unknown operator, and so does a `primitive` applied to an empty argument list. The compiler theorem therefore says that compilation agrees with a permissive semantics; it does not say that malformed programs are detected. A reader who wants "the compiler rejects bad input" will not find it here, and no amount of restating the theorem will produce it.

Evaluation is defined twice. The fuel-free `eval` recurses structurally and is what every proof is stated against. A fueled `evaluate` also exists, returning zero when fuel runs out. Section 6 relates them.

The target is a seven-instruction stack machine.

```
inductive StackInst where
  | pushInt  : Int → StackInst
  | pushAtom : String → StackInst
  | pushLocal : Nat → StackInst
  | slide : StackInst
  | callOp   : String → StackInst
  | jump : Nat → StackInst
  | jumpIfFalse : Nat → StackInst
```

There is no program counter. `execute` recurses on the instruction list itself, and a relative forward jump is `List.drop`:

```
def execute (insts : List StackInst) (env : Env) (stack : Stack) : Stack :=
  match insts with
  | [] => stack
  | .jump n :: is => execute (is.drop n) env stack
  | .jumpIfFalse n :: is =>
    match stack with
    | v :: rest =>
      if v = 0 then execute (is.drop n) env rest
      else execute is env rest
    | _ => stack
  | i :: is => execute is env (runInst i env stack)
termination_by insts.length
```

Termination is immediate because `drop` never lengthens a list. I had expected jump handling to be the hard part of the development, on the strength of how awkward relative offsets are in an interpreter with mutable state. It took an afternoon. The whole of the jump-offset reasoning reduces to one arithmetic lemma about dropping past a prefix, $\mathrm{drop}\,(|l_1|+1)\,(l_1 +\!\!+ x :: l_2) = l_2$, and that lemma is four lines. Forward-only jumps into a list you are already holding are not a difficult verification problem, whatever they are in a real machine.

3 The statement that does not work

The obvious thing to try to prove is that running the compiled code produces the value of the expression. Stated on an empty stack, that is $\mathrm{execute}(\mathrm{compile}(e), \rho, [\,]) = [\mathrm{eval}(e, \rho)]$, and it is true. It is also useless, because it is not strong enough to be its own induction hypothesis. Compiling `binOp` concatenates the code for the two operands, and by the time the right operand runs, the left operand's value is sitting on the stack. An induction hypothesis about the empty stack says nothing about that situation. The fix is routine: generalise over the incoming stack, and prove

$$\forall e, \rho, \sigma. \quad \mathrm{execute}(\mathrm{compile}(e), \rho, \sigma) = \mathrm{eval}(e, \rho) :: \sigma.$$

The second failure is less routine and took considerably longer to diagnose. Consider the compilation of a conditional:

```
| .cond c t e =>
  let tc := compile ctx t
  let ec := compile ctx e
  compile ctx c ++ [.jumpIfFalse (tc.length + 1)] ++ tc ++ [.jump ec.length] ++ ec
```

To take a single step through `jumpIfFalse` the proof needs to know that the stack is non-empty, which is a consequence of the correctness of the condition's code. But to decompose the concatenation at all, the proof needs to know that running $l_1 +\!\!+ l_2$ is the same as running l_2 on the output of l_1. Call that the append property. Neither is derivable from the other: the append property is used inside the correctness argument, and correctness is used inside the append argument. Attempting to prove either alone leaves a goal that cannot be closed without the other, and the shape of the leftover goal is not obviously a hint that a second property is missing.

The resolution is to prove one conjunction by a single induction.

Theorem 1 (`compile_correct_and_append`)**.** *For every compile-time context* Γ *and every expression* e*,*

$$\forall \rho, \gamma, \sigma.\ \mathrm{Matches}(\Gamma, \sigma, \rho, \gamma) \Rightarrow \mathrm{execute}(\mathrm{compile}(\Gamma, e), \gamma, \sigma) = \mathrm{eval}(e, \rho) :: \sigma,$$
$$\textit{and} \quad \forall l_2, \gamma, \sigma.\ \mathrm{execute}(\mathrm{compile}(\Gamma, e) +\!\!+ l_2, \gamma, \sigma) = \mathrm{execute}(l_2, \gamma, \mathrm{execute}(\mathrm{compile}(\Gamma, e), \gamma, \sigma)).$$

The headline theorem is a projection out of the first component at $\Gamma = [\,]$, where the Matches hypothesis becomes trivial:

```
theorem compiler_correct (e : LispExpr) (env : Env) (s : Stack) :
    execute (compile [] e) env s = eval e env :: s :=
  (compile_correct_and_append [] e).1 env env s (matches_nil env s)
```

Strengthening an induction hypothesis is standard advice, and I had read it many times before it was any use to me. What the advice does not convey is that the strengthening is not a small perturbation of the original statement. The append property is not a variant of correctness with an extra quantifier; it is a different proposition, about instruction lists and stacks, with no mention of `eval` at all. Finding it required abandoning the assumption that the theorem I wanted was close to the theorem I could prove.

3.1 Two Lean details that cost an afternoon each

Neither of these is deep, and both are the kind of thing that is obvious once someone tells you.

The first: `simp only [execute]` does not fire on a concrete non-jump head such as `.callOp op :: is`. The reason is that the catch-all branch `i :: is` in the definition of `execute` overlaps the `.jump` and `.jumpIfFalse` branches above it, so the equation lemma Lean generates for that branch carries side conditions asserting that the head is neither. Those side conditions are not discharged automatically when the head is a specific constructor. The symptom is a `simp` call that reports no progress on a goal that looks like it should reduce by computation. The fix is a small collection of hand-stated reduction lemmas (`execute_callOp`, `execute_pushInt`, `execute_jump`, and one for each branch of `jumpIfFalse`), each proven by `simp [execute, runInst]` where the head is concrete enough for the side condition to close.

The second: in Lean 4, `::` binds at precedence 67 and `++` at 65, so `a :: l ++ l2` parses as `(a :: l) ++ l2`. Compiled instruction lists are built by concatenation and then need to be normalised into the `head :: (tail ++ l2)` shape that the reduction lemmas above expect. `List.append_assoc` and `List.singleton_append` are not sufficient for this; `List.cons_append` and `List.nil_append` are also required, and omitting them leaves goals that differ from the target only in association.

There is a third habit worth recording because it looks like a mistake in the source. `rw` rewrites the first matching instantiation and its syntactic duplicates, not every instantiation. In the append half of Theorem 1 both sides of the goal contain a compiled subterm, and a single `rw [ihe_append, ihe_correct]` reduces only the left. The proofs therefore repeat the same rewrite pair a second time to reduce the right. Reading the source, the repetition looks like an accident.

4 Locals on the stack without a depth counter

Let-bound variables live on the value stack. `pushLocal i` reads the slot i positions down, and `slide` discards the binding sitting underneath a computed result. Compilation therefore needs a compile-time notion of which variable is at which depth, which is what the context Γ in Theorem 1 is for.

The complication is that a local's index is not fixed. Every temporary pushed above it shifts it by one. The most visible instance is the right operand of a binary operation, which runs with the left operand's value already on the stack, so a variable that was at slot 0 is at slot 1 for the duration.

The obvious implementation carries an integer depth alongside the context and adds it to every lookup. I wrote that first. It works, and reasoning about it is unpleasant, because the invariant relating the context, the counter and the stack has three moving parts and every lemma has to re-establish all three.

What is in the development instead is a context of type `List (Option String)`, where an anonymous slot is pushed wherever a temporary appears:

```
abbrev CEnv := List (Option String)

| .binOp op a b => compile ctx a ++ compile (none :: ctx) b ++ [.callOp op]
| .letE x v b => compile ctx v ++ compile (some x :: ctx) b ++ [.slide]
```

A `none` entry carries no name, so no lookup can ever match it, but it occupies an index, so `lookupSlot` returns the shifted depth without any arithmetic being written down anywhere. The counter and the context become the same object. Branches of a conditional compile at the unmodified context, because `jumpIfFalse` pops the condition and the branches therefore run at the original depth; the argument tail of a primitive compiles at `none :: ctx`, because the fold accumulator occupies a slot.

Correctness is stated against a relation tying a context to a runtime stack. Writing Γ for the context, σ for the stack, ρ for the local environment and γ for the global one:

$$\mathrm{Matches}(\Gamma, \sigma, \rho, \gamma) \;\equiv\; \forall x.\;\; \rho(x) = \begin{cases} \sigma[i] & \text{if } \mathrm{lookupSlot}(\Gamma, x) = \text{some } i, \\ \gamma(x) & \text{otherwise.} \end{cases}$$

Two lemmas do essentially all the work. `matches_push_anon` says the relation survives pushing a value under `none :: ctx`, and `matches_push_bind` says it survives pushing a value under `some x :: ctx` provided the environment is updated at x. Both are about fifteen lines and both split on whether `lookupSlot` succeeds.

4.1 An auxiliary definition that looks redundant

The append half of Theorem 1 quantifies over instruction lists and stacks and has no environment in scope at all. The `cond` and `primitive` cases of that half nevertheless need to know that the incoming stack is a cons, and that fact is only available through the correctness half, which does mention an environment. The proof therefore has to produce an environment from nothing.

```
def envOf (ctx : CEnv) (s : Stack) (genv : Env) : Env :=
  fun x =>
    match lookupSlot ctx x with
    | some i => s.getD i 0
    | none => genv x

theorem matches_envOf (ctx : CEnv) (s : Stack) (genv : Env) :
    Matches ctx s (envOf ctx s genv) genv := fun _ => rfl
```

This is Matches read as a definition rather than a proposition, which is why the theorem holds by `rfl` and not by an argument. Reading the source it appears to duplicate the relation for no reason. Deleting it breaks four proofs.

5 Primitives of arbitrary arity

`primitive` holds a `List LispExpr`, and this interacts badly with the induction principle Lean derives automatically. For a constructor containing a nested list, the induction hypothesis has the shape $\forall x \in \mathit{args}.\ \mathrm{motive}(x)$ rather than a hypothesis per argument. A direct `induction e with` case split does not go through, and this is precisely why the language carries a separate two-argument `binOp` constructor at all: it was added to keep the core proof tractable while the variadic case was unresolved, and it has been kept as the simple path.

The usual remedies are a hand-written recursor or threading a `List.Forall` argument through the statement. The development does neither. Instead `compile` and `eval` are each defined in a `mutual` block with list-shaped helpers, and the proof is structured to mirror those definitions exactly:

```
def compilePrimArgs (ctx : CEnv) (op : String) : List LispExpr → List StackInst
  | [] => [.pushInt 0]
  | [e] => compile ctx e
  | e :: es => compile ctx e ++ compileRestWithOps (none :: ctx) op es

def compileRestWithOps (ctx : CEnv) (op : String) : List LispExpr → List StackInst
  | [] => []
  | e :: es => compile ctx e ++ [.callOp op] ++ compileRestWithOps ctx op es
```

Lean accepts the mutual recursion because each helper recurses on a strict sublist while `compile` recurses on a strict subterm. The induction in the proof is then structural over the definitions as written, rather than over the derived recursor, which sidesteps the problem instead

of working around it. The corresponding lemmas (`compilePrimArgs_correct_and_append`, `compileRestWithOps_correct` and `compileRestWithOps_append`) sit inside the same `mutual` block as Theorem 1. The last two are stated on a stack of the form *acc* :: *rest* rather than an arbitrary σ, because the fold accumulator is already present by the time that code runs. Getting that shape right took longer than proving the lemmas once it was right.

Semantics are a genuine left fold, so `(+ 1 2 3 4)` compiles to four pushes interleaved with three `callOp` instructions, and the proof covers it.

6 Fuel, and a bug found by writing a statement down

The fueled evaluator `evaluate` had no theorems about it for most of this project's life. Everything was proven about `eval`, and any write-up that mentioned `evaluate` was overclaiming. The missing statement is that enough fuel gives the same answer:

$$\forall e, \rho, n. \quad n \geq \mathrm{depth}(e) \implies \mathrm{evaluate}(e, \rho, n) = \mathrm{eval}(e, \rho).$$

Writing that down is what exposed the bug. The `primitive` case of `LispExpr.depth` returned `1`. But `evaluate` recurses into a primitive's arguments at $\mathit{fuel} - 1$, so a primitive's depth is one more than the deepest of its arguments, not one. The expression `(+ 1 2 3 4)` claimed depth 1 and needs depth 2. The theorem above was not merely unproven, it was false.

Nothing had ever behaved incorrectly. `depth` was dead code, called by nothing, so no test could have detected the error and no amount of running the compiler would have surfaced it. It became visible at the moment I tried to state a property that mentioned it, which is a smaller and more specific claim about formal methods than the ones usually made for them, and one I now believe on the basis of having experienced it once.

The repair makes `depth` mutual, mirroring the structure of `eval`:

```
mutual
def LispExpr.depth : LispExpr → Nat
  | .atom _ => 1
  | .num _ => 1
  | .binOp _ a b => 1 + max a.depth b.depth
  | .cond c t e => 1 + max c.depth (max t.depth e.depth)
  | .letE _ v b => 1 + max v.depth b.depth
  | .primitive _ args => 1 + LispExpr.depthArgs args

def LispExpr.depthArgs : List LispExpr → Nat
  | [] => 0
  | e :: es => max e.depth (LispExpr.depthArgs es)
end
```

With that in place the bridge is proven mutually with a companion lemma about argument lists. The one point of interest is the termination measure. The main lemma recurses at $\mathit{fuel}-1$; the companion recurses at the *same* fuel on a structurally smaller argument list. Neither ordering is well-founded alone, and the measure is the lexicographic pair (fuel, sizeOf(e)): the first component drops when descending into a subexpression, the second when walking the argument list. Lean accepts it without further help once the pair is written.

```
theorem evaluate_eq_eval (fuel : Nat) (e : LispExpr) (env : Env)
    (h : e.depth ≤ fuel) : evaluate e env fuel = eval e env
```

The corollary I actually wanted is that any two sufficient budgets agree, `evaluate_stable`, which is the precise sense in which fuel is currently vestigial. It stops being vestigial as soon as recursive function application is added, since `depth` will no longer bound evaluation, and that is the reason the parameter is still there.

7 Constant folding

The optimizer folds a `binOp` over two literals into a literal, and drops the dead branch of a `cond` whose condition has folded to a literal. Everything else recurses without folding. There is deliberately no constant propagation into `letE` bodies: that requires substitution and an argument about capture, which is a real piece of work and disproportionate to a pass this size.

Soundness is $\text{eval}(\text{optimize}(e), \rho) = \text{eval}(e, \rho)$, proven mutually with the corresponding statement about argument lists. The `binOp` and `cond` cases both `split` on the match against the already-optimized subterms and then rewrite the induction hypotheses through the equation hypotheses that `split` introduces, which is the only part of the development where the tactic script is doing something less than obvious.

Because both theorems are equations about `eval`, they compose without any glue:

```
theorem compile_optimize_correct (e : LispExpr) (env : Env) (s : Stack) :
    execute (compile [] (optimize e)) env s = eval e env :: s := by
  rw [compiler_correct, optimize_sound]
```

Measured on three expressions: `6*7+8` compiles to five instructions unoptimized and one optimized; a conditional with a literal condition goes from nine to one; and `x+y` over two unbound atoms stays at three, which is the case that would break if the folding were unsound.

8 What is checked

The development is 700 lines of Lean across nine files and builds under Lean 4.11.0 with no `sorry` and no `admit`. Table 1 lists each headline theorem with its location. Every entry has been checked with `#print axioms` and returns `[propext, Quot.sound]`, so no result depends on `sorryAx`, and none depends on `Classical.choice`.

Theorem	Location	Statement
`compiler_correct`	`Correctness:215`	compilation preserves `eval`
`execute_compile_append`	`Correctness:219`	compiled code composes under `++`
`compile_correct_and_append`	`Correctness:18`	both of the above, one induction
`compilePrimArgs_correct_and_append`	`Correctness:131`	variadic case of the above
`optimize_sound`	`OptimizerCorrect:11`	folding preserves `eval`
`compile_optimize_correct`	`OptimizerCorrect:59`	the two composed
`evaluate_eq_eval`	`FuelBridge:7`	enough fuel agrees with `eval`
`evaluate_stable`	`FuelBridge:58`	any two sufficient budgets agree

Table 1: Headline results. All return `[propext, Quot.sound]` under `#print axioms`.

Three limitations are worth restating without softening. Totality means the theorem is conditional on well-formedness that is never checked: a program referencing an unbound variable compiles and runs and the theorem holds of it, because both sides agree that the variable is zero. There is no parser, so the verified pipeline starts at an abstract syntax tree that a user cannot currently produce except by writing Lean. And the source language has no binding construct more interesting than `letE`, which is the point at which this development stops being an exercise and starts being a project: a closure has to capture an environment, environments stop being addressable as stack offsets, and the Matches relation that carries the present proof no longer says anything useful.

References

[1] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science*, volume 19 of Proceedings of Symposia in Applied Mathematics, pages 33–41. American Mathematical Society, 1967.

[2] T. Nipkow and G. Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014.

[3] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[4] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 179–191, 2014.

[5] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction (CADE-28)*, pages 625–635. Springer, 2021.