Verified Arithmetic to Wasm Compiler in Lean4

CADE LUEKER, CU Boulder, USA

For the last three decades Coq has led the charge in the programming languages wing of the formal verification and proof assistant world. Lean has started to gain ground in the mathematics community since its release in 2013, but has not yet made the leap to the world of programming languages. This paper is an attempt to show the utility of using Lean for studying programming languages by defining the semantics for two small programming languages and verifying a compiler from one to the other.

Additional Key Words and Phrases: Lean4, Coq, Wasm, Compiler, Verification

1 Introduction

This project can be broken down into three main parts, each of which serves not only as an academic exploration of programming languages and their verification but also as an attempt to show the utility of the Lean programming language in programming language research. The project defines two languages with compatible semantics:

- Integer Arithmetic, addition, subtraction, multiplication, and division.
- Assign string variables to integer values.
- Retrieve integer values from string variables when executing arithmetic expressions
- Sequencing commands

The first language **Stack** was the more challenging and unique. Stack models webassembly instructions serving as a very reduced version of the language. It holds and manipulates integers on a stack while keeping track of variable names in a total map representing state. Stack is the target language of the compiler portion of this project.

The second language **SIMP** is much less interesting and stands for simplified IMP, a language explored in depth by Pierce in his online textbook "Logical Foundations" [5]. It includes the semantics listed above, and also makes use of a total map to represent state, and is the source language for the compiler.

Each language has its syntax, semantics, and evaluations formally defined with Lean. Evaluation has been shown to be deterministic for each, using Lean's proof assistant capabilities. Finally a compiler was written from SIMP to Stack and parts of it have been proven correct. A correct compiler means that there is **semantic preservation** between the source language and the resulting compiled target language [4].

I have accomplished most of my goals, but could not verify the preservation of all semantic properties that I had wanted. I was able to verify the preservation of arithmetic evaluation but not of equivalent final states for source and target language. I ran out of time and had to spend the last day writing this report. In the initial CompCert paper it is estimated that the creation of a verified compiler for a large subset of C would take about 3 "person years" [4] and for a course project constituting several weeks of work, what I have finished appears to be reasonable progress.

2 Related Works and Inspiration

My work was inspired by the CompCert project [1, 7] and several projects on formalizing Wasm semantics in Coq [2, 6]. I used the work on CompCert to understand what constitutes an observable behavior that a correct compiler must preserve, and the work on Wasm semantics to understand the formalization of a stack based language and how to bridge its syntax to its semantics. When it came to understanding stack based languages Robert Kleffner's masters thesis did a great job expanding on the formalization of stack based languages, specifically how they handle binary operations and state [3]. Lastly the Wasm language creators have a strong PL background and provide well formed documentation for the operations of the language.

Publication date: December 2024.

Throughout this semester, the graduate course "Foundations of Programming Languages" has exclusively used Pierce’s online textbook "Logical Foundations" for instruction and assignments. The detailed description of formalizing semantics for imperative languages was a guiding light during the formalization of SIMP [5].

I initially started the project in Coq, and although I felt that it was still unique enough to warrant a course project, I did not think that it was unique enough to warrant further research. The commonality between all of the projects and textbooks mentioned is the use of Coq. Lean has been expanding in the realm of mathematics but has yet to catch on and in an attempt to see how useful it could be in this realm I decided that despite the added effort of learning to use the language it would be more worthwhile from a research standpoint.

2.1 Using Lean4

As I was learning Lean I turned to the lean-community’s online book, which focuses on the basics of theorem proving with Lean [7]. It is rather straightforward and gives good insights into the language. Coming from a semester of using Coq it was difficult at times to break one way of thinking for another, but I found the transition to be relatively straight forward. One downside of Lean is that it is still considered beta software by its creators, meaning that some things will break, and they did. Luckily it was only the documentation generation tool, and I was able to find a workaround. Because of the fragile nature of versioning in the current Lean ecosystem I used the **nix** package management system with "flakes" to create a reproducible and declarative development environment. This can be found on my github https://github.com/CadeMichael/cl_comp/blob/main/leanTheories/flake.nix.

3 Stack

For the semantics and evaluation sections I will focus on Stack, as it is the interesting part of the project and the semantics of SIMP are similar enough to Pierce’s IMP that they do not need to be detailed but can be found with full documentation on the project’s github https://github.com/CadeMichael/cl_comp/tree/main/leanTheories

The syntax of stack is fairly simple and based heavily on the syntax of integer operations in Wasm, down to the names of operations.

$$
\begin{array}{rcl}
\text{Instr} & ::= & \text{const } Z \\
 & | & \text{binop} \\
 & | & \text{load String} \\
 & | & \text{set String} \\
\\
\text{Binop} & ::= & \text{add} \\
 & | & \text{minus} \\
 & | & \text{mult} \\
 & | & \text{div}
\end{array}
$$

The operational semantics of Stack can be boiled down to these four transition relations, written as inference rules. They define the transition of the "context" of a Stack program after the execution of an instruction. The execution context of Stack is represented by a stack (List Int) and a state (total map from String to Int). This context is written as a tuple in Lean ($stack \times state$).

$$\frac{s = [y :: x :: sx]}{([y :: x :: sx], st) \xrightarrow{\text{binop } op} ((op\ x\ y) :: sx, st)} \qquad \text{(Binary Operation)}$$

$$\frac{s = [x :: sx] \quad v : \text{String}}{([x :: sx], st) \xrightarrow{\text{set } v} ([sx], st[v \mapsto x])} \qquad \text{(Set Operation)}$$

$$\frac{s : \text{stack} \quad st(v) = x}{(s, st) \xrightarrow{\text{load } v} ((x :: s), st)} \qquad \text{(Load Operation)}$$

$$\frac{x : \text{Int}}{(s, st) \xrightarrow{\text{Const } x} ((x :: s), st)} \qquad \text{(Const)}$$

4 Evaluation

For binary operations I utilized a function that takes the name of an operation and performs the corresponding Lean operation on two integers. By being a Lean function this definition is deterministic and doesn't need to be proved deterministic. To define evaluation I used two **inductive** definitions, and proved each to be deterministic.

The first relation defines the execution of one instruction. The second defines the execution of a list of instructions. I found the inductive proof style of Lean to be very clean and mirror functional programming to the extent that it feels more like programming than writing logical proofs. The evaluation relation and part of the lean proofs are given below and can be found on github.

Execution of a single instruction

```
inductive ieval : inst -> (stack x state) -> (stack x state) -> Prop where
  | I_Const: forall (n : Int) (s : stack) (st : state),
    ieval (.Const n) (s, st) ((n :: s), st)
  | I_Binop: forall (op : binop) (x y : Int) (s : stack) (st : state),
    ieval (.Binop op) ((y :: x :: s), st) (((bo_eval op x y) :: s), st)
  | I_Set: forall (v : String) (x : Int) (s : stack) (st : state),
    ieval (.Set v) ((x :: s), st) (s, st[v |-> x])
  | I_Load : forall (v : String) (x : Int) (s : stack) (st : state),
    st v = x -> ieval (.Load v) (s, st) ((x :: s), st)
```

Execution of a sequence of instructions and determinism proof

```
/-! Evaluation relation for a list of instructions -/
inductive seval : List inst -> (stack x state) -> (stack x state) -> Prop where
  | NilI s:
    seval [] s s
  | ConsI i is s s1 s2 st st1 st2:
    ieval i (s, st) (s1, st1) ->
    seval is (s1, st1) (s2, st2) ->
    seval (i :: is) (s, st) (s2, st2)
```

```
/-! Show that executing a sequence of instructions is deterministic -/
theorem seval_determ {i s s1 s2 st st1 st2}
  (hl : seval i (s, st) (s1, st1))
  (hr : seval i (s, st) (s2, st2)):
  s1 = s2 /\ st1 = st2 :=
  by
    cases hl
    case NilI =>
      cases hr
      case NilI =>
        exact (rfl, rfl)
    case ConsI i is s1' st1' hi hs =>
      cases hr
      case ConsI s'' st1'' hi' hs' =>
        have h1 : s1' = s'' /\ st1' = st1'' := by
          apply ieval_determ
          exact hi
          exact hi'
        let (h1l, h1r) := h1
        subst h1r
        subst h1l
        apply seval_determ
        exact hs
        exact hs'
```

I decided to include the determinism proof to illustrate the functional style of Lean proofs and how it is very readable, even for those unfamiliar with Lean. This makes it a great choice for programming language study.

4.1 Lean4 vs Coq

Defining the semantics of Stack was as far as I got with my Coq implementation and just from the proof of determinism it is clear where the languages diverge. The inductive definitions and function definitions are almost identical but the explicit pattern matching proof style of Lean sets it above Coq in my opinion. The Coq proof is much more complex in that you have to be in the proof environment to get an idea of what is being proven and how. Despite the verbosity I will include the Coq proof to highlight this difference.
Determinism of a sequence of instructions

```
Theorem seval_determ:
  forall c s s1 s2, (seval c s s1) -> (seval c s s2) -> s1 = s2.
Proof.
  intros.
  generalize dependent s2.
  induction H.
  - intros.
    inversion H0. (* H0 : seval [] s s2 *)
    reflexivity.
  - intros.
    inversion H1. (* H1 : seval (i :: is) s0 s3 *)
    subst.
    assert (s1 = s5).
      {
        apply ieval_determ with (i := i) (s := s0); assumption.
      }
    subst.
    apply IHseval in H7.
    assumption.
Qed.
```

5 Compilation

Compilation was fairly simple to implement but very difficult to prove. I spent a great deal of effort on the semantics of the Stack language as I felt that it was novel enough to warrant well formed proofs, leaving not as much time as I would have liked to spend on the compiler. The compiler is a relatively small function in lean that recursively unwinds SIMP commands until they are atomic syntactic SIMP structures, which cannot be reduced further, and translates them to Stack syntax, finally weaving them into a list of Stack instructions in a way that preserves the behavioral semantics of SIMP. The full code and executable examples are in the github under the directory LeanTheories.

```
def comp_aexp (a : aexp) : (List inst) :=
  match a with
  | .ANum i        => [.Const i]
  | .AId v         => [.Load v]
  | .APlus a1 a2   => (comp_aexp a1) ++ (comp_aexp a2) ++ [(.Binop .B_Add)]
  | .AMinus a1 a2  => (comp_aexp a1) ++ (comp_aexp a2) ++ [(.Binop .B_Minus)]
  | .AMult a1 a2   => (comp_aexp a1) ++ (comp_aexp a2) ++ [(.Binop .B_Mult)]
  | .ADiv a1 a2    => (comp_aexp a1) ++ (comp_aexp a2) ++ [(.Binop .B_Div)]

def comp_com (c : com) : (List inst) :=
  match c with
  | .CAsgn x a   => (comp_aexp a) ++ [.Set x]
  | .CSeq c1 c2  => (comp_com c1) ++ (comp_com c2)
```

To prove the **bisumulation** of this compilation function we need to choose what behaviors are observable and must be preserved by compilation [4]. I was able to successfully show that the output of arithmetic is preserved and because it is an observable behavior its preservation is crucial to a certified compiler. This proof was very difficult as I could not find many resources on PL using Lean, but I found that the learning and exploration process was easier and made more sense than that of Coq. The full proof of arithmetic preservation is on the github.

Below I have the theorem statements for the preservation of arithmetic and state changes. Showing the bisumulation of state manipulation was the theorem that I was unable to complete on time but I can alert you as to when this is accomplished over the next week.

```
theorem comp_aexp_cert {a st i}: -- preservation of arithmetic
  aeval st a = i ->
  seval (comp_aexp a) (s, st) (i::s, st) :=

theorem comp_state_cert {c s st st1 st2} -- preservation of state changes
  (hl : ceval c st st1)
  (hr : seval (comp_com c) (empty_stack, st) (s, st2)):
  st1 = st2 :=
```

6 Future Work

I believe that the results and future work of this project revolve mainly around Lean. By defining two simple languages' syntax, semantics, and evaluation along with a compiler from one to the other in Lean shows it is ready for Programming Language research. Additionally I believe that research into a certified compiler to a larger subset of Wasm would be a great asset. Especially as Wasm becomes more widely used and relied upon by the web browsers used by billions of people daily. If I had the time allotted to the CompCert project I would hope to have a large subset of a source and target language defined in Lean, such that a language like Python could be compiled to Wasm allowing better, performant software to be created for the web.

References

[1] Sandrine Blazy. 2024. From Mechanized Semantics to Verified Compilation: The Clight Semantics of CompCert. In *Fundamental Approaches to Software Engineering*, Dirk Beyer and Ana Cavalcanti (Eds.). Lecture Notes in Computer Science, Vol. 14573. Springer Nature Switzerland, Cham, 1–21. https://doi.org/10.1007/978-3-031-57259-3_1

[2] Joachim Breitner, Philippa Gardner, Jaehyun Lee, Sam Lindley, Matija Pretnar, Xiaojia Rao, Andreas Rossberg, et al. 2023. Wasm SpecTec: Engineering a Formal Language Standard. *arXiv* (Nov. 2023). http://arxiv.org/abs/2311.07223

[3] Robert Kleffner. 2017. *A Foundation for Typed Concatenative Languages.* Master's thesis. Northeastern University, Boston, Massachusetts. https://www2.ccs.neu.edu/racket/pubs/dissertation-kleffner.pdf

[4] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[5] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Brent Yorgey, et al. 2024. *Logical Foundations* (current ed.). University of Pennsylvania. https://softwarefoundations.cis.upenn.edu/lf-current/ Online textbook.

[6] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 1096–1120. https://doi.org/10.1145/3591265

[7] The Lean Community. 2024. *Theorem Proving in Lean 4.* lean-lang.org. https://lean-lang.org/theorem_proving_in_lean4/ Online book.