Verified Arithmetic to Wasm Compiler in Lean4

Using Lean for Programming Languages research

- 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.
- Project Git Repo

Syntax and Semantics

- we will use BNF to show the syntax of each language.
  - in Lean these are represented as inductive types
- we will use transition relations to show the big step operational semantics of each language
  - in Lean these are represented as inductive relations

Simp

- SIMP := Simpler Imp
  - SIMP is a boiled down imp with arithmetic, assignment, and sequence.
  - it is used to represent an imperative language being compiled to Wasm.
  - future work would see SIMP be expanded to the point where it models CLite, Python etc…

Syntax

- Using inductive types to define arithmetic and commands for SIMP

```
/- Arithmetic -/
inductive aexp : Type where
  | ANum   (i : Int)
  | AId    (x : String)
  | APlus  (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult  (a1 a2 : aexp)
  | ADiv   (a1 a2 : aexp)

/- Commands -/
inductive com : Type where
  | CAsgn (x : String) (a : aexp)
  | CSeq  (c1 c2 : com)
```

Semantics

- using a function to define evaluation for arithmetic and an inductive relation for evaluating commands

```
def aeval (st : state) (a : aexp) : Int := /- define arithmetic evaluation with a function -/
  match a with
  | .ANum i        ⇒ i
  | .AId x         ⇒ st x
  | .APlus a1 a2   ⇒ (aeval st a1) + (aeval st a2)
  | .AMinus a1 a2  ⇒ (aeval st a1) - (aeval st a2)
  | .AMult a1 a2   ⇒ (aeval st a1) * (aeval st a2)
  | .ADiv a1 a2    ⇒ (aeval st a1) / (aeval st a2)

inductive ceval : com → state → state → Prop where /- define SIMP evaluation with an inductive relation -/
  | C_Asgn : ∀ st a i x,
    aeval st a = i → ceval (.CAsgn x a) st (st[x !→ i])
  | C_Seq : ∀ c1 c2 st st' st'',
    ceval c1 st st' → ceval c2 st' st'' →
    ceval (.CSeq c1 c2) st st''

theorem ceval_determ {c st st1 st2} /- determinism of SIMP evaluation, full proof in repo -/
  (hl : ceval c st st1)
  (hr : ceval c st st2):
  st1 = st2 :=
```

Stack

- SIMP is very generic and most PL courses cover the formulation of an IMP like language.
- Stack is much more unique, and represents a small subset of WASM.
  - Wasm is a stack based language used for high performance tasks in browsers.
  - the evaluation context of Stack consists of a **stack** of values and a **total map** representing state.
  - the Stack language has assignment, constant integers, declaring variables, and integer arithmetic.
- sample Wasm code

```
(module
    (func (export "_start") (result i32)
    ;; load 80 onto the stack
    i32.const 80
    ;; load 10 onto the stack
    i32.const 10
    ;; divide
    i32.div_u
    return
    ) ;; outputs 80 / 10 ⟹ 8
)
```

Syntax

- we define *inst* for an instruction and *binop* for a binary operation on integers.

```
/-! Define all binary operations -/
inductive binop : Type where
  | B_Add
  | B_Minus
  | B_Mult
  | B_Div

/-! Define all instructions -/
inductive inst : Type where
  | Const (i: Int)
  | Binop (op : binop)
  | Set   (v : String)
  | Load  (v : String)
```

Semantics

- we utilize functions for the evaluation of binary operations and inductive relations to show how Stack executes a single instruction and a list of instructions.

```
def bo_eval (op : binop) (x y : Int) : Int :=
  match op with
  | .B_Add    ⇒ x + y
  | .B_Minus  ⇒ x - y
  | .B_Mult   ⇒ x * y
  | .B_Div    ⇒ x / y

inductive ieval : inst → (stack × state) → (stack × state) → Prop where /- proven deterministic -/
  | I_Const: ∀ (n : Int) (s : stack) (st : state),
    ieval (.Const n) (s, st) ((n :: s), st)
  | I_Binop: ∀ (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: ∀ (v : String) (x : Int) (s : stack) (st : state),
    ieval (.Set v) ((x :: s), st) (s, st[v !→ x])
  | I_Load : ∀ (v : String) (x : Int) (s : stack) (st : state),
    st v = x → ieval (.Load v) (s, st) ((x :: s), st)

inductive seval : List inst → (stack × state) → (stack × state) → Prop where /- proven deterministic -/
  | 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)
```

Coq vs Lean

- for the most part the proofs of determinism have been omitted for brevity but I will highlight one to show how proofs in Lean differ from those in Coq.
  - proofs in Lean mirror pattern matching in functional programming languages, making them more clear for readers, and easier to reason about for those writing proofs.
  - in Coq proofs, unless they are fully commented, seem opaque and nebulous unless you are in a proof state seeing the hypotheses being manipulated
  - This was the most intriguing aspect of Lean that I found during my self taught crash course to prepare for this project.
    - it is also what makes Lean appear to be extremely underutilized in the world of PL research
  - I will include the determinism proof of `seval` to highlight this functional style

Lean proof of Stack's determinism

```
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         /- explicit pattern matching against inductive cases -/
    case NilI ⇒     /- subproof for 'seval.NilI' -/
      cases hr
      case NilI ⇒
        exact ⟨rfl, rfl⟩
    case ConsI i is s1' st1' hi hs ⇒ /- subproof for 'seval.ConsI' -/
      cases hr
      case ConsI s'' st1'' hi' hs' ⇒
        have h1 : s1' = s'' ∧ st1' = st1'' := by /- creating new hypothesis -/
          apply ieval_determ
          exact hi
          exact hi'
        let ⟨h1l, h1r⟩ := h1 /- splitting conjuctive hypothesis -/
        subst h1r
        subst h1l
        apply seval_determ
        exact hs
        exact hs'
```

Compilation

- in order for a compiler to be "verified" it must verify the preservation of certain semantic properties
- for instance arithmetic in the source language (language being compiled) must behave the same as arithmetic in the target language.
  - for source language S, target language T, and behavior B
    - $S \Downarrow B \Rightarrow T \Downarrow B$
- for these behaviors we assume the source program is *correct* meaning it can be executed and will not result in an error.
- for this project I was able to write a small compiler and verify the preservation of arithmetic behaviors but could not go further with other behaviors due to limited time.

Compiler function

- the compiler is written as a function in Lean

```
def comp_aexp (a : aexp) : (List inst) := /- compile SIMP aexp's to Stack instructions -/
  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) := /- compile SIMP commands to Stack instructions -/
  match c with
  | .CAsgn x a   ⇒ (comp_aexp a) ++ [.Set x]
  | .CSeq c1 c2  ⇒ (comp_com c1) ++ (comp_com c2)
```

Compiler Proofs of behavioral preservation

- preserving the behaviors of arithmetic expressions from SIMP to Stack
  - `aeval` of `a` produces the same `i` as `seval` of `comp_aexp a`

```
theorem comp_aexp_cert {a st i}:
  aeval st a = i → seval (comp_aexp a) (s, st) (i::s, st) := by
    induction a generalizing i s with
    | ANum i' ⇒
      intro h
      rw [aeval] at h
      rw [comp_aexp]
      rw [h]
      apply seval.ConsI
      apply ieval.I_Const
      apply seval.NilI
    | AId x ⇒
      intro h
      rw [aeval] at h
      rw[comp_aexp]
      apply seval.ConsI
      apply ieval.I_Load
      rw [h]
      apply seval.NilI
    | APlus a1 a2 ha1 ha2 | AMinus a1 a2 ha1 ha2 | AMult a1 a2 ha1 ha2 | ADiv a1 a2 ha1 ha2 ⇒ /- omitted -/
```

Final Thoughts

- the behavioral that I could not prove was that the states remained the same after execution.
  - part of this was due to the complexity of the hypotheses and evaluation of a stack language.
- with more time I would hope to learn Lean wherein I could
  - finish the state equivalence proof
  - expand the features of SIMP and Stack to the point where SIMP gets closer to CLite and Stack to Wasm
  - write a parser that takes in a source program so it doesn't need to be defined as an AST in Lean.
  - write a function to output a Wasm file instead of a list of Stack instructions
- I feel that I have shown the utility of Lean for PL research and how it is more than capable of formalizing languages, writing functions (compilers), and formalizing proofs about the langauges it defines.
- Coq is still the leader of PL research but I hope to show that with some enthusiastic Lean users, it might not remain so.

References and Inspiration

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

References cont.

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