Simple formally verified compiler in Lean

Let's talk about Emacs

Translator

Assumptions:

- Arithmetic is correctly parsed
- Only addition
- Only integers

Data types

Calculator:

```
data Instruction = Loadi Int
                 | Add
```

Algebraic expression:

```
data AExp = N Int
          | Plus Aexp Aexp
```

Example:

```
-- 4 + 5
[ Loadi 4, Loadi 5, Add]
```

Example:

```
-- Note: :: is cons and : is hastype
translate : Aexp -> [Instruction]
| (N i) := [Loadi i]
| (Plus a b) := (translate a) ++ (translate b) ++ [Add]
```

Example:

```
translate : Aexp -> [Instruction]
| (N i) := [Loadi i]
| (Plus a b) := (translate a) ++ (translate b) ++ [Add]
```

```
translate (Plus (N 4) (N 3))
```

Example:

```
translate : Aexp -> [Instruction]
| (N i) := [Loadi i]
| (Plus a b) := (translate a) ++ (translate b) ++ [Add]
```

```
translate (Plus (N 4) (N 3)) ==
(translate (N 4)) ++ (translate (N 3)) ++ [Add] ==
```

Example:

```
translate : Aexp -> [Instruction]
| (N i) := [Loadi i]
| (Plus a b) := (translate a) ++ (translate b) ++ [Add]
```

translate (Plus (N 4) (N 3)) ==
(translate (N 4)) ++ (translate (N 3)) ++ [Add] ==
[Loadi 4] ++ [Loadi 3] ++ [Add] ==

Example:

```
translate : Aexp -> [Instruction]
| (N i) := [Loadi i]
| (Plus a b) := (translate a) ++ (translate b) ++ [Add]
```

```
translate (Plus (N 4) (N 3)) ==
(translate (N 4)) ++ (translate (N 3)) ++ [Add] ==
(Loadi 4) ++ (Loadi 3) ++ [Add] ==
[Loadi 4, Loadi 3, Add]
```

How do we know the translation is correct?

How do we know the translation is correct?

Why should I care?

Write a couple of tests and move on, it's just a calculator that nobody uses anymore

How do we know the translation is correct?

Well, if we add these things to the calculator:

- Variables because those are nice to have when calculating
- Conditional branch instruction

Simple formally verified compiler in Lean

Simple language:

- integers
- addition
- variables
- if statements
- while loops
- superset of the calculator

Back to the question

How do we know the translation is correct?

We could write a proof

The value of the arithmetic expression should be put on the top of the stack after the translated version is run on the calculator

`calculate stk (translate e) == (eval e) :: stk`

We could write a proof

`calculate stk (translate e) == (eval e) :: stk`

Example

```
[Loadi 4, Loadi 3, Add]
```

```
4 :: stk
3 :: 4 :: stk
7 :: stk
```

Proof by induction

Base case: The expression is a number

```
calculate stk (translate (N n))
  ==
calculate stk (Loadi n)
  ==
  n :: stk
```

Induction case: The expression is an addition

Induction hypotheses:

```
calculate stk (translate a) == eval a :: stk
calculate stk (translate b) == eval b :: stk
```

Proof:

```
calculate stk (translate (Plus a b))
  ==
calculate stk (translate a ++ translate b ++ [Add])
  ==
calculate (eval a :: stk) (translate b ++ [Add])
  ==
calculate (eval b :: eval a :: stk) [Add]
  ==
 (eval b + eval a) :: stk
```

Ok, but how do we know that the translation program does what we think it does?

We can obviously never be 100% sure, for all we know we hallucinate everything.

How can we be even more sure?

Let's take a step back

What techniques do we know for proving theorems that we think should be correct? What proof techniques do we know?

Proof Techniques

- Natural deduction

Natural Deduction

Natural deduction isn't enough

Propositions as Types

I will rush through this for the interest of time, but if you are interested see Wadler's paper *Propositions as Types*

- propositions = types
- proofs = programs

```
foo : (a -> c) -> (b -> c) -> (Either a b) -> c
foo f g ab =
  case ab of
    Left  a -> f a
    Right b -> g b
```

Proofs about programs in the same language

The interactive theorem prover Lean

- A dependently typed functional programming language
- A language for mathematics

Types

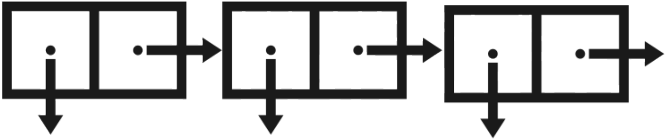
- Proofs about programs in the same language
- The language can check that the proofs are correct
- If program changes the proof is invalid

Types

```
data Nat = Z
         | Suc Nat
```

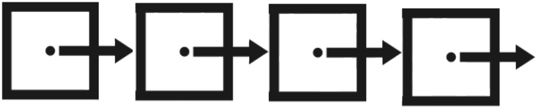
Types

```
data List a = Nil
            | Cons a List
```



Types

```
data Nat = Z
         | Suc Nat
```



```
add : Nat -> Nat -> Nat
add a Z = a
add a (Suc b) = add (Suc a) b
```

Semantics of the stack machine

$$\frac{}{P \vdash (i, s, stk) \Rightarrow (i+1, s, n :: stk)} \; P[i] = \text{loadi } n$$

$$\frac{}{P \vdash (i, s, a :: b :: stk) \Rightarrow (i+1, s, (a+b) :: stk)} \; P[i] = \text{add}$$

Figure: Small-step semantics for one instruction in the stack machine

Semantics of the stack machine in Lean

```
inductive iexec :  instr -> config -> config -> Prop
| loadi (i : ℤ) (s : state) (stk : list ℤ)
        (n : ℤ):
  iexec (instr.loadi n) (i    , s,       stk)
                        (i + 1, s, n :: stk)

| add   (i : ℤ) (s : state) (stk : list ℤ)
        (a b : ℤ) :
  iexec instr.add (i    , s, a :: b  :: stk)
                  (i + 1, s, (a + b) :: stk)
```

Note: This type can't be constructed in normal Haskell

The proof in Lean

The base case from before but in Lean:

```
show (acomp (aexp.N n)) ⊢
      (0, s, stk) ⇒*
      (1, s,
       aval (aexp.N n) s :: stk),
exact star.single (exec1.exec1 (by simp)
                               (iexec.loadi _)),
```

Simple formally verified compiler in Lean

- semantics, compiler and proof written in Lean
- Proof of correctness for terminating programs

The end

Resources

- Propositions as Types[1]
- CompCert: a formally verified optimizing C compiler[2]
- Hitchhiker's Guide to Logical Verification[3]
- Concrete Semantics with Isabelle/HOL[4]
- Lean[5]

[1] http://www.cs.bc.edu/~muller/teaching/lc/WadlerPropositionsAsTypes.pdf

[2] https://compcert.org/

[3] https://github.com/blanchette/logical_verification_2020/raw/master/hitchhikers_guide

[4] http://concrete-semantics.org/

[5] https://leanprover-community.github.io/