UPPSALA
UNIVERSITET

21 097
Examensarbete 15 hp
November 2021

Simple formally verified compiler in Lean

Leo Okawa Ericson

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Simple formally verified compiler in Lean

Leo Okawa Ericson

Computer checked proofs that a compiler is correct are important for increasing the confidence in programs. This report presents a simple compiler and a proof that the compiler is correct for terminating evaluations using the interactive theorem prover Lean, based on Concrete Semantics: with Isabelle/HOL. The compiler compiles a simple imperative language defined by a big-step semantics, to a stack machine defined by a small-step semantics. The proof is done by induction on the big-step semantics. Because the compiler, semantics and proofs are all written in and check by Lean, we can have great confidence that the proof, and by extension, the compiler is correct. The aim is that the proof is understandable for people new to program (and especially compiler) verification, and that it can serve as good entry point. In particular the correctness proof of arithmetic expressions is explained in detail.

Handledare: Jasmin Blanchette
Ämnesgranskare: Lars-Henrik Eriksson
Examinator: Johannes Borgström
21 097
Tryckt av: Reprocentralen ITC

Contents

1 Introduction **1**
1.1 Problem description 1

2 Background **1**
2.1 Lean 2
2.2 Curry-Howard Correspondence 2
2.3 Dependent types 3
2.4 Tactics 5

3 Method **6**
3.1 Source language semantics 6
3.2 Source language semantics formalized 8
3.3 Target machine language semantics 10
3.4 Formalized target language semantics 12
3.5 Compilation 13
3.6 Correctness statement 15
3.7 Proofs 15
3.7.1 Appending programs 15
3.7.2 Correctness of arithmetic expressions 16

4 Result **20**

5 Discussion **20**
5.1 $\Rightarrow*$ or $\Rightarrow k$ 20
5.2 Proved correct only in the forward direction 21
5.3 Difference between Lean and Isabelle 22
5.4 Factors not accounted for 23

6 Related Work **23**

7 Conclusion and future work **24**

Glossary **25**

Symbols **25**

8 Appendix **26**

1 Introduction

1.1 Problem description

As Kästner et al. note in their aptly named paper "Closing the gap – The formally verified optimizing compiler CompCert" [4], there is a gap in the formal verification of software in security critical contexts. There are established methods for verifying software source code, and processors are increasingly more often partly formally specified and verified. The layer between the source code and hardware, the compiler, is often a complex piece of software and is therefore prone to have bugs. Miscompilation errors, when the compiler silently generates incorrect code, can break properties of a program that are proven by analyzing the source code. Kästner et al. have filled this gap by implementing CompCert, a formally verified optimizing compiler for C using the interactive theorem prover and programming language Coq.[1]

CompCert has required several years to develop for a team of developers resulting in many lines of code, and is therefore not the best point of entry for an aspiring user of interactive theorem provers. A simpler example with a simpler source and target language would therefore be useful. *Concrete Semantics: with Isabelle/HOL* [10] provides such an example. They present a compiler for a simple imperative language called `while` that compiles to a simple lower level stack machine, and a computer checked proof that the compiler is correct written in the proof assistant Isabelle/HOL. I present a translation of `while`'s semantics to another interactive theorem prover Lean from *The Hitchhiker's Guide to Formal Verification* [2] in section 3.2, and my translation of the stack machine's semantics in section 3.4. I then define what "correctness" means for this compiler in section 3.6 and have proved that the compiler is correct for terminating programs, in section 3.7 I explain how the proof works.

A compiler f is correct if for all programs p, p and $f(p)$ both yield the same result when run in the same configuration. Both `while` and the stack machine have the same state type, a function from variable names to integers, so defining what "same result" means is easy: both p and $f(p)$ should terminate with the same state, or both should not terminate at all. The approach taken in this report to prove this correctness property is to formalize a semantics for both languages in Lean, and then prove that the compiler preserves the semantics of any given program.

2 Background

Previous experience of operational semantics is assumed, see *Semantics with Applications* [9] or other introductory textbooks for semantics of programming languages. No prior knowledge of Lean or other interactive theorem provers is assumed, but it is beyond the scope of this report to teach Lean. Readers should be able to understand the idea behind the Lean proofs if they are familiar with an ML-like language like Haskell, but for actually learning Lean *Theorem Proving in Lean* [1] or *The Hitchhiker's Guide to Formal*

[1] `https://coq.inria.fr/`

Verification [2] is recommended.

2.1 Lean

Lean is an interactive theorem prover (often called proof assistant), a computer tool for proving mathematical statements. In contrast with automatic theorem provers which try to prove or disprove a given mathematical statement automatically, an interactive theorem prover can interact with the user. In Lean proofs are written by a human in a combination of a functional programming language and an imperative tactic language (see section 2.4). Feedback is continually given to the human by showing a continually updating view of the state of the proof (the available propositions and the next formula to prove), and perhaps most importantly, Lean checks that all the proofs are correct.

2.2 Curry-Howard Correspondence

There is a deep correspondence between logic using natural deduction and statically typed functional programming, often called the Curry-Howard Correspondence, or *Propositions as Types*. This is usually explained using simply typed lambda calculus, but for brevity's sake I will I will only show this correspondence for logical implication and conjunction informally. Curious readers are encouraged to read further in Wadler's excellent paper [11] for more on the concept and its history.

Let ψ and ϕ range over propositions. $[\psi]$ signifies that the proposition ψ is discharged. The introduction and elimination rules can then be defined using natural deduction.

$$\frac{\begin{array}{c}[\psi]^1\\ \vdots\\ \phi\end{array}}{\psi \to \phi}\,{\to}\mathrm{I}^1 \qquad \frac{\psi \to \phi \qquad \psi}{\phi}\,{\to}\mathrm{E}$$

$$\frac{\psi \qquad \phi}{\rho \wedge \phi}\,{\wedge}\mathrm{I} \qquad \frac{\psi \wedge \phi}{\rho}\,{\wedge}\mathrm{E}_1 \qquad \frac{\psi \wedge \phi}{\phi}\,{\wedge}\mathrm{E}_2$$

Lambda expressions and tuples can be defined in a very similar way. Let a and b range over types and x, y, z range over terms. The first two rules describe the creation and application of a lambda expression respectively. The last three rules describe tuple construction and deconstruction respectively. These rules look very similar to the logic rules, and if you squint your eyes and ignore the parts before the colons they look almost identical.

$$\frac{\begin{array}{c}[x:a]^1\\ \vdots\\ y:b\end{array}}{(\lambda(x:a).y):a\to b}\to\mathrm{I}^1 \qquad \frac{(\lambda x.y):a\to b \qquad z:a}{((\lambda x.y)\ z):b}\to\mathrm{E}$$

$$\frac{x:a \qquad y:b}{(x,\ y):(a\times b)}\times\mathrm{I} \qquad \frac{(x,\ y):(a\times b)}{x:a}\times\mathrm{E}_1 \qquad \frac{(x,\ y):(a\times b)}{y:b}\times\mathrm{E}_2$$

$\lambda(x:a).y$ can be thought of as an anonymous function that takes an argument x of type a and has the body y. Function application is done by juxtaposition like in Haskell. $\to$I says that $\lambda x.y$ can be constructed, if y (with type b) can be constructed under the assumption that x is a free variable with type a. The resulting expression then has type $a \to b$. $\to$E says that you can apply the value $z : a$ to the function $((\lambda x.y)\ z)$. The result is of type b.

Wadler describes Howard's observations as follows:

> - Conjunction A & B corresponds to Cartesian product $A \times B$, that is, a record with two fields, also known as a pair. A proof of the proposition A & B consists of a proof of A and a proof of B. Similarly, a value of type $A \times B$ consists of a value of type A and a value of type B.
>
> [...]
>
> - Implication $A \supset B$ corresponds to function space $A \to B$. A proof of the proposition $A \supset B$ consists of a procedure that given a proof of A yields a proof of B . Similarly, a value of type $A \to B$ consists of a function that when applied to a value of type A returns a value of type B. [11, p. 3]

It is important to note that functions have to be *total*, i.e. be defined for all arguments, for this correspondence to be correct. A non-terminating function (e.g. $f(x) = f(x)$) is therefore not allowed. Functions in Lean are allowed to be recursive, but they have to be proven to be terminating. Lean can automatically prove termination if the arguments to the recursive call is strictly smaller.

2.3 Dependent types

There is a similar correspondence between different logic systems and type systems, of interest for this report is dependent types, the underlying theory that Lean uses, which corresponds to first order logic with the connectives $\forall$ and $\exists$ [11]. This means that types can depend not only on other types but also on terms (see the vector example below).

One consequence is that proofs and programs can be written in the same language, which is especially useful for verifying software. It does not mean however, that proofs and programs are treated like they are the same by Lean. Lean is *proof irrelevant*, which

means that all proofs for the same property are treated as equal [8, p. 24]. Intuitively this makes sense, it does not matter how a proof is constructed as long as the proof is correct.

The same treatment would not make sense for types and programs, two programs of the same type are not necessarily equal. For example the numbers 4 and 5 are not equal even though they both have the same type. Another example: addition and subtraction of integers are not equal despite both functions having the same type: $\mathbb{Z} \to \mathbb{Z}$.

Lean has inductive types like many statically typed functional languages like Haskell. Below is a definition of a linked list. `T` is any type. The first constructor `nil` takes no arguments and gives back a term of type `list T`. The second constructor takes an argument `hd` of type `T` and another argument `tl` of type `list T`, and gives back a term of type `list T`. `T` is a fixed argument (as given by the `list (T : Type)`) on the first line, which means that the occurrences of `list` are implicitly given `T` as arguments inside the definition. The last `Type` on the first line indicates that `list` constructs a type as opposed to a proposition (further down I give an example of that).

```
inductive list (T : Type) : Type
| nil : list
| cons (hd : T) (tl : list) : list
```

In contrast to Haskell, Lean allows for naming constructor arguments. Here is an equivalent version that looks more like Haskell:

```
inductive list (T : Type) : Type
| nil : list
| cons  : T -> list -> list
```

Here is the equivalent Haskell data type:

```
data List t = Nil
            | Cons t (List t)
```

Lean supports a more general way to create inductive types called inductive families. [1, p.112] They work like normal inductive types but instead of constructing a type they construct a function that has a sequence of argument types and returns a type. One example is the vector type defined below which is a linked list but with its length attached. Notice that the return type of the constructors depends on a *value* (`zero` and `succ n` respectively) and not a type, something that is possible only with dependent type theory. The type of a vector has information about its length. `vec.cons 1 vec.nil` and `vec.nil` have different types since their lengths are different.

```
inductive vector (α : Type) : ℕ → Type
| nil  {}                                : vector zero
| cons {n : ℕ} (a : α) (v : vector n) : vector (succ n)
-- {n : ℕ} is an implicit argument. It suffices to write
-- vec.cons 1 (vec.nil) and Lean will infer the first argument
-- n from the third argument v.
```

An inductive predicate is way to define a function on the form `... -> Prop` by using an inductive family, where `...` is a sequence of argument types [2, p.83], and `Prop` is

the type of propositions. The difference between `Prop` and `Type` is roughly that `Prop` is used for statements that can be proved (or disproved) and `Type` is for data structures in programs. There is more nuance than this, curious readers are encouraged to read *Theorem Proving in Lean* [1].

As an example, consider the set of even natural numbers which can be defined as the set of all numbers that are even. It can be defined in Lean like this:

```
inductive even : ℕ → Prop
| zero                                  : even 0
| add_two (k : ℕ) ( heven : even k ) : even (k + 2)
```

To prove that a given number `n` is even, you have to construct the element `even n` using the provided constructors. Or in other words, a proof that `n` is a member of the even numbers is a construction of `even n`. Here is a proof that 2 is an even number using this inductive predicate:

```
lemma even_2 : even 2 :=
even.add_two 0 even.zero
```

2.4 Tactics

Proofs in dependent type theory are essentially natural deduction trees in the shape of a program in a programming language. Most (if not all) mathematicians do not write in the style of natural deduction for any moderately complex proof because it is verbose and overly explicit. To alleviate this Lean provides a tactic system that automates the generation of proof terms (sub-trees in natural deduction, or expressions in a functional programming language). Tactics are meta-programs written in Lean that has access to internal APIs such as the current proof state (i.e. what lemmas and hypotheses that are available) and generates new proof terms. These terms are then checked by Lean's kernel to ensure that the proof is still valid [3, 8].

Tactic commands can be used to construct a proof by putting a `begin ... end` block wherever Lean expects a proof term. Below is a list of relevant tactics used in this report. A more complete list of tactics exists on Mathlib's website.[2]

- `have` is used to prove sub-proofs.

  ```
  lemma prop_comp (a b c : Prop)
                  (ha : a)
                  (hab : a → b)
                  (hbc : b → c) : c :=
  begin
    have hb : b := hab ha,
    have hc : c := hbc hb,
    exact hc,
  end
  ```

[2] `https://leanprover-community.github.io/mathlib_docs/tactics.html`

- `simp` simplifies the target formula using known simplification rules. `simp at <hypothesis>` applies the simplification on `<hypothesis>`.
- `abel at <hypothesis>` normalizes expressions in the language of additive, commutative monoids and groups in `<hypothesis>`. In this report it is used to normalize additions of integers.

3 Method

3.1 Source language semantics

The source language is a simple imperative language with simple boolean and arithmetic expressions; and if and while statements. This should be familiar to readers of *Concrete Semantics* or *Semantics with Applications* as the same languages is described in those books. See fig. 1 for the semantics. From now on $\longrightarrow$ denotes the big-step semantics relation for the source language. s is the program state, defined as a function from strings (variable names) to integers. $s\{x \mapsto aval\ a\ s\}$ is (in Haskell-like syntax) short-hand for:

```
\y -> if y == x
then aval a s
else s x
```

$$
\begin{aligned}
aval\ [\![i]\!]\ s &= i \\
aval\ [\![x]\!]\ s &= s\ x \\
aval\ [\![a_1 + a_2]\!]\ s &= aval [\![a_1]\!]\ s + aval\ [\![a_2]\!]\ s
\end{aligned}
$$

$$
\begin{aligned}
bval\ [\![b]\!]\ s &= b \\
bval\ [\![\neg b]\!]\ s &= \neg bval\ [\![b]\!]\ s \\
bval\ [\![b_1 \wedge b_2]\!]\ s &= bval\ [\![b_1]\!]\ s \wedge bval\ [\![b_2]\!]\ s \\
bval\ [\![b_1 < b_2]\!]\ s &= bval\ [\![b_1]\!]\ s < bval\ [\![b_2]\!]\ s
\end{aligned}
$$

$$\frac{}{(\texttt{skip}, s) \longrightarrow s}\ \text{skip}$$

$$\frac{}{(x := a, s) \longrightarrow s\{x \mapsto aval\ a\ s\}}\ \text{assign}$$

$$\frac{(S, s) \longrightarrow t \quad (T, t) \longrightarrow u}{(S; T, s) \longrightarrow u}\ \text{seq}$$

$$\frac{bval\ b\ s \quad (c_1, s) \mapsto t}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \longrightarrow t)}\ \text{if-true}$$

$$\frac{\neg bval\ b\ s \quad (c_2, s) \mapsto t}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \longrightarrow t)}\ \text{if-false}$$

$$\frac{\neg bval\ b\ s}{(\texttt{while}\ b\ \texttt{do}\ c \longrightarrow s)}\ \text{while-false}$$

$$\frac{bval\ b\ s \quad (c, s_1) \longrightarrow s_2 \quad (\texttt{while}\ b\ c, s_2) \longrightarrow s_3}{(\texttt{while}\ b\ \texttt{do}\ c \longrightarrow s_3)}\ \text{while-true}$$

Figure 1: Big-step semantics.
Premises are written above the horizontal line, conclusions are below the line. Each rule's name is written to the right. i ranges over integers, x ranges over variable names, s is the as a function from variable names to integers, a ranges over arithmetic expressions, b ranges over boolean expressions and c ranges over statements.

3.2 Source language semantics formalized

The `while` language has very simple arithmetic expressions. An arithmetic expression is either an natural number literal (`N`), a variable (`V`) or an addition of two arithmetic expressions (`Plus`).

```
inductive aexp : Type
| N    : int -> aexp
| V    : string -> aexp
| Plus : aexp -> aexp -> aexp
```

Boolean expressions feature boolean literals (`Bc`), negation (`Not`), conjugation (`And`) and a comparison of arithmetic expressions (`Less`).

```
inductive bexp : Type
| Bc : bool -> bexp
| Not : bexp -> bexp
| And : bexp -> bexp -> bexp
| Less : aexp -> aexp -> bexp
```

The top-level syntactic element is the statement. Executing a statement either does nothing (`skip`), assigns a value to a variable (`assign`), executes two statements in sequence (`seq`), branches (`ite`) or loops (`while`). `seq S T` is sometimes abbreviated as `S ;; T`.

```
inductive stmt : Type
| skip   : stmt
| assign : string → aexp → stmt
| seq    : stmt → stmt → stmt
| ite    : bexp → stmt → stmt → stmt
| while  : bexp → stmt → stmt
```

The semantics of the arithmetic and boolean expressions are fairly self-explanatory. They can be implemented as ordinary recursive functions with pattern-matching because they always terminate.

```
def aval : aexp -> LoVe.state -> int
| (aexp.N i) _ := i
| (aexp.V x) s := s x
| (aexp.Plus a b) s := (aval a s) + (aval b s)

def bval : bexp -> LoVe.state -> bool
| (bexp.Bc b) _ := b
| (bexp.Not x) s := not (bval x s)
| (bexp.And a b) s := (bval a s) ∧ (bval b s)
| (bexp.Less a b) s := (aval a s) < (aval b s)
```

The semantics for the source language is a big-step semantics, which means that it models the entire execution of the program at once. This is a problem because function definitions in Lean have to be total, they have to terminate with a well defined

result for every input. [2] `while` is a Turing complete language which means that it is generally impossible to determine whether a given `while` program will terminate. [10, p. 73] Therefore the big-step semantics for `while` cannot be formalized using a function definition. Instead both *Concrete Semantics* and *The Hitchhiker's Guide* define the semantics for `while` as a relation using an inductive predicate using Isabelle and Lean respectively. [2, p. 128] [10, p. 79]

```
inductive big_step : stmt × LoVe.state → LoVe.state → Prop
| skip {s} :
  big_step (stmt.skip, s) s
| assign {x a s} :
  big_step (stmt.assign x a, s) (s{x ↦ aval a s})
| seq {S T s t u} (hS : big_step (S, s) t)
    (hT : big_step (T, t) u) :
  big_step (S ;; T, s) u
| ite_true {b : bexp} {S T s t} (hcond : bval b s)
    (hbody : big_step (S, s) t) :
  big_step (stmt.ite b S T, s) t
| ite_false {b : bexp} {S T s t} (hcond : ¬ bval b s)
    (hbody : big_step (T, s) t) :
  big_step (stmt.ite b S T, s) t
| while_true {b : bexp} {S s t u} (hcond : bval b s)
    (hbody : big_step (S, s) t)
    (hrest : big_step (stmt.while b S, t) u) :
  big_step (stmt.while b S, s) u
| while_false {b : bexp} {S s} (hcond : ¬ bval b s) :
  big_step (stmt.while b S, s) s
```

Arguments in curly brackets (e.g. `{s}`) are implicit arguments and can be used when Lean can infer them on its own. For example in the `while_true` case `{b : bexp}` can be inferred from the explicit `(hcond : bval b s)`.

This is an equivalent semantics to the one in fig. 1. Each constructor corresponds to one rule in the formal system in fig. 1. All the premises and side-conditions are written as arguments to each constructor, and the conclusion is written as arguments to the inductive predicate. The `skip` constructor for example has no arguments (except the implicit state `s`), and gives `(stmt.skip, s)` and `s` as arguments to `big_step`. This is interpreted to mean that `(stmt.skip, s)` and `s` are in the `big_step` relation i.e. according to this semantics running the program `skip` in state `s` results in state `s`, which is the same result as described in fig. 1.

A construction of `big_step (stmt, s) s'` for some states `s`, `s'` and a statement `stmt` is a proof that the statement `stmt` runs from state `s` to state `s'`. For example, `@big_step.skip s` is a proof that `skip` runs from state `s` to state `s`.[3] Here is a more elaborate example:

[3] The `@` sign tells Lean that the implicit arguments should be made explicit. In this case the implicit `{s}` in `big_step.skip` is made explicit.

```
lemma double_seq {s s' s'' s'''} {S T U : stmt}
                     ( hS : (S, s) ⟶ s')
                     ( hT : (T, s') ⟶ s'')
                     ( hU : (U, s'') ⟶ s''') :
                     ((S ;; T) ;; U, s) ⟶ s'''
:=
begin
  exact big_step.seq (big_step.seq hS hT) hU,
end
```

The lemma says that given proofs of three evaluations where each evaluation ends in the next evaluation's start state, a new evaluation can be derived from `s` to `s'''`. The proof is simply repeated applications of `big_step.seq`.

3.3 Target machine language semantics

The target for compilation is a simple stack machine. Like the source language it has a memory defined as a function from variable names to integers, and a stack that is defined as a list of integers. The semantics for the stack machine is described in fig. 2 as a small-step semantics. The stack machine's configuration is defined as a 3-tuple (i, s, stk) where the integer i is the instruction index, s the memory and stk the stack. P is a program and is defined as a list of instructions. None of the rules have a premise but they all have preconditions written on the right side for when the rule can be applied. For example, the first rule can be applied when the ith item in P is "`loadi` n". The effect of evaluating that instruction is that the program counter i is incremented, the state is left unchanged, and the integer n is pushed on the stack.

$\Rightarrow$ denotes the small-step relation for one instruction in the stack machine.

Each rule describes the execution of one instruction but doesn't say how instructions should be chained together. This is done by taking the reflexive transitive closure on the $\Rightarrow$ relation, written

$$P \vdash (i, s, stk) \Rightarrow* (i', s', stk')$$

Informally $c \Rightarrow* c'$ can be interpreted as "the stack machine can get from c to c' by executing $n \geq 0$ steps. Formally it is defined as the smallest superset of the $\Rightarrow$ relation that has the reflexivity and transitivity properties. This means that for any configuration c and program P, $P \vdash c \Rightarrow* c$ holds because the relation is reflexive. And, for any configurations c, c' and program P, $P \vdash c \Rightarrow* c'$ holds if $\exists c'', P \vdash c \Rightarrow* c'' \wedge P \vdash c'' \Rightarrow* c'$ because the relation is transitive.

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

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

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

$$\frac{}{P \vdash (i, s, a :: stk) \Rightarrow (i+1, s\{x \mapsto a\}, stk)}\, P[i] = \text{store } x$$

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

$$\frac{}{P \vdash (i, s, a :: b :: stk) \Rightarrow (i+1+n, s, stk)}\, P[i] = \text{jmpless } n \wedge b < a$$

$$\frac{}{P \vdash (i, s, a :: b :: stk) \Rightarrow (i+1, s, stk)}\, P[i] = \text{jmpless } n \wedge b \not< a$$

$$\frac{}{P \vdash (i, s, a :: b :: stk) \Rightarrow (i+1+n, s, stk)}\, P[i] = \text{jmpge } n \wedge b \geq a$$

$$\frac{}{P \vdash (i, s, a :: b :: stk) \Rightarrow (i+1, s, stk)}\, P[i] = \text{jmpge } n \wedge b \not\geq a$$

Figure 2: Small-step semantics for one instruction in the stack machine

3.4 Formalized target language semantics

The instructions of the stack machine are implemented in a straightforward manner. The following is translated in a similar way as the translation in section 3.2.

```
inductive instr : Type
| loadi   : int    -> instr
| load    : string -> instr
| add     :           instr
| store   : string -> instr
| jmp     : int    -> instr
| jmpless : int    -> instr
| jmpge   : int    -> instr
```

At first glance it looks like the small-step semantics for a single stack machine instruction could be formalized using a function. Each instruction terminates after one step after all. But this is not true because, once again, functions in Lean have to be total but the semantics is undefined for certain configurations. For example $\Rightarrow$ is not defined when $P[i] =$ `add` and the stack is empty. I therefore use an inductive predicate for the stack machine's semantics.

```
inductive iexec :  instr -> config -> config -> Prop
| loadi {i s stk} (n : int):
  iexec (instr.loadi n) (i,s, stk) (i + 1, s, n :: stk)
| load  {i s stk} (x : string) :
  iexec (instr.load x) (i, s, stk) (i + 1, s, s x :: stk)
| add   {i s stk a b} :
  iexec instr.add (i, s, a :: b :: stk) (i + 1, s, (a + b) :: stk)
| store {i s stk a} (x : string) :
  iexec (instr.store x) (i, s, a :: stk) (i + 1, s{x ↦ a}, stk)
| jmp {i s stk} (n : int) :
  iexec (instr.jmp n) (i,s, stk) (i + 1 + n, s, stk)
| jmpless_true {i s stk a b} (n : int) (hcond : b < a):
  iexec (instr.jmpless n) (i,s , a :: b :: stk) (i + 1 + n, s, stk)
| jmpless_false {i s stk a b} (n : int) (hcond : ¬ b < a):
  iexec (instr.jmpless n) (i, s, a :: b :: stk) (i + 1, s, stk)
| jmpge_true {i s stk a b} (n : int) (hcond : b ≥ a):
  iexec (instr.jmpge n) (i,s , a :: b :: stk) (i + 1 + n, s, stk)
| jmpge_false {i s stk a b} (n : int) (hcond : ¬ b ≥ a):
  iexec (instr.jmpge n) (i, s, a :: b :: stk) (i + 1, s, stk)
```

`exec1` is defined to to implement the $P[i] = instr$ parts of the preconditions in fig. 2. The notation `p ⊢ c ⟹ c'` is used to abbreviate `exec1 p c c'`

```
inductive exec1 : program -> config -> config -> Prop
| exec1 {p s stk c' x} {n : ℕ}
        (hnth : list.nth p n = some x )
```

```
      (hiexec : sem.iexec x
                          (n, s, stk)
                          c'):
  exec1 p (n, s, stk) c'
```

3.5 Compilation

The compilation of arithmetic expressions is fairly straightforward:

```
def acomp : aexp -> program
| (aexp.N i) := [instr.loadi i]
| (aexp.V x) := [instr.load x]
| (aexp.Plus a b) := (acomp a) ++ (acomp b) ++ [instr.add]
```

Concrete Semantics explains boolean compilation better than I do:

> The compilation schema for boolean expressions is best motivated by a preview of the layout of the code generated from *WHILE b DO c* shown in Figure 3 [figure number changed]. Arrows indicate jump instructions. Let *cb* be code generated from *b*. If *b* evaluates to $True$, the execution of *cb* should lead to the end of *cb*, continue with the execution of the code for *c* and jump back to the beginning of *cb*. If *b* evaluates to $False$, the execution of *cb* should jump behind all of the loop code. For example, when executing the compiled code for *WHILE And* b_1 b_2 *DO c*, after having found that b_1 evaluates to False we can safely jump out of the loop. There can be multiple such jumps: think of *And* b_1(*And* b_2 b_3).
>
> To support this schema, the *bexp* compiler takes two further parameters in addition to *b*: an offset *n* and a flag $f :: bool$ that determines for which value of *b* the generated code should jump to offset *n*. This enables us to perform a small bit of optimization: boolean constants do not need to execute any code; they either compile to nothing or to a jump to the offset, depending on the value of *f*. The *Not* case simply inverts *f*. The *And* case performs shortcut evaluation as explained above. The *Less* operator uses the *acomp* compiler for a_1 and a_2 and then selects the appropriate compare and jump instruction according to *f*.[10, p. 100]

Note that `cond` behaves like an if-then-else expression. The first argument is a boolean, and the two next arguments are the then-branch and the else-branch respectively. ↑ is a coercion operator that coerces the length from a natural number to an integer.

```
def bcomp : bexp -> bool -> int -> program
| (bexp.Bc v) f n := cond (v = f) [instr.jmp n] []
| (bexp.Not b) f n := bcomp b (!f) n
| (bexp.And b₁ b₂) f n :=
  let cb₂ := bcomp b₂ f n in
  let m := cond f (↑cb₂.length) (↑cb₂.length + n) in
```

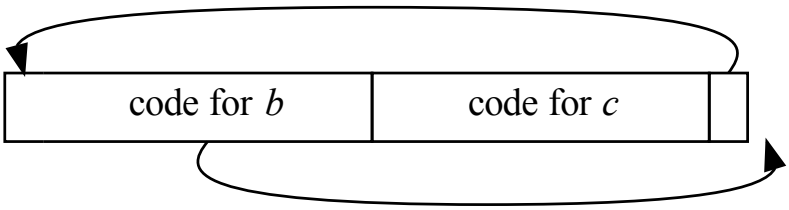

Figure 3: Compilation of *WHILE b DO c* from *Concrete Semantics*

```
  let cb₁ := bcomp b₁ ff m in
  cb₁ ++ cb₂
| (bexp.Less a₁ a₂) f n :=
  (acomp.acomp a₁) ++ (acomp.acomp a₂) ++ (cond f
                                                [instr.jmpless n]
                                                [instr.jmpge n])

def ccomp : stmt -> program
| (stmt.skip)        := []
| (stmt.assign x a)  := (acomp.acomp a) ++ [instr.store x]
| (stmt.seq a b)     := (ccomp a) ++ (ccomp b)
| (stmt.ite b c₁ c₂) :=
  let cc₁ := ccomp c₁ in
  let cc₂ := ccomp c₂ in
  -- If b is false, jump over the =then"-clause
  let cb  := bcomp.bcomp b ff (( list.length cc₁ ) + 1)
  -- The added jump instruction is there so that execution jumps over the
  -- "else"-clause after the "then"-clause is done.
  in cb ++ cc₁ ++ [instr.jmp (list.length cc₂)] ++ cc₂
| (stmt.while b c) :=
  let cc           := ccomp c in
  -- If b is false, jump over the body
  let cb           := bcomp.bcomp b ff (list.length cc + 1)
  -- The added jump instruction is there so that execution jumps back to
  -- the condition after executing the body
  in cb ++ cc ++ [instr.jmp (- (list.length cb + list.length cc + 1) )]
```

3.6 Correctness statement

The correctness statement is straightforward to formulate because the two languages have the same state type. The compilation is correct if both programs end up in the same state if they started in the same state. Below is the formal definition that is used in this report. Note that $\longrightarrow$ is the big-step relation for the source language, $\leftrightarrow$ is logical equivalence, $\rightarrow$ is logical implication and $\Rightarrow*$ is the small-step semantics for the target language.

```
theorem correct {c s t stk} :
  (c, s) ⟶ t
         ↔
  (ccomp c) ⊢ (0, s, stk) ⇒*
              (↑(list.length (ccomp c)), t, stk) :=
```

Unfortunately, due to time constraints only the forward implication was proven, i.e.:

```
theorem correct {c s t stk} :
  (c, s) ⟶ t
         →
  (ccomp c) ⊢ (0, s, stk) ⇒*
              (↑(list.length (ccomp c)), t, stk) :=
```

3.7 Proofs

3.7.1 Appending programs

Stack machine programs have some useful properties when they are concatenated:

- Appending a program from the right does not change whether the stack machine can reach a given configuration. More specifically, if the stack machine can reach a configuration, it can still reach that configuration if a program is appended from the right.

  ```
  lemma exec_append_right {p c c'} (p') :
        (p ⊢  c ⇒* c') → (p ++ p' ) ⊢ c ⇒* c'
  ```

- The result is similar when a program is appended from the left. The difference is that the program counter `i` needs to be adjusted so that it still points to the first instruction of program `p`.

  ```
  lemma exec_append_left {p s s' stk stk'} (p') {i i' : int} :
        (p ⊢ (i, s, stk) ⇒* (i', s', stk' )) →
          (p' ++ p ) ⊢ (i +  p'.length, s, stk) ⇒*
                       (i' + p'.length, s', stk')
  ```

3.7.2 Correctness of arithmetic expressions

The proof of the correctness statement has two sub-proofs: a proof that arithmetic expressions is compiled correctly, and a proof that boolean expressions are compiled correctly. All code examples in this section is part of the proof for the correctness of arithmetic expressions, except where otherwise noted.

The simplest correctness proof is that of arithmetic expression. The correctness statement is that evaluating a compiled arithmetic expression should put the program counter at the end of the program and put the value of the arithmetic expression on the top of the stack. The state is unchanged.

```
lemma correct {a s stk}  :
              (acomp a) ⊢ (0, s, stk) ⇒*
                          ↑(list.length (acomp a),
                            s,
                            (aval a s) :: stk) :=
begin
```

The proof is done by induction on the arithmetic expression `a`. Recall that an arithmetic expression has two base cases (`aexp.N` and `aexp.V`) and one inductive case (`aexp.Plus`). The two base cases are proven by proving the correctness statement but with `a` substituted for `aexp.N` and `aexp.V` respectively. This is trivially proven by taking a single step of the `⇒*` relation using the `loadi` and `load` cases respectively.

```
  induction' a,
  case aexp.N : n {
    show (acomp (aexp.N n)) ⊢
          (0, s, stk) ⇒*
          (↑(acomp (aexp.N n)).length,
           s,
           aval (aexp.N n) s :: stk),
    -- star.single takes an element of a relation ⟹ and creates the
    -- corresponding element of ⇒*
    exact star.single (exec1.exec1 (begin simp end) (iexec.loadi _)),
  },
  case aexp.V : x {
    show (acomp (aexp.V x)) ⊢
          (0, s, stk) ⇒*
          (↑(acomp (aexp.V x)).length,
           s,
           aval (aexp.V x) s :: stk),
    exact star.single (exec1.exec1 (begin simp end) (iexec.load _)),
  },
```

The inductive `Plus` case is more interesting. I will first explain using an informal syntax, and then show the corresponding Lean code. There are two induction hypotheses `ih_a` and `ih_a_1` that are the correctness statement for two arithmetic expressions `a` and

`a_1` respectively. The goal is to prove that the correctness statement holds for `acomp (aexp.Plus a a_1)`, which is compiled to `acomp a ++ (acomp a_1 ++ [instr.add])`. In other words, the goal is to construct the following term:

```
acomp a ++ acomp a_1 ++ [instr.add] ⊢
   (0, s, stk) ⇒*
   (list.length (acomp a_1) + list.length (acomp a) + 1,
    s,
    (aval a_1 s + aval a s) :: stk)
```

These are the induction hypotheses:

```
acomp a ⊢
  (0, s, stk) ⇒* (length (acomp a), s, aval a s :: stk)

-- The initial stack is chosen to be aval a s :: stk
acomp a_1 ⊢
   (0, s, aval a s :: stk) ⇒*
   (0,
    s,
    aval a_1 s :: aval a s :: stk)
```

The following terms can be obtained by appending to the right of, and appending to the left of the respective program.

```
-- acomp a_1 ++ [instr.add] appended to the right
l := acomp a ++ acomp a_1 ++ [instr.add] ⊢
       (0, s, stk) ⇒* (length (acomp a), s, aval a s :: stk)

-- acomp a appended to the left, and [instr.add] appended to the right
r := acomp a ++ acomp a_1 ++ [instr.add] ⊢
       (length (acomp a),
        s,
        aval a s :: stk) ⇒*
       (length (acomp a_1) + length (acomp a),
        s,
        aval a_1 s :: aval a s :: stk)
```

Since ⇒* is transitive, and the right side of the first term and the left side of the second term match, the following new term can be obtained.

```
acomp a ++ acomp a_1 ++ [instr.add] ⊢
   (0, s, stk) ⇒*
   (length (acomp a_1) + length (acomp a),
    s,
    aval a_1 s :: aval a s :: stk)
```

The following term can be directly constructed because it is only one step of the ⇒ relation.

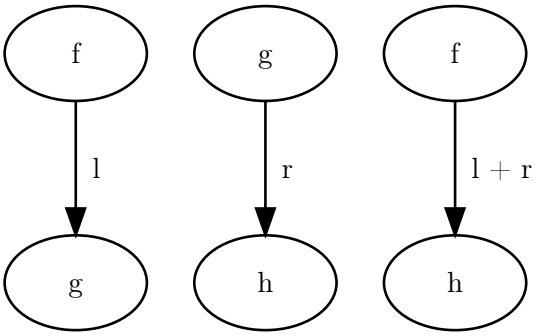

Figure 4: Constructing a new term using transitivity.

```
[instr.add] ⊢
  (0, s, aval a_1 s :: aval a s :: stk) ⇒*
  (1, s, (aval a_1 s + aval a s) :: stk)
-- acomp a ++ acomp a_1 appended to the left
acomp a ++ acomp a_1 ++ [instr.add] ⊢
  (length (acomp a_1) + length (acomp a),
   s,
   aval_1 s :: aval a s :: stk) ⇒*
  (length (acomp a_1) + length (acomp a) + 1,
   s,
  (aval a_1 s + aval a s) stk)
```

The terms once again match on the left and right hand sides, so the following term can be constructed using transitivity. This is the sought after term, and the proof is therefore concluded.

```
acomp a ++ acomp a_1 ++ [instr.add] ⊢
   (0, s, stk) ⇒*
   (length (acomp a_1) + length (acomp a) + 1,
    s,
    (aval a_1 s + aval a s) :: stk)
```

Here is the `Plus` case in written in Lean.

```
  case aexp.Plus {
    show (acomp (aexp.Plus a a_1)) ⊢
              (0, s, stk) ⇒*
              (↑(list.length (acomp (aexp.Plus a a_1))),
               s,
               aval (aexp.Plus a a_1) s :: stk),
    simp,
    -- Run the program through the first arithmetic sub-expression
    -- Apppend so that we get the full program instead of just one
```

```
-- sub-expression
have h_run_a : (acomp a ++ (acomp a_1 ++ [instr.add])) ⊢
                    (0, s, stk) ⇒*
                    (↑(list.length (acomp a)),
                     s,
                     aval a s :: stk)
  := exec_append_right (acomp a_1 ++ [instr.add]) (@ih_a s stk),
clear ih_a,
-- Run the program through the second arithmetic
-- sub-expression
have h_run_a_1 : (acomp a_1) ⊢
                      (0, s, aval a s :: stk) ⇒*
                      (↑(list.length (acomp a_1)),
                       s,
                       aval a_1 s :: aval a s :: stk)
  := @ih_a_1 s ((aval a s) :: stk),
clear ih_a_1,
-- apppend so that we get the full program instead of just one
-- sub-expression
have h_run_a_1' : (acomp a ++ acomp a_1 ++ [instr.add]) ⊢
                      (0 + ↑(list.length (acomp a)),
                       s,
                       aval a s :: stk) ⇒*
                      (↑(list.length (acomp a_1)) +
                        ↑(list.length (acomp a)),
                       s,
                       aval a_1 s :: aval a s :: stk)
  := exec_append_right [instr.add]
                      (exec_append_left (acomp a)  h_run_a_1),
simp at h_run_a_1',

-- Use transitivity to construct an evaluation of the first
-- two sub-expressions
have h_run_a_and_a_1 := star.trans h_run_a h_run_a_1',
clear h_run_a h_run_a_1 h_run_a_1',

-- There is only one instruction left, instr.add. We can
-- therefore construct the final star with h_run_a_1' and a
-- new [instr.add] ⊢ _ ⟹ _

--for that we need one hnth
have hnth : list.nth ((acomp a ++ (acomp a_1 ++ [instr.add])))
                     ((acomp a_1).length +
                      (acomp a).length)
```

```
                = some instr.add :=
    begin
      rw list.nth_append_right,
      simp,
      simp,
    end,
    -- and a hiexec
    have hiexec  :=
      @iexec.add (↑(acomp a_1).length + ↑(acomp a).length)
                     s stk (aval a_1 s) (aval a s),
    have hexec1_add := exec1.exec1 hnth hiexec,
    -- normalize additions
    abel at h_run_a_and_a_1,
    abel at hexec1_add,
    abel,
    -- Use transitivity to construct an evaluation of the
    -- whole program
    exact star.trans h_run_a_and_a_1 (star.single hexec1_add),
  }
end
```

The other correctness proofs have the same structure. They are induction proofs on the boolean expression and statement respectively. For each inductive case the goal is to construct a target element in the $\Rightarrow*$ relation where the program is a concatenation of the programs used in the induction hypotheses. The other proofs are much more involved with a lot of cases, so they are not included here.

4 Result

The result of the work presented here is a formalization of: the semantics of the language `while`, the semantics of a simple stack machine, and a proof that a compiler from while to the stack machine is correct for terminating evaluations. This was done using the interactive theorem prover Lean. The full proof can be found in the appendix (section 8) or on Mathlib's Github repository.[4] The final correctness proof (called `correct` in the code), and especially the proof of the correctness of arithmetic expressions (`acomp_correct`), are written in a very explicit style with little automation to improve the readability.

5 Discussion

5.1 $\Rightarrow*$ or $\Rightarrow k$

One common technique for proving theorems about a small-step semantics is to use $\Rightarrow k$, which is interpreted as k steps of evaluation. This allows for more precise descriptions of

[4] `https://github.com/Zetagon/mathlib/blob/master/archive/whilecc.lean`

the execution steps and opens up the possibility for other induction principles such as mathematical or complete induction. Some readers may wonder why this approach was not used here instead of $\Rightarrow*$. The boring answer is that *Concrete Semantics* from which the proof was translated also uses $\Rightarrow*$ for this part of the proof, most likely because the extra flexibility is not needed. However, they do use $\Rightarrow k$ for the more involved backwards direction.

5.2 Proved correct only in the forward direction

Due to time constraints only one direction of the equivalence was proved: if the source program terminates in state `s`, so should the compiled program (see section 3.6). This means that the proof says something only when the premise is true, when the source program terminates. The compiled program could do anything if the source program does not terminate as far as the proof is concerned. One way to fill this hole is to prove the other direction: if the compiled program terminates in state `s`, so should the source program. See *Concrete Semantics* [10, p. 105] for more details on how this can be done.

Another potential change to make is to use small-step semantics instead of big-step semantics for the source language. Big-step semantics does not handle non-termination very well as it can only describe terminating evaluations. All non-terminating evaluations are put in the same black box removing any possibility of distinguishing them. Small-step semantics on the other hand can talk about each individual step of an evaluation, even if the evaluation consists of an infinite number of steps.

Below is an alternative correctness statement that uses a hypothetical small-step relation $\longrightarrow*$ for the semantics of the source language instead of the big-step semantics used in this paper. Note that the program counter can start at any number and land on any number, as opposed to starting at 0 and ending at `length (ccomp S)`. Similarly there are no restrictions on the stack. This is because this correctness statement does not, in contrast to the correctness statement in section 3.6, aim to describe the complete evaluation, from start to finish. Instead it describes *all possible* evaluation sequences even if they do not arrive at a terminating state.

```
( (S, s) ⟶* (T, t) ) → ( ∃ i i' stk stk', ccomp S ⊢ (i, s, stk) ⇒* (i
    ', t, stk') )
```

The correctness statement states that for all state transitions in the source program's evaluation, an equivalent state transition exists for the compiled program. Even if the evaluation never halts, there will be zero or more state transitions, so there are no program-state pairs for which the evaluation does not satisfy the predicate. The astute reader may notice that some form of proof that the right hand side is deterministic, either that the stack machine is deterministic or, that all compiled programs (which requires a proof that `while` is deterministic). This is because the correctness statement alone only says that there is **an** equivalent stack machine evaluation, not that it is the **only** evaluation. In other words a proof of that statement alone does not guarantee that there is not an evaluation that is incorrect, which is why the additional determinism proof is required.

This is not so immediately useful for a language with no side effects except mutation, but in languages with more side effects non-termination is still useful. It is then important that each side effect (which are part of the intermediate steps of the evaluation) is performed correctly in the right order. Sadly this approach makes some optimizations harder, or perhaps impossible, to formalize as they can remove state transitions that do not affect the final result.

Another problem is that, as Leroy et al. note [6], big-step semantics is easier to use for correctness proofs of compilers (from higher level to lower level languages). With big-step semantics the proofs are often simple structural induction proofs on the structure of the evaluation derivations.[5] Nipkow et al. also describe similar difficulties in *Concrete Semantics* [10, p. 105] when they compare the difficulty of proving the forward (big-step $\rightarrow$ small-step) to the backward (small-step $\rightarrow$ big-step) direction of the correctness proof of the compiler used in this report.

5.3 Difference between Lean and Isabelle

Lean and Isabelle/HOL behave differently which necessitated some changes in the formalized semantics.

Functions in Lean are required to be total, meaning that they need to be defined for all values in the domain. This means that non-termination and partial pattern matching are not allowed since their values are not always well-defined. Function definitions in Isabelle/HOL however, allows for both. Partial pattern matching and non-terminating function definitions results in a value, but the concrete value might not be known. `hd` for example is defined like this:

```
fun hd :: 'a list ⇒ 'a
hd (x # xs) = x
```

`hd []` is defined, but the value is unknown. Such a function is called "underdefined"[10, p. 14].

Concrete Semantics uses `hd` when defining `iexec` which makes the translation to Lean non-trivial since partial patterns are illegal in Lean. As described in section 3.4 I have chosen to implement `iexec` with an inductive predicate instead of as a function like *Concrete Semantics* does. This has in practice not lead to any issues since the compilation ensures that there is always enough elements on the stack for the instructions that use the stack. For example, the `add` instruction can only come from a compilation of `Plus` a_1 a_2, which means that there will always be two integers on the stack when an `add` is evaluated.

```
-- Concrete Semantic's definition
definition exec1 :: instr list ⇒ config ⇒ config ⇒ bool where
P ⊢ c → c' =
(∃ i s stk. c = (i , s, stk ) ∧ c'  = iexec (P !! i ) c ∧ i ≤ size P )
```

[5] Readers interested in this problem want to read their paper [6] where they propose a technique to describe both terminating and non-terminating evaluations using big-step semantics.

`exec1` (or `p ⊢ c ⇒* c'`) as defined in *Concrete Semantics* is hard to work with in Lean. There are two ways to get an element at a given index from a list: `nth : list α → nat → option α` and `nth_le (l : list α) (n : ℕ) : n < l.length → α` [6]. A direct translation of the Isabelle code above would use `nth_le`:

```
inductive exec1 : program -> config -> config -> Prop
| exec1 {p s stk c'} {i : ℕ}
        (hle : (i < list.length p))
        (hiexec : sem.iexec (nth_le p i hle)
                            (i, s, stk)
                            c'):
  exec1 p (i, s, stk) c'
```

In this simplified version of a definition of `exec1` that was used early in development `hiexec` is dependent on the proof `hle`, which creates an unnecessarily tight binding between the arguments of `exec1`. The version that is used in this report (see section 3.4) uses `nth` instead which creates less dependence between the arguments. It made proof writing easier and should not affect the validity of the proof since they express the same condition: that there exists an ith element in the program p.

5.4 Factors not accounted for

The correctness of the compilation of (terminating) programs have been proven, but that does not mean that everything is accounted for.

- Parsing

  The compiler does not include nor formalize a parser.

- Interpretation of semantics

  The semantics of both the source language and the target language have been translated by the author to Lean. This translation is not machine checked. Any faults in the semantics will of course invalidate the correctness proof.

6 Related Work

The most obvious related work is that of Nipkow et al. with *Concrete Semantics* [10]. They present the semantics for `while` and stack machine, the compiler and correctness proof that I have translated in this report. They also show how to formalize a simple type system and prove its consistency, and how to do various program analyses and optimizations (definite initialization analysis, constant folding and propagation, and live variable analysis) and prove their correctness, all in the interactive theorem prover Isabelle/HOL.

Baanen et al.'s work with *The Hitchhiker's Guide* [2] is partly inspired by *Concrete Semantics*. Relevant to this report is that they show a method of formalizing both a

[6] `option` is equivalent to `Maybe` in Haskell.

big-step and small-step semantics (notably they present the formalization of `while`'s semantics used in this report), and how to prove theorems about them.

Jinja is a Java-like language that compiles to the Java Virtual Machine. The compiler has been formally verified to be correct and the language verified to be type safe [5]. Similarly to the proof presented here, the correctness proof only holds for terminating evaluations. It could be the case that some compiled programs' evaluation terminates even though the source programs' evaluation diverges.

Leroy et al. present another approach to cover the case of non-terminating evaluations using a form of big-step semantics that uses coinduction [6]. This allows them to avoid proving the backwards case (if the compiled program terminates in state s, so should the source program) which can be difficult, especially if the compilation does certain optimizations. Kästner et al. use this approach in CompCert, a formally verified optimizing C compiler [4, 6]. CompCert is written in Coq, an interactive theorem prover that is fairly similar to Lean since they both use share the same underlying theoretical framework: the dependent type theory Calculus of Constructions. In contrast *Concrete Semantics* and Jinja use Isabelle/HOL which has higher-order logic as base.

Finally Xi Wang translated a compiler for arithmetic expressions, and the associated correctness proof based on the work of Mccarthy et al. [7]. The source and target language is similar to the arithmetic expressions and the stack machine presented in this report, with the notable exception that jump expressions are left out since they are not needed for arithmetic. Their work can be found in Mathlib's Github repository[7].

7 Conclusion and future work

This report presents a translation of the forward direction of the correctness proof presented in *Concrete Semantics: with Isabelle/HOL*, proving that the compiler for the simple imperative language `while` is correct for terminating programs. This translation is from Isabelle/HOL that *Concrete Semantics* uses into Lean. There is still much left to translate *Concrete Semantics*, most notably the proof that `while` is correct for non-terminating evaluations, but also the static type system and optimizations. Future work also includes making the language usable on common platforms by compiling stack machine programs to a more commonly used instruction set such as x86 assembly, Java bytecode or WebAssembly I hope this work can serve as a foundation and useful resource for others who are getting into interactive theorem proving and program verification in Lean.

[7] `https://github.com/leanprover-community/mathlib/blob/80d02347a7866c7ea0f9c6c67d14655b830de594/archive/arithcc.lean`

Glossary

Symbols

$\longrightarrow$ The big-step semantics relation of the source language 6, 7, 15

$\Rightarrow$ The small-step semantics relation for one instruction of the stack machine 10

$\Rightarrow*$ The small-step semantics relation for the target language 10, 15

$\leftrightarrow$ Logical equivalence 15

$\rightarrow$ Logical implication or function type signature 15

References

[1] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean. Release 3.23.0.* Jan. 21, 2021. URL: `https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf` (visited on 05/18/2021) (cit. on pp. 1, 4, 5).

[2] Anne Baanen, Alexander Bentkamp, Jasmin Blanchette, Johannes Hölzl, and Jannis Limperg. *The Hitchhiker's Guide to Formal Verification.* 2020 (cit. on pp. 1, 2, 4, 9, 23).

[3] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. "A Metaprogramming Framework for Formal Verification". In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017), pp. 1–29. DOI: `10.1145/3110278` (cit. on p. 5).

[4] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. "Closing the gap – The formally verified optimizing compiler CompCert". In: *SSS'17: Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium.* CreateSpace, 2017, pp. 163–180 (cit. on pp. 1, 24).

[5] Gerwin Klein and Tobias Nipkow. "A machine-checked model for a Java-like language, virtual machine, and compiler". In: *ACM Transactions on Programming Languages and Systems* 28.4 (July 2006). Jinja, pp. 619–695. DOI: `10.1145/1146809.1146811` (cit. on p. 24).

[6] Xavier Leroy and Hervé Grall. "Coinductive big-step operational semantics". In: *Information and Computation* 207.2 (2009), pp. 284–304. URL: `http://xavierleroy.org/publi/coindsem-journal.pdf` (cit. on pp. 22, 24).

[7] John Mccarthy and James Painter. "Correctness of a compiler for arithmetic expressions". In: American Mathematical Society, 1967, pp. 33–41 (cit. on p. 24).

[8] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25*. Automated Deduction - CADE-25. Springer International Publishing, 2015, pp. 378–388. DOI: 10.1007/978-3-319-21401-6_26 (cit. on pp. 4, 5).

[9] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer London, 2007. DOI: 10.1007/978-1-84628-692-6 (cit. on pp. 1, 6).

[10] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: with Isabelle/HOL*. Cham: Springer, 2014. DOI: 10.1007/978-3-319-10542-0 (cit. on pp. 1, 6, 9, 13, 14, 21–24).

[11] Philip Wadler. "Propositions As Types". In: *Communications of the ACM* 58.12 (2015), pp. 75–84. DOI: 10.1145/2699407 (cit. on pp. 2, 3).

8 Appendix

```
import algebra
import data.real.basic
import data.vector
import tactic.explode
import tactic.find
import tactic.induction
import tactic.linarith
import tactic.rcases
import tactic.rewrite
import tactic.ring_exp
import tactic.tidy
import tactic.where

set_option pp.beta true
set_option pp.coercions true
set_option pp.generalized_field_notation false
set_option trace.check true
set_option pp.notation true
/- # LoVe Library
 Copied from https://github.com/blanchette/logical_verification_2020/
 -/
namespace LoVe
namespace rtc

inductive star {α : Sort*} (r : α → α → Prop) (a : α) : α → Prop
| refl {}    : star a
```

```
| tail {b c} : star b → r b c → star c

attribute [refl] star.refl

namespace star

variables {α : Sort*} {r : α → α → Prop} {a b c d : α}

@[trans] lemma trans (hab : star r a b) (hbc : star r b c) :
  star r a c :=
begin
  induction' hbc,
  case refl {
    assumption },
  case tail : c d hbc hcd hac {
    exact (tail (hac hab)) hcd }
end

lemma single (hab : r a b) :
  star r a b :=
refl.tail hab
end star
end rtc

def state :=
string → int

def state.update (name : string) (val : int) (s : state) : state :=
λname', if name' = name then val else s name'

notation s `{` name ` ↦ ` val `}` := state.update name val s

instance : has_emptyc state :=
{ emptyc := λ_, 0 }

export rtc
end LoVe

open LoVe

namespace bool
lemma or_intro (b : bool) : b = tt ∨ b = ff :=
begin
  cases' em (b = ff),
```

```
  {
    apply or.intro_right, exact h,
  },
  { simp at h,
    apply or.intro_left,
    exact h,
  },
end
end bool

lemma le_iff_not_lt {a b : int} :  a ≤ b -> ¬ b < a :=
begin
  intro h,
  simp,
  assumption,
end

lemma not_le_iff_lt {a b : int} :  ¬ a ≤ b -> b < a :=
begin
  intro h,
  simp * at *,
end

lemma cond_add (P) (a b : ℤ) : (cond P a (a + b) = a + (cond P 0 b)) :=
begin
  cases' P,
  { simp, },
  { simp, },
end

namespace list

lemma nth_some_append : ∀ {α : Type} {l : list α} {l' : list α} x i,
                          list.nth l i = some x → list.nth (l ++ l') i
   = some x :=
begin
  intros α l l' x i,
  induction' l,
  { intro h,
    exfalso,
    simp at h,
    assumption,
  },
  { intro h,
```

```
    cases' i,
    { simp at h,
      simp,
      assumption,
    },
    { simp,
      simp at h,
      exact @ih l' x i h,
    }
  }
end

lemma nth_some_append_left : ∀ {α : Type} {l : list α} {l' : list α} x i,
                             list.nth l i = some x → list.nth (l' ++ l) (i
    + list.length l') = some x :=
begin
  intros α l l' x i,
  induction' l',
  { intro h,
    simp,
    assumption,
  },
  { intro h,
    cases' i,
    { simp,
      simp at h,
      have ih' := @ih l x 0,
      simp at ih',
      exact ih' h,
    },
    { simp [list.nth],
      rw nat.succ_add,
      simp [list.nth],
      have ih' := @ih l x (i + 1),
      have haddeq : (i + (list.length l' + 1)) = (i + 1 + list.length l')
    :=
        by linarith,
      rw haddeq,
      exact ih' h,
    },
  },
end
```

```
lemma zero_le_length_add_one {α : Type}  {xs : list α} : 0 ≤ ↑(length xs)
    + (1 : ℤ) :=
begin
  induction' xs,
  { simp, linarith, },
  { simp, linarith, }
end

end list

section sem

inductive aexp : Type
| N    : int → aexp
| V    : string → aexp
| Plus : aexp → aexp → aexp


@[simp] def aval : aexp → LoVe.state → int
| (aexp.N i) _ := i
| (aexp.V x) s := s x
| (aexp.Plus a b) s := (aval a s) + (aval b s)


inductive bexp : Type
| Bc : bool → bexp
| Not : bexp → bexp
| And : bexp → bexp → bexp
| Less : aexp → aexp → bexp


@[simp] def bval : bexp → LoVe.state → bool
| (bexp.Bc b) _ := b
| (bexp.Not x) s := not (bval x s)
| (bexp.And a b) s := (bval a s) ∧ (bval b s)
| (bexp.Less a b) s := (aval a s) < (aval b s)

inductive stmt : Type
| skip   : stmt
| assign : string → aexp → stmt
| seq    : stmt → stmt → stmt
| ite    : bexp → stmt → stmt → stmt
| while  : bexp → stmt → stmt
```

```
infixr ` ;; ` : 90 := stmt.seq

inductive big_step : stmt × LoVe.state → LoVe.state → Prop
| skip {s} :
  big_step (stmt.skip, s) s
| assign {x a s} :
  big_step (stmt.assign x a, s) (s{x ↦ aval a s})
| seq {S T s t u} (hS : big_step (S, s) t)
    (hT : big_step (T, t) u) :
  big_step (S ;; T, s) u
| ite_true {b : bexp} {S T s t} (hcond : bval b s)
    (hbody : big_step (S, s) t) :
  big_step (stmt.ite b S T, s) t
| ite_false {b : bexp} {S T s t} (hcond : ¬ bval b s)
    (hbody : big_step (T, s) t) :
  big_step (stmt.ite b S T, s) t
| while_true {b : bexp} {S s t u} (hcond : bval b s)
    (hbody : big_step (S, s) t)
    (hrest : big_step (stmt.while b S, t) u) :
  big_step (stmt.while b S, s) u
| while_false {b : bexp} {S s} (hcond : ¬ bval b s) :
  big_step (stmt.while b S, s) s

infix ` ⟶ ` : 110 := big_step

inductive instr : Type
| loadi   : int    → instr
| load    : string → instr
| add     :          instr
| store   : string → instr
| jmp     : int    → instr
| jmpless : int    → instr
| jmpge   : int    → instr


notation `config` := int × state × list int
notation `program` := list instr


inductive iexec :  instr → config → config → Prop
| loadi {i s stk} (n : int):
  iexec (instr.loadi n) (i,s, stk) (i + 1, s, n :: stk)
| load  {i s stk} (x : string) :
```

```
  iexec (instr.load x) (i, s, stk) (i + 1, s, s x :: stk)
| add   {i s stk a b} :
  iexec instr.add (i, s, a :: b :: stk) (i + 1, s, (a + b) :: stk)
| store {i s stk a} (x : string) :
  iexec (instr.store x) (i, s, a :: stk) (i + 1, s{x ↦ a}, stk)
| jmp {i s stk} (n : int) :
  iexec (instr.jmp n) (i,s, stk) (i + 1 + n, s, stk)
| jmpless_true {i s stk a b} (n : int) (hcond : b < a):
  iexec (instr.jmpless n) (i,s , a :: b :: stk) (i + 1 + n, s, stk)
| jmpless_false {i s stk a b} (n : int) (hcond : ¬ b < a):
  iexec (instr.jmpless n) (i, s, a :: b :: stk) (i + 1, s, stk)
| jmpge_true {i s stk a b} (n : int) (hcond : b ≥ a):
  iexec (instr.jmpge n) (i,s , a :: b :: stk) (i + 1 + n, s, stk)
| jmpge_false {i s stk a b} (n : int) (hcond : ¬ b ≥ a):
  iexec (instr.jmpge n) (i, s, a :: b :: stk) (i + 1, s, stk)


inductive exec1 : program → config → config → Prop
| exec1 {p s stk c' x} {n : ℕ}
        (hnth : list.nth p n = some x )
        (hiexec : iexec x
                        (n, s, stk)
                        c'):
  exec1 p (n, s, stk) c'

reserve infixl `⊢`:70

reserve infixr ` ⇒* ` : 40
reserve infixr ` ⟹ ` : 40

notation p ` ⊢ ` c ` ⇒* ` :100 c'  := (star (exec1 p) c c')
notation p ` ⊢ ` c ` ⟹ ` :100 c'  := exec1 p c c'

lemma exec1_append {p p' : program } {c c' : config}: (p ⊢ c ⟹ c') →
    ((p ++ p') ⊢ c ⟹ c') :=
begin
  intro h,
  cases' h,
  apply exec1.exec1 (@list.nth_some_append instr p p' x n hnth),
  {
    assumption,
  }
end
```

```
lemma relocate_instruction {x i i' s  stk} {s' : LoVe.state} {stk' : list
     int} (n) :
       iexec x (i + n, s, stk) (i' + n, s', stk') ↔ iexec x (i, s, stk) (
    i', s', stk')
:=
begin
  split,
  { intro h,
    cases' h,
    any_goals {
      have hi : i + 1 = i' := by linarith,
      rw ← hi,
    },
    any_goals {
      have hi : i + 1 + n_1 = i' :=
            by linarith,
      rw ← hi,
    },
    {
      apply iexec.loadi,
    },
    {
      apply iexec.load,
    },
    {
      apply iexec.add,
    },
    {
      apply iexec.store,
    },
    {
      apply iexec.jmp,
    },
    {
      apply @iexec.jmpless_true i s stk a b n_1 hcond,
    },
    {
      apply @iexec.jmpless_false i s stk a b n_1 hcond,
    },
    {
      apply @iexec.jmpge_true i s stk a b n_1 hcond,
    },
    {
      apply @iexec.jmpge_false i s stk a b n_1 hcond,
```

```
  }
},
{ intro h,
  have tmp_add : i + 1 + n = (i + n) + 1 := by linarith,
  cases' h,
  {
    rw tmp_add,
    apply iexec.loadi,
  },
  {
    rw tmp_add,
    apply iexec.load,
  },
  {
    rw tmp_add,
    apply iexec.add,
  },
  {
    rw tmp_add,
    apply iexec.store,
  },
  {
    have tmp_add : i + 1 + n_1 + n = ((i + n) + 1) + n_1 := by linarith,

    rw tmp_add,
    apply iexec.jmp,
  },
  {
    have tmp_add : i + 1 + n_1 + n = ((i + n) + 1) + n_1 := by linarith,

    rw tmp_add,
    apply iexec.jmpless_true n_1 hcond,
  },
  {
    rw tmp_add,
    apply iexec.jmpless_false n_1 hcond,
  },
  {
    have tmp_add : i + 1 + n_1 + n = ((i + n) + 1) + n_1 := by linarith,

    rw tmp_add,
    apply iexec.jmpge_true n_1 hcond,
  },
  {
```

```
      rw tmp_add,
      apply iexec.jmpge_false n_1 hcond,
    },

  },
end

lemma exec1_append_left {p p' : program} { s s' : LoVe.state }
                        {stk stk' : list int} {i i' : int} :
      (p ⊢ (i, s, stk) ⟹ (i', s', stk' )) →
        (p' ++ p ) ⊢ (i + list.length p', s, stk) ⟹
                     (i' + list.length p', s', stk') :=
begin
  intro h,
  cases' h,
  apply exec1.exec1 (@list.nth_some_append_left instr p p' x n hnth),
  apply (iff.elim_right (@relocate_instruction x n i' s stk s' stk' (list.
   length p'))),
  assumption,
end

lemma exec_append_right {p c c'} (p') :
      (p ⊢  c ⇒* c') → (p ++ p' ) ⊢ c ⇒* c' :=
begin
  intro h,
  cases' c with i s,
  cases' s with s stk,
  cases' c' with i' s',
  cases' s' with s' stk',
  induction' h,
  { apply star.refl, },
  { have h_2 : exec1 (p ++ p') b (i', s', stk') := exec1_append h_1,
    cases' h,
    { exact star.tail (star.refl) (h_2), },
    { have h_3' : exec1 (p ++ p') b b_1 :=  exec1_append h_3,
      clear h_3,
      rename h_3' h_3,
      cases' b_1 with bi bs,
      cases' bs with bs bstk,
      have ih' :_ := @ih p' bi bs bstk,
      have tmp : (bi, bs, bstk) = (bi, bs, bstk) := by refl,
      have ih'' :_ := ih' tmp,
      have tmp' :_ := star.tail ih'' h_2,
      assumption,
```

```
    },
  },
end

lemma exec_append_left {p s s' stk stk'} (p') {i i' : int} :
      (p ⊢ (i, s, stk) ⇒* (i', s', stk' )) →
        (p' ++ p ) ⊢ (i +  p'.length, s, stk) ⇒*
                     (i' + p'.length, s', stk') :=
begin
  intro h,
  induction' h,
  { apply star.refl, },
  {
    cases' b with bi bs,
    cases' bs with bs bstk,
    have tmp: (bi, bs, bstk) = (bi, bs, bstk) := by refl,
    have ih' := @ih bs bstk p' bi,
    clear ih,
    have tmp' := ih' tmp,
    apply star.tail,
    { exact tmp', },
    { exact exec1_append_left h_1
    }
  },
end

def aexp.repr : aexp → string
| (aexp.N n) := "N " ++ (repr n)
| (aexp.V x) := "V " ++ (repr x)
| (aexp.Plus a b) := (aexp.repr a) ++ (aexp.repr b)
instance : has_repr aexp := ⟨aexp.repr⟩

def bexp.repr : bexp → string
| (bexp.Bc b) := "Bc " ++ repr b
| (bexp.Not b) := "Not" ++ bexp.repr b
| (bexp.And a b) := (bexp.repr a ) ++ " ∧ " ++ (bexp.repr b )
| (bexp.Less a b) := (repr a ) ++ " < " ++ (repr b )
instance : has_repr bexp := ⟨bexp.repr⟩

def instr.repr : instr → string
| (instr.loadi n) := "loadi " ++ repr n
| (instr.load x) := "load " ++ repr x
| (instr.add) := "add"
| (instr.store x) := "store " ++ repr x
```

```
| (instr.jmp n) := "jmp " ++ repr n
| (instr.jmpless n) := "jmpless " ++ repr n
| (instr.jmpge n) := "jmpge " ++ repr n

instance : has_repr instr := ⟨instr.repr⟩


end sem



namespace acomp

@[simp] def acomp : aexp → program
| (aexp.N i) := [instr.loadi i]
| (aexp.V x) := [instr.load x]
| (aexp.Plus a b) := (acomp a) ++ (acomp b) ++ [instr.add]



lemma correct {a s stk}  :
              (acomp a) ⊢ (0, s, stk) ⇒*
                          ↑(list.length (acomp a),
                            s,
                            (aval a s) :: stk) :=
begin
  induction' a,
  case aexp.N : n {
    show (acomp (aexp.N n)) ⊢
          (0, s, stk) ⇒*
          (↑(acomp (aexp.N n)).length,
           s,
           aval (aexp.N n) s :: stk),
    -- star.single takes an element of a relation ⟹ and creates the
    -- corresponding element of ⇒*
    exact star.single (exec1.exec1 (begin simp end) (iexec.loadi _)),
  },
  case aexp.V : x {
    show (acomp (aexp.V x)) ⊢
          (0, s, stk) ⇒*
          (↑(acomp (aexp.V x)).length,
           s,
           aval (aexp.V x) s :: stk),
    exact star.single (exec1.exec1 (begin simp end) (iexec.load _)),
```

```
},
case aexp.Plus {
  show (acomp (aexp.Plus a a_1)) ⊢
            (0, s, stk) ⇒*
            (↑(list.length (acomp (aexp.Plus a a_1))),
             s,
             aval (aexp.Plus a a_1) s :: stk),
  simp,
  -- Run the program through the first arithmetic sub-expression
  -- Apppend so that we get the full program instead of just one
  -- sub-expression
  have h_run_a : (acomp a ++ (acomp a_1 ++ [instr.add])) ⊢
                      (0, s, stk) ⇒*
                      (↑(list.length (acomp a)),
                       s,
                       aval a s :: stk)
    := exec_append_right (acomp a_1 ++ [instr.add]) (@ih_a s stk),
  clear ih_a,
  -- Run the program through the second arithmetic
  -- sub-expression
  have h_run_a_1 : (acomp a_1) ⊢
                        (0, s, aval a s :: stk) ⇒*
                        (↑(list.length (acomp a_1)),
                         s,
                         aval a_1 s :: aval a s :: stk)
    := @ih_a_1 s ((aval a s) :: stk),
  clear ih_a_1,
  -- apppend so that we get the full program instead of just one
  -- sub-expression
  have h_run_a_1' : (acomp a ++ acomp a_1 ++ [instr.add]) ⊢
                        (0 + ↑(list.length (acomp a)),
                         s,
                         aval a s :: stk) ⇒*
                        (↑(list.length (acomp a_1)) +
                          ↑(list.length (acomp a)),
                         s,
                         aval a_1 s :: aval a s :: stk)
    := exec_append_right [instr.add]
                        (exec_append_left (acomp a)  h_run_a_1),
  simp at h_run_a_1',

  -- Use transitivity to construct an evaluation of the first
  -- two sub-expressions
  have h_run_a_and_a_1 := star.trans h_run_a h_run_a_1',
```

```
    clear h_run_a h_run_a_1 h_run_a_1',

    -- There is only one instruction left, instr.add. We can
    -- therefore construct the final star with h_run_a_1' and a
    -- new [instr.add] ⊢ _ ⟹ _

    --for that we need one hnth
    have hnth : list.nth ((acomp a ++ (acomp a_1 ++ [instr.add])))
                         ((acomp a_1).length +
                          (acomp a).length)
                = some instr.add :=
    begin
      rw list.nth_append_right,
      simp,
      simp,
    end,
    -- and a hiexec
    have hiexec  :=
      @iexec.add (↑(acomp a_1).length + ↑(acomp a).length)
                 s stk (aval a_1 s) (aval a s),
    have hexec1_add := exec1.exec1 hnth hiexec,
    -- rearrange some additions so they match with the goal
    abel at h_run_a_and_a_1,
    abel at hexec1_add,
    abel,
    -- Use transitivity to construct an evaluation of the
    -- whole program
    exact star.trans h_run_a_and_a_1 (star.single hexec1_add),
  }
end


end acomp

namespace bcomp
@[simp] def bcomp : bexp → bool → int → program
| (bexp.Bc v) f n := cond (v = f) [instr.jmp n] []
| (bexp.Not b) f n := bcomp b (!f) n
| (bexp.And b₁ b₂) f n :=
  let cb₂ := bcomp b₂ f n in
  let m := cond f (↑cb₂.length) (↑cb₂.length + n) in
  let cb₁ := bcomp b₁ ff m in
  cb₁ ++ cb₂
| (bexp.Less a₁ a₂) f n :=
```

```
  (acomp.acomp a₁) ++ (acomp.acomp a₂) ++ (cond f
                                                [instr.jmpless n]
                                                [instr.jmpge n])

#eval ( bcomp (bexp.And (bexp.Less (aexp.V "x") (aexp.V "y")) (bexp.Bc
    true)) false 3)
-- https://leanprover.zulipchat.com/#narrow/stream/113489-new-members/
    topic/Pattern.20matching.20with.20dependent.20types/near/211204742

@[simp] def bcomp_len  : bexp → bool → ℤ → ℕ
| (bexp.Bc v) f n := cond (v = f) 1 0
| (bexp.Not b) f n := bcomp_len b (!f) n
| (bexp.And b₁ b₂) f n :=
  let cb₂ := bcomp_len b₂ f n in
  let m := cond f (↑cb₂) (↑cb₂ + n) in
  let cb₁ := bcomp_len b₁ false m in
  cb₁ + cb₂
| (bexp.Less a₁ a₂) f n :=
  (acomp.acomp a₁).length + (acomp.acomp a₂).length + 1

lemma bcomp_len_eq_bcomp_len_n { b f n n' } : (bcomp_len b f n) = (
    bcomp_len b f n') :=
begin
  induction' b,
  all_goals {
    simp,
  },
  {
    exact ih,
  },
  {
    rw @ih_b_1 f n n',
    rw @ih_b ff (cond f ↑(bcomp_len b_1 f n') (↑(bcomp_len b_1 f n') + n))

                (cond f ↑(bcomp_len b_1 f n') (↑(bcomp_len b_1 f n') + n')
   ),
  },
end

lemma bcomp_len_eq_length_bcomp {b f n } : (bcomp_len b f n) = (bcomp b f
     n).length :=
begin
  induction' b,
  {
```

```
    show bcomp_len (bexp.Bc b) f n = list.length (bcomp (bexp.Bc b) f n),
    simp,
    cases' b,
    all_goals {
      cases' f,
      all_goals {
        simp,
      },
    },
  },
  {
    show bcomp_len (bexp.Not b) f n = list.length (bcomp (bexp.Not b) f n)
    ,
    simp,
    exact @ih (!f) n,
  },
  {
    show bcomp_len (bexp.And b b_1) f n = list.length (bcomp (bexp.And b
    b_1) f n),
    simp,
    cases' f ,
    {
      simp,
      rw ← @ih_b_1 ff n,
      rw ← @ih_b ff (↑(bcomp_len b_1 ff n) + n),
    },
    {
      simp,
      rw ← @ih_b_1 tt n,
      rw ← @ih_b ff ↑(bcomp_len b_1 tt n),
    },
  },
  {
    show bcomp_len (bexp.Less x x_1) f n = list.length (bcomp (bexp.Less
    x x_1) f n),
    simp,
    cases' f,
    all_goals {
      simp,
      refl,
    },
  },
end
```

```
-- The length of a jump doesn't change how many instructions are
    generated by bcomp
lemma length_bcomp_eq_length_bcomp {b f n}  (n') : (bcomp b f n).length =
     (bcomp b f n').length :=
begin
  rw ← bcomp_len_eq_length_bcomp,
  rw ← bcomp_len_eq_length_bcomp,
  exact bcomp_len_eq_bcomp_len_n,
end

-- Useful for normalizing `length (bcomp b f n)` for all values of n
@[simp] lemma length_bcomp_eq_length_bcomp_zero {b f n} :
              (bcomp b f n).length = (bcomp b f 0).length :=
by apply length_bcomp_eq_length_bcomp



lemma correct { b f s stk } {n : ℤ} (hpos : 0 ≤ n) :
              (bcomp b f n) ⊢
                   (0, s, stk) ⇒*
                   (↑(bcomp b f n).length + (cond (bval b s = f) n 0),
                    s, stk) :=
begin
  simp,
  induction' b,
  case bexp.Bc {
    show (bcomp (bexp.Bc b) f n) ⊢
              (0, s, stk) ⇒*
              (↑(list.length (bcomp (bexp.Bc b) f 0)) +
                 ite (bval (bexp.Bc b) s = f)
                     n
                     0,
               s,
               stk),

    cases' (em (b = f)),
    {
      simp [h],
      show [instr.jmp n] ⊢ (0, s, stk) ⇒* (1 + n, s, stk),
      exact star.single (@exec1.exec1 [instr.jmp n] s stk (_) _ _
                         (by simp)
                         (@iexec.jmp 0 s stk n)),
    },
    {
```

```
    simp,
    show (ite (b = f) [instr.jmp n] list.nil) ⊢
                      (0, s, stk) ⇒*
                      (↑((ite (b = f)
                              [instr.jmp 0]
                              list.nil).length) +
                          ite (b = f)
                              n
                              0,
                      s,
                      stk),
    rw ← ite_not,
    simp [h],
  }
},
case bexp.Not {
  simp,
  have ih' : (bcomp b (!f) n) ⊢
             (0, s, stk) ⇒*
             (↑(bcomp b (!f) 0).length +
               ite (bval b s = !f)
                   n
                   0,
              s,
              stk)
       := @ih (!f) s stk n hpos,
  -- We know that hpos : 0 ≤ n, therefore n must be a ℕ
  cases' n,
  case int.of_nat {
    clear hpos,
    simp at ih',
    simp,
    cases f,
    repeat {
      simp,
      simp at ih',
      exact ih',
    },
  },
  case neg_succ_of_nat {
    simp at hpos, contradiction,
  },
},
case bexp.And {
```

```
show (bcomp (b.And b_1) f n) ⊢
      (0, s, stk) ⇒*
      (↑(bcomp (b.And b_1) f 0).length +
               ite (bval (b.And b_1) s = f)
                   n
                   0,
       s,
       stk),
  -- We know that hpos : 0 ≤ n, therefore n must be a ℕ
  cases' n,
  {
    have ih_b' : (bcomp b ff (↑((bcomp b_1 f ↑n).length) + cond f 0 ↑n))
  ⊢
                  (0, s, stk) ⇒*
                  (↑(bcomp b ff 0).length +
                    ite (bval b s = ff)
                        (↑(bcomp b_1 f ↑n).length + cond f 0 ↑n)
                        0,
                   s,
                   stk)
    := @ih_b ff s stk ((bcomp b_1 f n).length + (cond f 0 n))
    (begin
      -- show  0 ≤ ↑((bcomp b_1 f ↑n).length) + cond f 0 ↑n,
      cases' f,
      {
        simp,
        norm_cast,
        simp,
      },
      {simp,}
    end),
    have ih_b_1' := @ih_b_1 f s stk n hpos,
    clear ih_b,
    clear ih_b_1,
    simp [bcomp] at *,
    -- There are many terms on the form (list.length (bcomp b ff n))
  for
    -- different values of n. The value of those terms do not depend on
   n so
    -- we can simplify with ‘length_bcomp_eq_length_bcomp_zero‘ and
  call them
    -- with the same name.
    generalize h_lenb : (list.length (bcomp b ff 0)) = lenb,
    generalize h_lenb' : (list.length (bcomp b_1 f 0)) = lenb_1,
```

```
  have ih_b'' := exec_append_right (bcomp b_1 f ↑n) ih_b',
  clear ih_b',
  rw cond_add f lenb_1 n,
  rw [h_lenb, h_lenb'] at *,
  apply star.trans ih_b'',
  cases' bool.or_intro (bval b s) with hbval_eq_tt hbval_eq_ff,
  {
    simp [hbval_eq_tt],
    cases' f,
    {
      simp [hbval_eq_tt],
      have ih_b_1'' := exec_append_left (bcomp b ff (↑lenb_1 + ↑n))
ih_b_1',
      simp [hbval_eq_tt] at ih_b_1'',
      rw [h_lenb', h_lenb] at *,
      nth_rewrite 0 [add_assoc],
      nth_rewrite 1 [add_comm],
      exact ih_b_1'',
    },
    {
      simp [hbval_eq_tt],
      have ih_b_1'' := exec_append_left (bcomp b ff (↑lenb_1 )) ih_b_1
',
      simp [hbval_eq_tt] at ih_b_1'',
      rw [h_lenb', h_lenb] at *,
      nth_rewrite 0 [add_assoc],
      nth_rewrite 0 [add_comm],
      apply ih_b_1'',
    }
  },
  {
    simp [hbval_eq_ff],
    have ih_b_1'' := exec_append_left (bcomp b ff (↑lenb_1 + cond f 0
 ↑n)) ih_b_1',
    simp at ih_b_1'',
    rw [h_lenb, h_lenb'] at ih_b_1'',
    cases' f,
    {
      simp,
      nth_rewrite 0 add_assoc,
    },
    {
      simp,
    }
```

```
    }
  },
  {
    exfalso, simp at hpos, assumption,
  }
},
{
  cases' n,
  {
    simp, clear hpos,
    have hx := exec_append_right (acomp.acomp x_1) (@acomp.correct x s
 stk),
    have hx_1 := exec_append_left (acomp.acomp x) (@acomp.correct x_1 s
  ((aval x s) :: stk)),
    simp at hx_1,
    have tmp := exec_append_right (cond f [instr.jmpless ↑n] [instr.
 jmpge ↑n]) (star.trans hx hx_1),
    rw list.append_assoc at tmp,
    apply star.trans tmp,
    rw add_assoc,
    nth_rewrite 1 add_comm,
    apply exec_append_left (acomp.acomp x),
    rw ← (int.zero_add ↑(list.length (acomp.acomp x_1))),
    nth_rewrite 2 [add_comm],
    nth_rewrite 1 [int.zero_add],
    rw add_assoc,
    nth_rewrite 2 [add_comm],
    rw ← add_assoc,
    apply exec_append_left (acomp.acomp x_1),
    cases' f,
    {
      simp,
      apply star.single,
      apply @exec1.exec1 _ _ _ _ (instr.jmpge n) _,
      { simp },
      {
        cases' em (aval x_1 s ≤ aval x s),
        {
          simp [h],
          {
            apply iexec.jmpge_true,
            { simp, exact h },
          },
        },
```

```
          {
            simp [h],
            {
              apply iexec.jmpge_false,
              { simp, simp at h, exact h },
            },
          }
        }
      },
      {
        simp,
        apply star.single,
        apply @exec1.exec1 _ _ _ _ (instr.jmpless n) _,
        { simp },
        {
          cases' em (aval x_1 s ≤ aval x s),
          {
            simp [le_iff_not_lt h],
            {
              apply iexec.jmpless_false,
              { simp, exact h },
            },
          },
          {
            simp [not_le_iff_lt h],
            {
              apply iexec.jmpless_true,
              {exact not_le_iff_lt h},
            },
          }
        }
      }
    },
    {
      exfalso, simp at hpos, exact hpos,
    }
  },
end

end bcomp

@[simp] def ccomp : stmt → program
| (stmt.skip)        := []
| (stmt.assign x a)  := (acomp.acomp a) ++ [instr.store x]
```

```
| (stmt.seq a b)     := (ccomp a) ++ (ccomp b)
| (stmt.ite b c₁ c₂) :=
  let cc₁ := ccomp c₁ in
  let cc₂ := ccomp c₂ in
  let cb  := bcomp.bcomp b ff (( list.length cc₁ ) + 1)
  in cb ++ cc₁ ++ [instr.jmp (list.length cc₂)] ++ cc₂
| (stmt.while b c) :=
  let cc           := ccomp c in
  let cb           := bcomp.bcomp b ff (list.length cc + 1)
  in cb ++ cc ++ [instr.jmp (- (list.length cb + list.length cc + 1) )]

#eval ccomp (stmt.ite (bexp.Less (aexp.V "u") (aexp.N 1)) (stmt.assign "u
    " (aexp.Plus (aexp.V "u") (aexp.N 1))) (stmt.assign "v" (aexp.V "u")))

lemma coe_bool_implies_eq_tt {b : bool} : ((↑b) : Prop) → b = tt :=
begin
{
  intro hb,
  cases' b,
  { simp, simp at hb, assumption, },
  { refl, }
}
end

lemma coe_not_bool_implies_eq_ff {b : bool} : ((¬ ↑b) : Prop) → b = ff
    :=
begin
{
  intro hb,
  cases' b,
  { simp, },
  {simp at hb, exfalso, assumption,   }
}
end

theorem correct {c s t stk} :
  (c, s) ⟶ t → (ccomp c) ⊢
                (0, s, stk) ⇒*
                (↑(ccomp c).length, t, stk) :=
begin
  intro hbig,
  induction' hbig,
  case big_step.skip {
    apply star.refl,
```

```
},
case big_step.assign {
  simp,
  have h_exec_a : (acomp.acomp a) ⊢ (0, t, stk) ⇒*
                                    (↑(acomp.acomp a).length,
                                     t,
                                     aval a t :: stk)
       := @acomp.correct a t stk,
  have h_exec_store :[instr.store x] ⊢ (↑0, t, aval a t :: stk) ⇒*
                                        (1, t{x ↦ aval a t}, stk)
  :=
    star.single (@exec1.exec1 [instr.store x]
                              t
                              (aval a t :: stk)
                              (1, t{x ↦ aval a t}, stk) _  0
                              (by simp )
                              (iexec.store x)),
  have h_exec_a' : (acomp.acomp a ++ [instr.store x]) ⊢
                   (0, t, stk) ⇒*
                   (↑((acomp.acomp a).length),
                    t,
                    aval a t :: stk)
  := exec_append_right [instr.store x] h_exec_a,
  have h_exec_store' : (acomp.acomp a ++ [instr.store x]) ⊢
                       (↑0 + ↑((acomp.acomp a).length),
                        t,
                        aval a t :: stk) ⇒*
                       (1 + ↑((acomp.acomp a).length), t{x ↦ aval a t
 }, stk)
  := exec_append_left (acomp.acomp a) h_exec_store,
  simp at h_exec_store',
  rw add_comm at h_exec_store',
  exact star.trans h_exec_a' h_exec_store',
},
case big_step.seq {
  simp,
  have ih :(ccomp S ++ ccomp T) ⊢ (0, s, stk) ⇒* (↑((ccomp S).length),
   t, stk)
       := exec_append_right (ccomp T) (@ih_hbig stk),
  have ih1 : (ccomp S ++ ccomp T) ⊢
             (0 + ↑(ccomp S).length, t, stk) ⇒*
             (↑(ccomp T).length + ↑(ccomp S).length,
              t_1, stk)
        := exec_append_left (ccomp S) (@ih_hbig_1 stk),
```

```
  simp at ih1,
  rw add_comm at ih1,
  exact star.trans ih ih1,
},
case big_step.ite_true {
  show (ccomp (stmt.ite b S T)) ⊢
       (0, s, stk) ⇒*
       (↑((ccomp (stmt.ite b S T)).length),
        t, stk),
  simp,
  have hS : (ccomp S) ⊢ (0, s, stk) ⇒* (↑((ccomp S).length), t, stk)
       := @ih stk,
  have hbool : (bcomp.bcomp b ff (↑((ccomp S).length) + 1)) ⊢
               (0, s, stk) ⇒*
               (↑((bcomp.bcomp b ff 0).length),
                s,
                stk)
   :=
   begin
      have tmp := @bcomp.correct b ff s stk ((ccomp S).length + 1)
                        (list.zero_le_length_add_one),
      have hcond' : bval b s = tt
          := coe_bool_implies_eq_tt hcond,
      simp [hcond, hcond'] at tmp,
      exact tmp,
   end,
  have hexec_bool : (bcomp.bcomp b ff
                             (1 + ↑((ccomp S).length)) ++
                              (ccomp S ++
                               instr.jmp ↑((ccomp T).length) :: ccomp T))
  ⊢
               (0, s, stk) ⇒*
               (↑((bcomp.bcomp b ff 0).length), s, stk)
  :=
  begin
    have tmp := exec_append_right (instr.jmp ↑(list.length (ccomp T))
                                   :: ccomp T)
                                  (exec_append_right (ccomp S) hbool),
    rw list.append_assoc at tmp,
    abel at tmp,
    exact tmp,
  end,
  have hexec_S : (bcomp.bcomp b ff (1 + ↑(ccomp S).length) ++
```

```
                (ccomp S ++ instr.jmp ↑((ccomp T).length) :: ccomp T))
⊢
               (↑(bcomp.bcomp b ff 0).length, s, stk) ⇒*
               (↑(ccomp S).length + ↑(bcomp.bcomp b ff 0).length,
                t, stk)

:=
begin
  have tmp := exec_append_right (instr.jmp ↑(list.length (ccomp T)) ::
 ccomp T)
                (exec_append_left (bcomp.bcomp b ff (↑(list.length (
ccomp S)) + 1))
                                  (@ih stk)),
  simp at tmp,
  abel at tmp,
  exact tmp,
end,
have hjmp : [instr.jmp ↑((ccomp T).length)] ⊢
            (↑0, t, stk) ⇒*
            (1 + ↑((ccomp T).length), t, stk)
:= star.single
     (@exec1.exec1 [instr.jmp (ccomp T).length] t stk
                   (1 + (ccomp T).length, t, stk)
                   (instr.jmp (ccomp T).length)
                   0 (by simp)
                   (iexec.jmp (ccomp T).length)),

have hexec_jmp : (bcomp.bcomp b ff (1 + ↑((ccomp S).length)) ++
                   (ccomp S ++ instr.jmp ↑((ccomp T).length) :: ccomp
 T))
                 ⊢
                 (↑((ccomp S).length) + ↑((bcomp.bcomp b ff 0).
length),
                  t, stk)
                 ⇒*
                 (1 + (↑(ccomp S).length +
                   (↑(ccomp T).length +
                    ↑(bcomp.bcomp b ff 0).length)),
                  t, stk)
:=
begin
  have tmp :=
       exec_append_left (bcomp.bcomp b ff
```

```
                                        (↑(list.length (ccomp S)) + 1) ++

                                          (ccomp S))
                                        (exec_append_right (ccomp T)
  hjmp ),
    simp at tmp,
    abel at tmp,
    exact tmp,
  end,
  abel,
  exact star.trans (star.trans hexec_bool hexec_S) hexec_jmp,
},
case big_step.ite_false {
  simp,
  have hbool : (bcomp.bcomp b ff (1 + ↑((ccomp S).length))) ⊢
               (0, s, stk) ⇒*
               (1 + (↑((ccomp S).length) + ↑((bcomp.bcomp b ff 0).
  length)),
                s, stk)
  :=
  begin
    have tmp := @bcomp.correct b ff s stk ((ccomp S).length + 1)
                               (list.zero_le_length_add_one),
    have hcond' := coe_not_bool_implies_eq_ff hcond,
    simp [hcond, hcond'] at tmp,
    abel at tmp,
    exact tmp,
  end,
  have hexec_bool : (bcomp.bcomp b ff (1 + ↑((ccomp S).length)) ++
                       (ccomp S ++ instr.jmp ↑((ccomp T).length) ::
  ccomp T))
                    ⊢
                    (0, s, stk) ⇒*
                    (1 + (↑((ccomp S).length) + ↑((bcomp.bcomp b ff 0).
  length)),
                    s, stk)
  :=
  begin
    have tmp := exec_append_right (instr.jmp ↑(list.length (ccomp T)) ::
   ccomp T)
                  (exec_append_right (ccomp S) hbool),
    rw list.append_assoc at tmp,
    exact tmp,
  end,
```

```
  have hS : (ccomp T) ⊢ (0, s, stk) ⇒*
                        (↑((ccomp T).length), t, stk)
      := @ih stk,
  have hexec_S : (bcomp.bcomp b ff (1 + ↑(ccomp S).length) ++
                  (ccomp S ++
                   instr.jmp ↑((ccomp T).length) :: ccomp T))
                 ⊢
                 (1 + (↑((ccomp S).length) + ↑((bcomp.bcomp b ff 0).
  length)),
                  s, stk)
                 ⇒*
                 (1 + (↑(ccomp S).length
                       + (↑(ccomp T).length + ↑(bcomp.bcomp b ff 0).
  length)),
                  t, stk)
  :=
  begin
    have tmp :=
         exec_append_left ((bcomp.bcomp b ff (↑(list.length (ccomp S)) +
   1))
                             ++ (ccomp S) ++ [instr.jmp ↑(list.length (
  ccomp T))])
                           hS,
    simp at tmp,
    abel at tmp,
    exact tmp,
  end,
  abel,
  exact star.trans hexec_bool hexec_S,
},
case big_step.while_true {
  rename hbig hbig_S,
  rename hbig_1 hbig_while_b,
  rename ih_hbig ih_exec_S,
  rename ih_hbig_1 ih_exec_while_b,
  simp [hcond, coe_bool_implies_eq_tt hcond] at ih_exec_while_b,
  simp,
  set n := (-(1 : ℤ) + (-↑(list.length (ccomp S))
                        + -↑(list.length (bcomp.bcomp b ff 0))))
      with hn,
  have hexec_b : (bcomp.bcomp b ff (↑(ccomp S).length + 1)) ⊢
                 (0, s, stk) ⇒*
                 (↑(bcomp.bcomp b ff 0).length,
                  s,
```

```
                stk)
     :=
begin
     have tmp := @bcomp.correct b ff s stk (↑(list.length (ccomp S)) +
 1)
                                  (list.zero_le_length_add_one),
     simp [hcond, coe_bool_implies_eq_tt hcond] at tmp,
     exact tmp,
end ,
have hexec_b' :
(bcomp.bcomp b ff (↑((ccomp S).length) + 1) ++
      (ccomp S ++ [instr.jmp n])) ⊢
        (0, s, stk) ⇒*
        (↑((bcomp.bcomp b ff 0).length), s, stk)
:= exec_append_right ((ccomp S ++ [instr.jmp n]))
              hexec_b,
abel at hexec_b',
clear hexec_b,
have hexec_S : (bcomp.bcomp b ff (1 + ↑(ccomp S).length) ++
                (ccomp S ++ [instr.jmp n]))
                ⊢
                (↑(bcomp.bcomp b ff 0).length,
                 s,
                 stk) ⇒*
                (↑(ccomp S).length + ↑(bcomp.bcomp b ff 0).length,
                 t,
                 stk)
     := begin
        have tmp := exec_append_left (bcomp.bcomp b ff (↑(list.length
 (ccomp S)) + 1))
                     (exec_append_right [instr.jmp n]
                     (@ih_exec_S stk)),
        simp [hcond, coe_bool_implies_eq_tt hcond] at tmp,
        abel at tmp,
        exact tmp,
        end,
clear ih_exec_S,
have hexec_jmp : (bcomp.bcomp b ff (1 + ↑(ccomp S).length) ++
                  (ccomp S ++ [instr.jmp n]))
                  ⊢
                  (↑(ccomp S).length + ↑(bcomp.bcomp b ff 0).length,
                   t,
                   stk) ⇒*
                  (0, t, stk)
```

```
                    :=
        begin
        have tmp := star.single
                        (@exec1.exec1 [instr.jmp n]
                                      t stk _
                                      (instr.jmp n)
                                      0 (by simp)
                                      (iexec.jmp n)),
        have tmp' := exec_append_left (bcomp.bcomp b ff (↑(list.length (
    ccomp S)) + 1) ++ ccomp S) tmp,
        simp at tmp',
        abel at tmp',
        exact tmp'
        end,
    have hexec_b_S := star.trans hexec_b' hexec_S,
    clear hexec_b' hexec_S,
    abel,
    have h_exec_b_S_jmp := star.trans hexec_b_S hexec_jmp,
    abel at ih_exec_while_b,
    exact star.trans h_exec_b_S_jmp (@ih_exec_while_b stk),
  },
  case big_step.while_false {
    simp,
    have hexec_b : (bcomp.bcomp b ff (↑((ccomp S).length) + 1)) ⊢
                   (0, t, stk) ⇒*
                   (↑(bcomp.bcomp b ff (↑((ccomp S).length) + 1)).length
     +
                       cond (to_bool (bval b t = ff))
                            (↑((ccomp S).length) + 1)
                            0,
                    t,
                    stk)
        := @bcomp.correct b ff t stk (↑(list.length (ccomp S)) + 1)
                                     (list.zero_le_length_add_one),
    simp [hcond, coe_not_bool_implies_eq_ff hcond] at hexec_b,
    let n : ℤ := -1 + (-↑(list.length (ccomp S)) + -↑(list.length (bcomp.
    bcomp b ff 0))),
    exact exec_append_right (ccomp S ++ [instr.jmp n])
                            hexec_b,
  },
end
```