Bachelor of Computer Science

Bachelor thesis

Logical Verification of a Compiler in Lean

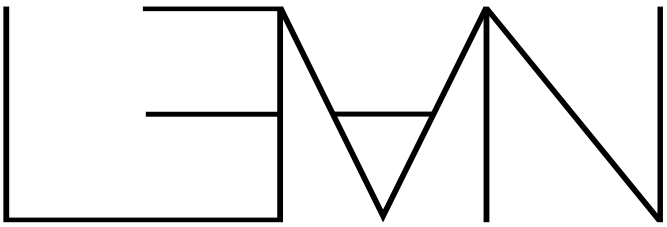
by

Martynas Rimkevičius

September 12, 2023

Supervisor: prof. dr. Jasmin Christian Blanchette

Second reader: Anne Baanen



Department of Computer Science
Faculty of Sciences

VU UNIVERSITY AMSTERDAM

Abstract

Lean is an interactive theorem prover that can be used to formalise complex pen–and–paper proofs, and guarantee their correctness. In this thesis, I port formalized compiler proofs from Isabelle/HOL, a more automated theorem prover, to Lean 3. The compiler construction and verification are discussed in *Concrete Semantics* [1]. Comparisons are made between the proofs in Isabelle and Lean, showing how the proofs look with less automation.

Title: Logical Verification of a Compiler in Lean
Author: Martynas Rimkevičius, m.rimkevicius@student.vu.nl, 2708302
Supervisor: prof. dr. Jasmin Christian Blanchette
Second reader: Anne Baanen
Date: September 12, 2023

Department of Computer Science
VU University Amsterdam
de Boelelaan 1111, 1081 HV Amsterdam
`http://www.cs.vu.nl/`

Contents

1 Introduction

Interactive theorem provers are used to develop formalizations of logical and mathematical concepts. Lean is an interactive theorem prover designed to formalize mathematical and logical proofs, ensuring their correctness. In contrast, informal pen-and-paper proofs lack this guarantee and pose a risk of introducing unnoticed errors that can undermine an entire research paper. This risk is especially prevalent when mathematicians integrate and combine proofs from different sources. However, in Lean, the presence of errors is eliminated as the program highlights any mistakes and handles lengthy and complex proofs.

This thesis contributes to the formalized specification and correctness proof of a compiler in Lean 3, and showcases the intricacies of compiler correctness proofs. Operational semantics proofs on paper have been around since the concept of compilation – translating an abstraction to machine code and executing it correctly – and it has been formalized in most other interactive theorem provers. However, the details of these proofs, especially the ones that I am translating from *Concrete Semantics* by Nipkow and Klein [1] which are written in Isabelle/HOL, are not clear enough because they are highly automated. Lean, compared to Isabelle/HOL, is less automated and offers more control to the user over what the system itself is doing. This is why understanding the proofs that utilize automation in Isabelle is harder than understanding explicitly written out proofs in Lean. One such comparison could be seen in *The Hitchhiker's Guide to Logical Verification* by Baanen *et al.* [2] which shows Lean verification of a simple imperative language that has a similar structure to Isabelle verification covered in *Concrete Semantics*. However, *The Hitchhiker's Guide* does not cover compiler verification, which is why I port the verification infrastructure covered in *Concrete Semantics* from Isabelle to Lean. The Lean code of the complete proofs (without the use of `sorry`) is available on GitHub[1].

In this thesis, I will give a brief introduction to the Lean theorem prover. Then I will show the creation of the stack machine, first its instructions and machine executions with their verification, followed by the compilation expressions and their correctness proofs. The related work discusses other approaches to operational verification, and I outline ideas for future work in the conclusion.

[1] `https://github.com/MRimk/LogicalCompilerVerification`

2 The Lean proof assistant

In this section, I go over the most relevant to the paper high–level Lean topics. The topics include type theory, inductive types and proof tactics. I also compare them to Isabelle/HOL so that later proof comparisons would be as clear as possible.

The Lean proof assistant is based on a version of dependent type theory known as *Calculus of Constructions* with inductive types [3], but it also has simple (i.e. non–dependent type theory), which I both introduce.

Simple type theory in Lean is used to associate every expression with a type. This means that there is no room for different interpretations. For example, each expression, variable, function, or constant will have a type, for instance – an integer. What makes the Lean type theory powerful is that, from the existing types, we can build new ones. For example, if α and β are types, the $\alpha \to \beta$ is a function type out of them, and $\alpha \times \beta$ is a product type.

Lean extends simple type theory by treating the types themselves as objects. This means that types have to have a type, even the basic ones such as `nat` or `bool`, have a type `Type`. In proofs this allows us to operate on types the same way we operate on objects.

To define an object or a function of a certain type in Lean, `def` is used to do that. Objects are used in proofs, and proofs are also the ones that verify their correctness. The general form of a definition is as follows: `def foo : α := bar` (the type can sometimes be inferred, but the best practice is to specify it).

Lean also has inductive types. Inductive types in Lean allow us to create types from scratch. A lot of definitions in Lean are made using inductive types, which allow us to use induction in the proofs or to define a recursive type. Inductive types are built from separate constructors, which are used to construct new objects of such type using previously defined values. The inductive type creates abstraction over the constructors and makes it easier to reason about each object collectively. This abstraction also allows for recursion in the types themselves, where the input becomes the type itself. This is because each constructor acts like a definition of that type; therefore, it can have its type passed as an argument. An example of inductive type definition in Lean is as follows:

```
inductive list (T : Type) : Type :=
| nil {} : list T
| cons   : T → list T → list T
```

In Isabelle/HOL types are used in a similar manner. They are defined using the datatype keyword, and they are already built as inductive types with possible several constructors. As an example, I show the same inductive type list defined in Isabelle:

```
datatype (set: 'a) list =
    Nil  ("[]")
  | Cons (hd: 'a) (tl: "'a list")  (infixr "#" 65)
```

The next important part is proof tactics in Lean. The tactics are recognized by the keywords `begin ... end` and `by`. The tactic proofs are backward reasoning proofs, meaning the goal is broken down into smaller sections that are proven separately to construct a full proof. The advantage of these proofs is that the goal can be very complex, but by splitting it into smaller problems it makes it possible to achieve the goal. There is the opposite proof strategy – structured proofs, which are proofs that begin with hypotheses and assumptions and work towards the goal. They are mainly used for forward reasoning. In practice, both of these strategies are used together to split the goal into smaller parts and use hypotheses to prove subgoals.

Tactics in Lean can be partially automated, which is usually helpful to use to avoid long and tedious proofs. Some of the most used automated tactics in my case are `simp`, which applies several trivial rewriting steps (can be by using a given hypothesis) that simplify the goal; `linarith` that combines existing (in)equalities to find a contradiction between them; `ring_nf`, which solves equations that are in a commutative (semi)ring structure; `finish` that simplifies the goal, eliminates quantifiers, and looks for contradiction; `exact`, which applies the hypothesis that is the same as the goal. Since most of these tactics help prove trivial goals or complete a set of trivial operations, they do not interfere with the clarity of the proof.

In Isabelle/HOL, there are more automated proofs. There is *simp* which is similar to Lean's, but there is also *auto*, which applies simplification with a small amount of proof search (it is hard to say what exactly *auto* performs due to its heuristic nature) [4]. Some of the automated proofs that are used more frequently are *fastforce*, which tries harder at simplifying than auto, and it works on the first subgoal only where it either succeeds or fails, and *blast* which is commonly used to prove complex logical goals. The automated proofs are said to be complete if they can prove all true formulas, but there is no complete proof method for Isabelle, which is why there are different automated proof methods.

3 Stack machine instructions

To set up a stack machine, I begin by defining the instruction set architecture and semantics. The definitions in this section are heavily inspired by Isabelle/HOL definitions in *Concrete Semantics* [1], yet they are expanded upon and adapted to suit Lean's structure. Moreover, the stack machine is designed to fit a minimalist imperative language called WHILE defined in *The Hitchhiker's Guide* [2].

I start by defining the basis of our stack machine – variables, values, and the stack. Since this is a simple stack machine, the supported value type is an integer, and it is also possible to store variables whose names are of the string type. In this case, the stack is created to save the values, and it is defined as a list of integers:

```
def vname := string
def stack := list ℤ
```

The way the variables are connected to the values is through a state function, which is essentially the machine's memory. State is a mapping of `vname` $\rightarrow$ $\mathbb{Z}$ such that the value can be assigned and updated. I have used the state definition from *The Hitchhiker's Guide*, where the state is defined as a mapping of `string` $\rightarrow$ $\mathbb{N}$ but since the variables are integers and the program counter is going to be an integer, the definition was adapted accordingly. One of the key operations on the state that makes it work as a mapping is the update operation:

```
def state.update (name : vname) (val : ℤ) (s : state) : state :=
λname', if name' = name then val else s name'
```

This operation either assigns a new value to the existing variable or creates a new variable and does it each time without additional operations being required. This update is possible because it has several proven lemmas, which are imported from *The Hitchhiker's Guide*, that can be used for further simplification in the application.

The instruction set is comprised of simple, assembly–level instructions to put the value in the stack, do arithmetic operations on the values, change the variable, and change the order of execution. The definition of instruction is of the previously–discussed inductive type, which makes each instruction a separate constructor.

```
inductive instr : Type
| LOADI : ℤ → instr
| LOAD : vname → instr
| ADD : instr
| SUB : instr
| MUL : instr
| DIV : instr
```

```
| STORE : vname → instr
| JMP : int → instr
| JMPLESS : int → instr
| JMPGE : int → instr
| NOP : instr
```

In the definition above, the instruction `LOADI` pushes the integer on the stack, `LOAD` pushes the value of the variable on the stack, `ADD`, `SUB`, `MUL`, `DIV` do corresponding arithmetic operations with the top two integer values of the stack. `STORE` instruction stores the stack's top value in memory under a given variable name. `JMP` jumps, or changes the program counter, by a given relative value. `JPLESS` and `JMPGE` compare the two top stack values and jump if the second one is less or if the second one is greater or equal accordingly.

In Isabelle, the definitions of machine instructions and their architecture are very similar because it does not differ as much when defining types and functions from Lean.

4 Machine executions

Stack machine instructions are executed in steps, which are transitions between machine configurations. Machine configuration is defined as follows:

```
def config := ℤ × state × stack
```

A configuration is a tuple of a program counter, the current state of memory, and the current state of the stack.

The machine step function `iexec` is a transition from a given configuration to another one by a specific step. The function applies the instruction to the stack and the state and, in the process, changes the program counter by one or as specified by the instruction. In Isabelle, `iexec` definition is very similar, the only difference is the use of the abbreviated form for `stk.tail.tail` and `list.cons` symbol is `#` instead of `::` like in Lean. The specification of `iexec` in Lean is as follows:

```
def iexec : instr → config → config
| (LOADI n) (i, s, stk) := (i + 1, s, n :: stk)
| (LOAD v) (i, s, stk) := (i + 1, s, s v :: stk)
| ADD (i, s, stk) :=
    (i + 1, s, (stk.tail.head + stk.head) :: stk.tail.tail)
| SUB (i, s, stk) :=
    (i + 1, s, (stk.tail.head - stk.head) :: stk.tail.tail)
| MUL (i, s, stk) :=
    (i + 1, s, (stk.tail.head * stk.head) :: stk.tail.tail)
| DIV (i, s, stk) :=
    (i + 1, s, (stk.tail.head / stk.head) :: stk.tail.tail)
| (STORE v) (i, s, stk) := (i + 1, s{v ↦ (stk.head)}, stk.tail)
| (JMP n) (i, s, stk) := (i + 1 + n, s, stk)
| (JMPLESS n) (i, s, stk) :=
  if (stk.tail.head) < (stk.head)
  then (i + 1 + n, s, stk.tail.tail)
  else (i + 1, s, stk.tail.tail)
| (JMPGE n) (i, s, stk) :=
  if (stk.tail.head) ≥ (stk.head)
  then (i + 1 + n, s, stk.tail.tail)
  else (i + 1, s, stk.tail.tail)
| NOP (i, s, stk) := (i, s, stk)
```

Having this step defined is not enough to check whether it is a valid execution, since it just applies the instruction to the configuration. For that, I define a boolean check

function that checks whether, given a list of instructions (a program) and two configurations, there can be a single instruction step taken at the first configuration's program counter location:

```
def exec1  (li : list instr) (c : config)  (c' : config) : Prop :=
( c' = iexec (nth li c.fst) c ∧ 0 ≤ c.fst ∧ c.fst < li.length)
```

In Isabelle, `exec1` function is defined as a predicate where program `P` semantically entails a step from configuration `c` to configuration `c'`. The main difference is that an existential quantifier is used to describe the configuration `c`. The other syntactic difference is that there is an infix notation defined for later readability. The definition in Isabelle is as follows:

```
definition
  exec1 :: "instr list ⇒ config ⇒ config ⇒ bool"
     ("(_/ ⊢ (_ →/ _))" [59,0,59] 60)
where
  "P ⊢ c → c' =
  (∃i s stk. c = (i,s,stk) ∧ c' = iexec(P!!i) (i,s,stk) ∧
  0 ≤ i ∧ i < size P)"
```

At the same time, `exec1` checks whether the first configuration's program counter is valid, or is within the bounds of the instruction list. This check is important because the program counter is an integer, therefore could be less than zero, and it could also perform a wrong step if it is out of bounds. Since the program counter is an integer, I create a new definition of the nth item in the list, since in mathlib (the Lean mathematical library) [5] there is only a definition of the nth being a natural number. This introduction is where the instruction `NOP` comes in. It is defined such that getting an instruction would not be an option of the instruction object, but the instruction object itself. However, the option is required to define a case when the list is empty and thus returns option none. In my case, since it is very specific, the `nth` function works only on instruction lists:

```
def nth : list instr → ℤ → instr
| (a :: l) n := if (n = 0)
  then a
  else nth l (n - 1)
| list.nil n := NOP
```

After `exec1` – checking whether there is a step between two configurations – the definition can be expanded to a check whether there exists a set of steps in a list of instructions between two configurations – `exec`. The `exec1` can also be understood as a relation between two configurations; therefore `exec` is constructed with a reflexive transitive closure `star` of relation `exec1`.

To continue building on top of the machine executions, they have to be functionally verified. This verification is done for singular and multiple–step executions, and mostly concerns expanding the instruction list forward or backward. It is important to be able to

append or prepend new instructions and know how it will change the execution, because if a command of multiple instructions is created to be executed, and that command needs to be concatenated to another command, both of these commands' lists are concatenated.

I start by translating `exec1_appendR` and `exec_appendR` lemmas from Isabelle:

```
lemma exec1_appendR {li c c' li'} (h: exec1 li c c'):
exec1 (li ++ li') c c'

lemma exec_appendR {li c c' li'} (h: exec li c c'):
exec (li ++ li') c c'
```

These lemmas show that if a new instruction list is appended to the right of the current list, the execution does not change because the program counter is pointing to the same initial list. To prove these lemmas, I create and prove lemma `nth_append`, which applies `nth` to the correct list according to the size of `i`:

```
lemma nth_append {l1 l2 : list instr} {i : ℤ}
(h_nneg : 0 ≤ i) :
nth (l1 ++ l2) i = (
  if i < int.of_nat (list.length l1)
  then nth l1 i
  else nth l2 (i - list.length l1))
```

In Isabelle, the proof of `nth_append` applies the induction on the first list length and uses *auto simp*, which automatically proves this lemma. In Lean, however, the proof is based on induction but had to be manually split into cases to eliminate if–then–else structures and operate on existing inequalities. The proof is straightforward since it covers simple cases, but it requires some writing to get it done with the manual creation of hypotheses and rewrites.

Using `nth_append` lemma, the proof of `exec1_appendR` is not easy to implement but is quite clear – show that the program counter is always in–bounds of the first list. To do so the empty list case hypotheses simplify to be `false` (from which any goal can be proven) since the program counter has to be in–between 0 and the length of the list, which is 0. The non–empty list case's main idea is to use the hypothesis `h` from which we know that according to `exec1` definition: `i < li.length`.

Lemma `exec_appendR` is also based on induction (this time `induction'` tactic, which generalizes induction hypothesis by default – useful in `star` proofs [6]), just this time on `star` induction, which is the same as Isabelle, but I define the steps myself. In Isabelle, there is an automatic proof used – *fastforce* – which is based on *star.step* (reducing reflexive transitive closure to a singular step) and using `exec1_appendR`. In Lean, using the singular `star.single` (equivalent to Isabelle's *star.step*) seems to not be enough. Therefore I use `star.tail` lemma, which splits the goal into a `star` step to a transitive configuration, and from that configuration a singular step to the end configuration.

In the same way that there are two lemmas for appending the instruction list with a new list and preserving the execution order, there are two lemmas that show how the order is preserved when the lists are prepended (appended to the left). When the list is

added to the left of the current list, the program counter has to be shifted by the length of the added list in the starting configuration and the ending configuration.

```
lemma exec1_appendL {i i' :ℤ} {li li' s stk s' stk'}
(h_li : exec1 li (i, s, stk) (i', s', stk')) :
exec1 (li' ++ li) ((list.length li') + i, s, stk)
    ((list.length li') + i', s', stk')

lemma exec_appendL {i i' :ℤ} (li li' i s stk i' s' stk')
(h_single : exec li (i, s, stk) (i', s', stk')) :
exec (li' ++ li) (list.length li' + i, s, stk)
    (list.length li' + i', s', stk')
```

To prove `exec1_appendL` shown above, I use the `nth_append` lemma and the definition of `exec1`. It also requires to use `by_cases` tactic, which is necessary to eliminate `ite (i < 0)` (if–then–else) that appears from `nth_append`. Additionally, the lemma `iexec_shift` that is defined in *Concrete Semantics* is necessary to simplify the left append, and that lemma states that `iexec` operation is not affected by the shift:

```
lemma iexec_shift {instr n i' s' stk' i s stk}:
((n+i',s',stk') = iexec instr (n+i, s, stk)) = ((i',s',stk') =
    iexec instr (i,s,stk))
```

This lemma is proven using `cases` tactic on `instr`, thus proving that this is true for all instructions. All cases could be simplified into two – shift when there is *JMP* instruction (e.g. `JMPGE`), and shift when there is not (e.g. arithmetic instructions). Both cases are proven the same way, except when there is a *JMP* instruction the equality of program counters contains the size of the jump, so I use two supportive lemmas to reuse the same proofs. Both supportive lemmas state a similar thing – if the program counter changes a certain way, when shift is applied, it will change the same way, and they have similar mathematical simplification using a `iff.intro` tactic since one side simplifies because of the other being true and vice versa.

Out of the intuitive lemmas where the new instruction list is appended to the left or right of the execution, there are three more lemmas defined in *Concrete Semantics*. These lemmas are necessary for later use since they allow reasoning about `exec` in a more simple manner than always expanding its definition to `star`.

The most used lemma in the later stages of this project is `exec_append_trans` lemma, which shows that the separate executions of two lists can become one execution if the first execution has the same end configuration as the second execution's start configuration:

```
lemma exec_append_trans {li' li s stk  s' stk' s'' stk''}
{i i' j'' i'': ℤ}
(h_li: exec li (0, s, stk) (i', s', stk'))
(h_i': int.of_nat (list.length li) <= i' )
(h_li': exec li' (i' - list.length li, s', stk') (i'', s'', stk''))
(h_j'': j'' = list.length li + i''):
exec (li ++ li') (0, s, stk) (j'', s'', stk'')
```

The lemma statement contains several hypotheses, which together allow me to build transitive steps for two (or more) instruction lists concatenated together in later proofs, thus breaking down the goal into smaller subgoals. In order to prove this lemma, I use `star.trans` lemma, which simplifies easily using the initial hypotheses and previously defined `exec_appendR`. The Isabelle proof is automatic (uses *metis*) but it does specify that it uses *star_trans* rule.

5 Expressions

I define the compiler in three separate steps, as it is done in the *Concrete Semantics*, beginning with arithmetic expressions, then boolean expressions, and then commands. Proofs of the correctness of these expressions follow in the next section.

5.1 Arithmetic expressions

Arithmetic expressions are similar to instruction definitions because each expression corresponds to an instruction that is put in the list. The `num` and `var` expressions compile to `LOADI` and `LOAD` instructions, respectively, with the same exact arguments, whereas arithmetic operations compile to recursive compilations of parts of the operation. Recursive compilation here means that if an arithmetic expression is built out of other arithmetic expressions, they will be compiled into a singular list, preserving the order of operations. The order is preserved because the execution of the list of arithmetic expressions is strictly from left to right, therefore addition, subtraction, multiplication and division compile correctly.

```
def acomp : aexp → list instr
| (num n) := [LOADI n]
| (var x) := [LOAD x]
| (add e₁ e₂) := acomp e₁ ++ acomp e₂ ++ [ADD]
| (sub e₁ e₂) := acomp e₁ ++ acomp e₂ ++ [SUB]
| (mul e₁ e₂) := acomp e₁ ++ acomp e₂ ++ [MUL]
| (div e₁ e₂) := acomp e₁ ++ acomp e₂ ++ [DIV]
```

5.2 Boolean expressions

Boolean expressions are a little trickier to define since they do not directly correspond to instructions. In this case, boolean expression compilation takes additional parameters compared to arithmetic, of which flag `f` of type `Prop` indicates what value of the boolean expression represents the true case, and offset `n` is used to specify how much the program counter needs to be shifted by.

```
noncomputable def bcomp : bexp → Prop → ℤ → list instr
| (Bc v) f n := if (v = f) then [JMP n] else []
| (Not b) f n :=  bcomp b (¬ f) n
| (And b1 b2) f n :=
```

```
    let cb2 := bcomp b2 f n,
      m := if (f = true) then int.of_nat (list.length cb2) else int.
   of_nat (list.length cb2) + n,
      cb1 := bcomp b1 false m
    in cb1 ++ cb2
| (Less a1 a2) f n := acomp a1 ++ acomp a2 ++
  (if (f = true) then [JMPLESS n] else [JMPGE n])
```

Boolean constant or `Bc` case is simple – if the constant is equal to the flag, then create a jump; otherwise, nothing is returned. The `Not` case is self–explanatory – it changes the flag to be the opposite, such that during the compilation it compares the negated case. `Less` is creating either `JMPLESS` or `JMPGE` instruction depending on the flag `f`. The `And` expression compilation builds a list of instructions, where if the first compilation is true, the second is executed; otherwise, after the first compilation, the jump is set to be over the second and by n. There is no way to make these compilations parallel, so they have to be executed sequentially, which makes the definition of the expression complicated.

When defining `bcomp` cases I use `noncomputable` flag because the definition contains an if–then–else construction that depends on a `Prop f`. The conversion of a `Prop` to `bool` (to evaluate the condition) uses classical constructions, which do not have computational content (are not executable) [3].

5.3 Commands

The main idea of compiled commands is a program that will perform the same state transformation and end with a program counter at the `(ccomp c).length`. During that transformation, commands can take multiple intermediate state steps or have intermediate stack values but, in the end, they will result in a compiled program.

```
noncomputable def ccomp : com → list instr
| com.SKIP := []
| (Assign x a) := acomp a ++ [STORE x]
| (Seq c1 c2) := ccomp c1 ++ ccomp c2
| (If b c1 c2) := (
  let cc1 := ccomp c1,
    cc2 := ccomp c2,
    cb := bcomp b false (list.length cc1 + 1)
  in cb ++ cc1 ++ (JMP (list.length cc2) :: cc2)
)
| (While b c) := (
  let cc := ccomp c,
    cb := bcomp b false (list.length cc + 1)
  in cb ++ cc ++ [JMP (-(list.length cb + list.length cc + 1))]
)
```

The commands are mostly trivial except for the ones that have branches: `If` and `While`. `If` compiles to the instruction list, where if the boolean expression is true, `cc1`(*then* branch) follows the condition without jumps and afterwards jumps over `cc2`. Otherwise, `cc2` (*else* branch) is executed by jumping from the condition over the `cc1` branch and the `JMP` instruction. `While` command is in a similar manner as `If`, just in this case, instead of *then* branch, there is either exiting the loop if the condition is false or jumping back to the condition after the while body.

The command compilation also uses the `noncomputable` flag because it uses `bcomp`, which is already marked as `noncomputable`.

6 Compilation correctness

To have a complete compiler, I prove for each expression the correctness statement, which essentially means that the compiled expression executes as intended.

6.1 Arithmetic expressions

The correctness of arithmetic expressions (like all correctness proofs) follows an induction on the arithmetic expression. Induction in this case, inspired by the Isabelle proof, generalizes stack such that it could be specified in each induction hypothesis as needed. However, I don't apply this in this thesis. These induction cases can be split into two parts – numbers and variables, and operations with them because the proofs for these grouped cases are very similar. The Isabelle proof is based on induction but is completed automatically by *fastforce*.

```
lemma acomp_correct {a : aexp}  {s : state}  {stk : stack} :
exec (acomp a) (0, s, stk) (list.length (acomp a), s, (eval a s) :: stk)
```

6.1.1 Numbers and variables

Since the proofs for numbers (constants) and variables are very similar, they can be covered together. The compilation creates a list of a single instruction – `LOADI` or `LOAD`. This means that a single–step execution is enough to verify a single instruction. Therefore, I use `star.single` lemma to simplify the goal, and I use `exec1I` lemma to create `exec1 [LOADI a] (0, s, stk) (1, s, a :: stk)` subgoal (in the `var` case it is `[LOAD a]`).

6.1.2 Operations with arithmetic expressions

To prove the correctness of arithmetic operations, I use `star.trans` lemma twice to show how three lists are concatenated together. Both `acomp a1` and `acomp a2` compilations come from the respective inductive hypotheses, and the only part that I manually create is the arithmetic operator instruction (e.g. `[DIV]`) execution itself. To create arithmetic instruction execution, I use `star.single` lemma to transform multiple steps into one `exec1` step and use `exec1I` to create the goal execution.

6.2 Boolean expressions

Boolean expression compilation correctness is based on induction, where it generalizes `f` (boolean flag) and `n` (offset used in a jump instruction), with initial hypotheses being that `0 ≤ n`. Having an arbitrary flag and offset helps in proofs where the induction hypothesis (later *IH*) is very simple for separate executions and I need to create a sequential process out of those hypotheses, or I need to specify `f`. All of the case proofs involve splitting the goal using `by_cases` tactic or `cases classical.em`, which create a true and false case for a boolean expression, which, in turn, eliminates if–then–else in the goal.

```
lemma bcomp_correct (n: ℤ) ( b f s stk)
(h_nneg : 0 ≤  n) :
exec (bcomp b f n) (0, s, stk) (list.length (bcomp b f n) + (if (f =
   bval b s) then n else 0), s, stk)
```

6.2.1 Not

In Isabelle, the *Not* case generalized f is specified to be $\sim f$ (not f), and then *fastforce* completes the proof.

In Lean, the proof is more involved. Inspired by the Isabelle proof, I first specify *IH* to use `¬f` instead of `f`. Then I split the if statement into a true case and a false case to simplify the goal. In the true case, where `b` evaluates to be equal to `f`, simplifying the goal and *IH* is enough to get the same result, and in the false case simplifying is enough with a manual proof of negation of the equation is equivalent to one side being equal to the negation of the right-hand side (`¬ (f = bval b s) → f = ¬ bval b s`).

6.2.2 And

In Isabelle, the *And* case is more specified than other proofs, meaning that the splitting of cases and *IH* specification are explicitly defined. After these mentions, *fastforce* automatic steps finish the case.

In Lean, as in the `Not` case, I take inspiration from the explicit information the Isabelle proof gave me and used it in my favor. As described in the *Concrete Semantics*, the case split is on `f = true` since the jump after the first part of execution depends on the `f` (shown in the definition in 5.2). In the true case of this split *IH* is specified to have `f` as false and `n` as `int.of_nat(list.length (bcomp b2 f n))`, whereas in the false case `n` is `int.of_nat (list.length (bcomp b2 f n)) + n`. Here I use `int.of_nat` because the length of a list is a natural number. In both cases, however, the proof follows the same structure but differs in small part due to the `f` being true or false. Since the execution is of two instruction lists built during the boolean compilation, I use `exec_append_trans` lemma to split the goal into smaller subgoals and to show how these two lists combine into one. The first list part is relatively easy to prove using specified *IH* and the second part requires more details to get to the goal. The second

part has an if–then–else in its goal program counter, which means that another case split on the value of the first `And` part is created. In one of the cases, the first part is false, which means that the conjunction is false, so it jumps over the second part, making it a reflexive execution. In the other case, the jump depends on the second part being true or false, which means that it needs to be executed to get its value, so there is no jump. The proof of the `And` case is complex because it has to specify all these boolean values and cover all combinations of them.

6.2.3 Less

The `Less` proof is constructed out of two applications of `exec_append_trans` lemma to add together two arithmetic expressions and the jump command. To create arithmetic expressions I could use already proven `acomp_correct` lemma, which makes the proof process easier. To create jump instructions I manually create an execution with an if–then–else, such that I could simplify it and prove it in smaller cases rather than combining cases bottom–up. The if–then–else simplification cases are of similar structure where `exec` is transformed into `exec1` by `star.single` lemma, and then relevant comparison simplification is applied which is straightforward.

In Isabelle, this is one of the cases of boolean compilation that is proven automatically by *fastforce*.

6.3 Commands

The command correctness proof shows the correct representation of the source program in the machine program. When the program is correctly represented, it means that in the machine code, it should cause the same state change, and it should terminate if and only if the source program terminates. Termination can be expressed as a program counter reaching the end of the compiled expression, i.e. `i = (ccomp c).length`, if it started from 0. In the same way, the stack should not change.

The state change can be expressed as `(c, s)` $\Longrightarrow$ `t`, which is called big–step operational semantics. This can be read as "Starting with the state `s`, executing `c` will terminate with a state `t`." I use modified big–step semantics defined from *The Hitchhiker's Guide* that is used to create a minimalist imperative programming language interpreter.

When considering the compiler, the semantics play a crucial role in ensuring that the source code emulates the compiled code. This guarantees that every final state produced by the compiled code is depicted in the source code. Thus, all compiled code can be considered trusted if it terminates. To know that it terminates it needs to be proven that the compiled code represents correctly the source code [1]. I will do only the latter part which is easier and it can be understood as follows: "if the source program executes from state s to state t, then the compiled program will as well." Due to time constraints, I am not able to prove that the compiled code is trusted if it terminates.

```
lemma ccomp_bigstep {c : com} { s : state } {t : state } (stk : stack)
(h_step : (c, s) ⟹ t) :
exec (ccomp c) (0, s, stk) (list.length (ccomp c), t, stk)
```

The proof of this lemma is by induction on big–step semantics `h_step`. I use `induction'` tactic to generalize induction hypotheses (easier to use them) and to get smaller cases, which are easier to work with, generated by big–step semantics lemmas.

6.3.1 Assign

The `Assign` command is constructed out of an arithmetic expression and `STORE` instruction, so `exec_append_trans` lemma is applied to split the two into separate cases. The arithmetic expression can be proven by applying `acomp_correct` lemma and for `STORE` instruction `star.single` lemma is used. To create the instruction, a hypothesis that matches the goal pattern is manually defined and simplified such that is true. This proof is relatively simple compared to other command cases, and that can also be seen in Isabelle, where this case is proven by *fastforce* and *simp*.

6.3.2 Sequence

The command `Seq` is compiled out of two commands. In this case, in the proof for each of the commands, there is a relevant *IH*. These induction hypotheses are applied to show both commands in the `exec_append_trans` application. Since these execution hypotheses are generalized in a way where they follow the transitive configuration (because of `induction'`), there is no further work to be done.

In Isabelle, the proof follows the same structure by showing transitivity from the first part's start configuration to the second part's end configuration from induction hypotheses. These parts are joined together by automatic proof *blast* which uses *star_trans*; this is essentially the same as using `exec_append_trans` in Lean.

6.3.3 If

`If` is split into two true and false cases by the `induction'` which made the proof easier to create and manage. Both cases follow a similar structure of connecting the condition to *then* case or the condition to *else* case. The condition is created using `bcomp_correct` lemma, but the resulting cases had to be manually defined. In the true case, there is another application of `exec_append_trans` lemma to create a *then* command compilation joined with a jump over *else* case. Jump is proven with `star.single` lemma and simplification, where the *then* case is created from the *IH*. On the other hand, the false case can be constructed using `exec_appendL_if` lemma, because it has a starting program counter position after *then* case and the jump instruction. This lemma states that if the program counter is more than or equal to the list length, which is appended to the left, the execution continues in the initial list, in this case, the initial list is the jump instruction concatenated to *else* case. To skip the jump instruction `exec_cons_1`

lemma can be used, which states that if a single instruction is added to the left of the list, the execution continues in the initial list bounds but everything is shifted by one.

In Isabelle, this proof is automatic using *fastforce*.

6.3.4 While

The `While` case proof is the most involved proof of this project. In Lean, the `induction'` separates it into `while_true` (execute the code block (later – body) while the condition is true) and `while_false` (continue execution without executing the body) cases.

`While_false` case is simple and similar to if *else* case, where the condition is concatenated with the skip over the body. However, in the while loop, if the condition is false, the execution stops, so using `exec_append_trans` lemma the boolean expression, which is created using `bcomp_correct`, and execution of nothing, which is created with `star.refl`, are connected together.

`While_true` case is complex because it is a loop, so there are three executions to be connected and the loop execution itself. Inspired by the Isabelle proof of `While` case, I have defined all executions separately and then used `exec_append_trans` to connect them. All three execution steps can probably be proven together, but separating them using `exec_appendR` or `exec_appendL` makes it easier to understand the goal and work towards it. The three steps of execution are the condition, then the body execution part, and jumping back to the condition after the body execution is done. All of the steps are relatively simple when simplified from the initial goal because the initial list is puzzling to look at. In each step, I define manual equalities that I use to rewrite the goal to contain "`+ 0`" or some negative list length because otherwise the `exec_appendL` lemma would not work (it requires something to be added to the program counter, not only appended list). After simplifying the goals in each execution, the condition is finished using `bcomp_correct` lemma, the body section is finished using *IH* of the singular execution, and the jump back is finished with `star.single` application. To complete the while, there is the last part which is the loop execution itself, which is the compilation of the while. It might seem confusing why there is this execution of the while, but after a singular loop execution, the loop continues, thus it follows the singular execution. To prove the top–level while that follows individual execution steps, while *IH* can be applied.

7 Related work

In this section, I cover similar formal verification systems to what I have covered in this paper. This includes verification in proof assistants such as Isabelle [1, 7] and Coq [8, 9] and program verifiers Why3 [10] and Stainless [11].

Interactive proof assistants include Isabelle and Coq, in which the proofs are automated to the extent that the user does not need to manually create full proofs. One of the works comparing these theorem provers is by Czajka *et al.* [8]. They compare the assistants by translating theorems and proofs from Isabelle to Coq from *Concrete Semantics* by Nipkow and Klein [1] like I do to Lean 3, but with additional chapters, which in my case have already been covered in *The Hitchhiker's Guide* by Baanen *et al.* [2]. Their goal is to improve the CoqHammer tool, which searches for Coq proofs while showing the similarities and differences to Isabelle. Similar to Isabelle's *auto* or other automated tactics, CoqHammer has the main tactic *hammer*, which combines machine learning and automated reasoning techniques to accomplish goals automatically. It also provides *sauto* which is a strong version of the Coq tactic *auto* to simplify the goal as much as possible and perform rewrites using existing hints from an importable database. Even though Coq has automation, Czajka *et al.* [8] show that Isabelle does perform faster (on the same hardware) and has fewer lines of code and tactics used than Coq.

I have shown a small example of formal verification, but there are real–world examples of formally verified systems. One of the examples is C11 compiler formalization in Coq by Krebbers [9]. The compiler is formalized in normalized CH_2O semantics, which is an incomplete but sufficient representation of the compiler in Coq. The normalized semantics were created such that if there is proof for any given program with respect to them, the proof would hold for any existing C compiler that adheres to the standard. It also allows improvement in future versions of the C compiler standard and addresses issues with indeterminate memory and pointers which are important in such a language as C. Another example of a real–world application of this is WebAssembly mechanization and verification in Isabelle by Watt [7]. The verification of this programming language allowed Watt to fix several errors in the official WebAssembly specification and create an executable verified interpreter that could be used instead of the official interpreter in the future.

In addition to proof assistants, there are program verifiers that create dedicated environments to verify the behavior of programs written in some specific programming language. They can help one verify programs written in some imperative language, and these program verifiers have a high level of automation in proofs because the languages specified in them are less expensive than in proof assistants. One of the program verifiers, Why3, is discussed by Clochard *et al.* [10]. Why3 was also used to create an

interpreter and prove the correctness of a simple programming language, and it is designed to be even more automated and fast. Automatization is relevant when proving not a small compiler but a complex program. Another example of a program verifier is Stainless, which is a Scala program verifier developed by the LARA group at EPFL and presented by Voirol [11]. To verify them, Scala programs are translated into an intermediate verification language using an Inox interpreter, and then programs can be verified with some automation in the system using Stainless.

8 Conclusion

In this thesis, I presented the formalized operational semantics of a compiler in Lean 3. I have shown the difference between the Lean theorem prover from Isabelle/HOL. By doing so, I have explained what could be behind automatic proofs that are used to verify a compiler modelled in the same way in Isabelle as it is in Lean. Overall, the proofs seemed to be more explicit and detailed compared to the Isabelle proofs, which could have been expected due to the differences in tactics between these theorem provers; however, I did not know how involved each proof would be. Additionally, I have shown related work that has been done in other interactive proof assistants and current work in program verifiers that do a similar job as my implemented example, but on a higher scale.

I have encountered some difficulties in this thesis. One of the difficulties was that the material I was covering was far more complex than my Lean knowledge at the start of the project, but after getting used to the syntax and understanding the higher–level ideas of the proofs from *Concrete Semantics*, it got easier. One of the helpful steps that I could take in this thesis was using the material covered in *The Hitchhiker's Guide* such as definitions and lemmas of `star`, `state`, and `big-step` semantics. I had to adjust some of these definitions but most of the groundwork for them was already done. Therefore, I could focus on compiler construction and verification, which I achieved without any unfinished proofs (without using `sorry`). Due to the lack of time, I have not finished full command compilation verification, since I have not implemented `small-step` semantics to prove that all compiled code is trusted.

Future work for this project could be to finish the complete compiler verification and include it in *The Hitchhiker's Guide* as a chapter along with the other chapters inspired by *Concrete Semantics*. Another work idea could be to port this verification infrastructure from Lean 3 to Lean 4. Lean 4 is the newer version of Lean, which includes new features but is not backwards compatible with Lean 3, so this ported example could show the capabilities of Lean 4.

Bibliography

[1] T. Nipkow and G. Klein, *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014.

[2] A. Baanen, A. Bentkamp, J. Blanchette, J. Hölzl, and J. Limperg, "The Hitchhiker's Guide to Logical Verification." `https://github.com/blanchette/logical_verification_2020/raw/master/hitchhikers_guide.pdf`, 2020.

[3] J. Avigad, L. d. Moura, and S. Kong, "Theorem proving in Lean." `https://leanprover.github.io/theorem_proving_in_lean/`, Apr 2023.

[4] J. C. Blanchette, L. Bulwahn, and T. Nipkow, "Automatic proof and disproof in Isabelle/HOL," in *Frontiers of Combining Systems: 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings 8*, pp. 12–27, Springer, 2011.

[5] The Mathlib Community, "The Lean mathematical library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, (New York, NY, USA), p. 367–381, Association for Computing Machinery, 2020.

[6] J. Limperg, "A novice-friendly induction tactic for Lean," in *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021* (C. Hritcu and A. Popescu, eds.), pp. 199–211, ACM, 2021.

[7] C. Watt, "Mechanising and verifying the WebAssembly specification," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, (New York, NY, USA), p. 53–65, Association for Computing Machinery, 2018.

[8] Ł. Czajka, B. Ekici, and C. Kaliszyk, "Concrete semantics with Coq and CoqHammer," in *Intelligent Computer Mathematics* (F. Rabe, W. M. Farmer, G. O. Passmore, and A. Youssef, eds.), (Cham), pp. 53–59, Springer International Publishing, 2018.

[9] R. J. Krebbers, *The C standard formalized in Coq*. PhD thesis, Radbout University Nijmegen, 2015.

[10] M. Clochard, J.-C. Filliâtre, C. Marché, and A. Paskevich, "Formalizing semantics with an automatic program verifier," in *Verified Software: Theories, Tools and Experiments* (D. Giannakopoulou and D. Kroening, eds.), (Cham), pp. 37–51, Springer International Publishing, 2014.

[11] N. C. Y. Voirol, *Verified Functional Programming*. PhD thesis, Lausanne, 2019.