Formal Verification with Lean

Stefan-Octavian Radu

Julius-Maximilians-Universität Würzburg
Institute for Computer Science Chair for Computer Science VI
Am Hubland, D-97074 Würzburg.
informatik.uni-wuerzburg.de/is

Abstract. Lean is a proof assistant primarily used in the field of mathematics. In this paper we discuss how this powerful tool can be used to reason about programs, using formal methods. We cover three approaches to formal semantics, the Lean programming language and the features that set it apart. We conclude with a practical example where we reason about programs written in an unusual programming language.

Keywords: Formal Verification · Functional Programming · Lean

1 Introduction

Lean is a project started at Microsoft Research, with the main goal of offering a public library to hold all mathematical knowledge. It is a proof assistant, as well as a powerful functional language, based on dependent types. Dependently typed languages treat types as first-class objects, and Lean is no exception to this. The added expressiveness of the type system allows users to embed proofs into the types themselves. While typically used for mathematical theorems, we want to discuss how Lean and the unique features that it offers can be utilised for formal verification. We will take a look at the options available for formalising the semantics of programming languages and reasoning about them. In this regard, we will give a high-level overview of *Operational Semantics*, *Axiomatic Semantics* and *Denotational Semantics*. Next, we will give an overview of the Lean programming language, and will discuss how its features can be used for theorem proving. Lastly, we will give a practical example in which we will describe the approach we took to formalise the semantics of the esoteric programming language Brainf*ck. We will also present a few theorems that we proved based on the formalised semantics.

2 Formal Verification

"Formal verification is the act of proving or disproving the correctness of a system with respect to a certain formal specification or property, using formal methods of mathematics" [14]. In this work, we will focus on the formal verification of imperative programming languages. In this context, formal verification refers to

formally specifying the syntax and semantics of the programming language. This path of action enables us to reason about concrete programs and prove properties of the semantics. In this section, we will only give a high level overview of three approaches to describing the formal semantics of a programming language: *Operational Semantics*, *Denotational Semantics* and *Axiomatic Semantics*. A more in-depth coverage of program semantics can be seen in chapters 9-11 of *"The Hitchhikers Guide to Logical Verification"* [7].

2.1 Operational Semantics

An operational semantics can be thought of as an *idealised interpretor*, which describes in a formal way how the program instructions alters the state of the program. There are two main variants: *big-step* semantics and *small-step* semantics.

Big-step semantics enable us to reason in terms of transitions from an initial state to a final state of the program. More concretely, big-step semantics is based on relations such as: $(S, s) \Rightarrow t$. Intuitively, this relationship suggests that the program `S` executed in the starting state `s` will terminate in the final state `t`. The semantics of a programming language is usually presented as a set of derivation rules for each of the instructions in the target language.

Small-step semantics are more complex, but enable us to reason in more detail, compared to the big-step semantics. Small-step semantics is based on relations such as: $(S, s) \Rightarrow (T, t)$. Intuitively, this relationship suggests that executing one step of program `S` in the starting state `s` will terminate with the program `T` left to be executed in state `t`.

With both the big-step and the small-step semantics defined, one can prove the equivalence between them:

$$(S, s) \Rightarrow t \Leftrightarrow (S, s) \Rightarrow *(skip, t)$$

where $\Rightarrow *$ is the reflexive transitive closure of the binary relationship defined with $\Rightarrow$, and `skip` is the null operation. With the semantics defined, one can start reasoning about concrete programs, or start proving properties such as termination, or determinism about the language. A concrete example of defining the big-step semantics for a programming language and using it for reasoning about programs can be seen in section 4.

2.2 Axiomatic Semantics (Floyd-Hoare Logic)

The Floyd-Hoare Logic (also called *axiomatic semantics*) is a formal system based on logic derivation rules, for reasoning rigorously about the correctness of computer programs. The system was first proposed by Tony Hoare in 1969 [12]. The ideas stem from the previous works of Robert W. Floyd who proposed a similar system in 1967 [11].

Hoare logic is most useful for reasoning about concrete programs in order to prove them correct. The building blocks of Hoare Logic are Hoare Triples, of

the form $\{P\}\ C\ \{Q\}$. Here P is a precondition, C is the program, and Q is the postcondition. We read a Hoare rule the following way: *"If the precondition P is true before S is executed and the execution terminates normally, the postcondition Q is true at termination"* [7]. From the interpretation, we can notice that Hoare Logic can only guarantee partial correctness. This is because the above statement only holds, if the program terminates. If the program does not terminate, the behaviour is assumed to be arbitrary.

For reasoning about concrete programs, a set of derivation rules based on Hoare Triples for the target language is typically used. Before ending the overview of Floyd-Hoare Logic, let's consider a concrete example for a sequential composition operation in an arbitrary language. The corresponding derivation rule is:

$$\frac{\{P\}\ S\ \{R\} \qquad \{R\}\ T\ \{Q\}}{\{P\}\ S;T\ \{Q\}}\ \text{SEQ}$$

For $\{P\}\ S;T\ \{Q\}$ to hold, we must be able to execute S and T sequentially. As such, there must exist an intermediary condition R, such that $\{P\}\ S\ \{R\}$ holds with R as postcondition, and $\{R\}\ T\ \{Q\}$ holds with R as precondition.

Those interested to further explore this topic, can look into the Dafny project.

2.3 Denotational Semantics

Denotational semantics are used to express what programs "mean". It can be views as an idealised compiler that converts programs to mathematics. In this sense, every program is mapped to a mathematical object, such as:

$$[\![\ ]\!] : \mathit{syntax} \to \mathit{semantics}$$

A key property of denotational semantics is *compositionality*: the meaning of a compound statement must be defined in terms of the meanings of its components [7]. A fully compositional definition enables reasoning in terms of equations, which can be more convenient than previous approaches, such as *big-step semantics*. What we are looking for structurally recursive equations such as:

$$[\![S\ ;T]\!] = \ \ldots\ [\![S]\!]\ \ldots\ [\![T]\!]$$

In practice, the denotational semantics of a programming language can be expressed in a relational manner, as a set of relations $\{\mathit{State} \times \mathit{State}\}$, where *State* denotes a theoretically possible point during the execution of the program (state of memory, variables, etc.). In the end of the discussion of denotational semantics, let's consider a more concrete example, by revisiting the sequential composition of two statements `S` and `T`, which was mentioned above. Let's assume that $[\![S]\!] = r_1$ and $[\![T]\!] = r_2$. Then, $[\![S\ ;T]\!]$ is equal to the relational composition of r_1 and r_2:

$$r_1 \circ r_2 = \{(a, c) \mid \exists b, (a, b) \in r_1 \wedge (b, c) \in r_2\}$$

3 The Lean programming language

Lean is a project which was started in 2013 by Leonardo de Moura while working at Microsoft Research [6]. It is released under the open-source Apache 2.0 license, which allows it to be easily used and extended by the community.

With regards to its applications, Lean is a multi-purpose tool. First of all it is a theorem prover, based on dependent types 3.2, and more specifically on the system of *Calculus of Constructions* [6]. As mentioned in the "Functional programming in Lean" book [8], *"Dependent type theory unites the worlds of programs and proofs"*. This enables Lean to also be a fully capable, general-purpose programming language. As a matter of fact, Lean is (partially) implemented in Lean.

In this section we will give a high level overview of Lean. We will firstly cover its basic syntax and features as a programming language. Then we will discuss dependent types and how they enable Lean to be used as a theorem prover. Lastly we will go over Lean's *Tactic Mode*, which enables its users to write clear and elegant mathematical proofs.

3.1 Functional Programming in Lean

"When viewed as a programming language, Lean is a *strict*, *pure*, functional programming language, with *dependent types*" [8]. It is *strict*, in the sense that when calling a function, the function body is only evaluated after the function arguments have been evaluated. It is *pure*, in the sense that functions do not have side effects. This also means that evaluating a function multiple times with the same arguments will yield the same result. In these regards it is similar to Haskell [1], but differs from it by not being lazily evaluated and having a syntax more similar to OCaml [2].

Simple Definitions and Functions. In Lean we can use the `def` keyword to define anything from constants to complex functions. See listing 1.1 for examples.

```
-- simple definitions
def a: Nat := 42
def hw: String := "hello␣" ++ "world!"
-- #eval evaluates an expression inline and shows the result
#eval a  -- 42
#eval hw -- "hello world!"
-- defining functions
def add1 (k: Nat): Nat := k + 3
def add2 (a b: Nat) := a + b
#eval add1 3   -- 6
#eval add2 3 3 -- 6
-- highlighting a more complex function and currying
def applyTwice (f: Nat  Nat) (k: Nat) := f (f k)
#eval applyTwice (add 3) 4 -- 10
```

Listing 1.1. Examples of simple definitions and function definitions in Lean

Defining new types. Most of the non-primitive data types in Lean are inductive types. Those are types allow choices (instantiating an element to be one from a set, see listing 1.2) and also allow instances of themselves as part of the definition (see listing 1.3). Lean also supports the creation of structures, which group multiple values together, and are very similar to structures found in imperative languages such as C or Rust [8].

```
inductive Bool where
  | false : Bool
  | true : Bool
```

Listing 1.2. The Bool type has two constructors without parameters: one for `true` and one for `false`

```
inductive Nat where
  -- just zero (0)
  | zero : Nat
  -- successor of k (k + 1)
  | succ (k : Nat): Nat
```

Listing 1.3. Recursive definition of natural numbers

Inductive types are very useful, because we can reason based on their structure, and also we can use induction to prove mathematical properties of them.

Recursion and pattern matching. When using inductive types, pattern matching can be used in order to handle each of the type's constructors independently. For example, to calculate the factorial of a natural number, we can give the function's implementation for the two constructors of the `Nat` data type, utilising the `match` keyword (see listing 1.4).

```
def factorial n :=
  match n with
--| 0
  | zero => 1
--| k + 1
  | succ k => (k + 1) * factorial k

#eval factorial 5 -- 120
```

Listing 1.4. Using recursion and pattern matching in Lean

We also notice how in listing 1.4 Lean can infer the type of the argument and the return type of the function from the match case, so we don't have to explicitly state them.

Lists and higher-order functions. Lists are part of the standard library of Lean, and are one of the building blocks of what we will present in the next section. In listing 1.5 we show some common List functions, as well as how lists can be used in conjunction with higher-order functions such as `map`, `foldr`, `filter`. One more unusual feature of Lean is the that it facilitates the introduction and usage of mathematical (unicode) symbols. This becomes especially useful when

working with mathematical proofs. An example of this can also be seen in the 1.5 listing, where the λ symbol is used as an equivalent to the `fun` keyword, which introduces a lambda abstraction.

```
#eval [1, 2, 3]            -- [1, 2, 3]
#eval 5 :: [1, 2]          -- [5, 1, 2]
#eval [1, 2] ++ [3, 4]     -- [1, 2, 3, 4]
#eval (List.range 3)       -- [0, 1, 2]

#eval (List.range 3).map (fun x => x * 2)         -- [0, 2, 4]

#eval (List.range 3).foldl (fun x y => x + y) 0   -- 6
#eval (List.range 3).foldl (λ x y => x + y) 0     -- 6

#eval (List.range 3).filter (λ x => x ≤ 1)        -- [0, 1]
#eval (List.range 3).filter (. ≤ 1)               -- [0, 1]
```

Listing 1.5. An example of using lists and higher-order functions in Lean. Usage of lambda abstractions and syntactic sugar can be observed.

Other features of Lean. Lean is a fully capable programming language. As such it includes a complex array of features such as: structures, type classes, functors, applicative functors, mondas, monad transformers. We will not cover such features since they are outside the scope of this work. However, we provided listing 1.6 as an example of the `IO monad` and the `do` notation.

```
def main: IO Unit := do
  let stdin <- IO.getStdin
  let stdout <- IO.getStdout

  IO.print "What␣is␣your␣name?:␣"
  let input <- stdin.getLine
  let name := input.dropRightWhile Char.isWhitespace

  stdout.putStrLn s!"Hello,␣{name}!"
```

Listing 1.6. Hello World example in Lean

3.2 Dependent Types

Dependent types stem from "Type Theory", the main idea of which is that every expression has a type. For example if `a` and `b` have the type `bool`, the expression `a && b` is also of type `bool`. This also means that we can create new types by combining types. As such, if α and β are types $\alpha \to \beta$ would be a function from α to β.

In Lean, types are considered first-class objects of the language. This fact, in turn, means that each type has a type of itself [6]. In lean one can use the

`#check` command to find the type of an expression. An exemplification of these claims can be seen in listing 1.7.

```
def α: Type := Bool
def β: Type  Type := List

#check α         -- Type
#check β Nat     -- Type
#check p = p     -- Prop
#check Prop      -- Type
#check Type      -- Type 1
#check Type 1    -- Type 2
#check Type 2    -- Type 3
-- etc.
```

Listing 1.7. Types are first class objects in Lean

A subtle aspect of the language can be noticed in 1.7 listing. Lean has an infinite hierarchy of types. This prevents the underlying logic from being unsound, by avoiding *Girard's Paradox* [10]. *Girard's Paradox* is the "type-theoretical analogue of *Russell's Paradox*" from set theory, which preceded type theory as a foundational theory for mathematics [13] [9].

What makes dependent types *dependent* is that types can depend on terms of other types. The distinction between terms (or values) and their types is blurred and programs can be evaluated to types. A good example that illustrates this ideas is the creation of an array of fixed length. A term of this type would depend firstly on the type of the values it holds, and secondly on the value which states its length. We could then use this type in functions which only accept fixed length arrays, without needing to specify the exact length, or know it at compile time. By extending the expressiveness of types in such a way, programmers can catch more errors at compile time. Listing 1.8 exemplifies such a use case.

```
inductive Vect (α : Type) : Nat  Type where
  | nil  : Vect α 0
  | cons : α  Vect α n  Vect α (n + 1)
  deriving Repr

def replicate {α: Type} (x: α) (n: Nat): Vect α n :=
  match n with
  | 0     => .nil
  | k + 1 => .cons x (replicate x k)

#eval replicate 42 5 -- [42, 42, 42, 42, 42]
```

Listing 1.8. Defining a fixed-size list type and defining a function which produces a fixed-size list based on input parameters

"Dependently typed programs are, by their nature, proof carrying code" [5]. If we see types as logical formulas, then the program which computes an element

of that type is a proof of the formula. If we can build it, then it is true. In the listing 1.8, the `replicate` function is a proof that $\alpha \to$ `x` $\to$ `n` $\to$ `(Vect` α `n)`. It is clearly not very intuitive how this analogy would be useful, but it becomes more apparent when the type resembles more "classic" theorems, such as 1.9. We can leverage these properties to prove various properties of programming languages (see section 4).

3.3 Tactic Mode

In Lean, *theorems* (introduced with the `theorem` keyword) are similar to definitions (introduced using the `def` keyword). The difference is that they cannot be called and they signal to the compiler not to unfold the definition. The usage of theorems is similar to their usage in mathematics, which is to record formally proven statements, based on axioms or other proofs. In Lean, the `example` keyword, introduces unnamed theorems [6].

Tactic mode is an alternative to using functions for theorem proving. It is introduced through the `by` keyword. Tactic mode uses *tactics*, which are commands that one can use in order to build proofs in an incremental way. This is achieved by reasoning on goals and modifying them step by step until the proof is complete [6]. Listings 1.9 and 1.10 show equivalent proofs written functionally and by using tactic mode respectively.

```
example: p ∨ q → q ∨ p :=
  fun (h: p  q) =>
    Or.elim h Or.inr Or.inl
```

Listing 1.9. A proof example for a theorem expressed in propositional logic. We use the Or.elim theorem and the Or.inl and Or.inr constructurs for a logical Or expression, which are built into the standard Lean library. See 1.1 for the definitions of these constructs.

```
example: p ∨ q → q ∨ p :=
by
  intro (h: p  q)
  apply Or.elim
  . exact h
  . apply Or.inr
  . apply Or.inl
```

Listing 1.10. An equivalent implementation to 1.9 can be given using tactics.

```
#check @Or.inl  -- @Or.inl : ∀ {a b : Prop}, a → a ∨ b
#check @Or.inr  -- @Or.inr : ∀ {a b : Prop}, b → a ∨ b
#check @Or.elim -- @Or.elim : ∀ {a b c : Prop},
                --     a ∨ b → (a → c) → (b → c) → c
```

Listing 1.11. Useful definitions to better understand examples 1.9 and 1.10.

Throughout the next section we will use a number of different tactics to construct proofs about our chosen programming language. We will cover the main tactics used, and briefly discuss the way they interact contribute to the proof.

intro. If the goal is $p \to q$, then `intro h` will introduce `h: p` as a hypothesis, and then change the goal to `q` [3].

exact. If the goal is a statement `P`, then `exact h` will prove the goal and complete the proof, if `h` is a proof of `P` (has type `P`) [3].

apply. If we consider the case when we have a goal of `Q` to be proved, and we know `t:` $P \to Q$, then `apply t` will change the goal to `P`. Apply can be used to work backwards on goals. The reasoning behind it is that if we want to prove `Q`, by knowing `t`, it is enough to prove `P` [3].

cases. The cases tactics is used to reason about a part of a proof in terms of its building blocks. For example, if we have a hypothesis `h`, which depends on c_1, c_2, c_3, by using `cases h` we can reason about c_1, c_2, c_3 individually [3].

rfl. The `rfl` tactic stands for "reflexivity" and proves goals of the for `A = B`, if `A` and `B` are provably identical (e.g. $2 + 2 = 4$ is a goal which can be closed by using `rlf`) [3].

rw. The `rw` tactic stands for "rewrite" and is a way of substitution. If `h` is a proof of an equality `X = Y`, then `rw [h]` will change all occurrences of `X` in the goal to `Y`. There also exist a number of variants of this tactic (e.g for repeated rewrites, or for a reverse rewrite of `Y` to `X`) [3].

simp. The `simp` tactic stands for "simplify". It automatically rewrites every lemma provided to it, as well as every lemma tagged with the `@[simp]` keyword, which can be used by users to increase the power Lean's automation [3].

induction. If $n : N$ is a part of the goal of a proof, we can use `induction n with` to prove the goal by induction on `n`. This tactic splits the goal in two cases: the initial (zero) case, and then the inductive (successor) case, which is also paired with the inductive hypothesis.

4 A practical example

This section is inspired from chapter 9 of the *"The Hitchhikers Guide to Logical Verification"* book [7]. However, instead of using a theoretical language, we chose to formally specify the infamous esoteric programming language Brainf*ck. Even though it doesn't have many practical applications, it makes an interesting study-case, which is only facilitated by the language's small syntax.

We will define our variant of Brainf*ck's big-step semantics and cover some examples of how it can be used to prove the correctness programs, such as a `swap` procedure. All the code is available on github.

4.1 Brainf*ck

Brainf*ck has well known for its extremely small set of instructions. The syntax is comprised of only 8 instructions: `+-<>.,[]`. Its runtime is very simple as well, consisting of only a linear memory array and a pointer to the current memory cell.

State. Let's define our program state:

```
structure State : Type where
  inp: List Nat    -- input array
  out: String      -- output string
  before: List Nat -- values to the left of the current cell
  current: Nat     -- value of current cell
  after: List Nat  -- values to the right of the current cell
```

We represent the memory from three components: the current cell and two lists representing the elements to the left and right of the current cell respectively. This model enables us to have theoretically infinite memory. We use `Nat` as the memory cell's data type, which is an infinite size unsigned integer. While not reflecting the most popular implementations of the language, the `Nat` is easier to reason about, than for example an `UInt8`. We also have the input and output explicitly defined, which enables us to reason about possible inputs and outputs.

State operations. Having the state of the program defined, we can further define functions which describe how the state changes when one of the basic operations is applied on it. Listing 1.12 shows two examples of such functions. For more information on the meaning of each of the operations of Brainf*ck the appendix A can be consulted.

```
def applyPInc (s: State): State :=
    match s.after with
    | [] => s -- do nothing at the end of the band
    | h :: t => ⟨s.inp, s.out, *s :: s.before, h, t⟩

def applyVInc (s: State): State :=
    ⟨s.inp, s.out, s.before, *s + 1, s.after⟩

def applyInput (s: State): State :=
  match s.inp with
  | [] => ⟨[], s.out, s.before, *s, s.after⟩
  | h :: t => ⟨t, s.out, s.before, h, s.after⟩
```

Listing 1.12. Implementations for functions which modify the state after pointer increase (`>`), value increase (`+`) and input (`,`) operations.

Notice how we use ⟨ and ⟩ pairs to construct state structures. Also, we used another one of Lean's features to create a dereferencing-like notation for the current cell's value from a state `s: notation "*" s:100 => State.current s`.

Syntax. To be able to reason about concrete programs and the language itself, we will also define the Brainf*ck's syntax. We will name it `Op`, as can be seen in listing 1.13.

```
inductive Op : Type where
  | pInc      : Op              -- >
  | pDec      : Op              -- <
  | vInc      : Op              -- +
  | vDec      : Op              -- -
  | input     : Op              -- ,
  | output    : Op              -- .
  | brakPair  : Op -> Op        -- [S]
  | seq       : Op -> Op -> Op  -- S T
```

Listing 1.13. Syntax of the language implemented in Lean.

In addition to the known operations, we introduce the sequencing operation, which is used to chain operations together in a more complex program. We can now define programs using the newly created syntax:

```
#eval (brakPair ( seq vDec (
        seq pInc ( seq vInc pDec))): Op) -- [->+<]
```

Listing 1.14. Defining programs using the Op syntax

As listing 1.14 suggests, the syntax is rather cumbersome to use. However, we can define more convenient notations, so we can write the same program like this: `#eval ([~_>_+_<]: Op) -- [->+<]`. The notations are not a perfect match, because some symbols are reserved by Lean. Refer to appendix B for the full definition.

We now how all the building blocks necessary for defining the big-step semantics.

4.2 Big-step semantics

As mentioned before, the big-step semantics of a programming language is a set of transitions, defined formally as relations of the form $(S, s) \Rightarrow t$. The transitions for most of the operations are rather easy to define, with the final state being just the result of the previously defined state functions (see 1.12), called with the starting state as the parameter:

$$\frac{}{(S, s) \Rightarrow \texttt{s.applyOperation}} \text{ BF OPERATION}$$

The sequencing operation is defined as follows:

$$\frac{(S, s) \Rightarrow t \qquad (T, t) \Rightarrow u}{(S\_T, s) \Rightarrow u} \text{ BF SEQ}$$

The intuitive interpretation is that we can sequence operations S and T in state s such that they terminate in state u, only if there exists an intermediary state t, such that executing S in state s terminates in state t and then executing T in state t terminates in state u.

The loop operation is the most difficult to implement. In order to avoid infinite loops in our proofs, we must split it based on the condition. As such, we get two derivation rules, one for each truth value:

$$\frac{(S, s) \Rightarrow t \qquad ([S], t) \Rightarrow u}{([S], s) \Rightarrow u} \text{ BF LOOP-TRUE IF } {}^*\text{S} \neq 0$$

$$\frac{}{([S], s) \Rightarrow s} \text{ BF LOOP-FALSE IF } {}^*\text{S} = 0$$

The *loop-true* rule gives us a method of unravelling a loop iteration, in order to reason about it independently. The *loop-false* rule is trivial. Listing 1.15 contains the full implementation of the big-step semantics in Lean. We will use this further in order to write proofs on concrete programs.

```
inductive BigStep: Op × State → State → Prop where
  | nop  (s: State): BigStep (Op.nop, s)  s
  | pInc (s: State): BigStep (Op.pInc, s) s.applyPInc
  | pDec (s: State): BigStep (Op.pDec, s) s.applyPDec
  | vInc s: BigStep (Op.vInc, s) s.applyVInc
  | vDec s: BigStep (Op.vDec, s) s.applyVDec
  | brakPairTrue {ops} {s t u: State}
    (c: *s  0)
    (body: BigStep (ops, s) t)
    (rest: BigStep ((Op.brakPair ops), t) u):
      BigStep (Op.brakPair ops, s) u
  | brakPairFalse ops (s: State) (c: *s = 0):
      BigStep (Op.brakPair ops, s) s
  | seq (S s T t u)
    (h:  BigStep (S, s) t)
    (h': BigStep (T, t) u):
      BigStep ((Op.seq S T), s) u
  | input s: BigStep (Op.input, s) s.applyInput
  | output s: BigStep (Op.output, s) s.applyOutput
```

Listing 1.15. Implementation of big-step semantics for the Brainf*uck programming language

4.3 Proving Theorems

In this subsection we will discuss a few proofs that we made using the previously defined big-step semantics.

Zeroing a cell `[-]` is a very simple program which subtracts one from the current cell until its value is reduced to zero. Let's prove it!

```
#eval ([~]: Op) -- [-]
theorem dec_n {n: Nat}: ([~], (State.mk [] "" [] n []))
  ⟹ State.mk [] "" [] 0 [] :=
```

```
  by
    induction n
    case zero =>
      . apply BigStep.brakPairFalse
        . simp
    case succ d hd =>
      . apply BigStep.brakPairTrue
        . simp
        . apply BigStep.vDec
        . rw [State.applyVDec]
          simp
          assumption
```

Listing 1.16. Proof by induction in Lean, that `[-]` reduces a cell to 0

Listing 1.16 contains the full proof. The proof can be done by induction on n. The zero case is trivial, and the inductive step can be proved by reducing the state to the inductive hypothesis by unwrapping the loop once.

Adding two input numbers together. This theorem is a bit more difficult, and certainly takes more effort to prove. The distinctive aspect of this proof, compared to the previous one, is that we need to used induction on one of the variables, while generalizing the other. The *shortened* proof can be seen in listing 1.17.

```
def bfSumIn: Op := ,_>_,_<_[-_>_+_<]
#eval bfSumIn -- ,>,<[->+<]

theorem bfSum: (bfSumIn,
    (State.mk (a :: b :: i) o l x (y :: r)))
  ⟹ State.mk i o l 0 ((a + b) :: r) :=
  by
    rw [bfSumIn]
    apply BigStep.seq
    apply BigStep.input
    -- ... more use of apply
    apply BigStep.pDec
    rw [State.applyPDec]
    repeat rw [State.applyInput]
    rw [State.applyPInc]
    simp
    rw [bfAddition]
    induction a generalizing b with
    | zero =>
      rw [Nat.zero_eq]
      rw [Nat.zero_add]
      apply BigStep.brakPairFalse
      . rw [State.current]
    | succ k h =>
      apply BigStep.brakPairTrue
```

```
      . rw [State.current]
        simp
      . apply BigStep.seq
        apply BigStep.vDec
        -- ... more use of apply
        apply BigStep.pDec
      . rw [State.applyVDec]
        rw [State.applyPDec]
        rw [State.applyPInc]
        rw [State.applyVInc]
        simp
        rw [Nat.succ_eq_add_one]
        rw [Nat.add_assoc]
        rw [Nat.add_comm 1 b]
        exact @h (b + 1)
```

Listing 1.17. Proof in Lean that the algorithm `,>,<[->+<]` takes to numbers from the input and computes their sum.

Swapping two values. As part of our experiments, we also managed to prove correct the algorithm for swapping two values: `>[<+>-]>[<+>-]<<[>>+<<-]`. It can be split in three parts, which can be proved individually, using similar techniques as previously seen. Let's consider that `swap'` and `swap''` are proofs that the individual component behave as expected. Listing 1.18 shows the final proof, which combines the previously mentioned proofs of the individual components. We chose to present this format, because the integral proof would be too long to present in this format. The full implementation can, however, be inspected online on the github.

```
theorem swap: (bfSwap, State.mk [] "" l 0 (x :: y :: r))
  \Longrightarrow State.mk [] "" l 0 (y :: x :: r) :=
  by
    rw [bfSwap]   -- full program
    rw [bfSwapTX] -- first component: >[<+>-]
    apply BigStep.seq
    . apply BigStep.seq
      . apply BigStep.pInc
      . rw [State.applyPInc]
        simp
        exact swap' l (y :: r) x 0
    rw [bfSwapXY] -- second component: >[<+>-]
    apply BigStep.seq
    . apply BigStep.seq
      . apply BigStep.pInc
      . rw [State.applyPInc]
        simp
        exact swap' (x :: l) r y 0
    rw [bfSwapYT] -- third component: <<[>>+<<-]
```

```
20      apply BigStep.seq
21      . apply BigStep.pDec
22      . apply BigStep.seq
23        . apply BigStep.pDec
24        . repeat rw [State.applyPDec]
25          simp
26          have h' := swap'' l r x 0 y
27          rw [Nat.zero_add] at h'
28          assumption
```

Listing 1.18. Proof that the program `>[<+>-]>[<+>-]<<[>>+<<-]` successfully swaps two consecutive memory cells

5 Conclusions

In this paper we gave a new perspective for a programming language that is mostly known for formalising mathematics. We discussed formal verification in the context of formalising programming languages and reasoning about them and about programs written in those languages. We gave a short overview on three possible approaches to specifying semantics. We gave an overview of Lean, a powerful proof assistant, as well as a capable functional programming language with dependent types. We discussed its features and how it can be used for theorem proving. In the end we presented our experiments with giving a big-step semantics for the esoteric Brainf*ck programming language. We also presented some theorems that we proved using the defined big-step semantics. We hope that this work inspires others to consider Lean, or other dependently-typed language for their next project.

References

1. Haskell Language (2024), https://www.haskell.org, online; accessed 4. Mar. 2024
2. Welcome to a World of OCaml (2024), https://ocaml.org, online; accessed 4. Mar. 2024
3. Natural Number Game (Mar 2024), https://adam.math.hhu.de/#/g/leanprover-community/nng4, online; accessed 7. Mar. 2024
4. Brainf*ck - Esolang (Feb 2024), https://esolangs.org/wiki/Brainfuck, online; accessed 8. Mar. 2024
5. Altenkirch, T., McBride, C., McKinna, J.: Why dependent types matter. Manuscript, available online (2005), https://www.cs.nott.ac.uk/~psztxa/publ/ydtm.pdf
6. Avigad, J., De Moura, L., Kong, S.: Theorem proving in lean. Online: https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf (2021)
7. Baanen, A., Bentkamp, A., Blanchette, J., Hölzl, J., Limperg, J.: The hitchhikers guide to logical verification (2023 edition) (2023), https://lean-forward.github.io/hitchhikers-guide/2023, online; accessed 8. Mar. 2024
8. Contributors: Introduction - Functional Programming in Lean (Mar 2024), https://lean-lang.org/functional_programming_in_lean/introduction.html, online; accessed 4. Mar. 2024
9. Contributors to Wikimedia projects: System U - Wikipedia (Nov 2023), https://en.wikipedia.org/w/index.php?title=System_U&oldid=1183340249, online; accessed 5. Mar. 2024
10. Coquand, T.: An analysis of Girard's paradox. Ph.D. thesis, INRIA (1986)
11. Floyd, R.W.: Assigning meanings to programs. In: Program Verification: Fundamental Issues in Computer Science, pp. 65–81. Springer (1993)
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969)
13. Irvine, A.D., Deutsch, H.: Russells paradox (1995)
14. Sanghavi, A.: What is formal verification? EE Times_Asia (2010)

Appendix

A Brainf*ck syntax

We are proving here a complete description of Brainf*ck's syntax [4]:

Symbol	Description
>	Move the pointer to the right
<	Move the pointer to the left
+	Increment the memory cell at the pointer
-	Decrement the memory cell at the pointer
.	Output the character signified by the cell at the pointer
,	Input a character and store it in the cell at the pointer
[	Jump past the matching] if the cell at the pointer is 0
]	Jump back to the matching [if the cell at the pointer is nonzero

B Syntax Notation

```
notation ">"    => Op.pInc
notation "<"    => Op.pDec
notation "+"    => Op.vInc
notation "~"    => Op.vDec
notation "^"    => Op.output
notation ","    => Op.input
notation "[" ops "]" => (Op.brakPair ops)
notation a:50 "_" b:51  => Op.seq a b

-- the following become equivalent definitions of [->+<]
#eval (brakPair (seq vDec (seq pInc (seq vInc pDec))): Op)
#eval ([~_>_+_<]: Op)
```

Listing 1.19. Notations used for the defined operations. The goal of this is to simplify program declaration