A Hoare-logic-based verifier for an imperative DSL

Ronald Huidrom

Dec 22, 2025

Abstract

We design an imperative language and a mechanized Hoare-logic-based verifier for the same. The design is implemented and proved sound in Lean theorem prover. Then we present some verified case studies.

Contents

Chapter 1

Introduction

Consider the following function definition:

```
def divide (x : ℕ) (y : ℕ) (h : y ≠ 0) := x / y
```

This is the usual definition of a division function for natural numbers, except that instead of taking two arguments it takes three. By `x : ℕ`, we mean that `x` is a term having the type `ℕ` (the colon annotates the type). The third argument `h` has the type `y ≠ 0`. This is a proposition and, furthermore, it depends on the value of the term `y`.

This is an instance of *dependent type theory*, in which types may depend on terms. By the Curry–Howard isomorphism and the *propositions-as-types, proofs-as-terms* viewpoint, a term of type `y ≠ 0` is precisely a proof of that proposition. Therefore, the function `divide` can only be evaluated when we supply a proof that `y` is nonzero as its third argument.

A typical evaluation of the function would be:

```
#eval divide 10 2 (by decide)
```

Without going into details (which are delegated to later chapters), `by decide` constructs a term of the required type, in this case a proof of `2 ≠ 0`.

Now consider the following code:

```
inductive NonEmptyList (α : Type) where
| mk : (xs : List α) → xs ≠ [] → NonEmptyList α
```

The keyword `inductive` creates an inductive data type named `NonEmptyList α`. Here, `α` is an arbitrary type (for example, `String`), in which case `NonEmptyList String` denotes the type of non-empty lists of strings.

This inductive type has a single constructor `mk`. Outside the definition, it is referred to as `NonEmptyList.mk`, where the dot denotes a namespace (or simply `.mk` when the namespace can be inferred).

This means that there is exactly one way to construct a term of type `NonEmptyList α`, namely by using the constructor

```
NonEmptyList.mk : (xs : List α) → xs ≠ [] → NonEmptyList α
```

The type of this constructor can be understood as follows: it takes two arguments,

- a term `xs` of type `List α`, and
- a term of type `xs ≠ []`,

and returns a term of type `NonEmptyList α`.

The second argument is particularly interesting: it requires a proof that the first argument `xs` is non-empty. This is again an instance of dependent type theory, since the type `xs ≠ []` depends on the term `xs`.

Now consider the following definition:

```
def NonEmptyList.head {α} : NonEmptyList α → α
| .mk (x :: _) _ ⇒ x
| .mk [] h ⇒ (h rfl).elim
```

This definition implements the well-known `head` function. Its type is

```
NonEmptyList α → α
```

meaning that it takes a non-empty list of elements of type `α` and returns an element of type `α`.

In the function body, we use pattern matching to destruct the possible cases of a term of type `NonEmptyList α`. Since such a term can only be constructed using its constructor (this follows from the inductive definition), the cases listed above are exhaustive.

The first case is when the list has the form `x :: _`, in which case `x` is clearly the first element of the list. (The underscore `_` acts as a wildcard matching the remainder of the list. For example, the list `[1,2,3]` is written as `1 :: [2, 3]`.)

The second case considers the empty list `[]`. However, in this case `h` is a proof that `[]` is non-empty. Examining the type of `NonEmptyList.mk` reveals that `h` has type `[] ≠ []`. This proposition is definitionally equal to `[] = [] → False`. Since `rfl` is a proof of `[] = []`, the expression `h rfl` yields a contradiction, that is, a term of type `False`.

The expression `(h rfl).elim` invokes `False.elim`, which expresses the logical principle that from a contradiction, anything follows. This allows us to construct a term of type `α`, thereby convincing Lean that this case is impossible.

All the code discussed above is written in Lean. We have seen that the type of a function can already guarantee its correctness: impossible cases are ruled out by logical reasoning. By allowing types to depend on runtime values, we eliminate entire classes of runtime errors.

Lean is both a general-purpose functional programming language and an interactive theorem prover, based on the calculus of inductive constructions and dependent type theory.

1.1 Introduction

Consider the following function definition:

```
def divide (x : ℕ) (y : ℕ) (h : y ≠ 0) := x / y
```

The declaration above is the usual integer division operation on natural numbers, except that the function takes an additional *proof* argument. By `x : ℕ` we mean that `x` is a term of type `ℕ`; the third parameter `h` has type `y ≠ 0`, a proposition that depends on the value of `y`. In other words, the function is not just *typed* but *indexed* by a proposition: it is a dependent function.

Under the Curry–Howard correspondence (propositions-as-types, proofs-as-terms), a term of type `y ≠ 0` is exactly a witness (a proof) that `y` is nonzero. Thus `divide` can only be applied when the caller supplies such a witness. For example:

```
#eval divide 10 2 (by decide)
```

Here `by decide` constructs the required proof term (a certificate that `2 ≠ 0`), and the evaluation proceeds without any runtime check.

Now consider this inductive type:

```
inductive NonEmptyList (α : Type) where
| mk : (xs : List α) → xs ≠ [] → NonEmptyList α
```

The `inductive` declaration defines `NonEmptyList α` as the type of non-empty lists over `α`. Its single constructor, `NonEmptyList.mk`, has the dependent type

```
NonEmptyList.mk : (xs : List α) → xs ≠ [] → NonEmptyList α
```

which reads: given a list `xs : List α` and a proof of `xs ≠ []`, construct an element of `NonEmptyList α`. The second argument is a *proof obligation* that enforces the invariant "the list is not empty" at the type level.

We can now implement a safe head projection:

```
def NonEmptyList.head {α} : NonEmptyList α → α
| .mk (x :: _) _ ⇒ x
| .mk [] h ⇒ (h rfl).elim
```

The pattern match is exhaustive because every `NonEmptyList α` is built with `.mk`. In the first clause the payload has the form `x :: _` and we return the head `x`. The second clause is logically impossible: it matches the empty list `[]` while `h` is a proof of `[] ≠ []`. Applying `h` to `rfl` (the canonical proof of judgmental equality `[] = []`) yields a term of type `False`; invoking `(h rfl).elim` eliminates the contradiction (by `False.elim`) and thus produces a term of the expected return type. In short, the impossible branch is discharged by logical contradiction rather than by a runtime error.

These examples illustrate a central design point of dependent-type proof assistants: invariants that would otherwise require runtime checks are encoded in the type system and discharged by supplying proof terms. By making propositions first-class and allowing types to depend on terms, Lean (based on the Calculus of Inductive Constructions) moves many correctness obligations from runtime into the type checker. The payoff is strong: programs and proofs can be composed so that certain classes of runtime failures are provably impossible.

Chapter 2

The Imperative DSL

We design a minimal imperative language sufficient to express classical algorithms such as binary search. The language is intentionally small, allowing us to focus on the logical foundations of verification rather than engineering concerns.

2.1 Operational semantics

2.1.1 Big-step (natural) operational semantics

We adopt a *big-step* (also called *natural*) operational semantics for our imperative DSL. Intuitively, a big-step judgment

$$\langle c, s\rangle \Downarrow s'$$

asserts that executing command c in initial state s *terminates* and produces final state s'. The big-step semantics is given by a small collection of inference rules (one rule per syntactic form); each rule is a specification of how one execution step (or one inference of a case) yields a final state. In contrast to small-step semantics, the big-step form directly relates whole computations to their results and thus is particularly convenient when proving partial-correctness properties (such as soundness of Hoare logic) and when constructing verification condition generators.

Below we present the operational rules in standard inference–rule form. For clarity we write states as s, s', s'', commands as c, c_1, c_2, and use the notation $\langle c, s\rangle \Downarrow s'$ for the big-step relation.

Inference rules (big-step).

(skip) The `skip` command does nothing.

$$\frac{}{\langle \texttt{skip}, s\rangle \Downarrow s}$$

(assign) Assignment updates the state at the named variable with the value of the expression.

$$\frac{}{\langle x := e, s\rangle \Downarrow s[x \mapsto \mathsf{eval}(e, s)]}$$

(seq) Sequential composition executes the first command to an intermediate state and then the second.

$$\frac{\langle c_1, s\rangle \Downarrow s' \qquad \langle c_2, s'\rangle \Downarrow s''}{\langle c_1; c_2, s\rangle \Downarrow s''}$$

(if_true) If the condition holds, execute the then-branch.

$$\frac{\mathsf{evalB}(b, s) \qquad \langle c_{then}, s\rangle \Downarrow s'}{\langle \texttt{if } b \texttt{ then } c_{then} \texttt{ else } c_{else}, s\rangle \Downarrow s'}$$

(if_false) Otherwise, execute the else-branch.

$$\frac{\neg\mathsf{evalB}(b, s) \qquad \langle c_{else}, s\rangle \Downarrow s'}{\langle \texttt{if } b \texttt{ then } c_{then} \texttt{ else } c_{else}, s\rangle \Downarrow s'}$$

(while_false) If the loop guard is false initially, the loop terminates immediately.

$$\frac{\neg\mathsf{evalB}(b, s)}{\langle \texttt{while } b \texttt{ do } c, s\rangle \Downarrow s}$$

(while_true) If the guard is true, execute one iteration and continue.

$$\frac{\mathsf{evalB}(b, s) \qquad \langle c, s\rangle \Downarrow s' \qquad \langle \texttt{while } b \texttt{ do } c, s'\rangle \Downarrow s''}{\langle \texttt{while } b \texttt{ do } c, s\rangle \Downarrow s''}$$

The rules above are the standard natural semantics for the command language: they are *compositional* (the conclusion of a rule is determined from the conclusions of its premises) and they explicitly witness termination. In particular, the **while_true** rule unrolls one iteration of the loop and then recurses, so a proof of a loop execution is finite precisely when the loop terminates; non-terminating executions produce no finite derivation and therefore no judgment $\langle c, s\rangle \Downarrow s'$.

2.1.2 The Lean representation: the inductive relation `Exec`

In the formal development we encode the above semantics as an inductive relation in Lean named `Exec`. The relation has the (dependent) type

```
Exec : Cmd → State → State → Prop
```

so that a value (term) of type `Exec c s s'` is a formal derivation witnessing $\langle c, s\rangle \Downarrow s'$. Below is the Lean declaration:

```
inductive Exec : Cmd → State → State → Prop
| skip : Exec Cmd.skip s s
| assign : Exec (Cmd.assign x e) s (s.update x (e.eval s))
| seq : Exec c₁ s s' → Exec c₂ s' s'' → Exec (Cmd.seq c₁ c₂) s s''
| ite_true : b.eval s → Exec ct s s' → Exec (Cmd.ite b ct ce) s s'
| ite_false : ¬ b.eval s → Exec ce s s' → Exec (Cmd.ite b ct ce) s s'
| while_false : ¬ b.eval s → Exec (Cmd.while b I c) s s
| while_true : b.eval s → Exec c s s' → Exec (Cmd.while b I c) s' s'' →
              Exec (Cmd.while b I c) s s''
```

2.1.3 How to build `Exec` terms

Each constructor of `Exec` corresponds exactly to one of the inference rules above. A proof object of type `Exec c s s'` is a finite tree whose internal nodes are applications of constructors and whose leaves are applications of leaf constructors (e.g., `skip` and `assign`). We now describe the constructors and how to form terms in practice.

`skip`. The constructor `Exec.skip` (named `skip` in the inductive block) is a leaf constructor. For any state `s` the term `Exec.skip` inhabits `Exec Cmd.skip s s`. Thus the trivial derivation $\langle \texttt{skip}, s\rangle \Downarrow s$ is witnessed by the canonical term `Exec.skip`.

`assign`. The assignment constructor is also a leaf: to prove `Exec (Cmd.assign x e) s s'`, the constructor requires that `s' = s.update x (e.eval s)`. Concretely, for any state `s` the term `Exec.assign` is a witness of

```
Exec (Cmd.assign x e) s (s.update x (e.eval s))
```

No additional premises are needed because the evaluation of the right-hand expression $\mathsf{eval}(e, s)$ is computed meta-level (by the definitional evaluator `e.eval s`).

`seq`. The sequential composition constructor is *binary* and nontrivial. If we have:

```
h_1 : Exec c₁ s s'  and  h_2 : Exec c₂ s' s''
```

then the term `Exec.seq h_1 h_2` has type

```
Exec (Cmd.seq c₁ c₂) s s''
```

Thus, to build a proof that a sequence terminates in state `s''`, we must provide a proof for the first component reaching an intermediate state `s'` and a proof that the second component reaches `s''` from `s'`.

`ite_true` / `ite_false`. Conditionals have two constructors corresponding to the two branches. To construct a term of type `Exec (Cmd.ite b ct ce) s s'` in the *true* case we must provide:

```
hb : b.eval s  and  ht : Exec ct s s'.
```

Then `Exec.ite_true hb ht` is a witness of the execution along the then-branch. The false case is analogous and uses `Exec.ite_false`.

`while_false`. If the guard is false in the starting state, then the loop terminates immediately and the constructor `Exec.while_false` (given a proof of $\neg b.\mathit{eval}\ s$) builds a term of type `Exec (Cmd.while b I c) s s`.

`while_true`. The `while_true` constructor implements one unrolling of the loop. To construct a witness of type

```
Exec (Cmd.while b I c) s s''
```

we must provide:

1. a proof `hb : b.eval s` that the guard holds at the current state,
2. a proof `hbody : Exec c s s'` that one iteration executes from `s` to `s'`, and
3. a proof `hrest : Exec (Cmd.while b I c) s' s''` that the remainder of the loop executes from `s'` to `s''`.

One then forms `Exec.while_true hb hbody hrest`. Because the final premise is itself an `Exec` judgment, proofs of loop executions are nested finite trees that correspond to successive unrollings. A complete derivation is finite exactly when the loop terminates in a finite number of iterations.

2.1.4 Key meta-properties and remarks

1. **Derivations are finite trees.** Since `Exec` is declared *inductively*, every inhabitant (proof term) is a finite, well-founded tree built from the constructors. This enforces the standard semantic reading: *a big-step derivation exists precisely when the program terminates*; nonterminating computations have no finite `Exec` witness.
2. **Not recursion in the proof-theoretic sense.** Although several constructors (notably `seq` and `while_true`) mention `Exec` in their premises, this is inductive recursion — not unrestricted recursion — and Lean checks that the definition is well-formed. Inductive definitions produce the least fixed point; hence all constructed proofs are finite by construction.
3. **Constructing proofs vs. running programs.** A term of type `Exec c s s'` is a *certificate* (derivation) rather than an imperative execution performed by the Lean VM. When the program consists of pure assignments and conditionals evaluated by definitional evaluation of subexpressions, the premises used in constructors will often be inhabited by canonical constructors (e.g., `Exec.assign`) without additional work; however, for loops a proof requires exhibiting a finite sequence of iterations (or a loop invariant if reasoning by Hoare logic).
4. **Partial correctness and Hoare triples.** The big-step relation provides the semantic basis for partial correctness. We define the semantic Hoare triple by

$$\{P\}\ c\ \{Q\} \quad \overset{\text{def}}{\iff} \quad \forall s\, s',\ \langle c, s\rangle \Downarrow s' \ \wedge\ P(s)\ \Rightarrow\ Q(s').$$

   In Lean notation this is written directly as:

```
def Hoare (P : Assertion) (c : Cmd) (Q : Assertion) : Prop :=
  ∀ s s', Exec c s s' → P s → Q s'
```

   Soundness of a proof system (e.g. Hoare rules) or of a verification condition generator is proved with respect to this semantic definition.
5. **Determinism (optional remark).** For typical deterministic expression evaluation and the rules above, the big-step relation is deterministic: if $\langle c, s\rangle \Downarrow s_1$ and $\langle c, s\rangle \Downarrow s_2$ then $s_1 = s_2$. Determinism is convenient for equational reasoning and can be proved by induction on derivations; it is not strictly required for the verification machinery, but it simplifies many meta-proofs.
6. **Proof engineering.** In practice, one seldom constructs large `Exec` trees by hand; instead, we either:
   - reason about them abstractly using induction on derivations, or
   - avoid explicit derivations by using a verification condition generator (VCG) whose correctness is proved against the `Exec` relation.

   The VCG produces logical obligations (assertions) that imply the existence of an `Exec` derivation; thus the VCG plus discharged obligations yields a semantic guarantee à la the Hoare triple above.

2.1.5 A short concrete construction example

Consider the trivial program that performs two assignments in sequence:

```
def c1 : Cmd := Cmd.assign "x" (AExp.const 1)
def c2 : Cmd := Cmd.assign "y" (AExp.const 2)
def prog : Cmd := Cmd.seq c1 c2
```

Let `s` be an arbitrary state and define `s1 := s.update "x" 1` and `s2 := s1.update "y" 2`. Then we obtain the following concrete derivation terms:

```
# let h1 : Exec c1 s s1 := Exec.assign
# let h2 : Exec c2 s1 s2 := Exec.assign
# let hprog : Exec prog s s2 := Exec.seq h1 h2
```

In words: both assignments are leaf derivations (inhabited by `Exec.assign`); composing them with `Exec.seq` yields a derivation of the full sequence.

Concluding remark. The inductive relation `Exec` and its finite derivations give a transparent, machine-checkable account of terminating executions. It is the semantic backbone for correctness statements and for proving soundness of the verification infrastructure (VCG or Hoare proof system) that we build on top of the DSL.

The verification condition generator is implemented as a structurally recursive function over the abstract syntax of the language, entirely within Lean's definitional fragment, without the use of metaprogramming.