A Hoare-logic-based verifier for an imperative DSL

Ronald Huidrom

Dec 12, 2025

Abstract

We look at the Jones polynomial, its construction, applications and generalizations. In particular, we study the braid groups, its representations and arrive at a construction of the Jones polynomial. We discuss proofs of the Tait conjectures. Then we construct Khovanov homology whose graded Euler characteristic is the Jones polynomial.

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

Background

Chapter 3

Lezu: An Imperative DSL

Chapter 4

Hoare Logic