The Lean Proof Assistant

Seminar Deductive and Interactive Verifiers
FG Software Engineering (Prof. Dr. Reiner Hähnle)
Supervisor: Daniel Drodt

Fynn Godau

March 13, 2026

TECHNISCHE
UNIVERSITÄT
DARMSTADT

1. Overview

Proof assistant use cases can be grouped as follows: firstly, to formalize and prove mathematical statements, either as a tool to verify an existing proof in natural language or to help find new proofs. Secondly, by phrasing a property of a program as such a mathematical statement, proof assistants can also be used to verify program correctness, i. e. termination behavior and adherence to a specification, helping programmers find and eliminate bugs that violate the expected behaviors.

Lean 4 is a functional programming language, proof language and proof assistant, and supports both use cases. From functional programming, it inherits a number of concepts, like immutability of values and inductive types, of which we provide a reminder in Section 2. As a proof language, it implements the Calculus of Inductive Construction (CIC), which we introduce in Section 3; this blends in with its properties of a proof assistant, which is strengthened by its interactive proof mode, called tactic mode, that we introduce in the same section. As a demonstration, in Section 4, we discuss two different implementations of the `insertionSort` algorithm, and how its correctness can be proven in Lean (or what challenges arise). Section 5 considers the `mathlib` library, a comparison to two other proof assistants, and learning resources for Lean. Finally, the conclusion in Section 6 will summarize Lean's strengths.

2. Functional Programming in Lean

While Lean can be used as a fully functional programming language, and is used for software modeling and verification in practice [1], its mathematics-focused use cases have a much more active community [2]. For this reason, we only briefly review some important concepts from functional programming here.

Inductive types refers to sum types, defined in terms of an enumeration of possible constructors. Values of an inductive type can be destructed by means of pattern matching to discriminate which constructor was used to build the value, and to re-gain access to the "inputs" of the constructor. This is well-known from other languages such as Haskell or Scala. We show a common example as a reminder:

```
inductive IntList
  | nil : IntList
  | cons (h : Int) (t : IntList) : IntList

def sum (xs : IntList) : Int := match xs with
  | IntList.nil      => 0
  | IntList.cons h t => h + sum t
```

The syntax may remind one of Haskell as well: a function and the parameters it is applied with are separated by a space. Likewise, function signatures are (almost) always denoted as curried, and accordingly, function arrows ("→") associate to the right. This means that a function $\alpha \times \beta \to \gamma$ would be written as $\alpha \to \beta \to \gamma$ or, equivalently, $\alpha \to (\beta \to \gamma)$.

3. Calculus of Inductive Construction

The Calculus of Construction (CoC), a logical calculus that formalizes constructive proofs in a deductive manner, is attributed to Coquand and Huet [3]. Its extension to the Calculus of Inductive Construction (CIC) is due to Coquand, Paulin-Mohring [4], and Pfenning [5]; the difference can be described as adding inductive types (cf. Section 2) to the type system. Like Rocq, which we discuss in Section 5.3, Lean is an implementation of the CIC.

3.1. The Power of the Type-Checker

Before we consider how Lean handles general mathematical proofs, let us consider only proofs of the following type: "Given a function signature, show that there exists a total function with this signature." In a type-checked programming language, statements of this type—with respect to this language's type system—can be proven by giving a concrete implementation of the desired signature, then validating it using the type-checker.

There are two noteworthy caveats in this description: firstly, any such automatically verified proof must necessarily rely on the soundness of the type checker. Secondly, and more interestingly, the type system itself plays an important role: for example, a type system like Java's reference types, where `null` is a member of every type, renders the question completely uninteresting (as there are no empty types, hence every function can be implemented as the one always returning `null`). To the contrary, Lean's type system is expressive enough to meaningfully pose such questions:

```
variable (α β : Type)
example : α → (α → β) → β := fun x y => y x
example : Nat → β := sorry
```

In place of the `sorry` keyword, Lean's type checker would expect us to provide a function of the declared type `Nat→β` before the proof is considered complete, where `Nat` corresponds to $\mathbb{N}$. However, we don't know enough about the type β—it could be the empty type, and by common mathematical understanding, there is no total function of type $\mathbb{N} \to \emptyset$.[1]

3.2. Curry-Howard-Isomorphism

Having understood how implementations of functions prove the existence of a function of their type, we can now turn towards a concept that allows us to phrase general mathematical statements in terms of this framework. This concept, the *Curry-Howard-Isomorphism*, models arbitrary mathematical statements (propositions) as types, and proofs of a proposition as a value of that type. It follows that unprovable propositions are empty types, while there *are* concrete instances for every provable proposition—we call such types *inhabited*.

By the intuitive meaning of their names, this implies there to be some value of type `True`, while it should not be possible to yield a value of type `False`. And indeed, this is the case in Lean:[2]

```
example : True  := True.intro
example : False := sorry -- no proof can be filled in here
```

We now consider how more complex propositions can be written down as types.

Implication We can think of an implication $p \to q$ as a function that takes a proof for p and maps (or "translates") it to a proof for q. Under this interpretation, the ordinary function arrow "`→`" already models the implication. No separate meaning for the symbol "`→`" is needed: the change of perspective happens only in the mind of the user, not in Lean's code. For example, it doesn't matter whether `p` and `q` are propositions or data types in the following snippet:

```
example : p → (p → q) → q := fun p => (fun pq => pq p)
```

Logical connectives Logical "and" corresponds to a product type `And` (like $\times$, "provide a proof for both this and that"), while logical "or" corresponds to a sum type `Or` (inductive type, "prove either this or that"). Logical "not" is denoted in terms of an implication by rewriting as follows:

$$\neg p \iff \neg p \vee \texttt{False} \iff p \to \texttt{False}$$

The intuitive understanding is that p being false is equivalent to being able to derive a contradiction from a proof that p is true, where "contradiction" is denoted as a value of the known-empty type `False`. The following example demonstrates `And` together with logical "not" (in terms of this rewriting). The pointy brackets decompose the "and" statement into the proofs for its two parts.

```
example : (And p (p → False)) → False := fun ⟨ a, b ⟩ => b a
```

Quantifiers An existentially quantified proposition is modeled using a predicate `α→Prop`; instances (proofs) of such proposition consist of a witness and a proof that the predicate holds for the given witness.

Universal quantification is expressed as a function. Consider quantification over all values `x` of type `α`: such an `x` must be mapped to a proof for the desired proposition. For this to work, not only the proof, but also the proposition must be allowed to depend on the value `x`. This is possible thanks to *dependent types*, a feature of the type system. Dependent types allows us to denote statements like this (where `rfl` denotes a proof for reflexivity of "`=`"):

```
example : (x : α) → x = x := fun x => rfl
```

Accordingly, the "`∀`" symbol is defined such that the following two lines express exactly the same term by definition:

```
∀ x : α, p
(x : α) → p
```

Lean also defines the symbols `∧`, `∨`, `¬`, `∃` as syntactic sugar according what was discussed here.

One benefit of the Curry-Howard-Isomorphism is that it allows a program and its proofs to share a single syntax, and to live in the same type system. This will become apparent when inspecting the code and proofs of Section 4.

[1] You may notice that, even if β was somehow guaranteed not to be empty by the type system, we still couldn't implement the function for lack of a concrete value of type β. Hold that thought for Section 3.3.

[2] In reality, the technical details are a bit different: while Lean does typecheck the demonstrated code, it is not considered a *sound* proof because the `sorry` keyword is used, which uses the unsound axiom that every type is inhabited.

3.3. Classical Logic

The Calculus of Construction, and in particular how logical "not" is defined therein, leads to the inability to show some propositions that are easily provable classically. One such example is "double negation elimination":

```
example : ¬¬p → p := sorry
```

We rewrite the proposition using the definition of "¬":

```
example : ((p → False) → False) → p := sorry
```

To prove this statement, we would need to define a function that takes a function of type `(p→False)→False` as input and produces a proof for (i. e. a value of type) `p`. Such function doesn't exist, as nothing is given that would produce the desired outcome value.

To introduce classical reasoning to the CoC, we need to use additional axioms. The `axiom` keyword constructs a value of a specified type "out of thin air"—naturally, careless introduction of axioms can easily lead to unsound reasoning, but Lean ships with three core axioms that are "known to be consistent" [6, Section 8].

The most relevant of the three is the axiom of choice: "For every nonempty type, there is a function that picks an arbitrary element." It bridges the gap in constructive logic between "knowing a set not to be empty" and "knowing a concrete element of a set".

```
axiom choice (α : Sort u) : Nonempty α → α
```

Taken together with Lean's other two core axioms, one first derives `(p∨¬p)` (law of excluded middle[3]) and then `¬¬p→p`.

3.4. Tactic Mode

Writing down a function of the required type directly, like shown in the previous examples, is only one way of implementing CIC proofs in Lean. Additionally, Lean offers a second style called *tactic mode*, where users write a program that generates such function in the background, signaled using the `by` keyword. In a typical Lean setup, the user is interactively shown the current tactic mode proof state in the editor's sidebar, offering a proof-goals based approach: the user can advance towards the goals step-by-step by invoking programs called *tactics*, which on the user-facing side manipulate the proof goal, while in the background constructing the necessary parts of the underlying function. We give two examples of powerful tactics:

```
theorem linearEq : (a b c : Int) → (a - b < c) → (c + b > a) := by omega
theorem doubleNegationEqEm : (¬¬p ↔ p) ↔ (p ∨ ¬p) := by grind
```

The first tactic, `omega`, is a linear equation solver, and spares the user the manual application of rules related to the involved inequalities. The second tactic, `grind`, is very potent: it processes local definitions, as well as proofs for propositions included in Lean, in an attempt to automatically prove the desired statement. As a parameter, `grind` can also be given a list of additional definitions (axioms, function definitions, theorems) that it should consider. [6, Section 14.5]

Lean's original creator de Moura and Ullrich attribute Lean's "success [...] primarily [...] to its extensibility capabilities and metaprogramming framework" [8]. *Metaprogramming* refers to the ability to extend Lean's core components, like syntax and tactics, with user-provided definitions. Lean makes available a separate guidebook to serve as an introduction to metaprogramming [9].

4. Implementation and Correctness of `insertionSort`

In this section, we demonstrate Lean's capabilities by example of the `insertionSort` algorithm. First, we specify the algorithm itself as two functions: one helper function that inserts a new item into an already sorted list (`insertSorted`), and one main function that recursively re-inserts every item using the helper function. For the sake of simplicity, we restrict our implementation to lists of integers. The function here was shortened thanks to the solution that Oswald published on the web [10].

```
def IntList := List Int

def insertSorted (i : Int) : IntList → IntList
```

[3] This proof, known as Diaconescu's theorem [7], is long and sophisticated. To have Lean print the proof and its axioms:

```
#print Classical.em
#print axioms Classical.em
```

Manually deriving double negation elimination from `Classical.em` is a fun exercise for Lean beginners (the solution is at most three lines long), but the `grind` tactic can also handle it (see Section 3.4).

```
    | []      => [i]
    | y :: ys => if i < y then
        i :: y :: ys
      else
        y :: (insertSorted i ys)

def insertionSort : IntList → IntList
  | []      => []
  | x :: xs => insertSorted x (insertionSort xs)
```

4.1. `insertionSort` Returns a Sorted List

To talk about whether a list is sorted, we first need to define the property of "sortedness". In Lean, we do this by defining a function that takes a list as input and returns the proposition that "the given list is sorted", i. e., the function evaluates to a corresponding formula.

```
def sortedList : IntList → Prop
  | []           => true
  | _ :: []      => true
  | a :: b :: cs => (a ≤ b) ∧ sortedList (b :: cs)
```

As a first step, we show that `insertSorted` behaves as expected: provided that the list it receives as a parameter is already sorted, the list it returns shall be sorted as well. We can leave the details of the proof to the `grind` tactic; we only need to specify the structure of the proof: structural induction over the input list with the `induction` tactic, case distinction using `by_cases` by whether the new item is to be placed at the front of the list or not, and a pattern matching over the remaining list length. We pass `grind` the two definitions `sortedList` and `insertSorted`, so that it can unfold them internally.

```
theorem iSRetainsSorting (xs: IntList) (invariant : sortedList xs)
: ∀ x : Int, sortedList (insertSorted x xs) := by
  intro x
  induction xs with
  | nil => rfl
  | cons a bs th =>
    by_cases l : x < a
    · grind [sortedList, insertSorted]
    · match bs with
        | []     => grind [sortedList, insertSorted]
        | b :: t => grind [sortedList, insertSorted]
```

For the function `insertionSort`, we perform another induction over the list length. In the inductive step, we use the fact about `insertSorted` from the previous theorem (manually, without resorting to `grind`).

```
theorem iSIsSorted : ∀ xs : IntList, sortedList (insertionSort xs) := by
  intro xs
  induction xs with
  | nil => simp [insertionSort, sortedList]
  | cons h t ih =>
    unfold insertionSort
    exact iSRetainsSorting (insertionSort t) ih h
```

4.2. `insertionSort` Returns a Permutation of the Input List

While we now know that the result of `insertionSort` is always sorted, we still need to establish a relationship to its input, that is, we want to show that the output list is a permutation of the input. The permutation relation over lists is predefined in Lean as "~" in the `List` namespace [11]; it has four constructors, which we show here in simplified form:

```
inductive ~ : IntList → IntList → Prop
  | nil : [] ~ []
  | cons (x : Int) (l₁ l₂ : IntList) : l₁ ~ l₂ → (x :: l₁) ~ (x :: l₂)
  | swap (x y : Int) (l : IntList) : (y :: x :: l) ~ (x :: y :: l)
  | trans (l₁ l₂ l₃ : IntList) : l₁ ~ l₂ ∧ l₂ ~ l₃ → l₁ ~ l₃
```

The application of these constructors is the core of our proofs for the following theorems, which is shown in Appendix A.

```
open List
theorem insertSortedPerm (x : Int) (xs : IntList) : (x :: xs) ~ (insertSorted x xs) := sorry
theorem iSPerm : ∀ xs : IntList, xs ~ insertionSort xs := sorry
```

4.3. An Alternative Solution

Lean's documentation includes an implementation of `insertionSort` using arrays instead of lists [12, Section 10.6]. As this leads to a very different solution, we will highlight a few key differences here. The source code is also shown in Appendix B for reference.

Manual termination proof Every function in Lean requires a termination proof (unless appropriate modifiers to the contrary are added). Our implementation above is simple enough that Lean can derive a proof automatically. The array-based implementation, however, is more complex and requires the user to manually specify and prove correctness of a *decreasing measure* to show that only finitely many recursive steps are performed. As a part of this, the solution from the documentation proves that input and output array of the inner loop function have the same size.

In-place algorithm Lean code can be compiled as standalone programs. During a program run, the array swap operations used in the array-based algorithm are performed in-place (instead of by creating a new copy as usual) if this optimization is possible, i. e. if there are no remaining references to the previous copy of the array. The documentation demonstrates this by example, but is unable to provide a formal proof, as the "in-place property" is inaccessible to Lean's reasoning.

No correctness proof The algorithm shown in the appendix does not comprise correctness proofs, and the documentation makes no mention of it. Generally, proving correctness appears more difficult for algorithms that work on arrays compared to lists, due to the fact that Lean requires each array access to accompany a proof which demonstrates that the access is in bounds. As such, a proposition that talks about two positions in an array must itself already contain proofs that these positions exist, making the "sortedness" property difficult even to just write down.

5. Lean Ecosystem and Other Proof Assistants

In this section, consider `mathlib` as a core part of Lean's ecosystem, briefly introduce Isabelle and Rocq as two other proof assistants, and provide pointers on how Lean can be learned.

5.1. `mathlib`

Given a complex proposition, it can be hard to determine whether a given formalization matches the intended statement or not, and consequently, whether a given proof really shows the intended theorem. Hence, it is advantageous to use a common library that provides a canonical, peer-reviewed definition, which the library `mathlib` provides. Developed as a community project, it defines more complex mathematical concepts and propositions than those included in Lean itself; many community members aim for it to reach "modern, research-level mathematics" [2].

Many well-known theorems have already been formalized in `mathlib`, and can be used as "building blocks" in attempts to formalize or find new proofs. One prominent result of this effort is that a proof for the independence of the continuum hypothesis from ZFC was first formalized in Lean by Han and van Doorn in 2020, using `mathlib` [13], [14].

However, the library is an ongoing work-in-progress, and some areas of mathematics haven't seen much progress yet. As a point of reference, the project makes available a list of concepts that are considered to be "undergraduate" level but not yet available in `mathlib` [15].

5.2. Isabelle

The *Isabelle* system is developed in a modular way, allowing users to choose the logic they want to formalize in [16]. While *Isabelle/HOL*, "a version of classical higher-order logic resembling that of the HOL System" [17], is considered "the most widely used system" [16], some significant proofs have also been formalized in other systems. For example, the aforementioned proof that the continuum hypothesis is independent of ZFC was formalized in 2022, two years after its `mathlib` counterpart, by Gunther, Pagano, Sánchez Terraf and Steinberg in *Isabelle/ZF* [18], which is "a formulation of Zermelo-Fraenkel set theory on top of FOL" [17].

For details on how proofs are denoted in Isabelle/HOL compared to a CIC-based logic, see the comparison paper by Yushkovskiy [19].

5.3. Rocq

Rocq is a prominent, direct alternative to Lean, since it is also an implementation of the Calculus of Inductive Construction. Not only are they based in the same logic, the axioms that they respectively add on top of the CIC are also very similar:

> "[A]ll of Coq's[4] axioms taken together are implied by Lean's axioms, and the converse is true except for definitional proof irrelevance[5] and a computation rule for quotient types." [21]

Carneiro [21] also describes other differences and deviating design choices on a technical level. One such difference is that contrary to Rocq, which seems to consider this an important type-theoretic property, Lean's type system lacks the property of subject reduction [22], which can be summarized to mean "a term's type is always the same as the type of its evaluation".

On the technical level that these differences play out, deep type-theoretic knowledge is required to properly understand them, making it hard to grasp what, if any, the overall meaningful distinction even is. One blog post concludes "that what sets them apart are, in essence, cultural differences" [23]: Ramachandra therein describes that "Rocq [...] has always been very particular about sound type theoretic foundations", while Lean more pragmatically "seems to have taken a top-down approach, by focusing on writing real proofs as quickly as possible" [23]. Therefore, we leave the differences between these tools at a rather high-level overview.

5.4. Learning Lean

Lean provides introductory guidebooks from different angles: functional programming [12], theorem proving [24], and mathematics [25]. Section 3 can be understood as a condensed overview of the second, which introduces the concepts in depth and provides many more examples, without depending on subject-specific prior knowledge.

As an additional resource, a team at Heinrich Heine University Düsseldorf independently hosts a website called "Lean Game Server" [26], providing an interactive tutorial for tactics mode in Lean in terms of multiple sets of examples, like natural numbers and set theory.

Both resources can be considered appropriate starting points for beginners interested in learning Lean.

6. Conclusion

Lean is, overall, a powerful tool, allowing its users to write surprisingly clean proofs. For users with some background in functional programming, the concept of "programming by the target type" should already be familiar and can, after an initial period of mental adjustment to the Curry-Howard-Isomorphism, be directly applied to theorem proving in Lean. Compared to imperative languages, proofs regarding programs written in Lean do not need to consider mutability, avoiding many heap-related problems (e. g. the framing problem) that are inherent to verification tools based in mutable languages.

A Rocq user might not find immediate value in learning to use the Lean language, unless they seek to benefit from the efforts of the `mathlib` project.

While there is no compelling reason to use Lean as just a functional programming language—compared to long-standing alternatives with established software ecosystems for practical use—, it is a suitable tool for beginners seeking to learn their first proof assistant, and the "Theorem Proving in Lean 4" guidebook [24] is written such that it can support effective learning. However, as became apparent in Section 5.3, a much deeper type-theoretic understanding than could be conveyed here is required to fully understand the design choices that influence Lean's logical core.

References

[1] K. Hietala and E. Torlak, *Lean into verified software development*, 2024. [Online]. Available: `https://aws.amazon.com/blogs/opensource/lean-into-verified-software-development/` (visited on 03/11/2026).

[2] "The Lean mathematical library," *CoRR*, vol. abs/1910.09336, 2019. DOI: `10.48550/arXiv.1910.09336`.

[3] T. Coquand and G. P. Huet, "The calculus of constructions," *Inf. Comput.*, vol. 76, no. 2/3, pp. 95–120, 1988. DOI: `10.1016/0890-5401(88)90005-3`.

[4] T. Coquand and C. Paulin, "Inductively defined types," in *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, P. Martin-Löf and G. Mints, Eds., ser. Lecture Notes in Computer Science, vol. 417, Springer, 1988, pp. 50–66. DOI: `10.1007/3-540-52335-9_47`.

[4] Coq has since been renamed to Rocq.

[5] For definitional proof irrelevance, there has been some consideration how it can be added to Rocq without violating any type-theoretic properties that Rocq users desire [20].

[5] F. Pfenning and C. Paulin-Mohring, "Inductively defined types in the calculus of constructions," in *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings*, M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, Eds., ser. Lecture Notes in Computer Science, vol. 442, Springer, 1989, pp. 209–228. DOI: 10.1007/BFB0040259.

[6] *The Lean language reference*, 2026. [Online]. Available: https://lean-lang.org/doc/reference/4.29.0-rc6/ (visited on 03/11/2026).

[7] R. Diaconescu, "Axiom of choice and complementation," *Proc. American Mathematical Society*, vol. 51, no. 1, pp. 176–178, 1975, ISSN: 00029939, 10886826. DOI: 10.2307/2039868.

[8] L. de Moura and S. Ullrich, "The Lean 4 theorem prover and programming language," in *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, A. Platzer and G. Sutcliffe, Eds., ser. Lecture Notes in Computer Science, vol. 12699, Springer, 2021, pp. 625–635. DOI: 10.1007/978-3-030-79876-5_37.

[9] *Metaprogramming in Lean 4*. [Online]. Available: https://leanprover-community.github.io/lean4-metaprogramming-book/ (visited on 03/11/2026).

[10] J. Oswald, *Proving the correctness of insertion sort in Lean4*, 2024. [Online]. Available: https://jamesoswald.dev/posts/lean4-insertion-sort/ (visited on 03/03/2026).

[11] *List.Perm*. [Online]. Available: https://lean-lang.org/doc/api/Init/Data/List/Basic.html#List.Perm (visited on 03/11/2026).

[12] D. T. Christiansen, *Functional programming in Lean*, 2023. [Online]. Available: https://leanprover.github.io/functional_programming_in_lean/ (visited on 03/11/2026).

[13] J. M. Han and F. van Doorn, "A formalization of forcing and the unprovability of the continuum hypothesis," in *10th International Conference on Interactive Theorem Proving, ITP 2019, Portland, OR, USA, September 9-12, 2019*, J. Harrison, J. O'Leary, and A. Tolmach, Eds., ser. LIPIcs, vol. 141, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 19:1–19:19. DOI: 10.4230/LIPICS.ITP.2019.19.

[14] J. M. Han and F. van Doorn, "A formal proof of the independence of the continuum hypothesis," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, J. Blanchette and C. Hritcu, Eds., ACM, 2020, pp. 353–366. DOI: 10.1145/3372885.3373826.

[15] *Missing undergraduate mathematics in mathlib*. [Online]. Available: https://leanprover-community.github.io/undergrad_todo.html (visited on 03/11/2026).

[16] M. S. Nawaz, M. Malik, Y. Li, M. Sun, and M. I. U. Lali, "A survey on theorem provers in formal methods," *CoRR*, vol. abs/1912.03028, 2019. DOI: 10.48550/arXiv.1912.03028.

[17] *Isabelle documentation*. [Online]. Available: https://isabelle.in.tum.de/documentation.html (visited on 03/12/2026).

[18] E. Gunther, M. Pagano, P. S. Terraf, and M. Steinberg, "The independence of the continuum hypothesis in Isabelle/ZF," *Arch. Formal Proofs*, vol. 2022, 2022. [Online]. Available: https://www.isa-afp.org/entries/Independence%5C_CH.html (visited on 03/11/2026).

[19] A. Yushkovskiy, "Comparison of two theorem provers: Isabelle/HOL and Coq," *CoRR*, vol. abs/1808.09701, 2018. DOI: 10.48550/arXiv.1808.09701.

[20] G. Gilbert, J. Cockx, M. Sozeau, and N. Tabareau, "Definitional proof-irrelevance without K," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 3:1–3:28, 2019. DOI: 10.1145/3290316.

[21] M. Carneiro, *The type theory of Lean*, 2019. [Online]. Available: https://github.com/digama0/lean-type-theory/releases/download/v1.0/main.pdf (visited on 02/26/2026).

[22] *What is the subject reduction debate?* [Online]. Available: https://leanprover-community.github.io/archive/stream/113488-general/topic/What.20is.20the.20subject.20reduction.20debate.3F.html (visited on 03/12/2026).

[23] R. Ramachandra, *Lean versus Rocq: The cultural chasm*, 2020. [Online]. Available: https://artagnon.com/logic/leancoq (visited on 02/26/2026).

[24] J. Avigad, L. de Moura, S. Kong, and S. Ullrich, *Theorem proving in Lean 4*. [Online]. Available: https://lean-lang.org/theorem_proving_in_lean4/ (visited on 03/11/2026).

[25] J. Avigad and P. Massot, *Mathematics in Lean*. [Online]. Available: https://leanprover-community.github.io/mathematics_in_lean/ (visited on 03/11/2026).

[26] *Lean game server*. [Online]. Available: https://adam.math.hhu.de/ (visited on 03/11/2026).

[27] F. Godau, *The Lean proof assistant: Sample code*, 2026. [Online]. Available: https://cs.fynngodau.de/lean/code/ (visited on 03/11/2026).

A. Source Code for List-based `insertionSort`

For convenience, the source code shown here is additionally available for download [27, SortingList.lean].

The solution shown here was made more compact thanks to the solution published on the web by Oswald [10].

```
-- TYPES --
-- we use a List on Ints for the sake of readability
-- we could abstract to other types a that have a complete linear order [Ord a]
def IntList := List Int

-- ALGORITHMS --

-- algorithm for inserting a new item into an already sorted list
def insertSorted (i : Int) : IntList → IntList
    | [] => [i]
    | y :: ys => if i < y then
        i :: y :: ys
      else
        y :: (insertSorted i ys)

-- algorithm insertion sort
def insertionSort : IntList → IntList
  | [] => []
  | x :: xs => insertSorted x (insertionSort xs)

-- PROPERTY --
-- "List is sorted"
def sortedList : IntList → Prop
  | [] => true
  | _ :: [] => true
  | a :: b :: cs => (a ≤ b) ∧ sortedList (b :: cs)

-- THEOREM THAT LIST IS SORTED --

-- correctness proof for insertSorted
theorem iSRetainsSorting (xs: IntList) (invariant : sortedList xs) :
∀ x : Int, sortedList (insertSorted x xs) := by
  intro x
  induction xs with
  | nil => rfl
  | cons a bs th =>
    by_cases l : x < a
    · grind [sortedList, insertSorted]
    · match bs with
        | [] =>
          grind [sortedList, insertSorted]
        | b :: t =>
          grind [sortedList, insertSorted]

-- inductively use iSRetainsSorting to show correctness of insertionSort itself
theorem iSIsSorted : ∀ xs : IntList, sortedList (insertionSort xs) := by
  intro xs
  induction xs with
  | nil => simp [insertionSort, sortedList]
  | cons h t ih =>
    unfold insertionSort
    let betterFact := iSRetainsSorting (insertionSort t) ih h
    exact betterFact
```

```
-- THEOREM THAT OUTPUT LIST IS PERMUTATION OF INPUT LIST --

-- to access List.Perm as `~`
open List

theorem insertSortedPerm (x : Int) (xs : IntList) : (x :: xs) ~ (insertSorted x xs) := by
  induction xs with
  | nil => simp [insertSorted]
  | cons a bs th =>
    unfold insertSorted
    by_cases l : x < a
    · simp [l]
    · simp [l]
      let fact := List.Perm.swap a x bs
      let elaboratedTh := List.Perm.cons a th
      exact List.Perm.trans fact elaboratedTh

theorem iSPerm : ∀ xs : IntList, xs ~ insertionSort xs := by
  intro ys
  induction ys with
  | nil => unfold insertionSort; simp
  | cons a bs ih =>
    unfold insertionSort
    let fact₁ := List.Perm.cons a ih
    let fact₂ := insertSortedPerm a (insertionSort bs)
    exact List.Perm.trans fact₁ fact₂
```

B. Source Code for Array-based `insertionSort`

For convenience, the source code shown here is additionally available for download [27, SortingArray.lean].

It has been slightly adapted from its original version from the Lean documentation book "Functional Programming in Lean" [12, Section 10.6].

```
def insertSorted [Ord α] (arr : Array α) (i : Fin arr.size) : Array α :=
  match i with
  | ⟨0, _⟩ => arr
  | ⟨i' + 1, _⟩ =>
    match Ord.compare arr[i'] arr[i] with
    | .lt | .eq => arr
    | .gt =>
      insertSorted (arr.swap i' i) ⟨i', by simp; omega⟩

theorem insert_sorted_size_eq [Ord α] (len : Nat) (i : Nat) :
    (arr : Array α) →
    (isLt : i < arr.size) →
    (arr.size = len) →                                  -- look here
    (insertSorted arr ⟨i, isLt⟩).size = len := by       -- look here
  induction i with
  | zero =>
    intro arr isLt hLen
    simp [insertSorted]
    exact hLen
  | succ i' ih =>
    intro arr isLt hLen
    simp [insertSorted]
    split <;> try apply hLen
    simp [*]

def insertionSortLoop [Ord α] (arr : Array α) (i : Nat) : Array α :=
  if h : i < arr.size then
```

```
    have : (insertSorted arr ⟨i, h⟩).size - (i + 1) < arr.size - i := by
      let fact := insert_sorted_size_eq arr.size i arr h rfl
      rw [fact]
      omega
    insertionSortLoop (insertSorted arr ⟨i, h⟩) (i + 1)
  else
    arr
termination_by arr.size - i

def insertionSort [Ord α] (arr : Array α) : Array α :=
   insertionSortLoop arr 0
```