Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Proof Visualization for the Lean 4 theorem prover

Bachelorarbeit von

Niklas Fabian Bülow

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	M. Sc. Sebastian Ullrich
Abgabedatum:	05. April 2022

Abstract

Interaktive Theorembeweiser helfen nicht nur mathematische Beweise zu formalisieren, sondern stellen auch Werkzeuge zur Verfügung, welche Mathematiker:innen in der Konstruktion und Validierung von mathematischen Beweisen unterstützen sollen. Eine der entscheidenden Herausforderungen dieser Herangehensweise der Konstruktion von Beweisen mit Hilfe einer dedizierten Programmiersprache ist es, die Lesbarkeit und den einfachen Austausch mit Anderen, besonders mit Hilfe statischer Medien, beizubehalten. In dieser Arbeit wird die Unterstützung des interaktiven Lean 4 Theorembeweiser in Alectryon implementiert, einem Scripttool ursprünglich entwickelt für den Coq Theorembeweiser, welches versucht genau diese Herausforderungen anzugehen. Zusätzlich wird die Funktionalität von Alectryon erweitert, um noch weitere Metainformationen in dessen Ausgabe darzustellen.

Interactive theorem provers do not only help to formalize mathematical proofs, but they also provide tools to support mathematicians in the construction and validation of mathematical proofs. One of the critical challenges of this approach of constructing proofs using a dedicated programming language is maintaining the readability and shareability of proofs, especially on static media. Therefore this thesis implements support for the Lean 4 interactive theorem prover in Alectryon, a script tool initially developed for the Coq theorem prover that tries to address this challenge. Additionally, Alectryon's functionality is extended to present more meta-information in its output.

Contents

1 Introduction

In this chapter, we take a high-level tour over the key topics we need to understand for this thesis. We begin by looking at the history of mathematical proofs and how interactive theorem provers can be used to formalize proofs. More specifically, we make use of the Lean 4 theorem prover [1] and compare an informal and formal representation of an proof. Meanwhile, we cover multiple aspects on why we aim for this formalization of mathematical proofs. After that, we evaluate the challenges this formalization raises, and in doing so, we motivate the problem we try to solve in this thesis.

1.1 Formalizing mathematical proofs

The roots of mathematics were predominantly chaotic. Initially, there was no common framework for describing math or even describing what math in itself is. As a result, different cultures claimed different approaches to mathematical knowledge. However, Greek philosophers began describing a logic framework in the early years of the antique. One of the first known kinds of mathematical proof was a deductive method, which Euclid later used around 300BC to construct an axiomatic framework for the number system and fundamental geometry [2].

Philosophers and scientist all around the world built upon this fundamental logical structure of argumentation in the centuries after. The formalization of mathematics continued on from that point in history to the formalization we now know and use every day. For example, the first-order logic model developed, which provides us today with a base framework to construct a wide variety of logical proofs.

Especially in the mid 20th century, when the first computers were introduced as basic computing machines, a whole new scientific research field was born that attempted to make use of the computing power of these new machines. As a result, computers are nowadays used specifically for computational heavy applications and automation as they are much more efficient and less error-prone than humans regarding basic mathematical computations. Nowadays, science uses applications like computer algebra systems to solve computationally heavy problems in calculus, algebra, or other fields.

Additionally, computers are based on our fundamental understanding of logic by combining logical gates to produce more intricate functionality. So it was a natural question to ask if we can use computers to solve logical problems such as constructing proofs and verifying them [3]. Out of this idea, a handful of tools originated. Automated theorem provers, also known as ATP, take a mathematical problem as their input and attempt to prove it using built-in heuristics by themselves [4]. Another class of tools are interactive theorem provers, also known as proof assistants [5]. The aspect that characterizes this class of tools is that instead of trying to solve a problem solely on their own, they aid and work together with a human to construct a proof for the problem. We will focus on this class of tools in this thesis, especially on one implementation of this kind, the Lean 4 theorem prover [1].

Formal vs Informal Proof

Understanding the difference between a formal and informal proof is a key aspect for the topic of this thesis. Hence, we will compare both the formal and informal version of a simple proof, on a high-level, and later in Chapter 2 in more detail. The example proof may seem trivial at first, but by removing the burden of understanding the proof itself, we can focus on the representation of the proof instead.

Example 1.1.1 ((Informal) Example Proof)**.** *Let A and B each be a statement or proposition. Then the following statement holds:*

$$A \vee B \iff B \vee A$$

Proof.

Case 1. Let us first show $A \vee B \rightarrow B \vee A$. So we assume that $A \vee B$ is true. Consequently, either A or B is true. So if A is true, then no matter if B is true, $B \vee A$ is fulfilled. We can use the same argument for the case that B is true, in order to fulfill $B \vee A$.

Case 2. For the opposite direction $A \vee B \leftarrow B \vee A$, we can use the same approach as above. We only have to swap A with B.

□

As an individual, who is familiar with the basics of mathematical notation, this kind of proof should be fairly easy to read and follow through. It makes use of our typical skill of reading sentences and logical understanding, as each case describes the steps we need to take to achieve the goal in plain English sentences. However, exactly this can often lead to ambiguity due to the ambiguity of the English language itself.

Therefore, we strive to create either a framework, model, or even a whole language without or with less ambiguity than our typical spoken or written languages. Consequently, modern mathematics includes an intricate notation framework to reduce the ambiguity of mathematical proofs. In computer science, programming languages, specifically type-safe programming languages, are even more designed towards reducing and ideally removing any ambiguity from a written program to make sure the computer behaves exactly as expected.

Interactive theorem provers often define their interface by using a domain specific programming language that reduces ambiguity and improves conciseness of its input.

The Lean 4 theorem prover for example defines the Lean 4 language for constructing and validating proofs. Here is the same proof as above, but formalized using the Lean 4 theorem prover:

```
theorem exampleProof (a b : Prop) : a ∨ b ↔ b ∨ a := by
  apply Iff.intro
  . intro h
    cases h
    . apply Or.inr
      assumption
    . apply Or.inl
      assumption
  . intro h
    cases h
    . apply Or.inr
      assumption
    . apply Or.inl
      assumption
```

Listing 1.1: Example proof formalized in Lean 4

The formalized representation of our example proof is less intuitive to understand. This is mostly because being familiar with the fundamentals of Lean 4, which will be introduced in Chapter 2, is a requirement to comprehend the proof itself.

1.2 Challenges of sharing and reading formal proofs

The formalized version of the proof in Lean 4 has many benefits over the informal representation. First, it is less ambiguous and also more concise, as it follows a standardized set of notation. The Lean 4 theorem prover also ensures that the proof is valid and it provides additional tools to inspect or construct proofs. Finally, implementing a proof in a formalized format in a computer, makes it easier to reuse and to share a proof with others in form of source code or even a whole library or framework. However, there are also some downsides to this representation of a proof.

- First, the readability of the proof is worse compared to the informal representation, especially for someone unfamiliar with the Lean 4 language. In order to understand the proof, we first have to understand the concepts behind the Lean 4 theorem prover, and we also need to be proficient in our knowledge of the provided tactics and theorems of the language itself or of any external dependency.

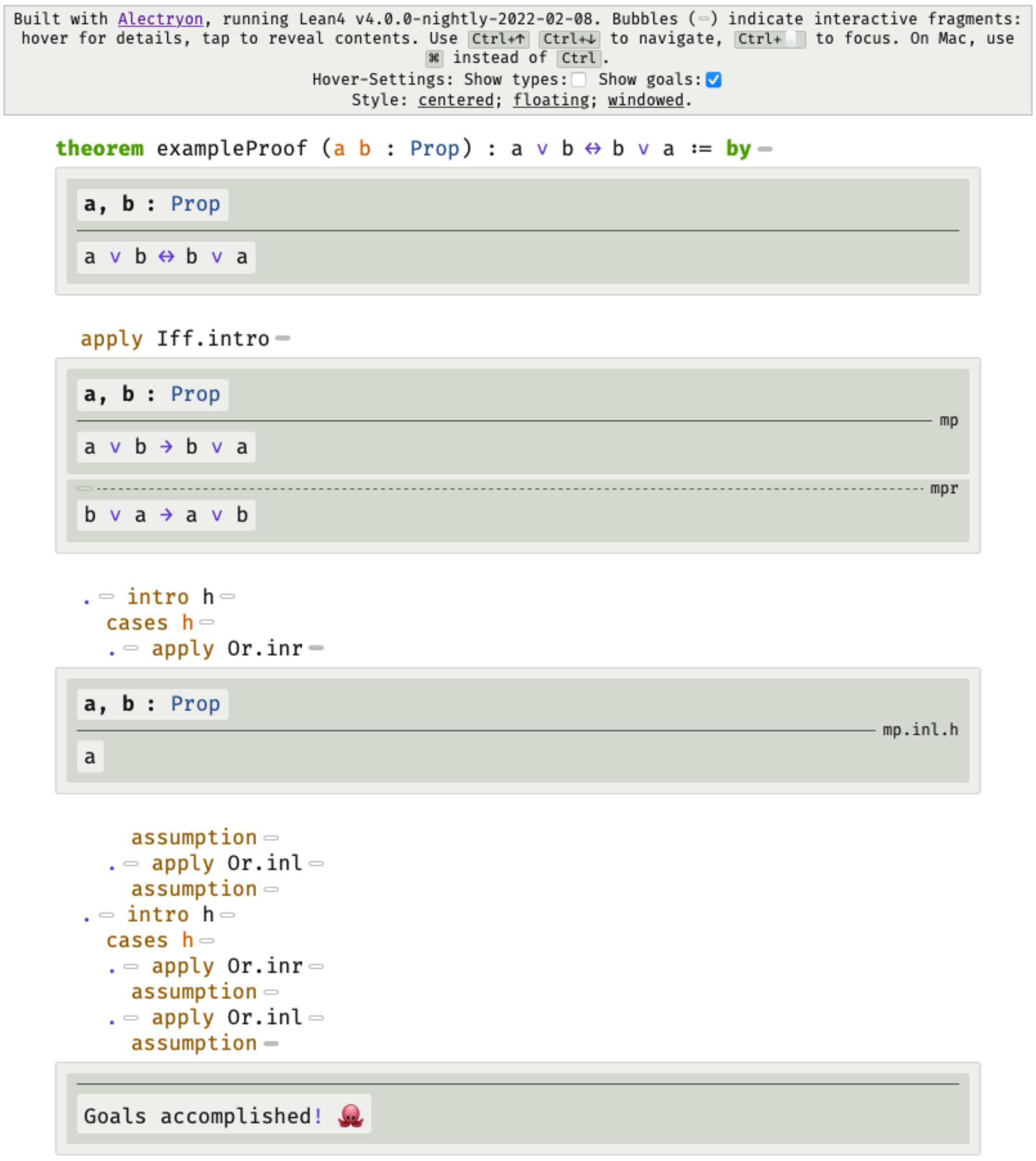
- Sharing the code with others is a benefit of the formalized approach, but it also raises the requirement that the person receiving the source code has to install both the correct Lean version and the same dependencies to make sure the behavior of the proof is the same.

These issues are increasingly present if the source code is shared only as a static resource, for example, in a LaTeX document or a website. In this case, running, inspecting, and understanding the static version of the proof is even more difficult. Previously either additional documentation in the source code was given to make it more understandable, or the reader had to manually copy-and-paste the code to use the interactive development environment of Lean 4, for example by using the Lean 4 Visual Studio Code extension[1].

This is why we implement Alectryon support for Lean 4 in this thesis. Alectryon is a tool, initially developed for the Coq theorem prover, that attempts to solve the issue of sharing static instances of interactive theorem prover source code [6]. It does so by analyzing the source code and embedding additional information into the static representation of the source code. One of the primary use cases of Alectryon is to generate an interactive website that allows others to read and inspect the source code with additional information, without the need of a fully configured development environment on their system. However, it also supports generating LaTeX documents and converting markup documents from and to source code [6], so called literate programming [7].

An example of Alectryon and the Lean 4 support, which is implement in this thesis, can be seen in Figure 1.1 on the next page. In this example, the proof source code of Listing 1.1 was used to generate the interactive website. The website shows intermediate information about the proof and gives hints about the Lean 4 version that has been used to compile the code. The following chapters will go into more detail about the functionality and features of Alectryon's Lean 4 support.

[1]`https://github.com/leanprover/vscode-lean4`



```
Built with Alectryon, running Lean4 v4.0.0-nightly-2022-02-08. Bubbles (⬭) indicate interactive fragments:
hover for details, tap to reveal contents. Use Ctrl+↑ Ctrl+↓ to navigate, Ctrl+🖱 to focus. On Mac, use
⌘ instead of Ctrl.
Hover-Settings: Show types: ☐ Show goals: ☑
Style: centered; floating; windowed.

theorem exampleProof (a b : Prop) : a ∨ b ↔ b ∨ a := by

  a, b : Prop
  ─────────────────────────────
  a ∨ b ↔ b ∨ a

  apply Iff.intro

  a, b : Prop
  ───────────────────────────── mp
  a ∨ b → b ∨ a
  ----------------------------- mpr
  b ∨ a → a ∨ b

  . intro h
    cases h
    . apply Or.inr

  a, b : Prop
  ───────────────────────────── mp.inl.h
  a

      assumption
    . apply Or.inl
      assumption
  . intro h
    cases h
    . apply Or.inr
      assumption
    . apply Or.inl
      assumption

  ─────────────────────────────
  Goals accomplished! 🐙
```

Figure 1.1: Website of our formalized Lean 4 example proof generated using Alectryon and LeanInk.

2 Fundamentals and related work

We now have a high-level understanding of the problem environment we are dealing with in this thesis. However, we first have to build up the theoretical basics and terminology of this environment in order to discuss the actual implementation and our solution to the problem as layed out in Chapter 1.

Therefore, in this chapter, we look at the fundamentals of the Lean 4 theorem prover [1] and its compiler. We introduce some language aspects of Lean 4 and explain how proofs in Lean 4 are constructed and verified. Thereafter, we dive into the structure of the Lean 4 compiler and the ideas it implements.

Finally, we explore the basic structure and functionality of Alectryon. We describe how Alectryon uses custom drivers to retrieve its metadata and finally how it uses this metadata to generate either an interactive or static representation of the input source.

2.1 Interactive theorem provers

Interactive theorem provers (ITP), or proof assistants, are software tools that assist a human specialist in constructing and verifying mathematical proofs [8]. This is different compared to automated theorem provers, that attempt to verify and construct solely on their own. An ITP therefore transforms the traditional way of constructing proofs mainly with paper and pen, into an assisted approach of solving a proof using a formalized format. Therefore they offer mathematicians a way to formalize and prove mathematical theories.

ITPs often implement a small kernel program that verifies a given proof term. These kernels are very small and usually easy to verify manually. Making sure that the proof checker can be checked for correctness themselves, is one key aspect. An ITP is just another program that may contain bugs, so verifying that the kernel is correct is just as important. This separation of an ITP into a small kernel and the rest of the ITPs implementation is known as de Brujin criterion [9]. It ensures that we have to trust only a small part of the implementation to ensure the overall soundness of the proof verification.

2.1.1 Interactive theorem provers and compilers

The formalization of a proof, when using an ITP, often comes in form of a programming language. As a result, ITPs and programming languages are based on a similar set of fundamentals.

One common implementation of programming languages is a compiler. Compilers are a complex systems that transform text into executable binaries that can be understood by computers [10]. The input text comes in form of the source text based on a programming language. Hereby, the compiler defines the language syntactically and semantically using a lexical grammar and an additional set of rules.

A compiler can be partitioned into a sequence of phases from initial input to final output. One common partition of its structure is: lexical analysis, syntactic analysis, semantic analysis, transformation, optimization and code generation.

2.2 Lean 4 theorem prover

The Lean theorem prover project was initially launched in 2013 by Leonardo de Moura at Microsoft Research [11]. Lean 4 is the fourth major version of the Lean theorem prover, and this is the version of Lean we focus on in this thesis. The fourth version of the Lean theorem prover is a reimplementation of the language using Lean itself, and it now can be used both for general-purpose programming and interactive theorem proving [1, 12]. We first take a look at the syntax of the Lean 4 language as a programming language, and thereafter in form of the interactive theorem prover.

2.2.1 Fundamentals

```
namespace Thesis

structure Rectangle where
  width : Nat
  height : Nat

inductive Tree where
  | leaf
  | left (child : Tree)
  | right (child : Tree)

def main (args: List String) : IO UInt32 := do
  let string : String = "Hello World!"
  IO.println string
  return 0

def and : Bool -> Bool -> Bool
  | true, true => true
  | _, _ => false

/- Multiline comment
 #check prints the type of the expression during compilation
 -/
```

```
#check and
-- #eval evaluates the expression during compilation
#eval 1+3
-- #print prints the value as a message during compilation
#print "Hello world"
```

Listing 2.1: Programming language primitives of Lean 4

The Lean 4 programming language is a functional and type-safe programming language, based on dependent type theory [1]. Listing 2.1 presents the basic primitives of Lean 4 as a programming language. Simple data structures in Lean 4 are defined by the `structure` keyword. Furthermore, Lean 4 supports enumerate or inductive types, which are defined using the `inductive` keyword. Inductive types are similar to enums, however in contrast to enums, they also support additional parameters for each of their cases.

The `def` keyword is used to define a function. In the example code we create a `main` function, which takes a single parameter `args` of type `List String`. The output type of the `main` function is `IO UInt32`, whereby `IO` is a monad for basic input output operations and a return code in form of a `UInt32`. Within our main function we define a immutable variable `string` using the `let` keyword.

The second function `and` implements the logical and operation by using pattern matching and Lean 4 support for higher-order functions. In addition to these basic language constructs, Lean 4 also has built-in support for hygienic macros, type classes, meta programming and more.

```
theorem proof (a: Prop) : a = a := by
  rfl

example (a: Prop) : a = a := proof
```

Listing 2.2: Proof assist primitives of Lean 4

Listing 2.2 presents the fundamental entry points of the language for implementing proofs using Lean 4. The keywords `example` and `theorem` both define a proof within Lean 4, whereby the latter defines a named proof instance, which therefore can be used and referenced by other proofs.

We already introduced a more complicated implementation of a proof in Lean for in Listing 1.1. In this example we use Lean's "tactic mode" to construct our proof [13]. The "tactic mode" can be accessed by using the `by` keyword at the start of our proof implementation. Within tactic mode we can use tactics to construct our proofs step-by-step. Tactic mode is a domain specific language (DSL) within Lean that improves the readability and construction of proofs. We will now go into more detail on how a proof in Lean 4 can be understood when using tactic mode.

2.2.2 Proofs in Lean 4

At first we come back to typical informal proof that we introduced in Chapter 1. We then transform this informal proof into a formal proof using the Lean 4 theorem prover in order to understand the structure and semantics of the formalized representation.

Example 2.2.1 (Example Proof)**.** *Let A and B each be a statement or proposition. Then the following statement holds:*

$$A \lor B \iff B \lor A$$

Proof.

Case 1. Let us first show $A \lor B \to B \lor A$. So we assume that $A \lor B$ is true. Consequently, either A or B is true. So if A is true, then no matter if B is true, $B \lor A$ is fulfilled. We can use the same argument for the case that B is true, in order to fulfill $B \lor A$.

Case 2. For the opposite direction $A \lor B \leftarrow B \lor A$, we can use the same approach as above. We only have to swap A with B.

□

Formal proof (Proof Tree and Natural Deduction)

Before we can transform the informal version of our proof into the formalized version using the Lean 4 theorem prover, we first take a look at a familiar concept: proof trees and the framework of natural deduction [5]. We can use a proof tree and natural deduction to represent the informal proof more formally:

$$\dfrac{\dfrac{\dfrac{}{A \lor B}\,{\scriptstyle (1)} \quad \dfrac{\dfrac{}{A}\,{\scriptstyle (3)}}{B \lor A}\,{\scriptstyle \lor I_r} \quad \dfrac{\dfrac{}{B}\,{\scriptstyle (3)}}{B \lor A}\,{\scriptstyle \lor I_l}}{B \lor A}\,{\scriptstyle (3)\lor E} \qquad \dfrac{\dfrac{}{B \lor A}\,{\scriptstyle (1)} \quad \dfrac{\dfrac{}{B}\,{\scriptstyle (2)}}{A \lor B}\,{\scriptstyle \lor I_r} \quad \dfrac{\dfrac{}{A}\,{\scriptstyle (2)}}{A \lor B}\,{\scriptstyle \lor I_l}}{A \lor B}\,{\scriptstyle (2)\lor E}}{A \lor B \leftrightarrow B \lor A}\,{\scriptstyle (1)\leftrightarrow I}$$

This may seem overwhelming at first, but it will ease the initial learning curve of understanding the formal proof in Lean 4.

Natural Deduction Rules

To make sense of the structure of this proof tree, we will introduce some high-level concepts of natural deduction proof trees. Prominently, the root of our proof tree is equivalent to the goal of our informal proof $A \lor B \leftrightarrow B \lor A$. Above the line of our root node are two hypotheses that we have to prove in order to prove our root node. In this case, the hypotheses are defined by the rule $\leftrightarrow I$. As we can see every line in our proof tree is annotated with either a rule name, index or both. The index describes either the origin of a new assumption in our proof tree or the application of an assumption we previously made in one of the steps in the proof tree. The rule names specify the rule taken at the specific position in the tree. Natural deduction is based on a set of rules of inference and elimination. However, we will only introduce the rules we used in our example proof tree. The general structure of a rule is:

$$\frac{H_1, H_2, H_3, \ldots}{G}\ {\scriptstyle (Index),Name}$$

Whereby H_n are hypotheses, or smaller proofs, that have to be fulfilled for our goal G to be proven. The $Index$ and $Name$ are optional annotations that help us understand the proof. For example, we may have to use the $Index$ to fulfill an assumption we made during the resolution of a rule.

$$\texttt{Iff (Bi-Implication):}\quad \frac{\begin{matrix}\overline{A}^{\,(1)} & \overline{B}^{\,(1)}\\ \vdots & \vdots \\ B & A\end{matrix}}{A \leftrightarrow B}\ {\scriptstyle (1)\leftrightarrow I}$$

The $\leftrightarrow I$ rule describes the proof of a bi-implication. It does so by splitting both sides of the bi-implication into two new hypotheses that must be proven by using the other side as an assumption. So in order to solve B we can use the assumption that A is true and vice versa.

Other rules we are using in our proof are the following:

$$\texttt{Disjunction:}\quad \frac{\begin{matrix} & \overline{D}^{\,(1)} & \overline{E}^{\,(1)}\\ & \vdots & \vdots \\ D \lor E & C & C\end{matrix}}{C}\ {\scriptstyle (1)\lor E} \qquad \frac{B}{A \lor B}\ {\scriptstyle \lor I_r} \qquad \frac{A}{A \lor B}\ {\scriptstyle \lor I_l}$$

These describe the rules for the logical disjunction operator $\lor$. Both $\lor I_r$ and $\lor I_l$ are fairly straightforward to understand, because the statement $A \lor B$ is true exactly if either or both A or B is true. So we can prove the statement by using both $\lor I_r$ and $\lor I_l$.

The $\lor E$ is the so-called disjunction elimination rule. It allows us to prove our goal C by using two assumptions that imply the following constraints $D \to C$ and $E \to C$. So in order to solve our proof, we have to make sure that our goal is true if D or E is true. In the example proof tree above, we use this property to apply the assumption we made during the $\leftrightarrow I$ rule at the tree's root.

Now that we understand the rules of the proof tree, we can start making sense of its structure. We start at the root of our tree, the goal of our informal proof. We use the $\leftrightarrow I$ rule to split our goal into two subgoals $A \lor B$ and $B \lor A$ and their respective assumptions. Furthermore, we then use the disjunction elimination to apply these assumptions on the subgoals. We then use the assumption to prove both sides of each subgoal using the $\lor I_r$ and $\lor I_l$ rules and our assumption of the disjunction elimination rule. As a result, we have proven all the hypotheses in the proof tree, our initial goal.

Comparing our proof tree to the informal proof earlier, we see some similarities. The rules we apply in our proof tree are similar to the steps we take in the informal proof. For example, in our informal proof, we first split our bi-implication problem into two smaller implication problems and solved each of the two cases independently. This is similar to the $\leftrightarrow I$ rule we apply at the root node of the proof tree.

Formal Proof (Lean 4)

Lean 4 follows the same idea as our proof tree. However, instead of drawing a proof tree as we did above, Lean 4 is a programming language that lets us apply the natural deduction using tactics [1].

Here is the same proof as above, but implemented using the Lean 4 theorem prover:

```
theorem exampleProof (a b : Prop) : a ∨ b ↔ b ∨ a := by
    apply Iff.intro
    . intro h
      cases h
      . apply Or.inr
        assumption
      . apply Or.inl
        assumption
    . intro h
      cases h
      . apply Or.inr
        assumption
      . apply Or.inl
        assumption
```

Listing 2.3: Example proof formalized in Lean 4

The basic structure is similar to our proof tree. The first line defines the name of our proof `exampleProof` our input parameters `a, b` and their type `Prop` ($\approx$ proposition), as well as the goal we want to achieve `a` $\lor$ `b` $\leftrightarrow$ `b` $\lor$ `a`. The following lines are the body of our theorem where we succinctly apply rules to achieve our goal. We can see rules similar to the ones we used for our proof tree. For example, `Iff.intro` is the $\leftrightarrow I$ rule that splits our goal into two cases. The `intro h` introduces a new assumption with the name `h`. By doing so, we implicitly apply the disjunction elimination rule. Finally `Or.inr` and `Or.inl` are equivalent to the $\lor I_r$ and $\lor I_l$ of our proof tree, so naturally we also use our assumption to proof each of the cases.

2.2.3 Tools for Lean 4

In addition to the Lean 4 programming language, there are tools that ease the development experience within the Lean environment. First, the Lean 4 ecosystem provides a package and dependency manager Lake that supports sharing and using external dependencies in ones own Lean 4 code. One example for this is the mathlib4[1] library that implements tactics and additional implementations for mathematical constructs and theorems, that are useful when constructing ones own proof.

Another helpful tool offered by the Lean 4 ecosystem is the Lean 4 language server. The Lean 4 language server can be accessed using `emacs` or the Visual Studio Code extension for Lean 4. It offers similar support to what we are going to implement in this thesis as it allows to explore and discover the Lean 4 source code interactively.

[1]`https://github.com/leanprover-community/mathlib4`

However contrary to our problem domain, the Lean 4 server depends on a working Lean 4 toolchain installation in order to resolve requests. It also resolves these request, for example a hover event, just-in-time and only at the specific location in the source. Our solution to the problem of this thesis on the other relies on a ahead-of-time analysis of the full Lean 4 code, that makes it possible to share a static analysis result with other tools.

2.2.4 Lean 4 Compiler

The Lean 4 compiler, itself implemented mostly using Lean 4, implements the Lean 4 programming language and transforms it into valid C code.

The first phase of the Lean 4 compiler combines both the lexical and syntactic analysis. A recursive-descent parser checks the input source code for syntactic correctness based on the Lean 4 grammar and generates an concrete syntax tree. Thereafter, a macro expander implements the extensive macro system within Lean 4.

The third phase of the Lean 4 compiler is the elaboration phase. The elaboration phase will be the main interaction point for the implementation of this thesis. An elaborator enhances and extends the expanded syntax tree that it receives from the macro expander, in order to make it easier for the Lean kernel to verify a proof [14]. The elaborator creates an `Infotree` with the additional metadata it gathered, which can be seen as a attributed abstract syntax tree. During the elaboration phase, the Lean 4 compiler also checks the well-formedness by doing a semantic analysis on the input. This includes type checking to ensure type correctness of the Lean 4 program.

Finally, the elaborator generates terms that the small Lean 4 kernel subsequently verifies.

Elaborator

The implementation of our solution will mainly interact with the elaborator and its `InfoTree`. Hence a thorough introduction of the basic structure of the `InfoTree` and the elaborator interface needs to be provided beforehand.

To access the elaboration phase from an external program we can use the frontend interface of the elaborator. The interface offers a method `Elab.runFrontend` that takes a Lean 4 input source and additional options as its parameters and then runs the elaborator based on the settings. In order to access the `InfoTree` after elaboration Lean offers the option `trace.Elab.info` to enable the output of the `InfoTrees`. We will later inline this method to access the `InfoTrees` and messages generated by the elaborator.

InfoTree

The `InfoTree` is a tree structure implemented using an inductive type in Lean. So each node within the `InfoTree` has zero or more children nodes.

Access to the `InfoTree` is granted by access to the root node.

```
inductive Info where
  | ofTacticInfo (i : TacticInfo)
  | ofTermInfo (i : TermInfo)
  | ofCommandInfo (i : CommandInfo)
  | ofMacroExpansionInfo (i : MacroExpansionInfo)
  | ofFieldInfo (i : FieldInfo)
  | ofCompletionInfo (i : CompletionInfo)
  deriving Inhabited

inductive InfoTree where
  -- The context object is created by ‘liftTermElabM‘ at ‘Command.lean‘
  | context (i : ContextInfo) (t : InfoTree)
  -- The children contains information for nested term elaboration and
    tactic evaluation
  | node (i : Info) (children : PersistentArray InfoTree)
  -- For user data
  | ofJson (j : Json)
  -- The elaborator creates holes (aka metavariables) for tactics and
    postponed terms
  | hole (mvarId : MVarId)
  deriving Inhabited
```

Listing 2.4: `InfoTree` implementation [15] in Lean 4

Listing 2.5 presents the implementation basis of the `InfoTree` in Lean 4. There are four node types `context`, `node`, `ofJson` and `hole`. The two types of interest for this thesis are `node` and `context` node. The `context` node is of interest because it contains a child node with potential additional information and because it defines a context that we have to use in order to resolve types in the `InfoTree`. The general `node` type is however the most important of all, as it contains specialized info about a specific part in the source. All of the info structures either extend the `ElabInfo` structure or have their own property `stx` that allows to refer to the specific syntax range of the input source.

```
structure TacticInfo extends ElabInfo where
  mctxBefore  : MetavarContext
  goalsBefore : List MVarId
  mctxAfter   : MetavarContext
  goalsAfter  : List MVarId
  deriving Inhabited
```

Listing 2.5: `TacticInfo` implementation [15]

Most notably the `TacticInfo` contains information about a specific tactic in a proof. It specifies both a context and the goals before and after the application of the tactic. Tactics can also appear nested in the `InfoTree`. One example for this is the `And.intro` tactic that introduces two new subgoals. In this case, we can use the `.-combinator` to create a new node that will appear as a nested tactic in one of the child-trees that can be accessed via `children` property.

The `TermInfo` describes properties about a term within the source text. Terms are expressions that are literals or references to variables or functions and any combination of them with additional binding operators. Similar to tactics, terms can be nested. An example of this case is a simple arithmetic term expression `1 + 2`, whereby the expression is a term, as well as both literals `1` and `2`. `TermInfo` contains a `expr` property that can be used to analyze the elaborated term to retrieve for example the type of the expression.

2.3 Alectryon

Alectryon is the script tool for which we are implementing Lean 4 support in this thesis. Alectryon is a tool written in Python and developed by Clément Pit-Claudel, that initially only supported the Coq theorem prover language [6, 9]. It is a library and command-line tool that takes code snippets of proofs written using an interactive theorem prover, analyses them and finally creates a representation of them in a document like LaTeX or HTML that can be statically presented without the requirement of a working development stack on the same machine. In addition to the source text, the generated output files also include helpful annotations of the source text with information about goal states and messages. Finally it offers a set of tools for literate programming with interactive theorem provers.

2.3.1 Literate Programming

Literate programming allows a user to simultaneously construct a proof while documenting it in the same file using a another language, for example markup languages. Both the programming language source code as well as the markup code are interspersed within the same file. It was first introduced in 1984 by Donald Knuth within the WEB programming system [7]. WEB supports embedding Pascal code into a TeX formatted file. Alectryon implements literate programming by transforming between both the theorem prover format and a markup language format. It provides extensive support for the reStructuredText markup language, as well as basic support for Markdown.

```
/-|
Literate programming
====================

The source code within these comment tags is content formatted
based on a markup language. In this case reStructuredText.
-/

theorem test (p q : Prop) (hp : p) (hq : q): p ∧ q ↔ q ∧ p := by
  -- proof tactics
```

Listing 2.6: Literate programming: Lean 4 format

```
Literate programming
==================

The source code within these comment tags is content formatted
based on a markup language. In this case reStructuredText.

.. lean4::

   theorem test (p q : Prop) (hp : p) (hq : q): p ∧ q ↔ q ∧ p := by
     -- proof tactics
```

Listing 2.7: Literate programming: reStructuredText format

An example of the literate programming support in Alectryon and with our implementation is displayed in Listing 2.6 and Listing 2.7. Listing 2.6 shows the file contents in the case of programming within a Lean 4 source file, while the latter shows the contents in the format of reStructuredText. In the first case, the user can then compile Listing 2.6 using the Lean 4 compiler.

Both formats have the code of the other format embedded within the same source file. The Lean 4 source code embeds the markup language contents in special multiline comments: `/-| -/`. reStructuredText on the other hand allows embedding of source code of other language by using directives. Directives are a standardized language feature in the reStructuredText markup specification [16]. The directive used in Listing 2.6 is `.. lean4::`, which is a custom specified directive with keyword `lean4` which Alectryon will use to identify the content language.

2.3.2 Processing Structure

In this chapter, we will outline the high-level structure of Alectryon's implementation. We then take a more detailed look at the two parts involved in this structure that are relevant for the implementation of this thesis.

Figure 2.1 depicts the high level structure of Alectryon in form of a state diagram. It describes each step that is involved from initial input of the source file to the output file. However, this diagram only describes one usage example of Alectryon's processing structure, that is most common for our use case.

Command-line Interface

First, the command-line interface of Alectryon is called with an input source file, either source code file or a literate document file, and additional arguments for example the output type. Alectryon offers a wide range of options to configure the processing of the input file, from configuring caching behavior to options for each backend and frontend available in Alectryon. The command-line interface infers the type of the input file based on its file extension and then defines the processing pipeline based on the file type and the command-line options.

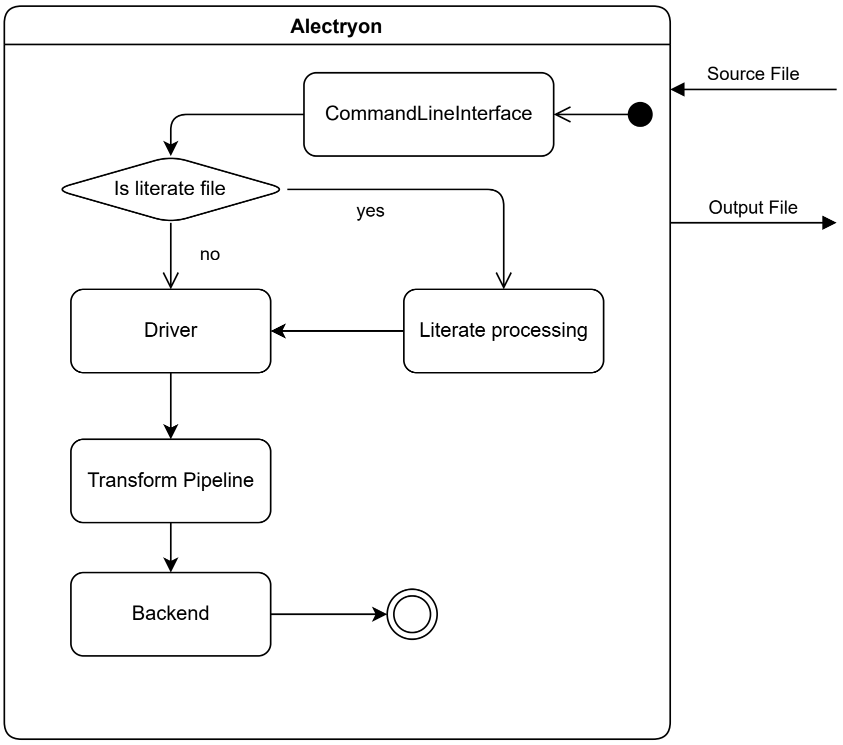


Figure 2.1: Overview of Alectryon processing structure

Driver

In case of a literate file, for example a reStructuredText file, Alectryon first extracts all code snippets from the source file before continuing with their specific drivers. A driver within Alectryon is a class specialized for a certain language for example Coq, that provides the necessary metadata for a chunk of code that it retrieves. There are two kind of drivers in Alectryon at the moment. The first kind uses a custom implementation and analyzes the a code-chunk on their own. The second kind propagates the code-chunks it retrieves to an external compiler or command-line program for a more complex analysis.

Transform Pipeline

After the driver completes its analysis of the source code, it returns the annotated code chunk in the transform-pipeline data model described in Figure 2.2. The return type is the source code partitioned into a list of `Fragment` where each `Fragment` is either a `Sentence` or `Text` fragment. The latter hereby is the most simplest type of `Fragment` as it describes a range of source text without any additional metadata. The `Sentence` fragment on the other hand includes information about any `Messages` or `Goal` information at the specific source text location. It is important to notice that the actual raw string contents of the code-chunk are embedded within the list of `Fragment`.

The list of `Fragment` is now propagated to Alectryon's transform pipeline. The

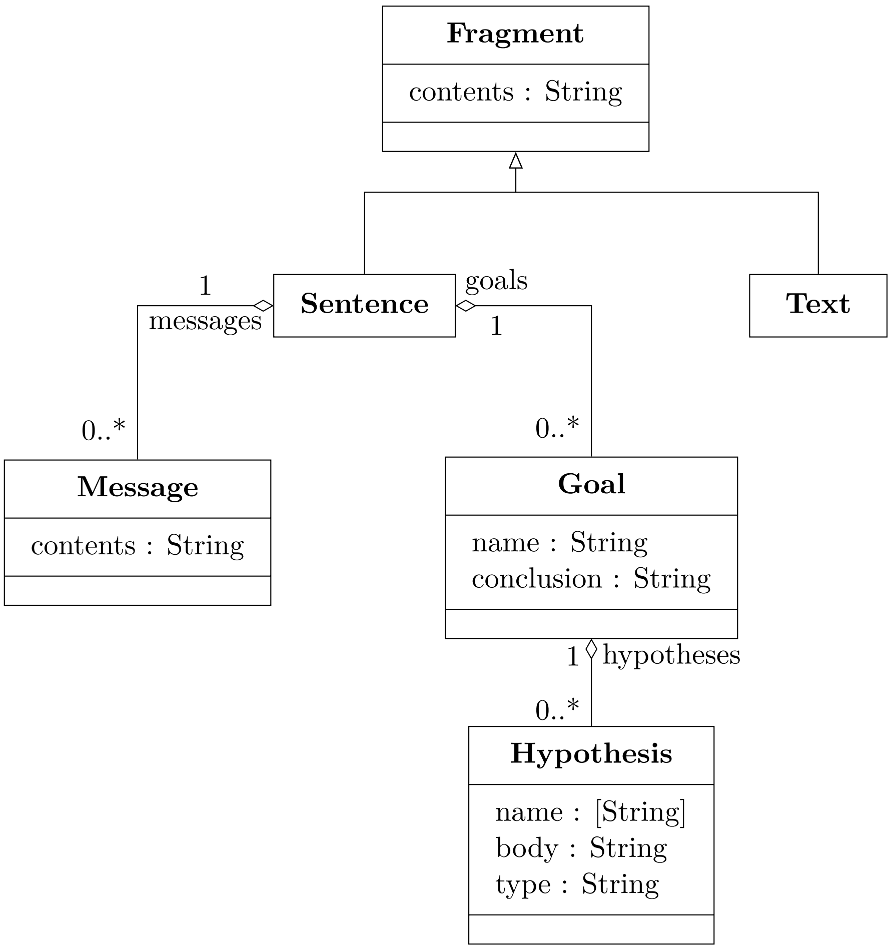


Figure 2.2: Overview of Alectryon transform pipeline data model

transform pipeline applies a sequence, of so-called transform functions, to the list. These transforms implement tasks for example partitioning or merging adjacent fragments or processing Alectryon IO annotations that are embedded within the source code. The transform pipeline also enriches each occurrence of a `Sentence` fragment with additional information it gathers during any transformations into a `RichSentence` model.

Backend

Finally after the transform pipeline is complete, Alectryon will propagate its result to the user-defined backend. A backend within Alectryon is a class that generates the output file based on the list of transform `Fragment` it receives. Alectryon currently offers a backend for HTML, LaTeX , reStructuredText, and JSON for caching purposes.

3 Design and implementation

In this chapter, we finally take a look at the implementation of LeanInk[1], the tool we developed, to add support for Lean 4 in Alectryon.

We first describe the minimum implementation to get Lean 4 support on par with the Coq support in Alectryon. By doing so, we take a look at how the source text is transferred over to LeanInk, which then extracts the metadata from the Lean 4 compiler's elaboration tree. After that, we discuss how we support source text that depends on any third-party framework or source files. Finally, we inspect how LeanInk annotates the source text with the extracted metadata to generate an output format that can be used by Alectryon or any other tool.

The second part of this chapter, shows how we extended Alectryon's data model and transform pipeline to extract and visualize even more information from the Lean 4 source file. This information enables Alectryon to display type information, documentation, and adds support of semantic syntax highlighting.

3.1 Lean 4 support in Alectryon

Before we can process any Lean 4 source text in LeanInk, we first have to transfer the source file from Alectryon to LeanInk. As described in Chapter 2, Alectryon defines an abstract `Driver` class. A `Driver` defines a specific frontend implementation within Alectryon. Its crucial aspects are that it provides information about the language version used for analysis and implements a function that annotates a given collection of input source chunks. These input source chunks depend on the initial file format used as the input for Alectryon. A simple `.lean` source file will result in a single chunk of code. A literate file, on the other, hand will provide a chunk for each code block within the file. An example for the latter case is a reStructuredText file with multiple `..lean4` code directives.

Alectryon also provides a more specific subclass of the `Driver` class, named `CLIDriver`, specifically developed for a frontend that interacts with an external command-line-based tool. Therefore, this class is natural for our Lean 4 driver, as LeanInk is an external binary with a simple command-line interface.

To route the source input file to our Lean4 driver in Alectryon, we register support for `.lean` source files and our Lean4 driver in Alectryon's command-line interface implementation. This ensures that Alectryon automatically infers `.lean` files with our new Lean 4 driver. Additionally, we configure support for Alectryon's literate

[1]`https://github.com/insightmind/LeanInk`

programming support. We accomplish this by specifying the `lean4` identifier for any Lean 4 code directives in reStructuredText or Markdown, thus Alectryon recognizes code blocks with this identifier in a literate file as a Lean 4 source code block and therefore propagates the contents of the block as a chunk to our Lean 4 driver. Further, we reuse the parser for Lean 3 comments already present in Alectryon as the multiline-command syntax is unchanged between Lean 3 and Lean 4.

3.1.1 Lean 4 driver

Now that we have configured all steps within Alectryon that can propagate the source input chunks to our Lean 4 driver, we can finally inspect the features of the Lean 4 driver. The Lean 4 driver is small and straightforward because LeanInk implements most of the features.

For annotation, the driver first combines all chunks into a single `EncodedDocument`, another helper class within Alectryon to combine and later recover the chunks from their annotated versions. It then writes this document into a `.lean` file within a temporary directory and calls LeanInk with the temporary file as its input. LeanInk will output a `.lean.leanInk` file after successful analysis. The driver then reads this file and parses the contents to Alectryon's data model. The LeanInk output data model uses the same data model as Alectryon proposes for its transform pipeline. Thus the driver only needs to recover the chunks from the annotated response and return them to Alectryon to continue with the next step.

The driver uses the `leanVersion` command in LeanInk's command-line interface to receive the Lean 4 version used for analysis.

3.1.2 LeanInk

Figure 3.1 on the next page depicts an overview of this structure within Alectryon. Additionally, it provides some insight into the internals of LeanInk, that we are covering in the following chapters. LeanInk uses a similar pipeline-based architecture typical for script tools. The command-line interface decides the parameters and structure of the pipeline based on the input it receives. Hence we first start describing LeanInk's command-line interface and then present each phases of the pipeline step-by-step.

Command-line interface

LeanInk offers a total of four different commands, each of them with shorter aliases:

- `help, -h <COMMAND>`
  The help command generates a useful help message for each command available.

- `version, -v`
  The version command prints the semantic version of the LeanInk instance.

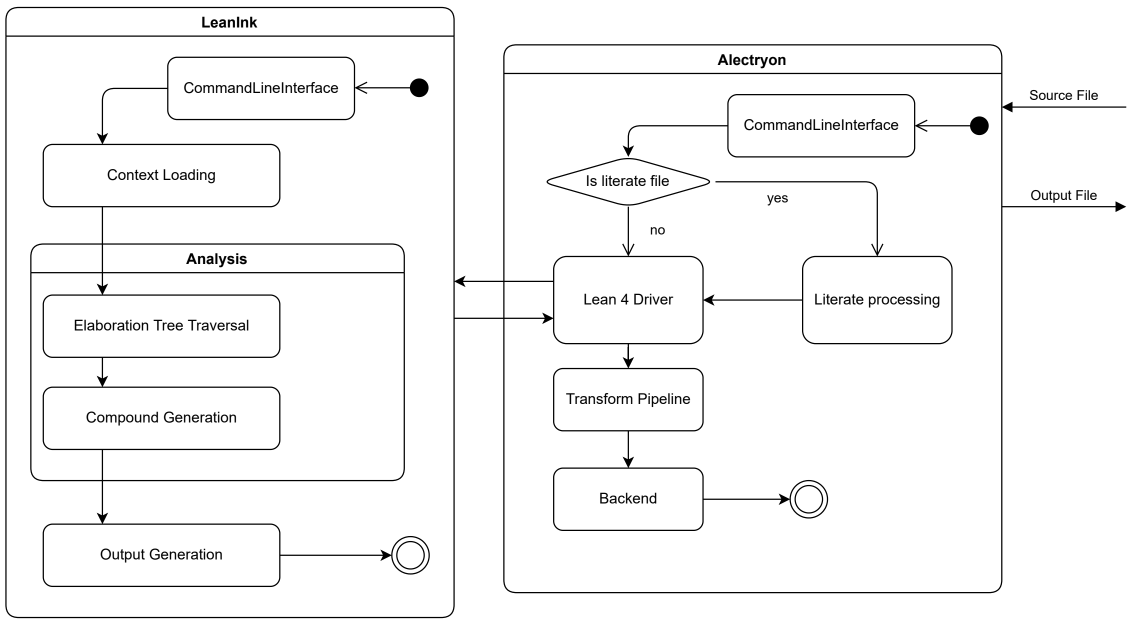


Figure 3.1: Overview of LeanInk integration for Alectryon

- `leanVersion, lv`
  The leanVersion command prints the Lean 4 version of this instance of LeanInk. This version of Lean 4 is the version LeanInk was compiled with and the supported Lean 4 version for any input.

- `analyze, a <INPUT_FILES>`
  The analyze command is the most important command of LeanInk's command line interface (CLI). It takes one or more `.lean` files as its input. In addition to that, it also defines a flag `--verbose` for verbose output of the analysis and annotation pipeline and an optional environment field `--lake`, which allows the caller to set a path to the `lakefile.lean` to discover and resolve any external dependencies of the input files. We also extend the functionality of this command in the second part of our implementation by adding three new flags `--x-enable-type-info`, to enable extraction of type info, `--x-enable-docStrings`, to enable extraction of a `docStrings` for methods and types and `--x-enable-semantic-token`, for semantic syntax highlighting.

A typical example for the default interaction with the LeanInk CLI is the following command:

```
#!/bin/bash
leanInk analyze ExampleProof.lean --lake ../lakefile.lean
```

In this example, the user calls LeanInk with the `ExampleProof.lean` input file. They specify the location of the lakefile as the input file has external dependencies. We will go into more detail about the `--lake` environment in the next chapter.

This call will result in a file called `ExampleProof.lean.leanInk` and its contents represent the annotated input file enriched with additional metadata and formatted using Alectryon's data model.

Context Loading

Before LeanInk can analyze the input source, it first needs to set up the compile environment for the input. The goal of the next step is to load all necessary search paths based on the imports of the source file.

In the first and simplest case, the source file only depends on the standard library of Lean 4. Therefore LeanInk initializes the search paths for the compilation step using Lean's built-in `Lean.initSearchPath` helper method. This method ensures that all required search paths to the Lean installation and the standard library (internally called `Init`) are loaded.

The second case of source file imports supported by Lean 4 are local project dependencies and third-party dependencies of external frameworks and libraries. LeanInk requires a reference to the projects `lakefile.lean` to resolve the search paths for both types of dependencies.

A `lakefile.lean` is a configuration file for the Lake[2] (Lean Make) build system and package manager. The configuration file specifies the project type, compilation instructions, and any external dependencies used by the project. A user can provide this `lakefile.lean` to the `analyze` command using the environment value `--lake`. LeanInk uses this path to spawn two processes of Lake.

The first process makes sure that all external dependencies are loaded and configured. LeanInk uses the `configure` command for Lake to download and build all external dependencies defined in the lakefile. The results of this command are usually cached and only required for a first configuration of the project. Therefore, Lake automatically skips this step if the external dependencies and their respective compiled `.olean` files are already available.

The second process that LeanInk spawns, uses Lake's `print-paths` command. This command takes the `lakefile.lean` path and all imported headers of our source file as its arguments. This process results in a list of all search paths necessary to build the source file. LeanInk uses the result to load the search paths for the compile environment within LeanInk to ensure the analysis phase can succeed.

Now that LeanInk has configured all search paths necessary to compile our source file, it moves to the actual analysis phase.

Analysis

The analysis phase consists of two steps. The first step traverses the Lean 4 `InfoTree` and extracts and formats all metadata required for the output. The second step uses the extracted metadata to generate a linear stream of annotated source text ranges to simplify the implementation of the output generation.

[2] `https://github.com/leanprover/lake`

`InfoTree` Traversal

The first step of the analysis is the elaboration tree traversal. It extracts all the necessary metadata from the `InfoTree` and formats them using appropriate data models. We already discussed the structure of the `InfoTree` in Chapter 2. Therefore, we only describe LeanInk's interaction with the tree in this chapter and then detail the tree traversal algorithm we use to extract the metadata.

To receive the `InfoTrees`, LeanInk inlines the `Elab.runFrontend` method and enables the `trace.Elab.info` option. Within the inlined method, the `IO.processCommands` is called, which compiles the source code and returns a result of type `Frontend.State`. LeanInk then accesses the generated list of `InfoTrees` for each `Command` in the source input using the `Frontend.State.commandState.infoState.trees` property, which it then analyzes sequentially.

Identifying the metadata

To identify the location of our metadata in the elaboration tree, we first have to look back at our example in Chapter 1 and the data structure of the elaboration tree that we described in Chapter 2.

Formal proofs in Lean 4 are constructed using tactics. Tactics are command-like building blocks that transform the initial parameters of the proof to the anticipated goal type. Our goal is to present intermediate information for each of these tactics in our proof. So in order to generate the `Sentence` data types that Alectryon expects, we have to extract all goals and their hypotheses for each tactic step in our proof. We can receive this information from the `TacticInfo` nodes in the elaboration tree. Each `TacticInfo` node contains information about the goals before (`TacticInfo.goalsBefore`) and after (`TacticInfo.goalsAfter`). For the Alectryon integration, we specifically want to present all goals that still have to be solved after the application of the tactic. Hence, we will use the information provided by the `TacticInfo.goalsAfter`.

Looking back at the example in Chapter 1 demonstrates the information we want to present for each tactic.

```
theorem exampleProof (a b : Prop) : a ∨ b ↔ b ∨ a := by
    apply Iff.intro
    /-
    Hypotheses: a b : Prop
    Goals:
        - left-case:    a ∨ b → b ∨ a
        - right-case:   b ∨ a → a ∨ b
    -/
```

In this case, the built-in `Iff.intro` tactic is applied to the initial goal, resulting in two new goals. Both goals share the same hypotheses. Another special case LeanInk has to handle are tactics that fulfill a goal. Here the `TacticInfo.goalsAfter` property is empty and LeanInk returns a goal that contains the text `Goals accomplished!`.

In addition to the intermediate steps, Alectryon also supports visualizing any messages that may occur during compilation:

```
#check Bool
-- Message: "Bool : Type"
-- or
#print "Hello World!"
-- Message: "Hello World!"
```

Other messages that may appear are error messages or warnings during compilation. The `commandState` of Lean's `Frontend.State` automatically returns all of these messages in the `Frontend.State.commandState.messages` property.

Tree-traversal algorithm

The result of our metadata extraction algorithm is a list of `Fragment` structures (Figure 3.2). A `Fragment` is an inductive type with either information about a tactic or message. The list is sorted based on the head position of its syntax range, meaning the list is sorted in order of the occurrence of the fragments in the source text. The `Fragments` that contain information about a tactic are extracted during the tree traversal. Ideally, the algorithm only extracts the most relevant information from the `InfoTree`, the smallest possible `TacticInfo` in the tree. The smallest `TacticInfo` at a specific position in the source text is the `TacticInfo` with the shortest syntax range that still contains the source text position. However, due to the possibility of nested tactics, we still have to extract every node with `TacticInfo`. LeanInk reduces the information to the smallest possible information for each source text location in the compound generation algorithm after the tree traversal.

LeanInk uses a depth-first-search approach to extract all `Fragments` with `Tactic` information. It utilizes a property of the `InfoTree` to reduce the cost of having to sort the list of `Fragments`. The property defines that the syntax-range of each parent node in the tree contains all syntax ranges of its child nodes. Therefore, LeanInk only has to merge the results of its recursive calls, which is possible in linear time.

Another case that needs to be handled by the tree-traversal algorithm is a check to only extract tactic information of tactics that are not part of a macro. LeanInk does not collect the information about macro tactics because it cannot match them to the source text later. After all, the syntax ranges are expanded and non-existent in the source text. Therefore the algorithm omits any `TacticInfo`, that is part of a macro.

- (1) We initialize the algorithm with the root node and the context of the root node. We receive the root node for each command in our Lean 4 source file by first calling the elaboration frontend of the Lean 4 compiler.

- (2) Whenever the current node is a context node LeanInk switches to the new context and uses the only child of the context node to continue the recursion.

- (3) Whenever the current node is a node that is based on an `info` property LeanInk can extract some metadata.

Algorithm 1 Tree-traversal

```
1: function EXTRACT($treeNode, context$)                                        ▷ (1)
2:     if $treeNode.type == context$ then                                       ▷ (2)
3:         return EXTRACT($treeNode.child, treeNode.context$)
4:     else if $treeNode.type == node$ then                                     ▷ (3)
5:         $children := treeNode.children$
6:         $context :=$ UPDATECONTEXT($treeNode$)
7:         $fragments :=$ GENFRAGMENT($treeNode$, $context$)                    ▷ (4)
8:         while $\neg children.isEmpty$ do
9:             $child := children.dropFirst()$
10:            $childFragments :=$ EXTRACT($child, context$)
11:            $fragments :=$ MERGELISTS($fragments$, $childFragments$)         ▷ (5)
12:        return $fragments$
13:    else
14:        return [ ]
```

- (4) For the initial implementation of the tree-traversal algorithm, LeanInk only extracts `TacticInfo` nodes. Because `TacticInfo` nodes can occur nested, it first generates a fragment for the current node and then recursively for every child node. It is important to mention that the `genFragment` method returns a sorted list of fragments based on the information of the current node. For the current implementation, this list only contains zero or one fragment. An empty list is returned only if the type of the `info` property does not match `TacticInfo` or the node is part of a macro expanded tactic.

- (5) LeanInk continues the traversal recursively for each child node of the current node. It then merges the result of the recursive call with the current state of the `fragments` list. The time complexity of the merge is linear because both the result and the current `fragments` list are sorted. The recursion terminates if the current node is a leaf node and therefore has no child nodes.

Finally, we merge the fragments from this algorithm with the messages that appeared during compilation using the elaboration frontend. In order to do so, we first transform each message we receive into an appropriate `MessageFragment` model. We then sort the messages and merge them with the result of the algorithm. Figure 3.2 provides a rough overview of the data model for both `TacticFragment` and `MessageFragment`. Although the initial Alectryon support only requires the information of goals after the application of the tactic, LeanInk still extracts the goals before allowing the output generator to decide which metadata is needed.

Compound Generation

The second part of the analysis receives the sorted list of `fragments` from the tree-traversal algorithm and transforms it into a linear stream of compounds of

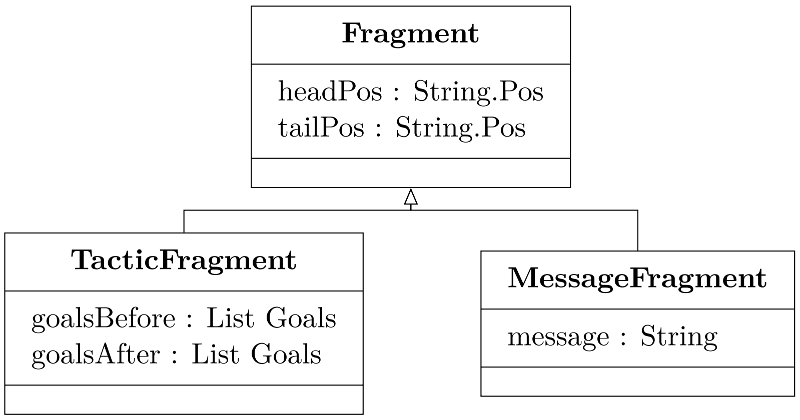


Figure 3.2: Fragment data models

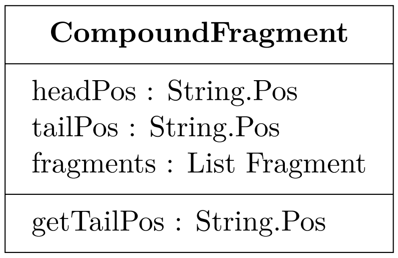


Figure 3.3: Data model of `CompoundFragment`

overlapping fragments. Because the list is only sorted based on the head position of each fragment, it may still contain fragments that overlap. One case where an overlap is non-avoidable is with nested tactics. In this case, the encapsulating tactic fragment overlaps with every nested tactic.

LeanInk generates a list of `CompoundFragments` to resolve this issue and simplify the implementation of the output generation. A `CompoundFragment` defines two properties (Figure 3.3). First, a `headPos` and `tailPos` property of the compound, describes the location in the source text the contained information takes effect. The third property is the list of overlapping fragments starting from the `headPos` location. Figure 3.4 describes the effect of the algorithm visually. The algorithm's input is shown above the arrow, displaying seven fragments that overlap at some positions in the syntax range. Below the arrow is the output of the compound generation algorithm. The result is a list of non-overlapping `CompoundFragments`. Therefore, this representation describes a partition of the source text with metadata annotations at each interval of a compound fragment. This property is helpful for the output generation of LeanInk because it ensures that the output generator can work linearly and does not have to care about annotating the same source text multiple times for multiple metadata annotations. It also allows the output generator to pick the required metadata at a specific position directly instead of searching through the complete list of fragments.

Algorithm 2 outlines the implementation of the compound generation algorithm

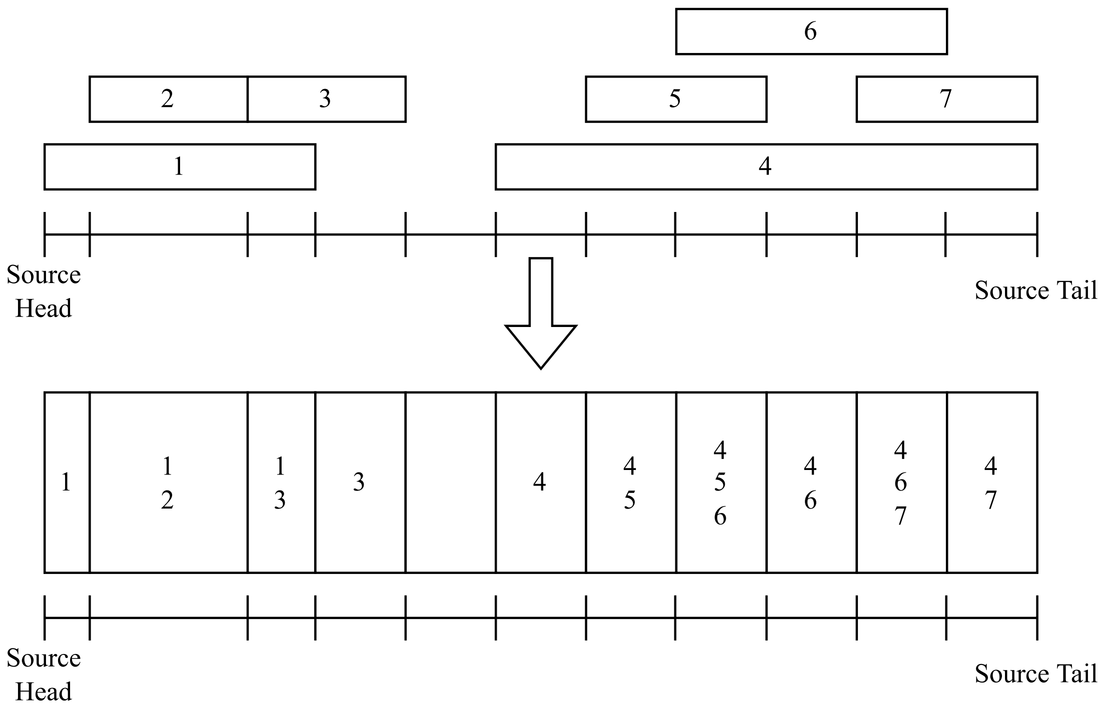


Figure 3.4: Input and output of compound generation

that achieves the transformation from an input of overlapping `Fragments` into a list of non-overlapping `CompoundFragments`.

- (1) The algorithm first creates a sorted list of indexed events from the `fragments` input list. More specifically, it creates a `HeadEvent` and a `TailEvent` for each fragment and assigns them their respective position and an index to identify a pair of head and tail events later. An example for the indexes is shown in Figure 3.4, where the order of the input list derives the index. The `genFragmentEvents` method then sorts the list of tail events and merges both lists into a single list of fragment events. However, this point in the implementation could benefit from future optimization by directly outputting a list of events instead of a list of `Fragments` in the tree traversal algorithm. The result is a list of events sorted in ascending order based on the event's position. After that, LeanInk initializes a first empty compound with the position set to the start of the source text. It then iterates over the list of fragment events.

- (2) For each event, the algorithm first creates a copy of the previous compound and adjusts its position to the position of the current event.

- (3) In the case of a `HeadEvent`, it inserts the fragment of the event to the newly generated `CompoundFragment`. In the case of a `TailEvent`, it removes the fragment of the event from the newly generated `CompoundFragment`. The insert and remove operations use the index of the event to identify the fragment

Algorithm 2 Compound Generation

```
function GENERATECOMPOUNDS($fragments$)
    $events$ :=GENFRAGMENTEVENTS($fragments$)                        ▷ ①
    $compounds$ := [GENEMPTYCOMPOUND]
    while $\neg events.isEmpty$ do
        $event := events.dropFirst()$
        $compound := compound.last.copy()$                           ▷ ②
        $compound.pos = event.pos$
        if $event.type = HeadEvent$ then                             ▷ ③
            $compound.insertFragment(event)$
        else
            $compound.removeFragment(event)$
        if $compounds.last.headPos \neq event.pos$ then              ▷ ④
            $compounds.last.fragments := compound.fragment$
        else
            $compounds.append(compound)$
    return $compounds$
```

in the compound. Otherwise, it could not remove a previously inserted fragment as it could not identify it in the list of fragments of the previous compound.

- ④ Finally, the algorithm updates the previous compound if the event's position is the same as the position of the previous compound. Otherwise, it appends the new compound to the `compounds` list. This check prevents more than one `CompoundFragment` from appearing at the same source text position, which would violate the partitioning property of the result.

For any compound in the list, the compound's `tailPos` is either the `headPos` of the next compound or the last `tailPos` of any of its fragments, whichever comes first. The last compound fragment is implicitly bound by the end of the source text. The algorithm's runtime is $O(n \log n + 2n)$, whereby $O(n \log n)$ is due to the initial sorting of the `TailEvent` list.

Output Generation

The final step in LeanInk receives a list of `CompoundFragments` and generates the output based on the specified `Output` method. LeanInk currently only supports a single `Output` method for use with Alectryon. However, implementing an additional `Output` method requires only a function that takes the list of `CompoundFragment` as its input. To use the new method, the user has to specify this function in an instance of the type `Output` and hand it over to the `runAnalysis` method in LeanInk.

In the case of the Alectryon `Output` method, we iterate over each compound fragment and transform it into an `Alectryon.Fragment`, introduced in Figure 2.2. LeanInk generates a `Text` instance whenever the list of fragments of a

`CompoundFragment` is empty. If the list is not empty, it generates a `Sentence` instance and populates it with the provided list of `Fragments`. Each `MessageFragment`, adds a `Message` to the `messages` property of the `Sentence` instance. Resolving the list of goals and hypotheses requires using the `TacticFragment.smallest?` function that returns the smallest `TacticFragment` used to populate the data in the instance. The smallest `TacticFragment` is the `TacticFragment` with the shortest syntax range.

We extract the raw source text using each compound's `headPos` and `tailPos`. Finally, the respective JSON-based `.lean.leanInk` file is generated by adopting the `ToJson` class and then outputted in the same folder of the input file. Alectryon then reads and processes it using the Lean 4 driver as described earlier.

3.2 Alectryon extension

Now that we have a fundamental understanding of how LeanInk works and how it implements the support of the base feature set for Lean 4 in Alectryon, we take a look at how we extended this functionality even more. We first detail which additional features we implement to make Alectryon more useful. We then present our additions to LeanInk to support the extraction of the additional metadata necessary to support the features. Finally, we describe our changes to Alectryon's transform pipeline, data model, and backends.

3.2.1 User interface

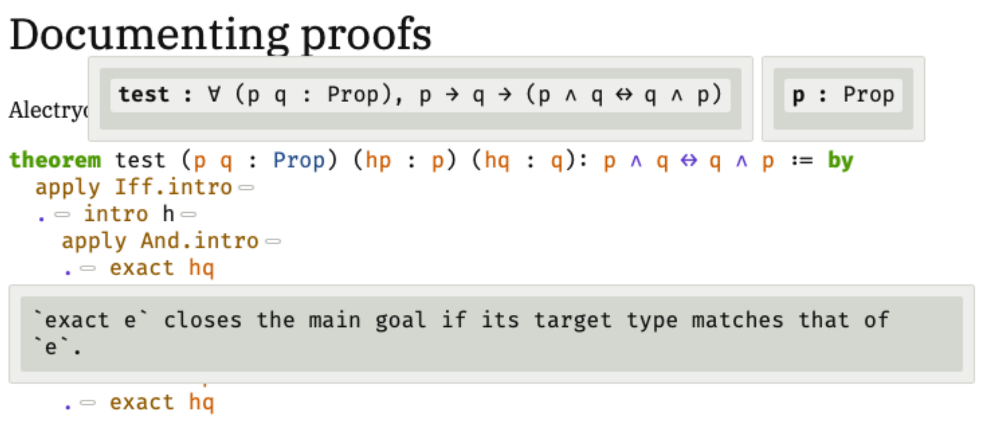
The extension of Alectryon's functionality includes the following additional features.

The first new feature is support for displaying additional token information. This information includes type information about variables, functions, and more, as well as support for displaying documentation, also called `docStrings`, of functions, classes, and other types. Both metadata information helps a user to explore a proof. The type metadata gives hints about how the proof is structured and about what the proof attempts to prove. The `docStrings` improve the discoverability of the proof as it allows the user to dig deeper into what specific tactics or function calls do.

Lastly, we add support for semantic syntax highlighting for Lean 4. Semantic syntax highlighting improves the overall readability of the proof, as it highlights specific tokens in the source text more accurately and therefore implicitly conveys semantic meaning to the reader.

Both additional features also improve the readability of general-purpose programming snippets because they offer a way to interact with the code similar to typical code editors.

Figure 3.5 presents an example with both features enabled. Semantic syntax highlighting accentuates tokens using different colors based on their semantic meaning. In addition to that, Alectryon now displays popovers above the source text tokens containing the new metadata. This example presents multiple hover events simultaneously. The first two hover events present the type information for the `test`



Documenting proofs

Alectry

```
test : ∀ (p q : Prop), p → q → (p ∧ q ↔ q ∧ p)
```

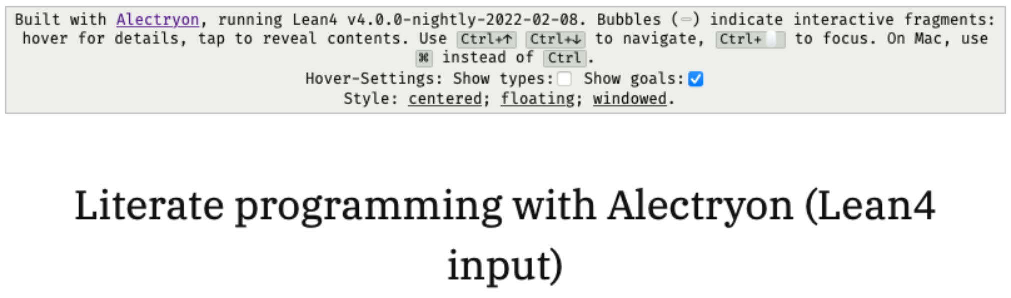
```
p : Prop
```

```
theorem test (p q : Prop) (hp : p) (hq : q): p ∧ q ↔ q ∧ p := by
  apply Iff.intro
  . intro h
    apply And.intro
    . exact hq
```

```
`exact e` closes the main goal if its target type matches that of
`e`.
```

```
    . exact hq
```

Figure 3.5: Additionally information popups and semantic syntax highlighting



```
Built with Alectryon, running Lean4 v4.0.0-nightly-2022-02-08. Bubbles (⊂⊃) indicate interactive fragments:
hover for details, tap to reveal contents. Use Ctrl+↑ Ctrl+↓ to navigate, Ctrl+ to focus. On Mac, use
⌘ instead of Ctrl.
Hover-Settings: Show types:☐ Show goals:☑
Style: centered; floating; windowed.
```

Literate programming with Alectryon (Lean4 input)

Figure 3.6: Update Alectryon header with hover settings

theorem and the parameter `p`. The last hover event displays the `docString` for the `exact` tactic call. One downside of the popovers is that the representation of the proof can become polluted and therefore harder to read, especially for tokens that both display the additional information alongside the tactic popovers. To resolve this issue, we add a new `Hover-Settings` section into Alectryon's header, as seen in Figure 3.6. This new section allows the user to either activate or deactivate both types of popovers in Alectryon's `HTML` presentation. Alectryon deactivates the new type of popover by default o preserve the old behavior.

3.2.2 Implementation design

To implement support for all of these new features, adjustments to the internal data model of Alectryon are necessary. Figure 3.6 reveals the adjustments to the data model. Comparing this class diagram to the initial data model described in Figure 2.2 exposes the key difference in the `Fragment` interface. Instead of storing a `String` attribute for the contents of each `Fragment`, we now store a list of `Token`. The `Token` class stores information about the raw source text (`raw` attribute) and additional, but optional, other attributes. The `link` property defines a clickable link to an external

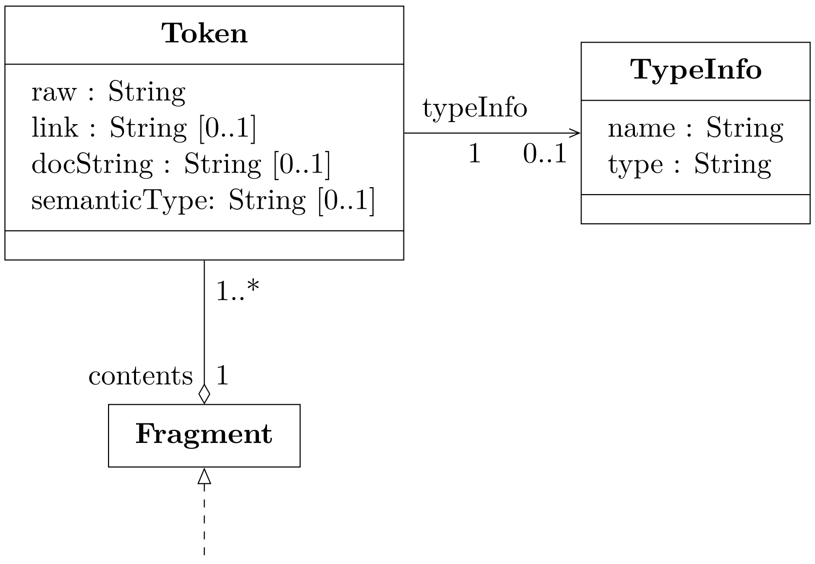


Figure 3.7: Updated data model of Alectryon

source. The `docString` and `typeinfo` attribute store the additional metadata as described earlier. Finally, the `semanticType` property store the `Pygments TokenType` of the token. `Pygments`[3] is the syntax highlighting framework used by Alectryon to support basic syntax highlighting for code snippets.

However, changing the contents field from a `String` type to our more complicated list of `Tokens` is not a straightforward task. The change becomes increasingly more difficult because Alectryon's transform pipeline heavily depends on various `str` and `Regex` operations to search and modify the fragments returned from the drivers. On the other hand, processing the additional metadata via an external data queue instead of propagating it through the transform pipeline would not be feasible because we would later have to reassign the metadata to a transformed source text. This is the key reason for changing the `Fragments` contents property from a simple string to a list of tokens.

To ensure that the data model change does not break the existing pipeline, a wrapper data class around our `Token` list is introduced that implements specific `str` and `Regex` operations. In addition to this, a suite of unit tests is implemented to make sure the newly introduced operations behave as expected. This is especially important in the Python programming language as it is not a statically typed language. All occurrences of operations for the previous contents type are replaced with the new operations of the new data class. Finally, a new step is added that makes sure the input array of `Fragments` uses the new data model for the rest of the pipeline, so all other drivers are automatically migrated to the new data model, requiring no changes to the drivers themselves.

[3]`https://pygments.org`

The final step to implement the support for the new metadata is to adjust Alectryon's `Backends`. Each `Backend` previously expected that the content property of a `Fragment` is of type `str`. This case previously called Pygments to highlight the source text and output it in a valid format. However, to integrate the new data class into the Backends a new function, `gen_token`, is added in Alectryon's backend interface. `gen_token` is called once for each token in the token array of each `Fragment`.

In the case of the `HTML` backend, the `gen_token` also calls an internal function `gen_typeinfo` that generates the new popovers above the source text. We reuse the existing `CSS` implementation in Alectryon to implement the styling of the new popovers and their contents. Additionally, we implement the new header settings in the generated website using JavaScript bindings and two new `CSS` classes called `type-info-hidden` and `output-hidden` each hiding one kind of popover.

3.2.3 LeanInk extension

We are now looking back at how to implement the support of these new features in LeanInk. First, we describe how we extract the type information and `docStrings` from the Lean 4 compiler. We introduce LeanInks internal data structure enhancements to support the more fine-grained data. Finally, we describe the extraction process for the support for semantic syntax highlighting.

All new features are deactivated by default to prevent usage until the changes in Alectryon are merged. As a result, the command-line interface of LeanInk now offers three new flags. Each flag enables the output and extraction of one of the three new metadata types. `--x-enable-type-info` enables the output of type information of tokens within the input source text. For example, parameters of a function or variables contain information about their type that LeanInk extracts. `--x-enable-docStrings` enables the extraction of `docStrings`. `docStrings` are special comments within Lean code to document a specific part in the code. Finally `--x-enable-semantic-token` enables the extraction of semantic token information. Alectryon uses that to implement semantic syntax highlighting. LeanInk defaults to the old data model of Alectryon if no flags are set because the changes in Alectryon automatically ensure that the new data model is used within the transform pipeline and to support older versions of Alectryon.

`InfoTree` traversal adjustments

The initial implementation of LeanInk made sure that the `InfoTree` traversal algorithm only required a single pass-through. Therefore, the metadata extraction of the type information, `docStrings`, and semantic types is implemented in the same tree traversal pass. In addition to saving a list of `Fragments` based on `TacticInfo`, we introduce a list of `TokenInfos`, that stores the new metadata. LeanInk uses the new list for the additional metadata. As per definition, a `Fragment` or `Tactic` may span over multiple `Tokens`. So separating both types during extraction allows LeanInk to independently process both types of metadata.

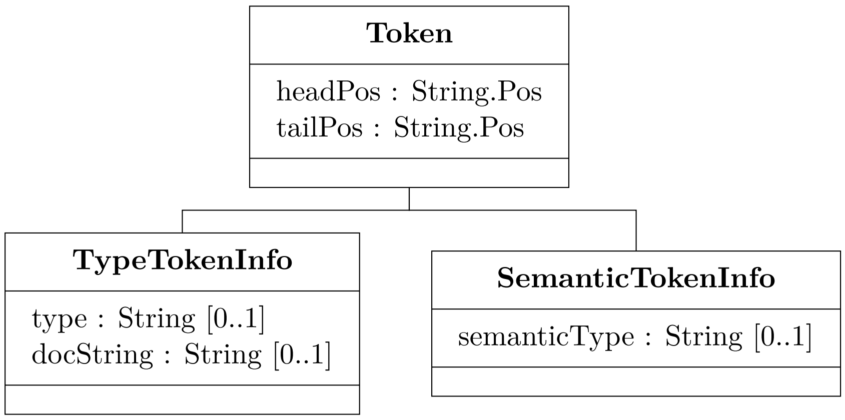


Figure 3.8: New token data model

Figure 3.8 depicts the data model for the new metadata information. A `Token` is similar to a `Fragment` defined by only a syntax-range. It describes a specific property in a small range of the source text. A subclass of `Token` then provides the additional metadata. In this case, `Token` currently has two subclasses, `TypeTokenInfo` and `SemanticTokenInfo`. The `TypeTokenInfo` contains the `docString` if available and the additional metadata about the tokens type. The `SemanticTokenInfo` provides information about the semantic type used for semantic syntax highlighting. Additional metadata can be extracted for future features by implementing another `Token` implementation.

Instead of only receiving our information from `TacticInfo` nodes, LeanInk now also extracts data from `TermInfo` and `FieldInfo` nodes. To populate the `TypeTokenInfo`, we extract the metadata from the smallest `TermInfo` or `FieldInfo` node for each branch in the tree. Again, we define the smallest node at a specific position in the source text as the node with the shortest syntax range that still contains the position. We also require this node to not be part of an expanded macro. LeanInk uses the property of the `InfoTree` to implement the comparison of the smallest node. A parent nodes syntax range includes all syntax ranges of child nodes. So if a child node is used to populate a TypeTokenInfo, the parent can no longer be the smallest node.

For the `SemanticTypeTokenInfo` we analyze the referenced syntax for each node in the tree. We use the `stx` property of each node to extract the ranges of semantic tokens in the source text.

The implementation of the new metadata extraction is contained only within the `genFragments` method. This method now returns both a list of generated `Fragments` and generated `Tokens`, instead of a single list of `Fragments`.

Finally, we propagate this new data with the list of fragments to the compound generation step.

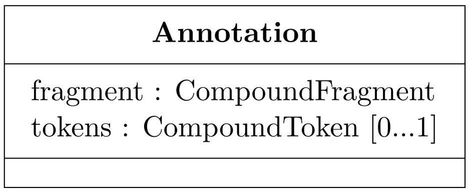


Figure 3.9: New return model of compound generation

Compound generation adjustments

The compound generation algorithm now receives a list of `Fragments` and another list of `Tokens`. As a result, the return model of LeanInk's compound generation also required an adjustment. Figure 3.9 displays the new model returned by the compound generation algorithm. It now returns a list of `Annotation` instances.

An `Annotation` instance contains a single `CompoundFragment` based on the `Fragment` type. This property is equivalent to the early implementation of the algorithm. However, in addition to the `fragment` property, the instance also contains a new `tokens` property. The `tokens` property defines a list of compounds based on the new `Token` type. It contains all `Tokens` in the source text range of the fragment. Additionally, the list is also transformed into a list of compounds to resolve overlaps of, for example, `SemanticTokenInfos` and `TypeTokenInfos`.

LeanInk now uses the algorithm to generate a compound list for the `Fragments` and afterward matches the new list of `Tokens` to each compound using the same algorithm again.

Output generation adjustments

The Alectryon output method only requires changes in generating the `contents` field of each `Alectryon.Fragment`. Instead of directly extracting the source text based on the syntax range of each `CompoundFragment`, it now has to extract the source text based on the list of token compounds in the fragment. For each token compound, LeanInk first searches for the smallest token information within this compound to retrieve the smallest `TypeTokenInfo` and `SemanticTokenInfo`. This is more difficult then the previous tactic info search because the syntax range is bound by the enclosing `CompoundFragment` and the token compound. As a result, LeanInk has to fill ranges that do not have any token compounds and trim the `Token` range that overlaps with the syntax range of its enclosing `CompoundFragment`.

4 Evaluation

This chapter evaluates the implementation, described in Chapter 3. First, a thorough analysis of an example `.html` output file, generated using Alectryon and LeanInk, outlines the improvements in readability of Lean 4 source code. Finally, an evaluation of both the performance and the filesize of Alectryon's output and LeanInks intermediate output reveals drawbacks and opportunity for improvements of the current implementation.

4.1 Output evaluation

Before evaluating the performance and filesize, this chapter first analyzes the visual output and readability improvements of an example `.html` output file generated by Alectryon and LeanInk. The example input file contains the following Lean 4 code:

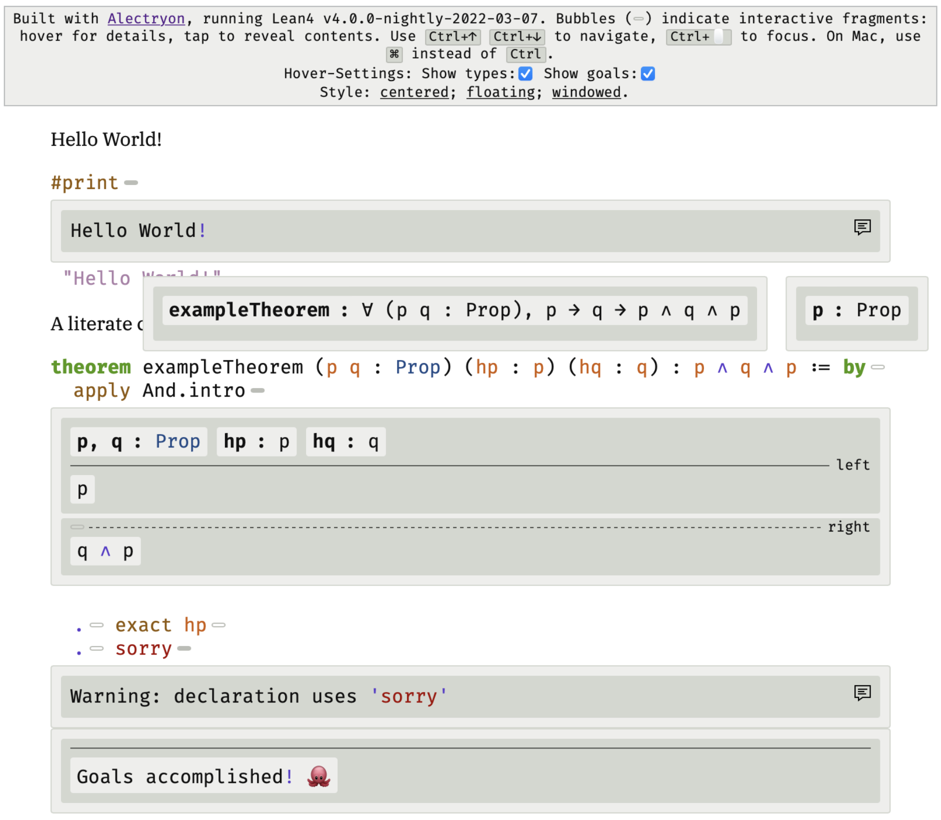
```
/-|
Hello World!
-/
#print "Hello World!"

/-|
A literate comment!
-/

theorem exampleTheorem (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p := by
  apply And.intro
  . exact hp
  . sorry
```

The resulting output website is presented in Figure 4.1. In this figure the hover events are manually triggered by the browser to show the variety of different kinds of information the output is able to display. As a result, in a normal use case, only a single popover is visible at a time. Starting from the top of the output file, the interleaved reStructuredText output is visible. The literate programming support that enables this feature, improves the readability of comments in the source text. However, the additional formatting features of reStructuredText outperform the standard comments syntax of LeanInk, hence a more natural and clearly structured document can be generated.

Messages, for example caused by `print`, are easily discoverable by the user. Especially the use of `sorry` is clearly marked as red within the source text, visualizing



```
Built with Alectryon, running Lean4 v4.0.0-nightly-2022-03-07. Bubbles ( ) indicate interactive fragments:
hover for details, tap to reveal contents. Use Ctrl+↑ Ctrl+↓ to navigate, Ctrl+  to focus. On Mac, use
⌘ instead of Ctrl.
Hover-Settings: Show types: ✓ Show goals: ✓
Style: centered; floating; windowed.
```

Hello World!

```
#print

Hello World!

"Hello

A literate c

exampleTheorem : ∀ (p q : Prop), p → q → p ∧ q ∧ p      p : Prop

theorem exampleTheorem (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p := by
  apply And.intro

p, q : Prop   hp : p   hq : q
──────────────────────────────────────── left
p
- - - - - - - - - - - - - - - - - - - - right
q ∧ p

  . exact hp
  . sorry

Warning: declaration uses 'sorry'

────────────────────────────────────────
Goals accomplished! 🐙
```

Figure 4.1: Example output

Configuration	$\overline{time}_{\text{full}} \pm \sigma$	$\overline{time}_{\text{trav}} \pm \sigma$	$\overline{time}_{\text{cg}} \pm \sigma$
b	$1.194\,s \pm 0.012\,s$	$1.178\,s \pm 0.017\,s$	$1.303\,s \pm 0.056\,s$
b+t	$1.426\,s \pm 0.016\,s$	$1.348\,s \pm 0.015\,s$	$1.427\,s \pm 0.020\,s$
b+t+d	$1.444\,s \pm 0.030\,s$	$1.354\,s \pm 0.021\,s$	$1.403\,s \pm 0.012\,s$
b+t+s	$1.476\,s \pm 0.017\,s$	$1.367\,s \pm 0.016\,s$	$1.443\,s \pm 0.017\,s$
b+t+d+s	$1.478\,s \pm 0.019\,s$	$1.379\,s \pm 0.029\,s$	$1.441\,s \pm 0.016\,s$

Table 4.1: LeanInk benchmark results

Configuration	$\overline{time}_{\text{exec}} \pm \sigma$
o-b	$1.839\,s \pm 0.022\,s$
n-b	$1.827\,s \pm 0.016\,s$
n-b+t	$2.843\,s \pm 0.040\,s$
n-b+t+d	$2.877\,s \pm 0.032\,s$
n-b+t+s	$2.981\,s \pm 0.035\,s$
n-b+t+d+s	$3.002\,s \pm 0.028\,s$

Table 4.2: Alectryon + LeanInk benchmark results

a warning of its usage.

In addition to this, semantic syntax highlighting improves the readability by coloring tokens of a specific category in the output. An example for this is the coloring of all variables and parameters that are defined by the theorem using the color orange. Hence, the user can implicitly recognizes the category of a token within the source text. Furthermore, with the support of semantic syntax highlighting, new keywords and future changes to the syntax of Lean 4, are automatically adapted by LeanInk, without the need to adjust the static syntax highlighter.

Finally the available popovers improve the discoverability of the source text. Type popovers allow the inspection of the source code and give more insights into the behavior of the implementation. On the other hand, tactic popovers display intermediate information of each step in the proof. In the example the tactic popover at `apply And.intro` displays all hypotheses at the location and both cases that are required for the proof to be completed. Lastly, the accomplishment of all goals is indicated by a short message `Goals accomplished!`.

All these additional features and information help the user to understand a proof or code snippet of Lean. Additionally, the website can now be shared and published on the internet, making it easy to both update the content later if necessary, while also making to content accessible to anyone.

However, although all these features improve the readability of the source code, the fundamental knowledge of the basic syntax and functionality of Lean is still required to fully comprehend the proof.

4.2 Performance

After evaluating the readability improvements, the following paragraphs now analyze the performance of both the implementation of LeanInk as well as the extension of Alectryon. The test device that runs the performance benchmarks in this chapter is a MacBook Pro (13-inch, 2018) with a 2.7 GHz (Turbo Boost of 4.5 GHz) Quad-Core Intel Core i7 8559U with 16 GB of 2133 MHz LPDDR3 memory. The operating system installed on this test device is macOS Monterey 12.3 (21E230). `hyperfine`[17] is the benchmarking command-line tool that benchmarks and collects the sample data for the performance analysis. Each LeanInk configuration runs with a warmup of five cycles, and a subsequent total of 50 cycles for data collection. Following the data collection, it then calculates the mean ($\overline{x}$) and standard deviation (σ) based on the collected samples.

The LeanInk instance is compiled and based on the Lean 4 version `4.0.0-nightly-2022-03-07` [15]. Additionally, the Alectryon instance runs on Python `3.9.10`.

The input for all test runs is the `Nat` implementation file `Basic.lean` which is located in the Lean 4 compiler repository under `lean4/src/Init/Data/Nat/` [15]. It is a relatively large Lean 4 file spanning 669 lines, which implements basic theorems and implementations for the `Nat` datatype bundled with the Lean 4 `Init` library.

Table 4.1 and Figure 4.2 depict the results of the performance benchmark for LeanInk. The configuration defines the usage of the `analyze` command. `b` describes the default behavior of the `analyze` command without any additional flags set. `t` sets the `--x-enable-type-info` flag, `d` sets the `--x-enable-docStrings` flag, and `s` sets the `--x-enable-semantic-token` flag. Therefore each of them enables one class of additional metadata, that is being extracted by LeanInk.

In addition to the overall mean runtime $\overline{time}_{\text{full}}$, both the mean runtime of $\overline{time}_{\text{trav}}$ and $\overline{time}_{\text{cg}}$ are depicted. $\overline{time}_{\text{trav}}$ is the mean runtime that includes the `InfoTree` traversal. $\overline{time}_{\text{cg}}$ includes the `InfoTree` traversal and the compound generation, however not the output generation.

The `b` configuration is the fastest configuration of LeanInk. This is because it only extracts the metadata for tactics and goals within a proof. Therefore, the configuration skips the additional compound generation for `Tokens` and the additional extraction steps during the `InfoTree` traversal. Analysis of the call stack show, the majority ($\approx 70\,\%$) of the execution time is due to the `Lean.Elab.processHeader` and `IO.processCommands` method calls, hence the compilation of the source file.

However, in the case of all other configurations both additional steps are required during analysis. In all theses configurations the actual difference only lies in the amount of `Tokens` that are extracted during the `InfoTree` traversal. These configurations show an overall increase of about $0.2\,s$ compared to the `b` configuration. However the increase of metadata extracted only slightly increase the runtime. Further analysis of the execution shows, the actual increase of execution time, is caused by the `Meta.ppExpr` pretty printer method call, which LeanInk uses to generate a readable output string of the inferred type of a `Token`. This discovery explains

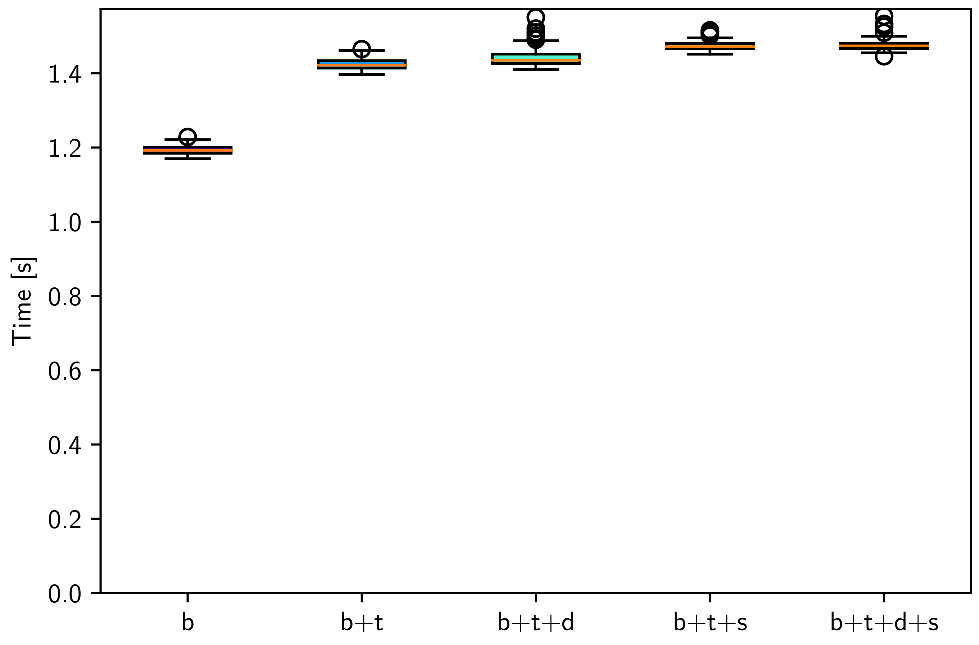


Figure 4.2: LeanInk benchmark

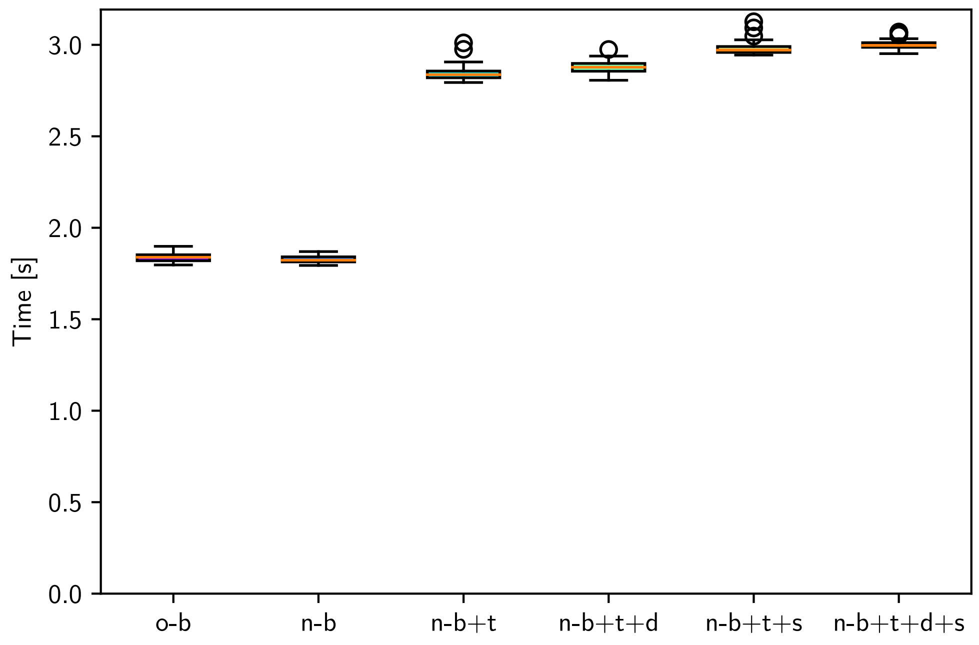


Figure 4.3: Alectryon and LeanInk benchmark

Configuration	`Basic.lean.leanInk`	`Basic.html`
o-b	177 kB	394 kB
n-b	177 kB	412 kB
n-b+t	886 kB	1.404 kB
n-b+t+d	916 kB	1.466 kB
n-b+t+s	992 kB	1.482 kB
n-b+t+d+s	1.037 kB	1.566 kB

Table 4.3: LeanInk + Alectryon filesize benchmark results

why the increase in runtime is similar for all non-default configurations, instead of scaling linearly with the amount of extracted `Tokens`. Consequently, the compound generation algorithm only has a minor influence in the overall runtime of LeanInk.

The performance analysis of Alectryon and its integration with LeanInk uses the same nomenclature for the configuration. Additionally, the `o` field defines the previous implementation of Alectryon mentioned in the first half of Chapter 3 with only support for the `b` configuration. This implementation did not alter the implementation and behavior of Alectryon's transform pipeline. Finally, the `n` field defines the new version of Alectryon, which includes the support for the new metadata and the changes to the transform pipeline and data model.

Table 4.2 and Figure 4.3 show the results of this analysis. The results clearly show no significant difference in execution time between the `o-b` and `n-b` configuration. Consequently, the changes to the transform pipeline and data model had no impact in the overall performance of Alectryon's default behavior. A reason for the similar behavior is the implementation design of Alectryon's new `Token` type. The execution of Alectryon using its default behavior, results only in a single `Token` for each `Fragment`, which is equivalent to the old implementation of a single `String` per `Fragment`.

All other configurations show a increased execution time compared to the default behavior. However, in addition to this, the execution time overhead caused by the increase of metadata is more significant than the difference in LeanInk shown by Figure 4.3. The significant increase in execution time in Alectryon is therefore only partially due to the increase of execution time of LeanInk. Another significant portion is caused by Alectryon's transform pipeline and backend implementations. Both modules have at least linear time complexity, hence their impact on the runtime scales at least linearly with the increase of metadata.

4.3 Filesize

In this last chapter, the filesize of Alectryon's and LeanInk's output formats are evaluated. Table 4.3 presents the filesize for each configuration. The `Basic.lean.leanInk` file is the output file generated by LeanInk. The other output file `Basic.html` is the output file generated by Alectryon and LeanInk using Alectryon's HTML backend.

The diagram clearly shows the significant increase of filesize caused by the additional metadata. The filesize of LeanInk's output increases by the factor of 5 when compared to the configurations `n-b` and `n-b+t`. All other subsequent increases of metadata similarly scale LeanInk's and Alectryon's output.

One final observation is the difference of filesize in Alectryon's output of `o-b` in comparison to `n-b`. Although LeanInk's output size stays similar between both configuration, Alectryon's HTML backend still causes an increase of filesize. This is due to data model change in Alectryon. The backend now receives a list of `Tokens` instead of a single `str` instance. As a result the HTML backend wraps the content of each `Token` into a separate `<div>` in the HTML output file.

5 Conclusion and future work

The final chapter concludes this thesis by taking into account the initial problem domain and comparing it to the results of the evaluation in Chapter 4. Finally, it points out opportunities for future work to expand the current functionality of LeanInk and Alectryon, and their integration with other tools.

5.1 Conclusion

Chapter 4 shows that the implementation of Lean support for Alectryon greatly improves the shareability and readability of Lean 4 code. Literate programming enables users to use proofs written in Lean 4 directly in a text document. Additionally, the presented Lean code in this document is highly discoverable and rich in information.

One major benefit of this approach is, that the shared code, does not require a working development environment and is therefore independent of a runtime. Hence, the code can easily be uploaded to a webserver and shared to other people all around the world. This greatly lowers the initial burden of writing proofs in Lean and submitting the code to other fellow mathematicians. In addition to this, the code contains additional information and hints about the functionality and implementation of the proof.

However, due to the extension of Alectryon, sharing code with other Lean developers is also an area that sees great improvements with this integration. Similar to mathematicians, they can easily share code snippets with other. The additional metadata about type information and semantic syntax highlighting allows them to understand the code without the need of running it.

As a result, the initial problem domain mentioned in Chapter 1 greatly decreases in severity. Although not all issues and problems have been resolved by the integration of LeanInk in Alectryon, the burden of overcoming these issues has dropped significantly.

5.2 Future work

There are some additional features and future work to improve the current implementation of LeanInk and its integration with other tools.

5.2.1 Integration with doc-gen4

Doc-gen4[1] is another tool currently in development for the Lean 4 theorem prover. It analyzes the compiled Lean 4 `.olean` object files of a project to generate a reference website containing all classes, structures, inductive types, and methods and their respective documentation in the Lean code. Additionally, it creates links between all references, making it easy to generate a helpful documentation reference for Lean 4 projects. A future ambition of doc-gen4 is to implement integration with LeanInk to improve the readability of proofs in its output. For this use case implementing another additional `Output` method in LeanInk would be a good starting point, allowing doc-gen4 to receive the necessary information it needs by using LeanInk as a dependency.

5.2.2 Lean-based document generation

Another future extension of LeanInk capabilities is a Lean 4 based document generator. In this case, LeanInk itself would generate an `HTML` file or similar output on its own instead of relying on Alectryon's implementation. A apparent downside of this is the loss of functionality and feature-set that Alectryon provides. However, on the other side, this addition would allow features that are more closely bound to the Lean 4 environment, for example, support for Lean 4 widgets.

5.2.3 Support of additional metadata in LaTeX backend

The extension of Alectryon's feature set described in Chapter 3 is currently not supported by the LaTeX backend, as this was out-of-scope for this thesis. Implementing the support for the LaTeX backend only requires changes to the LaTeX backend itself, as the rest of the pipeline already supports the new features. One major issue with supporting the additional metadata is that LaTeX does not allow any interaction of popovers similar to HTML. Therefore the LaTeX backend should include an extension of Alectryon's IO annotations feature set to allow the user to then define which additional information should be visible in the generated output to prevent a cluttered and thus a hardly readable output file.

[1]https://github.com/leanprover/doc-gen4

Bibliography

[1] L. de Moura and S. Ullrich, “The Lean 4 Theorem Prover and Programming Language,” in *Automated Deduction – CADE 28* (A. Platzer and G. Sutcliffe, eds.), (Cham), pp. 625–635, Springer International Publishing, 2021.

[2] T. L. Heath and Euclid, *The Thirteen Books of Euclid’s Elements, Books 1 and 2*. USA: Dover Publications, Inc., 1956.

[3] M. J. Beeson, “The Mechanization of Mathematics,” in *Alan Turing: Life and Legacy of a Great Thinker* (C. Teuscher, ed.), pp. 77–134, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[4] D. W. Loveland, *Automated theorem proving: A logical basis*. Elsevier, 2016.

[5] J. H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*. USA: Harper & Row Publishers, Inc., 1985.

[6] C. Pit-Claudel, “Untangling mechanized proofs,” in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020, (New York, NY, USA), p. 155–174, Association for Computing Machinery, 2020.

[7] D. E. Knuth, “Literate Programming,” *Comput. J.*, vol. 27, p. 97–111, May 1984.

[8] H. Geuvers, “Proof assistants: History, ideas and future,” *Sadhana*, vol. 34, pp. 3–25, Feb. 2009.

[9] T. C. D. Team, “The Coq Proof Assistant,” Jan. 2022.

[10] R. Wilhelm, H. Seidl, and S. Hack, *Compiler Design*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[11] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, “The Lean Theorem Prover System Description,” in *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings* (A. P. Felty and A. Middeldorp, eds.), vol. 9195 of *Lecture Notes in Computer Science*, pp. 378–388, Springer, 2015.

[12] S. K. Jeremy Avigad, Leonardo de Moura and S. Ullrich, *Theorem Proving in Lean 4*. 2021.

[13] A. Baanen, A. Bentkamp, J. Blanchette, J. Hölzl, and J. Limperg, “The Hitchhiker’s Guide to Logical Verification,” p. 203.

[14] L. de Moura, J. Avigad, S. Kong, and C. Roux, “Elaboration in Dependent Type Theory,” 2015.

[15] “Lean 4 Compiler source code.” `https://github.com/leanprover/lean4`, Commit: dbe9bf61c5dd13b1ca8ca450d590dfa3f220fe6c, 2022.

[16] D. Goodger, “reStructuredText Markup Specification,” Jan 2022.

[17] D. Peter, “hyperfine, a command-line benchmarking tool, Release: v1.13.0.” `https://github.com/sharkdp/hyperfine`, 2022.

Erklärung

Hiermit erkläre ich, Niklas Fabian Bülow, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift