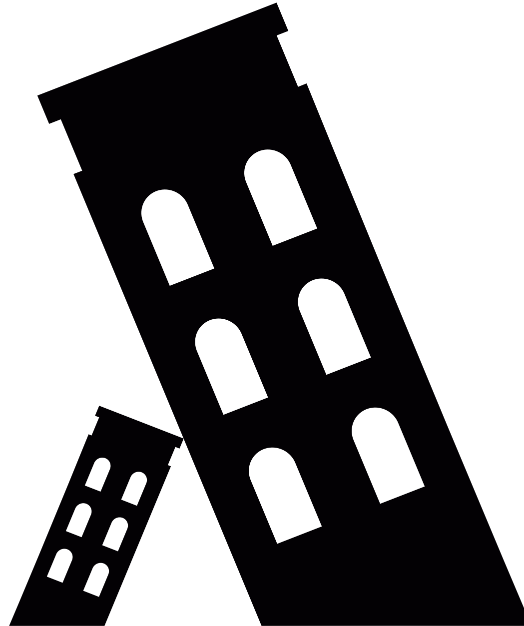
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Pisa

Facilitating conjecture generation in Lean

Master's thesis in Computer science and engineering

Nor Führ
Erik Nygren

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Master's thesis 2025

Pisa

Facilitating conjecture generation in Lean

Nor Führ
Erik Nygren

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Pisa
Facilitating conjecture generation in Lean
Nor Führ
Erik Nygren

© Nor Führ, Erik Nygren, 2025.

Supervisor: Sólrún Einarsdóttir, Department of Computer Science and Engineering
Examiner: Moa Johansson, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A logo for Pisa, showing two Leaning Towers of Pisa in the shape of a λ.
Derivative of "*Leaning Tower Of Pisa SVG Vector*" by SVG Repo, License CC0

Typeset in LaTeX
Gothenburg, Sweden 2025

Pisa
Facilitating conjecture generation in Lean
Nor Führ
Erik Nygren
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As time has passed, the rigour needed for mathematical proofs has increased. Interactive theorem provers were introduced to help with proof verification during proof development. This project aims to aid users while they use a particular interactive theorem prover, known as Lean 4, by facilitating the use of conjecture generation for the language. Conjecture generation can help users understand the domain that they are working in, or automate what proofs might be needed for a more intricate proof.

To facilitate conjecture generation in Lean, a tool called Pisa was introduced. It utilizes tooling from Haskell to generate conjectures. To be able to utilize Haskell tooling for conjecture generation, a translation from Lean to Haskell was created, that supports data type translation and has an interpreter for functions. The translation works for enumerable, recursive, and polymorphic data types. With the translation, generation of conjectures is performed and then translated back into Lean code for the user to prove them. This was all tied into a macro, that is callable from the users' editor, meaning that usage of Pisa can be done while adding and modifying definitions.

Keywords: Transpilation, Conjecture generation, Theory exploration, Interactive theorem prover, Interpreter.

Acknowledgements

Thank you Sólrún Einarsdóttir for suggesting a fantastic project. It has been great having you as our supervisor. You have helped us keep our spirits high.

I would like to thank Nick Smallbone for his time with debugging the usage of QuickSpec, I would not have understood its connection to twee-lib without you.

Further, I would like to thank Mario Carneiro for his introductory course to Lean 4, and for his assistance with introducing Lean's IR to us, this project wouldn't have gotten to where it is without you.

I would also like to thank everyone who helped proofread the paper. Erik and Adam, you have helped to keep the text understandable, by pointing out more fundamental assumptions of the paper. Anna, Carl, Samuel, and Simon for reading through the paper and ensuring that there common thread throughout the paper. William, you reading through the paper while speaking loudly lead to many good catches in our use of the language.

Nor Führ, Gothenburg, 2025-08-22

I will also extend my gratitude to our supervisor, Sólrún. Especially for being so generous with her time when there were many other important things vying for it.

Also, another thanks to Mario. Navigating the not so well documented internals of Lean and its code actions would have been a lot tougher without his guidance.

Erik Nygren, Gothenburg, 2025-08-22

Contents

1 Introduction **1**
- 1.1 Aim . . . 2
  - 1.1.1 Goals . . . 2
- 1.2 Limitations . . . 4
- 1.3 Overview of Pisa . . . 4

2 Background **5**
- 2.1 Dependent types . . . 5
- 2.2 Lean . . . 7
  - 2.2.1 Example of syntax . . . 7
  - 2.2.2 Extensible syntax . . . 9
  - 2.2.3 Typed lambda calculus . . . 9
  - 2.2.4 Intermediate representation . . . 10
- 2.3 QuickSpec . . . 12
  - 2.3.1 RoughSpec . . . 13
- 2.4 Hipster . . . 14
- 2.5 Hopster . . . 15

3 Technical Specification **17**
- 3.1 Lean interface . . . 17
  - 3.1.1 Exporter . . . 18
  - 3.1.2 Macro . . . 19
- 3.2 Haskell interface . . . 20
  - 3.2.1 Translation . . . 20
  - 3.2.2 Interpretation of Haskell . . . 21
  - 3.2.3 Conjecture generation . . . 22
  - 3.2.4 Reverse translation . . . 22
  - 3.2.5 Example . . . 22
- 3.3 Integration of subsystem . . . 24
- 3.4 Partial and unsafe definitions . . . 24
- 3.5 The supported subset of Lean . . . 24

4 Results **27**
- 4.1 Enumerated types . . . 28
- 4.2 Recursive types . . . 31
- 4.3 Polymorphic types . . . 32

5 Conclusion **35**

5.1 Discussion & Future work 35
5.1.1 Alternative approaches 36
5.2 Final remarks 37

Bibliography **39**

A Generated code & conjectures **I**
A.1 Enumerated types I
A.2 Recursive types IV
A.3 Polymorphic types VI
A.4 Precision and recall analysis for Enumerated types X
A.5 Precision and recall analysis for recursive types XIII
A.6 Paired polymorphic conjectures XV

List of Listings

1.1 Conjectures generated in the domain of Booleans with `not` and `and`. 2
2.1 Example of a vector defined with dependent types in Lean. 6
2.2 An alternative definition of `head` to the one in listing 2.1. 6
2.3 Example of inductive types declared in Lean. 7
2.4 Example of how definitions can be declared in Lean. 8
2.5 Example of a theorem proving elimination of double negations. . . . 9
2.6 Example of extensible syntax in Lean. 9
2.7 Pseudo Lean code demonstrating the auto generated function `.rec`. 10
2.8 Illustration of the IR representation for inductive Lean values. . . . 11
2.9 Illustration of the IR representation for enumerated Lean values. . . 11
2.10 Example declaring signatures that QuickSpec can utilize. 12
2.11 Outputs from running QuickSpec on listing 2.10. 12
2.12 Example of RoughSpec templates. 13
2.13 Example declaring signatures that RoughSpec can utilize. 13
2.14 Outputs from running RoughSpec on listing 2.13. 14
3.1 Lean code demonstrating transitive dependencies. 18
3.2 Example output from the exporting tool `pisa-lean`. 19
3.3 Type signature of the interpreter function, called `eval`. 20
3.4 Lists data type written in Lean. 21
3.5 listing 3.4 as converted into Haskell. 21
3.6 Example of auto-generated code based on `N` and `add`. 23
3.7 Output from running QuickSpec and RoughSpec on listing 3.6. . . 23
3.8 Lean translation of listing 3.7. 23
A.1 Benchmark with varying definitions for an enumerated type. I
A.2 The auto-generated translation of listing A.1. II
A.3 Conjectures generated by `#pisa` on the code in listing A.1. III
A.4 Benchmark with definitions for recursive natural numbers. IV
A.5 The auto generated code based on listing A.4. V
A.6 Conjectures generated by `#pisa` on the code in listing A.4. VI
A.7 Benchmark with definitions for polymorphic lists. VI
A.8 The auto-generated translation of listing A.7. VII
A.9 Conjectures generated by `#pisa` on the code in listing A.7. VIII
A.10 Benchmark with definitions for polymorphic binary trees. VIII
A.11 The auto-generated translation of listing A.10. IX
A.12 Conjectures generated by `#pisa` on the code in listing A.10. X

List of Conjecture sets

1 Precision and recall formulas.28

1 Generated by Pisa for the domain $\mathbb{B}$. 29
2 Mathlib equivalent for the domain $\mathbb{B}$. 30
3 Generated by Pisa for the domain $\mathbb{N}$. 31
4 Mathlib equivalent for the domain $\mathbb{N}$. 32
5 Generated by Pisa for the domain `List` α. 33
6 Generated by Pisa for the domain `Tree` α. 33
7 Mathlib equivalent for the domain `List` α. 34
8 Generated by Pisa for the domain $\mathbb{B}$ for comparison to mathlib. . . XI
9 Mathlib equivalent for the domain $\mathbb{B}$ for comparison. XII
10 Generated by Pisa for the domain $\mathbb{N}$ for comparison to mathlib. . . XIII
11 Mathlib equivalent for the domain $\mathbb{N}$ for comparison. XIV
12 Generated by Pisa for the domain `List` α for comparison to mathlib. XV
13 Mathlib equivalent for the domain `List` α for comparison. XVI

1
Introduction

Computer verification of mathematical proofs has become more and more prevalent. This is commonly accomplished using interactive theorem provers (ITPs). ITPs provide an interface for users to iteratively work on a proof, while the proof itself is verified to hold. During formalizations of proofs, an ITP can hide tedious aspects such as rearranging terms due to commutativity (Buzzard, 2024). An ITP will record all necessary steps that constitute a proof, and is then able to verify that the final proof is indeed correct (Geuvers, 2009). It will, however, not necessarily help with the discovery of a proof nor the presentation of it.

Tooling for the discovery of proofs was presented by Buchberger (2000) along with the concept of "Theory Exploration". Conceptually, while working on proofs, one would typically consider more than just a single theorem. The process is better described as an exploration "of an entire theory". In other words, one can extend the understanding of a field by looking at interactions between concepts in said field.

This idea of theory explorations has been further explored by QuickSpec (Smallbone et al., 2017) and then implemented in Hipster (Johansson et al., 2014). QuickSpec is a tool created using QuickCheck (Claessen & Hughes, 2000) that will, given a set of functions, heuristically find conjectures that seem to hold based on auto generated inputs to them. Hipster integrates the "Theory exploration" of QuickSpec into the ITP known as Isabelle/HOL, by way of exporting the current context of proofs and functions. Furthermore, there has been an extension of QuickSpec called RoughSpec (Einarsdóttir et al., 2021). RoughSpec is able to limit the search space of QuickSpec based on given shapes of lemmas. These can be in the form of associativity, commutativity, or identity but can also take other forms.

The idea for this project, called Pisa, is to implement a similar function as Hipster does, but for the ITP and programming language Lean (de Moura et al., 2015). Lean has a growing user-base with a mathematical focus. When exploring a new problem domain, a tool such as this could help users find new ways of thinking, or help users that are new to the field work through their own understanding of the domain.

1.1 Aim

The project aim is to facilitate conjecture generation within Lean, in a tool called Pisa. A transpiler (compilation from one language to another) from a subset of Lean to Haskell is needed in order to utilize existing frameworks in Haskell. QuickSpec and RoughSpec could then be used in conjunction with the transpiler to create Pisa.

Pisa allows a user to find conjectures that may assist in proving aspects in the domain they are working in. For example, if one is working in the domain of Booleans and asking Pisa for conjectures regarding the functions `not` and `and`, one could get conjectures such as the ones seen in listing 1.1. This leaves the user to implement the proofs of the conjectures. Further, no conjecture generation tool exists for Lean 4 as far as we, the authors, know. Pisa would therefore provide the functionality of conjecture generation for Lean.

```
theorem conjecture0                   :       not false = true
theorem conjecture1                   :        not true = false
theorem conjecture2 (x y : Bool)      :         and x y = and y x
theorem conjecture3 (x : Bool)        :         and x x = x
theorem conjecture4 (x : Bool)        :     and x false = false
theorem conjecture5 (x : Bool)        :      and x true = x
theorem conjecture6 (x : Bool)        :     not (not x) = x
theorem conjecture7 (x : Bool)        :   and x (not x) = false
theorem conjecture8 (x : Bool)        :      and true x = x
theorem conjecture9 (x y z : Bool) : and (and x y) z = and x (and y z)
```

Listing 1.1: Conjectures generated in the domain of Booleans with the functions `not` and `and`. The code snippet has been altered to improve readability.

As well as adding support for conjecture generation, a new backend for Lean would be created. The creation of a different backend was outlined as future work by de Moura and Ullrich, 2021. This could also allow for analysis on the generated Haskell code by other tools.

1.1.1 Goals

The problem statement is expanded into the following goals:

(A) Translation: Support for enumerated types

Theorems in Lean are primarily modeled with data types and functions. Simple types, such as Booleans, are essential to perform the bare necessities of Pisa. Without such capabilities, no more complex values can be created. To properly translate any Lean code, Pisa must be able to create a corresponding representation to these enumerable types when generating Haskell code.

(B) Translation: Support for recursive types

Furthering goal (A) is recursive data types. This would allow functions that utilize recursion to achieve their result. Recursive functions are not handled in the same way between Haskell and Lean, and therefore is a necessary part to ensure compatibility. One example of this is the natural numbers.

Recursion is by necessity a more involved process than one using only enumerable data types. It requires the generated code to handle calls to the same function and handling the operations after the sub call is finished. Further, the evaluation strategy that Lean uses is different from Haskell. This difference has to be handled in some way to ensure that the translation is representable of the original.

(C) Translation: Support for polymorphic types

Many simple types are similar and may be represented by a more generic type. For example, `List A` may contain elements of any data type `A`. This abstraction, called polymorphism, may be a little more intricate to create a correspondence when generating code and is not strictly necessary. However, it is commonly used and useful when reasoning about the shared properties that types have. Polymorphism can also lead to conjectures that hold more universally for a concept than instantiated types. Concepts such as eq. (1.1) holds a higher importance for a proof than eq. (1.2).

$$\forall A.\ \lambda a : \texttt{List}\ A \rightarrow \texttt{reverse}\ (\texttt{reverse}\ a) \equiv a \tag{1.1}$$

$$\lambda a : \texttt{List}\ \mathbb{N} \rightarrow \texttt{reverse}\ (\texttt{reverse}\ a) \equiv a \tag{1.2}$$

(D) Conjecture generation: Conjecture about the translation

The translation mentioned in goals (A) to (C), are created in order to facilitate the conjecture generation. Therefore, utilizing them as a base would allow the usage of tooling in Haskell to generate conjectures. With conjectures generated, one could then port them back into Lean for a better user experience.

(E) Tool: Integration of the subsystems

To complete Pisa the different components have to be integrated with each other. A unified tool would greatly improve ease of use as opposed to a bundle of separate utilities the user has to stitch together. A way of utilizing Pisa within a code editor would be a desirable function of the system.

(F) Conjecture generation: Relevance of the generated conjectures

For Pisa to help with explorations of theorems, it needs to produce relevant conjectures. A comparison to existing libraries would therefore indicate how well Pisa performs.

1.2 Limitations

As mentioned in section 1.1, Pisa aims to support a subset of Lean. This is due to Lean being a complete programming language and this thesis is solely focused on the ITP aspects of it. Therefore, a subset was used, and it is explained in section 3.5. However, for use as an ITP Pisa is still able to handle the most general cases. This means that the fundamentals of a user's chosen domain will be able to be found using Pisa.

1.3 Overview of Pisa

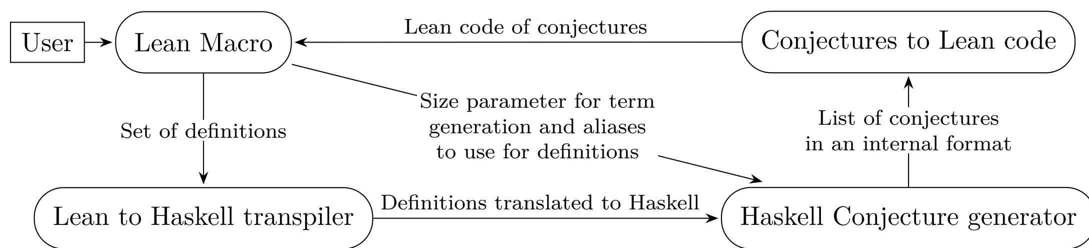


Figure 1.1: An overview of the information flow of Pisa.

Figure 1.1 shows a high level of how Pisa operates. The macro will traverse the structure of definitions provided by the user and collect them for translation. This is done by traversing both an intermediate representation of functions, and a typed lambda calculus for data types. These definitions are sent to the transpiler to generate Haskell versions of the definitions. With a Haskell representation of the Lean code, conjectures are generated utilizing QuickSpec and RoughSpec. Last, the conjectures generated are ported back into Lean where they are provided to the user.

2

Background

2.1 Dependent types

Dependent types extend the common notion of types in programming languages by qualifying a type on other values. This is similar to how polymorphism allows the inverse; values that depend on types (Pierce, 2002). A familiar example involves a representation of vectors. Their type may be contingent on the type of the values they contain, i.e. `Vector α` for any type `α`. Polymorphism enables generalized definitions, such as a function that appends any two similar instances of these vectors. With dependent types, properties can be encoded in the type, such as the vector's length. The type `Vector ℕ 3` informs us that a value contains three natural numbers.

Encoding properties in types becomes particularly useful with the realization of the Curry–Howard correspondence (Nordström et al., 1990). The correspondence relates propositions and their proofs with types and the construction of well typed values of said type. In other words, if a proposition can be represented with a type, finding a value of that type is a sufficient proof of the proposition. By encoding properties in the types, other propositions may be represented. As an example, eq. (2.1) states that the sum of two odd numbers is even. Without dependent types this could not be expressed as a type.

$$\lambda n : \mathbb{N} \rightarrow \lambda m : \mathbb{N} \rightarrow \texttt{odd}\ n \rightarrow \texttt{odd}\ m \rightarrow \texttt{even}\ (n + m) \tag{2.1}$$

This notion can also sometimes be conceptualized as embedding a proof in the type. Using the earlier example of vectors, an inductive Lean definition is provided in listing 2.1. By definition, the only possible construction of a vector with length 0 is the empty vector, `Nil`. Similarly, vectors of length $n + 1$ can only be constructed by adding an element to a vector of length n, using `Cons`. The definition thus ensures the length property is sound. When coming across a vector of length m in an expression accepted by the type checker, it can be presumed to indeed contain m items.

With a proof of the length in the type, functions can be constructed that otherwise would result in $\bot$. This is exemplified by the definition of `head` in listing 2.1. The function describes that it handles the domain of vectors with at least one element (due to the length being non-zero). It is therefore able to return the first element of the vector. Without dependent types the result type has to change. This could be done by wrapping the result in a type containing the value or an exception.

Sometimes this exception is not encoded by the type system and throws a runtime error.

```
inductive Vector : Type → Nat → Type where
  | Nil  : Vector α 0
  | Cons : α → Vector α n → Vector α (n+1)

def head (v : Vector α (n+1)) : α :=
  match v with
    | .Cons a _ => a

def append (v : Vector α n) (w : Vector α m) : Vector α (m+n) :=
  match v with
    | .Nil => w
    | .Cons a v' => .Cons a (append v' w)
```

Listing 2.1: Example of a vector defined with dependent types in Lean.

However, this embedding requires that any function operating on these types must express how it affects the property. A dependently typed definition of `append` for vectors has to describe that the combined vector has the combined length of the arguments. As can be seen in listing 2.1, since the length is a regular value the resulting length can be described by a simple addition of the two obtained in the arguments. With this signature Lean will ensure that the definition indeed does return a vector of the combined length. In this case the proof is trivially realized by expanding the definitions of `Nil`, `Cons`, and `append`. Lean will automatically do this without any further assistance.

What proofs to include in the types mostly depends on which guarantees are needed in the context. The signature of `append` is not sufficient to ensure that all values of the vectors are retained or that the final ordering is as one would expect. A more precise signature could be formulated with another definition for vectors, but may not be necessary in most situations. Likewise, the length of a list is often not necessary or possible to determine.

If a property is mostly irrelevant, the proof may instead be derived separately when needed. This is shown by the alternative definition of `head` in listing 2.2. In this example, the match is complete without a branch on `.Nil` as it would cause a contradiction with the premise that `as` is not empty.

```
def head (as : List α) (proof : as ≠ []) : α :=
  match as with
    | .Cons a _ => a
```

Listing 2.2: An alternative definition of `head` to the one in listing 2.1.
This version keeps the proof is separate from the primary type.

2.2 Lean

Lean is a dependently typed functional programming language that is combined with text editor plugins to make a proper interactive theorem prover (ITP). In this thesis Lean refers specifically to the latest version, Lean 4. It is significantly more extensible than earlier iterations and features a compiler that is able to generate relatively efficient C code (de Moura & Ullrich, 2021). The efficient compiler has enabled it to be mostly bootstrapped, however, the kernel used for verification is yet to be ported from C++.

Three of the concepts utilized by Lean are of particular relevance to this thesis. First, a typed lambda calculus that relates to the source code, encodes definitions, and the types of definition (Leanprover Community, 2025l). The lambda calculus allows for type checking and reductions. Second, a kernel that is a small, trusted computing base. The kernel is used for verification of correctness for Lean programs. The reason why it is small is so that it should itself be verifiable, in order to trust its output. Third, an intermediate representation (IR) which is used for compilation of programs, that allows for evaluation of statements or running entire programs (Leanprover Community, 2025k).

2.2.1 Example of syntax

Familiarity with Lean syntax may help to better understand parts of this thesis. The examples that follows should give a sense of the syntax used in some Lean listings. However, syntax in Lean is very extensible as will be explained later, and the same code may be written in multiple ways.

Algebraic data types can be defined as in listing 2.3. The start of such a declaration use the keyword `inductive`, which signifies a type that induction can be done on. Booleans are, for example, defined as the sum of two trivial values, `t` and `f` representing true and false respectively. Each variant of the sum is syntactically initiated by a vertical bar followed by its name and type.

```
inductive B where | t : B | f : B

inductive P where | p : B → B → P

inductive N where | z : N | s : N → N

inductive L : Type → Type where
  | Nil : L α
  | Cons : α → L α → L α
```

Listing 2.3: Example of inductive types declared in Lean.

All inductives in Lean are a sum of product types. In the second definition a product type, `P`, is exemplified by adding premises, or arguments, to the variant type. In

an inductive definition these arguments may be of the type that is constructed, as shown by `N`. Here, natural numbers are defined recursively as in the Peano axioms; a zero, `Z`, is trivially constructed, and by providing any number n a successive number $n + 1$ is constructed with `S n`.

A type itself may be parametric, that is to say the type itself takes arguments in the form of types. The final example in listing 2.3 demonstrates a definition of a polymorphic list, which takes a type as an argument. In this definition the given type is referred to by a free variable, `α`, that will be resolved by Lean from the context where the constructors are used. In `Cons t Nil` the type can be inferred to lists of Booleans since the first argument to `Cons` is a boolean, which is consistent with `Nil` since it is a list of any type.

Definitions associate any expression with a name. Some examples are given in listing 2.4. Since types are ostensibly values of the type `Type` there is no strict distinction between values and types. Thus, what would be defined as a type alias in languages such as Haskell where values and types are distinguished, is a regular definition, such as the example `NatList`. However for this thesis, definitions are typically functions, which is shown in the three last `def` in listing 2.4.

Case matching on types is done in two different ways: one where it is specified what is matched on, as can be seen in `not` and `add`, and the other where the case can be done instead of a lambda, which is seen in `sum`. The dot seen before the constructors in the `match` is to avoid opening the inductive types scope, that is to say adding the constructor as a function in the current context. The way Lean finds the right constructor using the dot syntax is by looking through the context for a constructor with that name and type.

```
def NatList := L N

def not (b : B) :=
  match b with | .t => .f | .f => .t

def add : N → N → N := λ n m =>
  match m with | .Z => n | .S m' => .S (add n m')

def sum : NatList → N
  | .Nil => .Z
  | .Cons a as => add a (sum as)
```

Listing 2.4: Example of how definitions can be declared in Lean.

Theorems are a special case of regular definitions that are used when proving propositions. The truth value of a proposition is proven by having a resulting type that is a `Prop`. Reflexivity or equality, is the proposition that is relevant for the conjectures generated by Pisa. Reflexivity states that the left and right hand sides are definitionally equivalent and has a special syntax which is `lhs = rhs` (Lean-prover Community, 2025i). The example in listing 2.5 is a theorem proving that

not (not b) is equivalent to b. This verified by the type checker, ensuring that both potential cases of b resolve to the same definition. In the case where b = .t, not (not .t) resolves to .t, and is therefore being able to construct rfl (rfl is one of the constructors of properties). Theorems are the basis of using Lean as a proof engine, since the resulting type should be a truthful statement.

```
theorem not_not (b : B) : not (not b) = b :=
  match b with | .t => rfl | .f => rfl
```

Listing 2.5: Example of a theorem in Lean. In this case proving elimination of double negations by constructing a reflexivity value.

2.2.2 Extensible syntax

A central feature of Lean is that the parser may be extended by the code that itself is parsing (Ullrich & de Moura, 2020). This is a more generalized approach to syntactic sugar supported by many other languages, including Lean 3. By supporting arbitrary syntax Lean can embed domain specific languages appropriate for the context in which a proof is developed. This greatly improves usability as an ITP.

The core idea to this extensibility is to quantify syntax elements and their rules, which is done through syntactic categories, e.g. identifiers, terms, or commands. Additional new categories may be devised, and any category can be extended with syntax rules. There are two kinds of rules, *macros* and *elaborators*. A macro transforms syntax into other syntax, which in turn may be described by another macro. When no macros are applicable to a piece of syntax it is finally evaluated by a matching elaborator. These produce computations with side effects, that have access to the internal compiler state.

Elaborators enable context aware syntax. An example provided by leanprover, 2025b, is the syntax a !: b, which means that a will only evaluate if it is not of the type b. In listing 2.6, !: allows the compiler to verify that the first term is not of a certain type, even if it depends on a larger context.

```
#eval ([1, 2, 3] !: String) -- Evaluates to [1,2,3]
#eval (5 !: Nat) -- Fails type checking due to 5 being of the type Nat
```

Listing 2.6: Example of extensible syntax, which elaborates a term to a different term that may or may not type check.

Elaboration produces expressions for the Lean kernel, that may either be evaluated for soundness or transformed into executable code. The former relying on the typed lambda calculus, and the latter on the IR, both described in the following sections.

2.2.3 Typed lambda calculus

Lean has a dependently typed lambda calculus that is used for type checking which enables the use of dependent types (de Moura et al., 2015). It has fundamental

declarations, but the main ones that are used for this thesis are `definitional`, `constructor`, `inductive` and `recursor` values. These all explain different aspects of the language, where `constructor` is the explanation of how the constructor works, `inductive` for data types, `definitional` for definitions (such as functions), and `recursor` for how recursion is done on inductives (Leanprover Community, 2025l). All the before mentioned, have a type, which is an expression.

Expressions are the lambda calculus in action. The lambda expressions have the standard application, lambda, and variable constructors one would expect. The lambda expressions use de Bruijn Indices, which uses numbers to represent what binder a variable references.[1] Worth of note, is the constructor `Const`. `Const` is how definitions are mentioned, for example in an application, in the current definition.

The `recursor` value is used to represent mathematical induction on `inductive` types. A set of functions are auto generated for each inductive type to use this. An example of an auto generated function is `.rec`. `.rec` is a unique part of the language, that is based on the structure of its inductive counterpart. This enables higher level concepts such as case matching and recursion to be simple to represent. Using the natural numbers as an example, as can be seen in listing 2.7. The first argument of `.rec` is the `motive`, which there may be more than of depending on what criteria is needed for recursion. The `motive` is what the resulting operation should resolve to, for example a boolean. In the case of a boolean the motive would look like the following: `motive : N → B`. Following this is how the cases should result in different resulting values. Last, is the value that recursion is performed on. The return type of `.rec` is dependent on the motive, as mentioned, therefore, the `motive` is applied to the value, to figure out what the resulting type is.

```
inductive N where
  | Z : N
  | S : N → N

N.rec {motive : N → Type}
      (Z : motive Z)
      (S : (a : N) → motive a → motive (S a))
      (t : N) :
      motive t
```

Listing 2.7: Pseudo Lean code demonstrating how the auto generated function `.rec` is added to natural numbers.

2.2.4 Intermediate representation

To construct efficient C code Lean definitions are transformed into an intermediate representation (IR). This additionally enables efficient interactive evaluation of Lean code. Built into Lean is a provided command `#eval`. `#eval` is used to evaluate most Lean expressions, similar to an external interpreter but within the document editor. For example, `#eval 4 + 4` would, as one would expect, print `8`.

[1] There also exists free and meta variables that do not use de Bruijn Indices.

The IR of Lean is based on λPure and λRc, explained in "Counting immutable beans: reference counting optimized for purely functional programming" by Ullrich and de Moura, 2021. The IR could be seen through the lens of C; it deals with a list of sequential instructions and branches of instructions. All values are represented by the following types, described by Leanprover Community, 2025k:

- float
- uint8
- uint16
- uint32
- uint64
- usize
- irrelevant
- object
- tobject
- float32
- struct
- union

For this thesis, values in Lean will usually be encoded as a constructor, based on an `inductive` tagged with an identifying type. In listing 2.8, `n` would be encoded as a constructor with the tag 1 due to being the second constructor in the inductive type. `n` would then have a pointer to the sub constructor `S(Z)` in its list of values associated to the constructor.

```
inductive N where
  | Z : N
  | S : N → N

-- zero =           Constructor (Tag 0) []
-- one  =           Constructor (Tag 1) [zero]
def n := S(S(Z)) -- Constructor (Tag 1) [one]
```

Listing 2.8: Illustration of the IR representation for some Lean values. Numbers constructed using an inductive Peano definitions with comments showing their respective IR representation.

In contrast to how listing 2.8 would be encoded, values of `B` in listing 2.9 would be encoded as an unsigned integer. Since no constructor in `B` requires any arguments, it is interpreted as an enum, since it only needs a tag. `true` would therefore be encoded to a 0, and `false` as 1.

```
inductive B where
  | t : B
  | f : B

def true  := .t -- 0 : uint8
def false := .f -- 1 : uint8
```

Listing 2.9: Illustration `true` and `false` encoding the Boolean value for true and false and comments with their respective IR representation.

2.3 QuickSpec

QuickSpec (Smallbone et al., 2017) is a theory exploration tool. With a list of functions, with their type signatures annotated, QuickSpec is able to test its way to conjectures using said functions. Based on these signatures that can be seen in listing 2.10, one can run QuickSpec on them to retrieve laws such as the ones outlined in listing 2.11. In this example, laws such as commutativity, associativity can be found, but also tautologies, as can be seen in number 8 and 9.

```
signatures :: [Sig]
signatures =
  [ con "0" (0 :: Int)
  , con "+" ((+) :: Int -> Int -> Int)
  , con "not" (not :: Bool -> Bool)
  , con "and" ((&&) :: Bool -> Bool -> Bool)
  , con "or" ((||) :: Bool -> Bool -> Bool)
  ]
```

Listing 2.10: Example declaring signatures that QuickSpec can utilize.

```
== Laws ==
  1. x + y = y + x
  2. and x y = and y x
  3. and x x = x
  4. or x y = or y x
  5. or x x = x
  6. x + 0 = x
  7. not (not x) = x
  8. and x (not x) = and y (not y)
  9. or x (not x) = or y (not y)
 10. (x + y) + z = x + (y + z)
 11. and (and x y) z = and x (and y z)
 12. and x (or x y) = x
 13. or x (and x y) = x
 14. or (or x y) z = or x (or y z)
 15. and (not x) (not y) = not (or x y)
 16. and (not x) (or x y) = and y (not x)
 17. and (or x y) (or x z) = or x (and y z)
```

Listing 2.11: Outputs from running QuickSpec on listing 2.10.

Signatures are what QuickSpec relies on to generate conjectures. For this thesis, the most common signature is `con`. `con` is used to represent a function that should be explored for conjectures. It may need to be specified on what types that the functions operate on, for example when using polymorphic functions since QuickSpec only allows for monomorphic ones.

When using QuickSpec, the user will have to write a list of signatures themselves. For usage in Pisa, these signatures will have to be autogenerated, since the user should not have to interact with the Haskell translation.

2.3.1 RoughSpec

RoughSpec (Einarsdóttir et al., 2021) expands on the work in QuickSpec. RoughSpec allows the user to provide templates of the shape of conjectures, that the user wants to explore. With these templates, RoughSpec is able to limit the search space more than QuickSpec, since QuickSpec will try to find all applicable laws within the search space.

The syntax of templates can be seen in listing 2.12. The `?F` means that there is a meta variable `F` which is a function, meanwhile `X` and `Y` are variables.

```
?F X Y = ?F Y X
?F (?F X Y) Z = ?F X (?F Y Z)
?F (?G X) = X
```

Listing 2.12: Example of RoughSpec templates for commutativity, associativity, and invertible functions.

As an example, using the same functions shown in listing 2.10, listing 2.13 was created. Using these, and the templates, one can generate the laws seen in listing 2.14. As one can see, even though the same functions are used as in the QuickSpec example, there are fewer results than listing 2.11. However, one law is found in this example that is not found by QuickSpec, which is `0 + x = x`. `0 + x = x` can be derived from `x + 0 = x` and `x + y = y + x`, which why it was discarded by QuickSpec.

One of the more important aspects for RoughSpec is the limitation in search space compared to QuickSpec. This allows for faster search times for the end user. Running a test on listing 2.13 on a computer with a AMD Ryzen 7 7735HS, and 16 Gigabytes of RAM at 6400 MHz, in Haskell's interpreter `ghci`, took 0.20 seconds for RoughSpec and 4.70 seconds for QuickSpec.

```
signatures :: [Sig]
signatures =
  [ con "0" (0 :: Int)
  , con "+" ((+) :: Int -> Int -> Int)
  , con "not" (not :: Bool -> Bool)
  , con "and" ((&&) :: Bool -> Bool -> Bool)
  , con "or" ((||) :: Bool -> Bool -> Bool)
  , template "comp-id" "?F (?G X) = X"
  , template "op-id-elem" "?F X ?G = X"
  , template "op-elem-id" "?F ?G X = X"
  , template "commutative" "?F X Y = ?F Y X"
  ]
```

Listing 2.13: Example declaring signatures that RoughSpec can utilize.

```
== Laws ==
Searching for comp-id properties...
  1. not (not x) = x
Searching for op-id-elem properties...
  2. x + 0 = x
Searching for op-elem-id properties...
  3. 0 + x = x
Searching for commutative properties...
  4. x + y = y + x
  5. and x y = and y x
  6. or x y = or y x
```

Listing 2.14: Outputs from running RoughSpec on listing 2.13.

A hybrid system, utilizing both QuickSpec and RoughSpec in tandem, is explored by Einarsdóttir et al., 2021. This allows QuickSpec to find smaller laws, while also allowing more specified templates for RoughSpec. RoughSpec is thus able to utilize the more general laws found by QuickSpec, rather than having the user provide general templates. Einarsdóttir et al. note that this helps pruning away larger properties, that are less elegant. An example of a less elegant property is `length [] + y = y`, which could expressed with `length [] = 0` and `0 + y = y`.

2.4 Hipster

Hipster is a similar project to Pisa, as in it adds a theory exploration tool into the ITP known as Isabelle/HOL (Johansson et al., 2014). This is accomplished using the following main parts. First, Isabelle/HOL has a built-in code generator that is able to translate definitions into Haskell. This allows the use of the second part, which is QuickSpec. QuickSpec allows Hipster to find conjectures on the definitions, and these conjectures are then translated in to the proof style of Isabelle/HOL. These conjectures are then attempted to be automatically proven by tooling in Isabelle/HOL, and are then presented to the end user.

Comparing this to Pisa, the main difference to this thesis is the presence of a code generator to Haskell. Since Lean does not have a code generator to Haskell, one has to be implemented to attempt the same approach that was taken by Hipster.

There is a difference in how functions that are partially defined are handled between Isabelle/HOL and Lean. In Isabelle/HOL, partially defined functions will, if using an unspecified input, return an arbitrary value. Meanwhile, in Haskell there is a specified behavior, which is crashing. Therefore, Hipster had to implement behavior to handle this difference. In Lean, you are unable to construct a partially defined function, without marking it as such. One has to match on all the cases that can occur from a data type. Further, if a function is marked as partial, it cannot be used in proofs. Thus, dealing with partial functions is not an aspect that needs to be handled.

One major aspect of Hipster that is not attempted is that of discarding lemmas that can be solved by routine reasoning. Pisa does not attempt to do this. Every part of the rewrite needs to be represented in the context, and this would therefore be a hindrance to the users of Pisa.

2.5 Hopster

Hopster is another theory exploration tool, made for HOL4 (Ricart, 2019). It is inspired by Hipster and therefore uses systems similar to those explained in section 2.4. However, similar to Lean, HOL4 does not have code generation to Haskell, and therefore a transpiler had to be created. Similarly to Hipster, Hopster does routine reasoning, but instead of throwing away trivial examples Hopster keeps them.

A difference that exists between Lean and HOL4 is the way that types are encoded. In HOL4, as Ricart mentions, there is a difference between types and terms. They exist on two different levels of the language, meanwhile, in Lean, such a distinction is harder to make. Lean uses dependent types which allows for values as types, as mentioned in section 2.1. This means that a more involved translation would be needed for Pisa.

Hopster uses TIP tools by Rosén and Smallbone, 2015, and therefore had to extend TIP tools to be able to preserve names. This was done by annotating function definition with comments labeling the HOL4 equivalent names. Pisa instead utilized a mapping between names when converting the naming scheme back, which was implemented in Pisa itself.

3

Technical Specification

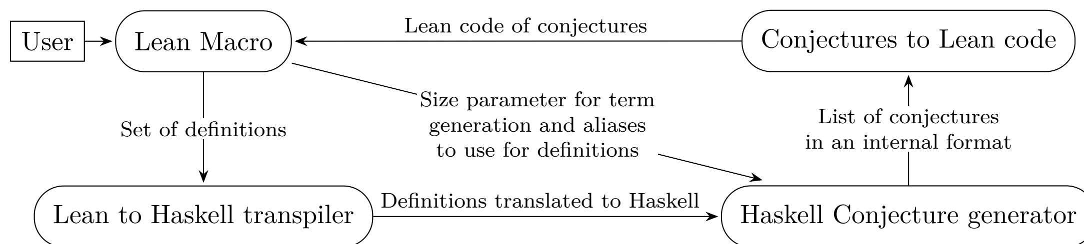


Figure 3.1: An overview of the information flow of Pisa.

Pisa is constructed by multiple distinct modules that will be explained in this chapter. An overview of how they interact can be seen in fig. 3.1.

A rundown of Pisa starts at the end user working with Lean as an ITP. They interact with Pisa by providing a set of definitions to explore through a macro within Lean. An optional size parameter can also be specified to control how large the generated terms may be when searching for conjectures. The given definitions, and all their transitive dependencies, are resolved and serialized by the exporter (Example can be seen in section 3.1.1). This information is then dispatched to an executable containing the remaining subsystems. Conjectures then replace the call site in the ITP upon request from the user by inserting the output of this executable.

To produce conjectures the executable uses QuickSpec and RoughSpec. This first requires that the serialized Lean definitions are transformed into Haskell code. This is presented in section 3.2.1. The generated Haskell code can then be run in an interpreter. This is utilized by QuickSpec and RoughSpec to generate a set of conjectures, that finally is transformed back into Lean syntax.

Since the differences between the languages are non-trivial multiple approaches could be taken to solve the problem. Several are presented and discussed in section 5.1.1.

3.1 Lean interface

Two modules of Pisa interface with Lean: a macro to interact with the complete system and an exporter to extract relevant definitions. The distinction of the modules is partly conceptual as there is a natural coupling between their implementations. Although the exporter does work as a standalone tool, it is tailored to the macro and other parts of Pisa. The macro depends on all parts of the system.

3.1.1 Exporter

Parsing arbitrary Lean syntax is non-trivial due to potential user defined parser extensions as explained in section 2.2.2. The most plausible and sound approach is therefore to utilize reflection within Lean. An existing solution, "ast_export" (Carneiro, 2024), generates a JSON representation of the abstract syntax tree for a set of definitions. However, determining the semantics of this syntax involves a similar problem due to extensibility. Another solution is "lean4export" (Ullrich et al., 2024) that exports fully elaborated terms (see section 2.2.3) compatible with the Lean 3 low-level format. Although *lean4export* is sufficient, some missing Lean 4 concepts and relatively slow performance make it inconvenient for the use in Pisa. A custom export tool was therefore created.

Given a list of identifiers, the custom export tool first resolves the minimum complete set of required definitions, that being the provided identifiers and their transitive dependencies. These definitions are then serialized as JSON documents, and together with a header of metadata, sent to the caller. The metadata informs of the root nodes and what identifiers are referred to them.

Transitive dependencies are fetched by traversing the declarations and the structure they hold. For example, if a user would be interested in conjectures regarding `or3` from listing 3.1, the translation would need the definitions of `B` and `or`. Through traversal of the internal representation, the dependencies are added as objects to also be exported, and then their definitions are also traversed, in the case of more nested definition.

```
inductive B where
  | t : B
  | f : B

def or : B → B → B
  | .t, _ => .t
  | .f, b => b

def or3 (a b c : B) : B := or (or a b) c
```

Listing 3.1: Lean code demonstrating transitive dependencies by defining a Boolean type as well as the disjunction operator for two and three values, `or` and `or3` respectively.

The exporter is available both as a self-contained executable, as functions within Lean. When used as an executable it can be called in the following manner:

```
pisa-lean Examples.Bool : Examples.Bool.B=B Examples.Bool.not=not
```

In this case `pisa-lean` is called with the namespace `Examples.Bool` and a list of functions within this namespace after the colon.[1] The output of `pisa-lean` will begin with a JSON list of the definition names used in the call to `pisa-lean`, called

[1]The `=` used in the example is to rename the functions to a more readable output.

roots. The reason that the names in the output are doubled in a pair formation is to allow for reverse translation explained in section 3.2.4. The output is followed by representations of inductives, constructors, and definitionals used in the roots. This can be seen in the example listing 3.2. The inductives, constructors, and definitionals are all encoded as separate JSON values.


```
[["Examples.Bool.B","B"], ["Examples.Bool.not","not"]]
{
  "conInfo": {
    "ctorInfo": {
      "val": {
        "type": {
          "const": {
            "us": [],
            "declName": "Examples.Bool.B"
          }
        },
        "numParams": 0,
        "numFields": 0,
        "name": "Examples.Bool.B.f",
        "levelParams": [],
        "isUnsafe": false,
        "induct": "Examples.Bool.B",
        "cidx": 1
      }
    }
  }
}
...truncated...
```


Listing 3.2: Example output from the exporting tool `pisa-lean`.

3.1.2 Macro

A macro was created utilizing the same concept as section 3.1.1, but for use within the language itself and to generate conjectures based on the output. It is called within a code editor with LSP-support. The calling is done via the macro `#pisa` followed by an optional number that is the size of terms for QuickSpec and RoughSpec, then the names of definitions. An example is `#pisa B not`. This will generate the same output as listing 3.2. However, the output will be piped to Pisa directly, which will generate the conjectures as explained in section 3.2. This output is then captured by the macro, which converts it into multiple `theorem` and applies the `sorry` keyword as a proof of each `theorem`. This then replaces the call site of the macro.

The `sorry` is a default implementation of something that is yet to be defined. `sorry` is of every type, and will give a warning during compilation. This allows the user to not have to do all the proofs at once, but can do the ones that feel relevant at the time.

3.2 Haskell interface

The Haskell interface will, with the output from the Lean interface (see section 3.1), generate conjectures. It goes through several steps to achieve this described below.

3.2.1 Translation

In order for the translation to hold a renaming of functions and data types need to occur due to Haskell being a more restrictive language. This means that the Haskell interface requires a mapping between the Haskell name and the Lean name. When a root is translated, they are added to a list of signatures with their Lean name instead of their Haskell counterpart. These signatures are then used in section 3.2.2. However, the names of types are not able to be retained in this way, and are instead handled by the method explained in section 3.2.4.

A key distinction between Lean and Haskell relates to dependent types. In Haskell types and values are two separate domains whereas types are values in Lean. To handle this aspect, a translation of data types and the type information was implemented together with an interpreter for the function bodies.

Translation of the data types is straightforward. Each data type has a set of constructors which are translated into their equivalent Haskell version. However, there are certain aspects that need to be handled in order to allow for QuickSpec and RoughSpec to be utilized. First is ensuring that the data types do not have type variables. Usually, one would utilize the data types `A` ... `E` to symbolize polymorphic aspects, but due to the need for an interpreter for the function bodies, a separate type known as `Poly` was introduced. `Poly` replaces each occurrence of polymorphism in the constructor and types of functions. The reason that the data types are kept at this level and not directly done for the intermediate representation is to be able to autogenerate them through QuickCheck, and ensuring invariants.

For the interpreter the intermediate representation from section 2.2.4 is used as the base language. There are particular aspects of this IR that one can ignore in the information sent by the exporter from section 3.1.1, since the exported code is not yet fully optimized.

The type signature in listing 3.3 is the one for the interpreter. `Decl` is the declaration used for evaluation, and the `Map` is the scope of other declarations that might be used in the body of the function. The list of `Val` is the arguments for the function. For this to work with QuickSpec, `[Val]` needs to contain values that conform to the expected shape of objects, that the function uses.

```
eval :: Decl -> Map Name Decl -> [Val] -> Val
```

Listing 3.3: Type signature of the interpreter function, called `eval`.

`Val` is the values in the IR language, and therefore the data type definitions previously mentioned need to be able to be converted. This is done by creating two functions, one from and one to `Val` for each type. Each input to the function can be

converted for use in the interpreter using `toVal`, and `fromVal` can be used after the computation. An example of how this is done can be seen from the Lean definition in listing 3.4 to the Haskell version in listing 3.5.

Another aspect that needs accommodation is how the IR expects values to be placed in constructors. One would expect that the values would be mapped in the same order as the types in the constructor. However, that is not the case. Instead the values of a type that is an enum, that is to say one that can be encoded as a unsigned integer, will appear before the other values that are not of an enum type. This means that a little fix in the from and to `Val` needed to be implemented to handle this.

```
inductive L (α : Type u) where
  | Nil : L α
  | Cons : α → L α → L α
```

Listing 3.4: Lists data type written in Lean.

```
newtype Poly = Poly Natural

fromValPoly :: Val -> Poly
fromValPoly (Unsigned a) = Poly a

toValPoly :: Poly -> Val
toValPoly (Poly a) = Unsigned a

data L where
  Nil :: L
  Cons :: Poly -> L -> L

toValL :: L -> Val
toValL Nil        = VCtor (Object 1 0) []
toValL (Cons b c) = VCtor (Object 1 1) [toValPoly b, toValL c]

fromValL :: Val -> L
fromValL (VCtor (Object _ 0) [])     = Nil
fromValL (VCtor (Object _ 1) [b, c] = Cons (fromValPoly b) (fromValL c)
```

Listing 3.5: listing 3.4 as converted into Haskell, using the `Poly` type. This includes the functions to transformation to and from values for the IR interpreter.

3.2.2 Interpretation of Haskell

With the translation to Haskell code, as outlined in section 3.2.1, one can utilize the library "hint" (The Hint Authors, 2025) to run the definitions and generate values. "hint" an interpreter for Haskell, built upon GHC's API. This allows Pisa to run the autogenerated code to fetch the signatures generated. The process above is similar to the tool "tip-spec" from "TIP: Tools for Inductive Provers" by Rosén and Smallbone, 2015.

3.2.3 Conjecture generation

QuickSpec is able to be run with a list of names and associated functions. In addition to the list, a max term size is given by the end user. Then, with the output of this as background, RoughSpec is called. It uses the general templates mentioned in section 2.3.1, to run through the same definitions. This will output a list of properties, in which the terms will use names from the translation. This is same approach that Einarsdóttir et al. did, in their hybrid approach, as explained in section 2.3.1.

One noteworthy aspect is that the functions the end user is interested in can be renamed in signatures generated for QuickSpec and RoughSpec. This allows Pisa to use the Lean naming, which is less strict than Haskell. The function names were converted to an acceptable Haskell version in the translation step, and the Lean counterpart used in the signature, as explained in section 3.2.1.

3.2.4 Reverse translation

Reverse translation is needed for the data types used to regain their Lean names instead of their Haskell counterparts. Further, reverse translations are needed when there is an implicit argument, such as in the case for `Nil` in the case of lists. The names of the functions that the theory is being explored on can be retained by other means, as mentioned in section 3.2.1.

To allow the user to easily use the generated conjectures, the names have to be reverted into the form they have in Lean. Throughout the process of translation, seen in section 3.2.1, a mapping between Haskell and Lean names are kept. This is, in this step, inverted. This allows a reversal of translation, meaning a translation from Haskell into their Lean counterpart. Further, this is then converted into theorems that are able to be copied into Lean code to start the proof aspect of these conjectures.

3.2.5 Example

As an example of what the different stages do, `N` and `add` will be used, assuming the definitions are exported and present.

Step 1: Translate the definitions, it should look similar to listing 3.6.
Step 2: Export the translation from step 1 to a Haskell file.
Step 3: Interpret the Haskell file to retrieve the signatures.
Step 4: Use QuickSpec and RoughSpec on the signatures to generate "laws" (conjectures), the conjectures generated in this example can be seen in listing 3.7.
Step 5: Reverse translate the "laws", creating the Lean version of these conjectures, as can be seen in listing 3.8.[2]

[2]It will actually be in a JSON format, that Lean then can import.

```
environment :: Map Name Decl
environment = fromList […truncated…]

data N where
  Z :: N
  S :: N -> N

toValN :: N -> Val
toValN Z = VCtor (Object 1 0) []
toValN (S b) = VCtor (Object 1 1) [toValN b]

fromValN :: Val -> N
fromValN (VCtor (Object _ 0) [ ]) = Z
fromValN (VCtor (Object _ 1) [b]) = S (fromValN b)

add = FDecl {f = "Examples.Nat.add", …truncated…}

signatures :: [Sig]
signatures =
  [ monoType (Proxy :: Proxy N)
  , con "add" (\a b -> fromValN (eval add environment
                                  [ toValN a, toValN b ]))
  ]
```

Listing 3.6: Example of auto-generated code based on `N` and `add`. It has been simplified be more readable.

```
== Laws ==
  1. add x y = add y x
  2. add (add x y) z = add x (add y z)
```

Listing 3.7: Output from running QuickSpec and RoughSpec on `signatures` from listing 3.6.

```
theorem conjecture0 (x y : N) :
  add x y = add y x := sorry
theorem conjecture1 (x y z : N) :
  add (add x y) z = add x (add y z) := sorry
```

Listing 3.8: Lean translation of listing 3.7.

3.3 Integration of subsystem

Pisa has been incorporated into a macro for use within Lean. A user utilizes `#pisa` within a namespace. They give an optional natural number which regulates the term size, and the identifiers they are interested in. This provides the user with a Language Server Protocol (LSP) code action. The code action will begin by fetching the definitions as explained in section 3.1. With these definitions, Pisa is called with a given size, and then the definitions are streamed to Pisa's process. After some time, the output of Pisa is parsed and then added to the file where the macro was called.

3.4 Partial and unsafe definitions

One difference between Lean and Haskell is how a function is defined. If a function doesn't properly define all the cases for a match in Lean it will not type check and therefore, not be able to be translated. In Haskell, this is a warning and will give an error at runtime. This means that by translating functions from Lean to Haskell, functions which do not match all the cases cannot occur. However, functions using the `sorry` keyword are still translated. The output of these functions utilize a lot of built-in definitions (handled by the runtime) that are not added by the exporter. These built-in definitions are not implemented by the interpreter. This means that these functions will potentially fail at runtime, which is to be expected. QuickSpec and RoughSpec will, because of this, fail to utilize this function if the execution is terminated with an error. In addition to what was mentioned above, data types and constructors can be marked as unsafe. They are handled by terminating the process, and mentioning that they contain unsafe values.

3.5 The supported subset of Lean

Pisa tries to support most of Lean. However, there are some features that are not able to be translated with the current implementation.

Enumerable and recursive data types are supported and functions utilizing them are able to be represented. Polymorphic data types are also representable, however, instantiating them with a type is not. This comes from the conversion to monomorphic representations, outlined in section 3.2.1.

Dependent types are not supported. The construction of values for these types hinders the implementation since the IR is able to handle certain aspects of dependent types, such as type level values in a constructor. But the types for functions would not be able to be converted as that would require dependent types in Haskell.

Type classes is another aspect that could be difficult to represent. Depending on what the type class encodes, it could be doable, but that would require knowledge of what the intention about said type class is aiming to do. To implement type classes, one would either have to translate the definitions of the class, or implement specific

versions of the common ones. Neither of these approaches were taken for this thesis. Further, a translation of the classes would also have to be supplied to the IR.

Not supporting type classes means that `IO` is not representable. Further, `IO` utilizes a lot of opaque definitions. Opaque definitions are definitions that are not exposed to the kernel (leanprover, 2025a). For `IO` specifically, these functions are usually implemented by the runtime.

Higher order functions are not able to be represented using the current implementation. By default, QuickCheck is able to generate functions for usage, which enables QuickSpec and RoughSpec to use higher order functions. With the current implementation of Pisa, this would require a layer between the generation of these functions, and their usage, in order to translate them into Lean IR.

4

Results

The result of this thesis can be compared to the aim outlined in section 1.1. With Pisa, a user is able to generate conjectures based on their own definitions, from a subset of Lean. It was feasible to do a translation from Lean to Haskell via a combination of the IR and Lambda calculus to encapsulate the different aspects of functions contra types. A combination of QuickSpec and RoughSpec were then able to generate conjectures that then were translated to Lean theorems, left to be proven by the user.

To show that Pisa is able to translate and conjecture about the goals, multiple data types were chosen that encapsulate the different aspects of goals (A) to (C). Booleans were chosen as the enumerable type, due to it being one of the fundamental types used by most programming languages. For recursive data types, natural numbers were chosen, encoded as Peano numbers. Natural numbers show a practical example in which recursion is needed to be able to use the basic functions one would expect. As for Polymorphic data types, lists and binary trees were chosen. They both use polymorphism, and are both recursive. Each data type and their associated functions were implemented. All the chosen data types and associated functions were able to be translated which means that the implemented Haskell interface is able to support these classes of types. Furthermore, the chosen data types and their associated functions are able to used for conjecture generation using the translation, thus fulfilling (D) "conjecture generation: Conjecture about the translation". Pisa was also able to be utilized from within the editor, using a macro, as outlined in 3.3 "integration of subsystem", fulfilling goal (E).

To show the relevance (see goal (F)) of the generated conjectures, the data types which has an equivalent in Mathlib (The mathlib Community, 2024) will be compared using precision and recall analysis (Powers, 2020). The formulas that will be used can be seen in equation set 1. Recall shows how well Pisa generates all of the potential true cases, while precision measures the degree of conjectures that can be found in mathlib based of the generated ones. Both of these values range between 0 and 1. Discussion of these results will be covered in section 5.1

$$TP = \text{No. lemmas both in generated conjectures and in mathlib}$$
$$FP = \text{No. lemmas in generated conjectures, but not in mathlib}$$
$$FN = \text{No. lemmas in mathlib, but not in generated conjectures}$$
$$Precision = \frac{TP}{TP + FP}$$
$$Recall = \frac{TP}{TP + FN}$$

Equation set 1: Precision and recall formulas.

4.1 Enumerated types

The boolean definition shows one of the base data types of which are enumerable. The definitions used, the generated code, and the generated conjectures in Lean can be found in appendix A.1. A noteworthy point is that the definitions were written with different types of syntax to show that a user can write in different ways without impacting the semantics. Using Pisa, the conjectures shown in conjecture set 1 are generated. In the resulting conjectures one can find conjectures regarding inverse function (eq. (4.11)), commutativity (eqs. (4.3) and (4.5)), associativity (eqs. (4.24) and (4.25)), De Morgan laws (eqs. (4.18) and (4.23)), among others. These conjectures portray a baseline for the domain, allowing more intricate proofs to use them.

To measure the relevance of the generated conjectures the following sources of mathlib were used: “SimpLemmas” (Leanprover Community, 2025j), “Bool.and” (Leanprover Community, 2025a), “Bool.or” (Leanprover Community, 2025c), and “Bool.distrubitivity” (Leanprover Community, 2025b). Some of the definitions had to be cut, due to utilizing functions not implemented, or equivalences which is not supported. The lemmas that is tested against can be seen in conjecture set 2. This results in a $TP = 22$, $FP = 3$, $FN = 13$, which means that $Recall = 0.6286$ and $Precision = 0.88$. For the exact list of categorization to generate the values, see appendix A.4.

$$\texttt{not false} = \texttt{true} \tag{4.1}$$
$$\texttt{not true} = \texttt{false} \tag{4.2}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{and}\ x\ y = \texttt{and}\ y\ x \tag{4.3}$$
$$\forall x : \mathbb{B}.\ \texttt{and}\ x\ x = x \tag{4.4}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{or}\ x\ y = \texttt{or}\ y\ x \tag{4.5}$$
$$\forall x : \mathbb{B}.\ \texttt{or}\ x\ x = x \tag{4.6}$$
$$\forall x : \mathbb{B}.\ \texttt{and}\ x\ \texttt{false} = \texttt{false} \tag{4.7}$$
$$\forall x : \mathbb{B}.\ \texttt{and}\ x\ \texttt{true} = x \tag{4.8}$$
$$\forall x : \mathbb{B}.\ \texttt{or}\ x\ \texttt{false} = x \tag{4.9}$$
$$\forall x : \mathbb{B}.\ \texttt{or}\ x\ \texttt{true} = \texttt{true} \tag{4.10}$$
$$\forall x : \mathbb{B}.\ \texttt{not}\ (\texttt{not}\ x) = x \tag{4.11}$$
$$\forall x : \mathbb{B}.\ \texttt{and}\ x\ (\texttt{not}\ x) = \texttt{false} \tag{4.12}$$
$$\forall x : \mathbb{B}.\ \texttt{or}\ x\ (\texttt{not}\ x) = \texttt{true} \tag{4.13}$$
$$\forall x\ y\ z : \mathbb{B}.\ \texttt{and}\ x\ (\texttt{and}\ y\ z) = \texttt{and}\ y\ (\texttt{and}\ x\ z) \tag{4.14}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{and}\ x\ (\texttt{or}\ x\ y) = x \tag{4.15}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{or}\ x\ (\texttt{and}\ x\ y) = x \tag{4.16}$$
$$\forall x\ y\ z : \mathbb{B}.\ \texttt{or}\ x\ (\texttt{or}\ y\ z) = \texttt{or}\ y\ (\texttt{or}\ x\ z) \tag{4.17}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{and}\ (\texttt{not}\ x)\ (\texttt{not}\ y) = \texttt{not}\ (\texttt{or}\ x\ y) \tag{4.18}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{and}\ (\texttt{not}\ x)\ (\texttt{or}\ x\ y) = \texttt{and}\ y\ (\texttt{not}\ x) \tag{4.19}$$
$$\forall x\ y\ z : \mathbb{B}.\ \texttt{and}\ (\texttt{or}\ x\ y)\ (\texttt{or}\ x\ z) = \texttt{or}\ x\ (\texttt{and}\ y\ z) \tag{4.20}$$
$$\forall x : \mathbb{B}.\ \texttt{and true}\ x = x \tag{4.21}$$
$$\forall x : \mathbb{B}.\ \texttt{or false}\ x = x \tag{4.22}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{or}\ (\texttt{not}\ x)\ (\texttt{not}\ y) = \texttt{not}\ (\texttt{and}\ x\ y) \tag{4.23}$$
$$\forall x\ y\ z : \mathbb{B}.\ \texttt{and}\ (\texttt{and}\ x\ y)\ z = \texttt{and}\ x\ (\texttt{and}\ y\ z) \tag{4.24}$$
$$\forall x\ y\ z : \mathbb{B}.\ \texttt{or}\ (\texttt{or}\ x\ y)\ z = \texttt{or}\ x\ (\texttt{or}\ y\ z) \tag{4.25}$$

Conjecture set 1: Generated by Pisa for the domain $\mathbb{B}$. The Lean versions of these conjectures can be seen in listing A.3.

$$(!\texttt{false}) = \texttt{true} \tag{4.26}$$
$$(!\texttt{true}) = \texttt{false} \tag{4.27}$$
$$\forall x\ y : \mathbb{B}.\ (x\ \&\&\ y) = (y\ \&\&\ x) \tag{4.28}$$
$$\forall b : \mathbb{B}.\ (b\ \&\&\ b) = b \tag{4.29}$$
$$\forall x\ y : \mathbb{B}.\ (x\ ||\ y) = (y\ ||\ x) \tag{4.30}$$
$$\forall b : \mathbb{B}.\ (b\ ||\ b) = b \tag{4.31}$$
$$\forall b : \mathbb{B}.\ (b\ \&\&\ \texttt{false}) = \texttt{false} \tag{4.32}$$
$$\forall b : \mathbb{B}.\ (b\ \&\&\ \texttt{true}) = b \tag{4.33}$$
$$\forall b : \mathbb{B}.\ (b\ ||\ \texttt{false}) = b \tag{4.34}$$
$$\forall b : \mathbb{B}.\ (b\ ||\ \texttt{true}) = \texttt{true} \tag{4.35}$$
$$\forall b : \mathbb{B}.\ (!!b) = b \tag{4.36}$$
$$\forall x : \mathbb{B}.\ (x\ \&\&\ !x) = \texttt{false} \tag{4.37}$$
$$\forall x : \mathbb{B}.\ (x\ ||\ !x) = \texttt{true} \tag{4.38}$$
$$\forall x\ y\ z : \mathbb{B}.\ (x\ \&\&\ (y\ \&\&\ z)) = (y\ \&\&\ (x\ \&\&\ z)) \tag{4.39}$$
$$\forall x\ y\ z : \mathbb{B}.\ (x\ ||\ (y\ ||\ z)) = (y\ ||\ (x\ ||\ z)) \tag{4.40}$$
$$\forall x\ y : \mathbb{B}.\ (!(x\ ||\ y)) = (!x\ \&\&\ !y) \tag{4.41}$$
$$\forall x\ y\ z : \mathbb{B}.\ (x\ ||\ y\ \&\&\ z) = ((x\ ||\ y)\ \&\&\ (x\ ||\ z)) \tag{4.42}$$
$$\forall b : \mathbb{B}.\ (\texttt{true}\ \&\&\ b) = b \tag{4.43}$$
$$\forall b : \mathbb{B}.\ (\texttt{false}\ ||\ b) = b \tag{4.44}$$
$$\forall x\ y : \mathbb{B}.\ (!(x\ \&\&\ y)) = (!x\ ||\ !y) \tag{4.45}$$
$$\forall a\ b\ c : \mathbb{B}.\ (a\ \&\&\ b\ \&\&\ c) = (a\ \&\&\ (b\ \&\&\ c)) \tag{4.46}$$
$$\forall a\ b\ c : \mathbb{B}.\ (a\ ||\ b\ ||\ c) = (a\ ||\ (b\ ||\ c)) \tag{4.47}$$
$$\forall b : \mathbb{B}.\ (\texttt{true}\ ||\ b) = \texttt{true} \tag{4.48}$$
$$\forall b : \mathbb{B}.\ (\texttt{false}\ \&\&\ b) = \texttt{false} \tag{4.49}$$
$$\forall a\ b : \mathbb{B}.\ (a\ \&\&\ (a\ \&\&\ b)) = (a\ \&\&\ b) \tag{4.50}$$
$$\forall a\ b : \mathbb{B}.\ ((a\ \&\&\ b)\ \&\&\ b) = (a\ \&\&\ b) \tag{4.51}$$
$$\forall x : \mathbb{B}.\ (!x\ \&\&\ x) = \texttt{false} \tag{4.52}$$
$$\forall x\ y\ z : \mathbb{B}.\ ((x\ \&\&\ y)\ \&\&\ z) = ((x\ \&\&\ z)\ \&\&\ y) \tag{4.53}$$
$$\forall a\ b : \mathbb{B}.\ (a\ ||\ (a\ ||\ b)) = (a\ ||\ b) \tag{4.54}$$
$$\forall a\ b : \mathbb{B}.\ ((a\ ||\ b)\ ||\ b) = (a\ ||\ b) \tag{4.55}$$
$$\forall x : \mathbb{B}.\ (!x\ ||\ x) = \texttt{true} \tag{4.56}$$
$$\forall x\ y\ z : \mathbb{B}.\ ((x\ ||\ y)\ ||\ z) = ((x\ ||\ z)\ ||\ y) \tag{4.57}$$
$$\forall x\ y\ z : \mathbb{B}.\ (x\ \&\&\ (y\ ||\ z)) = (x\ \&\&\ y\ ||\ x\ \&\&\ z) \tag{4.58}$$
$$\forall x\ y\ z : \mathbb{B}.\ ((x\ ||\ y)\ \&\&\ z) = (x\ \&\&\ z\ ||\ y\ \&\&\ z) \tag{4.59}$$
$$\forall x\ y\ z : \mathbb{B}.\ (x\ \&\&\ y\ ||\ z) = ((x\ ||\ z)\ \&\&\ (y\ ||\ z)) \tag{4.60}$$

Conjecture set 2: Mathlib equivalent for the domain $\mathbb{B}$.

4.2 Recursive types

For recursive data types, the Peano encoding of the natural numbers was chosen. The definitions used, the generated code, and the generated conjectures in lean can be found in appendix A.2. Using Pisa, the conjectures shown in conjecture set 3 are generated. In the resulting conjectures one can find conjectures regarding identity elements (eqs. (4.63), (4.66) and (4.73)), zero property (eq. (4.64)), commutativity (eqs. (4.61) and (4.62)), associativity (eqs. (4.74) and (4.75)), among others.

To measure the relevance of the generated conjectures the following sources of mathlib were used: “Nat.add theorems” (Leanprover Community, 2025g) and “Nat.mul theorems” (Leanprover Community, 2025h). Some of the definitions had to be cut, due to utilizing functions not implemented, or equivalences which is not supported. The lemmas that is tested against can be seen in conjecture set 4. This results in a $TP = 12$, $FP = 3$, $FN = 14$, which means that $Recall = 0.4615$ and $Precision = 0.8$. For the exact list of categorization to generate the values, see appendix A.5.

$$\forall x\ y : \mathbb{N}.\ \texttt{add}\ x\ y = \texttt{add}\ y\ x \tag{4.61}$$

$$\forall x\ y : \mathbb{N}.\ \texttt{mult}\ x\ y = \texttt{mult}\ y\ x \tag{4.62}$$

$$\forall x : \mathbb{N}.\ \texttt{add}\ x\ \texttt{zero} = x \tag{4.63}$$

$$\forall x : \mathbb{N}.\ \texttt{mult}\ x\ \texttt{zero} = \texttt{zero} \tag{4.64}$$

$$\forall x\ y : \mathbb{N}.\ \texttt{add}\ x\ (\texttt{succ}\ y) = \texttt{succ}\ (\texttt{add}\ x\ y) \tag{4.65}$$

$$\forall x : \mathbb{N}.\ \texttt{mult}\ x\ (\texttt{succ}\ \texttt{zero}) = x \tag{4.66}$$

$$\forall x\ y\ z : \mathbb{N}.\ \texttt{add}\ x\ (\texttt{add}\ y\ z) = \texttt{add}\ y\ (\texttt{add}\ x\ z) \tag{4.67}$$

$$\forall x\ y : \mathbb{N}.\ \texttt{add}\ x\ (\texttt{mult}\ x\ y) = \texttt{mult}\ x\ (\texttt{succ}\ y) \tag{4.68}$$

$$\forall x\ y : \mathbb{N}.\ \texttt{mult}\ x\ (\texttt{add}\ y\ y) = \texttt{mult}\ y\ (\texttt{add}\ x\ x) \tag{4.69}$$

$$\forall x\ y\ z : \mathbb{N}.\ \texttt{mult}\ x\ (\texttt{mult}\ y\ z) = \texttt{mult}\ y\ (\texttt{mult}\ x\ z) \tag{4.70}$$

$$\forall x\ y\ z : \mathbb{N}.\ \texttt{add}\ (\texttt{mult}\ x\ y)\ (\texttt{mult}\ x\ z) = \texttt{mult}\ x\ (\texttt{add}\ y\ z) \tag{4.71}$$

$$\forall x : \mathbb{N}.\ \texttt{succ}\ (\texttt{mult}\ x\ (\texttt{succ}\ (\texttt{succ}\ (\texttt{succ}\ x)))) = \texttt{add}\ x\ (\texttt{mult}\ (\texttt{succ}\ x)\ (\texttt{succ}\ x)) \tag{4.72}$$

$$\forall x : \mathbb{N}.\ \texttt{add}\ \texttt{zero}\ x = x \tag{4.73}$$

$$\forall x\ y\ z : \mathbb{N}.\ \texttt{add}\ (\texttt{add}\ x\ y)\ z = \texttt{add}\ x\ (\texttt{add}\ y\ z) \tag{4.74}$$

$$\forall x\ y\ z : \mathbb{N}.\ \texttt{mult}\ (\texttt{mult}\ x\ y)\ z = \texttt{mult}\ x\ (\texttt{mult}\ y\ z) \tag{4.75}$$

Conjecture set 3: Generated by Pisa for the domain $\mathbb{N}$. The Lean versions of these conjectures can be seen in listing A.6.

$$\forall n\ m : \mathbb{N}.\ n + m = m + n \tag{4.76}$$

$$\forall n\ m : \mathbb{N}.\ n * m = m * n \tag{4.77}$$

$$\forall n : \mathbb{N}.\ n * 0 = 0 \tag{4.78}$$

$$\forall n\ m : \mathbb{N}.\ n + \texttt{succ}\ m = \texttt{succ}\ (n + m) \tag{4.79}$$

$$\forall n : \mathbb{N}.\ n * 1 = n \tag{4.80}$$

$$\forall n\ m\ k : \mathbb{N}.\ n + (m + k) = m + (n + k) \tag{4.81}$$

$$(nm : Nat) : n * (m + 1) = n * m + n \tag{4.82}$$

$$\forall n\ m\ k : \mathbb{N}.\ n * (m * k) = m * (n * k) \tag{4.83}$$

$$\forall n\ m\ k : \mathbb{N}.\ n * (m + k) = n * m + n * k \tag{4.84}$$

$$\forall n : \mathbb{N}.\ 0 + n = n \tag{4.85}$$

$$\forall n\ m\ k : \mathbb{N}.\ (n + m) + k = n + (m + k) \tag{4.86}$$

$$\forall n\ m\ k : \mathbb{N}.\ (n * m) * k = n * (m * k) \tag{4.87}$$

$$\forall n\ m : \mathbb{N}.\ (\texttt{succ}\ n) + m = \texttt{succ}\ (n + m) \tag{4.88}$$

$$\forall n : \mathbb{N}.\ n + 1 = \texttt{succ}\ n \tag{4.89}$$

$$\forall n : \mathbb{N}.\ \texttt{succ}\ n = n + 1 \tag{4.90}$$

$$\forall n\ m\ k : \mathbb{N}.\ (n + m) + k = (n + k) + m \tag{4.91}$$

$$\forall n\ m : \mathbb{N}.\ n * \texttt{succ}\ m = n * m + n \tag{4.92}$$

$$\forall n : \mathbb{N}.\ 0 * n = 0 \tag{4.93}$$

$$\forall n\ m : \mathbb{N}.\ (\texttt{succ}\ n) * m = (n * m) + m \tag{4.94}$$

$$\forall n\ m : \mathbb{N}.\ (n + 1) * m = (n * m) + m \tag{4.95}$$

$$\forall n : \mathbb{N}.\ 1 * n = n \tag{4.96}$$

$$\forall n\ m\ k : \mathbb{N}.\ (n + m) * k = n * k + m * k \tag{4.97}$$

$$\forall n\ m\ k : \mathbb{N}.\ n * (m + k) = n * m + n * k \tag{4.98}$$

$$\forall n\ m\ k : \mathbb{N}.\ (n + m) * k = n * k + m * k \tag{4.99}$$

$$\forall n : \mathbb{n}.\ n * 2 = n + n \tag{4.100}$$

$$\forall n : \mathbb{n}.\ 2 * n = n + n \tag{4.101}$$

Conjecture set 4: Mathlib equivalent for the domain $\mathbb{N}$.

4.3 Polymorphic types

For polymorphic data types, lists and binary trees were defined. The definitions used, the generated code, and the generated conjectures in lean can be found in appendix A.3. Using Pisa, the conjectures shown in conjecture set 5 and 6 are generated. In the resulting conjectures one can find conjectures regarding inverse function (eqs. (4.105) and (4.116)), identity elements (eqs. (4.103) and (4.104)), associativity (eq. (4.107)), fixpoints (eqs. (4.102), (4.106), (4.111), (4.113) and (4.115)), and combinations of functions that are equal to another function (eqs. (4.112) and (4.114)).

To measure the relevance of the generated conjectures for `List`, since `Tree` doesn't have an equivalent implementation, the following sources of mathlib were used: "List.Lemmas.reverse" (Leanprover Community, 2025f), "List.append" (Leanprover Community, 2025d), and "List.reverse" (Leanprover Community, 2025e). Some of the definitions had to be cut, due to utilizing functions not implemented, or equivalences which is not supported. The lemmas that is tested against can be seen in conjecture set 4. This results in a $TP = 7$, $FP = 2$, $FN = 2$, which means that $\mathit{Recall} = 0.7778$ and $\mathit{Precision} = 0.7778$ for lists. For the exact list of categorization to generate the values, see appendix A.6.

As mentioned, `Tree` was not implemented in the same way as mathlib (Leanprover Community, 2025m). The implementation that can be found in mathlib has values in nodes instead of leaves, and the proofs associated are mostly about size and map functions.

$$\texttt{reverse Nil} = \texttt{Nil} \tag{4.102}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \texttt{List}\ \alpha.\ \texttt{append}\ x\ \texttt{Nil} = x \tag{4.103}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \texttt{List}\ \alpha.\ \texttt{append Nil}\ x = x \tag{4.104}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \texttt{List}\ \alpha.\ \texttt{reverse}\ (\texttt{reverse}\ x) = x \tag{4.105}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \alpha.\ \texttt{reverse}\ (\texttt{Cons}\ x\ \texttt{Nil}) = \texttt{Cons}\ x\ \texttt{Nil} \tag{4.106}$$

$$\begin{gathered}\forall \alpha : \texttt{Type}.\ \forall x\ y\ z : \texttt{List}\ \alpha. \\ \texttt{append}\ (\texttt{append}\ x\ y)\ z = \texttt{append}\ x\ (\texttt{append}\ y\ z)\end{gathered} \tag{4.107}$$

$$\begin{gathered}\forall \alpha : \texttt{Type}.\ \forall x : \alpha.\ \forall y\ z : \texttt{List}\ \alpha. \\ \texttt{Cons}\ x\ (\texttt{append}\ y\ z) = \texttt{append}\ (\texttt{Cons}\ x\ y)\ z\end{gathered} \tag{4.108}$$

$$\begin{gathered}\forall \alpha : \texttt{Type}.\ \forall x\ y : \texttt{List}\ \alpha. \\ \texttt{append}\ (\texttt{reverse}\ x)\ (\texttt{reverse}\ y) = \texttt{reverse}\ (\texttt{append}\ y\ x)\end{gathered} \tag{4.109}$$

$$\begin{gathered}\forall \alpha : \texttt{Type}.\ \forall x : \texttt{List}\ \alpha.\ \forall y\ z : \alpha. \\ \texttt{append}\ x\ (\texttt{Cons}\ y\ (\texttt{Cons}\ z\ \texttt{Nil})) = \\ \texttt{reverse}\ (\texttt{Cons}\ z\ (\texttt{Cons}\ y\ (\texttt{reverse}\ x)))\end{gathered} \tag{4.110}$$

Conjecture set 5: Generated by Pisa for the domain `List` α. The Lean versions of these conjectures can be seen in listing A.9.

$$\forall \alpha : \texttt{Type}.\ \forall x : \alpha.\ \texttt{leftmost}\ (\texttt{Leaf}\ x) = x \tag{4.111}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \texttt{Tree}\ \alpha.\ \texttt{leftmost}\ (\texttt{swap}\ x) = \texttt{rightmost}\ x \tag{4.112}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \alpha.\ \texttt{rightmost}\ (\texttt{Leaf}\ x) = x \tag{4.113}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \texttt{Tree}\ \alpha.\ \texttt{rightmost}\ (\texttt{swap}\ x) = \texttt{leftmost}\ x \tag{4.114}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \alpha.\ \texttt{swap}\ (\texttt{Leaf}\ x) = \texttt{Leaf}\ x \tag{4.115}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \texttt{Tree}\ \alpha.\ \texttt{swap}\ (\texttt{swap}\ x) = x \tag{4.116}$$

$$\forall \alpha : \texttt{Type}.\ \forall x\ y : \texttt{Tree}\ \alpha.\ \texttt{leftmost}\ (\texttt{Node}\ x\ y) = \texttt{leftmost}\ x \tag{4.117}$$

$$\forall \alpha : \texttt{Type}.\ \forall x\ y : \texttt{Tree}\ \alpha.\ \texttt{rightmost}\ (\texttt{Node}\ x\ y) = \texttt{rightmost}\ y \tag{4.118}$$

$$\forall \alpha : \texttt{Type}.\ \forall x\ y : \texttt{Tree}\ \alpha.\ \texttt{Node}\ (\texttt{swap}\ x)\ (\texttt{swap}\ y) = \texttt{swap}\ (\texttt{Node}\ y\ x) \tag{4.119}$$

Conjecture set 6: Generated by Pisa for the domain `Tree` α. The Lean versions of these conjectures can be seen in listing A.12.

$$\texttt{reverse []} = \texttt{[]} \tag{4.120}$$

$$\forall \alpha : \texttt{Type}.\ \forall as : \texttt{List}\ \alpha.\ as \texttt{ ++ []} = as \tag{4.121}$$

$$\forall \alpha : \texttt{Type}.\ \forall as : \texttt{List}\ \alpha.\ \texttt{[] ++ } as = as \tag{4.122}$$

$$\forall \alpha : \texttt{Type}.\ \forall as : \texttt{List}\ \alpha.\ as\texttt{.reverse.reverse} = as \tag{4.123}$$

$$\forall \alpha : \texttt{Type}.\ \forall as\ bs\ cs : \texttt{List}\ \alpha.\quad (as \texttt{ ++ } bs) \texttt{ ++ } cs = as \texttt{ ++ } (bs \texttt{ ++ } cs) \tag{4.124}$$

$$\forall \alpha : \texttt{Type}.\ \forall a : \alpha.\ \forall as\ bs : \texttt{List}\ \alpha.\quad (a :: as) \texttt{ ++ } bs = a :: (as \texttt{ ++ } bs) \tag{4.125}$$

$$\forall \alpha : \texttt{Type}.\ \forall as\ bs : \texttt{List}\ \alpha.\quad (as \texttt{ ++ } bs)\texttt{.reverse} = bs\texttt{.reverse ++ } as\texttt{.reverse} \tag{4.126}$$

$$\forall \alpha : \texttt{Type}.\ \forall as : \texttt{List}\ \alpha.\ \forall b : \alpha.\ \forall bs : \texttt{List}\ \alpha.\quad as \texttt{ ++ } b :: bs = as \texttt{ ++ [}b\texttt{] ++ } bs \tag{4.127}$$

$$\forall \alpha : \texttt{Type}.\ \forall a : \alpha.\ \forall as : \texttt{List}\ \alpha.\quad \texttt{reverse}\ (a :: as) = \texttt{reverse}\ as \texttt{ ++ [}a\texttt{]} \tag{4.128}$$

Conjecture set 7: Mathlib equivalent for the domain `List` α.

5

Conclusion

5.1 Discussion & Future work

Relevance was measured in chapter 4 using precision and recall analysis. The precision gathered shows that precision was around $80\% - 90\%$ for the different cases. This means that when Pisa generates conjectures, most of them will be a known lemma. When it comes to recall, it varied from $45\% - 80\%$. Pisa is therefore not able to find all the potential conjectures, using the default settings. Increasing the size term when call Pisa could help with this, but what is more likely the impacting factor is QuickSpec's pruning. For example, eq. (5.1) wasn't found by Pisa. However, it could be derived from eq. (5.2) and eq. (5.3). This means that the measurements taken here are a worst case scenario for this metric, but it still performed well in the polymorphic case. Further, some are defintionally equal, such as eqs. (5.4) and (5.5) due to $n + 1 = \texttt{succ}\ n$, but are split in mathlib.

$$\forall n\ m : \mathbb{N}.\ (\texttt{succ}\ n) + m = \texttt{succ}\ (n + m) \tag{5.1}$$

$$\forall n\ m : \mathbb{N}.\ n + \texttt{succ}\ m = \texttt{succ}\ (n + m) \tag{5.2}$$

$$\forall n\ m : \mathbb{N}.\ n + m = m + n \tag{5.3}$$

$$\forall n\ m : \mathbb{N}.\ (\texttt{succ}\ n) * m = (n * m) + m \tag{5.4}$$

$$\forall n\ m : \mathbb{N}.\ (n + 1) * m = (n * m) + m \tag{5.5}$$

Performance is an aspect that could have been evaluated due to the approach taken. The time for finding conjectures is an aspect of the work that influences the usability of the tool. We instead took the approach that it was more important to provide conjectures at some point, rather than to be fast. The reason for this is that utilizing an ITP will inherently have a slower type checking and compile time, therefore user are not as bothered by tooling taking a bit longer to execute.

Instantiations of polymorphic data types is a limitation with the current implementation. The implementation is not able to create an alias for these types, and therefore the ability to reason about these types are lost. This is a consequence of the monomorphic conversions to and from values and data types described in section 3.2.1. By instead constructing a type class for this conversion, Haskell could resolve these instantiations. However, `Poly` as a type would still be needed for the general cases when one wants to check polymorphic functions.

An auto proof engine as the final step was considered. However, this was not implemented for two main reasons, one being time constraints, the other being how the usage of auto proof engines such as “Aesop” (Limperg & From, 2023), or “Lean-auto” (Leanprover Community, 2024) work. The issue with how they work is that they usually require more user interaction than other auto proof engines for other languages. This can be in the user saying which lemmas are available, or how it can do case splits. This addition makes it harder to use with auto generated conjectures.

Lean’s syntax is extensible as mentioned in section 2.2.2. It is therefore hard to define where the base syntax ends and extended begins. Because of this, attempting to parse the source code is not applicable, without first resolving the extensions. An attempt at using “ast_export” by Carneiro, 2024 was done, but this approach was deemed too complicated because of the nature of the AST. Further, attempting to fetch transient dependencies based on the AST would be more difficult, if the extensions have not been resolved.

5.1.1 Alternative approaches

During the project, we saw multiple approaches we could have taken, and implemented some of them to a point.

The translation approach that was initially used for the project proved difficult to implement. It attempted to do a faithful translation from Lean to Haskell syntax. The issue that arose was the utilization of dependent types throughout the AST of the lambda calculus generated by the exporter. For example, when case splitting on a boolean, it will utilize the `recursor`, which uses dependent types. This made this approach unfeasible to implement, without hacking in edge cases. An idea that we thought of was to try to evaluate these statements when encountered, to find out what it would resolve to, since they should be fully satisfied. This was however not implemented.

Another approach that we tried was creating an interpreter for the AST of the lambda calculus itself. Like the pure translation, it hit snags as well. These snags were usually related to the `recursor`, which would be called in many different ways. Dependent types could theoretically be encoded using lambda expressions, since it is kept in the AST, however we believe it would have been difficult to auto generate arguments that keep these invariants with QuickCheck. This is the reason that the current implementation separates the values from the IR when generating them.

The approach that we did not attempt, but think is the most reasonable now is to reimplement QuickSpec or RoughSpec in Lean. With that, one could build up macros to utilize a tool in a similar way to Pisa. At the beginning of the project, we did not know of “Plausible” (Leanprover Community, 2025n), which fits the same part as QuickCheck does for QuickSpec and RoughSpec. Based on Plausible’s source code, it is heavily inspired by QuickCheck. This would avoid some issues, such as a translating the definitions into a different type system which doesn’t have the same capabilities as Lean. However, one thing that might happen is that certain issues may arise due to dependent types in the reimplementation, similarly to the

full translation approach. Therefore, we would recommend limiting the scope by utilizing a similar template system as RoughSpec, just to make something work.

5.2 Final remarks

Pisa is able to generate conjectures for Lean and translate a subset of Lean code into Haskell. The subset includes data types that are either enumerable, recursive or polymorphic. With a translation, Pisa is able to generate conjectures using QuickSpec and RoughSpec, which then can be ported back into Lean for the user to prove. Through precision and recall analysis, it was shown that Pisa has high precision ($80\% - 90\%$), but a wider range of recall ($45\% - 80\%$).

Pisa was integrated into a macro with an associated code action. This allows users to utilize Pisa within their editor while adding or changing definitions.

There are some limitations, mostly regarding more complex types. For example, instantiating polymorphic types (such as lists of natural numbers) is not supported. Further, dependent types and type classes are not constructable, which limits the potential of Pisa.

An extension to Pisa would be that of dependent types and type classes. To attempt this, our recommendation would be to reimplement the full functionality in Lean. We believe this has the best potential to be successful. Implementing Pisa fully in Lean would also allow for the usage of instantiation of Polymorphic data types.

Bibliography

Buchberger, B. (2000). Theory exploration with theorema. *Analele Universitatii Din Timisoara, Seria Matematica-Informatica*, *38*, 9–32. https://doi.org/10.1016/j.jal.2005.10.006

Buzzard, K. (2024). Mathematical reasoning and the computer. *Bulletin of the American Mathematical Society*, *61*(2), 211–224. https://doi.org/10.1090/bull/1833

Carneiro, M. (2024). *Ast_export* [Revision: 48092798045976347e413cedd4e3ecb9c76f1141]. Retrieved February 17, 2025, from https://github.com/digama0/ast_export

Claessen, K., & Hughes, J. (2000). Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, *35*(9), 268–279. https://doi.org/10.1145/357766.351266

de Moura, L., Kong, S., Avigad, J., van Doorn, F., & von Raumer, J. (2015). The lean theorem prover (system description). In A. P. Felty & A. Middeldorp (Eds.), *Automated deduction - cade-25* (pp. 378–388). Springer International Publishing. https://doi.org/10.1007/978-3-319-21401-6_26

de Moura, L., & Ullrich, S. (2021). The lean 4 theorem prover and programming language. In A. Platzer & G. Sutcliffe (Eds.), *Automated deduction – cade 28* (pp. 625–635). Springer International Publishing. https://doi.org/10.1007/978-3-030-79876-5_37

Einarsdóttir, S. H., Smallbone, N., & Johansson, M. (2021). Template-based theory exploration: Discovering properties of functional programs by testing. *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, 67–78. https://doi.org/10.1145/3462172.3462192

Geuvers, H. (2009). Proof assistants: History, ideas and future. *Sadhana*, *34*(1), 3–25. https://doi.org/10.1007/s12046-009-0001-5

Johansson, M., Rosén, D., Smallbone, N., & Claessen, K. (2014). Hipster: Integrating theory exploration in a proof assistant. In S. M. Watt, J. H. Davenport, A. P. Sexton, P. Sojka, & J. Urban (Eds.), *Intelligent computer mathematics* (pp. 108–122). Springer International Publishing. https://doi.org/10.1007/978-3-319-08434-3_9

leanprover. (2025a). *The lean language reference: Definitions*. Retrieved July 1, 2025, from https://lean-lang.org/doc/reference/4.19.0-rc2/Definitions/Definitions/

leanprover. (2025b). *The lean language reference: Elaborators*. Retrieved June 23, 2025, from https://lean-lang.org/doc/reference/4.19.0-rc2/Notations-and-Macros/Elaborators/

Leanprover Community. (2024). *Lean-auto*. Retrieved November 26, 2024, from https://reservoir.lean-lang.org/@leanprover-community/auto

Leanprover Community. (2025a). *Mathlib4 docs: Init.data.bool.and*. Retrieved August 20, 2025, from https://leanprover-community.github.io/mathlib4_docs/Init/Data/Bool.html#and

Leanprover Community. (2025b). *Mathlib4 docs: Init.data.bool.distributivity*. Retrieved August 20, 2025, from https://leanprover-community.github.io/mathlib4_docs/Init/Data/Bool.html#distributivity

Leanprover Community. (2025c). *Mathlib4 docs: Init.data.bool.or*. Retrieved August 20, 2025, from https://leanprover-community.github.io/mathlib4_docs/Init/Data/Bool.html#or

Leanprover Community. (2025d). *Mathlib4 docs: Init.data.list.basic append*. Retrieved August 20, 2025, from https://leanprover-community.github.io/mathlib4_docs/Init/Data/List/Basic.html#append

Leanprover Community. (2025e). *Mathlib4 docs: Init.data.list.basic reverse*. Retrieved August 20, 2025, from https://leanprover-community.github.io/mathlib4_docs/Init/Data/List/Basic.html#reverse

Leanprover Community. (2025f). *Mathlib4 docs: Init.data.list.lemmas reverse*. Retrieved August 20, 2025, from https://leanprover-community.github.io/mathlib4_docs/Init/Data/List/Lemmas.html#reverse

Leanprover Community. (2025g). *Mathlib4 docs: Init.data.nat.basic nat.add theorems*. Retrieved August 20, 2025, from https://leanprover-community.github.io/mathlib4_docs/Init/Data/Nat/Basic.html#Nat-add-theorems

Leanprover Community. (2025h). *Mathlib4 docs: Init.data.nat.basic nat.mul theorems*. Retrieved August 20, 2025, from https://leanprover-community.github.io/mathlib4_docs/Init/Data/Nat/Basic.html#Nat-mul-theorems

Leanprover Community. (2025i). *Mathlib4 docs: Init.prelude*. Retrieved June 30, 2025, from https://leanprover-community.github.io/mathlib4_docs/Init/Prelude.html#rfl

Leanprover Community. (2025j). *Mathlib4 docs: Init.simplemmas*. Retrieved August 20, 2025, from https://leanprover-community.github.io/mathlib4_docs/Init/SimpLemmas.html#Bool.or_false

Leanprover Community. (2025k). *Mathlib4 docs: Lean.compiler.ir.basic*. Retrieved June 23, 2025, from https://leanprover-community.github.io/mathlib4_docs/Lean/Compiler/IR/Basic.html

Leanprover Community. (2025l). *Mathlib4 docs: Lean.declaration*. Retrieved June 23, 2025, from https://leanprover-community.github.io/mathlib4_docs/Lean/Declaration.html

Leanprover Community. (2025m). *Mathlib4 docs: Mathlib.data.tree.basic binary tree*. Retrieved August 21, 2025, from https://leanprover-community.github.io/mathlib4_docs/Mathlib/Data/Tree/Basic.html#Binary-tree

Leanprover Community. (2025n). *Plausible*. Retrieved March 17, 2025, from https://reservoir.lean-lang.org/@leanprover-community/plausible

Limperg, J., & From, A. H. (2023). Aesop: White-box best-first proof search for lean. *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 253–266. https://doi.org/10.1145/3573105.3575671

Nordström, B., Petersson, K., & Smith, J. M. (1990). *Programming in martin-löf's type theory* (Vol. 200). Oxford University Press Oxford.

Pierce, B. C. (2002). In *Types and programming languages* (pp. 462–466). MIT press.

Powers, D. M. (2020). Evaluation: From precision, recall and f-measure to roc, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061*.

Ricart, J. (2019). *Hopster: Automated discovery of mathematical properties in hol* [Master's thesis, Chalmers tekniska högskola]. https://odr.chalmers.se/items/7dbaa0d9-97b7-468d-a16f-4e64507fb939

Rosén, D., & Smallbone, N. (2015). Tip: Tools for inductive provers. *Proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning - Volume 9450*, 219–232. https://doi.org/10.1007/978-3-662-48899-7_16

Smallbone, N., Johansson, M., Claessen, K., & Algehed, M. (2017). Quick specifications for the busy programmer. *Journal of Functional Programming*, *27*, e18. https://doi.org/10.1017/S0956796817000090

The Hint Authors. (2025). *Hint*. Retrieved March 5, 2025, from https://hackage.haskell.org/package/hint-0.9.0.8

The mathlib Community. (2024). *Mathlib4* [Revision: 1109970b9069835cb9d4cd69bf196c9fff7516f4]. Retrieved November 28, 2024, from https://github.com/leanprover-community/mathlib4

Ullrich, S., Carneiro, M., & Gadgil, S. (2024). *Lean4export* [Revision: d7978498941853b4ff8003b058f66e2142d3a763]. Retrieved February 10, 2025, from https://github.com/leanprover/lean4export

Ullrich, S., & de Moura, L. (2020). Beyond notations: Hygienic macro expansion for theorem proving languages. In N. Peltier & V. Sofronie-Stokkermans (Eds.), *Automated reasoning* (pp. 167–182). Springer International Publishing. https://doi.org/10.1007/978-3-030-51054-1_10

Ullrich, S., & de Moura, L. (2021). Counting immutable beans: Reference counting optimized for purely functional programming. *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. https://doi.org/10.1145/3412932.3412935

A

Generated code & conjectures

A.1 Enumerated types

```
inductive B where | t : B | f : B

def not (b : B) : B :=
  match b with | .t => .f | .f => .t

def and : B → B → B := λ a b =>
  match a with | .t => b | .f => .f

def or : B → B → B := λ
  | .t, _ => .t
  | .f, b => b

#pisa 7 B .t .f not and or
```

Listing A.1: A call to `#pisa` on definitions constructed using different approaches to demonstrate its ability to manage a variety code involving an enumerated type, that being booleans.

```
environment :: Map Name Decl
environment = fromList […truncated…]

data B where
  B_t :: B
  B_f :: B

toValB :: B -> Val
toValB B_t = Unsigned 0
toValB B_f = Unsigned 1

fromValB :: Val -> B
fromValB (Unsigned 0) -> B_t
fromValB (Unsigned 1) -> B_f

not = FDecl {f = "Example.not", …truncated…}

and = FDecl {f = "Example.and", …truncated…}

or = FDecl {f = "Example.or", …truncated…}

sigs :: [Sig]
sigs =
  [ monoType (Proxy :: Proxy Poly),
    monoType (Proxy :: Proxy B),
    con "B.t" B_t,
    con "B.f" B_f,
    con "not" (\a -> fromValB (eval not environment [toValB a])),
    con "and" (\a b -> fromValB
      (eval and environment [toValB a, toValB b])),
    con "or" (\a b -> fromValB
      (eval or environment [toValB a, toValB b]))
  ]
```

Listing A.2: The auto-generated translation of listing A.1.

```
theorem conjecture0             : not (.f : B) = (.t : B) := sorry
theorem conjecture1             : not (.t : B) = (.f : B) := sorry
theorem conjecture2 (x y : B)   : and x y = and y x := sorry
theorem conjecture3 (x : B)     : and x x = x := sorry
theorem conjecture4 (x y : B)   : or x y = or y x := sorry
theorem conjecture5 (x : B)     : or x x = x := sorry
theorem conjecture6 (x : B)     : and x (.f : B) = (.f : B) := sorry
theorem conjecture7 (x : B)     : and x (.t : B) = x := sorry
theorem conjecture8 (x : B)     : or x (.f : B) = x := sorry
theorem conjecture9 (x : B)     : or x (.t : B) = (.t : B) := sorry
theorem conjecture10 (x : B)    : not (not x) = x := sorry
theorem conjecture11 (x : B)    : and x (not x) = (.f : B) := sorry
theorem conjecture12 (x : B)    : or x (not x) = (.t : B) := sorry
theorem conjecture13 (x y z : B)
  : and x (and y z) = and y (and x z) := sorry
theorem conjecture14 (x y : B) : and x (or x y) = x := sorry
theorem conjecture15 (x y : B) : or x (and x y) = x := sorry
theorem conjecture16 (x y z : B)
  : or x (or y z) = or y (or x z) := sorry
theorem conjecture17 (x y : B)
  : and (not x) (not y) = not (or x y) := sorry
theorem conjecture18 (x y : B)
  : and (not x) (or x y) = and y (not x) := sorry
theorem conjecture19 (x y z : B)
  : and (or x y) (or x z) = or x (and y z) := sorry
theorem conjecture20 (x : B)    : and (.t : B) x = x := sorry
theorem conjecture21 (x : B)    : or (.f : B) x = x := sorry
theorem conjecture22 (x y : B)
  : or (not x) (not y) = not (and x y) := sorry
theorem conjecture23 (x y z : B)
  : and (and x y) z = and x (and y z) := sorry
theorem conjecture24 (x y z : B)
  : or (or x y) z = or x (or y z) := sorry
```

Listing A.3: Conjectures as generated by invoking the code action on `#pisa` in listing A.1.

A.2 Recursive types

```
inductive N where
  | Z : N
  | S : N → N

def add (n m : N) : N :=
  match m with
  | .Z => n
  | .S m' => .S (add n m')

def mult (n m : N) : N :=
  match m with
  | .Z => .Z
  | .S m' => add n (mult n m')

#pisa 7 N N.Z N.S add mult
```

Listing A.4: Demonstration that `#pisa` manage definitions for a recursive type, that being natural numbers.

```
environment :: Map Name Decl
environment = fromList […truncated…]

data N where
  N_Z :: N
  N_S :: N -> N

toValN :: N -> Val
toValN (N_Z)    = VCtor (Object 1 0) []
toValN (N_S b) = VCtor (Object 1 1) [toValN b]

fromValN :: Val -> N
fromValN (VCtor (Object _ 0) [ ]) = N_Z
fromValN (VCtor (Object _ 1) [b]) = N_S (fromValN b)

add = FDecl {f = "Example.add", …truncated…}

mult = FDecl {f = "Example.mult", …truncated…}

sigs :: [Sig]
sigs =
  [ monoType (Proxy :: Proxy Poly),
    monoType ((Proxy :: Proxy N)),
    con "N.Z" N_Z,
    con "N.S" N_S,
    con "add" (\a b -> fromValN
      (eval add environment [toValN a, toValN b])),
    con "mult" (\a b -> fromValN
      (eval mult environment [toValN a, toValN b]))
  ]
```

Listing A.5: The auto generated code based on listing A.4.

```
theorem conjecture0 (x y : N)    : add x y = add y x := sorry
theorem conjecture1 (x y : N)    : mult x y = mult y x := sorry
theorem conjecture2 (x : N)      : add x (.Z : N) = x := sorry
theorem conjecture3 (x : N)      : mult x (.Z : N) = (.Z : N) := sorry
theorem conjecture4 (x y : N)    : add x (.S y) = .S (add x y) := sorry
theorem conjecture5 (x : N)      : mult x (.S (.Z : N)) = x := sorry
theorem conjecture6 (x y z : N)
  : add x (add y z) = add y (add x z) := sorry
theorem conjecture7 (x y : N)
  : add x (mult x y) = mult x (.S y) := sorry
theorem conjecture8 (x y : N)
  : mult x (add y y) = mult y (add x x) := sorry
theorem conjecture9 (x y z : N)
  : mult x (mult y z) = mult y (mult x z) := sorry
theorem conjecture10 (x y z : N)
  : add (mult x y) (mult x z) = mult x (add y z) := sorry
theorem conjecture11 (x : N)
  : .S (mult x (.S (.S (.S x)))) = add x (mult (.S x) (.S x)) := sorry
theorem conjecture12 (x : N)
  : add (.Z : N) x = x := sorry
theorem conjecture13 (x y z : N)
  : add (add x y) z = add x (add y z) := sorry
theorem conjecture14 (x y z : N)
  : mult (mult x y) z = mult x (mult y z) := sorry
```

Listing A.6: Conjectures as generated by invoking the code action on `#pisa` in listing A.4.

A.3 Polymorphic types

```
inductive L : Type → Type where
  | Nil : L α
  | Cons : α → L α → L α

def append (xs : L α) (ys : L α) : (L α) :=
  match xs with
  | .Nil => ys
  | .Cons a as => .Cons a (append as ys)

def reverse (l : L α) : (L α) :=
  match l with
  | .Nil => .Nil
  | .Cons a as => append (reverse as) (.Cons a .Nil)

#pisa 7 L L.Nil L.Cons append reverse
```

Listing A.7: Demonstration that `#pisa` manage definitions for a polymorphic type, that being lists.

```
environment :: Map Name Decl
environment = fromList [...truncated...]

data L where
  L_Nil :: L
  L_Cons :: Poly -> L -> L

toValL :: L -> Val
toValL (L_Nil)      = VCtor (Object 1 0) []
toValL (L_Cons b c) = VCtor (Object 1 1) [toValPoly b, toValL c]

fromValL :: Val -> L
fromValL (VCtor (Object _ 0) [   ]) = L_Nil
fromValL (VCtor (Object _ 1) [b,c]) = L_Cons (fromValPoly b) (fromValL c)

append = FDecl {f = "Example.append", ...truncated...}

reverse = FDecl {f = "Example.reverse", ...truncated...}

sigs :: [Sig]
sigs =
  [ monoType (Proxy :: Proxy Poly),
    monoType ((Proxy :: Proxy L)),
    con "L.Nil" L_Nil,
    con "L.Cons" L_Cons,
    con "append" (\a b -> fromValL
      (eval append environment [toValL a, toValL b])),
    con "reverse" (\a -> fromValL (eval reverse environment [toValL a]))
  ]
```

Listing A.8: The auto-generated translation of listing A.7.

```
theorem conjecture0
  : reverse (.Nil : L α) = (.Nil : L α) := sorry
theorem conjecture1 (x : L α) : append x (.Nil : L α) = x := sorry
theorem conjecture2 (x : L α) : append (.Nil : L α) x = x := sorry
theorem conjecture3 (x : L α) : reverse (reverse x) = x := sorry
theorem conjecture4 (x : α)
  : reverse (.Cons x (.Nil : L α)) = .Cons x (.Nil : L α) := sorry
theorem conjecture5 (x y z : L α)
  : append (append x y) z = append x (append y z) := sorry
theorem conjecture6 (x  : α) (y z : L α)
  : .Cons x (append y z) = append (.Cons x y) z := sorry
theorem conjecture7 (x y : L α)
  : append (reverse x) (reverse y) = reverse (append y x) := sorry
theorem conjecture8 (x : L α) (y z : α)
  : append x (.Cons y (.Cons z (.Nil : L α)))
  = reverse (.Cons z (.Cons y (reverse x))) := sorry
```

Listing A.9: Conjectures as generated by invoking the code action on `#pisa` in listing A.7.

```
inductive T (α : Type u) where
  | Leaf : α → T α
  | Node : T α → T α → T α

def swap : T α → T α
  | .Leaf x => .Leaf x
  | .Node l r => .Node (swap r) (swap l)

def leftmost : T α → α
  | .Leaf x => x
  | .Node l _ => leftmost l

def rightmost : T α → α
  | .Leaf x => x
  | .Node _ r => rightmost r

#pisa 7 T T.Leaf T.Node swap leftmost rightmost
```

Listing A.10: Demonstration that `#pisa` manage definitions for a polymorphic type, that being binary trees.

```
environment :: Map Name Decl
environment = fromList […truncated…]

data T where
  T_Leaf :: Poly -> T
  T_Node :: T -> T -> T

toValT :: T -> Val
toValT (T_Leaf b) = VCtor (Object 1 0) [toValPoly b]
toValT (T_Node b c) = VCtor (Object 1 1) [toValT b, toValT c]

fromValT :: Val -> T
fromValT (VCtor (Object _ 0) [b  ]) = T_Leaf (fromValPoly b)
fromValT (VCtor (Object _ 1) [b,c]) = T_Node (fromValT b) (fromValT c)

swap = FDecl {f = "Example.swap", …truncated…}

leftmost = FDecl {f = "Example.leftmost", …truncated…}

rightmost = FDecl {f = "Example.rightmost", …truncated…}

sigs :: [Sig]
sigs =
  [ monoType (Proxy :: Proxy Poly),
    monoType ((Proxy :: Proxy T)),
    con "T.Leaf" T_Leaf,
    con "T.Node" T_Node,
    con "swap" (\a -> fromValT (eval swap environment [toValT a])),
    con "leftmost" (\a -> fromValPoly
      (eval leftmost environment [toValT a])),
    con "rightmost" (\a -> fromValPoly
      (eval rightmost environment [toValT a]))
  ]
```

Listing A.11: The auto-generated translation of listing A.10.

```
theorem conjecture0 (x : α)   : leftmost (.Leaf x) = x := sorry
theorem conjecture1 (x : T α) : leftmost (swap x) = rightmost x := sorry
theorem conjecture2 (x : α)   : rightmost (.Leaf x) = x := sorry
theorem conjecture3 (x : T α) : rightmost (swap x) = leftmost x := sorry
theorem conjecture4 (x : α)   : swap (.Leaf x) = .Leaf x := sorry
theorem conjecture5 (x : T α) : swap (swap x) = x := sorry
theorem conjecture6 (x y : T α)
  : leftmost (.Node x y) = leftmost x := sorry
theorem conjecture7 (x y : T α)
  : rightmost (.Node x y) = rightmost y := sorry
theorem conjecture8 (x y : T α)
  : .Node (swap x) (swap y) = swap (.Node y x) := sorry
```

Listing A.12: Conjectures as generated by invoking the code action on `#pisa` in listing A.10.

A.4 Precision and recall analysis for Enumerated types

True positives are the following (paired between generated and mathlib version):

- eqs. (A.1) and (A.26)
- eqs. (A.2) and (A.27)
- eqs. (A.3) and (A.28)
- eqs. (A.4) and (A.29)
- eqs. (A.5) and (A.30)
- eqs. (A.6) and (A.31)
- eqs. (A.7) and (A.32)
- eqs. (A.8) and (A.33)
- eqs. (A.9) and (A.34)
- eqs. (A.10) and (A.35)
- eqs. (A.11) and (A.36)
- eqs. (A.12) and (A.37)
- eqs. (A.13) and (A.38)
- eqs. (A.14) and (A.39)
- eqs. (A.17) and (A.40)
- eqs. (A.18) and (A.41)
- eqs. (A.20) and (A.42)
- eqs. (A.21) and (A.43)
- eqs. (A.22) and (A.44)
- eqs. (A.23) and (A.45)
- eqs. (A.24) and (A.46)
- eqs. (A.25) and (A.47)

False positives are the following (generated but not in mathlib):

- eq. (A.15)
- eq. (A.16)
- eq. (A.19)

False negative are the following (in mathlib but not generated):

- eq. (A.48)
- eq. (A.49)
- eq. (A.50)
- eq. (A.51)
- eq. (A.52)
- eq. (A.53)
- eq. (A.54)
- eq. (A.55)
- eq. (A.56)
- eq. (A.57)
- eq. (A.58)
- eq. (A.59)
- eq. (A.60)

$$\texttt{not false} = \texttt{true} \tag{A.1}$$
$$\texttt{not true} = \texttt{false} \tag{A.2}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{and}\ x\ y = \texttt{and}\ y\ x \tag{A.3}$$
$$\forall x : \mathbb{B}.\ \texttt{and}\ x\ x = x \tag{A.4}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{or}\ x\ y = \texttt{or}\ y\ x \tag{A.5}$$
$$\forall x : \mathbb{B}.\ \texttt{or}\ x\ x = x \tag{A.6}$$
$$\forall x : \mathbb{B}.\ \texttt{and}\ x\ \texttt{false} = \texttt{false} \tag{A.7}$$
$$\forall x : \mathbb{B}.\ \texttt{and}\ x\ \texttt{true} = x \tag{A.8}$$
$$\forall x : \mathbb{B}.\ \texttt{or}\ x\ \texttt{false} = x \tag{A.9}$$
$$\forall x : \mathbb{B}.\ \texttt{or}\ x\ \texttt{true} = \texttt{true} \tag{A.10}$$
$$\forall x : \mathbb{B}.\ \texttt{not}\ (\texttt{not}\ x) = x \tag{A.11}$$
$$\forall x : \mathbb{B}.\ \texttt{and}\ x\ (\texttt{not}\ x) = \texttt{false} \tag{A.12}$$
$$\forall x : \mathbb{B}.\ \texttt{or}\ x\ (\texttt{not}\ x) = \texttt{true} \tag{A.13}$$
$$\forall x\ y\ z : \mathbb{B}.\ \texttt{and}\ x\ (\texttt{and}\ y\ z) = \texttt{and}\ y\ (\texttt{and}\ x\ z) \tag{A.14}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{and}\ x\ (\texttt{or}\ x\ y) = x \tag{A.15}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{or}\ x\ (\texttt{and}\ x\ y) = x \tag{A.16}$$
$$\forall x\ y\ z : \mathbb{B}.\ \texttt{or}\ x\ (\texttt{or}\ y\ z) = \texttt{or}\ y\ (\texttt{or}\ x\ z) \tag{A.17}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{and}\ (\texttt{not}\ x)\ (\texttt{not}\ y) = \texttt{not}\ (\texttt{or}\ x\ y) \tag{A.18}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{and}\ (\texttt{not}\ x)\ (\texttt{or}\ x\ y) = \texttt{and}\ y\ (\texttt{not}\ x) \tag{A.19}$$
$$\forall x\ y\ z : \mathbb{B}.\ \texttt{and}\ (\texttt{or}\ x\ y)\ (\texttt{or}\ x\ z) = \texttt{or}\ x\ (\texttt{and}\ y\ z) \tag{A.20}$$
$$\forall x : \mathbb{B}.\ \texttt{and true}\ x = x \tag{A.21}$$
$$\forall x : \mathbb{B}.\ \texttt{or false}\ x = x \tag{A.22}$$
$$\forall x\ y : \mathbb{B}.\ \texttt{or}\ (\texttt{not}\ x)\ (\texttt{not}\ y) = \texttt{not}\ (\texttt{and}\ x\ y) \tag{A.23}$$
$$\forall x\ y\ z : \mathbb{B}.\ \texttt{and}\ (\texttt{and}\ x\ y)\ z = \texttt{and}\ x\ (\texttt{and}\ y\ z) \tag{A.24}$$
$$\forall x\ y\ z : \mathbb{B}.\ \texttt{or}\ (\texttt{or}\ x\ y)\ z = \texttt{or}\ x\ (\texttt{or}\ y\ z) \tag{A.25}$$

Conjecture set 8: Generated by Pisa for the domain $\mathbb{B}$ for comparison to mathlib.

$$(!\texttt{false}) = \texttt{true} \tag{A.26}$$
$$(!\texttt{true}) = \texttt{false} \tag{A.27}$$
$$\forall x\ y : \mathbb{B}.\ (x\ \&\&\ y) = (y\ \&\&\ x) \tag{A.28}$$
$$\forall b : \mathbb{B}.\ (b\ \&\&\ b) = b \tag{A.29}$$
$$\forall x\ y : \mathbb{B}.\ (x\ ||\ y) = (y\ ||\ x) \tag{A.30}$$
$$\forall b : \mathbb{B}.\ (b\ ||\ b) = b \tag{A.31}$$
$$\forall b : \mathbb{B}.\ (b\ \&\&\ \texttt{false}) = \texttt{false} \tag{A.32}$$
$$\forall b : \mathbb{B}.\ (b\ \&\&\ \texttt{true}) = b \tag{A.33}$$
$$\forall b : \mathbb{B}.\ (b\ ||\ \texttt{false}) = b \tag{A.34}$$
$$\forall b : \mathbb{B}.\ (b\ ||\ \texttt{true}) = \texttt{true} \tag{A.35}$$
$$\forall b : \mathbb{B}.\ (!!b) = b \tag{A.36}$$
$$\forall x : \mathbb{B}.\ (x\ \&\&\ !x) = \texttt{false} \tag{A.37}$$
$$\forall x : \mathbb{B}.\ (x\ ||\ !x) = \texttt{true} \tag{A.38}$$
$$\forall x\ y\ z : \mathbb{B}.\ (x\ \&\&\ (y\ \&\&\ z)) = (y\ \&\&\ (x\ \&\&\ z)) \tag{A.39}$$
$$\forall x\ y\ z : \mathbb{B}.\ (x\ ||\ (y\ ||\ z)) = (y\ ||\ (x\ ||\ z)) \tag{A.40}$$
$$\forall x\ y : \mathbb{B}.\ (!(x\ ||\ y)) = (!x\ \&\&\ !y) \tag{A.41}$$
$$\forall x\ y\ z : \mathbb{B}.\ (x\ ||\ y\ \&\&\ z) = ((x\ ||\ y)\ \&\&\ (x\ ||\ z)) \tag{A.42}$$
$$\forall b : \mathbb{B}.\ (\texttt{true}\ \&\&\ b) = b \tag{A.43}$$
$$\forall b : \mathbb{B}.\ (\texttt{false}\ ||\ b) = b \tag{A.44}$$
$$\forall x\ y : \mathbb{B}.\ (!(x\ \&\&\ y)) = (!x\ ||\ !y) \tag{A.45}$$
$$\forall a\ b\ c : \mathbb{B}.\ (a\ \&\&\ b\ \&\&\ c) = (a\ \&\&\ (b\ \&\&\ c)) \tag{A.46}$$
$$\forall a\ b\ c : \mathbb{B}.\ (a\ ||\ b\ ||\ c) = (a\ ||\ (b\ ||\ c)) \tag{A.47}$$
$$\forall b : \mathbb{B}.\ (\texttt{true}\ ||\ b) = \texttt{true} \tag{A.48}$$
$$\forall b : \mathbb{B}.\ (\texttt{false}\ \&\&\ b) = \texttt{false} \tag{A.49}$$
$$\forall a\ b : \mathbb{B}.\ (a\ \&\&\ (a\ \&\&\ b)) = (a\ \&\&\ b) \tag{A.50}$$
$$\forall a\ b : \mathbb{B}.\ ((a\ \&\&\ b)\ \&\&\ b) = (a\ \&\&\ b) \tag{A.51}$$
$$\forall x : \mathbb{B}.\ (!x\ \&\&\ x) = \texttt{false} \tag{A.52}$$
$$\forall x\ y\ z : \mathbb{B}.\ ((x\ \&\&\ y)\ \&\&\ z) = ((x\ \&\&\ z)\ \&\&\ y) \tag{A.53}$$
$$\forall a\ b : \mathbb{B}.\ (a\ ||\ (a\ ||\ b)) = (a\ ||\ b) \tag{A.54}$$
$$\forall a\ b : \mathbb{B}.\ ((a\ ||\ b)\ ||\ b) = (a\ ||\ b) \tag{A.55}$$
$$\forall x : \mathbb{B}.\ (!x\ ||\ x) = \texttt{true} \tag{A.56}$$
$$\forall x\ y\ z : \mathbb{B}.\ ((x\ ||\ y)\ ||\ z) = ((x\ ||\ z)\ ||\ y) \tag{A.57}$$
$$\forall x\ y\ z : \mathbb{B}.\ (x\ \&\&\ (y\ ||\ z)) = (x\ \&\&\ y\ ||\ x\ \&\&\ z) \tag{A.58}$$
$$\forall x\ y\ z : \mathbb{B}.\ ((x\ ||\ y)\ \&\&\ z) = (x\ \&\&\ z\ ||\ y\ \&\&\ z) \tag{A.59}$$
$$\forall x\ y\ z : \mathbb{B}.\ (x\ \&\&\ y\ ||\ z) = ((x\ ||\ z)\ \&\&\ (y\ ||\ z)) \tag{A.60}$$

Conjecture set 9: Mathlib equivalent for the domain $\mathbb{B}$ for comparison.

A.5 Precision and recall analysis for recursive types

True positives are the following (paired between generated and mathlib version):

- eqs. (A.61) and (A.76)
- eqs. (A.62) and (A.77)
- eqs. (A.64) and (A.78)
- eqs. (A.65) and (A.79)
- eqs. (A.66) and (A.80)
- eqs. (A.67) and (A.81)
- eqs. (A.68) and (A.82)
- eqs. (A.70) and (A.83)
- eqs. (A.71) and (A.84)
- eqs. (A.73) and (A.85)
- eqs. (A.74) and (A.86)
- eqs. (A.75) and (A.87)

False positives are the following (generated but not in mathlib):

- eq. (A.63)
- eq. (A.69)
- eq. (A.72)

False negative are the following (in mathlib but not generated):

- eq. (A.88)
- eq. (A.89)
- eq. (A.90)
- eq. (A.91)
- eq. (A.92)
- eq. (A.93)
- eq. (A.94)
- eq. (A.95)
- eq. (A.96)
- eq. (A.97)
- eq. (A.98)
- eq. (A.99)
- eq. (A.100)
- eq. (A.101)

$$\forall x\ y : \mathbb{N}.\ \texttt{add}\ x\ y = \texttt{add}\ y\ x \tag{A.61}$$

$$\forall x\ y : \mathbb{N}.\ \texttt{mult}\ x\ y = \texttt{mult}\ y\ x \tag{A.62}$$

$$\forall x : \mathbb{N}.\ \texttt{add}\ x\ \texttt{zero} = x \tag{A.63}$$

$$\forall x : \mathbb{N}.\ \texttt{mult}\ x\ \texttt{zero} = \texttt{zero} \tag{A.64}$$

$$\forall x\ y : \mathbb{N}.\ \texttt{add}\ x\ (\texttt{succ}\ y) = \texttt{succ}\ (\texttt{add}\ x\ y) \tag{A.65}$$

$$\forall x : \mathbb{N}.\ \texttt{mult}\ x\ (\texttt{succ}\ \texttt{zero}) = x \tag{A.66}$$

$$\forall x\ y\ z : \mathbb{N}.\ \texttt{add}\ x\ (\texttt{add}\ y\ z) = \texttt{add}\ y\ (\texttt{add}\ x\ z) \tag{A.67}$$

$$\forall x\ y : \mathbb{N}.\ \texttt{add}\ x\ (\texttt{mult}\ x\ y) = \texttt{mult}\ x\ (\texttt{succ}\ y) \tag{A.68}$$

$$\forall x\ y : \mathbb{N}.\ \texttt{mult}\ x\ (\texttt{add}\ y\ y) = \texttt{mult}\ y\ (\texttt{add}\ x\ x) \tag{A.69}$$

$$\forall x\ y\ z : \mathbb{N}.\ \texttt{mult}\ x\ (\texttt{mult}\ y\ z) = \texttt{mult}\ y\ (\texttt{mult}\ x\ z) \tag{A.70}$$

$$\forall x\ y\ z : \mathbb{N}.\ \texttt{add}\ (\texttt{mult}\ x\ y)\ (\texttt{mult}\ x\ z) = \texttt{mult}\ x\ (\texttt{add}\ y\ z) \tag{A.71}$$

$$\forall x : \mathbb{N}.$$
$$\texttt{succ}\ (\texttt{mult}\ x\ (\texttt{succ}\ (\texttt{succ}\ (\texttt{succ}\ x)))) = \texttt{add}\ x\ (\texttt{mult}\ (\texttt{succ}\ x)\ (\texttt{succ}\ x)) \tag{A.72}$$

$$\forall x : \mathbb{N}.\ \texttt{add}\ \texttt{zero}\ x = x \tag{A.73}$$

$$\forall x\ y\ z : \mathbb{N}.\ \texttt{add}\ (\texttt{add}\ x\ y)\ z = \texttt{add}\ x\ (\texttt{add}\ y\ z) \tag{A.74}$$

$$\forall x\ y\ z : \mathbb{N}.\ \texttt{mult}\ (\texttt{mult}\ x\ y)\ z = \texttt{mult}\ x\ (\texttt{mult}\ y\ z) \tag{A.75}$$

Conjecture set 10: Generated by Pisa for the domain ℕ for comparison to mathlib.

$$\forall n\ m : \mathbb{N}.\ n + m = m + n \tag{A.76}$$
$$\forall n\ m : \mathbb{N}.\ n * m = m * n \tag{A.77}$$
$$\forall n : \mathbb{N}.\ n * 0 = 0 \tag{A.78}$$
$$\forall n\ m : \mathbb{N}.\ n + \texttt{succ}\ m = \texttt{succ}\ (n + m) \tag{A.79}$$
$$\forall n : \mathbb{N}.\ n * 1 = n \tag{A.80}$$
$$\forall n\ m\ k : \mathbb{N}.\ n + (m + k) = m + (n + k) \tag{A.81}$$
$$(nm : Nat) : n * (m + 1) = n * m + n \tag{A.82}$$
$$\forall n\ m\ k : \mathbb{N}.\ n * (m * k) = m * (n * k) \tag{A.83}$$
$$\forall n\ m\ k : \mathbb{N}.\ n * (m + k) = n * m + n * k \tag{A.84}$$
$$\forall n : \mathbb{N}.\ 0 + n = n \tag{A.85}$$
$$\forall n\ m\ k : \mathbb{N}.\ (n + m) + k = n + (m + k) \tag{A.86}$$
$$\forall n\ m\ k : \mathbb{N}.\ (n * m) * k = n * (m * k) \tag{A.87}$$
$$\forall n\ m : \mathbb{N}.\ (\texttt{succ}\ n) + m = \texttt{succ}\ (n + m) \tag{A.88}$$
$$\forall n : \mathbb{N}.\ n + 1 = \texttt{succ}\ n \tag{A.89}$$
$$\forall n : \mathbb{N}.\ \texttt{succ}\ n = n + 1 \tag{A.90}$$
$$\forall n\ m\ k : \mathbb{N}.\ (n + m) + k = (n + k) + m \tag{A.91}$$
$$\forall n\ m : \mathbb{N}.\ n * \texttt{succ}\ m = n * m + n \tag{A.92}$$
$$\forall n : \mathbb{N}.\ 0 * n = 0 \tag{A.93}$$
$$\forall n\ m : \mathbb{N}.\ (\texttt{succ}\ n) * m = (n * m) + m \tag{A.94}$$
$$\forall n\ m : \mathbb{N}.\ (n + 1) * m = (n * m) + m \tag{A.95}$$
$$\forall n : \mathbb{N}.\ 1 * n = n \tag{A.96}$$
$$\forall n\ m\ k : \mathbb{N}.\ (n + m) * k = n * k + m * k \tag{A.97}$$
$$\forall n\ m\ k : \mathbb{N}.\ n * (m + k) = n * m + n * k \tag{A.98}$$
$$\forall n\ m\ k : \mathbb{N}.\ (n + m) * k = n * k + m * k \tag{A.99}$$
$$\forall n : \mathbb{n}.\ n * 2 = n + n \tag{A.100}$$
$$\forall n : \mathbb{n}.\ 2 * n = n + n \tag{A.101}$$

Conjecture set 11: Mathlib equivalent for the domain $\mathbb{N}$ for comparison.

A.6 Paired polymorphic conjectures

True positives are the following (paired between generated and mathlib version):

- eqs. (A.102) and (A.111)
- eqs. (A.103) and (A.112)
- eqs. (A.104) and (A.113)
- eqs. (A.105) and (A.114)
- eqs. (A.107) and (A.115)
- eqs. (A.108) and (A.116)
- eqs. (A.109) and (A.117)

False positives are the following (generated but not in mathlib):

- eq. (A.106)
- eq. (A.110)

False negative are the following (in mathlib but not generated):

- eq. (A.118)
- eq. (A.119)

$$\texttt{reverse Nil} = \texttt{Nil} \tag{A.102}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \texttt{List}\ \alpha.\ \texttt{append}\ x\ \texttt{Nil} = x \tag{A.103}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \texttt{List}\ \alpha.\ \texttt{append}\ \texttt{Nil}\ x = x \tag{A.104}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \texttt{List}\ \alpha.\ \texttt{reverse}\ (\texttt{reverse}\ x) = x \tag{A.105}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \alpha.\ \texttt{reverse}\ (\texttt{Cons}\ x\ \texttt{Nil}) = \texttt{Cons}\ x\ \texttt{Nil} \tag{A.106}$$

$$\forall \alpha : \texttt{Type}.\ \forall x\ y\ z : \texttt{List}\ \alpha.\ \texttt{append}\ (\texttt{append}\ x\ y)\ z = \texttt{append}\ x\ (\texttt{append}\ y\ z) \tag{A.107}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \alpha.\ \forall y\ z : \texttt{List}\ \alpha.\ \texttt{Cons}\ x\ (\texttt{append}\ y\ z) = \texttt{append}\ (\texttt{Cons}\ x\ y)\ z \tag{A.108}$$

$$\forall \alpha : \texttt{Type}.\ \forall x\ y : \texttt{List}\ \alpha.\ \texttt{append}\ (\texttt{reverse}\ x)\ (\texttt{reverse}\ y) = \texttt{reverse}\ (\texttt{append}\ y\ x) \tag{A.109}$$

$$\forall \alpha : \texttt{Type}.\ \forall x : \texttt{List}\ \alpha.\ \forall y\ z : \alpha.\ \texttt{append}\ x\ (\texttt{Cons}\ y\ (\texttt{Cons}\ z\ \texttt{Nil})) = \texttt{reverse}\ (\texttt{Cons}\ z\ (\texttt{Cons}\ y\ (\texttt{reverse}\ x))) \tag{A.110}$$

Conjecture set 12: Generated by Pisa for the domain `List` α for comparison to mathlib.

$$\texttt{reverse []} = \texttt{[]} \tag{A.111}$$

$$\forall \alpha : \texttt{Type}.\ \forall as : \texttt{List}\ \alpha.\ as \texttt{ ++ []} = as \tag{A.112}$$

$$\forall \alpha : \texttt{Type}.\ \forall as : \texttt{List}\ \alpha.\ \texttt{[] ++ } as = as \tag{A.113}$$

$$\forall \alpha : \texttt{Type}.\ \forall as : \texttt{List}\ \alpha.\ as.\texttt{reverse.reverse} = as \tag{A.114}$$

$$\forall \alpha : \texttt{Type}.\ \forall as\ bs\ cs : \texttt{List}\ \alpha.$$
$$(as \texttt{ ++ } bs) \texttt{ ++ } cs = as \texttt{ ++ } (bs \texttt{ ++ } cs) \tag{A.115}$$

$$\forall \alpha : \texttt{Type}.\ \forall a : \alpha.\ \forall as\ bs : \texttt{List}\ \alpha.$$
$$(a :: as) \texttt{ ++ } bs = a :: (as \texttt{ ++ } bs) \tag{A.116}$$

$$\forall \alpha : \texttt{Type}.\ \forall as\ bs : \texttt{List}\ \alpha.$$
$$(as \texttt{ ++ } bs).\texttt{reverse} = bs.\texttt{reverse ++ } as.\texttt{reverse} \tag{A.117}$$

$$\forall \alpha : \texttt{Type}.\ \forall as : \texttt{List}\ \alpha.\ \forall b : \alpha.\ \forall bs : \texttt{List}\ \alpha.$$
$$as \texttt{ ++ } b :: bs = as \texttt{ ++ [} b \texttt{] ++ } bs \tag{A.118}$$

$$\forall \alpha : \texttt{Type}.\ \forall a : \alpha.\ \forall as : \texttt{List}\ \alpha.$$
$$\texttt{reverse}\ (a :: as) = \texttt{reverse}\ as \texttt{ ++ [} a \texttt{]} \tag{A.119}$$

Conjecture set 13: Mathlib equivalent for the domain `List` α for comparison.