Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра інформатики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: **«Dependent Types for Formal Theorem Proving: A Case Study of Hall's Theorem»**

Виконав студент 4-го року навчання
спеціальності Комп'ютерні науки
Власенко Павло Борисович

Керівник: Жежерун. О. П.
кандидат наук, доцент

Рецензент:

_____

Кваліфікаційна робота захищена
з оцінкою _____

Секретар ЕК_____
(підпис)

«_____» _____ 20___р.

Київ - 2023

Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

доцент, кандидат наук.

_____*Гороховський С. С.*

(підпис)

“____”_____ 2023

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

для кваліфікаційної роботи

студенту 4-го курсу, факультету інформатики

Власенку Павлу Борисовичу

Тема: «Dependent Types for Formal Theorem Proving: A Case Study of Hall's Theorem»

Зміст кваліфікаційної роботи:

Abstract

1. Introduction
2. Foundations of type theory
3. Lean as an example of language with dependent types
4. Hall's theorem

Conclusion

Список літератури

Дата видачі “____”_____ 2023 Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Графік підготовки кваліфікаційної роботи до захисту

Графік узгоджено «_____» _____ 2023р.

№ з/п	Перелік робіт	Термін виконання етапу	Підпис наукового керівника	Дата ознайомлення наукового керівника	Примітка
1.	Отримання теми кваліфікаційної роботи.	20.12.2022			
2.	Ознайомлення з темою кваліфікаційної роботи.	10.03.2023			
3.	Розробка плану та структури роботи.	10.04.2023			
4.	Робота з науковою літературою, опис основних означень.	30.04.2023			
5.	Дослідження результатів отриманих в літературі.	05.05.2023			
6.	Робота над текстовим оформленням результатів.	10.05.2023			
7.	Попередній аналіз кваліфікаційної роботи. Виправлення помилок.	11.05.2023			
8.	Попередній захист кваліфікаційної роботи.	12.05.2023			
9.	Захист кваліфікаційної роботи.	29.05.2023			

Науковий керівник _____
(ПІБ)

Виконавець кваліфікаційної роботи _____
(ПІБ)

Contents

Chapter 1

Introduction

1.1 Abstract

This thesis investigates the role of dependent types and Curry-Howard isomorphism in formal theorem proving and programming. First, we highlight the connection between formal logic and type theory and demonstrate how dependent types allow us to encode complex properties like proofs or programs. Next, we introduce Lean, a language utilizing dependent types, and show practical examples to check that the program will never fail and be correct at compile time without needing tests. Finally, we will show an example of a more complex theorem defined in Lean – Hall's graph theorem and how to use its proof to write the verified program.

1.2 Introduction

Writing programs is not always a straightforward task. Mistakes can happen, and when they do, they can cause serious issues. To reduce the amount of bugs, unit tests are commonly used. However, they are not always perfect solution. For example, we can notice that strictly-typed languages such as C# or Java require far fewer tests compared to their dynamically-typed

counterparts like Python or Javascript. This characteristic is also noticeable in system programming languages: Rust, with its innovative borrow checker, can detect most memory errors at compile-time, preventing issues that could otherwise lead to memory leaks or vulnerabilities.

This is one of the reasons why the typed functional paradigm is rising in popularity. In the current landscape, nearly every language has an alternative to Haskell's *Maybe* and *Either* constructs, and numerous languages have adopted the concept of monads or effects. From this perspective, it becomes clear that the more expressive a type system is, the fewer potential errors can occur.

Dependent types take this expressiveness to another level. They can represent any term from higher-order logic as a type. With dependent types, it is possible to create types that represent a sorted list, an array of size n, or even articulate entire mathematical theorems.

In the first chapter, we will look into the theory underlying dependent types and type theory in general. Then, we will explore their functionality and discover how they allow us to express any logical proposition through the Curry-Howard isomorphism.

Next, we will look at Lean – a programming language and theorem prover that uses dependent types and can guarantee that a program will not fail and will be correct at compile time, obviating the need for most tests. This proposition stands in stark contrast to traditional programming practices, where unit testing has been a cornerstone of verification processes. However, the inherent limitations and imperfections of unit testing, including its inability to cover all possible runtime scenarios, have driven the exploration of more formal and rigorous program verification methods.

Much of this thesis is dedicated to demonstrating how Lean, with its inherent capability of supporting dependent types, can help circumvent the limitations of unit testing. First, we will implement a sorting algorithm

in Lean. This demonstration will focus on how the algorithm, despite not requiring to check array bounds at runtime, does not require tests and can be proven to never terminate incorrectly while always returning the correct result.

The concluding section of the research will present an example of a more complex theorem defined in Lean: Hall's graph theorem. Although this component plays a smaller role in our research than the focus on formal verification of programs, it is an important illustration of the potential of Lean and dependent types in advanced theorem proving.

This thesis, therefore, explores the role of dependent types and the Curry-Howard isomorphism in formal theorem proving and programming. While these concepts are undeniably complex, they offer a promising pathway toward a future where programs can be rigorously and formally verified to be error-free, presenting a paradigm shift in how we approach software reliability and correctness.

Chapter 2

Type theory

We all know about type systems from programming languages like Haskell or Java. However, this chapter will look at types from a more formal, mathematical perspective. Although this chapter does not aim to discuss any specific type of theory due to the huge variety of those, we provide a general overview of most of these theories' main principles and concepts.

2.1 History

The birth of type theory is often attributed to British philosopher and logician Bertrand Russell, who developed the concept as a means to avoid paradoxes in set theory. Russell's Paradox, a contradiction in naive set theory, is the most famous example of these paradoxes. The paradox happens when you consider the set of all sets that do not contain themselves, leading to a logical contradiction. To resolve this, Russell proposed a system called the "Theory of Types" in his work "Principia Mathematica," co-authored with Alfred North Whitehead, published between 1910 and 1913.

Russell's type theory introduced the concept of types, where every mathematical object has a type, and operations are restricted to specific types. This hierarchical organization of types prevented self-referential sets and thus

resolved the paradox.

In the mid-20th century, type theory saw further development under Alonzo Church, an American mathematician and logician. Church's simple theory of types, or simply type theory, was a formal logical language with a basis in the lambda calculus. The lambda calculus, another creation of Church, was a formal system that used function abstraction and application as the core principles of computation. This would later become fundamental in the development of programming languages and computer science.

The 1960s and 1970s saw the advent of more sophisticated type theories. Swedish logician Per Martin-Löf developed a system of constructive mathematics and logic known as Intuitionistic Type Theory, or Martin-Löf Type Theory (MLTT). This theory became a foundation for the design of proof assistants and functional programming languages, such as Lean.

Type theories are an area of active research, as demonstrated by homotopy type theory.

2.2 Basics

2.2.1 Terms, Types and Variables

In the realm of type theory, every distinct term is paired with a type. This pairing of a term and its type is generally represented in the format `term : type`. Natural numbers are a frequent type incorporated in type theory, typically denoted as $\mathbb{N}$ or simply `nat`. Boolean logic values also constitute another common type. Consequently, we can illustrate a few basic terms with their corresponding types as follows:

```
1 : nat
69 : nat
false : bool
```

Terms can be derived from other terms via the use of function calls. In the lexicon of type theory, the term for a function call is "function application". Function application entails the use of a term from a specified type, resulting in a term of another defined type. Simmilar to most functional languages, the notation for function application is function argument argument ..., as opposed to the conventional function(argument,argument, ...). In the context of natural numbers, it is feasible to delineate a function called add which accepts two natural numbers as inputs. Consequently, we can construct additional terms with their respective types as such:

```
add 0 0 : nat
add 1 2 : nat
add 1 (add 2 (add 0 1)) : nat
```

The last term includes parentheses to explicitly indicate the order of operations. Technically, a majority of type theories necessitate the presence of parentheses for each operation. However, in practicality, these parentheses are frequently omitted with the expectation that readers can leverage precedence and associativity to discern their implied locations. To facilitate ease of reading, a prevalent notation is to represent add x x y y as x + x + y + y. Hence, the previously mentioned terms could be reformulated as:

```
0 + 0 : nat
1 + 2 : nat
1 + (2 + (0 + 1)) : nat
```

Terms can also incorporate variables. These variables invariably possess a type. So, assuming x and y are variables of the type nat, the ensuing terms are also valid:

```
x : nat
x + 1 : nat
x + (x + y) : nat
```

There exist more types beyond nat and bool. We have previously observed the term add, which does not classify as a nat, but rather a function that computes to a nat when applied to two nat inputs. The type of add will be elaborated on subsequently. However, we first need to elucidate the concept of computation.

2.2.2 Computation

Type theory offers a built-in method for computation. Consider these distinct terms:

```
1 + 4 : nat
3 + 2 : nat
0 + 5 : nat
```

Despite their differences, they all compute to the term 5 : nat. In type theory, we use the terminology "reduction" or "reduce" to denote computation. For example, 0 + 5 : nat reduces to 5 : nat. This can be represented as 0 + 5 : nat $\twoheadrightarrow$ 5 : nat. The computation process is a mechanical one, accomplished through syntax rewriting.

Variable-inclusive terms can also be reduced. For instance, x + (1 + 4) : nat can be reduced to x + 5 : nat. We can reduce any part of a term, thanks to the Church-Rosser theorem.

A term that doesn't contain any variables and can't be further reduced is known as a "canonical term". All of the above terms reduce to 5 : nat, a canonical term. Canonical terms for natural numbers include:

```
0 : nat
1 : nat
2 : nat
etc.
```

It stands to reason that terms computing to the same term are equal. Given $x : \mathsf{nat}$, the terms $x + (1 + 4) : \mathsf{nat}$ and $x + (4 + 1) : \mathsf{nat}$ are equivalent as they both reduce to $x + 5 : \mathsf{nat}$. Equivalent terms can be substituted for one another. Equality is a complex notion in type theory, with numerous forms. The form of equality where two terms compute to the same term is referred to as "judgmental equality".

2.2.3 Functions

Functions are also terms. They are often defined by lambda terms, simmilar to lambda calculus.

A lambda term can be represented as $(\lambda\ \mathsf{variableName} : \mathsf{type1}\ .\ \mathsf{term})$ and carries the type $\mathsf{type1} \to \mathsf{type2}$. The type $\mathsf{type1} \to \mathsf{type2}$ suggests that the lambda term acts as a function that receives an argument of type $\mathsf{type1}$ and evaluates to a term of type $\mathsf{type2}$. The term nested within the lambda term should be a value of $\mathsf{type2}$, given the variable is of type $\mathsf{type1}$.

Consider the following example of a lambda term, a function that doubles its input:

$$(\lambda\ x : \mathsf{nat}\ .\ (\mathsf{add}\ x\ x)) : \mathsf{nat} \to \mathsf{nat}$$

Here, x is the variable name with a type nat. The term $(\mathsf{add}\ x\ x)$ is of type nat, presuming $x : \mathsf{nat}$. Therefore, the lambda term adopts the type $\mathsf{nat} \to \mathsf{nat}$, which implies that when a nat is used as an argument, it will evaluate to a nat. Reduction, alternatively known as computation, is applicable for lambda terms. When the function is invoked, the argument replaces the parameter.

We previously noted that function application is denoted by placing the parameter following the function term. So, if we intend to apply the preceding function with the parameter 5 of type nat, we notate:

$$(\lambda\ x : \mathsf{nat}\ .\ (\mathsf{add}\ x\ x))\ 5 : \mathsf{nat}$$

The lambda term was of type nat → nat, which meant that when a nat is used as an argument, it will yield a term of type nat. Since we have supplied it with the argument 5, the above term is of type nat. Reduction operates by substituting the argument 5 for the parameter x in the term (add x x), leading to the computation of:

```
(add 5 5) : nat
```

which naturally evaluates to

```
10 : nat
```

A lambda term is often referred to as an "anonymous function" as it lacks a name. For easier readability, a name is often assigned to a lambda term. This is purely a notational convention and does not alter the mathematical interpretation. Some scholars term it "notational equivalence". A name could be assigned to the above function using the notation:

```
double : nat → nat  ::= (λ x : nat . (add x x))
```

This is the same function as earlier, just notated differently. Therefore, the term

```
double 5 : nat
```

still evaluates to

```
10 : nat
```

2.2.4 Inductive types

Inductive types are a fundamental concept in type theory that allows us to define new types by specifying their constructors. An inductive type is defined by a collection of constructors, each of which specifies how to construct an element of the type.

They are often used in functional programming languages. For example, here is how defined list in Haskell:

```
data List a = Nil | Cons a (List a)
```

Or by using generalized algebraic data type syntax:

```
data List a where
  Nil :: List a
  Cons :: a -> List a -> List a
```

In type theory, an inductive type is typically defined by giving a set of introduction rules, which specify how to construct elements of the type, and a set of elimination rules, which specify how to use elements of the type. Set of introduction rules is defined as constructors in Haskell, and set of ellimination rules is defined by pattern matching. Types below can be defined by using ellimination rules.

2.2.4.1 Boolean

Boolean type is defined by two constructors

```
true : bool
false : bool
```

It's ellimination rule can be defined by function **if** such that:

```
if true b c ↠ b
if false b c ↠ c
```

2.2.4.2 Product type

It name comes from cartesian product and for two types A and B it produces type $A \times B$:

```
pair : A → B → A × B
```

It's ellimination rule can be defined by two functions:

```
first : A × B → A
second : A × B → B
```

2.2.4.3 Sum type

Simmilar to discriminated union, it has two constructors:

```
left : A → A + B
right : B → A + B
```

But only one ellimination rule:

```
ellimSum : A + B → (A → C) → (B → C) → C

f : A → C
g : B → C
ellimSum (left a) f g ↠ f a
ellimSum (right b) f g ↠ g b
```

2.2.4.4 Unit type

The unit type has only one simple constructor and consists of only one value. It's often written as top or 1 and it's single instance as $*$ or ()

The unit type is utilized to indicate the existence or computability of something. If we can create a function of type $\mathsf{top} \to A$ for a given type A,

we can conclude that A has one or more terms. When a type has at least one term, we say it is "inhabited".

2.2.4.5 Empty type

Writen as $\bot$ or 0, the empty type does not contain any terms. We cannot create it, so if there a function of type $\mathsf{A} \to \bot$, we can conclude that A is also an empty type or in other words, "uninhabited". By analogy to unit type, it shows that something does not exist or uncomputable.

Later we will discuss the usage of empty and unit types.

2.3 Dependent types

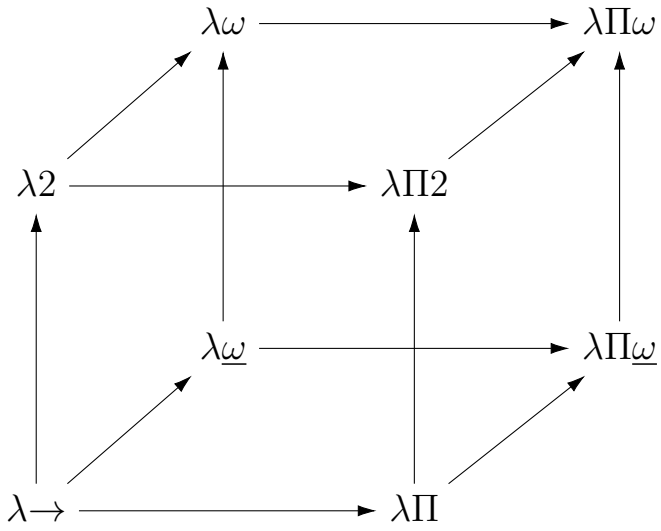
Dependent types are a concept in computer science and logic that dramatically enhances the power and expressiveness of type theory. In essence, dependent types allow types to be dependent on values, hence the term "dependent type." This seemingly simple idea can lead to remarkable consequences, enabling us to express sophisticated properties and invariants in our programs, making them safer, more reliable, and easier to reason about.

2.3.1 Lambda cube

To fully understand the significance and potential of dependent types, it is important to explore a fundamental theoretical framework that provides the foundation for studying dependent types - the Lambda Cube. Introduced by Henk Barendregt, the Lambda Cube is a conceptual model representing different forms of type systems in lambda calculus, a formal system in mathematical logic and computer science.

Lambda calculus forms the backbone of functional programming, a paradigm where functions are first-class entities, and computation is viewed as the eval-

uation of mathematical functions. The Lambda Cube extends lambda calculus in various dimensions, including types, polymorphism, and dependent types. It consists of eight points (or vertices), each representing a different kind of typed lambda calculus:



2.3.1.1 $\lambda \rightarrow$ Simply typed lambda calculus

This is the simplest system in the λ-cube. The only possible abstractions here are **terms which are depend on other terms** which we call functions. For example `f : int → bool` is a term which depending on provided term of type int, will return a corresponding term of type bool.

2.3.1.2 $\lambda 2$ System F

System F give us ability to create **terms which depend on types**, or in other words, it gives us polymorphism. Consider generic function `id<a> : a → a` which takes as argument any type, term of such type and return this term back.

2.3.1.3 $\lambda\underline{\omega}$ System $F\underline{\omega}$

System $F\underline{\omega}$ allows to create **types which depend on other types**. Sum and product types are perfect examples of such abstraction: type `pair`

<a,b> : a → b → pair<a,b> depends on two other types a and b.

2.3.1.4 $\lambda\omega$ System $F\omega$

Most modern programming languages use $\lambda\omega$ System, as they all support generics and polymorphism.

2.3.1.5 $\lambda\Pi$ LambdaΠ

Finally, this system introduces us to the concept of dependent types. Dependent types are types whose definition hinges on other terms, adding versatility to programming.

Consider a concrete example: a function, represented as f : *int* → Type . The function f is defined to take an integer and return a type. In other words, the resulting type that this function produces is dependent on the integer input it receives.

To further illustrate, we can introduce another function, g : (x : *int*) → f x. Here, function g takes an integer 'x' and applies function 'f' to 'x'. This demonstrates how the output type of 'f' can be used as the input to another function, further enhancing the system's adaptability.

The introduction of dependent types into our system gives us a lot powerful capabilities, the usefulness of which will be explored in following chapters. Not only do they allow for more flexible and expressive typing, but they also pave the way for more rigorous program correctness checks, making this system a truly powerful tool in the world of programming.

2.3.1.6 $\lambda\Pi\omega$ Calculus of constructions

The calculus of constructions is the most powerful system which combines features from all other types of systems. This is the system used in most languages with dependent types (as there is no reason to exclude simple

polymorphism if you have already implemented complex dependent types) like Coq and Lean.

In such a system, every term, including types themselves, has a type. More about this in the next chapter.

2.3.2 Universes

In the calculus of construction, every element has a type, including types themselves. For instance, consider simple types like *int* or bool. We can express that they are types by writing *int* : Type and bool : Type. Similarly, the function f : *int* → Type from the previous example returns a term of type Type. However, one might wonder about the type of Type itself. Even Type has a type, which can be specified as follows:

```
Type : Type 1
```

More over, "Type 1" also has it's type "Type 2" and so on:

```
Type : Type 1
Type 1 : Type 2
Type 2 : Type 3
Type 3 : Type 4
...
```

Types are usually called universes and written as U_0, U_1, U_2, etc.

2.3.3 Dependent types

We already showed what is a dependent type in λΠ LambdaΠ type system. Usually, there are two ways of constructing dependent types:

2.3.3.1 Product type

Let A : U be a type from universe U. Consider a dependent type B : A $\rightarrow$ U which for every term a of type A assigns a type B(a). Then, a type of B : A $\rightarrow$ U is called a **dependent product type or** $\Pi - type$ and usually it's written as $\Pi_{x:A}B(x)$.

It's called a product type because simmilar to functions from set theory, it's instances could be represented as tuples - product types. For example, for $B(a) = X_a$ type $\Pi_{x:\mathbb{N}}B(x)$ devolves into product $X_0 \times X_1 \times X_2 \times X_3...$

Note that if B is constant function to some specific type C, then $\Pi_{x:A}B(x)$ is equal to A $\rightarrow$ C

2.3.3.2 Sum type

Now, for same A : U and B : A $\rightarrow$ U consider a term (a, b) where a : A and b : B(a). We can define type of this term as $\sum_{x:A} B(x)$. We can also represent it as $X_0 + X_1 + X_2 + X_3...$ in case if $A = \mathbb{N}$

Note that if B is constant function to some specific type C, then $\sum_{x:A} B(x)$ is equal to $A \times C$

2.4 The Curry–Howard isomorphism

Now, look at an important concept: the Curry–Howard isomorphism. This special relationship between intuitionistic logic and type theory makes it possible to write proofs as programs. Before we can understand it, though, we need to know what intuitionistic logic is:

2.4.1 Intuitionistic logic

Intuitionistic logic, also known as constructive logic, is a type of mathematical logic that deviates from classical logic primarily in its interpretation

of the concept of truth. It was developed as an attempt to reflect the process of mathematical construction more accurately than classical logic.

2.4.1.1 True and false

While classical logic holds that any statement is either true or false - a principle is known as the law of excluded middle (LEM), intuitionistic logic does not accept this. Instead, it asserts that the truth of a statement is equivalent to our ability to prove it. In other words, in intuitionistic logic, a statement is considered true if and only if we can construct a proof for it and false if and only if we can construct a proof of its negation.

For example, let us look at the proposition "Either it will rain tomorrow, or it will not rain tomorrow." In classical logic, it is accepted as true because of LEM. However, in intuitionistic logic, we cannot prove such a statement, as we need to have information about whether it will rain tomorrow or not.

Another key principle that differs between the two is the principle of double negation. In classical logic, a statement is equivalent to the double negation of that statement (i.e., $\neg\neg P = P$). However, this is not the case in intuitionistic logic because a proof of $\neg\neg P$ only indicates that we cannot prove that P is false, not that we can prove P is true.

2.4.2 Propositional logic

The Curry–Howard isomorphism is a deep and foundational link between two major areas of mathematics: intuitionistic logic and type theory. This remarkable isomorphism reveals a correspondence that draws parallels between the constructs of intuitionistic logic and the constructs of type theory.

Here is a table summarizing this profound correspondence:

Let's take closer look at eache element of this table. Instead of general lambda calculus syntax, we'll use pseudocode simmilar to Haskell.

Logic side	Type theory side
True formula	Unit type or $\top$ type
False formula	Empty type or $\bot$ type
Implication	Function type
Conjunction	Product type
Disjunction	Sum type
Universal quantification	Π type
Existential quantification	Σ type

2.4.2.1 True and false types

Due to constructive nature of intuitionistic logic, to provide a proof of some proposition, we need to provide a construction of this proposition. So we can define true-formula as $\top$ type with single constructor:

```
data True = ()
```

But we cannot prove false formula, as it is false. So, false formula would have a type without any terms - $\bot$ type. Let's write it as

```
data False
```

2.4.2.2 Implication

Implication can be written as function. To prove statement $p \to q \to p$, we need just to write a function which takes p, q and returns p:

```
theoremImpl : p -> q -> p
theoremImpl p _ = p
```

Note that $p \to q$ is not a tautology. So, it's impossible to write a function of type `p -> q` and thus, prove a statement $p \to q$.

By having implication, we can define negation as $\neg p = p \to false$. Let's prove a statement $(p \to q) \to \neg q \to \neg p$:

```
type Not = p -> False
t : (p -> q) -> Not q -> Not p
t : (p -> q) -> (q -> False) -> (p -> False)
t pToQ notQ = fun (hp : p) -> notQ (pToQ hp)
```

2.4.2.3 Conjunction and disjunction

To prove that $a \wedge b$ we need to have proofs for both a and b. That's why we can define conjunction as product type:

```
data And a b = (a, b)

-- proof that a ∧ b → a
t : And a b -> a
t (a, _) = a
```

Simmilary, to prove that $a \vee b$ we need to have a proof for either a or b. So it corresponds to sum type:

```
data Or a b = Left a | Right b

-- proof that a ∨ b → (a → c) → (b → c) → c
t : Or a b -> (a -> c) -> (b -> c) -> c
t (Left a) atoc btoc = atoc a
t (Right b) atoc btoc = btoc b
```

2.4.2.4 Logical equality

By having an implication and conjunction, we can define equality. With this, we can prove more complex things like distributivity:

```
type Eq a b = And (a -> b) (b -> a)

-- p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r)
theorem distr : Eq (And p (Or q r)) (Or (And p q) (
  And p r))
distr = (a, b) where
      a (p, (Left q)) = Left (p, q)
      a (p, (Right r)) = Right (p, r)
      b (Left (p, q)) = (p, Left q)
      b (Right (p, r)) = (p, Right r)
```

2.4.2.5 Axioms

Axioms are statemnts, which are accepted without any proofs. We can denote them as term of some type, without constructing this term. For example, let's introduce law of excluded middle:

```
axiom exclM : Or a (Not a)

theorem doubleNegation : Not (Not p) -> p
doubleNegation notNotP = match exclM with
  | Left p => p
  | Right notP => absurd <| notNotP notP
```

For proving double nageation we used function absurd. It's type is absurd : false → a and it shows principle of explosion - everything comes from

false. We cannot define it with regular Haskell syntax but it is important part of type theory and it is in Lean's standart library.

2.4.3 First and higher order logic

2.4.3.1 Universal quantifier

In previous discussions, we were dealing only with simple propositional logic. However, to use higher-order logic, it becomes necessary to introduce the concept of predicates. For example, suppose we have a predicate P, which requires an argument of a particular type a. How might we represent this in code? The immediate solution might be writing it as $\mathsf{P : a \to b}$, where b is a specific type. However, this approach is insufficient because the result of a predicate may be of varying types - it could be either $\top$ or $\bot$.

An alternative consideration might be to use the form $P : a \to \top \vee \bot$. This approach, however, is also flawed. It is impossible to represent the application of such a predicate at the type level. This is where the utility of dependent types comes into play.

If we represent this predicate as $\mathsf{P : a \to Type}$, we are then able to write type propositions in the form of $\mathsf{example : a \to b \to And\ (P\ a)\ (P\ b) \to P\ a}$. This is equivalent to the logical proposition $\forall a \forall b, P(a) \wedge P(b) \to P(a)$. It's also important to note that $\mathsf{P : a \to Type}$ can be rewritten as $\Pi x : a P(x)$, which equals $\forall x \in a, P(x)$. So the previously mentioned example could be formally expressed as having a type $example : \Pi_{x:a}\Pi_{y:b}P(x) \times P(y) \to P(x)$. This means we can utilize Π types as universal quantifiers. Through this level of expressiveness, we can also limit the scope of the quantifier to something like $\Pi_{x:a}\Pi_{y:b}P(x) \times P(y) \to \Pi_{z:a}P(z)$.

2.4.3.2 Existential quantifier

Constructive logic requires us to provide a specific term when proving the existence of something. Therefore, to express the proposition $\exists a, P(a)$, we simply need to return such an a. This can be accomplished using the $\sum$ types described above.

Consider the following example:

```
-- equals to ∃xP(x) ∧ Q(x) → ∃z, Q(z) ∧ P(z)
```
$$\texttt{example : } \sum_{x:a} P(x) \times Q(x) \to \sum_{z:a} Q(z) \times P(z)$$

This demonstrates the expressive power of $\sum$ types in representing existential quantifiers.

2.4.3.3 Higher order logic

Higher order logic leverages the concept of dependent types. The interesting aspect about dependent types is that they can create other dependent types. Consider the following code:

```
-- equals to ∃P∀xP(x) → ∃P∃yP(y)
```
$$\texttt{example : } \sum_{P:a\to Type} \Pi_{x:a} P(x) \to \Pi_{P:a\to Type} \Pi_{y:a} P(y)$$

In this example, you can observe how we've used a $\sum$ type to create a new predicate that in turn takes an argument of Π type. This serves to illustrate the way in which dependent types can be used to form other dependent types, adding a further layer of complexity and power to higher order logic.

Chapter 3

Lean

In this particular chapter, our primary focus will be on the Lean programming language. Lean is an interactive theorem prover and functional programming language that supports general functional programming, dependent types, and theorem proving.

We will start by introducing the basics of Lean and dependent types. We aim to provide a clear understanding that is easy to follow.

Next, we will implement a memory safe collection using dependent types. This will be done in such a way that runtime checks for index out-of-range errors are unnecessary.

In the end result, we will design a sorting algorithm using Lean, which will be formally proven to always terminate and yield a correct result.

3.1 Why Lean

Plenty of programming languages use dependent types, like Coq, Agda, and Idris. However, Lean stands out for several reasons.

First, Lean is excellent for writing both programs and mathematical proofs. Whereas Coq is designed only as an interactive theorem prover and Idris as the programming language, the last version of Lean has not only

one of the most extensive math libraries but also a great design for writing practical programs. For example, it has a very powerful macros system, which even allows one to write Python-like imperative programs in a purely functional language. Lean is brilliant about making code run fast. Lean can change this into faster code if you write code that is usually slow because of how pure functional code handles data – like linked lists or piano numbers. It can convert these into more efficient forms, like arrays and integers. This is a significant advantage of Lean – it not only allows to run programs that are proven never to fail or give errors but also allows to make them fast.

Second, it is more user-friendly than other similar languages. Whereas Coq was developed in the 90s and has weird cumbersome syntax and plenty of nonstandardized libraries, Lean is less than a decade old, and the latest version, Lean 4, which is still in development, feels like a completely new language. It has one extensive, systematic library with many proofs and theorems called Mathlib. You can see it as an attempt to formalize all mathematics, as it contains theorems starting from simple arithmetics and ending with topology or category theory.

Moreover, compared to languages like Agda or Idris, it has way bigger community support, contributions from Microsoft Research, and incredible tooling.

So these were the reasons why we chose Lean.

3.2 Language basics

3.2.1 Functions, definitions and types

Definitions are introduced using the `def` keyword:

```
def hello := "Hello world"
```

We can also show the type explicitly:

```
def hello : String := "Hello world"
```

In same way we can define functions

```
def add a b := a + b
```

Or

```
def add (a : Int) (b : Int) : Int := a + b
```

Not only terms can be defined:

```
def str : Type := String
def hi : str := "hi"
```

There is `#eval` keyword: to evaluate some expression:

```
#eval add 2 2 --ouputs 4
```

Or check expression's type by using `#check`:

```
#check add 2 2 -- outputs Int
#check add 2 -- outputs Int -> Int -> Int
#check add 2 -- outputs Int -> Int
```

3.2.2 Polymorphism

To write a generic function, we need to provide additional type argument:

```
def id (α : Type) (a : α) : α := a

#eval id Int 10
#eval id String "hi"
```

To make type argument implicit, we can put it in curly braces:

```
def id {α : Type} (a : α) : α := a

#eval id 10
#eval id "hi"
```

If we will have many functions on same generic type, we can make it as variable:

```
variable {α : Type} {β : Type}
def id (a : α) : α := a
def const (a : α) (b : β) : α := a

#eval const 10 "hi"
```

As you can see, unicode symbols are very common in Lean. This is because it's used as theorem prover so it uses a lot of math symbols.

3.2.3 Structures and inductive types

We can define structures in language:

```
structure Point (t : Type) where
  x : Float
  y : Float
  value : t
deriving Repr
def sample := Point.mk 10.0 11.0 "str"
```

`deriving Repr` means that this structure automatically implements typeclass `Repr`, simmilar to typeclass `Show` in Haskell. More about typeclasses in next chapter.

We also have `with` syntax:

```
def zeroX {α : Type} (p : Point α) : Point α :=
  { p with x := 0 }
```

Most types are defined using inductive types. For example, boolean type can be defined like:

```
inductive Bool where
  | false : Bool
  | true : Bool
```

And linked list like:

```
inductive List (t : Type) where
  | nils : List t
  | cons (value : t) (tail : List t) : List t
```

3.2.4 Typeclasses, monads and do notation

Just like Haskell, Lean supports typeclasses:

```
class Add (α : Type u) where
  add : α → α → α

instance [Add t] : Add (Point t)
  where add p1 p2 := { x := p1.x + p2.x, y := p1.y + p2.y, value :=
   p1.value + p2.value }
```

And also it has most known monads, like `IO`, `Reader`, `State`, etc. Here is example of simple IO program:

```
def main : IO Unit := do
  IO.println s!"Hi, type your name"
  let stdIn ← IO.getStdin
  let name ← stdIn.getLine
  IO.println s!"Hello, {name}!"
```

Besides simple chaining of `Bind` methods, it is also possible to write local mutable functions, while and for loops, etc. Here is example of bubble sort algorithm:

```
  def bubbleSort (arr : Array Int): Id (Array Int) := do
    let mut swapped := true
    let mut n := arr.size
    let mut arr := arr
    while swapped do
      swapped := false
      for i in [0:n-1] do
        if i+1 < arr.size then
          if arr[i] > arr[i+1] then
```

```
            let x : Int := arr[i]
            arr := arr.set! i <| arr[i+1]
            arr := arr.set! (i+1) x
            swapped := true
      n := n - 1
    arr
```

It uses simple `Id` monad just to allow do syntax. Note that language is still purely functional, as these mutable variables are only local. Another interesting fact is that `while`, `for` and even `if` constructions in language are defined through marcos system, they are not built into the language. But we won't cover this topic as it's not relevant to dependent types.

3.3 Propositions and simple proofs

In this section we will see how language could be used as prover and prove some simple statements, as well as look at common language constructions for theorem proving.

3.3.1 Types are first class citizens

3.3.1.1 Universes

We can check types not only of terms, but of types themselves too:

```
#check 10 -- returns Nat
#check Nat -- returns Type
```

But we can also check type of `Type`:

```
#check Type -- returns Type 1
```

`Type 1` also has it's own type `Type 2`, and so on:

```
#check Type -- returns Type 1
#check Type 1 -- returns Type 2
#check Type 2 -- returns Type 3
```

```
#check Type 3 -- returns Type 4
```

So Lean, as it's based on calculus of coinductive constructions, supports universes. We can write this hirerarchy in this table:

sort	Prop (Sort 0)	Type (Sort 1)	Type 1 (Sort 2)	Type 2 (Sort 3)
type	True	Bool	Nat → Type	Type → Type 1
term	trivial	true	fun n ⇒ Fin n	fun (_ : Type) ⇒ Type

3.3.1.2 Prop universe

Note `Sort 0` universe in table above. This is a unique universe of types used for writing propositions. It contains ⊤ type, ⊥ type, conjunction, disjunction, and so on. For example, ⊤ and ⊥ types are defined as:

```
inductive True : Prop where
  | intro : True


inductive False : Prop
  -- no constructors for empty type
```

So Lean is not strictly and fully qualifies for Curry-Horward correspondence, as only types from `Prop` universe can be used in propositions and theorems.

3.3.2 Simple proofs

3.3.2.1 Universal quantifier

As Lean supports dependent types, let's prove statement from 2.4.3.1 Universal quantifier:

```
variable {α : Type} {p : α -> Prop}
theorem t : (∀ x y : α, p x ∧ p y) -> ∀ z : α, p z :=
  fun h: ∀ x y : α, p x ∧ p y =>
  fun z : α => (h z z).left
```

The keyword `theorem` in Lean serves a similar role to `def`, but it's used to indicate that the definition is used as a theorem, rather than as a variable.

Here, $p : \alpha \to \text{Prop}$ is our predicate. It's a function that takes a variable of type α and maps it to a proposition in the `Prop` universe.

Our goal is to prove the statement: given that $\forall x, y : \alpha, p(x) \wedge p(y)$, it implies $\forall z : \alpha, p(z)$. Notice that implication is equivalent to a function. So, to prove our statement, we need to define a function that takes as input $\forall x, y : \alpha, p(x) \wedge p(y)$ and produces as output $\forall z : \alpha, p(z)$.

That's what we're doing with the function h. This function takes a function (the universally quantified conjunction $\forall x, y : \alpha, p(x) \wedge p(y)$) as input.

Remember that instances of Π types are functions, and to prove $\forall z : \alpha, p(z)$, we need to provide a function that takes any term z of type α and returns $p(z)$.

We can derive $p(z)$ from the original proposition $\forall x, y : \alpha, p(x) \wedge p(y)$ by replacing both x and y with z. This substitution gives us $p(z) \wedge p(z)$.

Finally, in Lean, the method `.left` is used to extract the left component of a conjunction, so `h(z, z)).left` retrieves the left $p(z)$ from the conjunction $p(z) \wedge p(z)$, proving our theorem.

3.3.2.2 Existential quantifier

In the same way, let's prove theorem with $\sum$ types:

```
variable {α : Type} {p : α -> Prop}
theorem t2 : (∃ x : α, p x ∧ q x) -> ∃ x : α, q x ∧ p x  :=
  fun h: (∃ x : α, p x ∧ q x) =>
  Exists.elim h
    (fun w =>
     fun hw : p w ∧ q w =>
     show ∃ x, q x ∧ p x from ⟨w, hw.right, hw.left⟩)
```

`Exists.elim` is a function that takes two arguments: an existential state-

ment (like h), and a function that takes an instance of the existential statement and a proof that the instance satisfies the statement, and returns a conclusion. The goal of Exists.elim is to use the existential statement to prove the conclusion. In the end, this proof is equivalent to this:

Proof. Assume that there exists an x of type α that satisfies both $p(x)$ and $q(x, h)$. Then, we can prove that there exists an x of type α that satisfies both $q(x)$ and $p(x)$ by taking any instance w of α that satisfies $p(w) \wedge q(w)$, and constructing an existential proof that it satisfies $q(w) \wedge p(w)$. □

3.3.3 Equality

Equality is defined as reflexivity:

```
inductive Eq : α → α → Prop where
  | refl (a : α) : Eq a a
```

This constructor is very powerful - as Calculus of Constructions have a computational interpretation, and that the logical framework treats terms with a common reduct as the same, we can prove non obvious terms very easilly:

```
variable (α β : Type)

example (f : α → β) (a : α) : (fun x => f x) a = f a := Eq.refl _
example (a : α) (b : β) : (a, b).1 = a := Eq.refl _
example : 2 + 3 = 5 := Eq.refl _
```

There is very useful substitution principle - if $a = b$ and $P(a)$ holds, then $P(b)$ also holds. In Lean, it is defined as `Eq.subst` or by ▷ operator. By using substitution, we can define transitivity and symmetry:

```
def rfl {α : Sort u} {a : α} : Eq a a := Eq.refl a

theorem Eq.symm {α : Sort u} {a b : α} (h : Eq a b) : Eq b a :=
```

```
  h ▷ rfl

theorem Eq.trans {α : Sort u} {a b c : α} (h₁ : Eq a b) (h₂ : Eq b c) :
   Eq a c :=
  h₂ ▷ h₁
```

3.3.4 Natural numbers and recursion

3.3.4.1 Definition

Natural numbers are defined as inductive type:

```
  inductive Nat where
    | zero : Nat
    | succ (n : Nat) : Nat
```

They are defined as Peano numbers. The smallest natural number - 0, is defined by first constructor. Them, we can define 1 as $succ0$, 2 as $succ(succ0)$ and so on.

3.3.4.2 Recursion

Then, we can define addition and multiplication using following functions:

```
def Nat.add : (@& Nat) → (@& Nat) → Nat
  | a, Nat.zero   => a
  | a, Nat.succ b => Nat.succ (Nat.add a b)

def Nat.mul : (@& Nat) → (@& Nat) → Nat
  | _, 0          => 0
  | a, Nat.succ b => Nat.add (Nat.mul a b) a
```

3.3.4.3 Termination

Let's try implement division:

```
def div (n : nat) (k : nat) : nat :=
  if n < k then
    0
  else nat.succ (div (n - k) k)
```

But this code won't compile:

```
fail to show termination for

with errors
argument #1 was not used for structural recursion
  failed to eliminate recursive application

argument #2 was not used for structural recursion
  failed to eliminate recursive application

structural recursion cannot be used

failed to prove termination, use ‘termination_by‘ to
specify a well-founded relation
```

The requirement for functions in Lean to be provably terminable is set as a default. This means we cannot write and compile functions that we cannot prove to terminate – to prevent potential endless loops. Lean is intelligent enough to discern from pattern matching and the iterative decrement of $n-1$ in each recursive call, like in the examples of addition and multiplication, that recursion will not go indefinitely. However, it cannot guarantee the termination of the division without supporting evidence; thus, we need to furnish proof of this.

Alternatively, we could use the `partial` keyword in the function declaration. Partial functions refer to those that do not return a value for every possible input from their respective input types.

3.3.4.4 Comparison

We can define $n \leq m$ inductively - $n \leq n$ as first step, and $if\ n \leq m\ then\ n \leq m + 1$ as induction:

```
inductive Nat.le (n : Nat) : Nat → Prop
  | refl     : Nat.le n n
  | step {m} : Nat.le n m → Nat.le n (succ m)
```

Then, we can define $n \leq m$ as $n \leq m = n \leq m + 1$

```
Nat.lt (n m : Nat) : Prop :=
  Nat.le (succ n) m
```

And $n \geq m$ as $m \leq n$:

```
Nat.gt (n m : Nat) : Prop :=
  Nat.le m n
```

3.4 Tactics and automated theorem proving

Tactics are commands or instructions that describe how to build a proof. Informally, you might begin a mathematical proof by saying, "to prove the forward direction, unfold the definition, apply the previous lemma, and simplify." Just as these are instructions that tell the reader how to find the relevant proof, tactics are instructions that tell Lean how to construct a proof term. They naturally support an incremental style of writing proofs, in which you decompose a proof and work on goals one step at a time.

Here is example how tactics' usage:

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

```
  exact hp
  apply And.intro
  exact hq
  exact hp
```

The `apply` tactic applies an expression, viewed as denoting a function with zero or more arguments. It unifies the conclusion with the expression in the current goal, and creates new goals for the remaining arguments, provided that no later arguments depend on them. In the example above, the command apply `And.intro` yields two subgoals:

```
case left
p q : Prop
hp : p
hq : q
⊢ p

case right
p q : Prop
hp : p
hq : q
⊢ q ∧ p
```

The first goal is met with the command `exact hp`. The `exact` command is just a variant of `apply` which signals that the expression given should fill the goal exactly. It is good form to use it in a tactic proof, since its failure signals that something has gone wrong. It is also more robust than `apply`, since the elaborator takes the expected type, given by the target of the goal, into account when processing the expression that is being applied. In this case, however, `apply` would work just as well.

The assumption tactic looks through the assumptions in context of the current goal, and if there is one matching the conclusion, it applies it:

```
example (x y z w : Nat) (h₁ : x = y) (h₂ : y = z) (h₃ : z = w) : x = w :=
  by
```

```
  apply Eq.trans h₁
  apply Eq.trans h₂
  assumption   -- applied h₃
```

There are plenty other tactics, like `intro` which introduces hypothesis, simmilar to introducing a function without tactics, `cases` which is used for pattern matching. Or `generalize` which makes generalization from some concrete statement.

Another useful tactic is `rw`, which takes statement simmilar to `x = y` statements and replaces x with y for the next goal. With it, we and another `calc` tactic, we can write proofs as chain of equality, simmilar to how we used to solve tasks from calculus courses at school:

```
example (x y : Nat) : (x + y) * (x + y) = x * x + y * x + x * y + y * y
    :=
  calc
    (x + y) * (x + y) = (x + y) * x + (x + y) * y  := by rw [Nat.mul_add]
        _ = x * x + y * x + (x + y) * y            := by rw [Nat.add_mul]
        _ = x * x + y * x + (x * y + y * y)        := by rw [Nat.add_mul]
        _ = x * x + y * x + x * y + y * y          := by rw
    [←Nat.add_assoc]
```

`calc` tactic uses statements which are instances of `Trans` typeclass:

```
class Trans (r : α → β → Sort u) (s : β → γ → Sort v) (t : outParam
    (α → γ → Sort w)) where
  /-- Compose two proofs by transitivity, generalized over the relations
    involved. -/
  trans : r a b → s b c → t a c
```

3.4.1 Automated theorem proving

Based on *rewrite*, there are very powerful `simp` tactic. This tactic accepts multiple theorems which show equality, which look like `_ = _` and tries to apply them to get the proof. For example, we do not need to prove the

proposition from the previus example manually, `simp` tactic can do this for us. All we need to do is provide it with theorems of equality which it can use:

```
example (x y : Nat) : (x + y) * (x + y) = x * x + y * x + x * y + y * y
    :=
  by simp[Nat.add_assoc, Nat.mul_add, Nat.add_mul]
```

There are plenty of theorems added to `simp` tactic's "database". To add new, you can use `@[simp]` attribute:

```
@[simp] theorem add_assoc1 : ∀ (n m k : Nat), (n + m) + k = n + (m + k)
    := Nat.add_assoc
@[simp] theorem mul_add1 (n m k : Nat) : n * (m + k) = n * m + n * k :=
    Nat.mul_add n m k
@[simp] theorem add_mul1 (n m k : Nat) : (n + m) * k = n * k + m * k :=
    Nat.add_mul n m k

example (x y : Nat) : (x + y) * (x + y) = x * x + y * x + x * y + y * y
    :=
  by simp
```

Then, as we can see, proof of such complex statement can be proven entirely by `simp` tactic. So, Lean is capable of prooving simple theorems automatically.

3.5 Optimisations

In addition to robust proofs and an expressive type system, Lean incorporates several significant optimizations. One of these is its distinctive garbage collection process. Rather than employing a typical garbage collector, Lean utilizes a reference-counting method. The compiler identifies the number of references to an object in the code, and when the count drops to zero, the object is automatically deallocated.

Type	**Logical representation**	**Run-time Representation**
Nat	Unary, with one pointer from each Nat.succ	Efficient arbitrary-precision integers
Int	A sum type with constructors for positive or negative values, each containing a Nat	Efficient arbitrary-precision integers
UInt8, UInt16, UInt32, UInt64	A Fin with an appropriate bound	Fixed-precision machine integers
Char	A UInt32 paired with a proof that it's a valid code point	Ordinary characters
String	A structure that contains a List Char in a field called data	UTF-8-encoded string
Array α	A structure that contains a List α in a field called data	Packed arrays of pointers to α values
Sort u	A type	Erased completely
Proofs of propositions	Whatever data is suggested by the proposition when considered as a type of evidence	Erased completely

Lean also optimizes array operations using the same reference counting principle. A typical operation using the method `Array.set index value` in Lean, a purely functional language, would create a new array similar to the original, with one value changed. This would typically necessitate an operation cost of $O(n)$. However, if the reference counter identifies only one reference to such an array exists, the compiler will transform the code into an imperative array mutation, significantly reducing the operation cost to $O(1)$.

Lean provides optimizations of different types as well. For instance, during runtime, the compiler replaces all natural numbers of type `Nat` with regular integers. It also eliminates all proofs from the runtime, as they are only required during compilation for type checks. Even arrays, initially defined as wrappers around lists at the code level, are transformed into actual arrays by the compiler. Below is a table summarizing most of these transformations:

`Fin n` is an important and oftenly used type which shows some natural number i with a proof that $i \leq n$. It is defined as:

```
structure Fin (n : Nat) where
  val  : Nat
  isLt : val < n
```

3.6 Lists with type safe indexes

We already implemented bubble sort as an example. However, that example uses nonsafe `Array.get` - if we try to access an index greater than the array size, an error will be thrown. Let us look at how we can make a type sef variant of `List.get` method. The first thing we can do, is to wrap it in Option Monad:

```
def List.get? {α : Type u} : (as : List α) → Nat → Option α
  | nil, _ => none
  | cons a _,  0 => some a
  | cons _ as, Nat.succ i => get? as i
```

Note that if we skip the first case when the list is nil, the compiler will throw an error. So it forces us to write safe code and cover all possible cases.

However, what if we already have proof that accessing the index is less than the array size? We can define such a function using described above `Fin n` type:

```
def List.get {α : Type u} : (as : List α) → Fin as.length → α
  | cons a _,  ⟨0, _⟩ => a
  | cons _ as, ⟨Nat.succ i, h⟩ => get as ⟨i, Nat.le_of_succ_le_succ h⟩ --
    theorem that n + 1 <= m + 1 -> n <= m
```

In that case, the compiler will understand there is no case when the array is nil and provided index is less than nil.length. Notice that `List.length` is not even a property. Instead, it is a function that accepts an array and returns a natural number. This shows how dependent types are powerful.

Remember that Lean has optimization for Arrays, which look like lists in the code but are compiled into arrays in the end? Same applies to `Array.get` function – it is not only type-safe but also fast and compiles to the same, fast, without runtime checks code as in C or C++.

3.7 Formally proven sorting algorithm

Now let's rewrite bubble sort to use typesafe variant:

```
def bubbleSortSt (arr : Array Int): Id (Array Int) := do
  let mut swapped := true
  let mut n := arr.size
  let mut arr := arr
  while swapped do
    swapped := false
    for i in [0:n-1] do
      if h : i+1 < arr.size then
        let t1 := Nat.lt_of_succ_lt h
        if arr.get ⟨i, t1⟩ > arr.get ⟨i+1, h⟩ then
          arr := arr.swap ⟨i, t1⟩ ⟨i+1, h⟩
          swapped := true
    n := n - 1
  arr
```

`if h : i+1 < arr.size then` checks whether given boolean is true, and if it is true, it returns the corresponding proposition. So h is proof that $i+1 \leq arr.size$, then, by having this proof, we are constructing the proof $t1$ that i is also lesser than $arr.size$. After this, we are taking these proofs for accessing array elements and doing a swap.

However, this approach is as practical as getting index in `Option` monad – we are still checking the bounds of the array, even if it is obvious from `for i in [0:n-1]` that i is always less than the array size. Let us rewrite the algorithm to recursion and try to avoid this check. First, we will define the

inner loop:

```
def bubbleSortInnerLoop(arr: Array Int) (n: Fin arr.size): (Array Int) ×
    Bool :=
match n with
| ⟨0, _⟩ => (arr, false)
| ⟨n' +1, _⟩ =>
  have p : n' < arr.size := by
    simp [Nat.lt_of_succ_lt, *]
  if arr[n'] > arr[n] then
    let res := bubbleSortInnerLoop (arr.swap ⟨n', by assumption⟩ n) ⟨n',
   by simp[*]⟩
    (res.fst, true)
  else bubbleSortInnerLoop arr ⟨n', p⟩
```

This function return both array and `isSwapped`. There is a theorem that `Arr.swap` does not change array size, so we are calling recursion with just `by simp[*]` proof.

The outer loop is defined as:

```
def bubbleSortOuterLoop (arr: Array Int) (n : Fin arr.size) :=
match n with
| ⟨0, _⟩ => arr
| ⟨n' +1, isLt⟩ =>
  let (res, swapped) := bubbleSortInnerLoop arr n
  if not swapped then res else
  have p : n' < res.size := by
    simp[*,bubbleSortInnerLoopEqLength]
  bubbleSortOuterLoop res ⟨n', p⟩
```

However, to prove that $n' \leq res.size$, we need to prove that the inner loop does not change. Here is its definition and proof:

```
theorem bubbleSortInnerLoopEqLength (len : Nat) (i : Nat) :
    (arr : Array Int) → (isLt : i < arr.size) → (arr.size = len) →
    (bubbleSortInnerLoop arr ⟨i, isLt⟩).fst.size = len := by
  induction i with
```

```
  | zero =>
    intro arr isLt hLen
    simp [bubbleSortInnerLoop, *]
  | succ i' ih =>
    intro arr isLt hLen
    simp [bubbleSortInnerLoop, *]
    split <;> try assumption
    simp [*]
```

Finally, composing function will be:

```
def bubbleSortProven (arr: Array Int) :=
have t : arr.size < arr.size + 1 := by simp[Nat.lt_succ_self]
let n : Fin (arr.size + 1) := ⟨arr.size, t⟩
match n with
| ⟨0, _ ⟩  => arr
| ⟨succ n', lt⟩  =>
  have p1 : n' < arr.size := by
    simp [Nat.lt_of_succ_lt_succ, *]
  bubbleSortOuterLoop arr ⟨n', p1⟩
```

Note that Lean enforces us to check all cases. If we skipped matching on `(0, _ )`, compiler will throw an error that not all cases are covered.

3.7.0.1 Quicksort attempt

Let us try a harder algorithm – quicksort. First, let us define it without any type checks:

```
def qpartition (as : Array Int) (lo hi : Nat) : Nat × Array α :=
  let mid := (lo + hi) / 2
  let as  := if lt (as.get! mid) (as.get! lo) then as.swap! lo mid else
   as
  let as  := if lt (as.get! hi)  (as.get! lo) then as.swap! lo hi  else
   as
  let as  := if lt (as.get! mid) (as.get! hi) then as.swap! mid hi else
   as
```

```
  let pivot := as.get! hi
  partial let rec loop (as : Array α) (i j : Nat) :=
    if h : j < hi then
      if lt (as.get! j) pivot then
        let as := as.swap! i j
        loop as (i+1) (j+1)
      else
        loop as i (j+1)
    else
      let as := as.swap! i hi
      (i, as)
  loop as lo lo

partial def qsort (as : Array α) (low := 0) (high := as.size - 1) :
    Array α :=
  let rec sort (as : Array α) (low high : Nat) :=
    if low < high then
      let p := qpartition as low high;
      let mid := p.1
      let as  := p.2
      if mid >= high then as
      else
        let as := sort as low mid
        sort as (mid+1) high
    else as
  sort as low high
```

We can prove that that in `qpartition` after first 3 swaps size of array is the same:

```
def swapIfTrue (bool : Bool) (arr : Array Int) (i j : Fin arr.size):
    Array Int :=
  if bool then arr.swap i j else arr

theorem swap_eq_size (arr : Array Int) (i j : Fin arr.size) (bool :
    Bool) :
```

```
    (swapIfTrue bool arr i j).size = arr.size := by
  induction bool with
  | false => simp[swapIfTrue]
  | true => simp[swapIfTrue]

def qpartition (as : Array Int) (lo hi : Fin as.size) : Nat × Array Int
    :=
  let mid := divBy2 as.size lo hi

  let as  := swapIfTrue ((as.get  mid) < (as.get lo)) as lo hi
  let lo' : Fin as.size := ⟨lo.val, by simp[lo.isLt, swap_eq_size]⟩
  let hi' : Fin as.size := ⟨hi.val, by simp[hi.isLt, swap_eq_size]⟩

  let as  := swapIfTrue ((as.get hi') < (as.get lo')) as lo' hi'

  let mid' : Fin as.size := ⟨mid.val, by simp[mid.isLt, swap_eq_size]⟩
  let hi' : Fin as.size := ⟨hi.val, by simp[hi.isLt, swap_eq_size]⟩

  let as  := swapIfTrue ((as.get mid') < (as.get hi')) as mid' hi'

  let mid : Fin as.size := ⟨mid.val, by simp[mid.isLt, swap_eq_size]⟩
  let hi : Fin as.size := ⟨hi.val, by simp[hi.isLt, swap_eq_size]⟩
  let lo : Fin as.size := ⟨lo.val, by simp[lo.isLt, swap_eq_size]⟩
  let pivot := as.get hi
```

But prooving index safety in the inner loop becomes too complicated. Let us at least prove that the inner loop is not a partial function. It is somewhat simple – recursion occurs only when we $j \leq i$, and we always increase j in recursive calls. So all we have to do is write `termination_by _ => hi - j`. Doing this gives the compiler explanation that, at the very least, the inner loop is guaranteed never to encounter a deadlock situation.

3.7.0.2 Proving equivalence of functions

As comparing two sorting functions is too complicated, let us do this on a more simple example - summing all elements in the list. Consider two functions – one with tail call optimization and one without:

```
def slowSum : List Nat → Nat
  | [] => 0
  | x :: xs => x + slowSum xs
def sumTailCall' (acc : Nat) : List Nat → Nat
  | [] => acc
  | x :: xs => sumTailCall' (x + acc) xs
def sumTailCal (list : List Nat) : Nat :=
  sumTailCall' 0 list
```

Then, by applying tactics `funext` which takes any value xs and proves that $f(xs) = g(xs)$, and `induction`, which is mathematical induction, we can prove that $\forall xs slowSum(xs) = sumTailCal(xs)$:

```
theorem non_tail_sum_eq_tail_sum : slowSum = sumTailCal := by
  funext xs
  simp [sumTailCal]
  rw [←Nat.zero_add (slowSum xs)]
  exact non_tail_sum_eq_helper_accum xs 0


theorem non_tail_sum_eq_helper_accum (xs : List Nat) :
    (n : Nat) → n + slowSum xs = sumTailCall' n xs := by
  induction xs with
  | nil => intro n; rfl
  | cons y ys ih =>
    intro n
    simp [sumTailCall', sumTailCal]
    rw [←Nat.add_assoc]
    rw [Nat.add_comm y n]
    exact ih (n + y)
```

We can utilize this instrument for more than just simple algorithms; it

can also be applied to business logic. Imagine a program where business rules are defined as theorems – it would significantly reduce the number of tests required.

Chapter 4

Hall's Marriage Theorem

This chapter will examine how the Lean theorem prover can be applied to more complex mathematical theorems. We will focus on Hall's Marriage Theorem as we examine its formalization within the Lean framework. Based on this, we will undertake the challenge of formalizing a similar theorem. Additionally, we will develop a simple program that leverages the proof of this theorem for practical use.

4.1 Mathlib

Mathlib is a library for the Lean theorem prover. It is a community-driven project to develop a comprehensive, robust, well-documented library of formalized mathematics using Lean. The project is meant to support various applications, including verifying formal systems, formalizing mathematics, and teaching.

The Mathlib library covers various mathematical topics, from elementary mathematics (natural numbers and basic algebra) to more advanced areas (like topology, algebraic geometry, and number theory). It also provides foundational theories, such as set theory and logic.

Using Mathlib, users can leverage the work of others to simplify their

formalizations, relying on the library's definitions, theorems, and methods. It also serves as a platform for collaboration among users of Lean and encourages the sharing and reuse of formalized mathematics.

Mathlib is maintained by an active community of contributors, including professional mathematicians, computer scientists, students, and hobbyists.

Hall's theorem is also defined in Mathlib, so we will look at the sources of this library.

4.2 Definition

Hall's marriage theorem, proved by Philip Hall (1935), is a theorem with two equivalent formulations. In each case, the theorem gives a necessary and sufficient condition for an object to exist:

4.2.1 Combinatorial formulation

Consider $\mathcal{F}$ to be a finite collection of sets. Let X represent the union of all sets in $\mathcal{F}$, signifying the collection of all elements that exist in at least one set of $\mathcal{F}$. A transversal for $\mathcal{F}$ is a subset of X which can be composed by selecting a unique element from each set in $\mathcal{F}$. We can formalize this notion by defining a transversal as the range of an injective function $f : \mathcal{F} \to X$ such that $f(S) \in S$ for each $S \in \mathcal{F}$. A transversal is also known as a system of distinct representatives.

The family $\mathcal{F}$ meets the marriage condition if every subfamily of $\mathcal{F}$ consists of at least as many distinct elements as there are sets in it. Formally, for all $\mathcal{G} \subseteq \mathcal{F}$, $|\mathcal{G}| \leq |\bigcup_{S \in \mathcal{G}} S|$. If a transversal exists, then the marriage condition is met: the function f used to establish the transversal maps $\mathcal{G}$ to a subset of its union, of a size equal to $|\mathcal{G}|$, implying that the total union must be at least as large. The Hall's theorem states that the converse is also

valid:

Hall's Marriage Theorem 1. *A finite family $\mathcal{F}$ of sets has a transversal if and only if $\mathcal{F}$ satisfies the marriage condition.*

This is a formulation which already exists in mathlib.

4.2.2 Graph theoretic formulation

Let $G = (X, Y, E)$ be a finite bipartite graph with bipartite sets X and Y, and edge set E. An X-perfect matching (also known as an X-saturating matching) is a set of disjoint edges that covers every vertex in X.

For a subset W of X, let $N_G(W)$ denote the neighborhood of W in G, which is the set of all vertices in Y that are adjacent to at least one element of W. The marriage theorem in this formulation states that there exists an X-perfect matching if and only if for every subset W of X:

$$|W| \leq |N_G(W)|$$

In other words, every subset W of X must have a sufficient number of neighbors in Y.

There is no such formulation in mathlib, so we will define it.

4.2.3 Equivalence of the combinatorial formulation and the graph-theoretic formulation

Given any bipartite graph, denoted as $G = (X, Y, E)$, we can establish a finite collection of sets. This collection constitutes the neighborhood sets of all vertices within X. A unique representative system for this set collection directly corresponds to an X-perfect matching within G. This equivalency

demonstrates that the combinatorial formulation for finite sets and the graph-theoretic formulation for finite graphs are, in essence, equivalent.

The initial aim of this thesis was to establish this equivalence utilizing the Lean proof assistant and make significant contributions to the Mathlib library. However, the focus has since been redirected towards a more practical and applicable program.

4.3 Using Lean

4.3.1 Basics

Before moving to formulations, let's look how basic mathematical principles are defined in mathlib:

4.3.1.1 Sets

Set of type α is defined just as partial function on α:

```
def Set (α : Type u) := α → Prop
```

We can define a finite set of type α as some array, with proof that there are no dublicates in such array:

```
structure Finset (α : Type _) where
  val : Multiset α
  nodup : Nodup val
```

An important function we will use is `bunion`$_i$ `(s : Finset` α`) (t :` $\alpha \to$ `Finset` β`) : Finset` β which is simmilar to math expression $\bigcup_{x \in \alpha} t(x)$.

4.3.1.2 Finite type

For showing that some type l is finite, we have a typeclass, which requires such type to contain a bijection on some instance of descrived in previous chapter `Fin n` type:

```
class inductive Finite (a : Sort _) : Prop
  | intro (n : Nat) : a = Fin n -> Finite _
```

There is also a function `univ : α : Type u -> [inst : Fintype α] → Finset α` which makes a finite set from provided finite type.

4.3.1.3 Injective function

It's definition is very simple and easy to read:

```
def Injective (f : α → β) : Prop := ∀ {a₁ a₂}, f a₁ = f a₂ → a₁ = a₂
```

4.3.2 Combinatorial formulation in terms of Lean

Let l be a finite type, which would correspond to family F. Then, to express that l contains subsets of α, we need to provide a function $t : l \to Finset\alpha$.

Marriage condition can be defined as $\forall s : Finset, s.card(s.bunion_i t).card$.

And mathcing can be defined as injective function from l to α, that $\forall_{x:l} f(x) \in t(x)$

Then, entire theorem can be defined as:

```
theorem HallsComb {ι α : Type _} [Finite ι]
    [DecidableEq α] (t : ι → Finset α) :
    (∀ s : Finset ι, s.card ≤ (s.bunion_i t).card) ↔
      ∃ f : ι → α, Function.Injective f ∧ ∀ x, f x ∈ t x := _
```

4.3.3 Graph theoretic formulation in terms of Lean

Now, as exercise, let's define this theorem using graphs. We can define simple graph just as set of adjustment vertices, where vertices are defined as finite type:

```
structure SimpleGraph (V : Type u) [Fintype V] where
  Adj : V → V → Prop
```

A subgraph of a given graph is defined by a subset of vertices, edges, and the demonstration that all vertices from these edges belong to the vertex subset. Additionally, there must be proof that these edges exist in the original graph as well:

```
structure Subgraph {V : Type u} [Fintype V] (G : SimpleGraph V) where
  verts : Finset V
  Adj : V → V → Prop
  adj_sub : ∀ {v w : V}, Adj v w → G.Adj v w
  edge_vert : ∀ {v w : V}, Adj v w → v ∈ verts
```

Bipatride graph is defined by providing entire graph and two subgraphs X and Y, with proofs that these subgraphs are disjoint:

```
structure BiPatr (α : Type u) [Fintype α] [DecidableEq α] where
  G : SimpleGraph α
  X : Finset α
  Y : Finset α
  V : Subgraph G
  U : Subgraph G
  disJ : Disjoint X Y
  complete : V.verts = X ∧ U.verts = Y ∧ ∀ v ∈ V.verts, ∀ u ∈ U.verts, ¬
   G.Adj u u ∧ ¬ G.Adj v v
```

Then, perfect matching can be defined as:

```
def isPerfectMatching (G : BiPatr α) (M : Subgraph (G.G)): Prop :=
  ∀ v, v ∈ M.verts → ∃! w, M.Adj v w ∧ G.V.verts ⊆ M.verts
```

And neighborset as filter on adjustments:

```
def neighborSet (G : BiPatr α) [DecidableRel G.G.Adj] (v : α) : Finset α
   :=
  Finset.filter (G.G.Adj v) Finset.univ
```

By having defined neighborset, we can define marriage conditin:

```
  def condition (G : BiPatr α) [DecidableRel G.G.Adj]: Prop :=
    ∀ W : Finset α, W ⊆ G.V.verts -> W.card ≤ (W.bunionᵢ (neighborSet
   G)).card
```

In result, entire theorem can be defined as this code:

```
theorem hallGraph (G : BiPatr α) [DecidableRel G.G.Adj]:
(∀ W : Finset α, W ⊆ G.V.verts -> W.card ≤ (W.bunionᵢ (neighborSet
   G)).card) ↔
  (∃ M : Subgraph G.G, isPerfectMatching G M) := _
```

4.3.4 Real world example

Consider we have a dating app for heterosexual people. Let's make an algorithm which checks on likes people gave to each other, whether is possible create a couple for every person in a group.

For simplicity, let's consider only women's preference. Firstly, we need to create a structure for each person:

```
structure Man : Type where
id : Int
name : String

structure Woman : Type where
id : Int
name : String
liked : Finset Int -- a list of men given woman has liked
```

Then, by having a list of women with their likes, we can create such function:

```
def getMarriages (women : Women): Option (Woman -> Int) :=
  if proof : women.women.sublists.all (fun women => women.length ≤
   women.concatMap(fun w => w.liked).length) then
    (Finset.hall (fun woman => woman.liked).mp proof).choose
  else none
```

The function `mp` accepts an input $a \leftrightarrow b$ and yields an output $a \to b$. We demonstrate that for any given set of women, the total count of men who are found attractive by at least one woman in the set is always greater than or

equal to the set size. Hence, there exists an injective function f : Woman $\rightarrow$ Int, where $\forall$woman, f(woman) $\in$ (fun woman) $\implies$ woman.liked. This function is obtained by applying `choose`.

Chapter 5

Conclusion

In conclusion, this thesis delved into the realm of programming language types, exploring their theoretical foundations and practical applications. Throughout our research, we discovered the immense power of types as a tool for formal logic and mathematical proofs. Furthermore, by leveraging types, we were able to establish numerous simple theorems and define more complex ones, such as Hall's theorem.

In parallel, we showcased the practical utility of types, primarily dependent types, in real-world programming. Using dependent types can lead to safer and more reliable programs by minimizing runtime checks and redundant unit tests. The sorting algorithm we implemented and analyzed presented a compelling case for this argument, as it reduced runtime verifications and lessened the need for certain unit tests.

However, it turned out that dependent types are not always the best solution. For example, when trying to implement the quicksort algorithm, we came to the conclusion that the more complex algorithm gets, the more complex proof of its correctness becomes. It shows that even if dependent types are powerful tools, it is not very practical to use them to get rid of *all* errors – but they still could be useful in simpler fields for writing business logic.

We also should remember Gödel's incompleteness theorems as they tell us that not all propositions or theorems could be defined formally, with dependent types in our case. So we need to keep in mind that even if Lean is a handy tool, it cannot prove everything.

Overall, we can conclude that dependent types are very powerful tools, and we are sure that more and more programming languages will introduce them.

Bibliography

[1] Stanford Encyclopedia of Philosophy *Type Theory* (https://plato.stanford.edu/entries/type-theory/)

[2] Stanford Encyclopedia of Philosophy *Intuitionistic Type Theory* (https://plato.stanford.edu/entries/type-theory-intuitionistic/)

[3] Nino Guallart *An overview of type theories* (https://arxiv.org/pdf/1411.1029.pdf)

[4] Lean Community *Theorem Proving in Lean 4* (https://leanprover.github.io/theorem_proving_in_lean4/)

[5] David Thrane Christiansen *Functional Programming in Lean* (https://leanprover.github.io/functional_programming_in_lean)

[6] Lean Community *A mathlib overview* (https://leanprover-community.github.io/mathlib-overview.html)

[7] Hall, Philip (1935) *On Representatives of Subsets* (https://doi.org/10.1112%2Fjlms%2Fs1-10.37.26)

[8] Reichmeider, P.F. (1984) *The Equivalence of Some Combinatorial Matching Theorems* (http://robertborgersen.info/Presentations/GS-05R-1.pdf)

[9] Janis A. Barzdins and Edgars Celms *To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub* (`https://arxiv.org/pdf/2203.11115v1.pdf`)