Proofs and Programs

A Perspective on Formal Mathematics and Computation

Robert Culling

September 2, 2024

Contents

Chapter 1

Introduction

```
-- God created the integers;
inductive Peano where
  | zero : Peano
  | succ : Peano → Peano
```

Acknowledgements

These lecture notes have been made with reference to the following resources which should be consulted for further details on the subjects:

1. Logic and Structure, Drik van Dalen [2]
2. Type Theory and Functional Programming, Simon Thompson [3]
3. Stanford Encyclopedia of Philosophy
4. Theorem Proving in Lean 4, Jeremy Avigad et el.
5. MathLib, Leanprover Community.

These lecture notes were also informed by lecture notes originally authored by Douglas Bridges. Earlier adaptions of those notes had been made by Maarten McKubre-Jordens and Hannes Diener. Louis Warren showed great patience in helping me prepare the first time I taught the course. I am thankful to each of these men for bringing the subject to my attention, teaching me so much about it when I was an undergrad, and helping me develop as a mathematician.

Chapter 2

Proof Theory

2.1 Reasoning

When formalising an argument both the content of the terms in the argument and the way the statements are connected matter. We will first talk about propositional logic, which can be thought of as the formalisation of the connective structure of an argument. Only once that is covered will we move on to study first order predicate logic - this allows for content specific expressions. Together they will allow for the expression of mathematical definitions and theorems in such a way that the meta-analysis is amenable to computation.

2.2 Formal Language

In the previous section we began to uncover the structure of propositions and formalised a definition of an argument. We saw that some propositions were direct claims with no extra connective structure - we called these atomic propositions. Other propositions were constructed from atomic propositions by the connective grammar of the English language - these we called compound propositions.

In order to formalise the process of reasoning, we are going to strip away all the content-words in a proposition and substitute capital English letters for propositions. Furthermore, we will introduce certain symbols - logical connectives - to allow for the construction of compound propositions.

Let us denote the collection of all propositions, or the type of propositions, simply as Prop. Terms in Prop are defined according to the following inductive definition:

- **Atomic Propositions:** If P is a single propositional variable, then P is in Prop.
- **Negation:** If α is in Prop, then $(\neg\alpha)$ is in Prop.
- **Implication:** If α and β are in Prop, then $(\alpha \to \beta)$ is in Prop.
- **Conjunction:** If α and β are in Prop, then $(\alpha \wedge \beta)$ is in Prop.
- **Disjunction:** If α and β are in Prop, then $(\alpha \vee \beta)$ is in Prop.

We will make liberal use of the shorthand P : Prop for "P is a proposition" or "P is of type Prop." Propositions are also referred to as well-formed formulae.

Due to the inductive nature of the definition of Prop, each proposition can be represented as by its syntax tree. Each node in the tree is represented by one of the constructors for Prop

Specification of terms in a formal language, like Prop, is often written following the more compact notation of Backus-Naur form.

This compact notation is simply stating that a proposition (term of type Prop) is either an atom, or constructed from component propositions and a logical-connective. Following this form requires one to

always start with an atom and workup using the connectives to write a term of type prop.

2.3 Natural Deductions

Propositional logic has given us a language to express the connective structure of statements made up of many propositions. For now we will not consider any internal structure in the propositions, this will come later. What concerns us in the present section is the process of formalising theorems and proofs. Theorems are collections of statements - a set of hypotheses and a single conclusion - and proofs are some process of unfolding those hypotheses so as to justify the conclusion. To put it another way, it's an explanation of the conclusion in the presence of the hypotheses. What do we mean by justify the conclusion? One answer to this question has become known as the *Brouwer-Heyting-Kolmogorov Interpretation* [1].

Brouwer, Heyting, and Kolmogorov proposed the following (inductive) interpretation of what it should mean to prove statements involving propositional connectives:

$P \wedge Q$	to prove a conjunction we must provide both a proof of P and a proof of Q.
$P \to Q$	to prove an implication we must provide an algorithm for turning a proof of P into a proof of Q.
$P \vee Q$	to prove a disjunction we must provide either a proof of P or a proof of Q.
$\neg P$	to prove a negation we must provide an algorithm that turns a proof of P into a proof of $\perp$.

As this is an inductive definition, there are implicit starting assumptions in a proof. Theorems are hypothetical statements - if we know this, then we may conclude that - whether it be axioms or extra conditions specific to the theorem in question, there is some base case of assumed knowledge to start the inductive process from.

Note: There is a parallel, semantic, approach one can take to analyse the validity of an argument. There notions of True and False, truth tables, and valuations are developed and valid arguments are those that preserve truth. This approach will not be presented in these notes. The *completeness* and *soundness* theorems show that the semantic approach to logic is equivalent to the syntactic approach that we are to take here. One may read *Logic and Structure* for details on the semantics of classical logic [2].

We are going to develop a proof method that was first presented by Gerhard Gentzen, as such it is often referred to as the Gentzen Calculus or the natural deduction calculus. This method will develop proofs by unfolding the hypotheses of theorem in a manner consistent with the BHK. For now the alignment between the BHK and natural deductions will be loose, but as we study the process of computation more it will become air tight — proofs really will be algorithms!

Sequents and Deductions

Throughout these notes we will denote an arbitrary set of hypotheses of an argument by Σ. We will use the following notation to denote the fact that the proposition γ is to be proved from the hypotheses Σ:

$$\Sigma \vdash \gamma$$

We will refer to these as sequents. The symbol $\vdash$ is called a turnstile.

Example: Consider the claim that Q follows from the hypotheses $P \wedge R$ and $P \to Q$. This claim will be stated using the following sequent:

$$P \to Q,\ P \wedge R \ \vdash\ Q$$

Rather than making the claim in natural language.

Example: If we assume $P \wedge Q$, then (we will soon see) we may conclude $Q \wedge P$. In fact, the converse is true as well. That is, from $Q \wedge P$ we may conclude $P \wedge Q$. If we are making a claim that the sequent can be read in both directions, then we write the sequent as

$$P \wedge Q \dashv\vdash Q \wedge P$$

These sequents of course require two proofs, one in each direction; thus both sides of the sequent play the role of hypothesis and conclusion, in turn.

Sequents such as $\Sigma \vdash \gamma$ require natural deductions to witness them. We use the following notation to denote an arbitrary deduction

$$\overset{\Sigma}{\gamma}\mathcal{D}$$

witnessing the proof of the sequent $\Sigma \vdash \gamma$. The details on how to write a deduction witnessing a specific sequent follow in the next section on rules of inference. For now browse the following example to try and get a feel for what's going on. This proof will be revisited and explained later.

Definition: Currying

Example

$$\dfrac{\dfrac{\dfrac{\dfrac{\not{A}^{\,1} \quad \not{B}^{\,2}}{A \wedge B}\wedge I \qquad (A \wedge B) \to C}{C}\text{MP}}{B \to C}\to I, 2}{A \to (B \to C)}\to I, 1$$

For each of the connectives we will present *rules of inference* which dictate how to introduce (prove) and eliminate (make use of) compound statements made with each connective. Proofs then will be a finite tree of instances of these rules of inference.

2.3.1 Implication

Recall that the BHK interpretation of a proof of implication says that, to prove an implication $P \to Q$ we must provide an algorithm for turning a proof of P into a proof of Q. Ignoring the word algorithm again, this says we need a procedure for turning knowledge of P into knowledge of Q.

What this means is that to prove $P \to Q$ we should work under the assumption that we know P and use that assumption to prove Q. We add in a *temporary assumption* in order to write down the required procedure. To be clear, this says nothing about whether P is true; it simply puts us in the correct hypothetical situation to address the question of whether Q follows from the assumption of P. Without assuming knowledge of P how could one expect to determine whether Q would follow from it? Once Q has been derived from the temporary assumption P, we make it clear that P was only temporarily assumed for the purpose of the proof by crossing it out.

Definition: Implication Introduction

If ${}^{\Sigma}_{\beta}\mathcal{D}$ is a deduction of β from Σ, then the deduction can be extended to a deduction of $\alpha \to \beta$ from hypotheses $\Sigma \backslash \{\alpha\}$, which we depict as follows:

$$\dfrac{\begin{array}{c}\Sigma,\ \cancel{\alpha} \\ \mathcal{D} \\ \beta\end{array}}{\alpha \to \beta}\ \to I$$

In programming terms we can think of a proof of $P \to Q$ using implication introduction as a subroutine with a local variable P that returns Q. If we ever have a proof of P lying around, then we can use the subroutine to obtain a proof of Q. But writing the subroutine alone says nothing about whether one could ever prove P, just what can be done if P were known.

Keeping with this theme, if a proof of $P \to Q$ is to be thought of as a program, then we can eliminate (make use of) the implication by applying that program to a proof of P. This requires two proofs; one of $P \to Q$ and one of P. These are combined to produce a proof of Q. This rule of inference is known as *modus ponens*.

Definition: Implication Elimination (Modus Ponens)

If ${}^{\Sigma_1}_{\alpha\to\beta}\mathcal{D}_1$ and ${}^{\Sigma_2}_{\alpha}\mathcal{D}_2$ are deductions, then they can be extended to a deduction ${}^{\Sigma_1\cup\Sigma_2}_{\beta}\mathcal{D}$, which we depict as follows:

$$\dfrac{\begin{array}{cc}\Sigma_1 & \Sigma_2 \\ \mathcal{D}_1 & \mathcal{D}_2 \\ \alpha \to \beta & \alpha\end{array}}{\beta}\ \text{MP}$$

Example: Implication Composition

Example: I Combinator

Example: S Combinator

Example: K Combinator

This example trips a lot of students up. Keep your wits about you.

$$P \vdash Q \to P$$

2.3.2 Conjunction

Recall that the BHK interpretation of conjunction is that: A proof of the conjunction $P \wedge Q$ is an algorithm that takes in a proof of P and a proof Q. If we (for now) brush over the word algorithm, then this simply states that a proof of the conjunction $P \wedge Q$ is a pair consisting of a proof of P and a proof Q — To know $P \wedge Q$ is to know P and to know Q. The following rules of inference are a diagrammatic representation of this idea.

Definition: Conjunction Introduction

If $\begin{smallmatrix}\Sigma_1\\ \alpha\end{smallmatrix}\mathcal{D}_1$ and $\begin{smallmatrix}\Sigma_2\\ \beta\end{smallmatrix}\mathcal{D}_2$ are proofs of α and β respectively, then together they constitute a proof $\begin{smallmatrix}\Sigma_1 \cup \Sigma_2\\ \alpha \wedge \beta\end{smallmatrix}\mathcal{D}$, which we depict as follows:

$$\dfrac{\begin{matrix}\Sigma_1 & \Sigma_2\\ \mathcal{D}_1 & \mathcal{D}_2\\ \alpha & \beta\end{matrix}}{\alpha \wedge \beta}\ \wedge I$$

Similarly, if we have a knowledge of $P \wedge Q$, then that should mean we have knowledge of both P and Q separately. For this reason, a proof of $P \wedge Q$ can be extended to either a proof of P or a proof of Q. This is codified in the following *elimination* rules of the conjunction connective.

Definition: Conjunction Elimination

If $\begin{smallmatrix}\Sigma\\ \alpha \wedge \beta\end{smallmatrix}\mathcal{D}$ is a proof of $\alpha \wedge \beta$ from hypotheses Σ, then we can extend this proof in two ways to obtain proofs of α and β respectively:

$$\dfrac{\begin{matrix}\Sigma\\ \mathcal{D}\\ \alpha \wedge \beta\end{matrix}}{\alpha}\ \wedge E_\ell \qquad \dfrac{\begin{matrix}\Sigma\\ \mathcal{D}\\ \alpha \wedge \beta\end{matrix}}{\beta}\ \wedge E_r$$

Example: Conjunction Commutative

Provide a natural deduction to prove the follow sequent:

$$\alpha \wedge \beta \vdash \beta \wedge \alpha$$

$$\dfrac{\dfrac{\alpha \wedge \beta}{\beta}\ \wedge E_r \quad \dfrac{\alpha \wedge \beta}{\alpha}\ \wedge E_l}{\beta \wedge \alpha}\ \wedge I$$

Example: Conjunction Associative

Provide a natural deduction to prove the following sequent:

$$(\alpha \wedge \beta) \wedge \gamma \ \vdash\ \alpha \wedge (\beta \wedge \gamma)$$

This proof requires unfolding the hypotheses to the left of the $\vdash$ to uncover the conclusion on the right. In the end both a proof of α and a proof of $\beta \wedge \gamma$ need to come from the hypothesis $(\alpha \wedge \beta) \wedge \gamma$. However, these can be dealt with separately and combined together at the end. First, then, it suffices to provide a deduction witnessing:

$$(\alpha \wedge \beta) \wedge \gamma \ \vdash\ \alpha$$

$$\dfrac{\dfrac{(\alpha \wedge \beta) \wedge \gamma}{\alpha \wedge \beta}\ \wedge E_\ell}{\alpha}\ \wedge E_\ell$$

Secondly we provide a deduction witnessing:

$$(\alpha \wedge \beta) \wedge \gamma \ \vdash \ \beta \wedge \gamma$$

$$\dfrac{\dfrac{\dfrac{(\alpha \wedge \beta) \wedge \gamma}{\alpha \wedge \beta} \wedge E_\ell}{\beta} \wedge E_r \qquad \dfrac{\dfrac{(\alpha \wedge \beta) \wedge \gamma}{\gamma} \wedge E_r}{\beta \wedge \gamma} \wedge I$$

Following the BHK these can be combined using $\wedge$ introduction to provide a proof of the original sequent

$$\dfrac{\dfrac{\dfrac{(\alpha \wedge \beta) \wedge \gamma}{\alpha \wedge \beta} \wedge E_\ell}{\alpha} \wedge E_\ell \qquad \dfrac{\dfrac{\dfrac{(\alpha \wedge \beta) \wedge \gamma}{\alpha \wedge \beta} \wedge E_\ell}{\beta} \wedge E_r \qquad \dfrac{(\alpha \wedge \beta) \wedge \gamma}{\gamma} \wedge E_r}{\beta \wedge \gamma} \wedge I}{\alpha \wedge (\beta \wedge \gamma)} \wedge I$$

Example: Currying

Provide a natural deduction to prove the following sequent:

$$(\alpha \wedge \beta) \to \gamma \ \vdash \ \alpha \to (\beta \to \gamma)$$

Since implication is the outermost connective in the conclusion, the antecedent of the conclusion can be temporarily added to the hypotheses. This reduces the sequent to:

$$(\alpha \wedge \beta) \to \gamma, \ \alpha^1 \ \vdash \ \beta \to \gamma$$

The outermost connective is still an implication, so we can use this idea again to reduce the sequent further:

$$(\alpha \wedge \beta) \to \gamma, \ \alpha^1, \ \beta^2 \ \vdash \ \gamma$$

The following deduction witnesses the new sequent.

$$\dfrac{\dfrac{\cancel{\alpha}^{\,1} \quad \cancel{\beta}^{\,2}}{\alpha \wedge \beta} \wedge I \qquad (\alpha \wedge \beta) \to \gamma}{\gamma} \text{ MP}$$

This sequent can be extended using implication introduction twice to get a deduction witnessing the original sequent.

$$\dfrac{\dfrac{\dfrac{\cancel{\alpha}^{\,1} \quad \cancel{\beta}^{\,2}}{\alpha \wedge \beta} \wedge I \qquad (\alpha \wedge \beta) \to \gamma}{\gamma} \text{ MP}}{\beta \to \gamma} \to I, 2}{\alpha \to (\beta \to \gamma)} \to I, 1$$

The technique of introducing and discharging temporary assumptions in the course of proving hypothetical statements like $P \to Q$ has come to be known as the *deduction theorem*. Since we are following the Gentzen calculus formulation of propositional logic it hardly seems worth calling a theorem, but in other (e.g. Hilbert style) formulations of logic, it really is a theorem worth proving. Nonetheless we will refer to this technique as the deduction theorem and give the explanation of it here.

Deduction Theorem: $\Sigma \vdash \alpha \to \beta$ if and only if $\Sigma \cup \{\alpha\} \vdash \beta$

Proof: If there is a deduction witnessing $\Sigma \vdash \alpha \to \beta$, then adding the assumption α will allow the deduction to be extended by modus ponens to a deduction of β i.e. a deduction witnessing the sequent $\Sigma \cup \{\alpha\} \vdash \beta$.

On the other hand, if there is a deduction witnessing $\Sigma \cup \{\alpha\} \vdash \beta$, implication introduction extends

this to a deduction witnessing $\Sigma \vdash \alpha \to \beta$.

2.3.3 Disjunction

Again we return to the BHK interpretation of the connectives to motivate the rules of inference for disjunction. Of disjunction the BHK says: to prove $P \vee Q$ we must provide either a proof of P or a proof of Q. Thus the introduction rules are straightforward, if we have a deduction witnessing $\Sigma \vdash P$, then it can be extended, via disjunction introduction, to a proof of $\Sigma \vdash P \vee Q$, for *any* proposition Q. In fact there are two introduction rules, depending on which side of the disjunct one places the proposition P.

Definition: Disjunction Introduction

If $\overset{\Sigma}{\alpha}\mathcal{D}$ is a derivation of α from Σ, then this proof can be extended in two ways to obtain proofs of $\alpha \vee \beta$ and $\beta \vee \alpha$ respectively:

$$\dfrac{\begin{array}{c}\Sigma \\ \mathcal{D} \\ \alpha\end{array}}{\alpha \vee \beta}\ \vee I_L \qquad\qquad \dfrac{\begin{array}{c}\Sigma \\ \mathcal{D} \\ \alpha\end{array}}{\beta \vee \alpha}\ \vee I_R$$

What of disjunction elimination? What can be deduced if $P \vee Q$ is a hypotheses? Given nothing more than the proposition $P \vee Q$, we need to account for the fact that a proof of it may have come by way of a proof of P, or a proof of Q. Therefore a case analysis is required in the elimination of the disjunction $P \vee Q$. Two extra proofs must be provided; one proof of $P \to R$ and another proof of $Q \to R$.

If we have the assumption $P \vee Q$ - that is, we know that either at least P has a proof or Q has a proof - together with proofs that $P \to R$ and $Q \to R$, then we may extend these three proofs to a proof of R. Notice that the proof of R depends on the hypotheses of *all three* branches. Conjunction introduction and modus ponens combined two proofs into one, now we have a rule of inference that combines three proofs into one.

Definition: Disjunction Elimination

If $\overset{\Sigma_1}{\alpha \vee \beta}\mathcal{D}_1$, $\overset{\Sigma_2}{\alpha \to \gamma}\mathcal{D}_2$, and $\overset{\Sigma_2}{\beta \to \gamma}\mathcal{D}_3$ are derivations, then together these can be extended to a proof $\overset{\Sigma}{\gamma}\mathcal{D}$, where $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3$. We depict this extension as follows:

$$\dfrac{\begin{array}{ccc}\Sigma_1 & \Sigma_2 & \Sigma_3 \\ \mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 \\ \alpha \vee \beta & \alpha \to \gamma & \beta \to \gamma\end{array}}{\gamma}\ \vee_E$$

Note! Since disjunction elimination requires proofs that either disjunct lead to the same outcome, each disjunct can be temporarily added to the hypotheses in order to prove these implications.

Example: Disjunction Commutative

Provide a natural deduction to witness the following sequent:

$$\alpha \vee \beta \ \vdash\ \beta \vee \alpha$$

As is now custom, we first look to rewrite the sequent with temporary hypotheses. In this case, we see that $\alpha \vee \beta$ is a hypothesis and therefore we will need to use disjunction elimination. Two branches of this step will be proofs of implications with α and β as antecedent. For this reason we temporarily add these to our hypotheses to reduce the original problem to the following sequent:

$$\alpha \vee \beta, \alpha^1, \beta^2 \ \vdash\ \beta \vee \alpha$$

In fact we can reduce the sequent further into *two* sequents:

$$\alpha \vdash \beta \vee \alpha \qquad\qquad \beta \vdash \beta \vee \alpha$$

We have reduced the original problem to providing separate proofs to witness these sequents. These can be achieved using $\vee$ introduction.

$$\dfrac{\dfrac{\overline{\alpha}^{\,1}}{\beta \vee \alpha}\vee I_R}{\alpha \to (\beta \vee \alpha)} \to I,1 \qquad \dfrac{\dfrac{\overline{\beta}^{\,2}}{\beta \vee \alpha}\vee I_L}{\beta \to (\beta \vee \alpha)} \to I,2$$

Together they combine with the assumption $\alpha \vee \beta$ to get a deduction witnessing the original sequent.

$$\dfrac{\alpha \vee \beta \quad \dfrac{\dfrac{\overline{\not{\alpha}}^{\,1}}{\beta \vee \alpha}\vee I_R}{\alpha \to (\beta \vee \alpha)} \to I,1 \quad \dfrac{\dfrac{\overline{\not{\beta}}^{\,2}}{\beta \vee \alpha}\vee I_L}{\beta \to (\beta \vee \alpha)} \to I,2}{\beta \vee \alpha} \vee E$$

Example: Disjunction Associative

Provide a natural deduction to witness the following sequent:

$$(\alpha \vee \beta) \vee \gamma \ \vdash \ \alpha \vee (\beta \vee \gamma)$$

2.3.4 Negation

Recall the definition of $\neg\alpha := \alpha \to \bot$, where $\bot$ denotes the proposition *absurdity*. We think of $\neg\alpha$ as being the proposition that α leads to absurdity. Therefore, in order to prove the negation of a proposition, one has to prove the corresponding implication. This means there are no new rules of inference for the negation connective. Negation introduction is a special case of implication introduction, whereas negation elimination is a special case of modus ponens.

Example: Modus Tollens

Provide a natural deduction to prove the following sequent:

$$\alpha \to \beta,\ \neg\beta \ \vdash\ \neg\alpha$$

Notice that this sequent is equivalent to

$$\alpha \to \beta,\ \neg\beta \ \vdash\ \alpha \to \bot$$

When it is rewritten like this it is clearer that the deduction theorem can help us reduce the sequent to the following by adding α as a temporary hypothesis:

$$\alpha \to \beta,\ \neg\beta,\ \alpha^1 \vdash\ \bot$$

The following deduction witnesses the new sequent. Notice that $\neg\beta$ is an implication and therefore modus ponens is the correct elimination rule to use on it.

$$\dfrac{\dfrac{\overline{\alpha}^{\,1} \quad \alpha \to \beta}{\beta}\,\text{MP} \qquad \neg\beta}{\bot}\,\text{MP}$$

This deduction can be extended by implication introduction to witness the original sequent. Again, there is no special negation introduction rule. We just use implication introduction.

$$\dfrac{\dfrac{\dfrac{\overline{\alpha}^{\,1} \quad \alpha \to \beta}{\beta}\,\text{MP} \qquad \neg\beta}{\bot}\,\text{MP}}{\neg\alpha}\,\to I, 1$$

Example: Contraposition

Provide a natural deduction to prove the following sequent:

$$\alpha \to \beta \ \vdash\ \neg\beta \to \neg\alpha$$

The deduction rule can be applied twice, remembering to desugar the negations into implications. This reduces the sequent to the following:

$$\alpha \to \beta,\ \neg\beta^1,\ \alpha^2 \vdash\ \bot$$

The following deduction witnesses the reduced sequent:

$$\dfrac{\dfrac{\alpha \to \beta \quad \overline{\alpha}^{\,2}}{\beta}\,\text{MP} \qquad \dfrac{}{\neg\beta}\,1}{\bot}\,\text{MP}$$

This sequent can be extended using implication introduction to obtain a deduction witnessing the original sequent.

$$\dfrac{\dfrac{\dfrac{\alpha \to \beta \quad \overline{\alpha}^{\,2}}{\beta}\,\text{MP} \qquad \dfrac{}{\neg\beta}\,1}{\bot}\,\text{MP}}{\neg\alpha}\,\to I, 2}{\neg\beta \to \neg\alpha}\,\to I, 1$$

Proof by Refutation

We have defined the process of proving a negative statement $\neg\alpha$ to be the process of showing absurdity $\perp$ follows from the assumption of α. Put another way, to show something is *not the case* is to show that it leads to a contradiction. This method of proof has come to be known as *proof by refutation* or simply *refutation*. Here is an example of such a proof.

Example: $\sqrt{2}$ is irrational

Proof: This is a claim that something is not the case i.e. a negative claim. In particular is the claim that $\sqrt{2}$ is *not* rational. In order to prove this we suppose that $\sqrt{2}$ is rational. Thus, there are a, b such that $\sqrt{2} = \frac{a}{b}$ where a, b have no common factors - for they can be divided out to form a simpler fraction.
This implies there exist coprime $a, b \in \mathbb{N}$ such that $a^2 = 2b^2$. It follows that $2|a$ and hence $a = 2k$. Furthermore, we see $4k^2 = 2b^2$ and hence $b^2 = 2k^2$ and thus $2|b$. However, this contradicts the assumption that a, b are coprime. Therefore, no such a, b can exist.

This proof is often miscontrued as a *proof by contradiction*. The distinction between the two methods will be clarified later in these notes.

Minimal Logic

This page has each rule of inference of minimal logic. Use this when working on tutorials and when you are studying. What this page does not explicitly contain is any mention of the hypotheses that the conclusions depend on. Make sure you know which hypotheses the conclusion depends on when using any given rule of inference.

$$\frac{\begin{array}{cc}\Sigma_1 & \Sigma_2\\ \mathcal{D}_1 & \mathcal{D}_2\\ \alpha & \beta\end{array}}{\alpha \wedge \beta}\ \wedge I$$

Conjuction Introduction

$$\frac{\begin{array}{c}\Sigma\\ \mathcal{D}\\ \alpha \wedge \beta\end{array}}{\alpha}\ \wedge E_L$$

Conjuction Elimination L(eft)

$$\frac{\begin{array}{c}\Sigma\\ \mathcal{D}\\ \alpha \wedge \beta\end{array}}{\beta}\ \wedge E_R$$

Conjuction Elimination R(ight)

$$\frac{\begin{array}{c}\overline{\not{\alpha}}\\ \Sigma\\ \mathcal{D}\\ \beta\end{array}}{\alpha \to \beta}\ \to I$$

Implication Introduction

$$\frac{\begin{array}{cc}\Sigma_1 & \Sigma_2\\ \mathcal{D}_1 & \mathcal{D}_2\\ \alpha \to \beta & \alpha\end{array}}{\beta}\ \text{MP}$$

Implication Elimination (Modus Ponens)

2.3.5 Ex Falso Quodlibet

2.3.6 Reductio ad Absurdum

Previously we gave a proof witnessing the sequent

$$\alpha \to \beta \ \vdash \ \neg\beta \to \neg\alpha$$

Whenever there is a sequent with just one hypothesis and one conclsuion, the converse sequent should be considered. Is it possible to provide a deduction witnessing the converse

$$\neg\beta \to \neg\alpha \ \vdash \ \alpha \to \beta$$

That is to say, are these two implications logically equivalent? Let's see how much ground we can make on the proof.

Example: Not-Not is What?

Provide a natural deduction to prove the following sequent:

$$\neg\beta \to \neg\alpha \ \vdash \ \alpha \to \beta$$

Following the ideas we have already developed we add in temporary hypotheses according to the hypotheses and conclusion. This suggests that it is sufficient to give a proof witnessing the sequent:

$$\neg\beta \to \neg\alpha, \alpha^1, \neg\beta^2 \ \vdash \ \beta$$

The α is added in order to prove the hypothetical $\alpha \to \beta$ and $\neg\beta$ is added in order to make use of the hypothetical $\neg\beta \to \neg\alpha$.

$$\dfrac{\dfrac{\dfrac{\neg\beta \to \neg\alpha \quad \overline{\neg\beta}\ 2}{\neg\alpha}\ \text{MP} \quad \overline{\alpha}\ 1}{\bot}\ \text{MP}}{\neg\neg\beta}\ \to I, 2}{\alpha \to \neg\neg\beta}\ \to I, 1$$

Are we done? Does this derivation provide a proof of the original sequent? It all turns on whether $\neg\neg\beta$ implies β. That is to say, whether there is a deduction witnessing the following sequent:

$$\neg\neg\beta \ \vdash \ \beta$$

In fact, it can be shown (with tools[1] that we won't develop here) that the rules of inference so far introduced are *not sufficient* to prove this! The deduction provided above is the closest we can get to an intuitionistic proof of this sequent. Knowledge of $\neg\neg\beta$, interpreted by the BHK, is an algorithm for turning (a proof of) $\neg\beta$ into $\bot$ absurdity - this says nothing of it being a proof of β.

Intuitionistic logic must be extended in order to provide a proof of this sequent. Here is an example of a proof using this mode of reasoning, called *proof by contradiction* or *double negation elimination*.

Example: Well, they can't not exist!

Consider the following claim:

There exist irrational numbers x, y such that x^y is rational.

Proof: Suppose there are no such numbers. Since $\sqrt{2}$ is irrational, this implies $\sqrt{2}^{\sqrt{2}}$ is irrational. Therefore, both $x = \sqrt{2}$ and $y = \sqrt{2}^{\sqrt{2}}$ are irrational. However, it follows that $x^y = 2$ is rational. Contradicting the original claim that there are not such x, y. Therefore, there must be some irrational x, y such that x^y is rational.

This mode of reasoning is formalised into a rule of inference as follows.

[1] intuitionistic semantics

Definition: Reductio Ad Absurdum

If $\overset{\Sigma}{\perp}\mathcal{D}$ is a deduction of $\perp$ from Σ, then

$$\dfrac{\begin{array}{c}\Sigma, \cancel{\neg\alpha} \\ \mathcal{D} \\ \perp\end{array}}{\alpha}\ \text{RAA}$$

is a derivation of α from the assumptions $\Sigma\backslash\{\neg\alpha\}$.

It is distinguished from the intuitionistic ex falso quodlibet by allowing for the discharge of the $\neg\alpha$ assumption. It is this freedom to discharge assumptions that allows for the proof of theorems like double negation elimination and the logical equivalence of contraposition. Note that the discharged assumption must be the negation of the conclusion.

Example: Double Negation Elimination

Provide a natural deduction to prove the following sequent:

$$\neg\neg\alpha \vdash \alpha$$

$$\dfrac{\dfrac{\neg\neg\alpha \quad \overline{\neg\alpha}^{\,1}}{\perp}\ \text{MP}}{\alpha}\ \text{RAA}, 1$$

RAA allows for the completion of the proof of the equivalence of an implication with its contraposition:

Example: Contraposition

Provide a natural deduction to prove the following sequent:

$$\neg\beta \to \neg\alpha \vdash \alpha \to \beta$$

Following the ideas we have already developed we add in temporary hypotheses according to the hypotheses and conclusion. This suggests that it is sufficient to give a proof witnessing the sequent:

$$\neg\beta \to \neg\alpha, \alpha^1, \neg\beta^2 \vdash \beta$$

The α is added in order to prove the hypothetical $\alpha \to \beta$ and $\neg\beta$ is added in order to make use of the hypothetical $\neg\beta \to \neg\alpha$.

$$\dfrac{\dfrac{\dfrac{\neg\beta \to \neg\alpha \quad \overline{\neg\beta}^{\,2}}{\neg\alpha}\ \text{MP} \quad \overline{\alpha}^{\,1}}{\perp}\ \text{MP}}{\beta}\ \text{RAA}, 2}{\alpha \to \beta} \to I, 1$$

Example: Tis or Tisn't

Provide a natural deduction to prove the following sequent: $\vdash \alpha \vee \neg\alpha$

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\cancel{\alpha}}^{\,1}}{\alpha \vee \neg\alpha} \quad \dfrac{}{\cancel{\neg(\alpha \vee \neg\alpha)}}^{\,2}}{\perp}\ MP}{\neg\alpha} \to I, 1}{\alpha \vee \neg\alpha} \vee I \quad \dfrac{}{\cancel{\neg(\alpha \vee \neg\alpha)}}^{\,2}\ MP}{\perp}\ RAA, 2}{\alpha \vee \neg\alpha}$$

The logic obtained by adding in RAA as a rule of inference is known as *classical* logic. It is the logic followed by most mathematicians, although there are some dissenters and there are many famous names

among their ranks!

Although we have presented the bridging from intuitionistic, or minimal, logic to classical logic as adding in the rule of inference RAA, there are other modes of reasoning that determine equivalent logics; that is, rules of inference that agree on which formulae are theorems. Rather than assuming the rule of inference RAA, one might instead assume *the law of excluded middle* $P \vee \neg P$ for each proposition P. Below is the same claim about the existence of certain irrational numbers, but with the proof presented in a slightly different way.

Example: But which is it?

Consider the following claim:
There exists irrational numbers x, y such that x^y is rational.

Proof: Since $\sqrt{2}$ is irrational we consider the number $\sqrt{2}^{\sqrt{2}}$. This number is either rational or irrational.
If $\sqrt{2}^{\sqrt{2}}$ is rational, then the claim is true for $x = y = \sqrt{2}$ are an example of a pair with the claimed property.
If the number is irrational, then we compute

$$(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2$$

Certainly 2 is rational, therefore $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$ are an example of such irrational numbers.
In either case, there must exist irrational x, y such that x^y

There are theorems that are provable with the introduction of RAA that are not provable without it. These theorems can be proved by using the rules of inference derived so far: Intuionistic logic + RAA. However, since we already have a proof of LEM

$$\vdash \ P \vee \neg P$$

which does not rely on any hypotheses, we can also make use of that in our proofs. Indeed, this leads to more readable proofs than trying to do everything from RAA each time.

Example: Material Implication

The following sequent is not a theorem of intuitionistic logic

$$P \to Q \ \vdash \ \neg P \vee Q$$

However it this is provable with the introduction of classical logic. As we know that LEM is a theorem, we are free to use this fact in our proof. When we make use of the LEM we have to make a choice about which proposition α to use. For LEM is really infinitely many theorems of the form $\vdash \ \alpha \vee \neg\alpha$ one for each proposition α. Since the consequent is made up of two atomic propositions P and Q it seems that we should choose one of these. Indeed either will work. The proof using Q is presented here and the reader should provide the analogous proof using P.
Once we have chosen to use the theorem $\vdash \ Q \vee \neg Q$, we know there will be an $\vee$-elimination step in the proof. By now we know that $\vee$-elimination proofs have two implication subproofs. Altogether this means classical proofs using $Q \vee \neg Q$ ultimately come down to proving two implications; in this case we need proofs of $P \to Q \ \vdash Q \to (\neg P \vee Q)$ and $P \to Q \ \vdash \neg Q \to (\neg P \vee Q)$. These subproofs are in the second and third branch of the $\vee$-elimination below.

$$\dfrac{Q \vee \neg Q \quad \dfrac{\dfrac{\overline{Q}^{\,1}}{\neg P \vee Q}\vee I}{Q \to (\neg P \vee Q)} \to I, 1 \quad \dfrac{\dfrac{\dfrac{P \to Q \quad \overline{\neg Q}^{\,2}}{\neg P}\text{MT}}{\neg P \vee Q}\vee I}{\neg Q \to (\neg P \vee Q)} \to I, 2}{\neg P \vee Q} \vee E$$

The right-hand branch the proof was shortened using a *derived rule of inference*. We have already shown $\neg P$ follows from the hypotheses $P \to Q$ and $\neg Q$ - we call this modus tollens. We need not reproduce the same proof each time we are in this situation, we can simply state that we are using MT at that line in the proof. There will be more about derived rules later.

Step-by-Step Classical Proofs

Once it is decided that classical logic is required to prove the sequent $\Sigma \vdash \gamma$ and that LEM with proposition α will be used in the proof, the rest of the proof can be broken down into the following steps:

1. $\Sigma,\ \alpha^1 \vdash \gamma$,
2. $\Sigma,\ \neg\alpha^2 \vdash \gamma$,
3. Implication introductions on previous two proofs, and
4. $\vee$-elimination with $\alpha \vee \neg\alpha$.

Example: Pierce's Law

Provide a natural deduction to prove the following sequent:

$$\vdash\ ((P \to Q) \to P) \to P$$

This proof requires classical logic. However, it's best to make use of the minimal logic steps first, before moving onto use RAA or another classical mode of reasoning. This is a proof hypothetical of a hypothetical, so let's first assume the antecedent:

$$(P \to Q) \to P \vdash P$$

Now, since P dominates the statement, it seems a sensible choice to make use of the $P \vee \neg P$ instance of LEM. Following the steps outlined above we break this proof into the following steps:

1. $(P \to Q) \to P,\ P^1 \vdash P$,
2. $(P \to Q) \to P,\ \neg P^2 \vdash P$,
3. Implication introductions on previous two proofs, and
4. $\vee$-elimination with $P \vee \neg P$.

The first step is trivial in this case. We assume P, which is precisely the proposition we want to prove. To prove the next hypothetical we need to make use of the new hypothesis $\neg P$. The only way to do that is to assume P; this seems odd as it is the conclusion we are trying to draw! But we can discharge it later.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{P}^{\,3} \quad \overline{\neg P}^{\,2}}{\bot}\,\mathsf{MP}}{Q}\,\mathsf{XF}}{P \to Q} \to I,3 \qquad (P \to Q) \to P}{\dfrac{P}{\neg P \to P} \to I,2}\,\mathsf{MP}$$

Altogether the proof of Pierce's Law using LEM is:

$$\dfrac{\dfrac{P \vee \neg P \qquad \dfrac{\dfrac{\dfrac{\dfrac{\overline{P}^{\,3} \quad \overline{\neg P}^{\,2}}{\bot}\,\mathsf{MP}}{Q}\,\mathsf{XF}}{P \to Q} \to I,3 \qquad (P \to Q) \to P}{\dfrac{P}{\neg P \to P} \to I,2} \qquad P \to P}{P}\,\vee E}{((P \to Q) \to P) \to P} \to I,1$$

Observation: Notice that in the last two examples every step in the proofs are intuitionistic except for the assumption of LEM.

Derived Rules

Just as a programmer can refactor their code into functions/subroutines to make their code cleaner, so can we employ theorems already proved to make proofs shorter, cleaner, and easier to comprehend. Mathematicians do this in their work by organising results into lemma, theorems, and corollaries. We can make allowances for this in our notation by annotating the line in the proof with a name of a theorem; something to acknowledge that we are using a previously proven result and are not reproducing the proof.

Example: $\wedge$Comm and $\wedge$Assoc

Provide a natural deduction to prove the following sequent:

$$(\alpha \wedge \beta) \wedge \gamma \vdash \beta \wedge (\alpha \wedge \gamma)$$

In this proof we make use of previously proven commutativity and associativity of the conjunction connective. We could reproduce those proofs here, but we need only reference them with the labels $\wedge_{\text{comm}}$ and $\wedge_{\text{assoc}}$ respectively.

$$\dfrac{\dfrac{\dfrac{\dfrac{(\alpha \wedge \beta) \wedge \gamma}{\alpha \wedge \beta}\wedge E_l}{\beta \wedge \alpha}\wedge_{\text{comm}} \qquad \dfrac{(\alpha \wedge \beta) \wedge \gamma}{\gamma}\wedge E_r}{(\beta \wedge \alpha) \wedge \gamma}\wedge I}{\beta \wedge (\alpha \wedge \gamma)}\wedge_{\text{assoc}}$$

These references to previous results will be referred to as *derived rules of inference*. If the result to be called doesn't have an obvious name, then use the label THM to denote that a theorem is being called to justify that line in the proof.

Tips and Tricks

The list below contains a number of things to keep in mind when writing a natural deduction. It should be referenced if one is stuck for ideas!

- Connectives in the **hypotheses** are **eliminated**.
- Connectives in the conclusion are introduced.
- Rewrite the sequence with all temporary hypotheses added:
  - When introducing $\alpha \to \beta$, one should temporarily add the antecedent α to the hypotheses.
  - When eliminating $\alpha \vee \beta$, one should temporarily add each disjunct α, β to the set of hypotheses.
- Don't write the whole proof at once; use subproofs!
  - When using $\vee$ elimination, write each of the three branches separately. Bring them together at the end.
- Working backwards can be helpful. Particularly in an $\vee$ elimination step of a deduction.
- When writing proofs depending on classical logic, rather than relying on RAA directly, make use of LEM instead.

2.4 First Order Logic

2.4.1 Predicates

2.4.2 Quantifiers

2.4.3 First Order Languages

Chapter 3

Formal Mathematics

3.1 First Order Theories

3.2 Arithmetic

Chapter 4

Computation

4.1 Brouwer-Heyting-Kolmogorov

4.2 Untyped Lambda Calculus

Alpha Reduction

Beta Reduction

Example:

Compute the normal form of this λ-term by computing β-reductions.

$$\begin{aligned}(\lambda x.\ x)(\lambda y.\ y\ (\lambda z.\ z\ w)) &= (\lambda x.\ x)(\lambda y.\ y\ (\lambda z.\ z\ w)) \\ &=_\beta \lambda y.\ y\ (\lambda z.\ z\ w)\end{aligned}$$

Example:

Compute the normal form of this λ-term by computing β-reductions.

$$\begin{aligned}((\lambda x.\ \lambda y.\ y\ x)\ f)\ g &= ((\lambda x.\ \lambda y.\ y\ x)\ f)\ g \\ &=_\beta (\lambda y.\ y\ f)\ g \\ &=_\beta g\ f\end{aligned}$$

Example:

Some λ-terms do not have normal forms.

$$\begin{aligned}(\lambda x.\ x\ x\ x)(\lambda x.\ x\ x\ x) &= (\lambda x.\ x\ x\ x)(\lambda x.\ x\ x\ x) \\ &=_\beta (\lambda x.\ x\ x\ x)(\lambda x.\ x\ x\ x)(\lambda x.\ x\ x\ x) \\ &\vdots\end{aligned}$$

After each β-reduction an extra copy of $(\lambda x.\ x\ x\ x)$ will be added. Thus this term gets longer and longer and never settles to a normal form.

Reduction Strategies

If there are more than one β-redex in a single λ-term, then one has to make a choice about which to compute first. Since there are choices being made, one must also consider how the computation will evolve according to that choice. We will consider two different reduction strategies; call-by-name and call-by-value. To illustrate each of these methods let's consider the following λ-term:

$$(\lambda y.\ \lambda z.\ z)((\lambda x.\ x\ x)(\lambda x.\ x\ x))$$

This λ-term has two β redex indicated by the following colourings:

$$(\lambda y.\ \lambda z.\ z)((\lambda x.\ x\ x)(\lambda x.\ x\ x))$$

This redex has the left most abstraction-argument pair. However, the argument of this redex is itself a β-redex indicated by the next colouring:

$$(\lambda y.\ \lambda z.\ z)((\lambda x.\ x\ x)(\lambda x.\ x\ x))$$

We must make a choice: do we compute the outer redex first? Or, do we compute the redex in the argument first before passing it to the leftmost abstraction? Does it matter? Or will they always lead to the same thing. The first colouring, where we pass the unevaluated argument, is known as call-by-name (or lazy) evaluation:

$$(\lambda y.\ \lambda z.\ z)((\lambda x.\ x\ x)(\lambda x.\ x\ x)) =_\beta \lambda z.\ z$$

Following this has reduced the λ-term to normal form. If we instead compute the β-reduction on the argument first, then we are lead into an infinite loop and the λ-term never gets to a normal form. This choice of evaluation strategy does have consequences. If a λ-term has a normal form, then lazy evaluation will compute it. For this reason, if not otherwise stated, we will assume all reductions are done using call-by-name; lazy is the way!

Eta Reduction

4.3 Simply Typed Lambda Calculus

We have seen that λ-terms may not have a normal form. For example, we may search for a solution to a predicate that has no solution «ENTER EXAMPLE HERE» Reductions of λ-terms may not act predictably, or nomralise, if we don't take care to use them as intended. Consider the λ-term:

$$\lambda f.\ \lambda g.\ \text{COND (EQUAL? } f\ g) \text{ (SUCC } f) \text{ (SUCC } g)$$

This will reduce predictably if Church numerals are passed to f and g. However, if we aren't careful, then in the course of β-reduction arbitrary λ-terms could get passed to f and g. There is no telling what the program will do then. In order to limit these problems types were introduced to the λ-calculus by Alonso Church and Haskell Curry. Variables and data were assigned types and functions were declared with their domain (input) and codomain (output) types.

Types should be familiar to anyone with a little bit of programming experience. Python, for example, has a number of built in types including: int, bool, and float. Type-errors will also be familiar! These are the messages we get from a compiler or interpreter when the program we have written does not obey the languages type system. We have written a program for which the types do not compose properly! We have tried to concatenate an integer, compose floats, or some other such nonsense. Instead of running the program, which could act unpredictably, the compiler just says: "No" and, if you're lucky, tells you what the type error is. These features of programming languages are conceptual descendants of the ideas of Church and Curry. Mathematics students will also be familiar with the idea, if not the word type. When functions are explained in mathematics classes, great pains are taken to stress the importance of domain and codomain. If you get that wrong, then the function maybe undefined at some input passed to it. For example, you might compute an integral over an interval for which the integrand is undefined! Mathematics, too, abounds with different types: functions, vectors, matrices, natural numbers etc.

In this chapter we are going to study the interaction between different types, rather than study in detail any particular type. Later, in the chapter on Lean, we will consider specific types like the type of Natural numbers. Types are tags on data and programs to ensure they are used in the intended manner. First we will describe some language for talking about types and then introduce a number of ways of building types up from base types: the function type and two common algebraic data types - the product type (tuples) and the sum type (disjoint, or tagged, unions) - are studied. Since the data that we are computing with has types, we now need to take more care when writing λ-terms. We develop a calculus which we will refer to as the *typing deduction calculus* to ensure our λ-terms are *well-typed*. This calculus will involve a type constructor and type destructor for each new type.

Type Contexts

Throughout this chapter let us assume we are given a collection of atomic base types denoted by capital English letters:

$$P\ Q\ R\ S\ T\ \ldots : \text{Type}$$

If it helps, then these abstract types can be thought of as Nat, Bool, or Float. But really they are just names that could be swapped out for concrete types when ever we want. We are treating them more like type-variables, but we will just call them types. Using these types, and the compound data types that we will soon construct, we can begin to assign types to variables and programs. Recall that λ-terms are either a variable, or constructed from component λ-terms by either abstraction or application. For the rest of this chapter we will discuss how to consistently assign types to λ-terms.

All of our λ-terms are going to be written with respect to a set Σ of typing declarations that we will call a *typing context*, or simply a *context*. For example we might declare the following typing context:

$$\Sigma = \{x : P,\quad f : P \to Q,\quad g : (P \to Q) \to R\}$$

In this context we have a variable x assigned type P and two functions f and g assigned types $P \to Q$ and $(P \to Q) \to R$ respectively. Terms inside the typing context can be thought of as global variables available to write our programs. Much like an import statement at the top of a program in a modern programming language.

The problem that will be of primary interest to us is the problem of determining whether a term of a particular type can be derived from a given context. If Σ is a typing context and γ some type, then the following sequent is a claim that in the context Σ the γ is inhabited:

$$\Sigma \vdash \gamma$$

In this context we will refer to γ as the goal; it is the type that we are writing a λ-term to inhabit. It is the programmers job to calculate such a term to validate the claim made by the sequent.

Example:

Consider the type context

$$\Sigma = \{x : P,\quad f : P \to Q,\quad g : (P \to Q) \to R\}$$

Although we have not yet officially defined types like $P \to Q$, they are function types. They take in a term of the type to the left of $\to$ and return a term to the right of the $\to$ symbol. For which types can terms be computed in this context?
Certainly any term in the context should be deducible from the context. Therefore the following sequents are valid:

$\Sigma \vdash P$
$\Sigma \vdash P \to Q$
$\Sigma \vdash (P \to Q) \to R$

We can formalise this with the following typing rule for variables.
Variable Declaration
If $x : \alpha$ is a type declaration in a type context Σ, then we can always call that term in a program. That is to say, the following sequent is valid $\Sigma,\ x : \alpha \vdash \alpha$. We depict this in the following way:

$$\frac{}{x : \alpha}\ \text{var}$$

Given λ-terms of the appropriate types, we should be able to compose them to get terms of other types. Using this idea are there any other types for which terms can be formed in the context Σ? In the next section we formally introduce the $\to$ function type and define the typing rules for application and abstraction.

Function Types

To ensure that application of a λ-term is only applied to the correct type of variable, we define the following function type. If P and Q are types, then we denote by $P \to Q$ the type of functions that take in terms of type P and reduce to a term of type Q. We close the collection Type under this operation. That is, if $P\ Q$: Type, then $P \to Q$: Type.

Example: Function Types

If $P\ Q\ R\ S$: Type, then the following are examples of types that can be constructed.

$P \to R$
$R \to S$
$(P \to R) \to (P \to S)$
$P \to (Q \to S)$

We use the same binding convention for function types as for implication in propositional logic.

The function type constructor says that if a λ-term depends on a free variable $a : \alpha$, then this can be abstracted over to make a function which takes in a term of type α. Compare this to the process of taking a global variable in a program and factoring it out to write a routine which now takes that variable as input.

Definition: Function Type Constructor (Abstraction)

If ${}^{\Sigma}_{t:\beta}\mathcal{D}$ is a typing derivation of a term of $t : \beta$ from a context Σ containing an arbitrary term $a : \alpha$, then $\mathcal{D}$ can be extended to a typing derivation of a term $f : \alpha \to \beta$ from the context $\Sigma \backslash \{a : \alpha\}$, which we depict as follows:

$$\frac{\begin{array}{c}\Sigma,\ \cancel{a:\alpha} \\ \mathcal{D} \\ t : \beta\end{array}}{\lambda\ x : \alpha.\ t[a/x]\ :\ \alpha \to \beta}\ \lambda I$$

Where $t[a/x]$ denotes the term t with each free instance of a replaced by x.

Well-typed functions should only be passed terms of their domain type. We introduce the following function type destructor to our type derivation calculus to make sure this happens. This is function application with extra care to check the type of the input being passed to the function.

Definition: Function Type Destructor (Application)

If ${}^{\Sigma_1}_{f:\alpha\to\beta}\mathcal{D}_1$ and ${}^{\Sigma_2}_{a:\alpha}\mathcal{D}_2$ are typing derivations, then they can be extended to a derivation ${}^{\Sigma_1\cup\Sigma_2}_{(f\ a):\beta}\mathcal{D}$ which we depict as follows:

$$\frac{\begin{array}{cc}\Sigma_1 & \Sigma_2 \\ \mathcal{D}_1 & \mathcal{D}_2 \\ f : \alpha \to \beta & a : \alpha\end{array}}{(f\ a) : \beta}\ \text{app}$$

Example: Implication Composition

Provide a typing derivation to prove the following sequent:

$$\alpha \to \beta,\ \beta \to \gamma \ \vdash\ \alpha \to \gamma$$

$$\dfrac{\dfrac{\dfrac{\overline{\cancel{\alpha}}^{\,1} \quad \alpha \to \beta}{\beta}\ \text{MP} \qquad \beta \to \gamma}{\gamma}\ \text{MP}}{\alpha \to \gamma}\ \to I, 1$$

Example: I Combinator

Example: S Combinator

Example: K Combinator

Seeing this example again, now from the point-of-view of type theory, with an explicit term of type P, may help make it clearer.

$$P \ \vdash\ Q \to P$$

Product Types

Tuples are a data type common to both programmers and mathematicians. Programmers will often have the ability to utilise (nested) pairs to structure data. In mathematics we form the Cartesian product of two (or more) sets all the time. Our next data type will formalise this idea of forming a tuple of terms of (possibly) different types. We further close Type under the type forming operation $\times$ called the product type. That is, if $\alpha\ \beta$: Type, then $\alpha \times \beta$: Type.

Example: Product Types

Given the atomic base types

$$P\ Q\ R\ S\ T\ \ldots : \text{Type}$$

Together with the two constructors $\to$ and $\times$ we can form types with $\to$ and $\times$ nested arbitrarily

$$P \to (Q \times P) \qquad (P \to Q) \times P$$
$$(P \times Q) \times R \qquad ((P \times Q) \to R) \to (P \to (Q \to R))$$

Construction rules tell us how to form terms of a particular type. That is to say, they tell us how to write programs *into* that type. In order to form a tuple of type $\alpha \times \beta$ is necessary and sufficient to provide (typing derivations of) terms $a : \alpha$ and $b : \beta$. This is formalised as the following type constructor.

Definition: Product Type Constructor

If ${}^{\Sigma_1}_{a:\alpha}\mathcal{D}_1$ and ${}^{\Sigma_2}_{b:\beta}\mathcal{D}_2$ are typing derivations, then they can be extended to a deduction ${}^{\Sigma_1 \cup \Sigma_2}_{(a,\ b):\alpha\times\beta}\mathcal{D}$, which we depict as follows:

$$\frac{\begin{array}{cc}\Sigma_1 & \Sigma_2\\ \mathcal{D}_1 & \mathcal{D}_2\\ a : \alpha & b : \beta\end{array}}{(a, b) : \alpha \times \beta}\ \times$$

Constructors tell us how to write programs into the types they are constructing, whereas destructors tell us how to write programs out of the type they are destroying. There are two destructors for the product type, one for each component of the product. Elements of the product type are pairs and we can therefore project onto the FirST component or the SecoND component of the tuple. This is formalised in the following type destructor.

Definition: Product Type Destructors

If ${}^{\Sigma}_{\alpha\times\beta}\mathcal{D}$ is a typing derivation of $\alpha \times \beta$ in a typing context Σ, then we can extend this derivation in two ways to obtain terms of type α and β respectively:

$$\frac{\begin{array}{c}\Sigma\\ \mathcal{D}\\ t : \alpha \times \beta\end{array}}{\mathsf{fst}\ t : \alpha}\ \mathsf{fst} \qquad \frac{\begin{array}{c}\Sigma\\ \mathcal{D}\\ t : \alpha \times \beta\end{array}}{\mathsf{snd}\ t : \beta}\ \mathsf{snd}$$

With these new terms we have to say how computations work i.e. what are the β-reduction rules for fst and snd? If the structure of the term $t : \alpha \times \beta$ is known to be (e.g.) $t = (a, b)$ for $a : \alpha$ and $b : \beta$, then the following terms are considered β-redex and reduce according to the equations:

$$\mathsf{fst}\ (a, b) =_\beta a$$
$$\mathsf{snd}\ (a, b) =_\beta b$$

Example: Product Commutative

Provide a natural deduction to prove the follow sequent:

$$\alpha \wedge \beta \vdash \beta \wedge \alpha$$

$$\dfrac{\dfrac{\alpha \wedge \beta}{\beta}\wedge E_r \quad \dfrac{\alpha \wedge \beta}{\alpha}\wedge E_l}{\beta \wedge \alpha}\wedge I$$

Example: Product Associative

Provide a natural deduction to prove the following sequent:

$$(\alpha \wedge \beta) \wedge \gamma \ \vdash \ \alpha \wedge (\beta \wedge \gamma)$$

This proof requires unfolding the hypotheses to the left of the $\vdash$ to uncover the conclusion on the right. In the end both a proof of α and a proof of $\beta \wedge \gamma$ need to come from the hypothesis $(\alpha \wedge \beta) \wedge \gamma$. However, these can be dealt with separately and combined together at the end. First, then, it suffices to provide a deduction witnessing:

$$(\alpha \wedge \beta) \wedge \gamma \ \vdash \ \alpha$$

$$\dfrac{\dfrac{(\alpha \wedge \beta) \wedge \gamma}{\alpha \wedge \beta}\wedge E_\ell}{\alpha}\wedge E_\ell$$

Secondly we provide a deduction witnessing:

$$(\alpha \wedge \beta) \wedge \gamma \ \vdash \ \beta \wedge \gamma$$

$$\dfrac{\dfrac{\dfrac{(\alpha \wedge \beta) \wedge \gamma}{\alpha \wedge \beta}\wedge E_\ell}{\beta}\wedge E_r \quad \dfrac{(\alpha \wedge \beta) \wedge \gamma}{\gamma}\wedge E_r}{\beta \wedge \gamma}\wedge I$$

Following the BHK these can be combined using $\wedge$ introduction to provide a proof of the original sequent

$$\dfrac{\dfrac{\dfrac{(\alpha \wedge \beta) \wedge \gamma}{\alpha \wedge \beta}\wedge E_\ell}{\alpha}\wedge E_\ell \quad \dfrac{\dfrac{\dfrac{(\alpha \wedge \beta) \wedge \gamma}{\alpha \wedge \beta}\wedge E_\ell}{\beta}\wedge E_r \quad \dfrac{(\alpha \wedge \beta) \wedge \gamma}{\gamma}\wedge E_r}{\beta \wedge \gamma}\wedge I}{\alpha \wedge (\beta \wedge \gamma)}\wedge I$$

Example: Currying

Sum Types

Finally we introduce the sum of two types α and β, denote $\alpha + \beta$. This can be thought of as a bag consisting of objects that are either of type α or of type β but must be *tagged* with which type they are. We can throw terms of either type in $\alpha + \beta$ but we have to label them first. This loose collection of types is often referred to as an *enum* type, for example C/C++ have enums which behave like sum types. By nesting sum types a programmer can create custom types consisting of what ever terms/types they require.

In order to write a program into a sum type, it is sufficient to have a term of either type and place it in with a tag. In the constructor below we use inl(eft) to tag terms that are elements type on the left of the sum and inr(ight) to tag terms of the type on the right of the sum.

Definition: Sum Type Constructors

If ${}^{\Sigma}_{a:\alpha}\mathcal{D}$ is a typing derivation of a term $a : \alpha$ from a context Σ, then this derivation can be extended in two ways to obtain derivations of terms inhabiting either $\alpha + \beta$ and $\beta + \alpha$ respectively:

$$\frac{a \ : \ \alpha}{\text{inl } a \ : \ \alpha + \beta}\ \text{inl} \qquad \frac{a \ : \ \alpha}{\text{inr } a \ : \ \beta + \alpha}\ \text{inr}$$

Type destructors tell us how to write a program *out of* a particular type. Terms of sum types $\alpha + \beta$ are either a term of type α or a term of type β. Therefore, in order to write a program out of $\alpha + \beta$, we have to take into account both of these possibilities. That is, we need to provide a program to deal with the case that the term came from α and a program to deal with the case that the term came from β. This is formalised in the following destructor.

Definition: Sum Type Destructor

If ${}^{\Sigma_1}_{t:\alpha+\beta}\mathcal{D}_1$, ${}^{\Sigma_2}_{f:\alpha\to\gamma}\mathcal{D}_2$, and ${}^{\Sigma_3}_{g:\beta\to\gamma}\mathcal{D}_3$ are typing derivations, then together these can be extended to a derivation ${}^{\Sigma}_{t:\gamma}\mathcal{D}$, where $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3$. We depict this extension as follows:

$$\frac{\begin{array}{ccc}\Sigma_1 & \Sigma_2 & \Sigma_3 \\ \mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 \\ t : \alpha + \beta & f : \alpha \to \gamma & g : \beta \to \gamma\end{array}}{\text{cases } t \ f \ g \ : \ \gamma}\ \text{cases}$$

If the term cases is passed a term inl a for $a : \alpha$, then it will apply f to it. Otherwise it will be passed inr b for $b : \beta$ and will apply g to it. These β-redex are and reduction rules are described in the following equations:

$$\text{cases } (\text{inl } a) \ f \ g =_\beta f \ a$$

$$\text{cases } (\text{inr } b) \ f \ g =_\beta g \ b$$

Example: Sum Commutative

Provide a natural deduction to witness the following sequent:

$$\alpha \vee \beta \ \vdash \ \beta \vee \alpha$$

As is now custom, we first look to rewrite the sequent with temporary hypotheses. In this case, we see that $\alpha \vee \beta$ is a hypothesis and therefore we will need to use disjunction elimination. Two branches of this step will be proofs of implications with α and β as antecedent. For this reason we temporarily add these to our hypotheses to reduce the original problem to the following sequent:

$$\alpha \vee \beta, \alpha^1, \beta^2 \ \vdash \ \beta \vee \alpha$$

In fact we can reduce the sequent further into *two* sequents:

$$\alpha \vdash \beta \vee \alpha \qquad\qquad \beta \vdash \beta \vee \alpha$$

We have reduced the original problem to providing separate proofs to witness these sequents. These can be achieved using $\vee$ introduction.

$$\dfrac{\dfrac{\overline{\alpha}\ 1}{\beta \vee \alpha}\ \vee I_R}{\alpha \to (\beta \vee \alpha)} \to I, 1 \qquad \dfrac{\dfrac{\overline{\beta}\ 2}{\beta \vee \alpha}\ \vee I_L}{\beta \to (\beta \vee \alpha)} \to I, 2$$

Together they combine with the assumption $\alpha \vee \beta$ to get a deduction witnessing the original sequent.

$$\dfrac{\alpha \vee \beta \quad \dfrac{\dfrac{\overline{\not{\alpha}}\ 1}{\beta \vee \alpha}\ \vee I_R}{\alpha \to (\beta \vee \alpha)} \to I, 1 \quad \dfrac{\dfrac{\overline{\not{\beta}}\ 2}{\beta \vee \alpha}\ \vee I_L}{\beta \to (\beta \vee \alpha)} \to I, 2}{\beta \vee \alpha}\ \vee E$$

Example: Sum Associative

Provide a natural deduction to witness the following sequent:

$$(\alpha \vee \beta) \vee \gamma \ \vdash\ \alpha \vee (\beta \vee \gamma)$$

Empty Type

Example: Modus Tollens

Provide a natural deduction to prove the following sequent:

$$\alpha \to \beta,\ \neg\beta \ \vdash\ \neg\alpha$$

Notice that this sequent is equivalent to

$$\alpha \to \beta,\ \neg\beta \ \vdash\ \alpha \to \bot$$

When it is rewritten like this it is clearer that the deduction theorem can help us reduce the sequent to the following by adding α as a temporary hypothesis:

$$\alpha \to \beta,\ \neg\beta,\ \alpha^1 \vdash\ \bot$$

The following deduction witnesses the new sequent. Notice that $\neg\beta$ is an implication and therefore modus ponens is the correct elimination rule to use on it.

$$\dfrac{\dfrac{\overline{\alpha}^{\,1} \quad \alpha \to \beta}{\beta}\,\text{MP} \qquad \neg\beta}{\bot}\,\text{MP}$$

This deduction can be extended by implication introduction to witness the original sequent. Again, there is no special negation introduction rule. We just use implication introduction.

$$\dfrac{\dfrac{\dfrac{\overline{\alpha}^{\,1} \quad \alpha \to \beta}{\beta}\,\text{MP} \qquad \neg\beta}{\bot}\,\text{MP}}{\neg\alpha}\,\to I,1$$

Example: Contraposition

Provide a natural deduction to prove the following sequent:

$$\alpha \to \beta \ \vdash\ \neg\beta \to \neg\alpha$$

The deduction rule can be applied twice, remembering to desugar the negations into implications. This reduces the sequent to the following:

$$\alpha \to \beta,\ \neg\beta^1,\ \alpha^2 \vdash\ \bot$$

The following deduction witnesses the reduced sequent:

$$\dfrac{\dfrac{\alpha \to \beta \quad \overline{\alpha}^{\,2}}{\beta}\,\text{MP} \qquad \overline{\neg\beta}^{\,1}}{\bot}\,\text{MP}$$

This sequent can be extended using implication introduction to obtain a deduction witnessing the original sequent.

$$\dfrac{\dfrac{\dfrac{\alpha \to \beta \quad \overline{\alpha}^{\,2}}{\beta}\,\text{MP} \qquad \overline{\neg\beta}^{\,1}}{\bot}\,\text{MP}}{\neg\alpha}\,\to I,2}{\neg\beta \to \neg\alpha}\,\to I,1$$

4.4 Curry-Howard Isomorphism

Let's compare the following natural deduction and typing derivation.

$$\dfrac{\dfrac{\dfrac{\overline{P}^{\,1} \quad \overline{Q}^{\,2}}{P \wedge Q}\,\wedge I \qquad (P \wedge Q) \to R}{R}\,\mathsf{MP}}{\dfrac{Q \to R}{P \to (Q \to R)} \to I, 2} \to I, 1$$

$$\dfrac{\dfrac{\dfrac{\overline{a : P}^{\,1} \quad \overline{b : Q}^{\,2}}{(a, b) : P \times Q}\,\times \qquad f \;:\; (P \times Q) \to R}{f(a, b) \;:\; R}\,\mathsf{app}}{\dfrac{\lambda y.\ f(a, y) \;:\; Q \to R}{\lambda x.\ \lambda y.\ f(x, y) \;:\; P \to (Q \to R)}\ \lambda, 1}\ \lambda, 2$$

On the left we have propositions. On the right we have types. On the left is a natural deduction. On the right is a typing derivation for a program. These derivations are remarkably similar! Indeed they are precisely the same after translating with the following table:

Logic	Type Theory
Proposition P	Type P
Proof	Program
$\to$	Function type
$\wedge$	$\times$
$\wedge E_l$	FST
$\wedge E_r$	SND
$\vee$	$+$
$\vee I_l$	inl
$\vee I_r$	inr
$\vee E$	cases

If we interpret propositions as types, then all rules of inference have their type theoretic counterpart. For each introduction rule there is a type constructor and for each elimination rule there is a type destructor. For example, implication introduction corresponds to λ-abstraction i.e. factoring out a variable. Or-elimination corresponds to the typing rule for writing a program out of a sum-type; we can't know which type in the sum we will have, so we need to account for both possibilities, just like the two branches in Or-elimination. This ability to interchange between logic and type theory is known as the Curry-Howard Isomorphism, commonly referred to as the Curry-Howard Correspondence. In so far as we have presented the isomorphism here we could state it this way:

$$\text{Minimal Logic} = \text{Simple Type Theory}$$

In more concrete terms the isomorphism can be presented as a meta-theorem comparing those sequents that have derivations in minimal logic and those sequents which have typing derivations in simple type theory. Let Σ_L denote a set of propositions, α another proposition, and $\vdash_{\mathsf{ML}}$ denote the usual turnstile in logic while $\vdash_{\mathsf{STT}}$ denote the usual turnstile from simple type theory. If we interpret Σ_L as a set Σ_T of inhabited types and α as a type, then

$$\Sigma_L \;\vdash_{\mathsf{ML}}\; \alpha \qquad \text{if and only if} \qquad \Sigma_T \;\vdash_{\mathsf{STT}}\; \alpha$$

Under the interpretation of propositions-as-types, the well-typed terms of simple type theory correspond to specific natural deductions; the programs encode each step of the proof. Put simply: proofs are programs. Since $\lambda x.\ \lambda y.\ f(x, y)$ encodes a proof of the proposition $P \to (Q \to R)$ we refer to the program as a *proof-term* for the proposition. In order to prove the proposition it is sufficient to write a program of that type.

What do these programs do? Well the type signature on the abstractions $\lambda x : P.$ tell us the input of a program is a term of another type. We just said a term x of type P encodes a proof of P as a proposition. So these programs take in proofs and yield other proofs. In this sense the Curry-Howard Isomorphism is a concrete realization of the BHK-interpretation of the logical connectives. For this reason BHK-interpretation is often used synonymously with Curry-Howard Isomorphism. By carefully considering the process of computation Church, Curry, and Howard have realized the hope set out by Brouwer, Heyting, and Kolmogorov.

Chapter 5

Lean

This chapter is not intended to be a complete and comprehensive introduction to theorem proving in Lean, or indeed programming in Lean. However, it will give an idea of what's required to formalize mathematics and how to use the proof assistant feature of Lean. This chapter was developed by learning from the following two texts and reading mathlib. The reader is encouraged to visit these for more details about formalizing mathematics in Lean:

- Theorem Proving in Lean 4.
- Hitchhiker's Guide to Logical Verification.
- Mathlib.

5.1 Introduction

David Hilbert asked whether there is an algorithm for determining theoremhood? Whether proofs could be computed? In the century that followed, logicians took this question seriously and answered it emphatically; No, there can be no one algorithm that can decide theoremhood. Throughout this course we have studied some of the ideas that arose in this quest. Formal logic was developed so that mathematics and proofs could be expressed in a precise enough way that they are amenable to computation. We have seen how number theory can be written in this language by using the Peano Axioms and how theorems about the natural numbers can be proved. In order to answer Hilbert logicians had to clarify what was meant by "algorithm" and "computation". Turing Machines were advanced by Alan Turing, Recursive Functions by Kleene and Gödel, and the λ-calculus by Alonso Church - at this point there are many models of computation. In this course we first explored the nature of computation from the point-of-view of the λ-calculus. It is generally believed that the λ-calculus is capable of expressing any procedure that is deemed computable; that the general concept of computability is captured by β-reduction in the λ-calculus - this belief is known as the Church-Turing-Kleene Thesis - hereafter simply, Church's Thesis. In fact Church, Haskell, and others showed that the λ-calculus can express computations whose normalisation is undecidable. Church's thesis suggests this is saying the same thing as Turing's proof of the undecidability of the Halting Problem. There can be no λ-term which determines whether any other λ-term has a normal form. Nor any Turing machine that determines whether any other Turing machine will halt. Put plainly, Church showed that there are λ-terms which may go on computing forever, or stop in the very next second - we just can't know! In reaction to this Church, Haskell Curry, and others developed the typed λ-calculi. So long as programs were well-typed, they showed their programs were normalising. Of course this means giving up expressibility - these languages can't be Turing complete if all programs are known to normalise/halt - in favour of normalisation.

Following this thread in the theory of computation we were led back to the start of our story, back to natural deductions. Haskell Curry, William Howard and others have shown that the process of writing well-typed λ-terms is the very same process as intuitionistic natural deductions. This correspondence, now called the Curry-Howard Correspondence, tells us that proofs in logic can be encoded in λ-terms. If $t : T$ is a λ-term, then t encodes of a proof T interpreted as a proposition. Much of this work by logicians on logic, computation, and type theory is now continued in *programming language theory*. All of this work on computation has informed what we now call programming languages; the languages we use to cast spells into our devices. Functional programming languages like ML, the LISP family, Miranda, and Haskell were all ultimately inspired by the λ-calculus in both its typed and untyped variants. Following on from Curry and Howard, the logicians Jean-Yves Girard and Per Martin-Löf extended the Curry-Howard correspondence to first (and higher) order logics. These so called *dependently typed λ-calculi* have formed the foundation of a number of modern programming languages; Coq, Agda, Isabelle, and Lean are just a few of the many languages available to write and check formal proofs/programs. With the development of programming languages, we now longer use typing derivations to write programs. We write the program using a specific language (e.g. Haskell) and we use a type checker (another program) to verify the program we have written has the type we claim it does. This final chapter will explain how to move away from natural deductions and typing derivations, by writing our proof-terms in the programming language Lean.

Lean was released in 2013 by Leonardo de Moura, of Microsoft Research. At the core of the standard library of Lean are all the ideas we have studied in this course, along with many other modern developments in programming language theory. With these theoretical foundations one can write programs and have the Lean type checker check the proofs. Programmers in Lean have the ability to define their own types which allows the programmer to express mathematics in the language. More than just a type checker Lean is a *proof assistant*. It is a fully fledged programming language and therefore the user is able to define meta-programs (often referred to as tactics) to help generate proof-terms. Although we have written proof-terms explicitly (and by hand), we will soon see that the ability to write meta-programs is an indispensible feature for writing proofs about mathematics of even very limited sophistication.

5.2 Proof-terms in Lean

At this point we have all but written Lean programs already. Those well-typed programs written by typing derivations in the previous chapter are a few syntax changes away from being code accepted by the Lean type-checker. All of the syntax changes for writing programs in Lean are collated in the following table:

PL	λ	L∃∀N 4
$\wedge I$	(p, q)	And.intro p q
$\wedge E_l$	fst t	And.left t
$\wedge E_r$	snd t	And.right t
$\rightarrow I$	$\lambda\ p : P.$	$\lambda\ p : P =>$
$\rightarrow E$	$(f\ t)$	$(f\ t)$
$\vee I_l$	inl p	Or.intro_left <right-disj> p
$\vee I_r$	inr p	Or.intro_right <left-disj> p
$\vee E$	cases t f g	Or.elim t f g

Table 5.1: Syntax of logic, λ-calculus, and L∃∀N 4

For example, where we would use $\wedge$ introduction in propositional logic, or the product $\times$ constructor in simple type theory, we use the Lean expression And.intro. Although it's not quite as simple as swapping terms out place by place, it pretty much is that simple! We have been using the $\times$ type constructor as an infix operator where as And.intro is a prefix constructor in Lean.

Disjunction introduction requires an extra argument. If we were introducing $P \vee Q$ from a term $t : P$, then one would write Or.intro_left Q t to say that a term of type P is being introduced to the left of type Q. Previously we would have simply written inl p and left the type Q to be inferred from the deduction tree. Similarly, if given a term $t : Q$, then one would write Or.intro_right P t to introduce the term of type Q to the right of the disjunction with P.

Example: Program Composition

Earlier we derived the following proof-term witnessing the composability of programs.

$$\lambda x : P.\ g\ (f\ x)$$

When writing a proof in Lean extra structure is needed to tell Lean the theorem you're intending to prove. The following is a complete working example of Lean code for this theorem:

```
variable (P Q R : Prop)

--                          P → Q       Q → R  ⊢ P → R
theorem composition (f : P → Q) (g : Q → R) : P → R :=
  λ p : P =>
    g (f p)
```

Line-by-line this is what the code is doing. First we declare P, Q, and R to be propositional variables. Next, lines beginning with -- are comments; they are for human readers only, to help illustrate what the code is doing. This comment is indicating how to align our usual syntax with that of Lean on the next line. Lean programs can be opened by the "theorem" keyword and have the following structure:

theorem <name> <hypotheses> : <goal-type> := <proof-term>

This theorem is named composition. It has two hypotheses; $(f : P \rightarrow Q)$ and $(g : Q \rightarrow R)$. Our aim is to prove, under these hypotheses, the proposition $P \rightarrow R$; this is called the *goal* of the theorem. Finally, the proof-term is translated from the typing derivation earlier in the notes using the table at the beginning of this section. This proof-term has type identical to the goal and as such is sufficient for a proof of this theorem.

Example: I Combinator

In this example we consider the I combinator, or identity function. This program witnesses the proof of the following theorem:

$$\vdash P \to P$$

Notice this is a *theorem* and as such has no hypotheses. For this reason, the goal is stated directly after the name of the theorem.

```
variable (P : Prop)

--                    ⊢ P → P
theorem Icombinator : P → P :=
    λ p : P =>
        p
```

Example: S Combinator

In this example we show that the bound variables of the S combinator can be consistently assigned types. Again, this proof-term corresponds to a theorem with no hypotheses. Again, this is an exercise in *translation* as we have already derived a proof-term for this theorem. Write the corresponding term in Lean.

```
variable (P Q R : Prop)

--                    ⊢ (P → (Q → R)) → ((P → Q) → (P → R))
theorem Scombinator : (P → (Q → R)) → ((P → Q) → (P → R)) :=
λ f : P → (Q → R) =>
  λ g : P → Q =>
    λ w : P =>
      (f w) (g w)
```

There are three nested implications and therefore three abstractions. Equivalently, there are three applications of the deduction theorem.

Example: K Combinator

This theorem confuses many when viewed from the point of view of propositional logic. Some find this computational point-of-view clarifying.

```
variable (P Q : Prop)

--                    ⊢ P → (Q → P)
theorem Kcombinator : P → (Q → P) :=
λ p : P =>
  λ q : Q =>
    p
```

Given a term of type P (i.e. $\lambda\, p : P =>$) how does one obtain a term of type $Q \to P$? Such a term is a function from Q which returns a term of type P. Which P does it return? It returns the input p from the first abstraction. This is really the same as the I combinator proof; from a term of type P one can always obtain a term of type P.

Example: Conjunction Commutative

Earlier we derived the following proof-term witnessing the commutativity of conjunction:

$$(\text{snd } t, \text{fst } t) : Q \wedge P$$

Using the Table this proof-term can be written as follows:

```
variable (P Q : Prop)

--                        P ∧ Q  ⊢ Q ∧ P
theorem conj_comm (t : P ∧ Q) : Q ∧ P :=
    And.intro (And.right t) (And.left t)
```

Example: Conjunction Associative

Earlier we derived the following proof-term witnessing the associativity of conjunction:

$$(\text{fst } (\text{fst } t), (\text{snd } (\text{fst } t), \text{snd } t)) : P \wedge (Q \wedge R)$$

There is another way to project off the components of a term $t : P \wedge Q$ of a conjunction type. Instead of (fstt) one can simply use t.left to get the first component and t.right to get the second component. Using this form of $\wedge$ elimination instead the proof term can be written as follows:

```
variable (P Q R : Prop)

--                  ⊢ (P ∧ Q) ∧ R → P ∧ (Q ∧ R)
theorem conj_assoc : (P ∧ Q) ∧ R → P ∧ (Q ∧ R) :=
    λ t : (P ∧ Q) ∧ R =>
        And.intro (t.left.left)
                  (And.intro (t.left.right) (t.right))
```

Example: Currying

Let's revisit currying. Recall the following proof from earlier in the course where the two approaches have been blended together, some type theory notation with some propositional logic notation.

$$f : (P \times Q) \to R \vdash P \to (Q \to R)$$

$$\dfrac{\dfrac{f \ : \ (P \wedge Q) \to R \quad \dfrac{\overline{a : P}^{\,1} \quad \overline{b : Q}^{\,2}}{(a,b) : P \wedge Q}\wedge I}{f(a,b) \ : \ R}\text{MP}}{\dfrac{\lambda y.\ f(a,y) \ : \ Q \to R}{\lambda x.\lambda y.\ f(x,y) \ : \ P \to (Q \to R)}\lambda,2}\lambda,1$$

The resulting explicitly typed λ-term below

$$\lambda x : P.\ \lambda y : Q.\ f\ (x,y) \ : \ P \to (Q \to R)$$

is the proof-object witnessing the proof of the sequent. According to the syntax table (Table **??**) this is translated into the following Lean code:

```
variable (P Q R : Prop)

--                  (P ∧ Q) → R  ⊢ P → (Q → R)
```

```
theorem curry (f : (P ∧ Q) → R) : P → (Q → R) :=
    λ p : P =>
        λ q : Q =>
            f (And.intro p q)
```

This theorem is named "curry" and it uses one hypothesis $f : (P \wedge Q) \to R$ to prove the goal $P \to (Q \to R)$. In fact curry is a function that takes in a term $f : (P \wedge Q) \to R$ and returns a term of type $P \to (Q \to R)$; the program it returns is the proof-term derived above. So if one ever has a term of type $(P \wedge Q) \to R$, the program curry will turn it use it to write a term of type $P \to (Q \to R)$. This means it can be reused in future proofs.

Example: Uncurry

```
variable (P Q R : Prop)

--                      P → (Q → R)  ⊢  (P ∧ Q) → R
theorem uncurry (f :  P → (Q → R)) :  (P ∧ Q) → R :=
λ t : P ∧ Q =>
  (f (And.left t)) (And.right t)
```

The previous two examples show that the two propositions are logically equivalent

$$(P \wedge Q) \to R \dashv\vdash P \to (Q \to R)$$

We can prove this in Lean using the following code:

Example: Logical Equivalence

```
variable (P Q R : Prop)

theorem curry_equiv : (P ∧ Q) → R ↔ P → (Q → R) :=
    Iff.intro
        -- (P ∧ Q) → R ⊢ P → (Q → R)
        (curry P Q R)
        -- P → (Q → R) ⊢ (P ∧ Q) → R
        (uncurry P Q R)
```

Iff.intro (read: If and only if intro) has two arguments corresponding to the forward and reverse direction of the proof. One could explicitly write the proofs of these implications again. However, it suffices to state the name of the corresponding theorem with the appropriate arguments. In this case the arguments are simply the propositional variables used in the statement of the theorem.

Example: Disjunction Commutative

Earlier we derived the following proof-term witnessing the commutativity of disjunction:

$$\text{cases } t\ (\lambda\ p : P.\ \text{inr } p)\ (\lambda\ q : Q.\ \text{inl } Q) : Q \vee P$$

Using the Table **??** this proof-term can be written as follows:

```
variable (P Q : Prop)

--           P ∨ Q  ⊢ Q ∨ P
example (t : P ∨ Q) : Q ∨ P :=
Or.elim t
  (λ p : P =>
    Or.intro_right Q p)
  (λ q : Q =>
    Or.intro_left P q)
```

Example: Disjunction Associative

Earlier we derived the following proof-term witnessing the associativity of conjunction:

$\text{cases } t\ (\lambda\ x : P \vee Q.\ \text{cases } x\ (\lambda p : P.\ \text{inl } p)\ (\lambda q : Q.\ \text{inr } (\text{inl } q)))\ (\lambda r : R.\ \text{inr } (\text{inr } r)) : P \vee (Q \vee R)$

Table **??** can be used to translate this proof-term. There are three inputs for the cases (Or.elim) constructor, using indenting and aligning the components can help keep the code clear.

```
variable (P Q R : Prop)

--                      (P ∨ Q) ∨ R  ⊢ P ∨ (Q ∨ R)
theorem disj_assoc (t : (P ∨ Q) ∨ R) : P ∨ (Q ∨ R) :=
    Or.elim t
    (λ w1 : P ∨ Q =>
        Or.elim w1
        (λ w2 : P =>
            Or.intro_left (Q ∨ R) w2)
        (λ w3 : Q =>
            Or.intro_right P (Or.intro_left R w3)))
    (λ w4 : R =>
        Or.intro_right P (Or.intro_right Q w4))
```

Example: Modus Tollens

Earlier we derived the following proof-term witnessing Modus Tollens:

$$\lambda x : P.\ g\ (f\ p) : \neg P$$

Table **??** can be used to translate this proof-term. Notice that since the hypotheses and goal are stated as part of the theorem, Lean is able to infer the correct type of the abstraction and we need not write it explicitly.

```
variable (P Q : Prop)

--                          P → Q       ¬Q  ⊢ ¬P
theorem modus_tollens (f : P → Q) (g : ¬Q) : ¬P :=
λ p => g (f p)
```

Example: Contraposition

```
variable (P Q : Prop)

--                              P → Q  ⊢ ¬Q → ¬P
theorem icontraposition (f : P → Q) : ¬Q → ¬P :=
λ w : ¬Q =>
  modus_tollens P Q f w
```

5.3 Classical Logic in Lean

In so far as we have covered the idea of propositions-as-types, the correspondence is typically considered to be between intuitionistic logic and type theory[1]. Lean's type system has an axiom, called the axiom of choice, which allows for a *proof* of the Law of Excluded Middle. This result is often referred to as Diaconescu's theorem. This means that Lean is capable of expressing and checking theorems in classical logic. Typing the following prompt into Lean:

```
open Classical

-- em : ∀ P : Prop, P ∨ ¬P
#print em
```

Shows a long proof of LEM following the proof of Diaconescu's Theorem. Notice that the "Classical" name space must be open before this term is available. There are other elements of Lean's underlying type theory that allow for this proof, but these will not be covered here. For more on this the reader is encouraged to see the final chapter of Theorem Proving in Lean 4.

For those interested in proving theorems of classical logic in Lean it is enough to know a term in the Classical namespace can be called upon to use LEM in a proof. For example, we can write the proof that double negation elimination follows from the law of excluded middle in Lean as follows:

```
variable (P : Prop)
open Classical

--          ⊢ ¬¬P → P
theorem dne : ¬¬P → P :=
  λ nnp : ¬¬P =>
    Or.elim (em P)
      (λ p : P => p)
      (λ np : ¬P => False.elim (nnp np))
```

In fact the classical namespace allows for notation to write a proof by contradiction. Here is the same proof written with the byContradiction pattern:

```
variable (P : Prop)
open Classical

--          ⊢ ¬¬P → P
theorem dne : ¬¬P → P :=
    λ nnp : ¬¬P =>
        byContradiction
            (λ np : ¬P => nnp np)
```

These are sufficient for our purposes in this course. Further patterns for classical logic can be found in Theorem Proving in Lean 4.

[1] Work by Timothy Griffin has shown there to be a computational interpretation of classical logic, although this is not how Lean implements classical logic and not covered in these notes

5.4 Tactic Proofs in Lean

We have seen that Lean is capable of checking proofs that we write. In this section we will see that Lean is also a good assistant, capable of helping us compose proof-terms. This is indispensible for proofs of even moderate difficulty. Lean is a general purpose programming language, which means programmers can write programs to help generate proof-terms; so called, *metaprograms* or *tactics*. Therefore it is not necessary to write proof terms directly, but rather use (and write new) tatics to help expedite the process of writing proofs. In a sense this is similar to how computers understand binaries, but we write programs in higher level languages that are closer to how we think. CPUs don't naturally understand the language we write, our source-code must first be compiled (interpreted, or assembled) down to something the CPU can make sense of. Our proof-terms are the certificates checked by the type checker, but that does not mean we have to write them explicitly in that way. We can use higher-order tactics and "compile" them down to explicit proof terms; this processes is called *elaboration*. That is, elaboration takes source code using higher-order procedures to proof-terms, as compilation takes source code to binaries.

Tactics focus on the goal. Changing the goal and proof state into equivalent states. This is akin to our use of the deduction theorem when using temporary hypotheses. Or breaking an Or-elimination proof up into a few steps. By applying a tactic we are telling Lean "in order to prove this goal from these hypotheses, it suffices to prove this new goal from this new set of hypotheses". It can be helpful to have the corresponding natural deduction at hand when writing these proofs.

Example: Meta-programs aka Tactics

Let us return to the type corresponding to the S-combinator. In the previous section we wrote the corresponding proof-term directly into Lean. This time we will illustrate the use of "tactic mode" in Lean to prove this theorem.

```
variable (P Q R S T : Prop)

-- S combinator
theorem scombinator : (P → (Q → R)) → ((P → Q) → (P → R)) :=
    by
        intro f
        intro g
        intro p
        exact (f p) (g p)
```

First notice the use of the keyword "by" to indicate that tactic mode is being used. Each line after that must begin with a tactic. In this example two tactics are being used; intro, and exact. Along with the simplified syntax, tactic mode (in VSCode) has a second panel that keeps track of the proof state, or proof state; the live hypotheses and the goal. This changes each time a tactic is applied; this dynamic process is difficult to illustrate in a static document so make sure to watch the demonstrations in lectures and in labs.
Initially the proof state reads as follows:

```
1 goal
P Q R S T : Prop
⊢ (P → Q → R) → (P → Q) → P → R
```

Indicating the hypotheses and the goal. Note the only hypothesis is the fact that P Q R S T are all propositions.
Intro is analogous to our use of the deduction theorem. It moves the antecedent of an implication into the context and changes the goal accordingly. We are assuming the antecedent and Lean knows that it needs to be discharged later, so we don't bother writing the discharge line in the proof. There are three implications, therefore three introductions. Lean has the ability to infer the type being introduced, so it is sufficient to supply a name to the term of the type. After the first use of the intro tactic, the proof state changes to the following

```
1 goal
P Q R S T : Prop
f : P → Q → R
⊢ (P → Q) → P → R
```

Notice both the hypotheses and goal have changed. Lean infers the type to assign the term f and changes the goal according this new hypotheses. Two further introductions change the proof state to the following:

```
1 goal
P Q R S T : Prop
f : P → Q → R
g : P → Q
p : P
⊢ R
```

Under the assumption of two more hypotheses it is sufficient to give a term of type R. Exact is a line to say the proof is finished. It takes as an argument a term with the same type as the goal; in this case a term of type R must be supplied. Finally the proof state reads as follows:

```
No goals
```

Glorious! Our one goal has been achieved and we are done! But what does this tactic proof give us? It produces a proof term. This can always be accessed by the following command:

```
variable (P Q R : Prop)

-- S combinator
theorem scombinator : (P → (Q → R)) → ((P → Q) → (P → R)) :=
  by
    intro f
    intro g
    intro p
    exact (f p) (g p)

#print scombinator

------------------------

∀ (P Q R : Prop), (P → Q → R) → (P → Q) → P → R :=
fun P Q R f g p => f p (g p)
```

Notice that the proof-term written by our tactic proof is the same proof-term that we wrote explicitly in the previous section and, indeed, in the previous chapter on simple type theory. Tactic proofs still write λ-terms, these are the certificates that we use to verify the proof. This new method allows for a different experience in *writing the proof* but leads to the same output.

Example: I

n the previous example the two tactics were applied to the goal. Tactics can also change the proof state by working on hypotheses. In this example we will prove the commutativity of disjunction using a new tactic called "apply". Setting up the theorem and using the keyword "by" to indicate the proof will be written in tactic-mode, the proof state is presented as follows:

```
1 goal
P Q R S T : Prop
t : P ∨ Q
⊢ Q ∨ P
```

The key step in the pen-and-paper version of this proof is the use of $\vee$ elimination. Although this is (in some sense) the last step in the proof, it is often the first thing we would write down. From there we would proceed *backwards* up the two implication branches. This "backwards" working can be implemented in Lean using tactics. Lets "apply" $\vee$-elimination to the hypothesis $t : P \vee Q$:

```
variable (P Q : Prop)

theorem disj_comm (t : P ∨ Q) : Q ∨ P :=
by
  apply Or.elim t
```

First thing one might notice is the redline underneath "by"; that's simply saying we are not done yet. One will also see that we have moved from 1 goal to 2 goals!

```
2 goals

case left
P Q : Prop
t : P ∨ Q
⊢ P → Q ∨ P

case right
P Q : Prop
t : P ∨ Q
⊢ Q → Q ∨ P
```

Each new goal, or case, corresponds to the two implications required for the $\vee$ elimination step. These have to be tackled in the order presented when using this tactic.

```
variable (P Q : Prop)

theorem disj_comm (t : P ∨ Q) : Q ∨ P :=
by
  apply Or.elim t
  -- case left
  intro p
  apply Or.intro_right
  exact p
  -- case right
  intro ?
  apply ?
  exact ?
```

First we prove the implication $P \to Q \vee P$. This requires supposing the antecedent using the intro tactic. One can also "apply" terms to the goal. Since we are to prove a disjunction we can apply Or.intro_right. This amounts to saying, in order to prove the current goal, it is sufficient

to prove P. This changes the goal to P and we can use (exact) the assumption p introduced in this case. Finish the right-hand case by filling in the ? with the appropriate terms.

Example: I

n this example we will prove the associativity of conjunction. With this proof we will illustrate the use of one new tactic, the "have" tactic. Setting up the theorem and using "by" to indicate tactic mode we are presented with the following proof state:

```
1 goal
P Q R S T : Prop
t : (P ∧ Q) ∧ R
⊢ P ∧ Q ∧ R
```

As we are to prove a conjunction, the first move is to apply And.intro. This says it is sufficient to prove both sides of the conjunct and therefore splits the proof into two subproofs, or two cases.

```
2 goals

case left
P Q R S T : Prop
t : (P ∧ Q) ∧ R
⊢ P

case right
P Q R S T : Prop
t : (P ∧ Q) ∧ R
⊢ Q ∧ R
```

To settle the first goal, we must provide a term of type P. This can be obtained from our hypothesis t with two applications $\wedge$-elimination; being careful to pick the correct side. This can be done inline by nesting applications of And.elim. However, we will illustrate the use of "have" to break the proof down into steps. On the path proving P we first prove $P \wedge Q$. This can written in Lean in the following way:

```
variable (P Q R S T : Prop)

theorem conj_assoc (t : (P ∧ Q) ∧ R) : P ∧ (Q ∧ R) :=
by
    apply And.intro
    have h1 : P ∧ Q := And.left t
```

Accordingly the proof state (case left) updates to indicate the introduction of a new hypothesis.

```
2 goals

case left
P Q R S T : Prop
t : (P ∧ Q) ∧ R
h1 : P ∧ Q
⊢ P
```

Exact can close this case in one-line, but we will grab P of the term h1 using have again to illustrate the tactic.

```
variable (P Q R S T : Prop)

theorem conj_assoc (t : (P ∧ Q) ∧ R) : P ∧ (Q ∧ R) :=
by
    apply And.intro
    -- case left
```

```
    have h1 : P ∧ Q := And.left t
    have h2 : P     := And.left h1
    exact h2
```

At this point in the proof we are to prove the second case, the conclusion of which is also a conjunction. Another application of And.intro will split the second case into two sub-cases, thus we have nested cases - we best keep our wits about us; we can use the proof state display to keep track of which sub-(sub-sub-...) proof we are currently working on.

```
variable (P Q R S T : Prop)

theorem conj_assoc (t : (P ∧ Q) ∧ R) : P ∧ (Q ∧ R) :=
by
    apply And.intro
    -- case left
    have h1 : P ∧ Q := And.left t
    have h2 : P     := And.left h1
    exact h2
    -- left case of right case
    apply ?
    have h3 : ?   := ?
    have h4 : ?   := ?
    exact ?
    -- right case of right case
    exact ?
```

Complete the remainder of the proof by replacing ? with appropriate terms.

5.5 First-Order Logic in Lean

Our discussion on the Curry-Howard isomorphism has so far shown us that there is an equivalence between natural deductions in (intuitionistic) propositional logic and typing derivations of well-typed λ-terms in simple type theory. In fact the Curry-Howard isomorphism can be extended to intuitionistic first-order predicate logic. Natural deductions in first-order logic correspond to typing derivations in *dependent type theory*. Rather than cover dependent type theory by hand, like we did simple type theory, we will jump straight to writing proofs in Lean once a few preliminary ideas have been covered. For further reading about dependent types one can consult with either of the following [3] [4].

First-order predicate logic introduced predicates and quantifers to propositional logic. Let us first consider how we think of predicates in dependent type theory. We defined predicates as functions from the domain of quantification into the set Prop of propositions. Each value(s) of the domain is mapped to a particular proposition.

Example: P

arity (even/odd) provide examples of unary predicates on the type of natural numbers. For each natural number n the predicate "even" determines a proposition "even n" which may either be provable, or refutable. In Lean we can express this predicate as follows:

In the previous example the predicate "even" is a device that generates propositions from natural numbers. That is, for each natural number n the predicate "even" generates a different proposition "even n". Under the propositions-as-types isomorphism this is equivalent to saying that the predicate "even" generates a different type "even n" for each natural number. Since these types *depend* on the particular natural number n, they are referred to as *dependent types*. Verification of (e.g.) "even 2" or "$\neg$ even 71" requires providing a term of these type. What this looks like depends on the definition of even. We will return to this later.

Example: B

inary predicates, like "divides" or "equality", require two values in order to obtain a proposition. Therefore a binary predicate is a family of types parameterised by (depending on) two values of the domain type.

In general then an n-ary predicate on a domain type is a device for taking n terms of the domain type and returning a Prop:

We can therefore reason about an arbitrary predicate on an arbitrary type as we had been doing in the exercises of the first order logic tutorials. This will require understanding the type theoretic interpretation of the quantifers.

Since the Curry-Howard isomorphism is the rigourous realization of the BHK interpretation, it is sufficient for our purpose of writing proofs in Lean to revist the BHK interpreation of the quantifiers [1]. Intuitionistic proofs introducing quantifiers must be of the following form for an arbitrary predicate P:

$\forall$	a proof of $\forall x\ P\ x$ is an algorithm that, applied to *any* object x proves that $P(x)$.
$\exists$	to prove $\exists x\ P\ x$ one must construct an object x and prove that $P(x)$ holds.

Universally quantified statements (with no free variables) $\forall x\ P\ x$ are propositions. We aim to either prove them, or refute them. Under the Curry-Howard isomorphism this proposition is interpreted as a type $\forall x\ P\ x$. What does it mean to have a term of this type? We turn to the BHK to tell us; a term of type $\forall x\ P\ x$ is a function[2] that takes in an *arbitrary* term x (of some domain) and returns a proof/term of the proposition/type $P\ x$. Thus to write a proof of such a statement in Lean we are required to

[2]Notice that the domain type *depends* on the specific value of the input. For this reason such terms are ofern referred to as *dependent functions*.

write such a program. Furthermore, any hypotheses that are universally quantified act according to this definition; namely, as functions that return terms of specific types.

$\forall$-modus ponens. In this example we prove a universally quantified version of the common modus ponens rule of inference that we have been using. Instead of declaring propositional variables, we now declare type variables (α) and arbitrary (unary) predicates F, G, and H on that arbitrary type. First we write the statement of the theorem:

```
variable (α : Type)
variable (F G H : α → Prop)

theorem universal_mp (f1 : ∀ x : α, F x → G x) (f2 : ∀ x : α, F x) :
    ∀ x : α, G x := sorry
```

Our two hypotheses are $f1$ and $f2$, each of which are terms of types corresponding to universal quantifiers. For each $a : \alpha$, the term $f1$ gives a term $(f1\ a) : F\ a \to G\ a$. Notice that the name of the type *depends* on the $a : \alpha$ that is passed to the function $f1$. Similarly, for each $a : \alpha$, the term $f2$ gives a term $(f2\ a) : F\ a$. Where, again, the type of the codomain depends on the term passed to the function.

```
variable (α : Type)
variable (F G H : α → Prop)

theorem universal_mp (f1 : ∀ x : α, F x → G x) (f2 : ∀ x : α, F x) :
    ∀ x : α, G x :=
        by
        intro a


variable (α : Type)
variable (F G H : α → Prop)

theorem universal_mp (f1 : ∀ x : α, F x → G x) (f2 : ∀ x : α, F x) :
    ∀ x : α, G x :=
        by
        intro a
        have w1 : F a := f2 a
        have w2 : G a := (f1 a) w1
        exact w2
```

Existentially quantified statements are also propositions and are thus equally well thought of as types. Here the BHK tells us that a term of type $\exists x\ P\ x$ is a pair consisting of a term t of the domain and a term of the type $P\ t$ witnessing the proof that t satisfies the predicate P.

5.6 Peano Arithmetic in Lean

We have seen that the propositions-as-types correspondence allows for the encoding of first-order logic into type theory and now into Lean. Using this correspondence we can prove theorems about arbitrary propositions and first-order statements; theorems of logic. In order to prove statements about familiar mathematical objects - such as the natural numbers - these too must be encoded into our type theory. Rather than cover the pen-and-paper details of encoding the natural numbers into type theory, we will instead jump straight into writing the relevant Lean code. We will stop writing proof terms explicitly from now on and rely heavily on in-built tactics to aid our authoring of proofs.

We will use the Peano axioms to guide the representation of the type of natural numbers $\mathbb{N}$ in Lean. There are precisely two ways to construct a new natural number: either stating that zero is a natural number, or applying the successor constructor to a natural number already at hand. In Lean we can define the type as follows:

```
inductive ℕ where
| zero : ℕ
| succ : ℕ → ℕ
```

This defines the term "zero" to be a natural number and "succ" to be a function from natural numbers to natural numbers. Peano axioms 1 and 2 are implicit in this definition: Lean treats any two terms constructed with a different sequence of constructors as not equal, and succ as an injective function. For completeness we state the code for these "theorems" below, however for the purposes of the assessment it is not necessary to know how the tactic injection works.

```
theorem PA1 : ∀ x : ℕ, ¬(zero = succ x) :=
by
  intro a
  intro t
  injection t

theorem PA2 : ∀ m n : ℕ, (succ m) = (succ n) → m = n :=
  by
    intro m n
    intro t
    injection t
```

For more on the tactic "injection" one can consult this ??? document. At this point we can prove statments like $1 \neq 2$

```
theorem one_neq_two : (succ zero) ≠ succ (succ zero) :=
  by
    intro t
    have h : zero = succ zero
      := PA2 zero (succ zero) t
    exact PA1 zero h
```

Addition of Natural Numbers

Axioms 3 - 6 are recursive definitions for addition and multiplication of natural numbers. These axioms are encoded as functions on the type $\mathbb{N}$. Axiom three is the base case for the recursive definition of addition and axiom four is the recursive step. In Lean they determine the following function:

```
def add (m n : ℕ) : ℕ :=
match n with
    | zero   => m
    | succ n => succ (add m n)
```

In order to be consistent with the notation from earlier in the course we can use this definition of addition to prove theorems corresponding to PA3 and PA4.

```
theorem PA3 (m : ℕ) : m + zero = m := by rfl
theorem PA4 (m n : ℕ) : m + (succ n) = succ (m + n) := by rfl
```

These theorems introduce a new tactic "rfl" which is telling Lean these two things are *defined* to be equal, so there is nothing to prove. We have also denoted the add function with the infix "_+_" notation, rather than the prefix notation "add" which the function was defined with; this uses the typeclass system in Lean. It is enough for us to know that addition can be denoted with the infix notation. For more on the typeclass system and operator overloading one may consult this.

Example: Rewrite Tactic

We have written a natural deduction for the following theorem in the first-order language of Peano artihmetic.

$$\text{PA} \vdash \forall x : \mathbb{N},\ \text{succ}\ x = x + (\text{succ zero})$$

We enter the theorem in as follows:

```
theorem succ_eq_plusone : ∀ x : ℕ, succ x = x + (suc zero) :=
    by
        intro a
```

We start with "intro a" because we are proving a universally quantified statement. We need an arbitrary natural number to write the program showing the equation always holds. At this point the goal is as follows:

```
1 goal
a : ℕ
⊢ succ a = a + succ zero
```

Here we begin to depart from the natural deduction format setout earlier in the course. Of course, underneath the hood proof-terms are written. But the point of using meta-programs (tactics) is to make the process of writing proof-terms easier for the mathematician and computer scientist. Here we really begin to feel the difference.
Here we introduce a new tactic "rewrite" (forever abbreviated as "rw") which *acts on the goal* by applying known equations in the form of theorems. Each time we use a rw statement is as if we are saying "in order to prove the current goal, it would be sufficient to prove this updated goal" where the updated goal is obtained using a theorem. Which theorems do we apply? We consider the terms in the goal and use what we know to rw them.
In this case there is only one step of computation that could happen. On the right of the equation we see an instance of PA4; the right addition of a successor. This means we can apply PA4 - rw according to PA4 - to change the goal.

```
theorem succ_eq_plusone : ∀ x : ℕ, succ x = x + (succ zero) :=
by
    intro a
    rw [PA4]
```

With this step in the proof the goal changes to the following:

```
1 goal
a : ℕ
⊢ succ a = succ (a + zero)
```

Again, there is only one step that can be made; rw according to PA3 the right addition of zero.

```
theorem succ_eq_plusone : ∀ x : ℕ, succ x = x + (succ zero) :=
by
    intro a
    rw [PA4]
    rw [PA3]
```

At this point the goal changes to succ a = succ a which can be closed by Lean. You might wonder where rfl has gone? It is tied up in the "rw" tactic. Lean always checks for rfl when rw are performed.

In the previous example we used rw to substitute equal terms for equal terms thus (hopefully!) simplyfying the goal to something we can prove. In fact, one application of PA4 performs one step of computation; it computes the sum in the term. Similarly, one application of PA3 computes the resulting sum. In other words, we are applying β-reduction on redexes. Lean is able to do this itself using rfl! Terms don't have to be literally the same in order for rfl to close a goal; it is sufficient for them to be the same "upto computation". This simplifies the previous example considerably.

```
theorem succ_eq_plusone : ∀ x : ℕ, succ x = x + (succ zero) :=
by
    intro a
    rfl
```

This is only possible because _+_ is *defined* this way. We will need to write explicit rw statements when we are invoking theorems and not simply definitions. Note the "intro a" is still required because rfl works on equations, not on universally quantified statements.

Example: Mere Computation

It should not come as a surprise that $2 + 1 = 3$. So go ahead and prove this following the same idea as above. Give two proofs; one writing the rw statements explicitly, the other not.

```
theorem two_plus_one :
    (succ (succ zero)) + succ zero = succ (succ (succ zero)) :=
    by
        ...
```

Example: rfl

Although zero + zero is not literally the same string as the string zero, they are equivalent by definition of addition. That is, they are equivalent upto some computation. Therefore rfl will close this goal.

```
theorem basecase1 : zero + zero = zero :=
    by
        rfl
```

What is the one explicit rw statement that will also close this goal?

Example: No Mere Computation

Consider the following theorem of Peano arithmetic

$$\text{PA},\ 0 + n = n \ \vdash\ 0 + \text{succ}\ n = \text{succ}\ n$$

Here we have hypotheses other than simply the Peano axioms, so rfl won't close the goal. Go ahead and try it anyway and read the error message "zero + succ n is not definitionally equal to rhs succ n ". Of course this true. However, in the context of this theorem we have extra information, we have a witness to the equality $0 + n = n$. Close the following goal with two rw statements, one of which will use the hypotheses.

```
theorem inductionstep1 {n : ℕ} (t : zero + n = n) :
    zero + (succ n) = succ n :=
        by
            ...
```

Induction Proofs in Lean

The previous examples were the base case and inductive step for the proof by induction of the theorem

$$\text{PA} \vdash \forall x : \mathbb{N},\ zero + x = x$$

. Lean has a straightforward syntax for induction proofs.

```
-- PA ⊢ ∀ x : ℕ, 0 + succ x = succ x
theorem zero_add : ∀ x : ℕ, zero + x = x :=
    by
    intro k
    induction k with
    | zero      =>  rw [basestep1]
    | succ k ih =>  rw [inductionstep1 ih]
```

Induction first has a base case, corresponding to the line "| zero => " afterwhich follows the proof of the base case. Since we have already proved the base case, it is sufficient to call the name of that theorem. Next is the induction step "| succ k ih => " afterwhich follows the proof of the induction step using the induction hypotheses with name "ih". Again we have already provided a proof of the induction step, so it suffices to call that. In this case, the theorem has an argument so we need to supply that when calling the theorem.

One can instead write the proofs of the base case and induction step in place as follows:

```
-- PA ⊢ ∀ x : ℕ, 0 + succ x = succ x
theorem zero_add : ∀ x : ℕ, zero + x = x :=
  by
    intro k
    induction k with
    | zero      =>  rfl
    | succ k ih =>  rw [PA4, ih]
```

Notice that successive rewrites can be separated by commas, rather than writing each rw explicitly. It is considered good form to factor one's programs and this now extends to factoring proofs. Therefore one should prove the base case and induction step separately, and combine them together in another theorem. This factoring of proofs is not new to mathematicians, we push details to lemma all the time and simply refer to them when proving the more interesting theorems.

Multiplication of Natural Numbers

Peano axioms PA5 and PA6 suggest the following recursive definition of multiplication:

```
def mul (m n : ℕ) : ℕ :=
match n with
    | zero   => zero
    | succ n => add (mul m n) m
```

In order to be consistent with the notation from earlier in the course we can use this definition of multiplication to prove theorems corresponding to PA5 and PA6.

```
theorem PA5 (m : ℕ) : m * zero = zero := by rfl
theorem PA6 (m n : ℕ) : m * (succ n) = (m * n) + m := by rfl
```

Example: Computation

Prove that $3 \times 2 = 6$ using Lean. Remember you don't have to explicitly write each rw statement; but you can if you want.

```
example: (succ (succ (succ zero))) * (succ (succ zero))
            = (succ (succ (succ ( succ (succ (succ zero)))))) :=
by
    ...
```

Example: Calling Theorems as Procedures

Earlier in the course we proved the theorem

$$\text{PA} \vdash \forall x : \mathbb{N},\ x * 1 = x$$

With multiplication defined we can write this proof in Lean and verify it!

```
theorem mul_one : ∀ x : ℕ, x * (succ zero) = x :=
    by
        intro a
        ...
```

Will rfl close this? Are the two sides of this equation definitionally equal? No. Computation will simplify the left hand side, but not all the way down to x. To finish the proof we will need to call on a theorem.

```
theorem mul_one : ∀ x : ℕ, x * (succ zero) = x :=
    by
        intro a
        rw [PA6,PA5,...]
```

After applying the definition of multiplication with PA6 and PA5 (i.e. after computation) the goal is simplified to the following:

```
1 goal
a : ℕ
⊢ zero + a = a
```

Complete the proof by applying one more rw statement.

When will rfl close the goal and when do we need more? In the context of Peano arithmetic, we may use rfl when we are asking Lean to apply the *definition* of addition and/or multiplication. In other words, rfl will close the goal if one needs only axioms PA3 - PA6. Anything else is outside the *definition* of addition and multiplication. For Peano arithmetic, this implies that the proof explicitly, or implicitly by calling a theorem, relies on induction. Induction is an axiom extra to those of addition and multiplication and equalities proven by it are not proven *by the definition of addition and multiplication*. If explicit rw are required, then the "calc" tactic, illustrated in the next example, can make for more readable proofs.

Example: Calc Tactic

In this example we will revisit the theorem

$$\text{PA} \vdash \forall x : \mathbb{N},\ x * 1 = x$$

for the purposes of introducing the calc tactic.

```
theorem mul_one : ∀ x : ℕ, x * (succ zero) = x :=
by
  intro a
  calc a * (succ zero)
    _ = a * zero + a := by rw [PA6]
    _ = zero + a     := by rw [PA5]
    _ = a            := by rw [zero_add]
```

By using the calc tactic we can express the proof closer to how we would write it on paper in high school i.e. prior to learning how to write natural deductions. Starting on the left and performing algebraic manipulations (justified by relevant theorems, of course) line-by-line to eventually arrive at the right-hand side.

When using the calc tactic one can jump multiple steps in one-line, so long as it's justified in the rw statement. Indeed, one could write the above calc proof in a single line, rather than use three as in the example.

Lean Tactic Summary

At this stage we have introduced a number of tactics for helping us write proof-terms. Each of them are summarised in the table below, along with a sense of what they correspond to in a natural language, or pen-and-paper, proof of a theorem.

Lean 4 Tactics

Tactic	Summary	Natural Language
by	Opens tactic mode.	
intro(s)	Either (i) implication introduction, or (ii) $\forall$ introduction.	Let n be some natural...
exact	This term has the same type as the goal.	... as required
apply		
have	Introduces a local variable, or intermediate step.	So far we have...
rfl	Closes goals that are (up to normalization) definitionally equal.	... by definition
rw	Substitutes equals for equals.	
induction	Wraps base case and induction step into $\forall$ proof.	By induction...
calc	Line-by-line calculation showing $a = \cdots = b$.	Algebra calculation!

Below are incomplete portions of proofs from this chapter reproduced here just to have a note of their syntax on this summary page.

This is an example of the use of the induction tactic. One has to give the induction hypotheses a name in the second branch; it doesn't have to be ih, but it might as well be.

```
induction k with
| zero      =>  rfl
| succ k ih =>  rw [PA4, ih]
```

This is an example of the use of the calc tactic. One must justify each line of the calculation; notice this means that each line needs its own subproof openned using the by keyword.

```
calc a * (succ zero)
_ = a * zero + a := by rw [PA6]
_ = zero + a     := by rw [PA5]
_ = a            := by rw [zero_add]
```

Natural Numbers in Lean

```
inductive ℕ where
| zero : ℕ
| succ : ℕ → ℕ

open ℕ

theorem PA1 : ∀ x : ℕ, ¬(zero = succ x) :=
    by
    intro a t
    injection t

theorem PA2 : ∀ m n : ℕ, (succ m) = (succ n) → m = n :=
    by
    intro m n t
    injection t

def add (m n : ℕ) : ℕ :=
    match n with
    | zero   => m
    | succ n => succ (add m n)

-- The following two lines allow for the infix _+_
instance : Add ℕ where
    add := add

theorem PA3 (m : ℕ) : m + zero = m := by rfl
theorem PA4 (m n : ℕ) : m + (succ n) = succ (m + n) := by rfl

def mul (m n : ℕ) : ℕ :=
    match n with
    | zero   => zero
    | succ n => add (mul m n) m

-- The following two lines allow for the infix _*_
instance : Mul ℕ where
    mul := mul

theorem PA5 (m : ℕ) : m * zero = zero := by rfl
theorem PA6 (m n : ℕ) : m * (succ n) = (m * n) + m := by rfl
```

5.7 Prediates, Parity, and Order

It was once said that: God made the integers; all else is the work of Man. In this section we will see how the work of humans can be written in the language of Lean. Mathematicians are concerned with answering questions about the *properties* of natural numbers and how they *relate* to each other. We saw in the chapter on Formal Mathematics that these are encoded using predicates; the same is true in Lean.

Recall that a unary predicate on $\mathbb{N}$ is a function from $\mathbb{N}$ to Prop. Under the Curry-Howard correspondence this means that a predcicate is a function which takes in an $n : \mathbb{N}$ and returns a type parameterised by n; unless it's a constant type. For example, the parity of a natural number is a unary predicate on $\mathbb{N}$ which we may define as:

```
def even (m : ℕ) : Prop :=
    ∃ n : ℕ, m = 2*n

def odd (m : ℕ) : Prop :=
    ∃ n : ℕ, m = 2*n + 1
```

Similarly we can deifne binary predicates on $\mathbb{N}$ such as $\leq$ or divisibility.

```
def lte (m n : ℕ) : Prop :=
    ∃ x : ℕ, n = m + x

infix:40 " ≤ " => lte

def divides (m n : ℕ) : Prop :=
    ∃ x : ℕ, n = m * x -- Uniqueness!

-- To avoid clash with the vertical bar in pattern matching
-- We use the symbol obtained by typing \mid into the editor.
infix:30 " | " => divides
```

With these defintions more mathematics can be written in Lean. One can now prove a number of theorems from elementary number theory and verify the properties of the $\leq$ relation.

Bibliography

[1] Douglas Bridges, Erik Palmgren, and Hajime Ishihara. "Constructive Mathematics". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Fall 2022. Metaphysics Research Lab, Stanford University, 2022.

[2] Dirk van Dalen. *Logic and Structure*. Springer-Verlag, 1980.

[3] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley Publishers Ltd., 1991.

[4] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: `https://homotopytypetheory.org/book`, 2013.