Proof Assistants and Type Theory

by

Judah Towery

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Bachelors of Science
Pure Mathematics

The University of New Mexico

Albuquerque, New Mexico

April, 2025

Proof Assistants and Type Theory

by

Judah Towery

B.S., Pure Mathematics, University of New Mexico, 2025

Abstract

In this thesis, I will talk about the recent trend of the usage of proof assistants in mathematics and the type theories that form their logical foundations. First I go over a quick tutorial on the Lean proof assistant to demonstrate how this system is used to create a formal proof of the infinitude of prime numbers. I emphasize the interactive nature of formalizing an informal, natural language proof, and demonstrate how it tracks with our usual thought processes when going through a proof. Then I use the equational theories project formalization of magma theory as an example of a piece of mathematics that is uniquely amenable to being formalized, being difficult to approach without proof assistants. I talk about some interesting algebraic results on magmas and cover some of my contributions to the project. Lastly, I develop the formal language of dependent type theory which many proof assistants are based off of. Motivating type theory through the pitfalls of untyped lambda calculus, I prove some meta-theoretical properties of simply typed lambda calculus, and demonstrate the mathematical construction of functions and the natural numbers in dependent type theory.

Contents

Chapter 1

Interactive theorem proving

Interactive theorem provers, or proof assistants, are rapidly emerging as a new tool and subfield in mathematics. These proof assistant systems afford mathematicians an environment within which we can produce automatically checked and certified formal proofs of theorems that give an extra degree of certainty to the correctness of our proofs. Gradually, these systems are becoming more and more practical for usage in 'serious' mathematics, and their use is likely to only increase as time goes on. But how exactly do these proof assistants verify that a proof is correct? What does it even mean to prove something? In this chapter, I would like to give an overview of how to prove theorems with the Lean proof assistant and then give an answer to these questions.

1.1 Proving the infinitude of prime numbers with Lean

That there are infinitely many prime numbers is a classic result of number theory dating back to Euclid. Here is a paraphrasing of Euclid's proof from [Ore48]:

Theorem 1. *For any natural number n, there is a prime number larger than n.*

Proof. Let p be a prime factor of $n! + 1$. If $p \leq n$, then it divides $n!$. Since it divides $n! + 1$, it divides 1, a contradiction. Therefore, $p > n$. □

We can see that there are lots of intermediate steps that have been skipped as clear in this proof, and lots of unnamed lemmas used (thousands of them, even, going back down to the logical fundamentals). Further, this proof does not even technically show that there are infinitely many primes; we have only shown that there is no upper bound to the set of prime numbers, and we would have to further show that any unbounded-above set of positive integers is equinumerous to $\mathbb{N}$ to truly prove that there are infinitely many primes. This we shall skip for now.

Much of the labor in producing a formal proof of a theorem is finding out what all the skipped steps are and proving all of those unnamed lemmas. Interactive proof assistants do indeed assist considerably in working through and formalizing an 'informal' proof like this. We will use this theorem and its proof as a first instructive example of formalization, emphasizing the *interactive* nature of the process.

The first step in specifying a formal proof with Lean is stating the theorem itself, beginning with a theorem keyword, the name of the theorem, and then the assumptions given for the theorem. The one assumption given in this theorem is just any natural number n, which we can refer to in Lean as `n : ℕ`, which in Lean always includes 0 as well. Fortunately, we do not have to start with building the definition

of the universal quantifier or the natural numbers, as both of these are built in to the foundations of Lean, which we will be investigating in the next section. We begin the statement of a theorem named `infinite_primes` with the assumption of a natural number `n` as such:

```
theorem infinite_primes (n : ℕ)
```

The theorem is an existentially quantified statement, stating that there exists a $p \in \mathbb{N}$ such that p is prime and $p > n$. Note for one that it is not written at all in the original theorem statement that $p \in \mathbb{N}$, but that is the simplest choice for what set p should reside in, as the negative integers are irrelevant here. One thing that does not exist in the foundations of Lean is a definition of prime natural numbers, which we could conceivably construct ourselves now. This is, however, highly inconvenient, as we would also first have to define divisibility and then write potentially thousands of lines of code proving for ourselves many dozens of various lemmas of divisibility and primes that have nothing directly to do at all with the infinitude of primes. The most commonly taken option when working with mathematics in Lean is to instead use the pre-made definitions and lemmas from mathlib [Com20]. In order to not be distracted by these basic definitions and focus simply on proving our theorem, we will import the most basic theory of prime numbers from mathlib, given in the file `Mathlib.Data.Nat.Prime.Defs`. Exactly which theorems we have available from this file can be seen online here. Importing this also helpfully gives us the ordering relation $>$ on $\mathbb{N}$. So we import the theory of prime numbers and write the statement of the theorem, separated from the name and assumptions by a colon:

```
import Mathlib.Data.Nat.Prime.Defs
theorem infinite_primes (n : ℕ) : ∃ (p : ℕ), Nat.Prime p ∧ p > n
```

We are now ready to begin the proof. The symbols to indicate the start of a proof are `:=`, and we use the `by` keyword to enter tactics mode. Tactics are commands that represent each step taken through a proof. Generally, tactics are designed to resemble

the steps that mathematicians commonly make in informal proofs, so tactics-based proofs are ideal for formalizing and checking some specific informal proof of a theorem. Our code now looks like this:

```
import Mathlib.Data.Nat.Prime.Defs
theorem infinite_primes (n : ℕ) : ∃ (p : ℕ), Nat.Prime p ∧ p > n := by
```

The standard ways of installing and setting up Lean will generally have the Lean infoview running alongside the editor being used to make the proof. At this point in the proof, the infoview looks like this:

```
n : ℕ
⊢ ∃ p, Nat.Prime p ∧ p > n
```

The lines on top keep track of our assumptions and any other results that we have obtained in the process of our proof. The final line next to the $\vdash$ symbol is our goal; it is what we want to prove. As we step through the proof, the infoview will dynamically change to reflect the current state of the proof. This is a very useful logical organization tool and gives a highly interactive nature to writing a proof in Lean.

The first step taken in the proof is to choose p to be a prime factor of $n! + 1$. In order to do this, we need to first prove that there exists a prime factor of $n! + 1$, or more generally that there is a prime factor of any $n \geq 2$. This is not a theorem that exists in the mathlib file that we have imported, so in order to keep our imports minimal, I will state the theorem we want and automatically generate a temporary proof of it with `sorry`. The `sorry` tactic can generate a proof of absolutely anything, even `False`, but Lean recognizes that using `sorry` is cheating and will not accept a proof until any instance of `sorry` is removed. This tactic is helpful for building up proofs in blocks, so that we can create the structure of the proof first and then fill in all of the `sorry`s to complete the proof. Since this is a large theorem with a large

proof, I will write it outside of our current proof with its own name, as such:

```
import Mathlib.Data.Nat.Prime.Defs
theorem prime_factor {n : ℕ} (h : 2 ≤ n) : ∃ (p : ℕ), Nat.Prime p ∧ p ∣ n
    := by sorry
theorem infinite_primes (n : ℕ) : ∃ (p : ℕ), Nat.Prime p ∧ p > n := by
```

You may notice that in the statement of `prime_factor`, I wrote the assumption `n : ℕ` in curly brackets instead of parentheses. This indicates that the assumption is to be implicitly inferred from context by Lean, so that when we use the `prime_factor` theorem, we only have to provide a proof of $2 \leq n$ without redundantly specifying what n is beforehand. Lean has many automated inference abilities and sometimes many parts of parts of proofs can be outright removed, as Lean is able to infer some things itself automatically.

In order to use this p that is a prime factor of $n! + 1$, we have to first show that the condition `h : 2 ≤ n` is fulfilled for $n! + 1$. For this, we can use the `have` tactic, which allows us to create a sub-proof of some lemma within our larger proof. In this case, we want to prove that $2 \leq n! + 1$ for any n, so we write this with `have` as such:

```
import Mathlib.Data.Nat.Prime.Defs
theorem prime_factor {n : ℕ} (h : 2 ≤ n) : ∃ (p : ℕ), Nat.Prime p ∧ p ∣ n
    := by sorry
theorem infinite_primes (n : ℕ) : ∃ (p : ℕ), Nat.Prime p ∧ p > n := by
   have : 2 ≤ Nat.factorial n + 1 := by
```

We see then at this point that the infoview has updated to reflect our new sub-proof:

```
n : ℕ
⊢ 2 ≤ n.factorial + 1
```

To prove this simple theorem, my first reaction is to check the mathlib documen-

tation on the natural numbers to see what elementary tools are available to us. The theorem there that stands out to me is `Nat.add_one_le_add_one_iff` which states that $a+1 \leq b+1 \iff a \leq b$. What we want to prove is of the form $a+1 \leq b+1$, so this theorem tells us that it suffices to prove that $a \leq b$, or in other words, that $1 \leq n!$. To use this theorem, we will use the `rw` tactic, which stands for "rewrite." This tactic allows us to rewrite goals or assumptions according to equalities or equivalences: if we have a proof of $a = b$ and a goal or assumption with a in it, `rw` can rewrite that a to b, or a b to a; similarly with a proof of $P \iff Q$. We use this tactic as follows:

```
import Mathlib.Data.Nat.Prime.Defs
theorem prime_factor {n : ℕ} (h : 2 ≤ n) : ∃ (p : ℕ), Nat.Prime p ∧ p ∣ n
    := by sorry
theorem infinite_primes (n : ℕ) : ∃ (p : ℕ), Nat.Prime p ∧ p > n := by
   have : 2 ≤ Nat.factorial n + 1 := by
      rw [Nat.add_one_le_add_one_iff]
```

And our infoview updates as expected:

```
n : ℕ
⊢ 1 ≤ n.factorial
```

To prove this, we find from the same file in mathlib again another theorem that will bring us to the end: `Nat.add_one_le_of_lt`, which states that if $n < m$ then $n+1 \leq m$. So, if we want to show that $1 \leq n!$, we just have to show that $0 < n!$, which is a theorem given in the factorial file: `Nat.factorial_pos`. `Nat.add_one_le_of_lt` is not an equivalence, so we cannot use `rw` with it, but we can use the `apply` tactic. If we have a theorem of the form `h : P → Q`, and a goal of the form `Q`, then `apply h` will change our goal to `P`, which would then imply `Q`. So we apply `Nat.add_one_le_of_lt` as such:

```
import Mathlib.Data.Nat.Prime.Defs
theorem prime_factor {n : ℕ} (h : 2 ≤ n) : ∃ (p : ℕ), Nat.Prime p ∧ p ∣ n
    := by sorry
theorem infinite_primes (n : ℕ) : ∃ (p : ℕ), Nat.Prime p ∧ p > n := by
   have : 2 ≤ Nat.factorial n + 1 := by
      rw [Nat.add_one_le_add_one_iff]
      apply Nat.add_one_le_of_lt
```

Once again, the infoview updates as expected:

```
n : ℕ
⊢ 0 < n.factorial
```

As mentioned above, this is the theorem `Nat.factorial_pos` in mathlib, so we use the `exact` tactic which gives that we have already proved something which is exactly the goal. So, our proof ends up as

```
import Mathlib.Data.Nat.Prime.Defs
theorem prime_factor {n : ℕ} (h : 2 ≤ n) : ∃ (p : ℕ), Nat.Prime p ∧ p ∣ n
    := by sorry
theorem infinite_primes (n : ℕ) : ∃ (p : ℕ), Nat.Prime p ∧ p > n := by
   have : 2 ≤ Nat.factorial n + 1 := by
      rw [Nat.add_one_le_add_one_iff]
      apply Nat.add_one_le_of_lt
      exact Nat.factorial_pos n
```

With this, our infoview tells us that our goals are accomplished with a celebratory party popper emoji. But we only accomplished the goal for $2 \leq n! + 1$, and starting a new line then updates our infoview to our infinitely many primes theorem along with our brand new lemma that we just proved in the assumptions, with the default name `this`:

```
n : ℕ
this : 2 ≤ n.factorial + 1
⊢ ∃ p, Nat.Prime p ∧ p > n
```

Using our previous `prime_factor` theorem, we can now have that this prime factor exists:

```
import Mathlib.Data.Nat.Prime.Defs
theorem prime_factor {n : ℕ} (h : 2 ≤ n) : ∃ (p : ℕ), Nat.Prime p ∧ p ∣ n
    := by sorry
theorem infinite_primes (n : ℕ) : ∃ (p : ℕ), Nat.Prime p ∧ p > n := by
   have : 2 ≤ Nat.factorial n + 1 := by
      rw [Nat.add_one_le_add_one_iff]
      apply Nat.add_one_le_of_lt
      exact Nat.factorial_pos n
   have := prime_factor this
```

Which gives us the `prime_factor` theorem applied to $n! + 1$ in the assumptions, replacing the name `this` (we can give lemmas names with `have` and avoid the naming conflict, but it is not required, especially if we are not going to use the lemma more than once):

```
n : ℕ
this† : 2 ≤ n.factorial + 1
this : ∃ (p : ℕ), Nat.Prime p ∧ p ∣ n.factorial + 1
⊢ ∃ p, Nat.Prime p ∧ p > n
```

Now this is an existentially quantified statement, which is still not exactly what we want. We want the prime p that divides $n! + 1$, and there are a few options in Lean for accessing this prime from the existential statement. The simplest option is to use the `Exists.elim` theorem, which allows us to prove any proposition b from $\exists x, p(x)$ and $\forall (a : \alpha), p(a) \to b$ for some proposition p. We just proved the $\exists x, p(x)$ part,

which is now named `this`, so adding `apply Exists.elim this` to our proof will have us prove the $\forall(a : \alpha), p(a) \to b$ part. Our infoview after adding the apply statement reflects that exactly:

```
n : ℕ
this† : 2 ≤ n.factorial + 1
this : ∃ (p : ℕ), Nat.Prime p ∧ p ∣ n.factorial + 1
⊢ ∀ (a : ℕ), Nat.Prime p ∧ p ∣ n.factorial + 1 → ∃ p, Nat.Prime p ∧ p >
    n
```

Our goal is now a universally quantified conditional, and the way to introduce these into our assumptions is with the `intro` tactic. If we have a goal of the form `P → Q`, then `intro h` will give us the assumption `h : P` and change our goal to `Q`. This also can be used with introducing universally quantified variables into our assumptions as well. So we introduce the variable p and the condition into our assumptions with `intro p ph`, which brings us the infoview:

```
n : ℕ
this† : 2 ≤ n.factorial + 1
this : ∃ (p : ℕ), Nat.Prime p ∧ p ∣ n.factorial + 1
p : ℕ
ph : Nat.Prime p ∧ p ∣ n.factorial + 1
⊢ ∃ p, Nat.Prime p ∧ p > n
```

The equivalent tactic to `intro` for working with existentially quantified goals is `use`, which lets us use some specific object of the correct type as the object that satisfies the proposition we want to prove. So, adding `use p` to our proof, our infoview is:

```
n : ℕ
this† : 2 ≤ n.factorial + 1
this : ∃ (p : ℕ), Nat.Prime p ∧ p ∣ n.factorial + 1
p : ℕ
ph : Nat.Prime p ∧ p ∣ n.factorial + 1
⊢ Nat.Prime p ∧ p > n
```

We want to prove an and statement, so we can `apply And.intro`, which splits our infoview into two different goals, one for each part of the and statement we want to prove:

```
case h.left
n : ℕ
this† : 2 ≤ n.factorial + 1
this : ∃ (p : ℕ), Nat.Prime p ∧ p ∣ n.factorial + 1
p : ℕ
ph : Nat.Prime p ∧ p ∣ n.factorial + 1
⊢ Nat.Prime p

case h.right
n : ℕ
this† : 2 ≤ n.factorial + 1
this : ∃ (p : ℕ), Nat.Prime p ∧ p ∣ n.factorial + 1
p : ℕ
ph : Nat.Prime p ∧ p ∣ n.factorial + 1
⊢ p > n
```

We already have that `Nat.Prime p`, it is the first half of the and statement `ph`. We can access this with `ph.left`, so the tactic `exact ph.left` closes the first goal, leaving us with just the second goal of `p > n`.

Note that this was a fair amount of work to achieve what was done completely automatically in the first sentence of the informal proof. But all of these steps really do end up being necessary when it comes to explaining one's reasoning in the fullest possible detail. This is the demand made by proof assistant languages, though there do exist more efficient, yet less explanatory, methods for doing everything we have done above. What is important now is that we have finished formalizing the first sentence of the proof and can move onto the second, which states that "If $p \leq n$, then it divides $n!$. This is beginning a proof by contradiction, which is supported by the tactic `by_contra`. With a goal P, `by_contra` will give us $\neg P$ in our assumptions and changes our goal to `False`:

```
case h.right
n : ℕ
this† : 2 ≤ n.factorial + 1
this : ∃ (p : ℕ), Nat.Prime p ∧ p ∣ n.factorial + 1
p : ℕ
ph : Nat.Prime p ∧ p ∣ n.factorial + 1
np : ¬p > n
⊢ False
```

The statement $\neg p > n$ is equivalent to $p \leq n$, and the tactic `push_neg at np` is able to turn `np : ¬p > n` in our assumptions into `np : p ≤ n`, which is the contradictory assumption given in the informal proof. The next claim made in the proof is that $p \mid n!$, which we will begin a proof of with:

```
have : p ∣ Nat.factorial n := by
```

A quick search in mathlibb finds the theorem `Nat.dvd_factorial`, which states that if $0 < m$ and $m \leq n$, then $m \mid n!$. This conclusion matches the goal we want, so we apply this theorem:

```
have : p ∣ Nat.factorial n := by
```

```
apply Nat.dvd_factorial
```

This brings our infoview to two different goals, corresponding to $0 < p$ and $p \leq n$. Since p is prime, of course $0 < p$, which is represented by the theorem `Nat.Prime.pos`, so that `exact Nat.Prime.pos ph.left` closes the first goal, and the second condition $p \leq n$ is our assumption `np`, so `exact np` closes the second goal. Then `this : p ∣ Nat.factorial n` is added to our assumptions.

The reason why $p \mid n! + 1$ together with $p \mid n!$ implies that $p \mid 1$ is because if $k \mid m$ and $k \mid n$ then $k \mid m - n$. This is the theorem `Nat.dvd_sub`, which importantly also requires a proof of $n \leq m$. So we start our proof of $p \mid 1$ with the usual `have`:

```
have : p ∣ 1 := by
```

But directly using `apply Nat.dvd_sub` here gives us an error as this theorem requires our goal to be of the form `k ∣ n - m`, which it does not match yet. So we can backtrack and instead first prove that `p ∣ Nat.factorial n + 1 - Nat.factorial n`, which we can use `apply Nat.dvd_sub` on, which splits our infoview into three goals for each condition. The first thing we then have to prove is `n.factorial ≤ n.factorial + 1`. This is easy from basic arithmetic theorems, but I would like to use this opportunity to introduce the powerful proof automation tactics that exist in Lean.

One particular tactic that closes this goal is `linarith`. This tactic is designed to automatically calculate and close goals that are based on linear systems of equalities and inequalities. Another tactic that can close this goal is `simp`. This tactic is the "simplifier," and works by trying to apply many tagged lemmas and theorems previously proven at the same time to reduce the goal to a simplified state. There are many alternate ways to use `simp` as well, such as `simp?` which will print the list of theorems that it used to do the simplification. There are many more of these automation tactics designed for different contexts, such as `ring`, which can prove ring identities, `aesop`, which is a highly powerful proof generator that can search for a

very wide range of proofs, and so on. For now, I will import `linarith` and use it to close the $n! \leq n! + 1$ goal at hand.

The remaining two goals of `p ∣ Nat.factorial n + 1` and `n.factorial` are closed by `exact ph.right` and `exact this`.

Going back to our proof of `p ∣ 1`, we can now easily rewrite `p ∣ Nat.factorial n + 1 - Nat.factorial n` to `p ∣ 1` with `Nat.add_sub_cancel_left`.

At this point we now have that p is a prime and $p \mid 1$, and any mathematician would be convinced of the contradiction by now, but we still have some work to do to actually provide the proof of `False` that is required. There are a few ways to do this, but I chose to rewrite `p ∣ 1` to `p = 1` with `Nat.dvd_one`, and then used the theorem `Nat.Prime.ne_one` to get that `p ≠ 1`, at which point the `contradiction` tactic was able to automatically close the goal. Here is what the final proof looks like:

```
theorem infinite_primes (n : ℕ) : ∃ (p : ℕ), Nat.Prime p ∧ p > n := by
  have : 2 ≤ Nat.factorial n + 1 := by
    rw [Nat.add_one_le_add_one_iff]
    apply Nat.add_one_le_of_lt
    exact Nat.factorial_pos n
  have := prime_factor this
  apply Exists.elim this
  intro p ph
  use p
  apply And.intro
  exact ph.left
  by_contra np
  push_neg at np
  have : p ∣ Nat.factorial n := by
    apply Nat.dvd_factorial
    exact Nat.Prime.pos ph.left
```

```
    exact np
  have : p ∣ Nat.factorial n + 1 - Nat.factorial n := by
    apply Nat.dvd_sub
    linarith
    exact ph.right
    exact this
  have : p ∣ 1 := by
    rw [Nat.add_sub_cancel_left] at this
    exact this
  rw [Nat.dvd_one] at this
  have := Nat.Prime.ne_one ph.left
  contradiction
```

This is a highly verbose proof and far from the most optimal one that can be written in Lean, but it pretty exactly matches the line of reasoning used in the informal proof given at the beginning. This uses only some of the most fundamental tactics, and a whole wealth of various tactics can be seen in the mathlib tactics documentation. The proof of `Nat.exists_infinite_primes` in mathlib follows a very similar argument, but in only 9 lines after a lot of optimization and using the full range of tools afforded by mathlib. I will leave the proof of `prime_factor` as an exercise, with the hint to use the `by_cases` tactic to start a proof by cases on whether n is prime and to look up the documentation for the `induction'` tactic and use it with `Nat.strong_induction_on`.

1.2 Magma theory

What we have seen now with this proof of infinitely many primes is using a proof assistant to verify our human mathematics. This is currently the most common use

case for Lean and other proof assistants, but it clearly makes our human mathematics much more involved and adds layers of difficulty. There are, however, some types of mathematics that this sort of computerized system enables and makes significantly easier. As a particularly interesting example that has seen some work recently, I would like to demonstrate a bit of the theory of magmas.

1.2.1 Definition and examples

Definition 1. *A magma is a set M equipped with a binary operation $\diamond : M \times M \to M$.*

The only requirement on a binary operation in a magma is that it is closed. Of course, all of the familiar algebraic examples of groups, vector spaces, etc. form magmas. However, we are given almost nothing to work with at the level of generality of magmas. The number of possible magmas that can be constructed is far too large to have any clear structure. What is interesting about magmas is that they allow us a kind of 'playground' of binary relations. We can arbitrarily restrict the binary operation on a magma with laws.

Example 1. *A magma with an associative law is a magma for which $\forall x, y, z \in M, (x \diamond y) \diamond z = x \diamond (y \diamond z)$. In this case, the magma is called an associative magma.*

The associative law is one highly familiar to us from algebra, but still there are a huge amount of conceivable laws on magmas that are not necessarily familiar to us at all. An example due to Johan Commelin [Com24] is

Example 2. *Let S be the set of all strings on a countable alphabet A. For any two $x, y \in S$, define*

$$x \diamond y = \begin{cases} y & \text{if } x = y \text{ or } x \text{ ends with } yyy \\ xy & \text{otherwise.} \end{cases}$$

This forms a magma on S as the concatenation of two strings of some alphabet is also a string in that alphabet. This magma satisfies the following law:

Theorem 2. *If S is the magma defined in Example 2, then*

$$\forall x, y \in S, x = (((y \diamond x) \diamond x) \diamond x) \diamond x.$$

Proof. Let $x, y \in S$ be strings. By the construction of $\diamond$, we have $x \diamond x = x$. Assume that either $x = y$ or y ends with xxx. Then $y \diamond x = x$. Therefore

$$\begin{aligned}(((y \diamond x) \diamond x) \diamond x) \diamond x &= ((x \diamond x) \diamond x) \diamond x \\ &= (x \diamond x) \diamond x \\ &= x \diamond x \\ &= x.\end{aligned}$$

Now assume that $x \neq y$ and y does not end with xxx. Then $y \diamond x = yx$. Therefore

$$(((y \diamond x) \diamond x) \diamond x) \diamond x = ((yx \diamond x) \diamond x) \diamond x.$$

If $yx = x$, then we end up with the same proof as the first case, so assume $yx \neq x$. Then

$$((yx \diamond x) \diamond x) \diamond x = (yxx \diamond x) \diamond x.$$

Once again if $yxx = x$, then we end up in the first case, so assume $yxx \neq x$. Then

$$(yxx \diamond x) \diamond x = yxxx \diamond x.$$

We have that $yxxx$ ends with xxx. Therefore, $yxxx \diamond x = x$. Therefore, in all cases, we end up with $(((y \diamond x) \diamond x) \diamond x) \diamond x = x$. □

Here is a law that this magma does not satisfy:

Theorem 3. *If S is the magma defined in Example 2, then we do not have $\forall x, y, z \in S, x = (((y \diamond z) \diamond x) \diamond x) \diamond x$.*

Proof. Let $A = \mathbb{N}$, and consider the strings $x = 1, y = 2$, and $z = 3$. Then

$$\begin{aligned}(((y \diamond z) \diamond x) \diamond x) \diamond x &= (((2 \diamond 3) \diamond 1) \diamond 1) \diamond 1 \\ &= ((23 \diamond 1) \diamond 1) \diamond 1 \\ &= (231 \diamond 1) \diamond 1 \\ &= 2311 \diamond 1 \\ &= 23111 \neq 1.\end{aligned}$$

□

What we have shown then is a magma that satisfies one law but not the other, and therefore a counterexample to the implication between those two laws. That is, we have shown that if a magma M satisfies the law $\forall x, y \in M, x = (((y \diamond x) \diamond x) \diamond x) \diamond x$, then we do not necessarily have $\forall x, y, z \in M, x = (((y \diamond z) \diamond x) \diamond x) \diamond x$. It is this perspective that gives us a way to characterize magmas: by implications and non-implications between laws on magmas.

We can specify finite magmas with tables that describe the magma operation. Here is an example on the set $\{0, 1\}$ that is not associative, not commutative, and has no identity element:

$\diamond$	0	1
0	1	1
1	0	1

This magma also corresponds to logical implication on the set of booleans.

1.2.2 Detecting laws on magmas

Definition 2. *A magma M is trivial if $\forall x, y \in M, x = y$.*

In the case where a magma is trivial, the underlying set on the magma must have at most one element. There are a number of laws on magmas which, upon initial inspection, might not appear to describe only a singleton but which are actually equivalent to the trivial magma. When a magma has a law that makes it equivalent to the trivial magma, we call it a trivial law. Here is an example of a trivial law that I formalized [Tao24b]:

Theorem 4. *Let M be a magma. The law $\forall x, y, z \in M, x = y \diamond ((z \diamond x) \diamond (z \diamond z))$ is a trivial law.*

Proof. If M is trivial, then we immediately have $x = y \diamond ((z \diamond x) \diamond (z \diamond z))$ for any $x, y, z \in M$.

Assume that $\forall x, y, z \in M, x = y \diamond ((z \diamond x) \diamond (z \diamond z))$. Let $0, a \in M$. Let $1 = 0 \diamond 0$, and let $2 = 1 \diamond 1$. Substituting $x = z = 0$ into the given law, we get

$$\begin{aligned} &\forall y \in M, 0 = y \diamond ((0 \diamond 0) \diamond (0 \diamond 0)) \\ &\forall y \in M, 0 = y \diamond (1 \diamond 1) \\ &\forall y \in M, 0 = y \diamond 2. \end{aligned} \tag{1.1}$$

Then substituting $z = 1$ into the given law, we also get that

$$\begin{aligned} &\forall x, y \in M, x = y \diamond ((1 \diamond x) \diamond (1 \diamond 1)) \\ &\forall x, y \in M, x = y \diamond ((1 \diamond x) \diamond 2) \\ &\forall x, y \in M, x = y \diamond 0. \end{aligned} \tag{1.2}$$

The deduction from the second line to the final line is given by letting $y = 1 \diamond x$ in (1.1). Therefore, substituting $x = y = 0$ into (1.2), we have $0 = 0 \diamond 0$, and substituting $x = a$ and $y = 0$ into (1.2), we get $a = 0 \diamond 0$, so that $a = 0$. □

We can also use magma homomorphisms to detect if laws hold for magmas.

Definition 3. *For two magmas $(M, \diamond), (N, \cdot)$, a magma homomorphism is a function $f : M \to N$ such that $\forall x, y \in M, f(x \diamond y) = f(x) \cdot f(y)$.*

Then some basic expected algebraic results hold for magma homomorphisms, but also some fail:

Lemma 1. *For three magmas $(M, \diamond), (N, \cdot), (G, \bullet)$, and two magma homomorphisms $f : M \to N$, and $g : N \to G$, we have that $g \circ f : M \to G$ is a magma homomorphism.*

Proof. Let $x, y \in M$. Then

$$\begin{aligned}(g \circ f)(x \diamond y) &= g(f(x \diamond y)) \\ &= g(f(x) \cdot f(y)) \\ &= g(f(x)) \bullet g(f(y)) \\ &= (g \circ f)(x) \bullet (g \circ f)(y).\end{aligned}$$

□

With this, together with the fact that the composition of functions on sets is associative, and the fact that the identity morphism on magmas is trivially a magma homomorphism, we can speak of a category of magmas, **Mag**. There is one particularly interesting property of **Mag** that ends up being important for the Eckmann-Hilton argument [Wik24]:

Lemma 2. *For two magmas $(M_1, \diamond)$, $(M_2, \cdot)$, the Cartesian product $(M_1 \times M_2, \star)$ with $\star$ defined component-wise is a product in* **Mag**.

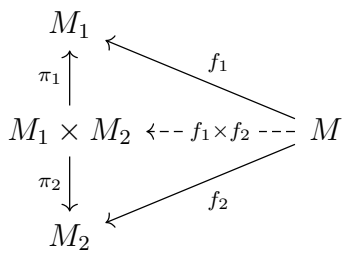
Proof. Let $(M_1, \diamond)$ and $(M_2, \cdot)$ be magmas, and define $(M_1 \times M_2, \star)$ with the binary operation $\star$ on $M_1 \times M_2$ defined component-wise: $(x, y) \star (z, w) = (x \diamond z, y \cdot w)$. Then we have projection morphisms $\pi_1 : M_1 \times M_2 \to M_1$ given by $(x, y) \mapsto x$,

and $\pi_2 : M_1 \times M_2 \to M_2$ given by $(x, y) \mapsto y$. Importantly, these are magma homomorphisms as

$$\begin{aligned}\pi_1((x_1, y_1) \star (x_2, y_2)) &= \pi_1(x_1 \diamond x_2, y_1 \cdot y_2) \\ &= x_1 \diamond x_2 \\ &= \pi_1(x_1, y_1) \diamond \pi_1(x_2, y_2),\end{aligned}$$

and similarly with π_2.

Let $(M, *)$ be any magma and consider any pair of magma homomorphisms $f_1 : M \to M_1$ and $f_2 : M \to M_2$. Then consider the function $f_1 \times f_2 : M \to M_1 \times M_2$, given by $x \mapsto (f_1(x), f_2(x))$. Then we need to show that the following diagram commutes uniquely:



We have that

$$\begin{aligned}(\pi_1 \circ (f_1 \times f_2))(x) &= \pi_1((f_1 \times f_2)(x)) \\ &= \pi_1(f_1(x), f_2(x)) \\ &= f_1(x),\end{aligned}$$

and

$$\begin{aligned}(\pi_2 \circ (f_1 \times f_2))(x) &= \pi_2((f_1 \times f_2)(x)) \\ &= \pi_2(f_1(x), f_2(x)) \\ &= f_2(x).\end{aligned}$$

Therefore $\pi_1 \circ (f_1 \times f_2) = f_1$ and $\pi_2 \circ (f_1 \times f_2) = f_2$. To verify that $f_1 \times f_2$ is a

magma homomorphism, we have:

$$\begin{aligned}(f_1 \times f_2)(x * y) &= (f_1(x * y), f_2(x * y)) \\ &= (f_1(x) \diamond f_1(y), f_2(x) \cdot f_2(y)) \\ &= (f_1(x), f_2(x)) \star (f_1(y), f_2(y)) \\ &= (f_1 \times f_2)(x) \star (f_1 \times f_2)(y).\end{aligned}$$

Then we lastly need to show that $f_1 \times f_2$ is the unique magma homomorphism such that $\pi_1 \circ (f_1 \times f_2) = f_1$ and $\pi_2 \circ (f_1 \times f_2) = f_2$. Assume that there is another magma homomorphism $f : M \to M_1 \times M_2$ such that $\pi_1 \circ f = f_1$ and $\pi_2 \circ f = f_2$. Then we have $\pi_1(f(x)) = \pi_1((f_1 \times f_2)(x))$ and $\pi_2(f(x)) = \pi_2((f_1 \times f_2)(x))$, and therefore if $f(x) = (x_1, y_1)$, we have $x_1 = f_1(x)$ and $y_1 = f_2(x)$, so that $f(x) = (f_1(x), f_2(x)) = (f_1 \times f_2)(x)$. □

Similarly, we also have coproducts in **Mag**, from the general construction given here [Bra]:

Theorem 5. *For two magmas $(M_1, \diamond)$, $(M_2, \cdot)$, the free product $(M_1 \coprod M_2, \star)$ is a coproduct in* **Mag**.

Proof. First, we will construct the free product of M_1 and M_2. For M_1 and M_2 considered as sets, let $M_1 + M_2$ denote the disjoint union of M_1 and M_2. Let $(M_1 + M_2)^s$ be the set of finite sequences with entries in $M_1 + M_2$. This is a magma with the operation defined by concatenation: $(a_1, \ldots, a_n) * (b_1, \ldots, b_m) = a_1, \ldots, a_n, b_1, \ldots, b_m$. An operation on $(M_1 + M_2)^s$ is an *elementary reduction* if it is of the following form:

$$a_1, \ldots, a_i, a_{i+1}, \ldots, a_n \mapsto a_1, \ldots, a_i a_{i+1}, \ldots, g_n \text{ if } a_i, a_{i+1} \in M_1 \text{ or } a_1, a_{i+1} \in M_2.$$

This kind of operation takes two adjacent elements in a sequence in $(M_1 + M_2)^s$ and combines them according to the magma operation of M_1 or M_2 if the two adjacent elements are in the same magma. Now define a relation $\sim$ on $(M_1 + M_2)^s$ for any

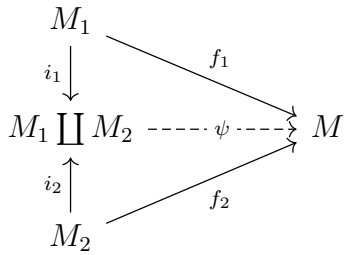
two $W_1, W_2 \in (M_1 + M_2)^s$ by $W_1 \sim W_2$ iff W_1 can be transformed to W_2 by a finite number of elementary reductions or their inverses (which would be factoring the element $a_i a_{i+1}$ in a sequence into two elements a_i, a_{i+1}).

We have $W_1 \sim W_1$ trivially. If $W_1 \sim W_2$, so that there is a sequence of elementary transformations from W_1 to W_2, then we can replace each elementary reduction in the sequence with its inverse to go from W_2 to W_1, and therefore $W_2 \sim W_1$, and also vice versa. If $W_1 \sim W_2$ and $W_2 \sim W_3$, then we can follow the sequence of transformations from W_1 to W_2 and then from W_2 to W_3 to transform W_1 to W_3, and therefore $W_1 \sim W_3$. Therefore, $\sim$ is an equivalence relation, so we can quotient out $(M_1 + M_2)^s$ by $\sim$ to get the *free product* of M_1 and M_2: $(M_1 + M_2)^s/\sim = M_1 \coprod M_2$, where the magma operation is given by $[a] \star [b] = [ab]$, where ab is the sequence ab in $M_1 + M_2$. We then have the natural inclusion morphisms $i_1 : M_1 \to M_1 \coprod M_2$ and $i_2 : M_2 \to M_1 \coprod M_2$ given by $a \mapsto [a]$, the set of all finite sequences in $M_1 + M_2$ that can be reduced to a. Importantly, these are magma homomorphisms, as $i_1(x \diamond y) = [x \diamond y]$, and the sequence $x \diamond y$ can be reduced to the sequence xy through the inverse mapping, so that $[x \diamond y] = [xy] = [x] \star [y] = i_1(x) \star i_1(y)$, and similarly with i_2.

Let $(M, *)$ be any magma and consider any pair of magma homomorphisms $f_1 : M_1 \to M$ and $f_2 : M_2 \to M$. Define $\psi : M_1 \coprod M_2 \to M$ as $[a_1, \ldots, a_n] \mapsto f_k(a_1) * \ldots * f_k(a_n)$, for $a_j \in M_k$. Then ψ is a magma homomorphism as

$$\begin{aligned}
\psi(a_1, \ldots, a_n \star b_1, \ldots, b_m) &= \psi(a_1, \ldots, a_n, b_1, \ldots, b_m) \\
&= f_k(a_1) * \ldots * f_k(a_n) * f_k(b_1) * \ldots * f_k(b_m) \\
&= \psi(a_1, \ldots, a_n) * \psi(b_1, \ldots b_m).
\end{aligned}$$

We need to show then that the following diagram commutes uniquely:



The diagram commutes as, for any $a \in M_1$,

$$\begin{aligned}(\psi \circ i_1)(a) &= \psi(i_1(a)) \\ &= \psi([a]) \\ &= f_1(a),\end{aligned}$$

and for any $b \in M_2$,

$$\begin{aligned}(\psi \circ i_2)(b) &= \psi(i_2(b)) \\ &= \psi([b]) \\ &= f_2(b).\end{aligned}$$

Assume there is another $\Psi : M_1 \coprod M_2 \to M$ such that $\Psi \circ i_1 = f_1$ and $\Psi \circ i_2 = f_2$. Therefore $\Psi \circ i_1 = \psi \circ i_1$, so that $\Psi([a]) = \psi([a])$ for any $a \in M_1$, and therefore $\Psi = \psi$.

□

Unital magmas are a particularly important class of magmas:

Definition 4. *A magma M is unital if there exists an $e \in M$ such that $\forall x \in M, e \diamond x = x \diamond e = x$.*

Example 3. *Let $(M, \diamond)$, $(N, \cdot)$ be unital magmas with identities $e_0 \in M$ and $e_1 \in N$. Let $f : M \to N$ be a magma homomorphism. Then we do not necessarily have $f(e_0) = e_1$.*

Proof. Consider $f : (\mathbb{Z}, \times) \to (\mathbb{Z}, \times)$ defined by $x \mapsto 0$. Then

$$\begin{aligned} f(x \times y) &= 0 \\ &= 0 \times 0 \\ &= f(x) \times f(y), \end{aligned}$$

so that f is a magma homomorphism. But the multiplicative identity in $(\mathbb{Z}, \times)$ is 1, and $f(1) = 0$. □

Unfortunately, magma homomorphisms rarely *directly* carry over properties such as triviality and unitality. For example:

Example 4. *Consider a trivial magma $(M, \diamond) = \{\star\}$ and any magma $(N, \cdot)$. If $f : M \to N$ is a magma homomorphism, then N is not necessarily trivial.*

Proof. Consider $N = \{1, 2\}$, with some closed operation $\cdot$ that has $1 \cdot 1 = 1$. Then define $f : M \to N$ by $\star \mapsto 1$. Then we have, for any $x, y \in M$,

$$\begin{aligned} f(x \diamond y) &= f(\star) \\ &= 1 \\ &= 1 \cdot 1 \\ &= f(\star) \cdot f(\star) \\ &= f(x) \cdot f(y). \end{aligned}$$

Therefore f is a magma homomorphism, but N is not trivial. □

The above example allows us to see properties about individual elements of N, that is, if $f : M \to N$ is a magma homomorphism with $f(\star) = y$, then it must be that $y = y \cdot y$, but this is much weaker than showing the complete law $\forall y \in N, y = y \cdot y$. We also trivially see that if f is surjective, then N must be trivial, since then N can have at most one element.

There are, however, more *indirect* ways to detect information about magmas with homomorphisms, usually requiring a bit more structure on the magmas. This theorem is known as the *Eckmann-Hilton argument* (and this is actually more general than the original argument):

Theorem 6. *Let $(M, \diamond)$ be a unital magma with identity element e_0. Then consider a unital magma $(M, \cdot)$ given by an operation $\cdot$ such that $\cdot : (M, \diamond) \times (M, \diamond) \to (M, \diamond)$ is a magma homomorphism, with identity element e_1. Then we have the following:*

1. *The exchange law holds: $\forall x, y, z, w \in M, (x \cdot y) \diamond (z \cdot w) = (x \diamond z) \cdot (y \diamond w)$;*
2. *$e_0 = e_1$;*
3. *$\forall x, y \in M, x \diamond y = x \cdot y$;*
4. *$(M, \diamond)$ is commutative (and therefore also $(M, \cdot)$ is);*
5. *$(M, \diamond)$ is associative (and therefore also $(M, \cdot)$ is).*

Proof. (1.) Recall from Lemma 2 that the binary operation $\star$ on $(M, \diamond) \times (M, \diamond)$ is defined component-wise, so that $(x, y) \star (z, w) = (x \diamond z, y \diamond w)$. Then since $\cdot$ is a magma homomorphism, we have

$$\begin{aligned} \cdot((x, y) * (z, w)) &= \cdot(x \diamond z, y \diamond w) \\ &= \cdot(x, y) \diamond \cdot(z, w). \end{aligned}$$

Therefore $(x \diamond z) \cdot (y \diamond w) = (x \cdot y) \diamond (z \cdot w)$.

(2.):

$$\begin{aligned} e_0 &= e_0 \diamond e_0 \\ &= (e_1 \cdot e_0) \diamond (e_0 \cdot e_1) \\ &= (e_1 \diamond e_0) \cdot (e_0 \diamond e_1) \\ &= e_1 \cdot e_1 \\ &= e_1. \end{aligned}$$

(3.):

$$\begin{aligned} x \diamond y &= (x \cdot e_1) \diamond (e_1 \cdot y) \\ &= (x \diamond e_1) \cdot (e_1 \diamond y) \\ &= (x \diamond e_0) \cdot (e_0 \diamond y) \\ &= x \cdot y. \end{aligned}$$

Now that we have these two results, we'll write the one identity in M as e, and write $x \diamond y = x \cdot y = xy$. Then the exchange law we have states that $(ab)(cd) = (ac)(bd)$.

(4.):

$$\begin{aligned} xy &= (ex)(ye) \\ &= (ey)(xe) \\ &= yx. \end{aligned}$$

(5.):

$$\begin{aligned} (xy)z &= (xy)(ez) \\ &= (xe)(yz) \\ &= x(yz). \end{aligned}$$

□

This theorem ends up having important consequences, such as in the proof of the commutativity of higher homotopy groups, and secretly it is a statement about monoidal categories [Yua12], but for our purposes it shows how constructing certain magma homomorphisms gives us laws on magmas.

1.2.3 Minimal axiomatizations and the usefulness of Lean

Generally speaking, proofs of magma implications tend to get quite large and unwieldy to do with pen and paper when we don't have nice constructions like the above

available, which is more often than not. As a demonstration, we can sometimes prove the equivalence between a single magma law and a more sophisticated algebraic structure. This is known generally as a 'minimal axiomatization' of the algebraic structure. Here is an interesting example from [MP74]:

Theorem 7. *M is a magma such that $\forall x, y, z \in M, x = (y \diamond z) \diamond (y \diamond (x \diamond z))$ if and only if M is an Abelian group of exponent 2.*

Proof. Let M be an Abelian group of exponent 2, so that 2 is the smallest positive natural number z such that $m^z = I$ for all $m \in M$. Then we have for any $x, y, z \in M$,

$$\begin{aligned}(y \diamond z) \diamond (y \diamond (x \diamond z)) &= (y \diamond y) \diamond (x \diamond (z \diamond z)) \\ &= e \diamond (x \diamond e) \\ &= x.\end{aligned}$$

Now let M be a magma such that $\forall x, y, z \in M, x = (y \diamond z) \diamond (y \diamond (x \diamond z))$. Substituting in $y = z = x \diamond y$, we get

$$\forall x, y \in M, x = ((x \diamond y) \diamond (x \diamond y)) \diamond ((x \diamond y) \diamond (x \diamond (x \diamond y))). \tag{1.3}$$

Similarly, substituting in $z = y$ and $y = x$, we get

$$\forall x, y, \in M, x = (x \diamond y) \diamond (x \diamond (x \diamond y)).$$

Observe then that this is the right half of (1.3), so therefore we have

$$\forall x, y \in M, x = ((x \diamond y) \diamond (x \diamond y)) \diamond x.$$

Changing variables here, we have

$$\forall y, z \in M, y = ((y \diamond z) \diamond (y \diamond z)) \diamond y. \tag{1.4}$$

Then substituting in $x = (y \diamond z) \diamond (y \diamond z)$, $y = x$, and $z = y$ into the assumed law, we have

$$\forall x, y, z \in M, (y \diamond z) \diamond (y \diamond z) = (x \diamond y) \diamond (x \diamond ((y \diamond z) \diamond (y \diamond z)) \diamond y).$$

We see then that this includes the term $((y \diamond z) \diamond (y \diamond z)) \diamond y$, which we found to be equal to y with (1.4), so we have

$$\forall x, y, z \in M, (y \diamond z) \diamond (y \diamond z) = (x \diamond y) \diamond (x \diamond y).$$

Changing variables here, we have

$$\forall y, z, w \in M, (y \diamond z) \diamond (y \diamond z) = (z \diamond w) \diamond (z \diamond w).$$

Therefore by transitivity we have

$$\forall x, y, z, w \in M, (x \diamond y) \diamond (x \diamond y) = (z \diamond w) \diamond (z \diamond w).$$

Now substituting in $x = x \diamond x$, $y = x \diamond (x \diamond x)$, $z = y \diamond y$, and $w = y \diamond (y \diamond y)$ here, we have

$$\begin{aligned} \forall x, y \in M, &((x \diamond x) \diamond (x \diamond (x \diamond x))) \diamond ((x \diamond x) \diamond (x \diamond (x \diamond x))) \\ &= ((y \diamond y) \diamond (y \diamond (y \diamond y))) \diamond ((y \diamond y) \diamond (y \diamond (y \diamond y))). \end{aligned} \tag{1.5}$$

Our given law for $x = y = z$ gives us also that

$$\forall x \in M, x = (x \diamond x) \diamond (x \diamond (x \diamond x)), \tag{1.6}$$

and similarly

$$\forall y \in M, y = (y \diamond y) \diamond (y \diamond (y \diamond y)).$$

So therefore (1.5) reduces to

$$\forall x, y \in M, x \diamond x = y \diamond y.$$

For any $y \in M$, let $e = y \diamond y$. Therefore we have

$$\forall x \in M, x \diamond x = e. \tag{1.7}$$

Substituting this back into (1.6), we get

$$\forall x \in M, x = e \diamond (x \diamond e), \tag{1.8}$$

and substituting in $y = z = e$ into the given law, we have

$$\forall x \in M, x = (e \diamond e) \diamond (e \diamond (x \diamond e)),$$

so that by (1.7),

$$\forall x \in M, x = e \diamond (e \diamond (x \diamond e)).$$

Observing then that the right half of this equation is equal to x by (1.8), we have

$$\forall x \in M, x = e \diamond x. \tag{1.9}$$

Then applying (1.8) to (1.9), we get

$$\forall x \in M, x = x \diamond e. \tag{1.10}$$

Moving on then, taking $z = e$ in the given law, we have

$$\forall x, y \in M, x = (y \diamond e) \diamond (y \diamond (x \diamond e)),$$

so that

$$\forall x, y \in M, x = y \diamond (y \diamond x). \tag{1.11}$$

Similarly taking $y = I$, we get

$$\forall x, z \in X, x = z \diamond (x \diamond z). \tag{1.12}$$

Substituting $z = x \diamond y$ into (1.12), we get

$$\forall x, y \in M, x = (x \diamond y) \diamond (x \diamond (x \diamond y)),$$

and using (1.11) on $x \diamond (x \diamond y)$, we get

$$\forall x, y \in M, x = (x \diamond y) \diamond y.$$

From this, we have

$$\forall x, y \in M, y \diamond x = y \diamond ((x \diamond y) \diamond y).$$

Substituting in $x = x \diamond y$ and $z = y$ into (1.12) then, we have that

$$\forall x, y \in M, x \diamond y = y \diamond ((x \diamond y) \diamond y),$$

and therefore by transitivity

$$\forall x, y \in M, x \diamond y = y \diamond x. \tag{1.13}$$

Lastly, substituting in $x = x \diamond (y \diamond z)$ and $z = x$ into the given law, we have

$$\forall x, y, z \in M, x \diamond (y \diamond z) = (y \diamond x) \diamond (y \diamond ((x \diamond (y \diamond z)) \diamond x)).$$

Therefore, by repeated applications of (1.11) and (1.13), we have

$$\begin{aligned} \forall x, y, z \in M, (y \diamond x) \diamond (y \diamond ((x \diamond (y \diamond z)) \diamond x)) &= (x \diamond y) \diamond (y \diamond ((x \diamond (y \diamond z)) \diamond x)) \\ &= (x \diamond y) \diamond (y \diamond (x \diamond (x \diamond (y \diamond z)))) \\ &= (x \diamond y) \diamond (y \diamond (y \diamond z)) \\ &= (x \diamond y) \diamond z. \end{aligned}$$

Therefore by transitivity,

$$\forall x, y \in M, x \diamond (y \diamond z) = (x \diamond y) \diamond z. \tag{1.14}$$

Therefore in all, (1.14) gives us that M is associative, (1.13) gives us that M is commutative, (1.9) and (1.10) give us that e is an identity element in M, and (1.7) gives us that each element is its own inverse (and therefore also M is of exponent 2). Therefore in all M is an Abelian group of exponent 2. □

This leads us to the question: for which other structures does a minimal axiomatization exist? Or, even more generally, can we characterize every single magma in

terms of their magma implications? As we have seen from how difficult these proofs can be with pen and paper, the answer is "not in any reasonable amount of time". This is where proof assistants come in as an incredibly helpful tool. Terence Tao has been leading the collaborative equational theories project using the Lean proof assistant to answer exactly this question of characterizing every magma with laws that contain at most four instances of the binary operation [Tao24a]. This project has been massively successful, with $22,028,804$ implications having been proven as of November 26, 2024.

The primary reason why Lean allows us to prove tens of millions of implications on magmas that we could never do manually in any remotely reasonable amount of time is because of its strong automation capabilities. For example, the `SimpleRewrites` folder has tens of thousands of proofs generated automatically by trying the exact same proof technique for every single implication and seeing which ones it works for. The most powerful automated technique uses the specialized automated theorem prover Vampire, which was able to do all but 130 of the 22 million possible implications alone. These final 130 implications, all conjectured to be false, seem to require some pretty sophisticated counterexamples, and there has been some interesting work on these troublesome laws.

These automated theorem prover techniques also allow us to find minimal axiomatizations of mathematical structures that might not have been possible to find without them. Here is an example of one that I worked on for the equational theories project:

Theorem 8. *M is a magma such that $\forall x, y, z \in M, x = (y \diamond ((x \diamond y) \diamond y)) \diamond (x \diamond (z \diamond y))$ if and only if M is a Boolean algebra defined in terms of the Sheffer stroke (NAND) operation.*

Proof. Let M be a Boolean algebra, so that we have the binary meet ($\wedge$) and join

$(\vee)$ operations and the unary complement $(\neg)$ operation. Define the Sheffer stroke magma $(M, \diamond)$ as for any $x, y \in M$, $x \diamond y = \neg(x \wedge y)$. Then we have

$$\begin{aligned}
(y \diamond ((x \diamond y) \diamond y)) \diamond (x \diamond (z \diamond y)) &= \neg(\neg(y \wedge \neg(\neg(x \wedge y) \wedge y)) \wedge \neg(x \wedge \neg(z \wedge y))) \\
&= \neg(\neg(y \wedge \neg(\neg x \vee (\neg y \wedge y))) \wedge \neg(x \wedge \neg(z \wedge y))) \\
&= \neg(\neg(y \wedge \neg(\neg x \vee F)) \wedge \neg(x \wedge \neg(z \wedge y))) \\
&= \neg(\neg(y \wedge \neg(\neg x)) \wedge \neg(x \wedge \neg(z \wedge y))) \\
&= \neg(\neg(y \wedge x) \wedge \neg(x \wedge \neg(z \wedge y))) \\
&= (x \wedge y) \vee (x \wedge \neg(z \wedge y)) \\
&= x \wedge (y \vee (x \wedge \neg(z \wedge y))) \\
&= x \wedge (y \vee (\neg z \vee \neg y)) \\
&= x \wedge ((y \vee \neg y) \vee \neg z) \\
&= x \wedge (T \vee \neg z) \\
&= x \wedge T \\
&= x.
\end{aligned}$$

Therefore the required law holds for the Sheffer stroke. The formal proof in Lean for this can be done with only one `simp` command, as seen on line 36 in [RT24a].

Assume M is a magma for which the given law holds. Automated theorem provers can show ([McC+02]) that the given law implies that:

$$\forall x \in M,\ x = (x \diamond x) \diamond (x \diamond x)$$

$$\forall x, y \in M,\ x \diamond x = x \diamond (y \diamond (y \diamond y))$$

$$\forall x, y, z \in M,\ (x \diamond (y \diamond z)) \diamond (x \diamond (y \diamond z)) = ((y \diamond y) \diamond x) \diamond ((z \diamond z) \diamond x).$$

These implications can be found in Lean starting at line 43 in [RT24a], with about 1500 lines of automatically generated Vampire proofs in [Tow24].

It was already known to Sheffer in 1913 ([She13]) that these are sufficient for a Boolean algebra, where the meet and join operations are defined in terms of the

Sheffer stroke: $x \vee y = (x \diamond x) \diamond (y \diamond y)$, and $x \wedge y = (x \diamond y) \diamond (x \diamond y)$. This verification in Lean can be found in [RT24b]. □

These implications would have been close to impossible to verify without computer assistance, so the framework and tools provided by the proof assistant systems involved in this proof were essential for creating this theory in the first place.

Here is how magmas are defined in the equational theories project:

```
class Magma (α : Type _) where
  op : α → α → α
```

One thing we did not learn in the previous section is typeclasses, which is what `Magma` is defined as. Typeclasses provide support for a form of inheritence in Lean. This means that we can prove properties of some typeclass, and then we can show that some other type is an *instance* of that typeclass, and then that type inherits all of the properties of the typeclass that we proved. For example, we can make a typeclass for groups which specifies an operation on a type α, $\alpha \to \alpha \to \alpha$, that is associative, etc., and then we can prove certain properties about groups in general with that typeclass. We can then show that a type with a binary operation, say integers on addition, is a group by proving that that binary operation is associative etc. on that type, and then we can use all the properties that we proved for general groups with that type. Magmas as defined here work the same way, though we *only* specify a binary operation on a type α with no further requirements.

To prove magma implications in Lean, we prove them on the level of the typeclass `Magma` without reference to any particular magma. Here is a formalization of the magma implication in Theorem 4 that I wrote for the equational theories project, where Equation 953 is the law $\forall x, y, z \in M, x = y \diamond ((z \diamond x) \diamond (z \diamond z))$, and Equation 2 is the law $\forall x, y \in M, x = y$:

```
theorem Equation953_implies_Equation2 (G : Type _) [Magma G] (h:
    Equation953 G) : Equation2 G := by
  intro x y
  have znx (z : G) : z ◇ ((x ◇ x) ◇ (x ◇ x)) = x := (h x z x).symm
  have hzzi := h x x (x ◇ x)
  have hyzi := h y x (x ◇ x)
  rw [znx] at hzzi hyzi
  exact hzzi.trans hyzi.symm
```

This is effectively an exact translation of the informal proof given for Theorem 4, though with a lot of compactification done. There are many different kinds of the fully automated proofs done with Lean, but here is a brief excerpt of one that I generated with Vampire for the proof of Theorem 8, demonstrating the complexity of calculation these systems are able to do:

```
  have eq9 (X0 X1 X2 X3 : G) : (X0 ◇ ((X1 ◇ X0) ◇ X0)) = (((X1 ◇ (X2 ◇ X0))
    ◇ (X1 ◇ (X1 ◇ (X2 ◇ X0)))) ◇ ((X0 ◇ ((X1 ◇ X0) ◇ X0)) ◇ (X3 ◇ (X1 ◇ (X2
    ◇ X0))))) := superpose eq7 eq7 -- superposition 7,7, 7 into 7, unify on
    (0).2 in 7 and (0).1.1.2.1 in 7
  have eq10 (X0 X1 X2 X3 : G) : (((X1 ◇ (X2 ◇ X0)) ◇ ((X3 ◇ (X1 ◇ (X2 ◇
    X0))) ◇ (X1 ◇ (X2 ◇ X0)))) ◇ (X3 ◇ X1)) = X3 := superpose eq7 eq7 --
    superposition 7,7, 7 into 7, unify on (0).2 in 7 and (0).1.2.2 in 7
  have eq11 (X0 X1 X2 : G) : (X0 ◇ ((X1 ◇ X0) ◇ X0)) = (((X2 ◇ X0) ◇ (((X0
    ◇ ((X1 ◇ X0) ◇ X0)) ◇ (X2 ◇ X0)) ◇ (X2 ◇ X0))) ◇ X1) := superpose eq7
    eq7 -- superposition 7,7, 7 into 7, unify on (0).2 in 7 and (0).1.2 in 7
```

Chapter 2

Dependent type theory

We have seen a bit about how to work with proof assistants now and what they are capable of doing, but we still don't exactly understand how they work. How does Lean know when the goals have been accomplished, that a proof is valid? To answer this, we have to get down to the logical underpinnings of proof assistants. (Almost) every proof assistant is an implementation of a formal language called *dependent type theory*. This language was designed to be able to express mathematical propositions and their proofs, and it works as a foundation for mathematics that is not set-based. In this chapter, we will first compare set theory to type theory, and then get an understanding of the development of dependent type theory through lambda calculus. From here, we will implement some mathematical objects with dependent type theory to get a truly fundamental perspective on how proof assistants verify math.

2.1 Set theories vs. type theories

Set theory and type theory are both attempts at making mathematical the idea of a "collection" or a "construction," and in the process form a foundation of mathematics.

The primary goal of type theory is to not just be a foundation of mathematics, but to provide a *decidable* foundation of mathematics. By decidable here, we mean that there exists a finite computation which decides whether a formula is a theorem of a logical system. Not every problem in every form of type theory is decidable, but the important one (especially for proof assistants) is that type-checking is decidable. Type-checking is the problem of checking whether a given term has a given type (or in set theoretical language, whether some object is contained in some set). [Geu08] gives a useful example to see the difference here between set theory and type theory. Consider the set theoretical statement

$$3 \in \{n \in \mathbb{N} \mid \forall x, y, z \in \mathbb{N}^+, x^n + y^n \neq z^n\}.$$

In order to decide whether this statement is true, we have to provide a proof that $\forall x, y, z \in \mathbb{N}^+, x^3 + y^3 \neq z^3$. In contrast, a *typing judgment* such as

$$3 + (7 * 8)^5 : \mathsf{Nat}$$

(read as "$3 + (7 * 8)^5$ is of type Nat", or natural number) does not require a proof, but rather represents a decidable computation, in that each of $3, 7, 8$, and 5 are of type Nat, and the operations $+, *$, and exponentiation take objects of type Nat to objects of type Nat, so the result of this computation having been done must also be of type Nat. The computation is done automatically: there is no other possibility for the type of $3 + (7 * 8)^5$ as all of the objects and functions involved act only on Nat. The closest we can get to the set theoretical example given above from a type theory perspective is make terms that are pairs of an $n : \mathsf{Nat}$ and a proof p of $\forall x, y, z : \mathsf{Nat}, x^n + y^n \neq z^n$, as *checking* that p is a proof for that n is a decidable process, whereas *searching* for the proof as required by the set theoretical standard is not. This ends up giving us the biggest difference between set theories and type theories, and the chief reason why type theories are much more amenable for computerization than set theories: set membership is generally undecidable, yet type-checking is decidable.

This leads to another difference, in that sets are extensional, while types are intentional. By extensional, we mean that any set theory will have the axiom of extensionality: two sets are equal if and only if they contain the same elements. For example, the sets

$$\{n \in \mathbb{N} \mid n \text{ is an even prime larger than } 3\}$$

and

$$\{n \in \mathbb{N} \mid n \text{ is an odd prime smaller than } 2\}$$

are equal to each other, as they contain the same elements (namely, no elements). Types, on the other hand, are intentional in that types are equal if they have the same representation. In the above example, the type of natural numbers that are even primes larger than 3 and the type of natural numbers that are odd primes smaller than 2 are not equal to each other, because the method of constructing a term that is a prime larger than 3 is different from constructing a term that is an odd prime smaller than 2: the two types are asking for two different constructions. The rules of construction of the two types are different, so from an intensional perspective, the types are not equal. From a philosophical perspective then, types are ways to collect together objects of the same 'structure'; all terms of a type are all constructed with the same algorithmic procedure.

This intensional perspective has some immediate corollaries that we will be able to get into more detail on later. One particularly interesting one is that each term has a unique type, whereas elements of a set can potentially be members of many sets.

2.2 Untyped lambda calculus

The first step to developing dependent type theory is through the untyped and then simply typed lambda calculi, since dependent type theory is an extension of the simply typed lambda calculus to cover predicate logic with quantifiers. This tracks with the historical development of dependent type theory as well. Much of the following exposition comes from chapter 1 of [HS08], and chapters 1–4 of [SU06].

Lambda calculus is intended to be a minimalist formal system for expressing all possible computations, which in the usual mathematical world is encapsulated by the idea of a function. The grammar of untyped lambda calculus allows us to express the idea of anonymous functions:

Definition 5. *Let $V = \{v_0, v_1, \ldots\}$ be an alphabet, with each $v \in V$ called a variable. Then, define the set Λ of λ-terms inductively with:*

1. *If $v \in V$, then $v \in \Lambda$ (these λ-terms are called atoms);*
2. *If $M, N \in \Lambda$, then $(MN) \in \Lambda$ (called an application);*
3. *If $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$ (called an abstraction).*

At this point, there is no meaning yet for the elements of Λ, we have only specified the valid ways of stringing together elements of V in the language of untyped lambda calculus. Some valid λ-terms are:

1. $\lambda x.x$;
2. $(\lambda x.x)(\lambda y.y)$;
3. $(\lambda xy.xxy)(\lambda x.xy)(\lambda x.xz)$;
4. $(x(\lambda x.(\lambda x.x)))$.

Note the similarity here to the usual anonymous function notation familiar from mathematics: the identity function is written as $x \mapsto x$. The λ-term $\lambda x.x$ is intended to express this exact idea: a bound variable x computes to x. Usually, functions are applied to particular terms, which is what a λ-term like $(\lambda x.x)y$ is intended to express. There isn't really a good way to express this with the anonymous function notation, and we usually write a λ-term like $(\lambda x.x)y$ as $\mathrm{id}(y)$, where id is defined as the function $x \mapsto x$ (and note also that I am intentionally not specifying the domain or codomain of id, as that is not a concept that is yet included in the *untyped* lambda calculus). Evaluating functions, such as $\mathrm{id}(y) = y$, is a fundamental feature of functions in the usual mathematical notation, and there are two fundamental operations defined in lambda calculus that encapsulate these features: α-conversion and β-reduction.

First, we need some important technical definitions that allow us to talk about λ-terms. Note that with the given definition of Λ, we can induct on a λ-term M by considering the three possible cases for M, if it is an atom, application, or abstraction. Thus, we are justified in making a definition such as

Definition 6. *For a λ-term M, the set of free variables $FV(M)$ is defined inductively as:*

1. $FV(x) = \{x\}$;
2. $FV(PQ) = FV(P) \cup FV(Q)$;
3. $FV(\lambda x.P) = FV(P) \setminus \{x\}$.

Intuitively, a free variable of M is a variable that is not bound by any abstraction $\lambda x.P$ that occurs within M. For example, consider this λ-term:

$$P \equiv (\lambda y.yx(\lambda z.y(\lambda a.b)z))vw.$$

To find the free variables of P, we break it down inductively: P is first of all the application of $(\lambda y.yx(\lambda z.y(\lambda a.b)z))v$ to w, so

$$FV(P) = FV((\lambda y.yx(\lambda z.y(\lambda a.b)z))v) \cup FV(w),$$

and $FV(w) = \{w\}$. Then we see that $(\lambda y.yx(\lambda z.y(\lambda a.b)z))v$ is the application of $\lambda y.yx(\lambda z.y(\lambda a.b)z)$ to v, so

$$FV((\lambda y.yx(\lambda z.y(\lambda a.b)z))v) = FV(\lambda y.yx(\lambda z.y(\lambda a.b)z)) \cup FV(v),$$

and $FV(v) = \{v\}$. Then $\lambda y.yx(\lambda z.y(\lambda a.b)z)$ is an abstraction, so that

$$FV(\lambda y.yx(\lambda z.y(\lambda a.b)z)) = FV(yx(\lambda z.y(\lambda a.b)z)) \setminus \{y\}.$$

Repeating this process, we ultimately end up with

$$FV(yx(\lambda z.y(\lambda a.b)z)) = \{y\} \cup \{x\} \cup (\{y\} \cup (\{b\} \setminus \{a\})) \setminus \{z\},$$

so that ultimately

$$FV(\lambda y.yx(\lambda z.y(\lambda a.b)z)) = \{x, y, b\} \setminus \{y\} = \{x, b\},$$

and therefore

$$FV(P) = \{x, b\} \cup \{v\} \cup \{w\} = \{x, b, v, w\}.$$

This matches with our intuitive idea of a free variable, as none of x, b, v or w are bounded by any abstraction in P.

For a formal system of functions, we want to have some way of evaluating a function for some value. The first technical step needed for this is an algorithmic way to substitute in terms for free variables. The following definition of substitution might seem a bit strange, but ultimately ends up being necessary to cover for a lot of possible side-cases that I will highlight.

Definition 7. *Let M, N, x be λ-terms. Then define the substitution of N for every free occurrence of x in M, denoted by $[N/x]M$, by induction on M with*

1. $[N/x]x = N$;
2. $[N/x]a = a$ *for any atom* $a \neq x$;
3. $[N/x](PQ) = ([N/x]P[N/x]Q)$;
4. $[N/x](\lambda x.P) = \lambda x.P$;
5. $[N/x](\lambda y.P) = \lambda y.P$ *for* $x \notin FV(P)$;
6. $[N/x](\lambda y.P) = \lambda y.[N/x]P$ *for* $x \in FV(P)$ *and* $y \notin FV(N)$;
7. $[N/x](\lambda y.P) = \lambda z.[N/z][z/y]P$ *for* $x \in FV(P)$ *and* $y \in FV(N)$.

The first two parts are obvious as substituting in N for x in the term x results in just N, and substituting in N for x in the term a results in a since there is no x to substitute for, and 3 works similarly. For 4, the term $\lambda x.P$ has $x \notin FV(P)$, so that there is no free occurrence of x in P, and therefore we cannot substitute N for any free occurrence of x, and we end up with just $\lambda x.P$.

For 5–7, we end up breaking on cases of whether $x \in FV(P)$ and whether $y \in FV(N)$. If $x \notin FV(P)$, then $[N/x](\lambda y.P) = \lambda y.P$ as again there is nothing to substitute N for in $\lambda y.P$. If $x \in FV(P)$, then we have to consider whether $y \in FV(N)$. For example, consider the substitution $[w/x](\lambda y.x)$. We reasonably expect this substitution to evaluate to $[w/x](\lambda y.x) = \lambda y.w$, as x is free in $\lambda y.x$. But then if we were to evaluate $[w/x](\lambda w.x)$ in the same way, we'd end up with $\lambda w.w$, which is an identity function, and not a constant function like $\lambda w.x$ started off as. In all then it is important to consider whether $y \in FV(N)$ to avoid name clashes like this that give unexpected results, so it makes most sense to evaluate this as in 7.

With this tool of substitution defined, we can now define the two fundamental operations of lambda calculus: α-conversion and β-reduction.

Definition 8. *Let a λ-term P contain an occurrence of $\lambda x.M$, and let $y \notin FV(M)$. Then an α-conversion in P is the substitution*

$$[(\lambda y.[y/x]M)/(\lambda x.M)]P.$$

Intuitively, α-conversion is a change of variables from $\lambda x.M$ to $\lambda y.M$, where a bound variable x is formally changed to another name y. We write $P_0 =_\alpha P_1$ if there is a finite amount of α-conversions that, when applied successively, results in P_1. For example, we have the α-conversion

$$\begin{aligned}[(\lambda y_0.[y_0/y](\lambda y.xy))/(\lambda y.xy)](\lambda x.(\lambda y.xy)) &= [(\lambda y_0.(\lambda y_0.xy_0))/(\lambda y.xy)](\lambda x.(\lambda y.xy)) \\ &= [(\lambda y_0.xy_0)/(\lambda y.xy)](\lambda x.(\lambda y.xy)) \\ &= \lambda x.(\lambda y_0.xy_0),\end{aligned}$$

so that $\lambda x.(\lambda y.xy) =_\alpha \lambda x.(\lambda y_0.xy_0)$. Similarly, we can make an α-conversion that gives us $\lambda x.(\lambda y_0.xy_0) =_\alpha \lambda y.(\lambda y_0.yy_0)$, and then $\lambda y(\lambda y_0.yy_0) =_\alpha \lambda y(\lambda x.yx)$. Therefore, there is a finite amount of α-conversions starting from $\lambda x.(\lambda y.xy)$ that, when successively applied one after the other, results in $\lambda y(\lambda x.yx)$, so that $\lambda xy.xy =_\alpha \lambda yx.yx$.

Importantly, we have the following:

Theorem 9. *$=_\alpha$ is an equivalence relation.*

Proof. Note that transitivity is immediate from our definition of α-conversion: if there is a finite amount of α-conversions that go from P_0 to P_1, and a finite amount of α-conversions that go from P_1 to P_2, then there is a finite amount of α-conversions that go from P_0 to P_2.

Let P be a λ-term containing $\lambda x.M$. We have that $x \notin FV(\lambda x.M)$, as x is bound in $\lambda x.M$. Then we have the α-conversion

$$(\lambda x.[x/x]M)/(\lambda x.M)]P = [(\lambda x.M)/(\lambda x.M)]P = P$$

goes from P to P, so that $P =_\alpha P$.

To show that if $P_0 =_\alpha P_1$ then $P_1 =_\alpha P_0$, it suffices to show that for any single α-conversion that goes directly from P_j to P_k, we can get another α-conversion that goes from P_k to P_j. Let P_j contain an occurrence of $\lambda x.M$, and let $y \notin FV(M)$. Then let

$$[(\lambda y.[y/x]M)/(\lambda x.M)]P_j = P_k$$

be an α-conversion from P_j to P_k, so that $P_j =_\alpha P_k$. Note that this implies that

$$P_j = [(\lambda x.[x/y]M)/(\lambda y.M)]P_k.$$

Since then $\lambda y.M$ occurs in P_j and $x \notin FV(\lambda y.M)$, we have that this is an α-conversion from P_k to P_j, so that $P_k =_\alpha P_j$.

□

Notably here, [SU06] actually defines the set of λ-terms defined earlier as the set of pre-terms, and then defines the set of λ-terms as the quotient of that set with respect to α-equivalence. This does lead to some technical differences in the following definition, but what we have now works just as well.

We now have the tools needed to define a mechanism for evaluating functions. This is done through so-called β-reduction:

Definition 9. *A term of the form $(\lambda x.M)N$ is called a β-redex. A term $[N/x]M$ is called a β-contractum. If P contains an occurrence of a β-redex, then the substitution*

$$[[N/x]M/(\lambda x.M)]P$$

is called a β-reduction of P. If a term contains no β-redexes, then it is in β-normal form.

The idea here is that the term $(\lambda x.M)N$ should be interpreted as evaluating the function $\lambda x.M$ at N, or substituting each instance of x in M with N, and β-reduction is the algorithmic process with which we accomplish this. We write $P \triangleright_\beta P'$ if P β-reduces to P'. For example, we can do

$$\begin{aligned}(\lambda x.xx)y &\triangleright_\beta [[y/x]xx/(\lambda x.xx)](\lambda x.xx) \\ &= [yy/(\lambda x.xx)](\lambda x.xx) \\ &= yy,\end{aligned}$$

so that $(\lambda x.xx)y \triangleright_\beta yy$. This matches our idea of function evaluation, as if we define a function f by $x \mapsto xx$, then $f(y) = yy$. When a term is in β-normal form, is when we think of a function as being fully evaluated and cannot be β-reduced further. For example, the β-normal form of $(\lambda x.xy)(\lambda u.vuu)$ is:

$$\begin{aligned}(\lambda x.xy)(\lambda u.vuu) &\triangleright_\beta (\lambda u.vuu)y \\ &\triangleright_\beta vyy.\end{aligned}$$

The most essential theorem on β-reduction is the Church-Rosser theorem, which essentially states that the β-normal form of a term is unique.

Theorem 10. *If $P \triangleright_\beta M$ and $P \triangleright_\beta N$, then there exists a term T such that $M \triangleright_\beta T$ and $N \triangleright_\beta T$.*

There is no quick or easy proof of this, so refer to page 282 of [HS08]. However, this theorem also guarantees that there is an issue with the untyped lambda calculus. That is, not every term can be reduced to a β-normal form. In computational terms, this means that there are programs in the untyped lambda calculus which do not terminate. For example:

$$\begin{aligned}(\lambda x.xx)(\lambda x.xx) &\triangleright_\beta [[(\lambda x.xx)/x]xx/(\lambda x.xx)](\lambda x.xx) \\ &= [(\lambda x.xx)(\lambda x.xx)/(\lambda x.xx)](\lambda x.xx) \\ &= (\lambda x.xx)(\lambda x.xx).\end{aligned}$$

This therefore uniquely infinitely reduces to itself, and as such never terminates. This is not a contradiction *per se*, but it does mean that the untyped lambda calculus is a non-computable theory, and especially for the purposes of a proof assistant, we want these kinds of non-computable expressions to be impossible. This is the problem that the typed lambda calculus solves, by introducing a typing relation on the set of λ-terms that guarantees the well-posedness of functions.

2.3 Natural deduction and typed lambda calculus

Type theory is usually discussed in the system of natural deduction, so I will first briefly introduce it. Natural deduction is a proof system that formalizes the semantics of propositional logic and forms the logical language in which much type theory ends up being expressed. I will first define all the required grammar for natural deduction:

Definition 10. *Natural deduction is defined with an infinite set* PV*, called the set of propositional variables, with a constant term* $\bot \in \mathrm{PV}$*, and the symbols* $\to$*,* $\vee$*,* $\wedge$*. The set* Φ *of formulas of propositional logic is defined as:*

1. *If* $\varphi \in \mathrm{PV}$*, then* $\varphi \in \Psi$*;*
2. *If* $\varphi, \psi \in \Psi$*, then* $(\varphi \to \psi), (\varphi \vee \psi), (\varphi \wedge \psi) \in \Psi$*.*

A judgment in natural deduction is a pair, written $\Gamma \vdash \varphi$ *(read as "*Γ *proves* φ*"), of a finite set of formulas* Γ *and a formula* φ*.* Γ *is called the context of the judgment, and* φ *is called the consequent.*

The set of judgments is defined by the following axioms of propositional logic in natural deduction (the notation

$$\frac{\Gamma \vdash \varphi \qquad \Gamma' \vdash \psi}{\Gamma'' \vdash \tau}$$

should be read as, "for $\Gamma, \Gamma', \Gamma'' \subseteq \Psi$*,* $\varphi, \psi, \tau \in \Psi$*, if* $\Gamma \vdash \varphi$ *and* $\Gamma' \vdash \psi$ *are judgments, then* $\Gamma'' \vdash \tau$ *is a judgment"):*

1. *If* $\Gamma \subseteq \Psi$ *and* $\varphi \in \Psi$*, then* $\Gamma, \varphi \vdash \varphi$ *(*Ax.*);*

2. $\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}$ *(*$\to$ I*);*

3. $\dfrac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}$ *(*$\to$ E*);*

4. $\dfrac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}$ *(*$\wedge$I*);*

5. $\dfrac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}$ *(*$\wedge$E1*);*

   $\dfrac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}$ *(*$\wedge$E2*);*

6. $\dfrac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}$ *(*$\vee$I1*);*

   $\dfrac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}$ *(*$\vee$I2*);*

7. $\dfrac{\Gamma, \varphi \vdash \tau \qquad \Gamma, \psi \vdash \tau \qquad \Gamma, \vdash \varphi \vee \psi}{\Gamma \vdash \tau}$ *(*$\vee$E*);*

8. $\dfrac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}$ *(*$\bot$E*).*

These judgment axioms are meant to model our intuitive idea of propositional logic. Observe the axiom rule, which states $\Gamma, \varphi \vdash \varphi$. This means that if we have that some formulas in Γ, and specifically some formula φ, are true, then we have a proof that φ is true. This forms our inductive base case for defining the set of judgments in natural deduction. From here, there are two types of judgment axioms: introduction

rules and elimination rules. Introduction rules tell us how to create judgments of a certain form, and elimination rules tell us how to deduce things from judgments of a certain form. For example, $\to$ I, or implication introduction, tells us that if from some formulas Γ, φ, we can prove that ψ is true, then we have a proof of $\varphi \to \psi$. This tells us the formula for constructing judgments of the form $\varphi \to \psi$: assume φ, then prove ψ. Similarly $\to$ E, or implication elimination, tells us that if we have that $\varphi \to \psi$ is true, and that φ is true, then we have a proof of ψ. This is the familiar modus ponens rule. Note that there is no introduction rule for $\bot$. This is because $\bot$ should be read as the propositional formula "False." We should not be able to construct a proof of False, so it has no introduction rules (that is the definition of $\bot$ in type theory). We can eliminate off of $\bot$ by, if we are able to construct a proof of $\bot$, then we have a proof of any proposition φ; this is exactly the principle of explosion.

These rules, from a set theoretical perspective, encode much of mathematics. This point is made very explicitly with the proof assistant Metamath, which starts explicitly from these implication rules, and ends up deriving a vast wealth of mathematics from those axioms alone. But these natural deduction rules are a logic built "on top of" a model of set theory. For the simply typed lambda calculus, we must build on the untyped lambda calculus defined earlier. To do this, for any λ-term, we have to associate a 'type' to each term. Primarily, we need to make sure that application to an abstraction enforces a type discipline that makes the types of the terms match: an abstraction M needs to have a function type like $\sigma \to \tau$, and to apply N to M, N must have type σ. The result MN will have the type τ. This is the same idea as making sure that you only evaluate functions for values that are in the domain of the function, and you end up with a result in the codomain of the function.

Definition 11. *Consider any finite or infinite sequence of symbols, called atomic types. Then the set of types is defined inductively as:*

1. *Every atomic type is a type;*

2. if σ and τ are types, then $(\sigma \to \tau)$ is a type, called a function type.

A type-assignment formula is an expression $X : \tau$, read 'X has the type τ', where X is a λ-term, and τ is a type. For any atomic types σ, τ, and any atom x and λ-terms M, N, with $x \notin FV(M)$, we define the system of type-assignments inductively on X with the following base-case, and type introduction and elimination rules:

1. $\Gamma, x : \tau \vdash x : \tau$ *(*Var.*);*

2. $$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : \sigma \to \tau}$$ *(*$\to$ i*);*

3. $$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$ *(*$\to$ e*).*

This provides an algorithm, importantly a polynomial-time one, with which we can judge the type of some compound λ-term from arbitrarily assigned types to atomic λ-terms. For example, if for some atom x, arbitrarily assigned the type τ, then we can determine the type of $\lambda x.x$ with the term introduction rule by:

$$\frac{\Gamma, x : \tau \vdash x : \tau \quad \text{(Var.)}}{\Gamma \vdash (\lambda x.x) : \tau \to \tau \quad (\to \text{i}).}$$

We see then that we have the judgment that $\lambda x.x$ has type $\tau \to \tau$, which makes sense as we expect $\lambda x.x$ to take in xs of type τ and output objects of type τ, namely, the exact same x as the input.

For a slightly more complex example, let $\Gamma \vdash x : \sigma \to \tau \to \rho$, $\Gamma \vdash y : \sigma \to \tau$, and $\Gamma \vdash z : \sigma$. Then, to determine the type of $\lambda xyz.xz(yz)$, we break it down inductively. We can write an abstraction as nested lambdas, so the term is $\lambda x.(\lambda y.(\lambda z.xz(yz)))$, and thus the derivation will start first by determining the type of $\lambda y.(\lambda z.xz(yz))$, which starts with determining $\lambda z.xz(yz)$. We have $\Gamma \vdash yz : \tau$ by the type elimination rule, and $\Gamma \vdash xz : \tau \to \rho$ similarly, so $\Gamma \vdash xz(yz) : \rho$, and therefore $\Gamma \vdash \lambda z.xz(yz) :$

$\sigma \to \rho$. Therefore $\Gamma \vdash \lambda y.(\lambda z.xz(yz)) : (\sigma \to \tau) \to \sigma \to \rho$, and therefore $\Gamma \vdash \lambda x.(\lambda y.(\lambda z.xz(yz))) : (\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho$.

We can already prove some simple properties about this system, notably some that distinguish it from a set theory. For example, terms must have unique types in this system, whereas elements can be members of multiple sets in set theories.

Theorem 11. *If* $\Gamma \vdash M : \sigma$ *and* $\Gamma \vdash M : \tau$*, then* $\sigma = \tau$*.*

Proof. Assume that $\Gamma \vdash M : \sigma$, $\Gamma \vdash M : \tau$, and $\sigma \neq \tau$. We induct with respect to M. If M is an atomic variable, so that $M = x : \sigma$ and $M = x : \tau$, then we have $\Gamma \vdash \lambda x.x : \sigma \to \tau$. But for any $\Gamma \vdash z : \sigma$ for which also $\Gamma \nvdash z : \tau$, then $\Gamma \vdash (\lambda x.x)z : \tau$, so that $\Gamma \vdash z : \tau$, a contradiction, so that $\sigma = \tau$.

Let M be an application, so that $M = NP : \sigma$ and $M = NP : \tau$, such that if $\Gamma \vdash N : \alpha$ and $\Gamma \vdash N : \beta$, then $\alpha = \beta$, and if $\Gamma \vdash P : \alpha$ and $\Gamma \vdash P : \beta$, then $\alpha = \beta$. Assume that $\sigma \neq \tau$. N must be a function type here, so that we have $\Gamma \vdash N : \rho \to \sigma$ and $\Gamma \vdash N : \rho \to \tau$ for some ρ, but $\rho \to \sigma \neq \rho \to \tau$, since $\sigma \neq \tau$, a contradiction to the inductive hypothesis. Therefore $\sigma = \tau$ in this case.

Let M be an abstraction, so that $M = (\lambda x.N) : \sigma$ and $M = (\lambda x.N) : \tau$, such that if $\Gamma \vdash N : \alpha$ and $\Gamma \vdash N : \beta$, then $\alpha = \beta$, and if $\Gamma \vdash x : \alpha$ and $\Gamma \vdash x : \beta$, then $\alpha = \beta$. Assume that $\sigma \neq \tau$. Here, σ and τ must be function types, so that for some ρ, π, υ, ξ, $\sigma = \rho \to \pi$ and $\tau = \upsilon \to \xi$. But then we must have $\Gamma \vdash N : \pi$ and $\Gamma \vdash N : \xi$, so therefore $\pi = \xi$, and that $\Gamma \vdash x : \rho$ and $\Gamma \vdash x : \upsilon$, so therefore $\rho = \upsilon$. But then $\sigma = \rho \to \pi = \upsilon \to \xi = \tau$, a contradiction to the inductive hypothesis. Therefore $\sigma = \tau$ in this case. Therefore the result holds by induction on λ-terms.

□

Note that this typing system is incapable of assigning types to every λ-term, on purpose. For example, the term xx has no way to be assigned a type, regardless of

what the type of x may be. This is why the typed lambda calculus gets rid of the non-computable functions that do not have a β-normal form, as those terms that do not have a β-normal form in the untyped lambda calculus are not typable.

Theorem 12. *Every well-typed expression in the type assignment system described above has a β-normal form.*

Proof. Define the degree $\delta(\tau)$ of a type τ inductively for some atomic type p and types τ, σ by $\delta(p) = 0$ and $\delta(\tau \to \sigma) = 1 + \max(\delta(\tau), \delta(\sigma))$. Then let the degree of a β-redex $\Delta = (\lambda x : \tau.P : \rho)R$ be $\delta(\tau \to \rho)$.

Let M be a well-typed λ-term. Let n_M be the number of β-redexes that occur in M that are of maximal degree, and let the maximal degree of β-redexes that occur in M be m_M. We will use the notation $M = (n_M, m_M)$. Let Δ be the right-most β-redex of maximal degree in M, that is, that the position of the first symbol of Δ is further to the right than any other β-redex in M. Let M' be the term obtained from β-reducing Δ in M. Then it suffices to show that $n_{M'} < n_M$, as then the process of β-reducing M from the right like this will gradually reduce n_M to 0, therefore putting M in β-normal form. We will do this by double induction on n_M and m_M.

If $m_M = 0$, then the maximal degree of any β-redex in $M = (n_M, 0)$ is 0, which means that the only β-redexes in $(n, 0)$ must be atomic terms, so that $(n, 0)$ must be in β-normal form. Now assume that for any $k < m$, (n_M, k) has a β-normal form, and we want to show that (n_M, m) has a β-normal form. If $n_M = 0$, then $(0, m)$ has no β-redexes of its maximal degree, and therefore there are no β-redexes in $(0, m)$, so that $(0, m)$ has a β-normal form. Then for any $\ell < n$, assume that (ℓ, m) has a β-normal form, and we want to show that (n, m) has a β-normal form. To show this, it suffices to show that if we β-reduce the rightmost β-redex of degree m in (n, m), then we cannot end up getting *more* β-redexes of degree m from it, as then (n, m) would reduce to (ℓ, m) for some $\ell < n$, which has a β-normal form by the inductive

hypothesis.

There are two possible cases in which the number of β-redexes will increase after reduction: by copying existing β-redexes or creating new ones. New β-redexes are created when a non-abstraction A that occurs in M is turned into an abstraction by the reduction of Δ, which is only possible when A is an atomic variable or $A = \Delta$. If A is an atomic variable, then the new β-redex is degree 0, so that it necessarily will have a lower degree than M started with, so the result holds in that case. If $A = \Delta$, then there are three cases that cause new β-redexes to form under reduction:

1. Δ has the form $(\lambda x : \rho \to \mu. \ldots xP : \rho \ldots)(\lambda y : \rho.Q : \mu)$. This has degree $\delta((\rho \to \mu) \to (\rho \to \mu))$ When reduced we end up with the β-redex $\cdots(\lambda y : \rho.Q : \mu)(P : \rho \ldots)$, which has degree $\delta(\rho \to \mu)$, which is less than Δ, so the degree after reduction is less than initially, so the result holds.

2. Δ has the form $((\lambda x : \tau.\lambda y : \rho.R : \mu)(P : \tau))(Q : \rho)$. This has degree $\delta(\tau \to \rho \to \mu)$. The inside reduces to $(\lambda y : \rho.R_1 : \mu)(q : \rho)$, and this has degree $\delta(\rho \to \mu)$, which is less than the degree of Δ, so the degree reduces in this case.

3. Δ has the form $((\lambda x : \rho \to \mu.x)(\lambda y : \rho.P : \mu))(Q : \rho)$. This has degree $\delta(\rho \to (\rho \to \mu))$ The inside reduces to $(\lambda y : \rho.P\mu)(Q : \rho)$, which has a lower degree of $\delta(\rho \to \mu)$, which is less than the degree of Δ, so the degree also reduces in this case.

The other case where the number of β-redexes increases by copying occurs when $\Delta = (\lambda x : \tau.P : \rho)(Q : \tau)$, where P contains more than one free occurrence of x, so that each β-redex in Q would be multiplied when reduced. But Δ is the rightmost redex of degree δ in the term M, so that each β-redex in Q is either a smaller degree than Δ, or gets reduced to β-redexes of degree smaller than δ, so that the degree of the terms in Δ after reduction would be smaller than δ in the end. Therefore, in all cases (n, m) reduces to (ℓ, m) for some $\ell < n$, so the result holds by induction.

□

This is the statement that the simply typed lambda calculus is *normalizing*. There is a stronger, and harder to prove, statement that the simply typed lambda calculus is *strongly* normalizing, which means that there is a bound on the length of every normalization sequence. Note that strong normalization, along with the Church-Rosser theorem lifted to the simply typed lambda calculus, implies the consistency of the simply typed lambda calculus.

Theorem 13. *The simply typed lambda calculus is consistent. That is, there are λ-terms M and N for which $\Gamma \nvdash M = N$.*

Proof. Let $\Gamma \vdash x : \sigma \to \tau \to \rho$, $\Gamma \vdash y : \sigma \to \tau$, $\Gamma \vdash z : \sigma$, with $\rho \neq \sigma$, $M = \lambda xy.x$, and $N = \lambda xyz.xz(yz)$, and assume that $\Gamma \vdash M = N$. Then

$$\begin{aligned}\Gamma \vdash \lambda xy.x = \lambda xyz.xz(yz) &\Leftrightarrow \Gamma \vdash \lambda xy.x =_\beta \lambda xyz.xz(yz) \\ &\Leftrightarrow \Gamma \vdash \exists P, \lambda xy.x \ \triangleright_\beta P \wedge \lambda xyz.xz(yz) \ \triangleright_\beta P,\end{aligned}$$

where the first implication is immediate from the definition of β-reduction, and the second implication is from the Church-Rosser theorem. But note that we found earlier that $\Gamma \vdash \lambda xyz.xz(yz) : (\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho$, and we can also find $\Gamma \vdash \lambda xy.x : (\sigma \to \tau) \to \sigma$, so that each of these terms uniquely β-reduce to P which, by the method of β-reduction detailed above, must then be such that $\Gamma \vdash P : \rho$ and $\Gamma \vdash P : \sigma$, a contradiction to Theorem 11 since $\rho \neq \sigma$. □

All of these theorems together form the basis for our trust in proof assistant systems. The type theories that are implemented by any proof assistant system have many more parts than the simply typed lambda calculus, and their consistency is generally an open problem, but they all build on top of this system. Most importantly, as far as implementing mathematics in these systems goes, there is a theorem called the Curry-Howard correspondence. The Curry-Howard correspondence gives the

fundamental relationship between propositional logic given in Definition 10, and the simply typed lambda calculus of Definition 11. We can already see an instance of this correspondence coming up even just in the statements of the definitions. The implication rules of propositional logic (with bold characters inserted for emphasis) are

$$1. \quad \frac{\Gamma, \boldsymbol{\varphi} \vdash \boldsymbol{\psi}}{\Gamma \vdash \boldsymbol{\varphi} \to \boldsymbol{\psi}} \, (\to \mathrm{I});$$

$$2. \quad \frac{\Gamma \vdash \boldsymbol{\varphi} \to \boldsymbol{\psi} \qquad \Gamma \vdash \boldsymbol{\varphi}}{\Gamma \vdash \boldsymbol{\psi}} \, (\to \mathrm{E}),$$

and the function type formation rules are

$$1. \quad \frac{\Gamma, x : \boldsymbol{\sigma} \vdash M : \boldsymbol{\tau}}{\Gamma \vdash (\lambda x.M) : \boldsymbol{\sigma} \to \boldsymbol{\tau}} \, (\to \mathrm{i});$$

$$2. \quad \frac{\Gamma \vdash M : \boldsymbol{\sigma} \to \boldsymbol{\tau} \qquad \Gamma \vdash N : \boldsymbol{\sigma}}{\Gamma \vdash MN : \boldsymbol{\tau}} \, (\to \mathrm{e}).$$

We can see immediately how the types in the type formation rules follow the same pattern as the propositions in the implication rules. The interpretation of this is that *proofs are programs and vice versa.* A proof in (constructive) propositional logic has an interpretation in the simply typed lambda calculus, where it is understood as a computation, and similarly any computation in the simply typed lambda calculus represents a proof. Precisely, a proof in propositional logic is a proof of the type of the term of the corresponding program, which is exactly the pattern that is followed all the time when working with proof assistants. To prove a proposition in a proof assistant, you have to construct a term that has that proposition as a type, which is exactly what the Curry-Howard correspondence states. Here is the precise statement of the Curry-Howard correspondence:

Theorem 14. *For a context* Γ*, define* $\mathrm{rg}(\Gamma) = \{\tau \mid (x : \tau) \in \Gamma, \text{ for some } x\}$*. Then*

1. *If* $\Gamma \vdash M : \varphi$ *in the simply typed lambda calculus, then* $\mathrm{rg}(\Gamma) \vdash \varphi$ *in propositional logic.*

2. *If* $\Delta \vdash \varphi$ *in propositional logic, then* $\Gamma \vdash M : \varphi$ *in the simply typed lambda calculus, for some* M *and some* Γ *with* $\mathrm{rg}(\Gamma) = \Delta$.

Proof. Assume that $\Gamma \vdash M : \varphi$ in the simply typed lambda calculus. Then $\varphi \in \mathrm{rg}(\Gamma)$, so by the Ax. rule of propositional logic, $\mathrm{rg}(\Gamma) \vdash \varphi$.

Assume that $\Delta \vdash \varphi$ in propositional logic. Let $\Gamma = \{(x_\varphi : \varphi) \mid \varphi \in \Delta\}$, where x_φ are any distinct variables. Then by the Var. rule of the simply typed lambda calculus, we have $\Gamma \vdash x_\varphi : \varphi$. □

The simply typed lambda calculus as we can see has a lot of nice properties, but it still is not sufficient for truly covering every pattern commonly used in mathematical practice. Chiefly, the propositional logic and simply typed lambda calculus that we have introduced do not have support for quantifiers. In order to extend our logic to at least a first-order theory which can express statements with $\forall$ or $\exists$, we need to introduce dependent types.

2.4 Dependent type theory and the construction of mathematical objects

Dependent type theory was made to increase the expressiveness of the type theories considered earlier to a wider range of possible functions. Intuitively, dependent type theory allows us to create new types out of terms of other types. Some functions that we might want to consider have the type of their outputs depending on the type of their inputs. For example, a function that concatenates two vectors of length

m and n outputs a vector of length $m + n$, so that the space or type of the output depends on the type of the inputs. We specifically will be considering Martin-Löf's intuitionistic dependent type theory as originally described here [Mar75], the first dependent type theory constructed, and the basis of what most proof assistants implement today. In this section, I will focus less on the meta-theoretical properties of dependent type theory as I did in the previous sections but instead will be focusing on how mathematical objects are constructed in dependent type theory.

The definition of the basic rules of Martin-Löf dependent type theory is not entirely similar to how the simply typed lambda calculus is defined, and there are significantly more rules, but most of these rules serve to make the same constructions of substitution and conversions/reductions that we have for the simply typed λ-calculus in a purely type theory context.

Definition 12. *There are four judgments in dependent type theory:*

1. *A typing judgment:*

   $$\Gamma \vdash A \text{ type};$$

2. *type judgmental equality:*

   $$\Gamma \vdash A = B \text{ type};$$

3. *term judgments:*

   $$\Gamma \vdash a : A.$$

4. *and term judgmental equality:*

   $$\Gamma \vdash a = b : A.$$

If $\Gamma \vdash A$ type*, then a family of types* $B(x)$ *over* A *is an object in a judgment* $\Gamma, x : A \vdash B(x)$ type*.*

There are six inference rules that govern the formation of types and terms:

$$\frac{\Gamma, x : A \vdash B(x) \text{ type}}{\Gamma \vdash A \text{ type}}\,\mathrm{F};$$

$$\frac{\Gamma \vdash A = B \text{ type}}{\Gamma \vdash A \text{ type}}\,\mathrm{eql_T};$$

$$\frac{\Gamma \vdash A = B \text{ type}}{\Gamma \vdash B \text{ type}}\,\mathrm{eqr_T};$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash A \text{ type}}\,\mathrm{T};$$

$$\frac{\Gamma \vdash a = b : A}{\Gamma \vdash a : A}\,\mathrm{eql_t};$$

$$\frac{\Gamma \vdash a = b : A}{\Gamma \vdash b : A}\,\mathrm{eqr_t}.$$

The rules governing judgmental equality state that judgmental equality is reflexive, symmetric, and transitive:

$$\frac{\Gamma \vdash A \text{ type}}{\Gamma \vdash A = A \text{ type}}\,\mathrm{id_T};$$

$$\frac{\Gamma \vdash A = B \text{ type}}{\Gamma \vdash B = A \text{ type}}\,\mathrm{rfl_T};$$

$$\frac{\Gamma \vdash A = B \text{ type} \qquad \Gamma \vdash B = C \text{ type}}{\Gamma \vdash A = C \text{ type}}\,\mathrm{trans_T};$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a = a : A}\,\mathrm{id_t};$$

$$\frac{\Gamma \vdash a = b : A}{\Gamma \vdash b = a : A}\,\mathrm{rfl_t};$$

$$\frac{\Gamma \vdash a = b : A \qquad \Gamma \vdash b = c : A}{\Gamma \vdash a = c : A}\,\mathrm{trans_t}.$$

Then for any judgment $\mathcal{J}$, we have the following variable conversion rule that allows us to substitute in any judgmentally equal type in a derivation:

$$\frac{\Gamma \vdash A = A' \text{ type} \qquad \Gamma, x : A, \Delta \vdash \mathcal{J}}{\Gamma, x : A', \Delta \vdash \mathcal{J}}\,\mathrm{conv}.$$

And we have the following three rules governing substitution, the first which shows us how to substitute, the second two that substitution respects judgmental equality:

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A, \Delta \vdash \mathcal{J}}{\Gamma, \Delta[a/x] \vdash \mathcal{J}[a/x]}\,\mathrm{S};$$

$$\frac{\Gamma \vdash a = a' : A \qquad \Gamma, x : A, \Delta \vdash B \text{ type}}{\Gamma, \Delta[a/x] \vdash B[a/x] = B[a'/x] \text{ type}}\,\mathrm{cong_T};$$

$$\frac{\Gamma \vdash a = a' : A \qquad \Gamma, x : A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] = b[a'/x] : B[a.x]}\,\mathrm{cong_t}.$$

We have a weakening rule, that allows us to expand a context by an arbitrary variable:

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, \Delta \vdash \mathcal{J}}{\Gamma, x : A, \Delta \vdash \mathcal{J}}\,\mathrm{W}.$$

And lastly we have a variable rule:

$$\frac{\Gamma \vdash A \text{ type}}{\Gamma, x : A \vdash x : A}\,\delta.$$

The chief thing that makes this system *dependent* is the type family construction $\Gamma, x : A \vdash B(x)$ type. In other words, for every term x of type A, we get a type $B(x)$. This can be thought of as an A-indexed family of types $B(x)$. We can use this to make a definition of dependent product types, or Π-types, which are necesary to define dependent function types:

Definition 13. *Let B be a type family over A. Then the Π-formation rule states that we can then form a type of the form $\Pi_{x:A}B(x)$, which can be thought of as the type of functions from $x : A$ to $B(x)$:*

$$\frac{\Gamma, x : A \vdash B(x) \text{ type}}{\Gamma \vdash \Pi_{x:A}B(x) \text{ type}}\,\Pi.$$

The Π-introduction rule then shows us how to construct terms of type $\Pi_{x:A}B(x)$:

$$\frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma \vdash \lambda x.b(x) : \Pi_{x:A}B(x)}\,\lambda.$$

The Π-elimination rule shows us how to use terms of type $\Pi_{x:A}B(x)$:

$$\frac{\Gamma \vdash f : \Pi_{x:A}B(x)}{\Gamma, x : A \vdash f(x) : B(x)}\,\text{ev}.$$

Then there are two Π-computation rules that give us how Π-types should behave. The β-rule, which gives us the equivalent of β-reduction:

$$\frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma, x : A \vdash (\lambda y.b(y))(x) = b(x) : B(x)}\,\beta,$$

and the η-rule, which ensures that Π-types are λ abstractions:

$$\frac{\Gamma \vdash f : \Pi_{x:A}B(x)}{\Gamma \vdash \lambda x.f(x) = f : \Pi_{x:A}B(x)}\,\eta.$$

Lastly we need rules that ensure that the construction and evaluation of Π*-types respects judgmental equality:*

$$\frac{\Gamma \vdash A = A' \text{ type} \qquad \Gamma, x : A \vdash B(x) = B'(x) \text{ type}}{\Gamma \vdash \Pi_{x:A} B(x) = \Pi_{x:A'} B'(x) \text{ type}} \Pi\text{eq};$$

$$\frac{\Gamma, x : A \vdash b(x) = b'(x) : B(x)}{\Gamma \vdash \lambda x.b(x) = \lambda x.b'(x) : \Pi_{x:A} B(x)} \lambda\text{eq};$$

$$\frac{\Gamma \vdash f = f' : \Pi_{x:A} B(X)}{\Gamma, x : A \vdash f(x) = f'(x) : B(x)} \text{eveq}.$$

This is all very abstract, so let us see how these rules are applied to construct something down-to-earth. For two types A and B, we define the type $A \to B$ through this derivation:

$$\cfrac{\cfrac{\cfrac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma, x : A \vdash B \text{ type}} \text{W}}{\Gamma \vdash \Pi_{x:A} B \text{ type}} \Pi}{\Gamma \vdash A \to B := \Pi_{x:A} B \text{ type}} \text{def}.$$

Here we simply combined the weakening rules and the Π-formation rules as applied to two arbitrary types, and defined the result as the type $A \to B$. From a derivation like this, we can cut out the middle parts to get the inference rule for function types:

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma \vdash A \to B \text{ type}} \to,$$

and we also get the following definition rule:

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma \vdash A \to B = \Pi_{x:A} B \text{ type}} \to_{\text{def}}.$$

We want to make sure that this actually has the properties that we usually expect for functions. First, I prove a helpful term conversion rule:

Lemma 3.

$$\frac{\Gamma \vdash A = A' \text{ type} \qquad \Gamma \vdash a : A}{\Gamma \vdash a : A'} \,\mathrm{conv_t}.$$

Proof.

$$\dfrac{\Gamma \vdash a : A \qquad \dfrac{\dfrac{\Gamma \vdash A = A' \text{ type}}{\Gamma \vdash A' = A \text{ type}}\,\mathrm{rfl_T} \qquad \dfrac{\dfrac{\Gamma \vdash A = A' \text{ type}}{\Gamma \vdash A' \text{ type}}\,\mathrm{eqr_T}}{\Gamma, x : A' \vdash x : A'}\,\delta}{\Gamma, x : A \vdash x : A'}\,\mathrm{conv}}{\Gamma \vdash a : A'}\,\mathrm{S}.$$

□

From $\mathrm{conv_t}$ and $\to_{\mathrm{def}}$, we can immediately derive λ-abstractions, function evaluation, β-reduction, and the η-rule for function types as we have defined them, as is desired for the usual behavior of functions:

Theorem 15.

$$\frac{\Gamma \vdash B \text{ type} \qquad \Gamma, x : A \vdash b(x) : B}{\Gamma \vdash \lambda x.b(x) : A \to B}\,\lambda_{\to};$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma, x : A \vdash f(x) : B}\,\mathrm{ev}_{\to};$$

$$\frac{\Gamma \vdash B \text{ type} \qquad \Gamma, x : A \vdash b(x) : B}{\Gamma, x : A \vdash (\lambda y.b(y))(x) = b(x) : B}\,\beta_{\to};$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash \lambda x.f(x) = f : A \to B}\,\eta_{\to}.$$

Proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x : A \vdash b(x) : B}{\Gamma, x : A \vdash B \text{ type}}\,\mathrm{T}}{\Gamma \vdash A \text{ type}}\,\mathrm{F} \qquad \Gamma \vdash B \text{ type}}{\Gamma \vdash A \to B = \Pi_{x:A} B \text{ type}}\,{\to_{\mathrm{def}}} \qquad \dfrac{\Gamma, x : A \vdash b(x) : B}{\Gamma \vdash \lambda x.b(x) : \Pi_{x:A} B}\,\lambda}{\Gamma \vdash \lambda x.b(x) : A \to B}\,\mathrm{conv_t}.$$

The rest follow from very similar proofs to this one. □

We have now completely specified the type of functions between types, and as such we have recovered the simply typed lambda calculus. We are now ready to start proving properties about functions. One primary thing we want to do with functions is compose them, and in order to do that, we need to construct an inference rule that takes in two functions $g : B \to C$ and $f : A \to B$ and provides a function $g \circ f : A \to C$, defined by $\lambda x.g(f(x))$. We derive this as such:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash f : A \to B}{\Gamma, x : A \vdash f(x) : B}\,\mathrm{ev}_{\to} \qquad \dfrac{\Gamma \vdash g : B \to C}{\Gamma, y : B \vdash g(y) : C}\,\mathrm{ev}_{\to}}{\Gamma, x : A \vdash g(f(x)) : C}\,\mathrm{S}}{\Gamma \vdash \lambda x.g(f(x)) : A \to C}\,\lambda_{\to}}{\Gamma \vdash g \circ f = \lambda x.g(f(x)) : A \to C}\,\mathrm{def}.$$

An important property of function composition is that it is associative, and we can prove that now:

Theorem 16.

$$\dfrac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash g : B \to C \qquad \Gamma \vdash h : C \to D}{\Gamma \vdash (h \circ g) \circ f = h \circ (g \circ f) : A \to D.}$$

Proof. We split the derivation into three parts in the interest of space. The basic idea is that $((h \circ g) \circ f)(x)$ and $(h \circ (g \circ f))(x)$ both evaluate to $h(g(f(x)))$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash h : C \to D \qquad \dfrac{\Gamma \vdash g : B \to C \qquad \Gamma \vdash f : A \to B}{\Gamma \vdash g \circ f : A \to C}\circ}{\Gamma \vdash h \circ (g \circ f) : A \to D}\circ}{\Gamma \vdash h \circ (g \circ f) = \lambda x.h((g \circ f)(x)) : A \to D}\circ_{\text{def}}}{\Gamma \vdash h \circ (g \circ f) = \lambda x.h(\lambda y.g(f(y))(x)) : A \to D}\circ_{\text{def}}}{\Gamma \vdash h \circ (g \circ f) = \lambda x.h(g(f(x))) : A \to D}\beta_{\to}.$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash h : C \to D \qquad \Gamma \vdash g : B \to C}{\Gamma \vdash h \circ g : B \to D}\circ \qquad \Gamma \vdash f : A \to B}{\Gamma \vdash (h \circ g) \circ f : A \to D}\circ}{\Gamma \vdash (h \circ g) \circ f = \lambda x.(h \circ g)(f(x)) : A \to D}\circ_{\text{def}}}{\Gamma \vdash (h \circ g) \circ f = \lambda x.\lambda y.h(g(y))(f(x)) : A \to D}\circ_{\text{def}}}{\Gamma \vdash (h \circ g) \circ f = \lambda x.h(g(f(x))) : A \to D}\beta_{\to}.$$

$$\dfrac{\begin{array}{c}\Gamma \vdash (h \circ g) \circ f = \lambda x.h(g(f(x))) : A \to D \\ \Gamma \vdash h \circ (g \circ f) = \lambda x.h(g(f(x))) : A \to D\end{array}}{\Gamma \vdash (h \circ g) \circ f = h \circ (g \circ f) : A \to D}\,\text{trans}_{\text{t}}.$$

□

These derivations are a lot of work to write at this level of rigor and formality, but this is, in many ways, one of the motivations for creating proof assistant systems. In Lean, function types as we specified earlier are built-in to the foundations of the language. Typically also function composition is given in the standard library, but if we wanted to specify it ourselves in Lean, we write

```
def mycomp (f : β → δ) (g : α → β) : α → δ := λ x => f (g x)
```

Just this snippet on its own works. All the work that we had to do with the derivation is the work of the typechecker, which can algorithmically decide that the thing that we want, namely $\lambda x.f(g(x))$, does indeed have the type $\alpha \to \delta$. So, to specify function composition, we only have to give it the formula that we want, and the typechecker guarantees that that formula is precisely the resulting composed function that we want.

From here, a proof of the associativity of composition is a triviality:

```
theorem mycomp_assoc (f : φ → δ) (g : β → φ) (h : α → β) : mycomp
 (mycomp f g) h = mycomp f (mycomp g h) := rfl
```

rfl is a type constructor that instructs Lean to check if the left hand side of the equals sign is judgmentally equal to the right hand side of the equal sign, and in this case Lean's typechecker is able to re-construct the derivation that was given above to show that. So, we can appreciate then the hard work that Lean is doing 'under the hood' so to speak, and it is all specified with polynomial-time algorithmic procedures. Lean really is saving us a lot of grief as far as being an implementation of dependent type theory goes. Lean also makes it explicit how Π-types as we have defined them are the same as universal quantification:

```
theorem univ (p : α → Prop) : (∀ x : α, p x) ↔ ((x : α) → p x) := by
 simp
```

Now we will study the implementation of the natural numbers in Lean, and understand by formulating it in plain dependent type theory. The natural numbers are defined in Lean as such:

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

There are two type constructors for natural numbers: zero and succ. These correspond to the introduction rules of natural numbers and they specify how we introduce new natural numbers. That is, they are either zero or the successor of some other natural number. There are no high-level assumptions, so these constructors exist in the empty context. The two introduction rules of $\mathbb{N}$ then are:

$$\frac{}{\vdash 0_{\mathbb{N}} : \mathbb{N}}$$

$$\frac{}{\text{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}$$

Note that these exist in an empty context, so that there are no requirements for constructing these terms. The natural numbers in Lean are defined as an inductive type, which makes Lean automatically generate an induction principle for the type. We can check what the induction principle is by checking the type of the recursor for the natural numbers:

```
Nat.rec : {motive : Nat → Sort u} → motive Nat.zero → ((n : Nat) →
 motive n → motive n.succ) → (t : Nat) → motive t
```

Here we can think of motive as a type family P over $\mathbb{N}$. Breaking down this type, we see that this looks a lot like the usual induction principle. If we can derive a type $P(0_{\mathbb{N}})$ and a type $\forall n, P(n) \to P(\text{succ}_{\mathbb{N}}(n))$, then we have that the type family can be applied for any natural number. We can re-write this into our notation as:

$$\frac{\begin{array}{c}\Gamma, n : \mathbb{N} \vdash P(n) \text{ type} \\ \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \\ \Gamma \vdash p_S : \Pi_{n:\mathbb{N}} P(n) \to P(\text{succ}_{\mathbb{N}}(n))\end{array}}{\Gamma \vdash \text{ind}_{\mathbb{N}}(p_0, p_S) : \Pi_{n:\mathbb{N}} P(n)}\, \mathbb{N}\text{−ind}$$

We then see that $\mathbb{N}-$ind is the rule that shows us how to construct a Π-type of the form $\Pi_{n:\mathbb{N}} P(n)$, so that some proposition P on the natural numbers holds for every natural number. To fully specify the natural numbers, we need computation rules for $\mathbb{N}$. These computation rules tell us how the function $\text{ind}_{\mathbb{N}}(p_0, p_S)$ works when applied to either zero or the successor of a natural number:

$$\frac{\begin{array}{c}\Gamma, n : \mathbb{N} \vdash P(n) \text{ type} \\ \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \\ \Gamma \vdash p_S : \Pi_{n:\mathbb{N}} P(n) \to P(\text{succ}_{\mathbb{N}}(n))\end{array}}{\Gamma \vdash \text{ind}_{\mathbb{N}}(p_0, p_S, 0_{\mathbb{N}}) = p_0 : P(0_{\mathbb{N}})}\, \mathbb{N}\text{−comp}_0$$

$$\frac{\begin{array}{c}\Gamma, n : \mathbb{N} \vdash P(n)\ \text{type} \\ \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \\ \Gamma \vdash p_S : \Pi_{n:\mathbb{N}} P(n) \to P(\text{succ}_{\mathbb{N}}(n))\end{array}}{\Gamma, n : \mathbb{N} \vdash \text{ind}_{\mathbb{N}}(p_0, p_S, \text{succ}_{\mathbb{N}}(n)) = p_S(n, \text{ind}_{\mathbb{N}}(p_0, p_S, n)) : P(\text{succ}_{\mathbb{N}}(n))}\ \mathbb{N}\text{−comp}_S$$

In normal language, $\mathbb{N}\text{−comp}_0$ tells us that for some $P(n)$ that holds for any n, $P(0)$ is the same as the inductive base case for P. $\mathbb{N}\text{−comp}_S$ tells us that for some $P(n)$ that holds for any n, $P(\text{succ}(n))$, or $P(n+1)$, is computed by applying $P(n)$ to the procedure given in the inductive step, p_S. With this, we have completely specified the natural numbers. There is not too much to prove about natural numbers at this point, so we will now define addition on $\mathbb{N}$ in a way that recovers all the properties we expect of addition and demonstrate the induction principle to prove some of these properties. Addition is defined by induction in Lean with pattern matching:

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

This states that $a+b$ is defined by induction on b with $a+0=0$, and $a+(b+1) = (a+b)+1$. To derive this formally for ourselves, we want to construct a term of the form $\text{add}_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$. Equivalently, we want to construct a term of the form $m : \mathbb{N} \vdash \text{add}_{\mathbb{N}}(m) : \mathbb{N} \to \mathbb{N} = \Pi_{n:\mathbb{N}} P(n)$, which the induction principle gives us the rules to do. To use the induction principle, we need to construct $m : \mathbb{N} \vdash \text{add−zero}_{\mathbb{N}}(m) : \mathbb{N}$ and $m : \mathbb{N} \vdash \text{add−succ}_{\mathbb{N}}(m) : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$. $\text{add−zero}_{\mathbb{N}}(m)$ is easy, as it is just m:

$$\frac{\dfrac{\dfrac{}{\vdash \mathbb{N}\ \text{type}}}{m : \mathbb{N} \vdash m : \mathbb{N}}\ \delta}{m : \mathbb{N} \vdash \text{add−zero}_{\mathbb{N}}(m) = m : \mathbb{N}}\ \text{def.}$$

For $\text{add−succ}_{\mathbb{N}}(m) : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$, we equivalently want to construct $n : \mathbb{N} \vdash \text{add−succ}_{\mathbb{N}}(m) : \mathbb{N} \to \mathbb{N}$. If we inductively assume that we have the result of

$\mathrm{add}(m,n) = x$, then to get the value of $\mathrm{add{-}succ}_{\mathbb{N}}(m,n,x) : \mathbb{N}$, we want to apply the successor function to x, so thus we should define $n : \mathbb{N} \vdash \mathrm{add{-}succ}_{\mathbb{N}}(m) = \mathrm{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}$, and we can recover the type $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ easily from here:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathrm{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}}{n : \mathbb{N} \vdash \mathrm{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}\,\mathrm{W}}{m : \mathbb{N}, n : \mathbb{N} \vdash \mathrm{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}\,\mathrm{W}}{m : \mathbb{N} \vdash \lambda n.\mathrm{succ}_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})}\,\lambda_{\to}}{m : \mathbb{N} \vdash \mathrm{add{-}succ}_{\mathbb{N}}(\mathrm{m}) = \lambda n.\mathrm{succ}_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})}\,\mathrm{def.}$$

Thus, we can finish the derivation with the induction principle:

$$\dfrac{\dfrac{m : \mathbb{N} \vdash \mathrm{add{-}zero}_{\mathbb{N}}(m) : \mathbb{N} \qquad m : \mathbb{N} \vdash \mathrm{add{-}succ}_{\mathbb{N}}(m) : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})}{m : \mathbb{N} \vdash \mathrm{ind}_{\mathbb{N}}(\mathrm{add{-}zero}_{\mathbb{N}}(m), \mathrm{add{-}succ}_{\mathbb{N}}(m)) : \mathbb{N} \to \mathbb{N}}\,\mathbb{N}\mathrm{-ind}}{m : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}}(m) = \mathrm{ind}_{\mathbb{N}}(\mathrm{add{-}zero}_{\mathbb{N}}(m), \mathrm{add{-}succ}_{\mathbb{N}}(m)) : \mathbb{N} \to \mathbb{N}}\,\mathrm{def.}$$

Now we can show from this derivation that the two defining features of addition, $m + 0 = m$ and $m + \mathrm{succ}(n) = \mathrm{succ}(m + n)$, hold:

Theorem 17. $\mathrm{add}_{\mathbb{N}}(m, 0_{\mathbb{N}}) = m$, $\mathrm{add}_{\mathbb{N}}(m, \mathrm{succ}_{\mathbb{N}}(n)) = \mathrm{succ}_{\mathbb{N}}(\mathrm{add}_{\mathbb{N}}(m,n))$.

Proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{m : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}}(m) : \mathbb{N} \to \mathbb{N}}\,\mathrm{add{-}}\mathbb{N}_{\mathrm{def}}}{m : \mathbb{N}, 0_{\mathbb{N}} : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}}(m, 0_{\mathbb{N}}) : \mathbb{N}}\,\mathrm{ev}_{\to}}{m : \mathbb{N}, 0_{\mathbb{N}} : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}}(m, 0_{\mathbb{N}}) = \mathrm{add{-}zero}_{\mathbb{N}}(m) : \mathbb{N}}\,\mathbb{N}\mathrm{-comp}_0}{m : \mathbb{N}, 0_{\mathbb{N}} : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}}(m, 0_{\mathbb{N}}) = m : \mathbb{N}}\,\mathrm{add{-}zero{-}}\mathbb{N}_{\mathrm{def.}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{m : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}}(m) : \mathbb{N} \to \mathbb{N}}\,\mathrm{add{-}}\mathbb{N}_{\mathrm{def}}}{m : \mathbb{N}, n : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}}(m) : \mathbb{N} \to \mathbb{N}}\,\mathrm{W}}{m : \mathbb{N}, n : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}}(m, \mathrm{succ}_{\mathbb{N}}(n)) : \mathbb{N}}\,\mathrm{ev}_{\to}}{m : \mathbb{N}, n : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}}(m, \mathrm{succ}_{\mathbb{N}}(n)) = \mathrm{add{-}succ}_{\mathbb{N}}(n, \mathrm{add}_{\mathbb{N}}(m,n)) : \mathbb{N}}\,\mathbb{N}\mathrm{-comp}_{\mathrm{S}}}{m : \mathbb{N}, n : \mathbb{N} \vdash \mathrm{add}_{\mathbb{N}}(m, \mathrm{succ}_{\mathbb{N}}(n)) = \mathrm{succ}_{\mathbb{N}}(\mathrm{add}_{\mathbb{N}}(m,n)) : \mathbb{N}}\,\mathrm{add{-}succ{-}}\mathbb{N}_{\mathrm{def.}}$$

□

These proofs as I am presenting them are already abridged versions, as writing out the whole derivation would be very difficult to fit on a page and significantly reduce the conceptual clarity of the derivation, so I leave just the essential parts and ideas. But when we move on to slightly more nontrivial statements, such as commutativity, the notation gets even more cumbersome, and at that point proof assistants make their case as a valid way to express these proofs very well. Both of the above theorems can be simply proven with `rfl` in Lean.

Chapter 3

Bibliography

[She13] Henry Maurice Sheffer. "A Set of Five Independent Postulates for Boolean Algebras, with Application to Logical Constants". In: *Transactions of the American Mathematical Society* 14.4 (1913), pp. 481–488. DOI: 10.2307/1988701.

[Ore48] Oystein Ore. *Number Theory and Its History*. New York: McGraw-Hill Book Company, Inc., 1948, p. 65.

[MP74] N.E. Mendelsohn and R. Padmanabhan. "Minimal Identities for Boolean Groups". In: *Journal of Algebra* 34.3 (1974), pp. 451–457. DOI: 10.1016/0021-8693(75)90169-6.

[Mar75] Per Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Logic Colloquium '73*. Ed. by H.E. Rose and J.C. Shepherdson. Vol. 80. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pp. 73–118. DOI: https://doi.org/10.1016/S0049-237X(08)71945-1. URL: https://www.sciencedirect.com/science/article/pii/S0049237X08719451.

[McC+02] William McCune et al. "Short Single Axioms for Boolean Algebra". In: *Journal of Automated Reasoning* 29 (2002), pp. 1–16. DOI: 10.1023/A:1020542009983.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006.

[Geu08] Herman Geuvers. *Introduction to Type Theory*. 2008. URL: http://www.cs.ru.nl/~herman/onderwijs/provingwithCA/paper-lncs.pdf.

[HS08] Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, 2008.

[Yua12] Qiaochu Yuan. *Centers, 2-categories, and the Eckmann-Hilton argument*. 2012. URL: https://qchu.wordpress.com/2012/02/06/centers-2-categories-and-the-eckmann-hilton-argument/ (visited on 12/16/2024).

[Com20] The mathlib Community. "The Lean Mathematical Library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 367–381. ISBN: 9781450370974. DOI: 10.1145/3372885.3373824. URL: https://doi.org/10.1145/3372885.3373824.

[Com24] Johan Commelin. *StringMagmas.lean*. https://github.com/teorth/equational_theories/blob/main/equational_theories/StringMagmas.lean. 2024.

[RT24a] Cody Roux and Judah Towery. *Sheffer.lean*. https://github.com/teorth/equational_theories/blob/main/equational_theories/Sheffer.lean. 2024.

[RT24b] Cody Roux and Judah Towery. *ShefferAlgebra.lean*. https://github.com/teorth/equational_theories/blob/main/equational_theories/ShefferAlgebra.lean. 2024.

[Tao24a] Terence Tao. *A pilot project in universal algebra to explore new ways to collaborate and use machine assistance?* 2024. URL: https://terrytao.wordpress.com/2024/09/25/a-pilot-project-in-universal-algebra-to-explore-new-ways-to-collaborate-and-use-machine-assistance/ (visited on 10/22/2024).

[Tao24b] Terence Tao. *Subgraph.lean*. https://github.com/teorth/equational_theories/blob/244b739c661328ec5b3075edb6eaa2d304199927/equational_theories/Subgraph.lean#L429. 2024.

[Tow24] Judah Towery. *Sheffer.lean*. https://github.com/teorth/equational_theories/blob/main/equational_theories/Generated/VampireProven/Sheffer.lean. 2024.

[Wik24] Wikipedia. *Magma (algebra)*. 2024. URL: https://en.wikipedia.org/wiki/Magma_(algebra)#Category_of_magmas (visited on 12/16/2024).

[Bra] Martin Brandenburg. *Set-theoretical description of the free product?* Mathematics Stack Exchange. URL: https://math.stackexchange.com/q/5381.