Learning how to implement Mathematical Proofs in Agda

Andrew Sneap - 1980030

MSc - Maths & Computer Science
Third Year Project 2020-2021
Supervised by Martín Escardó and Co-supervised by Todd Waugh Ambridge
Overleaf Word Count: 9962

Abstract

Proof assistants such as Agda are becoming increasingly popular, used to formalise existing proofs, and sometimes to verify the correctness of contentious proofs. Within the framework of dependent type theory, I construct the Integers, Rational numbers and Dedekind Reals, choosing to work constructively. I prove a number of properties and theorems related to these structures, noting the differences compared to writing proofs without the use of proof assistants. Finally, I discuss how my implementation of the Dedekind Reals can be completed and further developed, with the aim of proving that the Dedekind Reals form a complete Archimedean field.

Contents

1 Introduction

There are a number of proof assistants, including Coq, Lean and Agda. Proof assistants allow us to verify proofs, which is a valuable feature. Up to the correctness of the proof checker, a proof verified to be correct allows mathematicians to have faith they can use that proof without fear of the proof being incorrect. This is not unheard of; Neeman produced a counterexample to a theorem some had used in their own work [Nee02]. In my project, I wanted to combine my disciplines of Mathematics & Computer Science, and writing mathematical proofs using a proof assistant was the perfect choice.

The goal of my project is to learn how to implement mathematical proofs in the dependently typed functional programming language Agda. The secondary goal of my project is to implement the Dedekind Real numbers in Agda. By working towards the second goal of the project, I will learn the techniques used to write mathematical proofs in Agda, taking note of the differences in approach and rigour that arise when writing proofs in a constructive type-theoretical setting. I build upon the work of Escardó's developmental Agda library, TypeTopology [Esc19b]. In this development, the Naturals Numbers have been defined, along with Addition and Order of Natural Numbers.

I begin by describing the relationship of some of the concepts used in Mathematics and Type Theory. I go on to formalise a number of properties and theorems of the Natural Numbers, including distributivity and Euclid's algorithm. I encounter surprising difficulty when attempting some of these proofs, and note the challenges involved in translating seemingly complete mathematical proofs. I move on to the Integers and Rational, proving properties of each, and note the difficulty of working with more complicated mathematical structures.

I then define the Dedekind Reals, and prove that there is an embedding from the Rationals to the Dedekind Reals. I define the field axioms, and discuss the steps necessary to prove that the Dedekind Reals form a complete archimedian field, as well as potential extensions of the formalisation developed in this project.

The Dedekind Reals have been formalised in Coq [Unkc]. There has been some work on the Dedekind Reals in Agda [Rom]. This is not a sound implementation, and the Cubical Agda library contains no formalisation of the Reals at the time of writing this report, so working on the Reals allows me to work towards a goal that requires independent work.

2 Background

My supervisor suggested Agda as a suitable choice, and his strong background of univalent mathematics written in Agda was a valuable resource for me. At its core, Agda is a dependently typed programming language, so the first step required me to develop my knowledge of dependent type theory and Agda. I used various reading materials to accomplish this, including [BD09], [Nor09] and [Dyb18]. I was more familiar with the mathematical representations of structures, as opposed to the corresponding type theoretic representations.

The TypeTopology library uses a minimal Martin-Löf type theory (MLTT). We assume only as much as we need to prove results. A full description of the ethos can be seen here [Esc]. As a result, in this project I make assumptions only when required. The Cubical Agda library utilises a slightly different approach, where some provable results can be used without explicit assumption.

At times, we require the use of results that are proved using the univalence axiom. This axiom is at the heart of the developing field of univalent mathematics. It is also connected of a new branch of mathematics known as homotopy type theory, introduced here [Uni13].

This section briefly introduces correspondence of mathematics and type theory, in the context of the TypeTopology library, along with remarks about Agda in particular.

The HoTT Book [Uni13] summarises the relationship between type Theory, logic and set theory. In particular, in the Type Topology library I used the empty and one point types, $\mathbb{0}$, $\mathbb{1}$ which are

Types	Logic	Sets	Homotopy
A	proposition	set	space
$a : A$	proof	element	point
$B(x)$	predicate	family of sets	fibration
$b(x) : B(x)$	conditional proof	family of elements	section
$\mathbf{0}, \mathbf{1}$	$\bot, \top$	$\emptyset, \{\emptyset\}$	$\emptyset, *$
$A + B$	$A \vee B$	disjoint union	coproduct
$A \times B$	$A \wedge B$	set of pairs	product space
$A \to B$	$A \Rightarrow B$	set of functions	function space
$\sum_{(x:A)} B(x)$	$\exists_{x:A} B(x)$	disjoint sum	total space
$\prod_{(x:A)} B(x)$	$\forall_{x:A} B(x)$	product	space of sections
Id_A	equality $=$	$\{ (x, x) \mid x \in A \}$	path space A^I

Figure 1: [Uni13, p. 15]

interpreted mathematically as truth values. I use the type-formers + and x, which are interpreted as disjoint union and the Cartesian product. I use the Sigma (Σ) and Pi (Π) type, which can be interpreted as the "existence" and "for all". Note that this correspondence describes how the *language* of mathematics is encoded within type theory. In comparison to set theory, where the language of mathematics is distinguishable from the mathematical structures, these type-formers have types themselves. This also applies to propositions of set theory. We can take any mathematical proposition, for example the statement "5 is not a prime number", and translate this into type theory. We expect a statement of this type to have an answer in the form of a truth value. Intuitively, we can recognize propositions in mathematics, and with the above in mind we recognize that propositions themselves have types. This is distinguishable from the notion of logical propositions as types; the HoTT Book refers to truth value propositions as "mere propositions", and they are also knows as "subsingletons" with univalent mathematics. As an example, consider the following Agda code. I define two versions of the same function illustrative purposes.

```
is-prime : ℕ → 𝓤₀ ̇
is-prime n = (n ≥ 2) × ((x y : ℕ) → x * y ≡ n → (x ≡ 1) + (x ≡ n))

five-is-not-prime : 𝓤₀ ̇
five-is-not-prime = ¬ is-prime 5

five-is-not-prime' : 𝓤₀ ̇
five-is-not-prime' = is-prime 5 → 𝟘

five-is-not-prime-is-prop : Fun-Ext → is-prop five-is-not-prime
five-is-not-prime-is-prop fe = Π-is-prop fe (λ x → 𝟘-is-prop)
```

Figure 2: Prime Example

Defining is-prime as Escardó does in [Esc19a], we can write in the proposition that 5 is not a prime number. In type theoretic terms, this proposition is equivalent to saying that "given an inhabitant of the type "is-prime 5", we can produce an element of the empty type. This is known as proof of negation, and is not the same as proof by contradiction (see [Bau10]).

We can translate mathematical propositions into type theory, but it is not enough to write the function 5 is prime. As mentioned before, propositions have a type, and we must show that five-is-not-prime is an inhabitant of that type. Informally, a type is a proposition if any two inhabitants of that type are equal. Note that in this code sample I have defined the type of the proof that 5 is not a prime. I have not proved that this statement is true, and of course it is not true. I have proved that the statement is a proposition, which is intuitive; we would have expected that the answer to this statement is false.

I will not go into detail of the proof that the statement is a proposition, but note the use of Fun-Ext, Function Extensionality (FE). Firstly, see that the statement is a function. We want to show that any two inhabitants (proofs) of the statement are equal. In this setting, it does not make sense to ask if the two inhabitants are intensionally equal, there are many examples of theorems with multiple different proofs, but we care about the functions being extensionally equal, or pointwise equal. It has been shown (as stated in [Esc19a]) that it is not provable or disprovable within the framework of MLTT that pointwise-equal functions are equal. As a result, FE is an assumption we must make to complete the proof. Other assumptions that must be made are the law of excluded middle (LEM), propositional truncation (PT) and propositional extensionality (PE). In the project, LEM is not used. I have only needed FE and PT. Function Extensionality and Univalent foundations are implemented in Type Topology, and I use many functions from the library within this project.

Note the use of $\mathcal{U}_0$. This is the notation in TypeTopology for Universe levels, also commonly known as Set, or Prop, where Prop indicates a logical proposition. I can emphasize here, speaking univalently, that while all propositions live in some Universe $\mathcal{U}_n$, not all inhabitants of a Universes are propositions. It is for this reason the term subsingleton is preferred, however in this report I use the term proposition. Most of the propositions I use in this report live in the lowest Universe $\mathcal{U}_0$, with exceptions discussed at the end.

Emacs is the highly recommended IDE for writing Agda code. There is the availability of an extension in Visual Studio Code, but after a brief period using VS Code I switched to back to Emacs, encountering many highlighting issues.

Writing Agda code in Emacs is an enjoyable experience. One can write code interactively; by placing a '?' in place of a term, compiling the code replaces the ? with a hole. When writing proofs, one can write in ?'s liberally. Within the context of a hole, Emacs will tell you the type of the term it expects to be entered into the hole. Furthermore, for complicated holes that may be composed of multiple terms, one can write in partial terms, potentially with more ?'s, and refine the proof. For holes which have only one possible solution, Agda can fill in the hole by itself by solving the constraints of a problem. There is a rudimentary auto-solver, which can work for very simple proofs, but usually fills the hole with incorrect terms.

Agda can also case split. For example, functions with an integer input can be case split into positive and negative, natural numbers can be split into zero and successor of n for some n. In this project I use the exact-split option, which requires that clauses in a definition by pattern matching hold definitionally [Teab]. I also use the safe option, which "ensure that features leading to possible inconsistencies should be disabled" [Teaa]. Finally, I used the without-k option which disables streichers k axiom, which is not compatible with univalent mathematics (stated in [Esc19a]).

Remark. My code can be viewed in html here. Any code displayed in this report links directly to the html representation. The code may be compiled in the presence of the Type Topology library (renaming the existing Integers file), and type checks on Agda version 2.6.1.2. My name is at the top of any module I have produced as part of this project. In my files I have labelled any import I have used from the TypeTopology library with a comment.

3 Natural Numbers

The natural numbers, along with addition and order are already implemented within Type Topology. Prior to working with the library, I had proved commutativity and distributivity of natural numbers,

which was set as an exercise in [Dyb18]. Once I became more familiar with Agda, I began using the Type Topology library.

Note: The blue text in this report are clickable hyperlinks that link directly to the relevant code in a html format.

The majority of work in this project builds on the Natural Numbers. The principle of Natural Induction is also used extensively.

```
data ℕ : Set₀ where
 zero : ℕ
 succ : ℕ → ℕ
induction : {A : ℕ → 𝓤 ̇} → A 0 → ((k : ℕ) → A k → A(succ k)) → (n : ℕ) → A n
induction base step 0 = base
induction base step (succ n) = step n (induction base step n)
```

Figure 3: Natural Numbers Definition and Induction Principle

We proceed by building up proofs of properties of the natural numbers, similar to how mathematicians would build mathematical foundations using the Peano Axioms, for example in [Ham82], or [Kum09]. I state the Peano Axioms here for use in this report. We use 0 as in the initial element, as in [Ham82]. I omit the vast majority of proofs in both set theory and type theory in this report.

Definition 3.0.1 (Peano Axioms)**.** $\mathbb{N}$ is a set with the following properties

(P1) 0 is an element of $\mathbb{N}$

(P2) There exists a successor function $S : \mathbb{N} \to \mathbb{N}$

(P3) For all $m, n \in \mathbb{N}$, $m = n$ if and only if $S(n) = S(m)$

(P4) For all $m \in \mathbb{N}, S(m) = 0$ is false

(P5) Let $A \subset \mathbb{N}$. If $0 \in A$, and $m \in \mathbb{N} \to S(m) \in \mathbb{N}$, then $A = N$.

The 5th axiom refers to a subset A of N. The subset A can be described as a property of $\mathbb{N}$, and elements of A can be described as natural numbers that satisfy the property A. For example, if A is the set of prime numbers, then if $n \in A$, n is a prime number. In classical mathematics, the principle of Natural induction is assumed as an axiom. In Agda, we can construct the principle of Natural induction as a function, as above.

3.1 Addition and Properties

Addition is defined recursively as follows:

1. $\forall m \in \mathbb{N},\ m + 0 = m$
2. $\forall m, n \in \mathbb{N},\ m + S(n) = S(m + n)$

In foundational set theory, properties of Natural Numbers would be proved using the Peano Axioms, with examples below.

Lemma 3.1. $\forall m \in \mathbb{N},\ 0 + m = m$

Proof. By induction on m.

- Base Case : $0 + 0 = 0$ by definition of addition
- Inductive Hypothesis : Let $k \in \mathbb{N}$, and assume that $0 + k = k$

- Inductive Step :

$$\begin{aligned} 0 + S(k) &= S(0 + k) && \text{by definition of addition} \\ &= S(k) && \text{by the inductive hypothesis} \end{aligned}$$

□

Lemma 3.2. $\forall m, n \in \mathbb{N},\ S(m) + n = S(m + n)$

Theorem 3.3. *[Commutativity of Addition of Natural Numbers]*

Let $a, b \in \mathbb{N}$. *Then* $a + b = b + a$

Proof. By induction on b.

- Base Case :

$$\begin{aligned} a + 0 &= a && \text{by the definition of addition} \\ &= 0 + a && \text{Lemma 3.1} \end{aligned}$$

- Inductive Hypothesis : Let $k \in \mathbb{N}$, and $a + k = k + a$
- Inductive Step :

$$\begin{aligned} a + S(k) &= S(a + k) && \text{by the definition of addition} \\ &= S(k + a) && \text{by the inductive hypothesis} \\ &= S(k) + a && \text{by Lemma 3.2} \end{aligned}$$

□

The same two proofs in Agda are shown in Figure 4.

The structure of induction in type theory is similar to set theory. We provide a proof of the base case, and then provide a step function that provides an inhabitant of $A\ S(k)$ given an inhabitant of $A\ k$. These proofs exhibit a use of the identity type and equational reasoning within Agda. The identity type captures the notion of equality of terms in type theory. It has the single constructor refl, which is a proof that definitionally equal terms are equal. The terms on the left can be viewed as a list of equations as we would write in set theory. Inside the brackets we give the proof that the terms on either side are equal. Hence, we can read the first line of the base step of succ-left as such: $S(x) + 0 = Sx$ *by definition.*

The ap function proves that if we apply a function to two definitionally equal terms, the outputs are definitionally equal. Hence the second line of the base case can be read as such: $x = x+0$ *by definition.* Hence, by applying the succ function to either side, we have that $S(x) = S(x+0)$. Agda would allow us to omit lines that only need refl as proof of equality. We write these lines in for readability purposes.

Suppose that we had instead defined addition on Natural Numbers as follows:

1. $\forall m \in \mathbb{N},\ 0 + m = m$
2. $\forall m, n \in \mathbb{N},\ S(n) + m = S(n + m)$

Proofs of properties would change intensionally. For example, in the proof of commutativity, $x+S(k) = S(x + k)$ would no longer hold definitionally. I initially struggled with the concept of definitional equality. This may have been due to my lack of experience in foundational mathematics in general, as opposed to experience of mathematics in type theory.

Theorem 3.4 (Associativity of Addition of Natural Numbers)**.** *Let* $x, y, z \in \mathbb{N}$. *Then* $(x + y) + z = x + (y + z)$

```
succ-left : (x y : ℕ) → succ x + y ≡ succ (x + y)
succ-left x = induction base step
  where
    base : succ x + 0 ≡ succ (x + 0)
    base = succ x + 0   ≡⟨ refl          ⟩
           succ x       ≡⟨ ap succ refl ⟩
           succ (x + 0) ∎

    step : (k : ℕ) → succ x + k ≡ succ (x + k) → succ x + succ k ≡ succ (x + succ k)
    step k IH = succ x + succ k     ≡⟨ refl ⟩
                succ (succ x + k)   ≡⟨ ap succ IH ⟩
                succ (succ (x + k)) ≡⟨ refl ⟩
                succ (x + succ k)   ∎

+-comm : (x n : ℕ) → x + n ≡ n + x
+-comm x = induction base step
  where
    base : x + 0 ≡ 0 + x
    base = zero-left-neutral x ⁻¹

    step : (k : ℕ) → x + k ≡ k + x → x + succ k ≡ succ k + x
    step k IH = x + succ k    ≡⟨ refl ⟩
                succ (x + k)  ≡⟨ ap succ IH ⟩
                succ (k + x)  ≡⟨ succ-left k x ⁻¹ ⟩
                succ k + x ∎
```

Figure 4: Property of Successor & Proof of Commutativity

The proof of associativity is already contained within Type Topology. I now list some properties of addition I have proven.

Corollary 3.4.1 (Addition and Successor)**.** *Let $x, y \in \mathbb{N}$. Then*

- $x + S(y) = S(x + y)$
- $S(x) + y = S(x + y)$

Corollary 3.4.2 (Addition Cancellable)**.** *Let $x, y, z \in \mathbb{N}$, then*

- *If $z + x = z + y$, then $x = y$*
- *If $x + z = y + z$, then $x = y$*

Corollary 3.4.3 (Sum to Zero)**.** *Let $a, b \in \mathbb{N}$. If $a + b = 0$, then $b = 0$.*

3.2 Order

Ordering on the natural numbers can be defined in two equivalent ways, with the first described in [Kum09], and the second inductively.

1. Let $m, n \in \mathbb{N}$. Then define $m \leq n : \exists k \in \mathbb{N}$ such that $m + k = n$
2. - $0 \leq n$: True
   - $S(m) \leq 0$: False
   - $S(m) \leq S(n)$: $m \leq n$

   In either case, we have $m \geq n = n \leq m$

Strict Ordering can then be defined as an embedding of the above definition:

$$m < n : S(m) \leq n,\ m > n : m < n$$

I build on the work of Escardó in [Esc19b] on the ordering of Natural Numbers, who has already implemented the above definitions in Agda.

Following is a list of results I have proved on the ordering of natural numbers.

Theorem 3.5. *Let $x, y, w, z \in \mathbb{N}$. Then*

- *$x < y$, or $x = y$, or $y < x$ [Naturals Order Trichotomous]*
- *$x \leq y$ or $y \leq x$ [$\leq$-Dichotomy]*
- *if $x = y$, then $x \leq y$*
- *if $x < y$, then $x \neq y$*
- *if $x < y$, and $w < z$, then $x + w < y + z$*
- *if $x < y$, then $x < y + z$*
- *if $x < y$, then $\exists k \in \mathbb{N}$ such that $S(k) + x = y$*
- *Let $S \subset \mathbb{N}$, then if $\exists x, y \in S$ are both least elements of S, then $x = y$*

I needed to implement the trichotomy of natural numbers to prove the result that multiplication of natural numbers is cancellable in theorem 3.6. With my rudimentary experience of Agda at this point, this proved to be challenging. The first three cases are trivial (Agda's Auto-Solver will provide easy proofs). The problem then reduces to handling $(S(x) < S(y))$ or $(S(x) = S(y))$ or $(S(y) < S(x))$. With hindsight, the solution here is to recursively call the function and use trichotomy for x and y. This is proving trichotomy inductively using Agda's built-in pattern matching. At this time I failed to see this correspondence, and wrote the proof using the principle of natural induction.

3.3 Multiplication

Theorem 3.6. *Let $x, y, z \in \mathbb{N}$, then*

$x * 0 = 0$	*[Zero Right Is Zero]*
$0 * x = 0$	*[Zero Left Is Zero]*
$x * 1 = 0$	*[Mult Right ID]*
$1 * x = 0$	*[Mult Left ID]*
$x * (y + z) = x * y + x * z$	*[Distributivity of Mult over Naturals Left]*
$(x + y) * z = x * z + y * z$	*[Distributivity of Mult over Naturals Right]*
$x * y = y * x$	*[Commutativity of Multiplication]*
$(x * y) * z = x * (y * z)$	*[Associativity Of Multiplication]*
If $z > 0$, and $x * z = y * z$, then $x = y$	*[Multiplication Cancellable]*

3.4 Division

Division is defined as follows:

Definition 3.6.1 (Division of Natural Numbers)**.** Let $x, y \in \mathbb{N}$. Then define $x \mid y$ if $\exists k \in \mathbb{N}$ such that $x * k = y$.

The following proof is an adapted version of the division theorem from [Goo19, p. 12]. I needed to adapt the proof in [Goo19]; it defines division for integers, however at this stage I am only working with natural numbers.

Theorem 3.7. *Let $a, d \in \mathbb{N}$. Then there exists a unique $q, r \in \mathbb{N}$ such that $a = q * d + r$ and $r < d$.*

Proof. Let $q \in \mathbb{N}$ be the largest such that $q \leq \frac{a}{d}$. Let $r = a - q * d$. Then $a = q * d + r$, and $r \in \mathbb{N}$.

It must be shown than $r < d$. For a contradiction, suppose that $r \geq d$. Then $\frac{r}{d} \geq 1$, and $q + 1 \leq q + \frac{r}{d} = \frac{a}{d}$. But $q + 1 \geq q$, and q was chosen to be maximal such that $q \leq \frac{a}{d}$, so we have a contradiction and $r < d$.

To prove that q and r are unique, suppose that $q, q', r, r' \in \mathbb{N}$ such that

$$a = q * d + r = q' * d + r', r, r' < d.$$

If $q = q'$, then we are done. Suppose that $q \neq q'$. Without loss of generality, suppose that $q < q'$.

We now consider the cases $r = r'$ and $r \neq r'$. If $r = r'$ then we have

$$q * d + r = q' * d + r,$$

but we also have that

$$q * d + r < q' * d + r,$$

and we have a contradiction. Now suppose that $r \neq r'$. Then we have that either $r < r'$ or $r' < r$. If $r' < r$, then $\exists k \in \mathbb{N}$ such that $q * d + k = q' * d$, and since $q < q'$, $\exists m \in \mathbb{N}$ such that $k = m * d$. But $k < d$, and $k = m * d > d$, so we have a contradiction. A similar argument gives a contradiction for $r' < r$, and so by contradiction we have that $q = q'$, which gives $r = r'$ and we have proved uniqueness. □

The proof of the division theorem was my first major obstacle of the project. I had two major difficulties when attempting this proof.

Firstly, I needed a method to find constructively "the largest $q \in \mathbb{N}$ such that $q \leq \frac{a}{d}$". These lecture notes do not provide any hint as to how one may find such a q, and do not provide evidence that such a q actually exists. The assumption of existence of this q is intuitive, but do not hold trivially. This lack of rigour makes it difficult to translate the proof into Agda, which demands every proof be fully rigorous.

My project supervisor had provided a function for finding the minimum element satisfying a property, given an inhabitant of the property within a bound, by reduction to bounded minimisation. On his suggestion, I attempted to adapt his work into a function for bounded maximisation. Unfortunately, at that time my type theory competence was not at level that allowed me to complete this adaptation.

Secondly, I had to work with subtraction on the natural numbers in order to prove the that quotient and remainder posited by the theorem are unique. I had to realign my understanding of subtraction. Instead of a function that "takes" one number away from another, I found it necessary to think of subtraction in terms of addition. That is, informally, $a - b := \exists x \in \mathbb{N}$ such that $a + x = b$. By assuming that $b < a$, we can define subtraction of Natural Numbers in Agda.

I was initially unable to implement bounded maximisation, due to lack of experience with type theory. I implemented a proof of the division theorem using natural induction. In this case, I constructed the proof by experimenting with induction on both a and d, and case splitting on the k value in the inductive step. Agda was a crucial guide; by setting up the induction, Agda's goals can provide insight into how to complete the proof. In other cases, the goal type may be misleading. A common procedure for proofs in both type theory and set theory is to break the proof into sub-proofs. Without a pen and paper draft for guidance, it is easy to write in the type of a sub-proof which has no inhabitants, and waste time trying to prove false statements.

After my work on the integers and rationals, I was able to return to naturals division with more experience. I was now able to find a maximal element satisfying a property, given a bound and proof of inhabitance of the property. I was also able to implement a proof of the division theorem that more closely follows the proof given in [Goo19]. I was able to clean up my first proof of uniqueness of the division theorem, and prove that my two proofs of the division algorithm agree.

In this section, I proved some properties of division, including one surprisingly difficult property.

```
division : (a d : ℕ) → divisiontheorem a (succ d)
division a d = induction base step a
 where
  base : Σ q ꞉ ℕ , Σ r ꞉ ℕ , (0 ≡ q * succ d + r) × (r < succ d)
  base = 0 , (0 , (I , II))
   where
    I : 0 ≡ 0 * succ d + 0
    I = 0         ≡⟨ refl                                  ⟩
        0 + 0     ≡⟨ ap (0 +_) (zero-left-is-zero d ⁻¹) ⟩
        0 + 0 * d ∎

    II : 0 < succ d
    II = unique-to-𝟙 (0 < succ d)

  step : (k : ℕ)
       → Σ q ꞉ ℕ , Σ r ꞉ ℕ , (k ≡ q * succ d + r) × (r < succ d)
       → Σ q ꞉ ℕ , Σ r ꞉ ℕ , (succ k ≡ q * succ d + r) × (r < succ d)
  step k (q , r , e , l) = helper (<-split r d l)
   where
    helper : (r < d) + (r ≡ d) → Σ q ꞉ ℕ , Σ r ꞉ ℕ , (succ k ≡ q * succ d + r) × (r < succ d)
    helper (inl x) = q , succ r , ap succ e , x
    helper (inr x) = succ q , 0 , I , unique-to-𝟙 (0 < succ d)
     where
      I : succ k ≡ succ q + succ q * d
      I = succ k                          ≡⟨ ap succ e                                          ⟩
          succ (q + q * d + r)            ≡⟨ ap succ (ap (q + q * d +_) x)                      ⟩
          succ (q + q * d + d)            ≡⟨ ap succ (addition-associativity q (q * d) d)       ⟩
          succ (q + (q * d + d))          ≡⟨ succ-left q (q * d + d) ⁻¹                         ⟩
          succ q + (q * d + d)            ≡⟨ ap (succ q +_) (ap (_+ d) (mult-commutativity q d)) ⟩
          succ q + (d * q + d)            ≡⟨ ap (succ q +_) (addition-commutativity (d * q) d)  ⟩
          succ q + (d + d * q)            ≡⟨ ap (succ q +_) (mult-commutativity d (succ q))     ⟩
          succ q + succ q * d             ∎
```

Figure 5: Division of Natural Numbers

Theorem 3.8. *Let $x, y \in \mathbb{N}$. Suppose $x \mid y$ and $y \mid x$. Then $x = y$.*

Informally, the proof on pen and paper is shown as follows:

Proof. Since $x \mid y$ and $y \mid x$, $\exists a, b \in \mathbb{N}$ such that $a * x = y$ and $b * y = x$, and we have that

$$a * b * y = y$$

Since a, b are natural numbers, the only possibility is that $a * b = 1$, and so $a = 1$ and $b = 1$, and therefore $x = y$.

□

The last line of the proof is true, but it is not *trivially* true, and certainly does not translate directly into an Agda proof. In Agda, my proof that division is anti-symmetric requires that multiplication is cancellable. My proof that multiplication is cancellable requires a proof that multiplication is compatible with order, as well as the proof that natural numbers are a trichotomy with respect to order.

The results required to complete "trivial proofs" are often surprising. In classical mathematics, many details are usually omitted or left to the reader. It makes sense to omit such details, it isn't practical for a mathematician to verify that every aspect of the foundation of mathematics is true when writing proofs. With omissions, we lose rigour. It would still be nice to know for sure that *every* assumption made is *actually* true, regardless of its triviality. Using a proof assistant such as Agda with the safe parameter allow us be sure that a proof is fully rigorous, with every sub-proof readily available.

```
division-is-prop : (a d : ℕ) → is-prop (divisiontheorem a d)
division-is-prop a d (q₀ , r₀ , α , αₚ) (q₁ , r₁ , β , βₚ) = to-subtype-≡ I II
 where
  I : (q : ℕ) → is-prop (Σ r ꞉ ℕ , (a ≡ q * d + r) × (r < d))
  I q (r₀ , δ) (r₁ , γ) = to-subtype-≡ (λ r → ×-is-prop ℕ-is-set (<-is-prop-valued r d)) remainders-equal
   where
    remainders-equal : r₀ ≡ r₁
    remainders-equal = addition-left-cancellable r₀ r₁ (q * d) i
     where
      i : q * d + r₀ ≡ q * d + r₁
      i = q * d + r₀ ≡⟨ pr₁ δ ⁻¹ ⟩
          a          ≡⟨ pr₁ γ ⟩
          q * d + r₁ ∎

  assumption : q₀ * d + r₀ ≡ q₁ * d + r₁
  assumption = α ⁻¹ · β

  II-abstract : (q q' r r' : ℕ) → q * d + r ≡ q' * d + r' → q < q' → r < d → q ≡ q'
  II-abstract q q' r r' e l₁ l₂ = 𝟘-elim (not-less-than-itself (d * succ k) vii)
   where
    i : Σ k ꞉ ℕ , (succ k) + q ≡ q'
    i = subtraction'' q q' l₁

    k : ℕ
    k = pr₁ i

    μ : (succ k) + q ≡ q'
    μ = pr₂ i

    ii : q * d + r ≡ q * d + ((succ k) * d + r')
    ii = q * d + r                  ≡⟨ e ⟩
         q' * d + r'                ≡⟨ ap (λ - → - * d + r') (μ ⁻¹) ⟩
         ((succ k) + q) * d + r'    ≡⟨ ap (_+ r') (distributivity-mult-over-nat' (succ k) q d) ⟩
         (succ k) * d + q * d + r'  ≡⟨ ap (_+ r') (addition-commutativity ((succ k) * d) (q * d)) ⟩
         q * d + (succ k) * d + r'  ≡⟨ addition-associativity (q * d) ((succ k) * d) r' ⟩
         q * d + ((succ k) * d + r') ∎

    iii : r' + d * (succ k) ≡ r
    iii = r' + d * succ k        ≡⟨ ap (r' +_) (mult-commutativity d (succ k)) ⟩
          r' + (succ k) * d      ≡⟨ addition-commutativity r' ((succ k) * d) ⟩
          (succ k) * d + r'      ≡⟨ addition-left-cancellable r ((succ k) * d + r') (q * d) ii ⁻¹ ⟩
          r                      ∎

    iv : d * (succ k) ≤ r
    iv = cosubtraction (d * succ k) r (r' , iii)

    v : r < d * succ k
    v = less-than-pos-mult r d k l₂

    vii : d * succ k < d * succ k
    vii = division-is-prop-lemma (d * succ k) r (d * succ k) iv v

  II : q₀ ≡ q₁
  II = f (nat-order-trichotomous q₀ q₁)
   where
    f : (q₀ < q₁) + (q₀ ≡ q₁) + (q₁ < q₀) → q₀ ≡ q₁
    f (inl z)       = II-abstract q₀ q₁ r₀ r₁ assumption z αₚ
    f (inr (inl z)) = z
    f (inr (inr z)) = II-abstract q₁ q₀ r₁ r₀ (assumption ⁻¹) z βₚ ⁻¹
```

Figure 6: Division is a proposition

3.5 Highest Common Factor

I define the highest common factor (HCF) adapted from [Goo19].

Definition 3.8.1. Let $a, b \in \mathbb{N}$. Then the highest common factor h of a and b is the largest natural number h that is a common factor, denoted $h = \text{hcf}(a, b)$.

Informally, the notion of "largest natural number" in the case of the HCF is equivalent to saying that

```
division' : (a d : ℕ) → Σ q ꞉ ℕ , Σ r ꞉ ℕ , (a ≡ q * (succ d) + r) × (r < (succ d))
division' 0 d     = 0 , (0 , (I , II))
 where
  I : 0 ≡ 0 * succ d + 0
  I = 0         ≡⟨ refl                                ⟩
      0 + 0     ≡⟨ ap (0 +_) (zero-left-is-zero d ⁻¹) ⟩
      0 + 0 * d ∎

  II : 0 < succ d
  II = unique-to-𝟙 (0 < succ d)
division' (succ a) d = f (maximal-from-given' (λ - → - * succ d ≤ succ a) (succ a) (λ q → ≤-decidable (q * succ d)
 where
  i : (0 + 0 * d) ≤ succ a
  i = transport (_≤ succ a) (zero-left-is-zero (succ d) ⁻¹) (zero-minimal (succ a))

  ii : Σ k ꞉ ℕ , (k * succ d ≤ succ a) × (k ≤ succ a)
  ii = 0 , i , zero-minimal (succ a)

  f : Σ q ꞉ ℕ , q ≤ succ a × (q * succ d) ≤ succ a × ((n : ℕ) → n ≤ succ a → n * succ d ≤ succ a → n ≤ q)
    → Σ q ꞉ ℕ , Σ r ꞉ ℕ , (succ a ≡ q * succ d + r) × (r < succ d)
  f (q , l₁ , l₂ , qf) = g (subtraction (q * succ d) (succ a) l₂)
   where
    g : Σ r ꞉ ℕ , r + q * succ d ≡ succ a
      → Σ q ꞉ ℕ , Σ r ꞉ ℕ , (succ a ≡ q * succ d + r) × (r < succ d)
    g (r , rf) = q , r , I , II (order-split r (succ d))
     where
      I : succ a ≡ q * succ d + r
      I = succ a         ≡⟨ rf ⁻¹                                ⟩
          r + q * succ d ≡⟨ addition-commutativity r (q * succ d) ⟩
          q * succ d + r ∎

      II : r < succ d + r ≥ succ d → r < succ d
      II (inl z) = z
      II (inr z) = 𝟘-elim (not-less-than-itself q IV)
       where
        III : succ q * succ d ≤ succ a
        III = transport₂ _≤_ α (addition-commutativity (q * succ d) r · rf) β
         where
          α : q * succ d + succ d ≡ succ q * succ d
          α = q * succ d + succ d     ≡⟨ ap (q * succ d +_) (mult-left-id (succ d) ⁻¹) ⟩
              q * succ d + 1 * succ d ≡⟨ distributivity-mult-over-nat' q 1 (succ d) ⁻¹ ⟩
              (q + 1) * succ d        ≡⟨ refl ⟩
              succ q * succ d ∎

          β : q * succ d + succ d ≤ q * succ d + r
          β = ≤-n-monotone-left (succ d) r (q * succ d) z

        IV : q < q
        IV = qf (succ q) (product-less-than-cancellable (succ q) d (succ a) III) III

division-agrees-with-division' : (x y : ℕ) → division x y ≡ division' x y
division-agrees-with-division' a d = division-is-prop a (succ d) (division a d) (division' a d)
```

Figure 7: Division using bounded maximisation

the HCF can be divided by any given common divisor of two natural numbers, which is how this concept is formalised. The HCF was a challenging part of this project.

Recall that for any proposition we write in univalent type theory, we must prove that is it a proposition. Earlier, I showed a proof that the statement "5 is not a prime number" is a proposition, in section 2. In that case, I did not provide a proof of the statement. In fact, I wouldn't have been able to produce an inhabitant of that proposition, because it isn't true. A full proof of a proposition consists of a "proof", or algorithm which produces an inhabitant of the proposition, *and* a proof that the proposition is a subsingleton. In this case, the proposition is the existence of a HCF, and the algorithm is one that produces the HCF. The standard method of obtaining the HCF is Euclid's Algorithm. My challenge, therefore, was to translate Euclid's algorithm into Agda, and prove that it outputs the HCF, which took a substantial amount of time.

I now state an adapted lemma from [Goo19] without proof, and then an adapted version of Euclid's Algorithm for Natural Numbers.

Lemma 3.9. *Let $a, b, q, r \in \mathbb{N}$. Suppose that $a = q * b + r$. Then $hcf(a, b) = hcf(b, r)$.*

Theorem 3.10. *Euclid's Algorithm, adapted from [Goo19]*

Let $a, b \in \mathbb{N}$, with $a \geq b$. Let q, r be the quotient and remainder obtained by applying theorem 3.7 to a and b. Then $a = q * b + r$

Let h be the output of the following recursive function:

$$hcf(a,\ b) = \begin{cases} hcf(b, r) & r \geq 0 \\ b & r = 0 \end{cases}$$

Then $h = hcf(a, b)$.

Proof. The algorithm terminates at any step where $r_m = 0$, for some mth step of the algorithm. If this is case, then we have that $a_{m-1} = a_m * q_m$, so $a_{m-1} \mid a_m$ by the definition of division, and therefore $hcf(a_m, a_{m-1}) = a_{m-1} = h$. Consider step k, where $1 \leq k < m$. We have that $a_{k-1} = a_k * q_k + a_{k+1}$. In the initial case, we have that $a_1 = a_2 * q_1 + r_1$, where $a_1 = a, a_2 = b$. We now have a chain of equalities, and

$$h = a_m = hcf(a_{m-1}, a_m) = \cdots = hcf(a_{k-1}, a_k) = \cdots = hcf(a_1, a_2) = hcf(a, b).$$

Note that $a_1 > a_2 > \cdots$, so we must eventually reach some step where $a_{k+1} = 0$.

□

The first challenge is the algorithm. Agda requires any function called recursively to have an argument strictly smaller than the original argument (for two or more arguments this may happen lexicographically, see [Teac]). If we construct a function identical to the one above in Agda, it will compute our candidate for the HCF, but Agda's termination checker will flag this function. Agda cannot be sure that this function terminates, because it can not verify without help that this algorithm reduces at each step. I could not translate the above proof into Agda using pattern matching or the principle of induction without failing the termination check.

Using suggestions from my supervisor and the Agda club, I knew I needed to use Strong induction along with the second condition of theorem 3.7 that requires the remainder to be strictly below the divisor. Fortunately, in TypeTopology, strong induction is already implemented and named course of values induction. A lecture from Cornell University [Unkb], based on [Leh17], provides a proof of Euclid's algorithm using Strong induction. I was able to use this as a framework to implement the HCF into Agda, which also required a lemma concerning a property of division. The proof of the lemma is omitted in the report to save space.

Lemma 3.11. *Let $a, b, c, d \in \mathbb{N}$. Suppose $a * b = a * c + d$. Then $a \mid d$.*

This lemma serves as an example of a proof (HCF) requiring intermediate results that are not trivial. I completed this proof of this lemma using a pen and paper draft for guidance. The proof is intuitive and not difficult, but it was time-consuming to write and requires intermediate results itself (from theorem 3.5). It is generally difficult to predict that a result such as lemma 3.11 will be necessary, and writing proofs in type theory generally requires proving sub-proofs. By writing the proof of lemma 3.11 explicitly and not assuming its existence, we can be sure that the proof of HCF is sound; there is no room in the framework to build on false assumptions.

```
HCF : (a b : ℕ) → Σ h ꞉ ℕ , is-hcf h a b
HCF = course-of-values-induction (λ n → (b : ℕ) → Σ h ꞉ ℕ , is-hcf h n b) step
 where
  step : (n : ℕ) → ((m : ℕ) → m < n → (b : ℕ) → Σ h ꞉ ℕ , is-hcf h m b) → (b : ℕ) → Σ h ꞉ ℕ , is-hcf h n b
  step zero IH b = b , ((zero , refl) , 1 , refl) , (λ x → pr₂)
  step (succ n) IH b = I (division b n)
   where
    I : Σ q ꞉ ℕ , Σ r ꞉ ℕ , (b ≡ q * succ n + r) × (r < succ n) → Σ h ꞉ ℕ , is-hcf h (succ n) b
    I (q , r , e₀ , l) = II (IH r l (succ n))
     where
      II : Σ h ꞉ ℕ , is-hcf h r (succ n) → Σ h ꞉ ℕ , is-hcf h (succ n) b
      II (h , ((x , xₚ) , y , yₚ) , γ) = h , ((y , yₚ) , i) , ii
       where
        i : h ∣ b
        i = (q * y + x) , e₁
         where
          e₁ : h * (q * y + x) ≡ b
          e₁ = h * (q * y + x)          ≡⟨ distributivity-mult-over-nat h (q * y) x      ⟩
               h * (q * y) + h * x      ≡⟨ ap (λ z → h * (q * y) + z) xₚ                 ⟩
               h * (q * y) + r          ≡⟨ ap (_+ r) (mult-associativity h q y) ⁻¹       ⟩
               h * q * y + r            ≡⟨ ap (λ z → z * y + r) (mult-commutativity h q) ⟩
               q * h * y + r            ≡⟨ ap (_+ r) (mult-associativity q h y)          ⟩
               q * (h * y) + r          ≡⟨ ap (λ z → q * z + r) yₚ                       ⟩
               q * succ n + r           ≡⟨ e₀ ⁻¹ ⟩
               b                        ∎

        ii : (f : ℕ) → is-common-divisor f (succ n) b → f ∣ h
        ii f ((α , αₚ) , β , βₚ) = γ f ((hcflemma f β (q * α) r e₂) , (α , αₚ))
         where
          e₂ : f * β ≡ f * (q * α) + r
          e₂ = f * β           ≡⟨ βₚ                                          ⟩
               b               ≡⟨ e₀                                          ⟩
               q * succ n + r  ≡⟨ ap (λ z → q * z + r) (αₚ ⁻¹)                ⟩
               q * (f * α) + r ≡⟨ ap (_+ r) (mult-associativity q f α ⁻¹)     ⟩
               q * f * α + r   ≡⟨ ap (λ z → z * α + r) (mult-commutativity q f) ⟩
               f * q * α + r   ≡⟨ ap (_+ r ) (mult-associativity f q α)       ⟩
               f * (q * α) + r ∎
```

Figure 8: HCF

The second challenge is proving that the HCF is a proposition. More precisely, we want to prove that the statement "Given $a, b \in \mathbb{N}$, there exists a $d \in \mathbb{N}$ such that d is the HCF of a and b" is a proposition. HCF is defined as a Sigma Type, or a dependent pair. To prove equality of a dependent pair, then given two instances of the pair such that the first projections are equal, we need to know that the second projections (the type family, or dependent function) are also equal (see [Uni13, p. 109], or [Esc19a]). In the context of this example, this means that given $a, b \in \mathbb{N}$, we need to show that any two HCF's h and h' of a and b are equal, and also that the statement "h is a common factor of a, b" is a proposition. This statement itself consists of an independent pair of statements, "h is a common factor of a, b", and "given any common factor c of a, b, then "$c \mid h$". The second statement here requires the use of Function Extensionality.

Assuming FE, the HCF problem reduces to showing that division is a proposition. One can see from the definition that given a, b, division is a dependent pair of a Natural Number k, and a proof that $a * k = b$. In this case, we need to know that equality of natural numbers is a proposition, i.e that given $x, y \in \mathbb{N}$, the statement "$x = y$" is a proposition. If we generalize this idea to some arbitrary type A instead of $\mathbb{N}$, what we have is the idea of types as sets. As stated in [Esc19a], "a type is defined to be a set if there is at most one way for any two of its elements to be equal". In this case, in Type Topology it has already been proven that $\mathbb{N}$ is a set, and therefore it follows that equality of the naturals is a proposition. In [Uni13, p. 140], it's noted that this notion of the set is not the same as the sets used in classical mathematics, it's more similar to the sets used in structural mathematics

and category theory.

I use the implementation of FE already developed in [Esc19b] in this project, which obtains FE as a result of the Univalence axiom. For the purposes of learning how to implement mathematical proofs as in Agda, we can treat Function Extensionality as a black box axiom that allows us to prove that pointwise-equal functions are equal, respecting that this black box is rooted in the Univalent Foundations of Mathematics. The machinery to prove equality of Sigma types is also provided by TypeTopology.

```
is-hcf-is-prop : Fun-Ext → (d x y : ℕ) → is-prop (is-hcf (succ d) x y)
is-hcf-is-prop fe d x y p q = ×-is-prop (is-common-divisor-is-prop d x y) g p q
  where
    h : (f : ℕ) → is-common-divisor f x y → is-prop (f ∣ succ d)
    h 0        i x = 𝟘-elim (zero-does-not-divide-positive d x)
    h (succ f) i   = f ∣ (succ d) -is-prop

    g : is-prop ((f : ℕ) → is-common-divisor f x y → f ∣ succ d)
    g p' q' = Π₂-is-prop fe h p' q'
```

Figure 9: Is-HCF is a proposition

```
has-hcf-is-prop : Fun-Ext → (x y : ℕ) → is-prop (has-hcf x y)
has-hcf-is-prop fe x y (a , p , p') (b , q , q') = to-subtype-≡ I II
 where
  I : (d : ℕ) → is-prop (is-hcf (succ d) x y)
  I d = is-hcf-is-prop fe d x y

  II : a ≡ b
  II = succ-lc (∣-anti (succ a) (succ b) α β)
   where
    α : succ a ∣ succ b
    α = q' (succ a) p

    β : succ b ∣ succ a
    β = p' (succ b) q
```

Figure 10: Has-HCF is a proposition

We can only prove that HCF is a proposition when we restrict the potential HCF to be greater than zero. As per the definition of the highest common factor, h is required to be a common factor of a and b, which requires that h is a factor of a. "$0 \mid a$" is not a proposition in type theory. There are infinitely many $x \in \mathbb{N}$ such that $0 * x = b$, and therefore we cannot guarantee uniqueness of x for $0 \mid a$. This is not a problem, as we generally does not divide by 0. We do this by applying the succ constructor to the HCF in the statement that posits that the HCF exists. We also use this method looking forward to restrict the value of the denominator of a rational number to be greater than one.

To summarise, the function HCF outputs the HCF of two given Natural Numbers. has-hcf is a statement that posits the existence of a highest common factor greater than 0 for any two natural numbers. has-hcf-is-prop is a proof that the statement is in fact a proposition (speaking univalently), which guarantees that for all $a, b \in \mathbb{N}$, if we have two HCF's of a and b, the two HCF's are equal.

4 Integers

There are different ways to implement the Integers. Naively, one could use the Natural Numbers with a positive or negative sign, but this implementation introduces the ambiguity of positive and negative zero. To handle the ambiguity, an easy solution is to use the constructors pos and negsucc, where for $x \in \mathbb{N}$, we think of negsucc x as - S(x). This approach is the one used by the Cubical Agda library, and I adopt the same approach in this project.

Alternatively, one could define the integers as a Set Quotient as in [Uni13, p. 263], as a pair of Natural Numbers and an equivalence relation. There are disadvantages and advantages to each approach, some proofs may be easier to implement depending on the chosen approach. I chose the Inductive approach as it was more intuitive to me. With hindsight it might have been a better choice to use the Set Quotient implementation. Working on the Integers as a Set Quotient might have provided a better

foundation for me to work on the Rationals, and as an extension of the project it might explore re-implementing the integers to explore if the Set Quotient integers result in a cleaner implementation of the Rationals.

With implementations of Natural Numbers, HCF, being Coprime and a basic definition of the Integers, I naively wanted to work on the Rationals numbers directly. It quickly became apparent that this was not feasible. The Rational numbers will be defined as an Integer and Natural Number pair, and so proofs of properties of the Rationals requires proofs of properties of the Integers. Proving properties of the integers is not much more difficult than proving properties of the naturals numbers, with the key difference being the extra negative case to consider. Functions with an integer domain usually have to pattern match on at least two cases (positive and negative), with extra cases if considering zero is necessary. A particularly egregious example is the proof that the integers are trichotomous with respect to order, which considers 16 cases. Another distinction of proofs on the integers is the use of the induction principle. Proofs that require induction now require an extra step function to handle integers less than 0.

A second approach is to use the Principle of Natural Induction twice. Create two auxiliary functions, where the target for induction is replaced with only positive integers in the first auxiliary function, and by only negative integers in the second auxiliary function. By proving the property for the two auxiliary functions, the final proof then follows trivially.

The first approach is the intuitive extension of the inductive principle for natural numbers and subjectively may be more aesthetic than the second approach, but the second approach seems to result in more concise proof overall, at the expense of having to write two extra auxiliary functions for each induction. The Cubical Agda library contains some properties of addition, but has no implementation of multiplication and its properties. To prove properties of addition, the Cubical Agda library utilises an abtracted version of induction for particular properties, seen in [VM]. As an extension of this project to simplify the code, I could rewrite many of my proofs by abstracting proofs of some properties and trying to reduce similar code, I don't think this is necessary. I don't foresee too many occasions to use the abstracted code, and subjectively it feels less readable.

I have implemented following definitions and theorems related to the integers in Agda.

Definition 4.0.1.

- Addition - $+$
- Multiplication - $*$
- Negation - $(-)$
- Absolute Value - $||$
- Successor and Predecessor - $succ\mathbb{Z}$ and $pred\mathbb{Z}$
- Positivity Property
- Negativity Property
- "Is zero" property
- "Greater than zero"property
- Order - $<, \leq$

The following is not a complete list for brevity.

Theorem 4.1. *Let* $x, y, w, z \in \mathbb{Z}$. *Then*

- $x + (pos\ 0) = x$, *and* $x = (pos\ 0) + x$
- $succ\mathbb{Z}(x) + y = succ\mathbb{Z}(x + y) = x + succ\mathbb{Z}(y)$

- $pred\mathbb{Z}(x) + y = pred\mathbb{Z}(x + y) = x + pred\mathbb{Z}(y)$
- *If* $succ\mathbb{Z}(x) = succ\mathbb{Z}(y)$, *then* $x = y$
- *If* $pred\mathbb{Z}(x) = pred\mathbb{Z}(y)$, *then* $x = y$
- $x + y = y + x$
- $(x + y) + z = x + (y + z)$
- *If* $z + x = z + y$, *then* $x = y$
- $x * (pos\ 0) = pos\ 0 = (pos\ 0) * x$
- $x * (pos\ 1) = x = (pos\ 1) * x$
- $-(x + y) = (-x) + (-y)$
- $(x + y) * z = (x * z) + (y * z)$
- $x * y = y * x$
- $-(-x) = x$
- $(-x) * y = -(x * y) = x * (-y)$
- $(x * y) * z = x * (y * z)$
- *Addition of positive integers is equivalent to Addition of Natural Numbers*
- *Multiplication of positive integers is equivalent to Addition of Natural Numbers*
- $(x + (-x)) = (pos\ 0)$
- *If* $x < y$ *and* $y < z$, *then* $x < z$
- $x < y$ *or* $x = y$ *or* $y < x$
- *If* $(pos\ 0) < z$, *and* $x < y$, *then* $x * z < y * z$
- *If* $(pos\ 0) < z$, *and* $x * z < y * z$, *then* $x < y$
- *If* $z < (pos\ 0)$, *and* $x < y$, *then* $y * z < x * z$
- *If* $z \neq (pos\ 0)$, *and* $x * z = y * z$, *then* $x = y$

It was already proven in TypeTopology that $\mathbb{N}$ is a set, now I prove that $\mathbb{Z}$ is a set . I can adopt the same strategy that Escardó used in [Esc19b] to show that $\mathbb{N}$ is a set: Firstly, prove that $\mathbb{Z}$ is discrete (a type is discrete if it has decidable equality), and then use the proof that discrete types are sets.

5 Rationals

The rational numbers are more difficult to implement compared to the integers and natural numbers. The rational numbers can described as a set of equivalence classes, each of which has a canonical representation. Instead of working with equivalence classes, I work with the canonical rational numbers.

Definition 5.0.1 (Rational Numbers)**.** The Rationals, denoted $\mathbb{Q}$, are defined as an independent integer and natural number pair $x \in \mathbb{Z}, y \in \mathbb{N}$, with the condition that $|x|$ is coprime to y.

The rational numbers pose a number of challenge not present in the previous sections:

- Note that the natural number represents the number above it, to avoid the division by 0. This incrementation by one complicates definitions and proofs of the rationals, for example addition of two rationals must first correct the denominators, apply the addition and then decrement the resultant denominator.

- With both the Natural Numbers and Integers involved, we have to be fully rigorous with the operators. Addition on the Naturals and Addition on the integers are distinct operators, we cannot overload the operator as a classical mathematician might want. This usually means converting between Naturals and Integers as necessary within proofs. Addition and multiplication of rationals is defined in terms of addition and multiplication of integers and natural numbers, so is it common to deal with expressions with 4 terms or more; in this proof for example. As the example shows, this requires a re-arrangement of the terms to be able to apply the necessary additive and multiplicate laws. On pen and paper, these re-arrangements are trivial, and are used without justification. It would be impractical for a mathematician to justify every use of an additive or multiplicative rule. In type theory, each step of a re-arrangement must be justified. We can try to reduce the amount of lines needed for re-arrangements by using separate re-arrangement functions, but usually any particular re-arrangement is only used once or twice. It is time consuming to write the code for these re-arrangements of terms, but it's a small price to pay for the benefit of being fully rigorous with proofs.
- Proving equality of terms of the rationals is more difficult than equality of Naturals or Integers. Proving properties of the rationals requires proving equality of a dependent pair. The dependent pair consists of the integer and natural number pair, with a dependent function that requires the integer and natural number pair to be in their lowest terms. Usually, this equality must be proved after an operator has been applied to two rationals, and the result has been normalised by a "$to\mathbb{Q}$" operation, which brings a (potentially non-canonical) rational number to the canonical representation. A large proportion of the remainder of the project was spent tackling this problem.
- Most proofs require a lot of information. Specifically, keeping track of the normalisation factor when reducing a rational to its lowest terms is necessary. The types of terms within a proof become more complicated; since the Rationals are defined as a dependent pair we have to work with the projections of the pair. In general, proofs of properties of the rationals are less readable than proofs on the naturals and integers. It is very necessary to have a clear idea of the structure of a proof, and pen and paper drafts are very useful.

To compartmentalise the challenge, I defined a version of the rationals without the condition that the integer and natural number pair are in their lowest terms. The motivation behind this was the idea that I could implement proofs of the (potentially) non-canonical rationals, with the hope that it is then easier to extend these proofs to the canonical rationals.

Definition 5.0.2. The (potentially) non-canonical rationals are defined as an independent pair

$$\mathbb{Q}_n = \{(x, y) : \ x \in \mathbb{Z}, y \in \mathbb{N}\}$$

.

Definition 5.0.3 (Addition of free rationals)**.** Let $(x, a), (y, b) \in \mathbb{Q}_n$, where $x, y \in \mathbb{Z}, a, b \in \mathbb{N}$. Then define

$$(x, a) + (y, b) = ((x\ \mathbb{Z}^*\ (pos\ S(b)))\ \mathbb{Z}^+\ (y\ \mathbb{Z}^*\ (pos\ S(a))), pred(S(a)\ \mathbb{N}^*\ S(b))$$

Definition 5.0.4 (Multiplication of free rationals)**.** Let $(x, a), (y, b) \in \mathbb{Q}_n$, where $x, y \in \mathbb{Z}, a, b \in \mathbb{N}$. Then define

$$(x, a) * (y, b) = (x\ \mathbb{Z}^*\ y, pred(S(a)\ \mathbb{N}^*\ S(b))$$

Definition 5.0.5 (Order of free rationals)**.** Let $(x, a), (y, b) \in \mathbb{Q}_n$, where $x, y \in \mathbb{Z}, a, b \in \mathbb{N}$. Then define

$$(x, a) < (y, b) = (x\ \mathbb{Z}^*\ (pos\ S(b))\ \mathbb{Z}^<\ (y\ \mathbb{Z}^*\ (pos\ S(a))$$

Theorem 5.1. *Let $p, q, r \in \mathbb{Q}_n$. Then,*

$$p + q = q + p \qquad \textit{[Addition Commutative on free rationals]}$$
$$(p + q) + r = p + (q + r) \qquad \textit{[Addition Associative on free rationals]}$$
$$p * q = q * p \qquad \textit{[Multiplication Commutative on free rationals]}$$
$$(p * q) * r = p * (q * r) \qquad \textit{[Multiplication Associative on free rationals]}$$

With a goal of implementing addition and multiplication on the canonical rationals, we need a function that reduces a rational number to its lowest terms.

Definition 5.1.1. Let $q \in \mathbb{Q}_n$. Then define $to\mathbb{Q}(q)$ to be the function from $\mathbb{Q}_n \to \mathbb{Q}$ that sends a free rational number to its canonical representation in its equivalence class.

This definition is implemented using a function that divides two numbers by their HCF, with the proof that this results in a pair of coprime natural numbers.

Lemma 5.2. *Let $x, y \in \mathbb{N}$. Then there exists $h, x, y \in \mathbb{N}$ such that $h * a = x$, $h * b = y$ and x is coprime to y.*

We can now define addition, multiplication and order of rational numbers.

Definition 5.2.1. Let $((x, a), p), ((y, b), q) \in \mathbb{Q}$, where $x, y \in \mathbb{Z}$, $a, b \in \mathbb{N}$, and p, q are proofs that the respective rational number is in its lowest terms. Then,

$$\text{Addition of rationals} \qquad ((x, a), p) + ((y, b), q)) = to\mathbb{Q}\ ((x, a)\ \mathbb{Q}_n^+\ (y, b))$$
$$\text{Multiplication of rationals} \qquad ((x, a), p) * ((y, b), q)) = to\mathbb{Q}\ ((x, a)\ \mathbb{Q}_n^*\ (y, b))$$
$$\text{Order of rationals} \qquad ((x, a), p) < ((y, b), q)) = (x, a)\mathbb{Q}_n^<(y, b)$$

My representation of the rationals contains a proof that the rational is in its lowest terms, and addition of rational numbers does not necessarily preserve this property. For addition and multiplication, we must therefore normalise the resultant rational number. To define order, we can simply ignore the proof that the rationals are in the lowest terms, as being in lowest terms has no impact on the evaluation of the order of two rational numbers.

With these functions on the free rationals, there still remains the challenge of handling equality of the canonical rationals, $\mathbb{Q}$. Consider associativity of addition on $\mathbb{Q}$, i.e $((p + q) + r = p + (q + r)$. On the left hand side, we add p and q and normalise the result, and then add r to this and normalise the subsequent result. On the right hand side, the normalisation of terms happen from right to left. The challenge, therefore, is to prove that the result of addition is not affected by when the normalisation of the rational occurs. This was a particularly difficult challenge for me, and required input of both my supervisor and co-supervisor. Furthermore, it resulted it many of the functions defined previously. The following describes the setup required to be able to complete a proof of associativity of the rationals.

I needed to introduce an equivalence relation of the rationals. That is, the notion of two free rational numbers $(x, a), (y, b)$ being equivalent.

Definition 5.2.2. Let $(x, a), (y, b) \in \mathbb{Q}_n$. Then define $(x, a) \approx (y, b)$ as $x\ \mathbb{Z}^*\ (pos\ S(b)) = y\ \mathbb{Z}^*\ (pos\ S(a))$.

I also needed to prove the following lemma:

Lemma 5.3. *Let $(x, a), (y, b) \in \mathbb{Q}_n$, then $(x, a) \approx (y, b)$ if and only if $to\mathbb{Q}\ (x, a) = to\mathbb{Q}\ (y, b)$.*

The right to left implication can be shown without too much difficulty (see sub-proof II). The other direction required more work. Firstly, it required an extension of the function that normalises a rational number ($to\mathbb{Q}$), and secondly, it required an auxilliary proof:

Lemma 5.4. *Let $p, q \in \mathbb{Q}_n$. Suppose $p \approx q$, and both p and q are in their lowest terms. Then $p = q$.*

It required an extension of $to\mathbb{Q}$ in order to keep track of the data that was previously being lost to normalisation. When dividing by the highest common factor, it is necessary to keep track of this highest common factor. This was necessary because the assumed equivalence relation applies to free rational before normalisation. Suppose we have $(x, a), (y, b) \in \mathbb{Q}_n$, and we know that $x * (pos\ S(a)) \equiv y * (pos\ S(b))$. We are trying to prove equality of $((x', a'), p)$, $((y', b'), q)$, where (x', a'), (y', b') are the canonical representations of (x, a), (y, b) respectively, and p, q are proofs that the respective pair are in their lowest terms. We have no data connecting (x', a') and (y', b') directly, but we do know that $x * (pos\ S(a)) \equiv y * (pos\ S(b))$. Therefore, we need to use the fact that $(x, a) = ((pos\ m)\ \mathbb{Z}^*\ x', m\ \mathbb{N}^*\ a')$ for some $m \in \mathbb{N}$, with a similar result for (y, b). For this reason, I needed to keep track of the factor that relates a free rational to its canonical representation.

The proof of the lemma was more difficult than the extension. In order to prove that $p = q$, I needed a further auxilliary proof.

Lemma 5.5. *Let $a, b, c \in \mathbb{N}$. Suppose a is coprime to b, and suppose that $a \mid (b\ \mathbb{N}^*\ c)$, then $a \mid c$.*

Discussion with Alex Rice in the Agda Club led to a pen and paper proof of this lemma, but it required Bezout's Lemma. At this stage, I had only implemented the HCF on the natural numbers, and not the integers. Clearly, Bezout's lemma requires the use of integers and the aforementioned extension became necessary at this point. There was some time lost due to my approach at this stage of the project. Without fully considering that Bezout's Lemma required only the Extended Euclidean Algorithm, I implemented Division and HCF for the Integers. This was time consuming, and ultimately didn't provide the information I needed. There was similarity to the $to\mathbb{Q}$ data requirement; I would not be able to implement Bezout's Lemma without access to the data from the proof of HCF on natural numbers. I re-implemented the HCF on the naturals, by duplicating the code from my previous proof of the Natural Numbers, and extending it to include Euclid's Extended Algorithm. An extension of the project could tidy up duplicate code related to the HCF. The extra code for division of integers type-checks, but is not relevant to the remainder of this project. With Bezout's Lemma implemented, I could prove the right-to-left implication of lemma 5.3.

With this equivalence relation in place, I was still unable to directly prove associativity of addition of canonical rational numbers. I needed some more proofs related to the equivalence relation. In the case of the associativity, we want to prove that the left hand side is equal to the right hand side, where each side is a rational number. In the case of natural numbers, we can use equational reasoning to directly prove the equality of each side. We cannot directly prove the equality on the rationals using equational reasoning. The idea is that we can use a version of equational reasoning, but we instead use the equivalence relation, and then elevate equivalence to equality using lemma 5.3.

Theorem 5.6. *Let $p, q, r \in \mathbb{Q}_n$. Then*

$p \approx p$	*[$\approx$ Reflexive]*
If $p \approx q$, *then* $q \approx p$	*[$\approx$ Symmetric]*
If $p \approx q$, *and* $q \approx r$, *then* $p \approx r$	*[$\approx$ Transitive]*
$p \approx to\mathbb{Q}_n\ (to\mathbb{Q}\ p)$	*[Free rational $\approx$ Canonical rational]*
If $p \approx q$, *then* $(p\ \mathbb{Q}_n^+\ r) \approx (q\ \mathbb{Q}_n^+\ r)$	*[$\approx$ respects addition]*
If $p \approx q$, *then* $(p\ \mathbb{Q}_n^*\ r) \approx (q\ \mathbb{Q}_n^*\ r)$	*[$\approx$ respects multiplication]*

With these proofs, I could now finally implement a proof of the following theorem, which requires lemma 5.6, lemma 5.4 and lemma 5.3.

Theorem 5.7. *Let $p, q \in \mathbb{Q}_n$. Then*

$$to\mathbb{Q}\ (p\ \mathbb{Q}_n^+\ q) = (to\mathbb{Q}\ p) + (to\mathbb{Q}\ q)$$

To finally prove associativity of addition, I needed one last proof that extracts a free rational from a canonical rational. Intuitively, we can just take the canonical rational itself, without its lowest terms condition. This was necessary to be able to apply the above theorem to intermediary rational numbers in a proof.

Lemma 5.8. *Let $q \in \mathbb{Q}$. Then there exists $q' \in \mathbb{Q}_n$ such that $q = to\mathbb{Q}_n\ q'$.*

This was a surprising amount of work required to be able to prove a seemingly simple property of addition of canonical rational numbers. To summarise, it required the extension of HCF to obtain Bezout's lemma, the proof that addition on the free rationals is associative, an equivalence relation on the rationals and properties of this equivalence relation, and a proof that the $to\mathbb{Q}$ function can be applied before or after addition of free rationals for the same result. Fortunately, these foundations resulted in a more streamlined process of proving associativity of multiplication of rationals.

Theorem 5.9. *Let $p, q, r \in \mathbb{Q}$. Then*

- $p + q = q + r$
- $(p + q) + r = p + (q + r)$
- $p * q = q * p$
- $(p * q) * r = p * (q * r)$

Whilst lengthy, it is trivial to implement transitivity of order of rational numbers, and trichotomy of rational numbers, in comparison to the above properties.

Theorem 5.10. *Let $p, q, r \in \mathbb{Q}$.*

- *Suppose that $p < q$, and $q < r$. Then $p < r$*
- *$p < r$, or $p = q$, or $q < p$*

There are more properties of the rationals defined in the section below, however there is scope for more properties of the rationals to be implemented. At the very least, most of the properties from theorem 4.1 have an analogous proof for the rationals. I have not implemented all of these functions in this project. I have implemented the proofs strictly necessary for the goal of implementing the Dedekind Reals in Agda. Whilst the framework is in place to complete the analagous proofs, it will be a lengthy process, and some of the proofs may not be necessary to my post-project goal of proving that the Dedekind Reals form a complete archimedean field. Furthermore, there may be another strategy for implementing these functions that avoids some of the difficulty of working with the $to\mathbb{Q}$ function, and the equivalence $\approx$, which will be discussed in a later section.

6 Dedekind Reals

The real numbers are the biggest surprise to me, in terms of formal mathematical representation. The rational numbers are intuitively formalised, whereas it is not easy to intuitively describe the formal description of real numbers. This is surprising because the real numbers are the most familiar field to a mathematician. Many of the courses in an undergraduate degree use the real numbers as the basis of the course, but the construction of the reals is usually unmentioned, most likely due to the lack of intuitiveness.

In mathematics research, usually the real numbers are assumed and hence, also assumed is the foundational mathematics required to construct the reals, shown in this project. There is nothing inherently wrong with assuming these foundations, but is an interesting note that many mathematicians might have never seen, or thought about the construction of the reals, especially within applied mathematical fields such as physics or engineering.

Formally, the real numbers are a complete archimedean ordered field. There are two common constructions, as described at the start of chapter 11 in [Uni13], namely the Cauchy reals and the Dedekind

reals. Within classical mathematics, we could view the two constructions as isomorphic, but Lubarsky has shown that without the law of excluded middle or countable choice the two constructions do not coincide [LR15]. We choose to construct the constructive Dedekind Reals.

If we want to show the Dedekind reals form an complete archimedean ordered field, we need to formalise these concepts too. The field is a well known concept in Mathematics (for a definition see [Unka]), and is not difficult to translate into Agda. A field can be thought of as an abstraction of mathematical structures which have *nice* properties. With a description of the structure of a field, and the properties it has, we can prove properties that a field has in general, and any mathematical structure which satisfies the field axioms can inherently possess properties we prove for fields. A field is known as an *ordered* field if it is equipped with a relation operation, and satisfies a further two properties than a standard field requires. The difficulty lies in proving that a mathematical construction is indeed a field. I have partially shown the the rational numbers are an ordered field, there are still holes to be filled. See the commented out lines at the bottom of this html file.

An ordered field is archimedean if it satisfies the Archimedean principle. For the reals, this can have the following form, as stated in [Uni13]. I have wrote the type of this proof, commented out in this file.

Theorem 6.1 (Archimedean Principle of $\mathbb{R}$)**.** *Let $x, y \in \mathbb{R}$. Suppose $x < y$. Then there exists $q \in \mathbb{Q}$ such that $x < q < y$.*

The notion of completeness of an ordered field depends on the ordered field itself, but if the field satisfies one version of the completeness axiom, then it satisfies all of them. The notion of completeness of a Dedekind field is stated as in [Uni13], corollary 11.2.16. In the general sense, we are saying that any non-empty subset of $\mathbb{R}$ with an upper bound has a *least* upper bound.

Theorem 6.2. *The Dedekind reals are Dedekind complete: for every real-valued Dedekind cut (L, U) there exists a unique $x \in \mathbb{R}$ such that $L(y) = (y < x)$ and $R(y) = (x < y)$.*

Note that this notion of completeness is different to the notion of Cauchy completeness. Cauchy completeness can be generalised to completeness of a metric space, which is the notion of a metric space without any holes, or more formally a metric space in which every Cauchy approximation has a limit, ([Uni13], Definition 11.5.2).

We now introduce the Dedekind reals, with one slight variation to the HoTTBook [Uni13] definition of a Dedekind Cut, and notational differences. Here, we can understand the notation $L(p)$ to mean that the rational number p satisfies some property L.

Definition 6.2.1. A Dedekind Cut is a pair (L, R) of properties of rational numbers such that the following properties hold:

- Inhabited: $\exists\ p, q \in \mathbb{Q}$ such that $L(p) and R(q)$.
- Rounded: Let $p, q \in \mathbb{Q}$. Then

$$L(p) \iff \exists\ p' \in \mathbb{Q} \text{ such that } (p < p') \wedge L(p')$$

$$R(q) \iff \exists\ q' \in \mathbb{Q} \text{ such that } (q' < q) \wedge R(q').$$

- Disjoint: Let $p, q \in \mathbb{Q}$. Suppose that $L(p)$ and $R(q)$. Then $p < q$
- Located: Let $p, q \in \mathbb{Q}$. Suppose that $p < q$. Then $L(p) \vee R(q)$

We can picture a cut as a point on the real line, where everything below the point is contained in L, and everything above in R, where the point is not contained in either L or R. Inhabitedness says that for any cut we choose, there exists a rational on either side of the cut. Roundedness says that if we choose a rational number in either side of the cut, we can find a rational that is closer to the cut on that same side. Disjointness says that any rational in the left cut is strictly less than any rational in

the right cut. Locatedness says that we can place at least one side of any interval of rational numbers in the left or right cut.

The HoTT Book [Uni13] has a slight variation of the disjointness property that says that a rational number is never located in both sides of a cut. As part of an extension of this project, it could be shown that in the presence of the other conditions, these two variations of disjointness are equivalent.

The Dedekind Reals are the set of all Dedekind Cuts. That is, the set of predicate pairs that that satisfy the properties of a Dedekind Cut. The HoTT Book [Uni13] goes on to further state, without proof, that there is an embedding $\mathbb{Q} \to \mathbb{R}$. For $x \in \mathbb{Q}$, we choose the predicate pair for the cut as

$$L_x(p) = (p < r) \text{ and } R_x(q) = (x < q).$$

I defined the Dedekind Reals and implemented the embedding from $\mathbb{Q} \to \mathbb{R}$, with some difficulty. Disjointedness and locatedness follows quickly from transitivity of order of rationals, and trichotomy of rationals respectively. Inhabitedness required two extra proofs of the rationals.

Remark. Note that in the definition of the Dedekind Reals, instead of using a Sigma type, we use the truncated Sigma type. Recall that we work constructively, so when we use a Sigma type we must give a witness that satisfies the dependent function. The truncated Sigma "forgets" this witness. The only information a truncated Sigma type contains is the inhabitedness of a type, and logically we can view it as existence. Full explanations can be seen in [Uni13, p. 152] and [Esc19a]. Proposition Truncation must be assumed in the same manner as Function Extensionality, and the discussion of FE applies to Propositional Truncation.

Instead of assuming FE and PT for each individual proof, I make the assumption globally within the Dedekind Real module. I could make these assumptions as needed for each function, but it makes sense to make the assumptions globally within a module most functions require them.

Theorem 6.3. *Let $x \in \mathbb{Q}$. Then*

There exists $p \in \mathbb{Q}$ such that $p < x$	*[$\mathbb{Q}$ has no least element]*
There exists $q \in \mathbb{Q}$ such that $x < q$	*[$\mathbb{Q}$ has no maximum element]*

The proofs of these required a result from theorem 5.6, that a free rational is equivalent to its canonical representation. Roundedness demonstrates the general difficulty of proving properties of the rational numbers. If we are given $p, q \in \mathbb{Q}$ and we want to find $x \in \mathbb{Q}$ such that $p < x < q$, intuitively we take x as the midpoint of p and q. Following this intuition, I implemented the following lemma.

Lemma 6.4. *Let $p, q \in \mathbb{Q}$, and suppose that $p < q$. Then there exists $x \in \mathbb{Q}$ such that $p < x < q$.*

This was an extremely lengthy proof. There is no technique used in this proof that is any more difficult than anything previously used in this project, but it exemplifies the tedium that can arise when formally proving mathematical proofs within the setting of constructive type-theory. This proof demonstrates my ability in Agda to many of the concepts I have learned over this project at the same time. This proof contains equational reasoning, properties of natural numbers and integers, the use of dependent and independent pairs and projection pairs, use of an equivalence relation on the rationals and more. It does not demonstrate my ability to write *clean* proofs. Even as the writer of the proof, I find it difficult to follow. There is an art to writing clean proofs, and this is a challenge in both pen and paper proofs and proofs in Agda. Eventually, I would like to improve the readability of my proofs regarding the rational numbers. I would like to believe my proofs up to the rationals are readable. This is subjective, and there is always rooms for improvement.

This embedding $\mathbb{Q} \to \mathbb{R}$ concludes my mathematical formalisation in this project, and completing this proof was the most satisfying moment of the project. I was not able to finish my implementation of the Dedekind Reals within the permitted time.

7 Discussion and Future Work

I have learned Agda, and have improved my ability to write Agda code over the course of this project. Working with a proof assistant means that if I complete a proof that type-checks, then if I have correctly written the type, I can guarantee that the proof is "correct". The proof may not easy to follow. Writing clean code is important for any language, but writing mathematical proofs emphasises this importance. A proof is much less useful if others are unable to follow and interpret the proof. I am satisfied with the readability of my proofs up to the Rationals. The Rationals are challenging, and I would like to re-organise this code to be clearer and easier to follow. It would be useful to have a version of the proof as it might be presented in a mathematical paper alongside the Agda proofs.

There are several immediate results that can be worked on in the short term. We would like to show that the Dedekind Reals are an Archimedean Field. This can be seen by showing that the Dedekind Reals both satisfy the Field Axioms, and the Archimedean principle.

The HoTT book [Uni13] states the following operations in order to satisfy their version of the field axioms: addition, multiplication, inverse, minimum, maximum, the relations: less-than, less-than-or-equal and apartness. They also require the constants 0 and 1. The archimedean principle for the Dedekind Reals can be proved using the definition of the less-than relation. I have defined the Field axioms in Agda, and wrote in the functions, with holes, that prove that the Dedekind Reals form an Archimedean field. The field axioms are defined easily in Agda, but proving that the Dedekind Reals satisfy the field axioms is a lengthy task. I define addition of the Dedekind Reals as in [Uni13].

Definition 7.0.1 (Addition of Dedekind Reals)**.** For $z \in \mathbf{R}$, let L_z, R_z refer to the lower and upper cuts of z respectively.

Let $x, y \in \mathbf{R}$. Then define $x + y$ as:

$$L_{x+y}(p) := \exists (r, s \in \mathbb{R}) \text{ such that } L_x(r) \wedge L_y(s) \wedge p = r + s$$

$$R_{x+y}(q) := \exists (r, s \in \mathbb{R}) \text{ such that } R_x(r) \wedge R_y(s) \wedge q = r + s$$

The difficulty lies in proving that the definition is a cut. Solving this directly, this amounts to showing that $L_{x+y}(p), R_{x+y}(q)$ are inhabited, rounded, disjoint and located, which is not trivial. There is a process to follow; one must construct a proof outside of Agda and have a good understanding of how it works. It is not enough to simply write in the type that you need and hope that the proof follows easily, as one might do for some proofs of natural numbers and integers. With the length of lemma 6.4 in mind, it is clear there will be a lot of work involved in using all of information contained in a Dedekind Cut, along with implementing any required auxiliary proofs involving rational numbers. This process must be followed for all of the operations mentioned above, although the relations and constants are defined trivially. The first task following the conclusion of this project will be to implement all of these operations, and subsequently prove that they are an archidemean field.

The Field Axioms stated in the HoTT Book are not the same as my definition of the Field Axioms. I am aware that my definition of fields is insuffient in their current form. As discussed by Booij in his thesis [Boo20, p.67], we cannot constructively produce a multiplicative inverse of a number using the assumption that the number is not 0. My definition of the Field will therefore have to change for the purpose of working constructively.

Following this, I will show that the Dedekind reals are Dedekind complete, as defined in 6.2. I anticipate following the strategy shown in [Uni13, section 11.2.3]. At this point, with experience working on the operations of the Dedekind Reals, I do not think that proving completeness of the Dedekind Reals will pose much of a challenge, but of course there is always the opportunity for the formalisation to be surprising. My current definition of the Dedekind Reals uses univalent propositions of the rationals that live in the lowest universe, $\mathcal{U}_0$. My proof of completeness will therefore be correct up to this universe. The Reals are the first instance of a structure that lives in a universe higher than the lowest universe.

7.1 Equivalence Relation

If I explored developing more functions of the equivalence relation I introduced in definition 5.2.2, I might be able to eliminate code that re-occurs for functions of the rationals. Cleaning the code and further development of the equivalence will happen in tandem with work on the Dedekind reals. There is also scope for re-implementing the Integers and the Rationals as a set-quotient. This would be more tedious than difficult, and could result in some proofs of the rationals becoming shorter. It may also extend other proofs. I am interested to see if the set-quotient variants, which are closely related to the equivalence relation I defined on the rationals, would reduce the overall tedium of working with the rational numbers. The different implementations are isomorphic, and wouldn't have an effect on the correctness of proofs. This implementation is a side goal, and may happen before or after completing my work on the Dedekind Reals, depending on the difficulty encountered.

7.2 Extension of uniformly continuous functions on a dense set

As mentioned, implementing the operations of the real numbers directly will not be easy. It is worth exploring an alternative method of defining these operations. If a function is uniformly continuously on a dense subset, then it has a continuous extension to the whole set. We have the rational numbers, which are dense and can be embedded in the reals. By implementing the continuous extension theorem, it might be easier to implement operations and properties of the reals by extending the functions already implemented on the rationals. I have written, with holes (at the bottom of this file), the functions that will prove that the rationals form an archimedean ordered field. This strategy will also require the use of metric spaces, I would use O'Connor's PHD thesis [OCo09] as a framework when working with Uniformly Continuous Functions.

7.3 Unit Interval satisfies the Convex Body Axioms

Following completion of the proof that the Dedekind Reals form a complete archimedean field, an extension of the project could show that the unit interval $[0, 1]$ satisfy the convex body axioms, described by Escardó and Simpson in [ES01].

8 Conclusion

The goal of this project was to learn how to implement mathematical proofs within Agda. I thought the best way to accomplish this would be to work towards an implementation of the Dedekind Reals. I began by working on simple mathematical proofs, developing a deeper understanding of Agda, and more generally Type Theory. I encountered difficult challenges, including implementing HCF and Bezouts Lemma, working with more complex structures such as the rationals, and implementing proofs where the pen and paper proofs were not rigorous enough to translate directly.

In the process of formalising the construction of "simple" mathematical structures, I gained a deeper respect of the level of rigour required to write proofs of even more basic properties. I have noticed the difference between the completeness of the requirements of a proof assistant, and the assumptions usually made within the field of mathematics.

I have appreciated the assistance that Agda provides, with interactive proof solving in the form of holes in the code. Partial refinement of solutions, and providing the type of the goal can be extremely useful in compartmentalising proofs into easier to solve sub-proofs. I have noted that sometimes the goal type can lead to a focus on the wrong direction when attempting a proof, but with a pen and paper draft this can be minimised. It is certainly possible to write in the type of a proof, without checking that a proof actually exists.

Working towards the Dedekind Reals, I formalised properties of order and addition of Natural Numbers, multiplication and its properties, division, the extended euclidean algorithm. I have formalised integers and its operations, proving that it is a group with respect to addition. I have partly formalised

the rationals, including addition, multiplication and order, but some more work is required to prove that the rationals are a field.

I have learned about the connections between Type Theory and Mathematics, and noted the use of notions from Univalent Type Theory, for example the use of Function Extensionality, and the use of propositions in univalent mathematical sense. I have discussed the notion of assuming only as much as one needs when writing Mathematical Proofs, albeit breaking this notion for ease of use of FE and Propositional Truncation in the Dedekind Reals module.

With the formalisation of the Dedekind Reals incomplete, I have discussed the path forward to finish the implementation, which will involve a good amount of work, considering the complex structure of the Reals. Due to the vastness of the fields of Mathematics and Type Theory the scope for extending this project is wide, which gives me options to explore in my fourth year project.

References

[Bau10] Andrej Bauer. *Mathematics and Computation*. Mar. 2010. URL: http://math.andrej.com/2010/03/29/proof-of-negation-and-proof-by-contradiction/.

[BD09] Ana Bove and Peter Dybjer. "Dependent Types at Work". In: *Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*. Ed. by Ana Bove et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 57–99. ISBN: 978-3-642-03153-3. DOI: 10.1007/978-3-642-03153-3_2. URL: https://doi.org/10.1007/978-3-642-03153-3_2.

[Boo20] Auke Bart Booij. *Analysis in Univalent Type Theory*. 2020. URL: https://abooij.blogspot.com/p/phd-thesis.html.

[Dyb18] Peter Dybjer. *An Introduction to Programming and Proving in Agda (incomplete draft)*. 2018. URL: http://www.cse.chalmers.se/~peterd/papers/AgdaLectureNotes2018.pdf.

[ES01] Martin Escardo and Alex Simpson. "A Universal Characterization of the Closed Euclidean Interval." In: May 2001, pp. 115–125. DOI: 10.1109/LICS.2001.932488.

[Esc] Martín Hötzel Escardó. URL: https://www.cs.bham.ac.uk/~mhe/TypeTopology/index.html.

[Esc19a] Martín Hötzel Escardó. "Introduction to Univalent Foundations of Mathematics with Agda". In: *CoRR* abs/1911.00580 (2019). arXiv: 1911.00580. URL: http://arxiv.org/abs/1911.00580.

[Esc19b] Martín Hötzel Escardó. *Various new theorems in constructive univalent mathematics written in Agda*. https://github.com/martinescardo/TypeTopology/. Agda development. Feb. 2019.

[Goo19] Simon Goodwin. *1AC: Algebra 1*. 2019.

[Ham82] A. G. (Alan G.) Hamilton. *Numbers, sets and axioms : the apparatus of mathematics / A.G. Hamilton*. eng. Cambridge, 1982.

[Kum09] N. Kumar. "CONSTRUCTION OF NUMBER SYSTEMS". In: 2009.

[Leh17] Eric Lehman. *Mathematics for Computer Science*. London, GBR: Samurai Media Limited, 2017. ISBN: 9789888407064.

[LR15] Robert Lubarsky and Michael Rathjen. *On the Constructive Dedekind Reals*. 2015. arXiv: 1510.00641 [math.LO].

[Nee02] Amnon Neeman. "A counterexample to a 1961 "theorem" in homological algebra". In: *Inventiones mathematicae* 148.2 (May 2002), pp. 397–420. ISSN: 1432-1297. DOI: 10.1007/s002220100197. URL: https://doi.org/10.1007/s002220100197.

[Nor09] Ulf Norell. "Dependently Typed Programming in Agda". In: *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Ed. by Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 230–266. ISBN: 978-3-642-04652-0. DOI: 10.1007/978-3-642-04652-0_5. URL: https://doi.org/10.1007/978-3-642-04652-0_5.

[OCo09] Russel O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. 2009. URL: http://r6.ca/thesis.pdf.

[Rom] Mario Román.
URL: https://mroman42.github.io/cosmoi/posts/dedekindreals.html.

[Teaa] The agda Team. *Safe Agda¶*.
URL: https://agda.readthedocs.io/en/v2.6.0.1/language/safe-agda.html.

[Teab] The Agda Team. *Function Definitions¶*. URL:
https://agda.readthedocs.io/en/v2.6.0.1/language/function-definitions.html.

[Teac] The Agda Team. "Termination Checking". In: (). URL:
https://agda.readthedocs.io/en/v2.6.0.1/language/termination-checking.html.

[Uni13] The Univalent Foundations Program.
Homotopy Type Theory: Univalent Foundations of Mathematics.
Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[Unka] Unknown. *Definition:Field Axioms*.
URL: https://proofwiki.org/w/index.php?title=Definition:Field_Axioms.

[Unkb] Unknown. *Lecture 30: Number bases, Euclidean GCD algorithm, and strong induction*.
URL: http://www.cs.cornell.edu/courses/cs2800/2017sp/lectures/lec30-strongind.html.

[Unkc] Unknown. *Library Coq.Reals.ClassicalDedekindReals*.
URL: https://coq.inria.fr/library/Coq.Reals.ClassicalDedekindReals.html.

[VM] Andrea Vezzosi and Mörtberg. *Cubical Adga*. URL: https://github.com/agda/cubical/blob/master/Cubical/Data/Int/Properties.agda.

A Appendix

My code can be seen on Gitlab: https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2020/axs1575

The Gitlab contains a readme file which briefly describes how the code may be compiled: https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2020/axs1575/-/blob/master/Agda%20Code/README.txt

The best way to view the code is viewing it as HTML, which is fully syntax highlight and hyperlinked: https://adsneap.github.io/CSProject/AndrewIndex.html

My files display my name at the top.