Representing numbers and in Agda

Li Nuo

May 4, 2010

Contents

1 Introduction and Motivation **3**

2 What is Agda? **5**

3 What are numbers in regular mathematics? **8**

4 What are differences of numbers in computer science? **9**

5 How to define numbers in Agda? **10**

6 Related work **17**

6.1 The integers and rational numbers in Coq 17

6.2 The real numbers in Coq 18

6.3 The real numbers in other languages 19

7 The design of this project **19**

7.1 Definition part 20

7.2 Properties part 22

8 Implementation **26**

8.1 Proving theorems in Agda 26

8.2 Problems and Changes 27

9 Evaluation **30**

9.1 How to use the code 30

9.2 Feedback 30

10 Summary and future work **31**

Abstract

Recent development of dependently typed languages like Coq, Agda and Epigram provide programmers as well as mathematicians to prove theorems by writing programs, or more appropirately, constructing proofs. Agda, as one of the latest functional programming languages, is a flexible and convenient proof assisstant equipped with interactive environment for writing and checking proofs. The current version of standard library which is mostly built by Danielsson has included Boolean, algebraic structures, sets, relations etc. However, to prove most of theorems for numbers, it requires more definitions of the numbers beyond natural numbers and more axioms and theorems. The work is motivated by the numerous good features which gives Agda potential to be a good theorem prover. I will talk about these later along with introducing my code. An interesting discovery is that I can understand the natural of numbers more deeply when I use Agda to define numbers.

To solve the problem, I start this project, in which I will define the numbers including integers, rational numbers, real numbers and complex numbers and prove the basic properties of them as the tools for theorem proving. There have been some researches in representing numbers especially exact real numbers in other languages, for example Geuvers and Niqui's construction of real numbers in Coq [11], Claire Jones's definition of real numbers in In LEGO [12]. In this project, I will explore more suitable ways to represent numbers in Agda and try to configure out the construction of real numbers based on Bishop's real number system [3]. Although representing numbers in a programming languages must based on the ideas in mathematics, it still has quite a few differences. After we understand the numbers in programming languages, How to implement the definition of numbers in Agda? How to prove properties? What problems and issues that I addressed? What could be extended futher in the future?

1 Introduction and Motivation

Mathematics is the foundation of computer science. We could find the highly correlation between computer science and mathematics from the word "computer". In fact, as the science and technology of computer are developing faster and faster, we can see that it could contribute to mathematics more than computing.

Old chinese mathematicians using Counting rods to do computing and proving. The invention of more and more symbols for complex definitions increase the complexity of theorem proving exponentially. New inventions of tools such as paper, abacus and computer often strongly affect the development of mathematics. Before computer invented, mathematicians had to prove theorems on papers. The proofs always spread around a pile of papers. It is likely to make mistakes in proofs and it is hard for documentation. In order to eliminate mistakes, verification of proof is essential. However, even spending quite a long time on the verification, no one can ensure the correctness for complicated proofs.

Some proofs had been believed to be correct were found incorrect after years later. In 1879 Alfred Bray Kempe announced that he had proved the Four Colour problem. Until 11 years later, Percy John Heawood published a paper proving Kempe was wrong [22].

The old ways to conduct proving is very inefficient. Computer changed all the aspects of humans' lives. Computer scientists has created a lot of implementation of the theorem proving by computer programs. With the dependently typed language, such as Coq, Agda and Epigram, mathematicians and programmers could formally prove theorems by writing proofs as programs and the work of verification can be left to computers [2]. Computer could ensure the correctness of proofs. If we believe in the languages, we can believe in the correctness of proofs. The proof assistants has revealed their power in recent years. In 2004, Georges Gonthier and Benjamin Werner have completed their proof of the four color theorem using Coq [21]. Moreover computer could do some automated work which seems impossible to handle by hand.

Agda is one of the proof assistants and it is one of the latest in a series of dependently typed programming languages [5]. Unlike tactic-based proof assistant like Coq, it provides a more flexible way of constructing proofs [2]. It has great potential in the field of theorem proving. However as Agda is at its the early stage, it requires contribution to the standard library of Agda. Some of the basic mathematical definitions has been included, such as sets, logic symbols, relations, algebraic structure. It is enough to define more concepts.

For numbers, there are only natural numbers and part of integers defined in the standard library. However to prove most of the mathematical theorems, we need more definitions for other kinds of numbers, not only the integers but also rational numbers, real numbers and complex numbers. Their basic axioms like commutativity and associativity are essential as well.

The project aims at representing the numbers in Agda. The main objective is to define those numbers in most proper ways and proving some basic properties of them that are essential for theorem proving. There are various ways to define each kind of numbers, so to compare, evaluate and make a decision of them are also objectives.

The inner motivations to do this project are that I am interested in how mathematics can be translated into computer science, how to use computer science to help us solve mathematical problems and how to use mathematics to solve computer science problems. The interest in functional programming is also plays a quite important role. I had learnt to use Coq which is also a proof assistant like Agda to prove theorems using software tools. I found it interesting to formalise mathematics concepts and doing more than computations such as proving or facilitating calculations.

After undertaking the project I also found it is beneficial for other people as some other students from Chalmers told me they could improve their code by using the library code of numbers. To develop a part of a library is a new challenge in which I could gain more experience and learn more skills. At the same time I could also develop the insight of representing numbers and proving theorems in Agda from doing this project. To do a comparably big research project will also benefit my futher study and career lifes.

In this report, I will discuss from easy and familiar aspects to difficult and new aspects, from abstract concepts to concrete implementations and from the begin of the project to the future potential of the project. The codes of Agda in this report are pruned. I only leave the necessary parts of codes which is more readable but may not be executable.

2 What is Agda?

Agda is a dependently typed functional programming language. It is designed based on Per Martin-Löf Type Theory [17]. We can find three key elements in the definition of Agda, the "functional programming language", "dependently typed" and "Per Martin-Löf Type Theory".

- *Functional programming language.* As the name indicates that, functional programming languages emphasizes the application of functions rather than changing data in the imperative style like C++ and Java. The base of functional programming is lambda calculus. The key motivation to develop functional programmming language is to eliminating the side effects which means we can ensure the result will be the same no matter how many times we input the same data. There are several generations of functional programming languages, for example Lisp, Erlang, Haskell etc. Most of the applications of them are currently in the academic fields, however as the functional programming developed, more applications will be explored.

- *Dependent type.* Dependent types are types that depends on values of other types [4]. It is one of the most important features that makes Agda a proof assistant. In Haskell and other Hindley-Milner style languages, types and values are clearly distinct [15], In Agda, we can define types depending on values which means the border between types and values is vague. To illustrate what this means, the most common example is **Vector A n**.

  Definition 2.1.
  ```
  data Vec (A : Set) : ℕ → Set where
    [] : Vec A zero
    _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)
  ```

  It is a data type which represents a vector containing elements of type **A** and depends on a natural number **n** which is the length of the list. With the type checker of Agda, we can set more constraints in the type so that type-unmatched problems will always be detected by complier. Therefore we could define the function more precisely as there more partitions of types. For instance, to use the dependently typed vector, it could avoid defining a function which will cause exceptions like out of bounds in Java.

- *Per Martin-Löf Type Theory.* It has different names like Intuitionistic type theory or Constructive type theory and is developed by Per Martin-Löf in 1980s. It associated functional programs with proofs of mathematical propositions written as dependent types. That means we can now represent propositions we want to prove as types in Agda by dependent types and Curry-Howard isomorphism [5]. Then we only need to construct a program of the corresponding type to prove that propostion. For example:

  Proposition 2.1. n+0+0≡n : ∀ {n} → n + 0 + 0 ≡ n
  ```
  n+0+0≡n {zero} = refl
  n+0+0≡n {suc n} = suc n+0+0≡n
  ```

  As Nordström et al. [14] pointed out that we could express both specifications and programs at the same time when using the type theory to construct proofs using programs. The general approach to do theorem proving in Agda is as follows: First we give the name of the proposition and encode it as the type. Then we can gradually refine the goal to formalise a type-correct program namely the proof. There are no tactics like in Coq. However it is more flexible to construct a proof. The process of building proofs is very similar to the process of constructing proofs in regular mathematics. The logic behind it is that if we could construct an instance of the type (proposition), we prove it. It is actually the Curry-Howard isomorphism.

Agda is an extention of this type theory [13] with some nice features which could benefit the theorem proving,

- *Pattern matching.* The mechanism for dependently typed pattern matching is very powerful [1]. We could prove propositions case by case. In fact it is similar to the approach to prove propositions case by case in regular mathematics. We can also use view to pattern match a condition specially in Agda. For example,

  ```
  half : Nat → Nat
  half n with parity n
  half ∘ (k * 2) | even k = k
  half ∘ (1 + k * 2) | odd k = k
  ```

  Here the "parity" after with is a view function that allows us to pattern match on the result of it.

- *Recursive definition.*The availability of recursive definition enables programmers to prove propositions in the same manner of mathematical induction. Generally the natural numbers are defined inductively in fucntional programming languages. Then the program of natural numbers can be written using recursive style. There are a lot of types defined using recursive styles in Agda.

- *Construction of functions.* The construction of functions makes the proving more flexible. We could prove lemmas as we do in maths and reuse them as functions.

- *Lazy evaulation.* Lazy evaluation could eliminate unecessary operation because it is lazy to delay a computation until we need its result. It is often used to handle infinite data structures. [23]

As Agda is primarily used to undertake theorem proofs tasks, the designer enhanced it to be more professional proof assistant. There are several beneficial features facilitating theorem proving,

- *Type Checker.* Type checker is an essential part of Agda. It is the type checker that detect unmatched-type problem which means the proof is incorrect. It also shows the goals and the environment information related to a proof. Moreover a definition of function must cover all possible cases and must terminate as Agda are not allowed to crash [15]. The coverage checker makes sure that the patterns cover all possible cases [5]. And the termination checker will warn possiblily non-terminated error. The missing cases error will be reported by type checker. The suspected non-terminated definition can not be used by other ones. The type checker then ensures that the proof is complete and not been proved by itself. Also we are forced to write the type signature due to the presence of type checker.

- *Emacs interface.* It has a Emacs-based interface for interactively writing and verifying proofs. It allows programmers leaving part of the proof unfinished so that the type-checker will provide useful information of what is missing [5]. Therefore programmers could gradually refine their incomplete proofs of correct types.

- *Unicode support.* It supports Unicode identifiers and keywords like: ∀, ∃ etc. It also supports mixfix operators like: + , − etc. The benefits are obvious. Firstly we could define symbols which look the same and behave the same in mathematics. These are the expressions of commutativity for natural numbers, the first line is mathematical proposition and the second line is code in Agda:

  ```
  ∀ a b, a + b  =  b + a
  ∀ a b → a + b  =  b + a
  ```

  Secondly we could use symbols to replace some common-used properties to simply the proofs a lot. The following code was simplied using several symbols,

  ```
  ⟨_⟩  =  sym
  _>≡<_  =  trans
  _⋆*_  : ∀ {b c} → b ≡ c → (a  : ℕ) → b * a ≡ c * a
  ```

```
*-ex : ∀ a b c → a * (b * c) ≡ b * (a * c)
*-ex a b c = ⟨ *-assoc a b c ⟩ >≡< *-comm a b ⋆* c >≡< *-assoc b a c
```

Finally, we could use some other languages characters to define functions such as Chinese characters.

- *Code navigation.* We can simply navigate to the definition of functions from our current code. It is a wonderful features for programmers as it alleviate a great deal of work to look up the library.
- *Implicit arguments.* We could omit the arguments which could be inferred by the type-checker. In this way, we do not need to present obvious targets of some properties. For example,

  ```
  n+0≡n : ∀ {n} → n + 0 ≡ n
  z*0~0 (x+, x-) = n+0≡n
  ```

  The implicit argument in curly bracket is unnecessay to give explicitly when applying this property.
- *Module system.* The mechanism of parametrised modules makes it possible to define generic operations and prove a whole set of generic properties.
- *Coinduction.* We could define coinductive types like streams in Agda which are typically infinite data structures. It is a proof technique that could prove the equality satisfied all possible implementation of the specification defined in the codata. It is often used in conjunction with lazy evaluation. [19]

With these helpful features, Agda has the potentional be a more powerful proof assisstant. Therefore, in order to provide the availability of all the numbers, this project should be beneficial. With the numbers defined and basic properties proven, mathematicians could prove some famous theorems like Fermat's little theorem then.

3 What are numbers in regular mathematics?

To represent numbers, we should first make clear of the definitions of various kinds of numbers in regular mathematics. Numbers were invented to represent some quantity in the world,

- ℕ - *Natural numbers.* Natural numbers are the basics of all the numbers. It was invented when people used their finger to count things like days and foods. It denotes the result of incrementing, one finger, add one more finger and so on. Therefore, it is natural to define natural numbers as zero namely no finger, and 'suc' namely add one finger so that we could represent two using suc (suc zero).

- $\mathbb{Z}$ - *Integers.* To count more easily people invented operations like addition and subtraction. However people found it was not enough to represent all the results of subtraction. Then the negatives of natural numbers were invented and the numbers were extended to integers. Therefore despite the normal definition of integers, a pair of natural numbers can also represent the result of subtraction between them.
- $\mathbb{Q}$ - *Rational numbers.* Similarly, rational numbers could represent all the results of division between integers and here fractional numbers are included.
- $\mathbb{R}$ - *Real numbers.* Real numbers represent values of something real and the name is invented correspond to imaginary numbers. The extension here is the irrational numbers which cannot be represented as fractional numbers such as $\sqrt{2}, \pi$ etc.
- $\mathbb{C}$ - *Complex numbers.* The invention of complex numbers enables us to solve all the polynomial equations. The imaginary numbers represent something unreal and their squares are negative real numbers. A complex number has a real part and an imaginary part. Therefore it can be simply represented by a pair of real numbers, one for the real part the the other for the imaginary part.

When doing calculation in regular life, we seldomly care about what kind of number that is. After being familiar with using the axioms like commutativity and associativity, we also use them for all kinds of numbers unconsciously. However as computer are more stupid or critic, the things are different here.

4 What are differences of numbers in computer science?

1. In regular mathematics, each lower level numbers is a subset of the higher level ones. For example 3 is a natural number, but it is also a integer, a rational number, a real number and a complex number. In fact these kinds of types are a categorized subset of all of the numbers which have the same features.

   However in computer science, or we should say in most of the programming languages, different kinds of numbers are defined as different types. It means that 3 of type natural numbers is recognised as a different value to $+3$ of type integers, although they refer to the same number in mathematics. Therefore we should define operators and prove properties for each kind of numbers. If you wonder why not just define all of the number as a single type, I think the answer is that there is dependence between lower level numbers and higher level ones, for example, it is more convenient to define integers based on natural numbers as it is an extention of natural numbers. Especially for real numbers, to precisely construct real numbers we have to use natural numbers and rational numbers and we will talk about it later.

2. In regular mathematics, we postulate not only the existence of numbers but also some basic theorems as axioms like commutativity, associativity and distributivity. We can freely use them in computation for all kinds of numbers. We believe they are true from when we were young but we do not need to prove them to be true.

   However in programming languages, although these theorems looks trivial, as we construct the numbers from scratch, we do not have any proofs of these axioms. These axioms are actually provable. To enable users to prove more advanced properties, we have to figure out the proof of them based on the only postulate 'refl' which means we only need to admit it is true that if two things are the same then they are equal.

3. In regular mathematics, we could express numbers in different forms. For example, $\frac{1}{2}$ is the same as 0.5 and both representations of rational numbers can be used simultaneously. However, in Agda, if we choose to represent rational numbers in fractional form, we could not mix it with the decimal representation except when we use polymorphism.

   Generally, the rules for numbers are more rigorous in computer science. What is postulate in Agda is limited. For equality we only postulate when two numbers are the same they are equal,

   Definition 4.1.
   ```
   data _≡_ {a} {A : Set a} (x : A) : A → Set where
     refl : x ≡ x
   ```

   For unequality,

   Definition 4.2.
   ```
   data _≤_ : Rel ℕ zero where
     z≤n : ∀ {n} → zero ≤ n
     s≤s : ∀ {m n} (m≤n : m ≤ n) → suc m ≤ suc n
   ```

   all the other axioms inregular mathematics requires to be defined or to be proved.

5 How to define numbers in Agda?

We have understood what are numbers in mathematics. Then we will discuss how to represent them in Agda. There are several choices of definitions, some are more intuitive to people and some are more meaningful mathematically.

- ℕ - *Natural numbers.* In the base ten system, we can represent natural numbers by using digits from 0 to 9. However it is hard to write this definition in Agda as there are infinite digits. The inductive definition solves the problem efficiently. In Peano Arithmetic, the following inductive definition is widely used to define natural numbers and operations in programing languages like Haskell, Coq and Agda,

Definition 5.1. **data** ℕ : Set **where**

```
zero : ℕ
suc  : (n : ℕ) → ℕ
```

This can be seen as a natural definition because it simulates the process of incrementing numbers.

- ℤ - *Integers.* As we mentioned before, integers add negative numbers to natural numbers. To represent it normally, we only need to add a sign in front of natural numbers to differentiate positive and negative numbers,

  Definition 5.2. **data** ℤ : Set **where**

  ```
  +_   : (n : ℕ) → ℤ
  zero : ℤ
  -_   : (n : ℕ) → ℤ
  ```

  But in this definition the definition is ambiguous. If "$+0$", "-0" and "zero" all represent 0 then, the uniqueness of zero is lost and the basic definitional equality does not work for it. The definitional equality holds for two definitionally same elements. We can simply prove this equality by using "refl". If we need "$+0$" to represent "$+1$", it is better to define it like follows,

  Definition 5.3. **data** ℤ : Set **where**

  ```
  +suc_ : (n : ℕ) → ℤ
  zero  : ℤ
  -suc_ : (n : ℕ) → ℤ
  ```

  Although definition looks symmetric and non-misleading, it has three patterns which increase the number of cases to pattern match when doing definition and theorem proving. For instance, when we prove the associativity for $+$,

  Theorem 5.1 (Associativity). $\forall xyz, (x + y) + z = x + (y + z)$

  In the worst case, we have to pattern match on each of the three numbers so that there will be 27 cases at most. To reduce cases, we could combine zero and positive integers,

Definition 5.4. **data** $\mathbb{Z}$: Set **where**
```
  +_ : (n : ℕ) → ℤ
  -suc_ : (n : ℕ) → ℤ
```

This definition is my choice for normal integer in this project as there is less cases. The asymmetry does not make too much trouble.

All the definition above are intuitive definitions and I call the last one normal definition in this project. Alternatively, we could also use a pair of natural numbers to represent an intger as we discussed before,

Definition 5.5. $\mathbb{Z}_0 = \mathbb{N} \times \mathbb{N}$

it represents the result of subtraction between the two numbers, and the single pattern shorten the proof. However it is an incomplete definition unless we combine a propositional equality with it. Propositional equality is a relation defined specially according to mathematics. Here, we could easily found that $(3, 2)$ and $(2, 1)$ both represent $+1$ but will not be treated equal by definition because they are not the same. To be isomorphic to the set of the integers we have to define the equivalence relation of it so that they could form a setoid.

Definition 5.6. _∼_ : Rel $\mathbb{Z}_0$ zero
```
(x+, x-) ∼ (y+, y-) = (x+ ℕ+ y-) ≡ (y+ ℕ+ x-)
```

We should firstly prove the relation is equvalence.

a) reflexivity: ∀ a → a ∼ a

```
refl : Reflexive _∼_
```

b) symmetry: ∀ a b → a ∼ b → b ∼ a

```
sym : Symmetric _∼_
```

c) transitivity: ∀ a b c → a ∼ b / b ∼ c → a ∼ c

```
trans : Transitive _∼_
```

Now with these three properties we could prove it is equivalence relation,

```
_∼_isEquivalence : IsEquivalence _∼_
_∼_isEquivalence = record
  {refl = refl
  ;sym = sym
```

```
    ; trans = trans
    }
```

Then we can prove that $(\mathbb{Z}_0, \sim)$ is a setoid

```
isSetoid : Setoid _ _
isSetoid = record
  { Carrier = ℤ₀
  ; _≈_   = _∼_
  ; isEquivalence = _∼_isEquivalence
  }
```

I call this kind of integers as setoid integers in this project. The drawback of this representation is that it looks not intuitive to people so it is not a good choice for user. But the benefit for proving is precious so I explore the ways to define both normal definition and setoid one in this project and combine the advantage of both. The mechanism between the two definitions is to firstly define and prove properties for setoid integers and then define the operations of the normal definition on the setoid on using normalisation functions and denormalisation functions,

Normalisation

normalise the setoid integer to normal form

```
[_]           : ℤ₀ → ℤ
[m, 0]        = + m
[0, suc n]    = -suc n
[suc m, suc n] = [m, n]
```

denormalise the normal integer to one representative setoid integer

```
⌜_⌝        : ℤ → ℤ₀
⌜ + n ⌝    = n, 0
⌜ -suc n ⌝ = 0, suc n
```

Then we could prove the isomorphism of the setoid and the normal definition,

a) stability: nf is left inverse of dn

Theorem 5.2. *stable* : $\forall \{n\} \to [\ulcorner n \urcorner] \equiv n$

b) completeness: if it is true, then we can give the proof

Theorem 5.3. *compl* : $\forall \{n\} \to \ulcorner [n] \urcorner \sim n$

c) soundness: if we proved it, then it is true

Theorem 5.4. *sound* : $\forall \{x\, y\} \rightarrow x \sim y \rightarrow [x] \equiv [y]$

Using the setoid and the three theorems above, we could form the quotient type of integer.

```
ℤ-QuSig : QuSig isSetoid
ℤ-QuSig = record
  {Q      = ℤ
  ; [_]   = [_]
  ; sound = sound
  }
ℤ-Nf : Nf ℤ-QuSig
ℤ-Nf = record
  {emb    = ⌜_⌝
  ; compl = λ z → compl
  ; stable = λ z → stable
  }
```

The general quotient type theory was developed by Thomas Anberrée and Thorsten Altenkirch. It is a mechanism designed to simplify the theorem proof of the normal form by lifting the theorems of setoid form. However there is still some problems in applying the lift functions.

- $\mathbb{Q}$ - *Rational numbers.* Rational numbers as quotients of integers will be naturally defined as a pair of integers. The decimal representation which is more common is practice requires infinite digits which is inefficient and imprecise in computer science which use discrete mathematics in general.

  To construct the rational numbers in the fractional form, there are several choices:

  1. a pair of $\mathbb{Z}$

     Definition 5.7. **data** ℚ₀ : Set **where**
     ```
       _/_ : (n : ℤ) → (d : ℤ) → ℚ₀
     ```

     The benifit is the types of numerator and denominator are consistent. But the non-zero restriction of the denominator is hard to reflect in this definition.

  2. a pair of $\mathbb{N}$ with symbol

     Definition 5.8. **data** ℚ₀* : Set **where**
     ```
       _/suc_ : (n : ℕ) → (d : ℕ) → ℚ₀*
     data ℚ₀ : Set where
       +_ : ℚ₀* → ℚ₀
       -_ : ℚ₀* → ℚ₀
     ```

Compared to integers, to deal with lower natural numbers is simpler. However there are two cases which makes the definition of operatio ns and the proof much longer. Moreover, it makes little difference with the following definition,

3. $\mathbb{Z} \times \mathbb{N}$

Definition 5.9. **data** $\mathbb{Q}_0$: Set **where**
  /suc : (n : $\mathbb{Z}$) → (d : $\mathbb{N}$) → $\mathbb{Q}_0$

Because O is easier to excluded for $\mathbb{N}$, we choose to combine the sign with the numerator. We only need to show that we use n to denote successor of n in the denominator. This is what I choose in my project. Also, it is a setoid definition so that we should define equivalence relation for it,

Definition 5.10. _∼_ : Rel $\mathbb{Q}_0$
n1 /suc d1 ∼ n2 /suc d2 = n1 $\mathbb{Z}$*$\mathbb{N}$ suc d2 ≡ n2 $\mathbb{Z}$*$\mathbb{N}$ suc d1

There is also normal definition of rational numbers. When we simplify the numerator and denominator to become coprime, then we can infer they are the same from equality. As the representative of for each rational numberis unique, we get definitional equality,

Definition 5.11. **record** $\mathbb{Q}$: Set **where**
  constructor _/suc_, [_]
  field
    numerator : $\mathbb{Z}$
    denominator : $\mathbb{N}$
    isCoprime : Coprime ($\mathbb{Z}$to$\mathbb{N}$ n) (suc d)

This normal definition has been available in the 0.3 version standard library. However there is limited amount of functions defined.

- $\mathbb{R}$ - *Real numbers.* To define real numbers is much more complicated work. In computer science we mostly deals with discrete mathematics, but real numbers is continuous. Therefore in most commercial used programming languages, programmer use types like double, float which have finite decimals to represent a real number for practical use as they are more efficient. However it is not precise to omit the the infinit tail of an irration number. If we want to prove properties for real numbers, we need to define 'real' real numbers.

  There are various ways to construct real numbers : Cauchy sequences of rational numbers, Dedekind cuts, decimal expansion and etc. [20] In this project we follow the development of Bishop [3] to construct real numbers. It uses a Cauchy sequence, namely an infinite sequence of rational numbers whose elements become Arbitrarily close and converge to certain real number.

For example: To represent $\sqrt{2}$ we could use the sequence,

$$1, 1.4, 1.41, 1.414, 1.4142, 1.41421...$$

To generate the sequence we could use general function and a proof of the elements in the sequence generated from the general function complete Cauchy sequence, namely the sequence converges,

Definition 5.12.
```
record ℝ0 : Set where
  constructor _,[_]
  field
    f : ℕ → ℚ
    p : ∀ ϵ → (ϵ > 0) → ∃ λ N →
      (m n : ℕ) → (m > N) → (n > N) → | (f m) - (f n) |< ϵ
```

or use a stream of rational numbers which is a coninductive type in Agda,

Definition 5.13.
```
record ℝ0 : Set where
  constructor _,[_]
  field
    s : Stream ℚ
    p : ∀ ϵ → (ϵ > 0) → ∃ λ N →
      (m n : ℕ) → (m > N) → (n > N) →
      | (lookup m s) - (lookup n s) |< ϵ
```

There are infinite number of Cauchy sequences representing the same real numbers, so this is also a setoid definition. Two Cauchy sequences (X and Y) are called equivalent if and only if for every positive rational number ϵ, there exists an integer N such that $|Xn - Yn| < \epsilon$ for all $n > N$. Hence the equivalence relation could be defined as,

Definition 5.14.
```
_∼_ : Rel ℝ0
X,[_] ∼ Y,[_] =
  ∀ ϵ → is+ ϵ → ∃ λ N →
  (n : ℕ) → (n > N) → | (X n) - (Y n) | < ϵ
```

As we could not gives each of the infinite elements manually, Then the Taylor Series [25] should be used to generate a general functions of the sequence for different real numbers. It is better to define the real numbers as a general function rather than an infinite stream which may not contain the information of the general function. Therefore I use the first one in this project.

Actually, unlike integers and rational numbers, real numbers have no normal forms. Georg Cantor has famously proved that the set of reals is uncountable infinite in his first uncountability proof[7]. It means that the cardinality of reals, $\mathfrak{c}$, is strictly greater than the cardinality of the set of natural numbers, $\aleph_0$, which equals to the one of integers and the one of rational numbers, $\aleph_0 < \mathfrak{c}$

Another famous construction method is Dedekind cuts. It can be defined as a partition of an ordered field (A, B), such that A is closed downwards, B is closed upwards and both of them are non-empty. In fact as A determines B we can use only A to represent a real number.

Intuitively, a real number is represented by all rational numbers which are less than it. It looks simpler than Cauchy sequence, but it is difficult to formlise a way to generate the dedekind cut in Agda. For example, if we want to represent $\sqrt{2}$,

$$A = \{x \in \mathbb{Q} : x < 0 \bigvee x \times x < 2\}$$

There is no general way to construct a dedekind cut.

- $\mathbb{C}$ - *Complex numbers.* After defining real numbers, the complex numbers will be easy to define. It consists of a real part and an imaginary part whose coefficient is also a real number. Similarly we could represent it as a pair of real numbers but it is not a setoid definition as it is unique representation.

  Definition 5.15. $\mathbb{C} = \mathbb{R} \times \mathbb{R}$

6 Related work

6.1 The integers and rational numbers in Coq

Coq which is also a proof assistant has developed a more completed library. There is a binary integer definition written by Pierre Crégut. The integer defines on a binary positive natural number as follows,

Definition 6.1. Inductive positive : Set :=
| xI : positive → positive
| xO : positive → positive
| xH : positive.

It defines positive natural numbers in binary notation. XH represents 1, XI p represents p 1, and XO represents p 0. For example, XI (XO XH) represents 101 binary number, namely 5. Then the integers could be defined as

Definition 6.2. Inductive Z : Set :=
| Z0 : Z
| Zpos : positive $\rightarrow$ Z
| Zneg : positive $\rightarrow$ Z.

This definition is symmetric. ZO is 0, Zpos maps positives just inject positive natural numbers to positive integers, and Zneg constructs negative integers from positives. The operations are defined based on the operation of binary numbers. It is quite different to what I do in this project although the definition looks similar. Alternatively, there are also 31-bits integers designed to achieve hardware-efficient computations encoded by A. Spiwack.

In the definition of rational number, the advantage of define positives is enhanced as we only need a integer and positive to represent rational numbers in fractional forms. The other parts are quite similar to what I have done such as the definition of equivalence. Because the rational number is not in reduced form, it is also a setoid definition. The designer also prove the necessary properties for rational numbers. What is distinct is the field tactics are defined in the library. In addition there is a canonical representation of rational numbers which consists of the setoid definition and a proof that it is the same with its simplification. It is just an alternative way to write the normal form of rational numbers.

6.2 The real numbers in Coq

The construction of real numbers in Coq standard library are axiomatized (refer to the Coq standard library). It defines R as a parameter and the relations and operations based on the classical axioms of real numbers. Namely all the basic properties for proving are axioms like the regular mathematics.

Geuvers and Niqui [11] implements real numbers in a different way in Coq. Their work is backward developed. First they define real numbers and properties as axioms. Then they define rational numbers and real numbers based on the Cauchy sequence of $\mathbb{Q}$. What they need to prove is this construction satisfy the set of axioms they defined. They also prove their axioms are equivalent to the ones in [6].

Another approach to define reals in Coq is introduced by A. Ciaffaglione and P. Di Gianantonio [9]. They construct the reals using infinite streams by coinductive approach which is also available in Agda. They put emphasis on the efficiency of implementation.

Russell O'Connor [16] also build real number constructively based on implementation of compelte metric spaces. The basic functions and proofs are also included. Finally he also implement an automatical proving tatic for strict inequalities for real numbers.

6.3 The real numbers in other languages

There are also some brilliant constructions in other type theoretic proof assistants such as LEGO and NuPrl.

- In LEGO, Claire Jones [12] builds a real number as a collection of arbitrary small nested intervals with rational endpoints. It mainly focus two main areas, one is the construction of real numbers, the other is the compeletion of metric space.
- In NuPrl, J Chirimar, D Howe [8] firstly defines rational number and then defines the reals using regular Cauchy sequences also following Bishop's construction [3]. Finally they prove the Cauchy completeness of reals namely all Cauchy sequence converge.
- In HOL, John Harrison adopted another construction of real numbers, the dedekind cut. It has potential to be implented and tested in Agda to use dedekind cut method.
- Pietro Di Gianantonio [10] proposed an implementation of the real numbers by using the feature of functional programming languages, lazy evaluation which is also available in Agda. It discuss the computability of real numbers represented by infinite digits and related it to the domain theory.

These definitions explore different ways to represent real numbers more deeply and widely. The construction of integers and rational numbers are mostly similar except the binary and inaccurate digits representations. For real numbers, the Cauchy sequence with metric space is choosen primarily. There is various implementation of real numbers in most of the theorem prover.

7 The design of this project

The project consists of several parts catogorised by different kinds of numbers, integers, rational numbers, real numbers and complex numbers. For each part, there will be one or more alternative definitions. For each definition there are several files containing the proofs of properties. The definition which is most convenient to use will be put in the primary file for each kind of numbers. The name convention and layout of code has been adapted to the standard library so that it is easy for add these codes into the standard library. User could import and make use of these numbers in a similar way as natural numbers. However, due to the existence of definitions of integers and rational numbers implemented by Nils Anders Danielsson, I have to change the name slightly to avoid ambiguity. I list the general definitions and properties I have done in this project below.

7.1 Definition part

1. *Definition of numbers.* The definition should be placed at very first. For setoid definitions, a equivalence relation should be defined as well for examples,

$$\begin{aligned} \mathbb{Z}_0 &= \mathbb{N} \times \mathbb{N} \\ \_\sim\_ &: \mathsf{Rel}\ \mathbb{Z}_0 \end{aligned}$$

2. *Ordering relations.* With the equivalence relation defined, we could prove the equality. Likewise, to enable Agda to prove unequality, we should define the unequality relations for numbers, or as the convention the ordering relations. For example,

$$\begin{aligned} \_\leqslant\_ &: \mathsf{Rel}\ \mathbb{Z}_0 \\ \_<\_ &: \mathsf{Rel}\ \mathbb{Z}_0 \\ \_\geqslant\_ &: \mathsf{Rel}\ \mathbb{Z}_0 \\ \_>\_ &: \mathsf{Rel}\ \mathbb{Z}_0 \end{aligned}$$

3. *Sign functions.* There are negative numbers since integers. Therefore the sign function is necessary and useful in many proofs. For example,

   a) decomposition

$$\mathsf{sign} : \mathbb{Z} \rightarrow \mathsf{Sign}$$

   b) composition

$$\_\triangleleft\_ : \mathsf{Sign} \rightarrow \mathbb{N} \rightarrow \mathbb{Z}$$

   With the View "SignAbs",

```
data SignAbs : ℤ → Set where
  _◀_ : (s : Sign) (n : ℕ) → SignAbs (s ◃ n)
signAbs : ∀ m → SignAbs m
signAbs m = subst SignAbs (sign◃ m) (sign m ◀ (p m))
```

   we could use pattern match to decompose an integer to be a sign and a natural number.

4. *Abbreviation of conditions.* We often need these conditions in theorem proving. There are two advantages, first it makes the conditions consistent, second we do not need to unfold the definition since we can give the whole number to conditions. For example,

$$\begin{aligned} \mathsf{is0} &: \mathbb{Z}_0 \rightarrow \mathsf{Set} \\ \mathsf{is0}\ (\mathsf{a}, \mathsf{b}) &= \mathsf{a} \equiv \mathsf{b} \end{aligned}$$

$$
\begin{aligned}
&\neg 0\ :\ \mathbb{Z}_0 \to \mathsf{Set}\\
&\neg 0\ (a, b)\ =\ a \not\equiv b\\
&\text{is+}\ :\ \mathbb{Z}_0 \to \mathsf{Set}\\
&\text{is+}\ (a, b)\ =\ b\ \mathbb{N}{<}\ a\\
&\text{is-}\ :\ \mathbb{Z}_0 \to \mathsf{Set}\\
&\text{is-}\ (a, b)\ =\ a\ \mathbb{N}{<}\ b
\end{aligned}
$$

5. *Conversion functions.* We need to convert the current number to other type of number so that we can deal with the computation mixed with different types of numbers. For example,

   a) p - projection: absolute value but type is $\mathbb{N}$

   $\mathsf{projection}\ :\ \mathbb{Z}_0 \to \mathbb{N}$

   b) i - injection: a representative $\mathbb{Z}_0$ for $\mathbb{N}$ ($\mathbb{N} \subset \mathbb{Z}$)

   $\mathsf{injection}\ :\ \mathbb{N} \to \mathbb{Z}_0$

6. *Arithmetic.* Then we should define the operations. Most definitions in this project do not have mixed types. There are two main kinds of elementary operations, unary and binary operations, For example,

   a) Successor

   $\mathsf{suc}\ :\ \mathsf{Op}_1\ \mathbb{Z}_0$

   b) Predecessor

   $\mathsf{pred}\ :\ \mathsf{Op}_1\ \mathbb{Z}_0$

   c) Negation : $-(3-2) = 2-3$

   $\text{-\_}\ :\ \mathsf{Op}_1\ \mathbb{Z}_0$

   d) Absolute value : $\mathbb{Z}_0$

   $|\_|\ :\ \mathsf{Op}_1\ \mathbb{Z}_0$

   e) Addition : $(a-b)+(c-d) = (a+c)-(b+d)$

   $\_{+}\_\ :\ \mathsf{Op}_2\ \mathbb{Z}_0$

   f) Minus : (a - b) - (c - d) = (a + d) - (b + c)

   $\_\text{-}\_\ :\ \mathsf{Op}_2\ \mathbb{Z}_0$

g) Multiplication: (x - y) * (m - n) = (x * m + y * n) - (x * n + y * m))

```
_*_ : Op₂ ℤ₀
```

There are also operations with mixed types,

```
_ℕ*ℤ₀_ : ℕ → ℤ₀ → ℤ₀
n ℕ*ℤ₀ (z+, z-) = n ℕ* z+, n ℕ* z-
```

For rational numbers, there are more operations to be defined,

Inverse function: It does not define on zero.

```
inverse : (q : ℚ₀) → {p : ¬0 q} → ℚ₀
```

Division: We can define the division for $\mathbb{Q}_0$ because the results of division of ℕ or ℤ and $\mathbb{Q}_0$ are always $\mathbb{Q}_0$

```
_÷_ : (a b : ℚ₀) → {p : ¬0 b} → ℚ₀
```

For the power functions, as we have to use fractional forms, and even real numbers for the root functions, which is the reverse of power functions, I defined some of them in a separate files called "Power" which is defined after the rational numbers.

7.2 Properties part

Due to the inefficiency problem arose from too much code in one file, I separate the proofs of properties into several parts. The general catogorising approach of these properties are separate into three parts, basic properties which includes the basic properties which are essential for the second part which includes all the properties to form a commutative ring for integers or form a field for rational numbers. Finally, the advanced part contains some properties which can be proved after we proved the second part and usually include some lemmas for theorems of the higher level numbers.

Basic properties. In the basic the common properties are:

1. *some relations and conditions are decidable.* For example,

```
_≟_ : Decidable _≡_
0 ? : ∀ z → Dec (is0 z)
¬0? : ∀ z → Dec (¬0 z)
is+? : ∀ z → Dec (is+ z)
_≤?_ : Decidable _≤_
```

2. $(Num, =, \leqslant)$ *forms a total order.* For example,

```
decTotalOrder : DecTotalOrder
decTotalOrder = record
  {Carrier = ℤ
  ; _≈_   = _≡_
  ; _≤_   = _≤_
  ; isDecTotalOrder = record
    {isTotalOrder = record
      {isPartialOrder = record
        {isPreorder = record
          {isEquivalence = isEquivalence
          ; reflexive   = refl′
          ; trans       = trans
          ; ∼-resp-≈   = resp₂ _≤_
          }
        ; antisym = antisym
        }
      ; total = total
      }
    ; _≟_ = _≟_
    ; _≤?_ = _≤?_
    }
  }
```

3. *some other properties for ordering functions.* For example,

   $+$ preserves $\leqslant$

```
+-pres₂′ : ∀ {a b c d} → a ≤ b → c ≤ d → a + c ≤ b + d
```

   $+$ cancellation for $\leqslant$

```
+-cancel′ : ∀ a {b c} → a + b ≤ a + c → b ≤ c
+-cancel′ a = ℤ₀.+l-cancel′ ⌜ a ⌝ ∘ +compl≤
```

   Also the properties to form a setoid we mentioned before is also included in this file. The three basic thoerems which proves the equivalence relations, namely "reflexivity", "symmetry" and "transitivity" are widely used.

Commutative Ring (or Field) In the second part, there are a lot of axioms for operations which we used a lot in computations. For example,

1. the identity of $+$: $0 + z = z$ and $z + 0 = z$

```
+-identity : Identity (+ 0) _+_
+-identity = 0 +z≡z, z+0≡z
```

2. The Commutativity of +: $m * n = n * m$

```
+-comm : Commutative _+_
```

3. The Associativity of +: $(a + b) + c = a + (b + c)$

```
+-assoc : Associative _+_
```

4. $-\_$ is inverse: $x + (-x) = 0$ and $(-x) + x = 0$

```
+-inverse : Inverse (+ 0) -_ _+_
+-inverse = +-leftInverse, +-rightInverse
```

5. zero times any number is still zero:
   $0 * z = 0$
   $z * 0 = 0$

```
*-zero : Zero (+ 0) _*_
*-zero = 0 *z≡0, z*0≡0
```

6. The distributivity of + and $*$:
   $a * (b + c) = a * b + a * c$
   $(b + c) * a = b * a + c * a$

```
distrib-*-+ : _*_ DistributesOver _+_
distrib-*-+ = dist^l, dist^r
```

There are more properties than which are similar to these and finally they could form a ring or even a field. for example,

```
commutativeRing : CommutativeRing
commutativeRing = record
  {_+_   = _+_
  ; _*_  = _*_
  ; -_   = -_
  ; 0 #  = (+ 0)
  ; 1 #  = (+ 1)
  ; isCommutativeRing = isCommutativeRing
  }
```

With commutative ring we could build the ring solver which is an automatic prover and solver for simple propositions.

```
import Algebra.RingSolver.Simple as Solver
import Algebra.RingSolver.AlmostCommutativeRing as ACR
module RingSolver =
  Solver (ACR.fromCommutativeRing commutativeRing)
```

Advanced Properties The advanced properties part imports the second part so that some more advanced properties can be proved easier after we have the theorems proved before.

1. Integrity. The Integerity is the name for properties and it is also called mulitiplication cancellation fucntions. We need a non-zero condition for the cancelled term. For example,

   ```
   I-integrity : ∀ {m n} a → (p : ¬0 a) → a * m ≡ a * n → m ≡ n
   ```

2. $*$ preserves $\leqslant$, if n is non-negative in $a \leqslant b \rightarrow n * a \leqslant n * b$

   ```
   *-pres′ : ∀ {a b} n → a ⩽ b → (n, 0) * a ⩽ (n, 0) * b
   ```

3. Solve an equation: if $m * n \sim 0$ and m is not 0 then n must be 0

   ```
   solve0 : ∀ m {n} → (p : ¬0 m) → m * n ≡ + 0 → n ≡ + 0
   solve0' : ∀ m {n} → (p : ¬0 m) → n * m ≡ + 0 → n ≡ + 0
   ```

4. To simplfy some common actions to exchange the positions of some elements in polynomials when construct proof manually rather than using ring solvers, there is a set of exchange functions defined here. For example,

   ```
   *-ex₁ : ∀ a b c → a * (b * c) ≡ c * (b * a)
   *-exchange₁ : ∀ a b c d → (a * b) * (c * d) ≡ (a * d) * (c * b)
   *-exchange₂ : ∀ a b c d → (a * b) * (c * d) ≡ (c * b) * (a * d)
   ```

Besides these, when I need a lemma that seems useful generally or easier to prove in current number environment, I will also prove it in this part. For example,

```
*+-simp : ∀ a b → (+ a) * (+ b) ≡ + (a ℕ* b)
-*cong : ∀ m n → m * n ≡ (- m) * (- n)
```

Properties combined After all, a single file called "Properties" which combine all the properties is necessary to provide a easy access to all these properties. For example,

```
module Data.Integer.Properties where
open import Data.Integer.Properties.BasicProp public
open import Data.Integer.Properties.CommutativeRing public
open import Data.Integer.Properties.AdvancedProp public
```

It is also a better way to manage what properties to put in the whole properties file. After that we can add a set of properties without affecting the rest of properties if we put all properties in one single file.

8 Implementation

8.1 Proving theorems in Agda

After getting the numbers defined, we will discuss the approaches and tools used to prove theorems in Agda. Computer-aid formal reasoning is different but has some similarity to the reasoning in regular mathematics. We could omit some common properties like "commutativity" and the target of what the properties are applied to. In the real world, verification is a big problem as everyone will make mistakes and no "checker" will warn you. However to prove theorems in programming languages, although the rules are much stricter so that we have to present each small property used, we will make less mistakes by writing more complete proofs and by using reliable checker who is in charge of verification. In my experience, when I implemented a piece of proof in Agda, I found that I ignored some steps which looks simple but is actually not been proved. However the approaches of proving in Agda are similar to mathematical reasoning. I summarized two ways to organise the proofs which can be mixed used,

1. *Case analysis – Pattern match.* For example, natual numbers have two cases, zero and successor of another natural number. In regular mathematics, we could prove proposition of natural numbers using inductive reasoning; In Agda we could use pattern match as below,

   ```
   n*0+0=0 : ∀ {n} → n * 0 + 0 ≡ 0
   n*0+0=0 {zero} = refl
   n*0+0=0 {suc n} = n*0+0=0 {n}
   ```

   You can see the pattern match is the implementation of case analysis in programming languages according to Curry-Howard correspondence [24].

2. *Using lemmas – Using auxiliary functions.* In regular mathematics, we often prove complicated theorems by proving small lemmas. In programming languages, we can implement this idea trivially by using auxiliary functions. For example,

   ```
   ass-lem : ∀ a b c d e f → (a * (+suc b) + c * (+suc d))
   +-assoc : Associative _+_
   +-assoc (a /suc ad) (b /suc bd) (c /suc cd) =
     *-cong (ass-lem a bd b ad cd c) \$
     +_ ⋆ ⟨ ℕ.*-assoc (suc ad) (suc bd) (suc cd) ⟩
   ```

To use lemmas not only divides huge tasks into small pieces that is easier to prove, but also helps us reduce repeated proofs.

Proving in Agda also has some distinctions with proving in Coq. There is an impredicative universe *Prop* in Coq but not in Agda. The pattern matching mechanism is more flexible in Agda[15]. While in Coq we have tactics to prove the theorems, in Agda we can only write proofs in the form of programs until now. When doing theorem proving we are also equipped with two useful tools,

1. *Formal reasoning structure.* It is a simulation of the formal reasoning process in regular mathematics. We could prove propositions step by step without using transitivity every time. Every step is explict so that it is more readable. This is an example of using formal reasoning structure,

   ```
   begin
     x+ ℕ+ z- ℕ+ (y+ ℕ+ y-) ≡⟨ exchange x+ z- y+ y- ⟩
     x+ ℕ+ y- ℕ+ (y+ ℕ+ z-) ≡⟨ eq1 +=' eq2 ⟩
     z+ ℕ+ y- ℕ+ (y+ ℕ+ x-) ≡⟨ exchange z+ y- y+ x- ⟩
     z+ ℕ+ x- ℕ+ (y+ ℕ+ y-) ■
   ```

   It starts with begin and ends with QED. It allows reflexive refinement during proofs.

2. *Ring solver.* Ring solver is an automatic prover for simple equations which only contains operations including addition, negation, subtraction and multiplication. We can set up ring solver based on the proof of semiring or ring. It is especially helpful for complicated polynomials which have too many elements. To prove a proposition we only need to send it to the solver and the solver will automatically generate the proof we need. However every coin has two sides. Although it saves the time for the programmer it takes much longer time for type-checking. This is an example of how to use ring solver,

   ```
   z+0=z : RightIdentity (0, 0) _+_
   z+0=z (a', b') = solve 2 (λ a b → ...) refl a' b'
   ```

8.2 Problems and Changes

At first, starting using Agda which was unfamiliar to me is really a big chanllenge. Although it is similar to Coq to some extent, and similar to Haskell which Agda itself is implemented in, it is still a new way to program and do theorem proving. Because of the inadequate understanding of Agda and its library code, I encoutered several problems in undertaking this project.

Efficiency Problem The efficiency problem of type checker has been mentioned before. The computation time of type checking is not only related to the amount of the code in a single files but also to the complexity of referenced functions and properties. Therefore although some proofs for rational numbers looks similar to the ones for integers, it takes about double time to check the file. In the other hand, ring solver which is an automatical theorem prover can simplify the proof a lot but also sacrifice the efficiency instead. At first I use the ring solver of natural numbers to prove theorems for setoid integers, it works well but takes more time. However when I try to use the ring solver set up based on the commutative ring of setoid integers, the time spent on type checking is too long to refine the goals. The problem is extremely serious when I use the old version of rational numbers to define real numbers. At that time my computer were working work fully-loaded for about half an hour to make the definition of reals completely checked.

Changes. To solve the efficiency problem, firstly I divide large files into more parts such as the field of rational numbers. To reduce computation for type checker I have to give up the ring solver and choose to construct proofs manually. Then I use the formal reasoning structure introduced before. For some properties it is clear and short. But for bigger proofs, the intermediate steps are unecessary to present while they are too long to be structured well. After all I choose to prove theorems without the tools because I can write more compact proofs. Learning from some tricks used in Thomas Anberrée's quotient definition, I using a set of symbols to replace the properties to make them more comprehensive. For example,

```
⟨_⟩ : ∀ {A : Set} → Symmetric (_≡_ {A = A})
⟨_⟩ = sym
_>≡<_ : ∀ {A : Set} → Transitive (_≡_ {A = A})
_>≡<_ = trans
_⋆_ : ∀ {A B : Set} (f : A → B)
  {x y} → x ≡ y → f x ≡ f y
_⋆_ = cong
```

Unary function like "symmetry" can be more meaningful when it includes the equation it applied to. For transitivity, the symbol like a chain links all intermidiate proofs together which are more clearly presented. The congurence is also replaced by a infix operator to make the programs similar to the goals.

Then I revise all the lemmas to generalise the common part and discard useless ones. After rewritting all of them, the proofs become much more shorter and comprehensive with increased effciency to use. For instance, in the older version, to prove the transitivity for rational numbers, I defined 10 lemmas and there are nearly 50 lines of codes. in the new version there are no lemmas for it and the proof is as short as 10 lines. The time spent on type checking the definition with real number is 10 minutes for the older version, but 10 seconds for the newer version.

Stuck in difficult theorems For some theorems like the distributivity for normal integers, transitivity for setoid integers and rational numbers and integrity which are comparative much more difficult to prove. At first when I try to prove these properties, I was stuck for very long time. I followed the common steps to pattern match and prove case by case. But later I found there are too much cases and the types of goals are beyond my understanding.

To reduce the number of cases, firstly I defined setoid integers and redefined integers as we discussed before. Then I found that to prove lemmas first could also reduce cases. Defining normal integers based on setoid integers makes the proof simpler as well. We can prove theorems of normal integers by proving the ones of setoid integers and proving the isomorphism between the setoid the normal definition. The general quotient theory of Thorsten Altenkirch and Thomas Anberrée could provide easier way to lift theorems, however the current version still has some small questions that restrict it.

The types of goals are always unfolded and unreadable. Therefore I use formal reasoning structure at first. Although I rewrite the code without this tool in the end, it benefited refinement of goals with clear steps. To formalise a proof in Agda, it is better to configure it out in regular mathematics by hand. This is what I have learnt from undertaking this project. It also helps me find mistakes. For example, when I try to prove integrity, I forgot to add the condition of non-zero of the cancelled term. Wrong propositions are allowed to be written but unlikely to be proved. When I try to configure out the proof manually, I found this stupid mistake by trying to replace the cancelled term using 0.

When I explored in the library code, I found there are many useful functions and tools that could simplfy the proofs. To understand these codes more deeply, I tried to write the programs by myself. At the same time, I also learnt a lot from the library code writtern by Nils Anders Danielsson.

Other problems There are some other small problems that related to Agda itself. It is better use extraction functions rather than pattern match in defining relations and operations so that they are decoupled with the definition of setoid number,

Definition 8.1. $\_\sim\_\ :\ \mathsf{Rel}\ \mathbb{Z}_0$
$(\mathsf{x+}, \mathsf{x-}) \sim (\mathsf{y+}, \mathsf{y-})\ =\ (\mathsf{x+}\ \mathbb{N}{+}\ \mathsf{y-}) \equiv (\mathsf{y+}\ \mathbb{N}{+}\ \mathsf{x-})$

Definition 8.2. $\mathsf{pos}\ :\ \mathbb{Z}_0 \to \mathbb{N}$
$\mathsf{pos}\ (\mathsf{m}, \_)\ =\ \mathsf{m}$
$\mathsf{neg}\ :\ \mathbb{Z}_0 \to \mathbb{N}$
$\mathsf{neg}\ (\_, \mathsf{n})\ =\ \mathsf{n}$
$\_\sim\_\ :\ \mathsf{Rel}\ \mathbb{Z}_0$
$\mathsf{x} \sim \mathsf{y}\ =\ (\mathsf{pos}\ \mathsf{x}\ \mathbb{N}{+}\ \mathsf{neg}\ \mathsf{y}), (\mathsf{pos}\ \mathsf{y}\ \mathbb{N}{+}\ \mathsf{neg}\ \mathsf{x})$

However for the secound version, there is problem when using the transitivity proved based on it. Everytime when I use the transitivity I have to explicitly show the three elements involved. For the convenience, I gave up the version with extraction functions.

Also I found when we use "with", we could pattern match on impossible cases. For example, when I pattern match on "suc n", it still requires to prove the case "zero". These problems can be solved by developers of Agda and It will become more powerful in the future.

During the period of project, Agda has also changed a lot in both itself and its standard library. The relations, operations, algebra strucutures are mostly changed due to the added feature universe polymorphism. It enables programmers to define polymorphic data types on several levels [18]. The pattern match on record is also available right now in the development version 2.2.7 which will also benefit the theorem proving a lot.

9 Evaluation

9.1 How to use the code

Most of the code are tested to worked under Agda 2.2.6 except the definitions of normal rational number and real number. These two definitions use the pattern match of record type which is only available from 2.2.7 version which is a development version.

To use the numbers it is enough to just import the definition file of each type of numbers. What should be noted is that, the setoid definitions require the "Data.Product" library code and In all kinds of these numbers, "Data.Nat" which includes the definition of natural numbers is also essential. According to the dependencies, we should include the lower level of numbers if we want use the higher level ones.

To prove the properties, it is enough to just import the second part to use the ring solver for simple propositions. In addition we should add the following code,

```
open RingSolver
  using (prove; solve; _:=_; con; var; _:+_; _:*_; :-_; _:-_)
```

I have tested the ring solvers of each type of numbers. For example,

```
t : ∀ a b c d → (a + b) * (c + d) ∼ a * c + a * d + b * c + b * d
t = solve 4
  (λ a b c d → (a :+ b) :* (c :+ d) := a :* c :+ a :* d :+ b :* c :+ b :* d)
  zrefl
```

It takes about one minute to verify the correctness of the proof. Ring solver is a good choice for proving unusual used propositions.

Importing the "Properties" files, we could construct the proof manually. Some symbols for generic properties are defined in one single file "Symbols".

9.2 Feedback

I have changed the naming convention, organisation of files, layout of code and improve the readability of the code based on the feedback from Nils Anders

Danielsson who is the main designer of the standard library of Agda. I also get some feedback from some friends of mine who have knowledge of Agda. The feedbacks reflect some problems,

- *The definitions is hard to understand.* The definition requires more knowledge beyond the simple mathematics, especially for the real numbers, which seems difficult to use. I think the problem could be solved with reading the discussion of definitions of real numbers.
- *Ring solver.* It is quite slow compared to what they have exprienced before but is still acceptable. The positive aspect is that they find it is interesting and helpful to use ring solver.
- *Description.* The explanation for each property facilitates the understanding of the code. But if the emacs interface could fold the description with a button, the code will look clearer.

10 Summary and future work

In this project, the various approaches of representing numbers in programming languages are reviewed firstly. Then the exploration of best representation in Agda are undertaken along with the implementation. The definitions were gradually refined by considering the usability, readability and the effects on theorem proving. While being more familiar with Agda and computer-aid formal reasoning, I make use of the tools and library code to improve the implementation. I also learnt from the definition of numbers in other languages by literature review and from codes written by others. Taking the usability and efficiency of properties into consideration, I changed the approaches to prove them several times. The final version has been tested to be acceptable when using the properties proved.

Agda provides increasing number of beneficial features for theorem proving so that it will attract more users. The work of representing numbers can be extended futher in several aspects. Within Agda, other ways to represent each kind of number especially some other construction of exact real number are worth implementing, for example, the dedekind cut and infinite streams may perform better in efficiency. The implementation of quotient types is also a potential extention and we could apply it to the mechanism between setoid and normal forms. It is also interesting to use these definitions and properties in proving more complicated theorems. The work can also be extended in other languages. To review and compare the different representation of numbers in different proof assistants is also a nice topic.

References

[1] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. Pisigma: Dependent types without the sugar. submitted for publication, November 2009.

[2] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[3] Errett Bishop and Douglas Bridges. *Constructive Analysis*. Springer-Verlag, Berlin, Heidelberg, 1985.

[4] Ana Bove and Peter Dybjer. *Dependent Types at Work*. Springer-Verlag, Berlin, Heidelberg, 2009.

[5] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda — a functional language with dependent types. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] Douglas S. Bridges. Constructive mathematics: a foundation for computable analysis. *Theoretical Computer Science*, 219(1-2):95 – 109, 1999.

[7] Georg Cantor. On a property of the collection of all real algebraic numbers. *English translation: Ewald 1996*, pages 840 – 843, 1874.

[8] Jawahar Chirimar and Douglas J. Howe. Implementing constructive real analysis: Preliminary report. In *Constructivity in Computer Science, Summer Symposium*, pages 165–178, London, UK, 1992. Springer-Verlag.

[9] Alberto Ciaffaglione and Pietro Di Gianantonio. A certified, corecursive implementation of exact real numbers. *Theoretical Computer Science*, 351(1):39 – 51, 2006. Real Numbers and Computers.

[10] Pietro Di Gianantonio. Real number computability and domain theory. *Inf. Comput.*, 127(1):11–25, 1996.

[11] Herman Geuvers and Milad Niqui. Constructive reals in coq: Axioms and categoricity. In *TYPES '00: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 79–95, London, UK, 2002. Springer-Verlag.

[12] Claire Jones. Completing the rationals and metric spaces in lego. In *Papers presented at the second annual Workshop on Logical environments*, pages 297–316, New York, NY, USA, 1993. Cambridge University Press.

[13] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[14] B. Nordström, K. Petersson, and J. M. Smith. Martin löf's type theory, 1990.

[15] Ulf Norell. Dependently typed programming in agda. Available at: http://www.cse.chalmers.se/ ulfn/papers/afp08tutorial.pdf, 2008.

[16] Russell O'Connor. Certified exact transcendental real number computation in coq. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 246–261, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] The Agda Wiki. Main page, 2010. [Online; accessed 13-April-2010].

[18] The Agda Wiki. Universe polymorphism, 2010. [Online; accessed 30-April-2010].

[19] Wikipedia. Coinduction — Wikipedia, the free encyclopedia, 2010. [Online; accessed 20-April-2010].

[20] Wikipedia. Construction of the real numbers — Wikipedia, the free encyclopedia, 2010. [Online; accessed 20-April-2010].

[21] Wikipedia. Coq — Wikipedia, the free encyclopedia, 2010. [Online; accessed 20-April-2010].

[22] Wikipedia. Four color theorem — Wikipedia, the free encyclopedia, 2010. [Online; accessed 26-April-2010].

[23] Wikipedia. Lazy evaluation — Wikipedia, the free encyclopedia, 2010. [Online; accessed 20-April-2010].

[24] Wikipedia. Pattern matching — Wikipedia, the free encyclopedia, 2010. [Online; accessed 26-April-2010].

[25] Wikipedia. Taylor series — Wikipedia, the free encyclopedia, 2010. [Online; accessed 20-April-2010].