Formalizing Real Numbers in Agda

Martin Lundfall

June 12, 2015

Abstract

With his iconic book *Foundations of Constructive Analysis* (Bishop and Bridges 1985), Errett Bishop presented the constructive notion of a real number and showed that many of the important theorems of real analysis could be proven without using the law of the excluded middle. This paper aims to show how this notion can be formalized in the dependently typed programming language of Agda. Using the Agda Standard Library (v0.9) and additional work by the GitHub user sabry (sabry 2014), a major step towards formalizing the definition of real numbers and the equivalence relation on them is taken. In the process, an alternate definition of rational numbers in Agda is presented and some important statements on rationals are proven.

Contents

1 Introduction

1.1 Formal and informal mathematics

The term "Formalized mathematics" might strike readers unfamiliar with the topic as a pleonasm. What could possibly be more formal than the rigor in which mathematical proofs are presented? But as we will see, a formal approach to mathematics looks fundamentally different from traditional mathematical reasoning. In formal mathematics, one strives to lay out the complete path from axioms to conclusion, without any step along the way being subject to interpretation. To illustrate what is meant by this, consider a simple informal proof showing that the sum of two odd numbers is equal.

Definition 1 *Even and odd numbers.*
A natural number n is ***odd*** *if it can be written as n = 2k+1, for some natural number k.*
A natural number n is ***even*** *if it can be written as n = 2m, for some natural number m.*

Theorem 1 *The sum of two odd numbers is equal.*
Consider the odd numbers m = 2k + 1 and n = 2j + 1. Their sum can be rewritten as:
(2k + 1) + (2j + 1) = 2k + 2j + 2 = 2(k + j + 1)
Since k, j and 1 are all natural numbers, the right hand side of this equation is on the form $2m$ *and therefore even.* □

The proof is trivial and the result comes as no surprise. However, since it does not explicitly state how the conclusion is derived from the axioms, it is essentially informal. For a proof like this to be completely convincing, the reader must be able to fill in the gaps of our reasoning themselves and understand how the expressions we are using are being modified.

For example, in the step $2n_1 + 2n_2 + 2 = 2(n_1 + n_2 + 1)$, we assume familiarity with the definition of addition and multiplication and utilize the distributive property of multiplication over addition, although this is not stated. To accept that the left-hand side of the equation indeed equals the right-hand side we must recognize the transitivity of the equality relation. We are assumed to know how integers are defined, and accept that $n_1 + n_2 + 1$ is an integer without further motivation. Even the fact that our proof is written in plain English could be a source of ambiguity.

Of course, the fact that this proof contains some implicit reasoning does not jeopardize its validity. The omitted steps are all trivially proven, and the proof is easy to follow. If mathematics were always to be conducted with absolute rigor and formality, it would be a very tedious process.

Formalized mathematics is mathematical reasoning following a set of strict rules of syntax, and is therefore very amenable for algorithmic proof checking which can be done by a computer. This is why much of the work in formalized mathematics is being done in programming languages and proof assistants specifically designed for this purpose.

1.2 Classical and constructive mathematics

When formalizing mathematics, one has to make a decision on what logical framework to use. This is a disputed topic in logic, with many different answers to the question of what the "proper" foundation of mathematical reasoning should be. Two major and opposing approaches are classical and constructive mathematics. One of the biggest disagreements between these two sides lies in in the logical axiom known as the law of the excluded middle, which states that every proposition either has to be true or false. While this is accepted in classical mathematics, constructivists claim that this is not necessarily the case.

A common example is the famous unproved mathematical statement known as Goldbach's conjecture, which states: "every even integer greater than 2 can be expressed as the sum of two primes". According to constructivists, since no proof or disproof of Goldbach's conjecture exists, we are not justified in asserting "Goldbach's Conjecture is either true or false", a statement that is regarded as true in classical mathematics.

The difference in viewpoint between constructivists and classical mathematicians has some significant consequences for statements about real numbers. A common way of constructing a real number x is by letting x be a sequence of rational numbers $(x_i) = (x_0, x_1, x_2, \dots)$ where the difference between two elements $|x_m - x_n|$ becomes arbitrarily small as m and n increases.

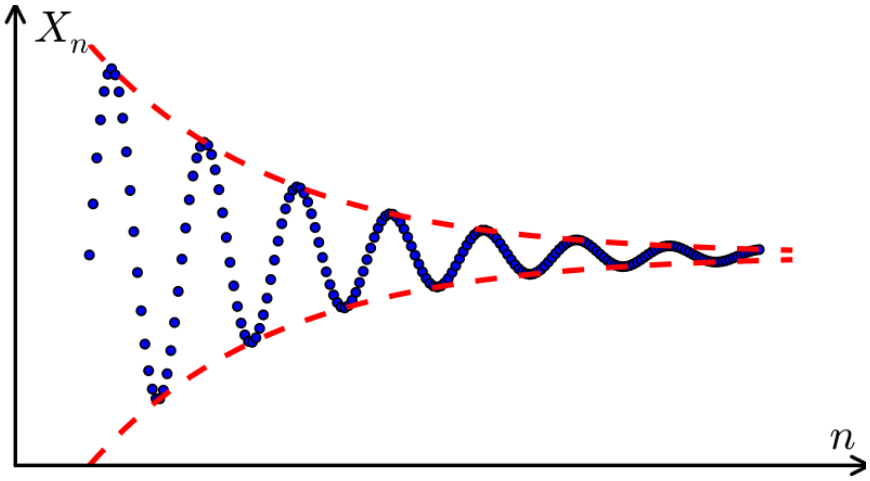

Figure 1: A sequence (x_i) of rational numbers

To investigate whether a real number, x is positive or negative, we consider elements of the sequence (x_i) and look at what value they approach as i becomes larger. For both constructivists and classical mathematicians, it is apparent that a number x cannot simultaneously both be *greater than or equal to* and *smaller than* a given value, but what are the consequences of this? For classical mathematicians, this means that we are

justified in asserting:

"For a real number, x, either $x \geq 0$ or $x < 0$"

while constructivists maintain that we cannot make such a claim. Constructivists argue that since there is a possibility that the sequence which makes up x alternates between being smaller than and greater than 0, we cannot be certain that either $x \geq 0$ or $x < 0$ holds.

1.3 Formal mathematics in Agda

In this work, I formalize the constructive notion of real numbers as described by Errett Bishop in Constructive Analysis (Bishop and Bridges 1985) in the proof assistant Agda. Agda is a functional programming language with a syntax similar to Haskell that essentially deals with two fundamental elements of mathematics: sets and functions. As an example, here is the definition of a natural number and addition with natural numbers, $\mathbb{N}$ in Agda:

```
data ℕ : Set where
  zero : ℕ
  suc  : (n : ℕ) → ℕ

_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)
```

We see that the natural numbers make up a set, whose elements can be created by two constructors, `zero` and `suc`. `suc` is short for the successor function, which given any natural number `n`, constructs the number `suc n`.
The number 3 for example, is written as `suc (suc (suc (zero)))` in this way. When defining functions on natural numbers, we first state the type of the function (addition takes two natural numbers and outputs a natural number as an answer) and then by matching the definition of $\mathbb{N}$ state how the output is constructed.
The fundamental equality that states that the left-hand side of the equation is identical to the right-hand side is also a set in Agda, with only one constructor:

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

If we ignore the arguments given in curly brackets for now, we see that `_≡_` is a set that can be constructed by refl if the type checker in Agda interprets the left-hand and right-hand sides of the equation as identical. To see how proofs are constructed in Agda, we will present a formalized version of the previously given proof that the sum of two odd numbers is even. Before we do so we need to know how to manipulate expressions using the function `cong`.

```
cong : ∀ {a b} {A : Set a} {B : Set b}
       (f : A → B) {x y} → x ≡ y → f x ≡ f y
cong f refl = refl
```

Again ignoring the arguments given in curly brackets, we see that this is another way of constructing an element of the type _≡_ given a function f and another element of _≡_. In other words, given a proof that $x \equiv y$, we can use `cong` to show that $f(x) \equiv f(y)$. We can define odd and even numbers in Agda in the same manner as we did in the informal definition, where n is even if there exists (written as Σ in Agda) a k, such that $n = 2k$ and n is odd if there exists a k, such that $n = 2k + 1$. The formalized version of this is as follows:

```
even : ℕ -> Set
even n = Σ ℕ (λ k -> (k + k) ≡ n)

odd : ℕ -> Set
odd n = Σ ℕ (λ k -> suc (k + k) ≡ n)
```

Before are ready to give the proof we need a couple of more lemmas. The following functions `+-assoc` and `+-comm` are proofs that addition is a commutative and associative function. Notice how `refl` and `cong` are used below to create instances of the type _≡_.

```
+-assoc : ∀ m n o → (m + n) + o ≡ m + (n + o)
+-assoc zero    _ _ = refl
+-assoc (suc m) n o = cong suc (+-assoc m n o)

+-right-identity : ∀ n → n + 0 ≡ n
+-right-identity zero = refl
+-right-identity (suc n) = cong suc (+-right-identity n)

+-suc : ∀ m n → m + suc n ≡ suc (m + n)
+-suc zero    n = refl
+-suc (suc m) n = cong suc (+-suc m n)

+-comm : ∀ m n → m + n ≡ n + m
+-comm zero    n = sym (+-right-identity n)
+-comm (suc m) n =
  begin
    suc m + n
  ≡⟨ refl ⟩
    suc (m + n)
  ≡⟨ cong suc (+-comm m n) ⟩
    suc (n + m)
  ≡⟨ sym (+-suc n m) ⟩
    n + suc m
  ■
```

The last lemma, `+-comm`, uses a method to prove equalities in Agda called equational reasoning in the last case. It is useful when multiple modifications of the left-hand side of the equation are required before getting an expression which is equal to the right. It has three components. The `begin` function simply enables the use of equational reasoning and the other two functions. The `_≡⟨ _ ⟩_` function takes three arguments, two expressions on either side and in between the angle brackets a proof that the two expressions are identical. The final ■ -function signifies that the proof is complete. Combined, these three functions creates an element of `_≡_` with the initial left-hand side and the final right-hand side of the equation as arguments.

It is worth mentioning the fact that there is no difference between a function and a proof in Agda. Proofs are simply functions that has the statement that is to be proven as type, and the path from assumptions to conclusion explicitly given in the definition of the function. This connection between computer programs and formal logic was discovered by Haskell Curry and William Alvin Howard in the middle of the 20th century and is known as the Curry-Howard correspondence (Curry, Hindley and Seldin 1980).

We are now ready to give a formalized version of the previously given proof that the sum of two odd numbers is equal. The idea of the proof is the same, but for every step in our reasoning we are required to use `cong` to modify our expression.

```
o+o : {m n : ℕ} → odd n → odd m → even (n + m)
o+o {m} {n} (j , p) (k , q) = suc (j + k) ,
  (begin
    suc (j + k + suc (j + k))
≡⟨ cong suc (ℕ+-comm (j + k) (suc (j + k)))  ⟩
    suc (suc (j + k + (j + k)))
≡⟨ cong (λ a -> suc (suc a)) (ℕ+-assoc j k (j + k))  ⟩
    suc (suc (j + (k + (j + k))))
≡⟨ cong (λ a -> suc (suc (j + a))) (sym (ℕ+-assoc k j k))  ⟩
    suc (suc (j + (k + j + k)))
≡⟨ cong (λ a -> suc (suc (j + (a + k)))) (ℕ+-comm k j)  ⟩
    suc (suc (j + (j + k + k)))
≡⟨ cong (λ a -> (suc (suc (j + a)))) (ℕ+-assoc j k k)  ⟩
    suc (suc (j + (j + (k + k))))
≡⟨ cong (λ a -> suc (suc a)) (sym (ℕ+-assoc j j (k + k)))  ⟩
    suc (suc (j + j + (k + k)))
≡⟨ cong suc (ℕ+-comm (suc (j + j)) (k + k)) ⟩
    suc (k + k) + suc (j + j) ≡⟨ cong₂ _+_  q p  ⟩
    m + n ≡⟨ ℕ+-comm m n  ⟩
    n + m
  ■)
```

When comparing this proof with the previous, informal one, one might gain an understanding of how much is taken for granted in the common, informal approach to mathematics. For a more thorough guide of Agda, I recommend checking out one of

tutorials available on the Agda wiki at:
http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Othertutorials.

The remaining segment of this document will assume some familiarity with theorem proving in Agda.

2 Constructing real numbers in Agda

This work is an Agda implementation of the constructive definition of real numbers as presented by Errett Bishop in *Constructive Analysis* (Bishop and Bridges 1985). It works with the current Agda Standard Library (v0.9) with one modified module, `Data/Rational.agda`. It is implemented in two new modules, `Data/Rational/Properties.agda` and `Data/Real.agda`.

These three modules are listed in their current state as attachments to this document. Updated versions can in the future be found at my GitHub at: https://github.com/MrChico/agda-stdlib.

2.1 Redefining rational numbers

The current current Agda Standard Library (v0.9) contains only a small module `Data.Rational` in which rational numbers are defined, but lacks the necessary functions to define real numbers. Therefore I have used the contributions made by the GitHub user sabry at (https://github.com/sabry/agda-stdlib/blob/master/src/Data/Rational.agda) as a springboard for this work.
I have made a significant change to the definition of a rational number by not requiring the numerator and denominator to be coprime. While working with rationals it became apparent that requiring coprimality made theorem proving highly demanding, both for human effort and computer power. Therefore, instead of:

```
record ℚ : Set where
  field
    numerator     : ℤ
    denominator-1 : ℕ
    isCoprime     : True (C.coprime? ∣ numerator ∣ (suc denominator-1))
```

we simply write:

```
record ℚ : Set where
  constructor _÷suc_
  field
    numerator     : ℤ
    denominator-1 : ℕ
```

Skipping the requirement of coprimality makes certain statements easier to prove, and defining functions a lot simpler. On the other hand, consider the equivalence relation on rational numbers:

```
_≃_ : Rel ℚ Level.zero
p ≃ q = numerator p ℤ.* (+ suc (denominator-1 q)) ≡
        numerator q ℤ.* (+ suc (denominator-1 p))
  where open ℚ
```

Since we can have several elements in the same equivalence class, this relation is no longer synonymous with the identically equal relation, $\equiv$. This makes us unable to modify expressions with functions like cong and subst, and instead we have to show that the functions we define on rationals preserve the equivalence relation.

It is also worth mentioning that we would rarely require rational numbers to be given on their coprime form in informal mathematics. The new definition makes working with rational numbers closer to how one might work with rational numbers using pen and paper. Compare the formal definitions with Bishop's definition of a rational number:
" For us, a *rational number* will be an expression of the form p/q where p and q are integers with $q \neq 0$. "
What still differs between the two definitions is the fact that Bishop allows negative numbers in the denominator, whereas we do not. This minor change does not have any significant impact on how we work with rational numbers, and is only for convenience. Constructing rational numbers in Agda by just allowing natural numbers larger than zero as denominators makes our definition slightly nicer.

With the new definition, the proof that _≃_ is an equivalence relation on rational numbers had to be rewritten. It is listed below:

```
--This is an equivalence relation
isEquivalence : IsEquivalence _≃_
isEquivalence = record {
  refl = refl ;
  sym = P.sym ;
  trans = λ {a}{b}{c} -> trans {a}{b}{c}
  }
    where
    trans : Transitive _≃_
    trans {a ÷suc b} {f ÷suc g} {x ÷suc y} ag≃fb fy≃xg =
      cancel-*-right (a ℤ.* (+ suc y)) (x ℤ.* (+ suc b)) (+ suc g) (λ {()})
        (P.trans ayg≃fby fby≃xbg)
      where
        agy≃fby : (a ℤ.* + suc g ℤ.* + suc y ≡ f ℤ.* + suc b ℤ.* + suc y)
        agy≃fby = cong (λ j -> (j ℤ.* + suc y)) (ag≃fb)
        ayg≃fby : (a ℤ.* + suc y ℤ.* + suc g ≡ f ℤ.* + suc b ℤ.* + suc y)
        ayg≃fby = P.trans (*-assoc a (+ suc y) (+ suc g))
          (P.trans (cong (λ j -> (a ℤ.* j )) (*-comm (+ suc y) (+ suc g)))
            (P.trans (P.sym (*-assoc a (+ suc g) (+ suc y))) agy≃fby))
        fyb≃xgb : (f ℤ.* + suc y ℤ.* + suc b ≡ x ℤ.* + suc g ℤ.* + suc b)
        fyb≃xgb = cong (λ j -> j ℤ.* (+ suc b)) fy≃xg
```

```
fby≃xgb : (f ℤ.* + suc b ℤ.* + suc y ≡ x ℤ.* + suc g ℤ.* + suc b)
fby≃xgb = P.trans (*-assoc f (+ suc b) (+ suc y))
  (P.trans (cong (λ j -> (f ℤ.* j )) (*-comm (+ suc b) (+ suc y)))
    (P.trans (P.sym (*-assoc f (+ suc y) (+ suc b))) fyb≃xgb))
fby≃xbg : (f ℤ.* + suc b ℤ.* + suc y ≡ x ℤ.* + suc b ℤ.* + suc g)
fby≃xbg = P.trans (P.trans (fby≃xgb) (*-assoc x (+ suc g) (+ suc b)))
  (P.trans (cong (λ j -> (x ℤ.* j)) (*-comm (+ suc g) (+ suc b)))
    (P.sym (*-assoc x (+ suc b) (+ suc g))))
```

A minor change also had to be made to show that the rational numbers are a totally ordered set with decidable equality. Compare the version of `Data/Rational.agda` found in the attachments of this document with the version in the Agda Standard Library (v0.9) for details.

2.2 Definition and equivalence of real numbers

The definition of a real number is almost identical to the definition given by Bishop, with one minor modification. Where Bishop describes a sequence as a mapping of the strictly positive integers to rationals, we will define it as a mapping from the natural numbers to rationals. This makes things a bit nicer. Where Bishop writes:

> A sequence of rational numbers is *regular* if
>
> $$|x_m - x_n| \leq m^{-1} + n^{-1} (m, n \in \mathbb{Z}^*)$$
>
> A *real number* is a regular sequence of rational numbers.

we will write the following:

```
--A real number is defined as sequence and
--a proof that the sequence is regular
record ℝ : Set where
  constructor Real
  field
    f : ℕ -> ℚ
    reg : {n m : ℕ} -> | f n ℚ.- f m | ℚ.≤ (suc m)⁻¹ ℚ.+ (suc n)⁻¹
```

Following Bishop, we then define an equivalence relation by:

```
-- Equality of real numbers.
infix 4 _≃_

_≃_ : Rel ℝ Level.zero
x ≃ y =  {n : ℕ} -> | ℝ.f x n - ℝ.f y n | ≤ (suc n)⁻¹ ℚ.+ (suc n)⁻¹
```

Equality looks very similar to regularity of a sequence, which makes sense. The difference between two rational numbers of the sequence becomes arbitrarily small as n increases. To prove that the relation $\simeq$ defines an equivalence relation on the real numbers we must show that it satisfies three conditions:

1. **Reflexivity** $x \simeq x$ for all $x \in \mathbb{R}$
2. **Symmetry** If $x \simeq y$ then $y \simeq x$ for all $x, y \in \mathbb{R}$
3. **Transitivity** If $x \simeq y$ and $y \simeq z$ then $x \simeq z$ for all $x, y, z \in \mathbb{R}$

Reflexivity is easy to show, all we have to do is give the proof that the sequence defining x converges:

```
--reflexitivity
refl≃ : {x : ℝ} -> (x ≃ x)
refl≃ {x} = ℝ.reg x
```

Symmetry seems like an easy task as well. Informally, all we need to show is that $|x_n - y_n| = |y_n - x_n|$. But showing this formally requires a bit of work. The strategy is as follows: first we show that $-(a-b) = b-a$ for all rational numbers a and b and then that $|-c| = |c|$ for a rational number c. To show the first part, we must first show that the statement is true for integers.
Here are the lemmas needed to show symmetry of the equivalence relation (found in `Data/Rational/Properties.agda`):

```
--Lemmas helping to show symmetry of the equivalence relation
--defined on the real numbers
⊖-swap : ∀ a b → a ℤ.⊖ b ≡ ℤ.- (b ℤ.⊖ a)
⊖-swap zero    zero    = refl
⊖-swap (suc _) zero    = refl
⊖-swap zero    (suc _) = refl
⊖-swap (suc a) (suc b) = ⊖-swap a b

ℤ-swap : (a b : ℤ) -> (ℤ.- (a ℤ.- b) ≡ b ℤ.- a)
ℤ-swap -[1+ n ] -[1+ n₁ ] = P.sym (⊖-swap n n₁)
ℤ-swap -[1+ n ] (+ zero) = refl
ℤ-swap -[1+ n ] (+ suc n₁) = trans (cong (λ a -> + suc (suc a))
  (+-comm n n₁)) (cong ℤ.+_ (P.sym (+-suc (suc n₁) n)))
ℤ-swap (+ zero) -[1+ n₁ ] = refl
ℤ-swap (+ suc n) -[1+ n₁ ] = cong -[1+_] (+-comm n (suc n₁))
ℤ-swap (+ zero) (+ zero) = refl
ℤ-swap (+ zero) (+ suc n₁) = cong ℤ.+_ (P.sym (+-right-identity (suc n₁)))
ℤ-swap (+ suc n) (+ zero) = cong -[1+_] (+-right-identity n)
ℤ-swap (+ suc n) (+ suc n₁) = P.sym (⊖-swap n₁ n)

ℚ-swap : (x y : ℚ) -> (- (y - x) ≡ x - y)
ℚ-swap (-[1+ n₁ ] ÷suc d₁) (-[1+ n₂ ] ÷suc d₂) =
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (-[1+ n₂ ] ℤ.* + suc d₁) (-[1+ n₁ ] ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))
```

```
ℚ-swap (-[1+ n₁ ] ÷suc d₁) ((+ zero) ÷suc d₂) =
  cong (λ a -> (-[1+ n₁ ] ℤ.* (+ suc d₂)) ℚ.÷suc (pred a))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap (-[1+ n₁ ] ÷suc d₁) ((+ suc n₂) ÷suc d₂) =
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (+ suc n₂ ℤ.* + suc d₁) (-[1+ n₁ ] ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap ((+ zero) ÷suc d₁) (-[1+ n₂ ] ÷suc d₂) =
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (-[1+ n₂ ] ℤ.* + suc d₁) (+ zero ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap ((+ suc n₁) ÷suc d₁) (-[1+ n₂ ] ÷suc d₂) =
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (-[1+ n₂ ] ℤ.* + suc d₁) (+ suc n₁ ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap ((+ zero) ÷suc d₁) ((+ zero) ÷suc d₂) =
  cong (λ a -> ((+ zero) ÷suc (pred a)))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap ((+ zero) ÷suc d₁) ((+ suc n₂) ÷suc d₂) =
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (+ suc n₂ ℤ.* + suc d₁) (+ zero ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap ((+ suc n) ÷suc d₁) ((+ zero) ÷suc d₂) =
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (+ zero ℤ.* + suc d₁) (+ suc n ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap ((+ suc n₁) ÷suc d₁) ((+ suc n₂) ÷suc d₂) =
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (+ suc n₂ ℤ.* + suc d₁) (+ suc n₁ ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))

ℚabs₁ : (x : ℚ) -> (| ℚ.- x | ≡ | x |)
ℚabs₁ (-[1+ n ] ÷suc d₁) = refl
ℚabs₁ ((+ zero) ÷suc d₁) = refl
ℚabs₁ ((+ suc n) ÷suc d₁) = refl

ℚabs₂ : (x y : ℚ) -> (| x - y | ≡ | y - x |)
ℚabs₂ x y = trans (cong |_| (P.sym (ℚ-swap x y) ))(ℚabs₁ (y - x))
```

Our proof that the relation $\simeq$ on real numbers is symmetric can now simply be given by substituting `| ℝ.f x n - ℝ.f y n |` for `| ℝ.f y n - ℝ.f x n |`. x≃y below is a variable which has x $\simeq$ y as type, which given a natural number n (as implicit argument) it will give the relation `| ℝ.f x n - ℝ.f y n |` $\leq$ `(suc n)`$^{-1}$ `ℚ.+ (suc n)`$^{-1}$.

```
--symmetry
sym≃ : {x y : ℝ} -> (x ≃ y -> y ≃ x)
sym≃ {x}{y} x≃y = λ {n} ->
  subst (λ a -> a ≤ (suc n)⁻¹ ℚ.+ (suc n)⁻¹)
  (ℚabs₂ (ℝ.f x n) (ℝ.f y n)) (x≃y {n})
```

Showing transitivity is a bit more of a challenge. Luckily, Bishop gives us a hint.

> (2.3) **Lemma.** The real numbers $x \equiv (x_n)$ and $y \equiv (y_n)$ are equal if and only if for each positive integer j there exists a positive integer N_j such that
>
> $$|x_n - y_n| \leq j^{-1} \quad (n \geq N_j)$$

(Bishop and Bridges 1985)
I have not yet proved the formal version of this lemma, but the definition is as follows:

```
postulate Bishopslem : {x y : ℝ} ->
           ({j : ℕ} -> ∃ λ N -> ({m : ℕ} ->
           | ℝ.f x (N ℕ.+ m) - ℝ.f y (N ℕ.+ m) | ≤ (suc j)⁻¹))
           -> (x ≃ y)
```

Note that here we write the all the natural numbers larger than N_j in as $N_j + m$, for any natural number m, instead of as n in Bishop's version. Using this lemma we can make a strategy of how to prove transitivity. The idea is this:
We can use the triangle inequality to show that the relation

```
| ℝ.f x n - ℝ.f z n | ≤
| ℝ.f x n - ℝ.f y n | + | ℝ.f y n - ℝ.f z n |
```

is true for all natural numbers n. Then, if we know that
`| ℝ.f x n - ℝ.f y n | ≤ (suc n)⁻¹ + (suc n)⁻¹` and
`| ℝ.f y n - ℝ.f z n | ≤ (suc n)⁻¹ + (suc n)⁻¹` for all natural numbers n, we can choose a N_j given any j such that the relation
`(suc n)⁻¹ + (suc n)⁻¹ + (suc n)⁻¹ + (suc n)⁻¹ ≤ (suc j)⁻¹`
is true for all $n \geq N_j$. We see that if we chose $N_j = 4j + 4$ we see that

$$\frac{1}{4j+4} + \frac{1}{4j+4} + \frac{1}{4j+4} + \frac{1}{4j+4} = \frac{4}{4j+4} = \frac{1}{j+1}$$

and that the relation will hold for all n greater than N_j To formalize this proof we will need a couple of additional lemmas. First, we need to show that addition with rational numbers preserves the previously defined equivalence relation on rationals. This is the the function named +-exist below. It looks quite substantial, but the proof mostly involves changing the order in which things are multiplied.

```
--Since the we have defined rationals without requiring coprimality,
--our equivalence relation ≈ is not synonymous with ≡ and therefore
```

```
--we cannot use subst or cong to modify expressions.
--Instead, we have to show that every function defined on rationals
--preserves the equality relation.
+-exist :  _+_ Preserves₂ _≃_ ⟶ _≃_ ⟶ _≃_
+-exist {p}{q}{x}{y} pq xy =  begin
  (pn ℤ.* xd ℤ.+ xn ℤ.* pd) ℤ.* (qd ℤ.* yd)
  ≡⟨ proj₂ ℤdistrib (qd ℤ.* yd) (pn ℤ.* xd) (xn ℤ.* pd)   ⟩
  pn ℤ.* xd ℤ.* (qd ℤ.* yd) ℤ.+ xn ℤ.* pd ℤ.* (qd ℤ.* yd)
  ≡⟨ cong₂ ℤ._+_ (ℤ*-assoc pn xd (qd ℤ.* yd)) (ℤ*-assoc xn pd (qd ℤ.* yd)) ⟩
  pn ℤ.* (xd ℤ.* (qd ℤ.* yd)) ℤ.+ xn ℤ.* (pd ℤ.* (qd ℤ.* yd))
  ≡⟨ cong₂ (λ a b -> pn ℤ.* a ℤ.+ xn ℤ.* (pd ℤ.* b))
    (P.sym (ℤ*-assoc xd qd yd)) (ℤ*-comm qd yd) ⟩
  pn ℤ.* (xd ℤ.* qd ℤ.* yd) ℤ.+ xn ℤ.* (pd ℤ.* (yd ℤ.* qd))
  ≡⟨ cong₂ (λ a b -> pn ℤ.* (a ℤ.* yd) ℤ.+ xn ℤ.* b)
    (ℤ*-comm xd qd) (P.sym (ℤ*-assoc pd yd qd)) ⟩
  pn ℤ.* (qd ℤ.* xd ℤ.* yd) ℤ.+ xn ℤ.* (pd ℤ.* yd ℤ.* qd)
  ≡⟨ cong₂ (λ a b -> pn ℤ.* a ℤ.+ xn ℤ.* (b ℤ.* qd))
    (ℤ*-assoc qd xd yd) (ℤ*-comm pd yd) ⟩
  pn ℤ.* (qd ℤ.* (xd ℤ.* yd)) ℤ.+ xn ℤ.* (yd ℤ.* pd ℤ.* qd)
  ≡⟨ cong₂ (λ a b -> a ℤ.+ xn ℤ.* b)
    (P.sym (ℤ*-assoc pn qd (xd ℤ.* yd))) (ℤ*-assoc yd pd qd) ⟩
  pn ℤ.* qd ℤ.* (xd ℤ.* yd) ℤ.+ xn ℤ.* (yd ℤ.* (pd ℤ.* qd))
  ≡⟨ cong₂ (λ a b -> a ℤ.* (xd ℤ.* yd) ℤ.+ b) pq
    (P.sym (ℤ*-assoc xn yd (pd ℤ.* qd))) ⟩
  qn ℤ.* pd ℤ.* (xd ℤ.* yd) ℤ.+ xn ℤ.* yd ℤ.* (pd ℤ.* qd)
  ≡⟨ cong₂ (λ a b -> a ℤ.+ b ℤ.* (pd ℤ.* qd))
    (ℤ*-assoc qn pd (xd ℤ.* yd)) xy ⟩
  qn ℤ.* (pd ℤ.* (xd ℤ.* yd)) ℤ.+ yn ℤ.* xd ℤ.* (pd ℤ.* qd)
  ≡⟨ cong₂ (λ a b -> qn ℤ.* (pd ℤ.* a) ℤ.+ yn ℤ.* xd ℤ.* b)
    (ℤ*-comm xd yd) (ℤ*-comm pd qd) ⟩
  qn ℤ.* (pd ℤ.* (yd ℤ.* xd)) ℤ.+ yn ℤ.* xd ℤ.* (qd ℤ.* pd)
  ≡⟨ cong₂ (λ a b -> qn ℤ.* a ℤ.+ b)
    (P.sym (ℤ*-assoc pd yd xd )) (ℤ*-assoc yn xd (qd ℤ.* pd)) ⟩
  qn ℤ.* (pd ℤ.* yd ℤ.* xd) ℤ.+ yn ℤ.* (xd ℤ.* (qd ℤ.* pd))
  ≡⟨ cong₂ (λ a b -> qn ℤ.* (a ℤ.* xd) ℤ.+ yn ℤ.* b)
    (ℤ*-comm pd yd) (P.sym (ℤ*-assoc xd qd pd )) ⟩
  qn ℤ.* (yd ℤ.* pd ℤ.* xd) ℤ.+ yn ℤ.* (xd ℤ.* qd ℤ.* pd)
  ≡⟨ cong₂ (λ a b -> qn ℤ.* a ℤ.+ yn ℤ.* (b ℤ.* pd))
    (ℤ*-assoc yd pd xd) (ℤ*-comm xd qd) ⟩
  qn ℤ.* (yd ℤ.* (pd ℤ.* xd)) ℤ.+ yn ℤ.* (qd ℤ.* xd ℤ.* pd)
  ≡⟨ cong₂ (λ a b -> a ℤ.+ yn ℤ.* b)
    (P.sym (ℤ*-assoc qn yd (pd ℤ.* xd))) (ℤ*-assoc qd xd pd) ⟩
  qn ℤ.* yd ℤ.* (pd ℤ.* xd) ℤ.+ yn ℤ.* (qd ℤ.* (xd ℤ.* pd))
```

```
  ≡⟨ cong (λ a -> qn ℤ.* yd ℤ.* (pd ℤ.* xd) ℤ.+ yn ℤ.* (qd ℤ.* a))
    (ℤ*-comm xd pd) ⟩
  qn ℤ.* yd ℤ.* (pd ℤ.* xd) ℤ.+ yn ℤ.* (qd ℤ.* (pd ℤ.* xd))
  ≡⟨ cong (λ a -> qn ℤ.* yd ℤ.* (pd ℤ.* xd) ℤ.+ a)
    (P.sym (ℤ*-assoc yn qd (pd ℤ.* xd))) ⟩
  qn ℤ.* yd ℤ.* (pd ℤ.* xd) ℤ.+ yn ℤ.* qd ℤ.* (pd ℤ.* xd)
  ≡⟨ P.sym (proj₂ ℤdistrib (pd ℤ.* xd) (qn ℤ.* yd) (yn ℤ.* qd)) ⟩
  (qn ℤ.* yd ℤ.+ yn ℤ.* qd) ℤ.* (pd ℤ.* xd)
        ■
        where
          open P.≡-Reasoning
          pn = ℚ.numerator p
          pd = ℚ.denominator p
          qn = ℚ.numerator q
          qd = ℚ.denominator q
          xn = ℚ.numerator x
          xd = ℚ.denominator x
          yn = ℚ.numerator y
          yd = ℚ.denominator y
```

We also want to be able to reduce the sum of four rationals $\frac{1}{4j+j}+\frac{1}{4j+j}+\frac{1}{4j+j}+\frac{1}{4j+j} = \frac{1}{j+1}$ as described in the proof idea above. For that we need the following lemmas.

```
+-red₁ : (n : ℕ) ->
  ((+ 1) ÷suc (suc (n ℕ.+ n)) ℚ.+
  (+ 1) ÷suc (suc (n ℕ.+ n)) ℚ.≃ (+ 1) ÷suc n)
+-red₁ n = begin
  ((+ 1) ℤ.* k ℤ.+ (+ 1) ℤ.* k) ℤ.* + suc n
  ≡⟨ cong (λ a -> ((a ℤ.+ a) ℤ.* + suc n)) (proj₁ ℤ*-identity k) ⟩
  (k ℤ.+ k) ℤ.* + suc n
  ≡⟨ cong (λ a -> a ℤ.* + suc n) (P.sym (lem k)) ⟩
  (+ 2) ℤ.* k ℤ.* (+ suc n)
  ≡⟨ cong (λ a -> a ℤ.* + suc n) (ℤ*-comm (+ 2) k) ⟩
  k ℤ.* (+ 2) ℤ.* (+ suc n)
  ≡⟨ ℤ*-assoc k (+ 2) (+ suc n) ⟩
  k ℤ.* ((+ 2) ℤ.* (+ suc n))
  ≡⟨ cong (λ a -> k ℤ.* a) (lem (+ suc n)) ⟩
  k ℤ.* (+ suc (n ℕ.+ suc n))
  ≡⟨ cong (λ a -> k ℤ.* + suc a) (+-comm n (suc n)) ⟩
  k ℤ.* k ≡⟨ P.sym (proj₁ ℤ*-identity (k ℤ.* k)) ⟩
  (+ 1) ℤ.* (k ℤ.* k)
  ■
    where
      open P.≡-Reasoning
```

```
      k = + suc (suc (n ℕ.+ n))
      lem : (j : ℤ) -> ((+ 2) ℤ.* j ≡ j ℤ.+ j)
      lem j = trans (proj₂ ℤdistrib j (+ 1) (+ 1))
        (cong₂ ℤ._+_ (proj₁ ℤ*-identity j) (proj₁ ℤ*-identity j))

+-red₂ : (n : ℕ) ->
       (((+ 1) ÷suc (suc ((suc (n ℕ.+ n)) ℕ.+ (suc (n ℕ.+ n)))) +
       (+ 1) ÷suc (suc ((suc (n ℕ.+ n)) ℕ.+ (suc (n ℕ.+ n))))))
      +
      ((+ 1) ÷suc (suc ((suc (n ℕ.+ n)) ℕ.+ (suc (n ℕ.+ n)))) +
      (+ 1) ÷suc (suc ((suc (n ℕ.+ n)) ℕ.+ (suc (n ℕ.+ n)))))
      ℚ.≃ ((+ 1) ÷suc n))
+-red₂ n = IsEquivalence.trans ℚ.isEquivalence {start} {middle} {end}
  (+-exist {1÷k + 1÷k}{1÷j}{1÷k + 1÷k}{1÷j} (+-red₁ j) (+-red₁ j))
    (+-red₁ n)
  where
    j = suc (n ℕ.+ n)
    k = suc (j ℕ.+ j)
    1÷j = (+ 1) ÷suc j
    1÷k = (+ 1) ÷suc k
    start = (1÷k + 1÷k) + (1÷k + 1÷k)
    middle = 1÷j + 1÷j
    end = (+ 1) ÷suc n
```

We also need a minor lemma proving that $\frac{1}{suc(m+n)} \leq \frac{1}{suc(m)}$ for all natural numbers m and n.

```
ℚ≤lem : {m n : ℕ} -> ((+ 1) ÷suc (m ℕ.+ n) ≤ (+ 1) ÷suc m)
ℚ≤lem {m}{n} =  *≤* (ℤ.+≤+ (ℕ.s≤s ((m≤m+n m n) +-mono (z≤n))))
```

The triangle inequality is not yet proven at this time, but left as a postulate.

```
postulate ℚtriang : (x y z : ℚ) -> (| x - z | ≤ | x - y | + | y - z |)
```

Just like we did for the equivalence relation on rational numbers we need to show that inequality of rational numbers is preserved under addition. The idea of this proof is similar to the one proving `+-exist`, but I have not been able to formalize it at this time.

```
postulate _ℚ+-mono_ :  _+_ Preserves₂ _≤_ ⟶ _≤_ ⟶ _≤_
```

Using these lemmas we are ready to give the proof of transitivity of the equivalence relation on real numbers:

```
  --transitivity
  trans≃ : {x y z : ℝ} -> (x ≃ y) -> (y ≃ z) -> (x ≃ z)
  trans≃ {x}{y}{z} x≃y y≃z = Bishopslem {x}{z} (λ {j} ->
```

```
    N {j} , λ {n} -> (begin
    | ℝ.f x (N {j} ℕ.+ n) - ℝ.f z (N {j} ℕ.+ n) |
      ∼⟨ ℚtriang (ℝ.f x (N {j} ℕ.+ n)) (ℝ.f y (N {j} ℕ.+ n)) (ℝ.f z (N {j} ℕ.+ n)) ⟩
    | ℝ.f x (N {j} ℕ.+ n) - ℝ.f y (N {j} ℕ.+ n) | +
    | ℝ.f y (N {j} ℕ.+ n) - ℝ.f z (N {j} ℕ.+ n) |
    ∼⟨ (x≃y {N {j} ℕ.+ n}) ℚ+-mono (y≃z {N {j} ℕ.+ n}) ⟩
    ((suc (N {j} ℕ.+ n))⁻¹ ℚ.+ (suc (N {j} ℕ.+ n))⁻¹) ℚ.+
    ((suc (N {j} ℕ.+ n))⁻¹ ℚ.+ (suc (N {j} ℕ.+ n))⁻¹)
    ∼⟨ ((ℚ≤lem {N {j}} {n}) ℚ+-mono (ℚ≤lem {N {j}} {n}))
           ℚ+-mono
         ((ℚ≤lem {N {j}} {n}) ℚ+-mono (ℚ≤lem {N {j}} {n})) ⟩
    ((suc (N {j}))⁻¹ ℚ.+ (suc (N {j}))⁻¹) ℚ.+
    ((suc (N {j}))⁻¹ ℚ.+ (suc (N {j}))⁻¹)
    ∼⟨ ≈->≤ (+-red₂ j) ⟩
    ((suc j)⁻¹ ■) ))
    where
      open DecTotalOrder ℚ.decTotalOrder using ()
        renaming (reflexive to ≈->≤; trans to ≤trans; isPreorder to ℚisPreorder)
      open Pre record {isPreorder = ℚisPreorder}
      N = λ {j} -> suc ((suc (j ℕ.+ j) ℕ.+ (suc (j ℕ.+ j))))
```

This proves that the relation defined on the real numbers indeed is an equivalence relation and we can create an instance of the type `IsEquivalence`.

```
isEquivalence : IsEquivalence _≃_
isEquivalence = record {
  refl = λ {x} -> refl≃ {x} ;
  sym = λ {x}{y} -> sym≃ {x}{y};
  trans = λ {a}{b}{c} -> trans≃ {a}{b}{c}
  }
```

3 Summary and future work

This work has established much of the framework necessary to formalize the constructive notion of a real number as described by Errett Bishop in *Constructive Analysis* (Bishop and Bridges 1985). To complete the proof that the relation we have defined on real numbers is an equivalence relation there are three major lemmas that remains to be solved:

1. The triangle inequality

   ```
   ℚtriang : (x y z : ℚ) -> (| x - z | ≤ | x - y | + | y - z |)
   ℚtriang x y z = ?
   ```

2. Bishop's lemma

```
Bishopslem : {x y : ℝ} ->
           ({j : ℕ} -> ∃ λ N -> ({m : ℕ} ->
           | ℝ.f x (N ℕ.+ m) - ℝ.f y (N ℕ.+ m) | ≤ (suc j)⁻¹))
           -> (x ≃ y)
Bishopslem {x}{y} p = ?
```

3. The fact that inequality of rational numbers is preserved over addition

```
_ℚ+-mono_ :  _+_ Preserves₂ _≤_ ⟶ _≤_ ⟶ _≤_
p₁≤q₁ ℚ+-mono p₂≤q₂ = ?
```

The third of these problems one is probably the easiest to solve, as we already showed that addition preserves the equality relation defined on rational numbers. The proof that addition preserves inequality on rationals is analogue to this proof, but we will also need to prove that addition preserves the inequality relation over integers first.

The triangle inequality can be solved informally solved like this:

Theorem 1 *For all rational numbers x and y, we have $|x - y| \leq |x| + |y|$*
Proof: Squaring the left-hand side of the inequality we get:

$$|x - y|^2 = (x - y)^2 = x^2 - 2xy + y^2 \leq |x|^2 + 2|x||y| + |y|^2 = (|x| + |y|)^2$$

By taking the square root of both sides we obtain the triangle inequality.

To show this formally, we need to show that taking a square root preserves inequality, that for all rational numbers a and b we have $|ab| = |a||b|$, that $a \leq |a|$ and that $a^2 = |a|^2$.

As for the lemma presented by Bishop, an informal proof is given in *Constructive Analysis* (Bishop and Bridges 1985).

3.1 Possible errors

The Agda type checker does not leave much room for errors to be made in the actual proving of statements, but mistakes can still occur in formal mathematics. One thing that can happen is that we define things wrong. Great care is therefore required while interpreting informal statements. If one writes the type of a proof wrong, proving it could be impossible.

In this work, I have left three theorems as unproved postulates. If it turns out that any of them are typed wrong, the proof that the equivalence relation on real numbers is transitive would no longer be correct. Such an error might be easy to fix with a minor adjustment, but could also require the proof to be completely rewritten.

4 Code

4.1 Data/Rational.agda

```
------------------------------------------------------------------------
-- The Agda standard library
--
-- Rational numbers
------------------------------------------------------------------------

module Data.Rational where

import Data.Integer.Multiplication.Properties as Mul
open import Algebra using (module CommutativeMonoid)
import Data.Sign as S
open import Data.Empty using (⊥)
open import Data.Unit using (⊤; tt)
import Data.Bool.Properties as Bool
open import Function
open import Data.Product
open import Data.Integer as ℤ using (ℤ; +_; -[1+_]; _◃_; sign)
import Data.Integer.Properties as ℤ
open import Data.Nat.GCD
open import Data.Nat.Divisibility as ℕDiv using (_∣_; divides; quotient)
open import Data.Nat as ℕ using (ℕ; zero; suc)
open import Data.Nat.Show renaming (show to ℕshow)
open import Data.Sum
open import Data.String using (String; _++_)
import Level
open import Relation.Nullary.Decidable
open import Relation.Nullary
open import Relation.Binary
open import Relation.Binary.Core using (_≢_)
open import Relation.Binary.PropositionalEquality as P using
  (_≡_; refl; subst; cong; cong₂)
open import Data.Integer.Properties using (cancel-*-right)
open P.≡-Reasoning
open CommutativeMonoid Mul.commutativeMonoid
  using ()
  renaming (assoc to *-assoc; comm to *-comm; identity to *-identity
           ; isCommutativeMonoid to *-isCommutativeMonoid
           ; isMonoid to *-isMonoid
           )

infix  8 -_ 1/_
infixl 7   _*_ _/_
```

```
infixl 6  _+_ _-_

------------------------------------------------------------------------
-- The definition

--Rational numbers
--Note that we do not require the arguments to be given in their reduced form

record ℚ : Set where
  constructor _÷suc_
  field
    numerator     : ℤ
    denominator-1 : ℕ

  denominator : ℤ
  denominator = + suc denominator-1

infixl 7 _÷_

_÷_ : (n : ℤ) (d : ℕ) -> {≠0 : False (ℕ._≟_ d 0)} -> ℚ
(n ÷ 0) {()}
n ÷ (suc d) = n ÷suc d

------------------------------------------------------------------------
--  Functions for reducing rational numbers to their coprime form

-- normalize takes two natural numbers, say 6 and 21 and their gcd 3, and
-- returns them normalized as 2 and 7
normalize : {m n g : ℕ} →
  {n≠0 : False (ℕ._≟_ n 0)} → {g≠0 : False (ℕ._≟_ g 0)} →
            GCD m n g → ℚ
normalize {m} {n} {0} {_} {()} _
normalize {m} {.0} {ℕ.suc g} {()} {_}
  (GCD.is (divides p m≡pg' , divides 0 refl) _)
normalize {m} {n} {ℕ.suc g} {_} {_}
  (GCD.is (divides p m≡pg' , divides (suc q) n≡qg') _) =
    ((+ p) ÷suc q)

--gcd that gives a proof that g is NonZero if one of its inputs are NonZero
gcd≠0 : (m n : ℕ) → {n≠0 : False (ℕ._≟_ n 0)} → ∃ λ d →
  GCD m n d × (False (ℕ._≟_ d 0))
gcd≠0 m n {m≠0} with gcd m n
gcd≠0 m n {m≠0} | (0 , GCD.is (_ , 0n) _) with ℕDiv.0|⇒≡0 0n
```

```
gcd≢0 m .0 {()} | (0 , GCD.is (_ , 0n) _) | refl
gcd≢0 m n {_} | (ℕ.suc d , G) = (ℕ.suc d , G , tt)

--Unary negation
-_ : ℚ → ℚ
- n ÷suc d = (ℤ.- n) ÷suc d

--Reduces a given rational number to its coprime form
reduce : ℚ -> ℚ
reduce ((+ 0) ÷suc d) = (+ 0 ÷ 1)
reduce (-[1+ n ] ÷suc d) =
  - normalize {ℤ.| -[1+ n ] |} {suc d} {proj₁ (gcd≢0 (suc n) (suc d) {_})} {_}
  {proj₂ (proj₂ (gcd≢0 (suc n) (suc d) {_}))}
  (proj₁( proj₂ (gcd≢0 (suc n) (suc d) {_})))
reduce ((+ n) ÷suc d) =
  normalize {ℤ.| + n |} {suc d} {proj₁ (gcd≢0 n (suc d) {_})} {_}
  {proj₂ (proj₂ (gcd≢0 n (suc d) {_}))}
  (proj₁ (proj₂ (gcd≢0 n (suc d) {_})))

------------------------------------------------------------------------------
-- Operations on rationals: reciprocal, multiplication, addition

-- reciprocal: requires a proof that the numerator is not zero

1/_ : (p : ℚ) → {≢0 : False (ℕ._≟_ (ℤ.| ℚ.numerator p |) 0)} → ℚ
1/_ ((+ 0) ÷suc d) {()}
1/_ ((+ suc n) ÷suc d) = (+ suc d) ÷suc n
1/_ (-[1+ n ] ÷suc d) = -[1+ d ] ÷suc n

--Multiplication and addition

_*_ : ℚ -> ℚ -> ℚ
(n₁ ÷suc d₁) * (n₂ ÷suc d₂) = ((n₁ ℤ.* n₂) ÷ (suc d₁ ℕ.* suc d₂))

_+_ :  ℚ -> ℚ -> ℚ
(n₁ ÷suc d₁) + (n₂ ÷suc d₂) =  ((n₁ ℤ.* + (suc d₂)) ℤ.+ (n₂ ℤ.* + (suc  d₁)))
  ÷ ((suc d₁) ℕ.* (suc d₂))

 -- subtraction and division

_-_ : ℚ → ℚ → ℚ
p₁ - p₂ = p₁ + (- p₂)
```

```
_/_ : (p₁ p₂ : ℚ) → {≢0 : False (ℕ._≟_ ℤ.| ℚ.numerator p₂ | 0)} → ℚ
_/_ p₁ p₂ {≢0} = p₁ * (1/_ p₂ {≢0})


--absolute value
|_| : ℚ -> ℚ
| n ÷suc d | = (+ ℤ.| n |) ÷suc d

-- conventional printed representation

show : ℚ → String
show p = ℤ.show (ℚ.numerator p) ++ "/" ++ ℕshow (ℕ.suc (ℚ.denominator-1 p))

------------------------------------------------------------------------
-- Equality

-- We define an equality relation on rational numbers in the conventional way

infix 4 _≃_

_≃_ : Rel ℚ Level.zero
p ≃ q = numerator p ℤ.* (+ suc (denominator-1 q)) ≡
        numerator q ℤ.* (+ suc (denominator-1 p))
  where open ℚ

--This is an equivalence relation
isEquivalence : IsEquivalence _≃_
isEquivalence = record {
  refl = refl ;
  sym = P.sym ;
  trans = λ {a}{b}{c} -> trans {a}{b}{c}
  }
    where
    trans : Transitive _≃_
    trans {a ÷suc b} {f ÷suc g} {x ÷suc y} ag≃fb fy≃xg =
      cancel-*-right (a ℤ.* (+ suc y)) (x ℤ.* (+ suc b)) (+ suc g) (λ {()})
        (P.trans ayg≃fby fby≃xbg)
      where
        agy≃fby : (a ℤ.* + suc g ℤ.* + suc y ≡ f ℤ.* + suc b ℤ.* + suc y)
        agy≃fby = cong (λ j -> (j ℤ.* + suc y)) (ag≃fb)
        ayg≃fby : (a ℤ.* + suc y ℤ.* + suc g ≡ f ℤ.* + suc b ℤ.* + suc y)
        ayg≃fby = P.trans (*-assoc a (+ suc y) (+ suc g))
          (P.trans (cong (λ j -> (a ℤ.* j )) (*-comm (+ suc y) (+ suc g)))
```

```
          (P.trans (P.sym (*-assoc a (+ suc g) (+ suc y))) agy≃fby))
        fyb≃xgb : (f ℤ.* + suc y ℤ.* + suc b ≡ x ℤ.* + suc g ℤ.* + suc b)
        fyb≃xgb = cong (λ j -> j ℤ.* (+ suc b)) fy≃xg
        fby≃xgb : (f ℤ.* + suc b ℤ.* + suc y ≡ x ℤ.* + suc g ℤ.* + suc b)
        fby≃xgb = P.trans (*-assoc f (+ suc b) (+ suc y))
          (P.trans (cong (λ j -> (f ℤ.* j )) (*-comm (+ suc b) (+ suc y)))
            (P.trans (P.sym (*-assoc f (+ suc y) (+ suc b))) fyb≃xgb))
        fby≃xbg : (f ℤ.* + suc b ℤ.* + suc y ≡ x ℤ.* + suc b ℤ.* + suc g)
        fby≃xbg = P.trans (P.trans (fby≃xgb) (*-assoc x (+ suc g) (+ suc b)))
          (P.trans (cong (λ j -> (x ℤ.* j)) (*-comm (+ suc g) (+ suc b)))
            (P.sym (*-assoc x (+ suc b) (+ suc g))))


------------------------------------------------------------------------
--Equality is decidable

infix 4 _≟_

_≟_ : Decidable {A = ℚ} _≃_
p ≟ q with ℚ.numerator p ℤ.* (+ suc (ℚ.denominator-1 q)) ℤ.≟
           ℚ.numerator q ℤ.* (+ suc (ℚ.denominator-1 p))
p ≟ q | yes pq≃qp = yes (pq≃qp)
p ≟ q | no ¬pq≃qp = no (¬pq≃qp)

------------------------------------------------------------------------
 -- Ordering

infix 4 _≤_ _≤?_

data _≤_ : ℚ → ℚ → Set where
  *≤* : ∀ {p q} →
        ℚ.numerator p ℤ.* (+ suc (ℚ.denominator-1 q)) ℤ.≤
        ℚ.numerator q ℤ.* (+ suc (ℚ.denominator-1 p)) →
        p ≤ q

drop-*≤* : ∀ {p q} → p ≤ q →
           ℚ.numerator p ℤ.* (+ suc (ℚ.denominator-1 q)) ℤ.≤
           ℚ.numerator q ℤ.* (+ suc (ℚ.denominator-1 p))
drop-*≤* (*≤* pq≤qp) = pq≤qp

_≤?_ : Decidable _≤_
p ≤? q with ℚ.numerator p ℤ.* (+ suc (ℚ.denominator-1 q)) ℤ.≤?
           ℚ.numerator q ℤ.* (+ suc (ℚ.denominator-1 p))
```

```
p ≤? q | yes pq≤qp = yes (*≤* pq≤qp)
p ≤? q | no ¬pq≤qp = no (λ { (*≤* pq≤qp) → ¬pq≤qp pq≤qp })

decTotalOrder : DecTotalOrder _ _ _
decTotalOrder = record
  { Carrier         = ℚ
  ; _≈_             = _≃_
  ; _≤_             = _≤_
  ; isDecTotalOrder = record
      { isTotalOrder = record
          { isPartialOrder = record
              { isPreorder = record
                  { isEquivalence = isEquivalence
                  ; reflexive     = refl′
                  ; trans         = trans
                  }
              ; antisym = antisym
              }
          ; total = total
          }
      ; _≟_  = _≟_
      ; _≤?_ = _≤?_
      }
  }
  where
  module ℤO = DecTotalOrder ℤ.decTotalOrder

  refl′ : _≃_ ⇒ _≤_
  refl′ p = *≤* (reflexive p)
    where
      open DecTotalOrder ℤ.decTotalOrder using (reflexive)

  trans : Transitive _≤_
  trans {i = p} {j = q} {k = r} (*≤* le₁) (*≤* le₂)
    = *≤* (ℤ.cancel-*-+-right-≤ _ _ _
            (lemma
              (ℚ.numerator p) ((+ suc (ℚ.denominator-1 p)))
              (ℚ.numerator q) ((+ suc (ℚ.denominator-1 q)))
              (ℚ.numerator r) ((+ suc (ℚ.denominator-1 r)))
              (ℤ.*-+-right-mono (ℚ.denominator-1 r) le₁)
              (ℤ.*-+-right-mono (ℚ.denominator-1 p) le₂)))
    where
```

```
    lemma : ∀ n₁ d₁ n₂ d₂ n₃ d₃ →
            n₁ ℤ.* d₂ ℤ.* d₃ ℤ.≤ n₂ ℤ.* d₁ ℤ.* d₃ →
            n₂ ℤ.* d₃ ℤ.* d₁ ℤ.≤ n₃ ℤ.* d₂ ℤ.* d₁ →
            n₁ ℤ.* d₃ ℤ.* d₂ ℤ.≤ n₃ ℤ.* d₁ ℤ.* d₂
    lemma n₁ d₁ n₂ d₂ n₃ d₃
      rewrite *-assoc n₁ d₂ d₃
            | *-comm d₂ d₃
            | P.sym (*-assoc n₁ d₃ d₂)
            | *-assoc n₃ d₂ d₁
            | *-comm d₂ d₁
            | P.sym (*-assoc n₃ d₁ d₂)
            | *-assoc n₂ d₁ d₃
            | *-comm d₁ d₃
            | P.sym (*-assoc n₂ d₃ d₁)
            = ℤO.trans

  antisym : Antisymmetric _≃_ _≤_
  antisym (*≤* le₁) (*≤* le₂) = (ℤO.antisym le₁ le₂)

  total : Total _≤_
  total p q =
    [ inj₁ ∘′ *≤* , inj₂ ∘′ *≤* ]′
      (ℤO.total (ℚ.numerator p ℤ.* (+ suc (ℚ.denominator-1 q)))
                (ℚ.numerator q ℤ.* (+ suc (ℚ.denominator-1 p))))
```

4.2 Data/Rational/Properties.agda

```
module Data.Rational.Properties where

open import Data.Sum
open import Relation.Nullary.Decidable
open import Data.Rational as ℚ using (ℚ; -_ ; _*_; _÷suc_; _-_; _+_; |_|;
  decTotalOrder; _≤_; *≤* ; _≤?_; _÷_; _≃_)
open import Data.Integer as ℤ using (ℤ; +_ ; -[1+_]) renaming
  (_-_ to ℤ_-_; _+_ to _ℤ+_; _*_ to  _ℤ*_;_≤_ to ℤ_≤_)
open import Data.Nat as ℕ using (ℕ; suc; zero; z≤n; pred) renaming (_≤_ to ℕ_≤_)
open import Data.Nat.Properties.Simple using (+-comm; +-suc; +-right-identity;
  *-comm)
open import Relation.Binary.Core using (_Preserves₂_⟶_⟶_; IsEquivalence)
open import Data.Nat.Properties using (m≤m+n; _+-mono_)
import Relation.Binary.PreorderReasoning as Pre
open import Relation.Binary.PropositionalEquality.Core using (trans; subst)
open import Algebra using (module CommutativeRing)
open import Data.Integer.Properties using (commutativeRing; abs-◃)
```

```
open import Relation.Binary.PropositionalEquality as P using (_≡_; refl;
  subst; cong; cong₂)
open import Data.Product
open CommutativeRing commutativeRing
  using ()
  renaming (distrib to ℤdistrib; +-assoc to ℤ+-assoc; *-assoc to ℤ*-assoc;
    *-comm to ℤ*-comm; +-comm to ℤ+-comm; *-identity to ℤ*-identity)


--This module contains various lemmas and additional functions rational numbers

_⁻¹ : (n : ℕ) -> {≢0 : False (ℕ._≟_ n 0)} -> ℚ
(n ⁻¹) {≢0} = ((+ 1) ÷ n) {≢0}

--Lemmas helping to show symmetry of the equivalence relation
--defined on the real numbers
⊖-swap : ∀ a b → a ℤ.⊖ b ≡ ℤ.- (b ℤ.⊖ a)
⊖-swap zero    zero    = refl
⊖-swap (suc _) zero    = refl
⊖-swap zero    (suc _) = refl
⊖-swap (suc a) (suc b) = ⊖-swap a b

ℤ-swap : (a b : ℤ) -> (ℤ.- (a ℤ.- b) ≡ b ℤ.- a)
ℤ-swap -[1+ n ] -[1+ n₁ ] = P.sym (⊖-swap n n₁)
ℤ-swap -[1+ n ] (+ zero) = refl
ℤ-swap -[1+ n ] (+ suc n₁) = trans (cong (λ a -> + suc (suc a))
  (+-comm n n₁)) (cong ℤ.+_ (P.sym (+-suc (suc n₁) n)))
ℤ-swap (+ zero) -[1+ n₁ ] = refl
ℤ-swap (+ suc n) -[1+ n₁ ] = cong -[1+_] (+-comm n (suc n₁))
ℤ-swap (+ zero) (+ zero) = refl
ℤ-swap (+ zero) (+ suc n₁) = cong ℤ.+_ (P.sym (+-right-identity (suc n₁)))
ℤ-swap (+ suc n) (+ zero) = cong -[1+_] (+-right-identity n)
ℤ-swap (+ suc n) (+ suc n₁) = P.sym (⊖-swap n₁ n)

ℚ-swap : (x y : ℚ) -> (- (y - x) ≡ x - y)
ℚ-swap (-[1+ n₁ ] ÷suc d₁) (-[1+ n₂ ] ÷suc d₂) =
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (-[1+ n₂ ] ℤ.* + suc d₁) (-[1+ n₁ ] ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap (-[1+ n₁ ] ÷suc d₁) ((+ zero) ÷suc d₂) =
  cong (λ a -> (-[1+ n₁ ] ℤ.* (+ suc d₂)) ℚ.÷suc (pred a))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap (-[1+ n₁ ] ÷suc d₁) ((+ suc n₂) ÷suc d₂) =
```

```
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (+ suc n₂ ℤ.* + suc d₁) (-[1+ n₁ ] ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap ((+ zero) ÷suc d₁) (-[1+ n₂ ] ÷suc d₂) =
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (-[1+ n₂ ] ℤ.* + suc d₁) (+ zero ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap ((+ suc n₁) ÷suc d₁) (-[1+ n₂ ] ÷suc d₂) =
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (-[1+ n₂ ] ℤ.* + suc d₁) (+ suc n₁ ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap ((+ zero) ÷suc d₁) ((+ zero) ÷suc d₂) =
  cong (λ a -> ((+ zero) ÷suc (pred a)))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap ((+ zero) ÷suc d₁) ((+ suc n₂) ÷suc d₂) =
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (+ suc n₂ ℤ.* + suc d₁) (+ zero ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap ((+ suc n) ÷suc d₁) ((+ zero) ÷suc d₂) =
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (+ zero ℤ.* + suc d₁) (+ suc n ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))
ℚ-swap ((+ suc n₁) ÷suc d₁) ((+ suc n₂) ÷suc d₂) =
  cong₂ (λ a b -> (a ÷suc (pred b)))
  (ℤ-swap (+ suc n₂ ℤ.* + suc d₁) (+ suc n₁ ℤ.* + suc d₂))
  (*-comm (suc d₂) (suc d₁))

ℚabs₁ : (x : ℚ) -> (| ℚ.- x | ≡ | x |)
ℚabs₁ (-[1+ n ] ÷suc d₁) = refl
ℚabs₁ ((+ zero) ÷suc d₁) = refl
ℚabs₁ ((+ suc n) ÷suc d₁) = refl

ℚabs₂ : (x y : ℚ) -> (| x - y | ≡ | y - x |)
ℚabs₂ x y = trans (cong |_| (P.sym (ℚ-swap x y) ))(ℚabs₁ (y - x))


--Since the we have defined rationals without requireming coprimality,
--our equivalence relation ≈ is not synonymous with ≡ and therefore
--we cannot use subst or cong to modify expressions.
--Instead, we have to show that every function defined on rationals
--preserves the equality relation.
+-exist :  _+_ Preserves₂ _≃_ ⟶ _≃_ ⟶ _≃_
+-exist {p}{q}{x}{y} pq xy =  begin
```

```
(pn ℤ.* xd ℤ.+ xn ℤ.* pd) ℤ.* (qd ℤ.* yd)
≡⟨ proj₂ ℤdistrib (qd ℤ.* yd) (pn ℤ.* xd) (xn ℤ.* pd)   ⟩
pn ℤ.* xd ℤ.* (qd ℤ.* yd) ℤ.+ xn ℤ.* pd ℤ.* (qd ℤ.* yd)
≡⟨ cong₂ ℤ._+_ (ℤ*-assoc pn xd (qd ℤ.* yd)) (ℤ*-assoc xn pd (qd ℤ.* yd)) ⟩
pn ℤ.* (xd ℤ.* (qd ℤ.* yd)) ℤ.+ xn ℤ.* (pd ℤ.* (qd ℤ.* yd))
≡⟨ cong₂ (λ a b -> pn ℤ.* a ℤ.+ xn ℤ.* (pd ℤ.* b))
  (P.sym (ℤ*-assoc xd qd yd)) (ℤ*-comm qd yd) ⟩
pn ℤ.* (xd ℤ.* qd ℤ.* yd) ℤ.+ xn ℤ.* (pd ℤ.* (yd ℤ.* qd))
≡⟨ cong₂ (λ a b -> pn ℤ.* (a ℤ.* yd) ℤ.+ xn ℤ.* b)
  (ℤ*-comm xd qd) (P.sym (ℤ*-assoc pd yd qd)) ⟩
pn ℤ.* (qd ℤ.* xd ℤ.* yd) ℤ.+ xn ℤ.* (pd ℤ.* yd ℤ.* qd)
≡⟨ cong₂ (λ a b -> pn ℤ.* a ℤ.+ xn ℤ.* (b ℤ.* qd))
  (ℤ*-assoc qd xd yd) (ℤ*-comm pd yd) ⟩
pn ℤ.* (qd ℤ.* (xd ℤ.* yd)) ℤ.+ xn ℤ.* (yd ℤ.* pd ℤ.* qd)
≡⟨ cong₂ (λ a b -> a ℤ.+ xn ℤ.* b)
  (P.sym (ℤ*-assoc pn qd (xd ℤ.* yd))) (ℤ*-assoc yd pd qd) ⟩
pn ℤ.* qd ℤ.* (xd ℤ.* yd) ℤ.+ xn ℤ.* (yd ℤ.* (pd ℤ.* qd))
≡⟨ cong₂ (λ a b -> a ℤ.* (xd ℤ.* yd) ℤ.+ b) pq
  (P.sym (ℤ*-assoc xn yd (pd ℤ.* qd))) ⟩
qn ℤ.* pd ℤ.* (xd ℤ.* yd) ℤ.+ xn ℤ.* yd ℤ.* (pd ℤ.* qd)
≡⟨ cong₂ (λ a b -> a ℤ.+ b ℤ.* (pd ℤ.* qd))
  (ℤ*-assoc qn pd (xd ℤ.* yd)) xy ⟩
qn ℤ.* (pd ℤ.* (xd ℤ.* yd)) ℤ.+ yn ℤ.* xd ℤ.* (pd ℤ.* qd)
≡⟨ cong₂ (λ a b -> qn ℤ.* (pd ℤ.* a) ℤ.+ yn ℤ.* xd ℤ.* b)
  (ℤ*-comm xd yd) (ℤ*-comm pd qd) ⟩
qn ℤ.* (pd ℤ.* (yd ℤ.* xd)) ℤ.+ yn ℤ.* xd ℤ.* (qd ℤ.* pd)
≡⟨ cong₂ (λ a b -> qn ℤ.* a ℤ.+ b)
  (P.sym (ℤ*-assoc pd yd xd )) (ℤ*-assoc yn xd (qd ℤ.* pd)) ⟩
qn ℤ.* (pd ℤ.* yd ℤ.* xd) ℤ.+ yn ℤ.* (xd ℤ.* (qd ℤ.* pd))
≡⟨ cong₂ (λ a b -> qn ℤ.* (a ℤ.* xd) ℤ.+ yn ℤ.* b)
  (ℤ*-comm pd yd) (P.sym (ℤ*-assoc xd qd pd )) ⟩
qn ℤ.* (yd ℤ.* pd ℤ.* xd) ℤ.+ yn ℤ.* (xd ℤ.* qd ℤ.* pd)
≡⟨ cong₂ (λ a b -> qn ℤ.* a ℤ.+ yn ℤ.* (b ℤ.* pd))
  (ℤ*-assoc yd pd xd) (ℤ*-comm xd qd) ⟩
qn ℤ.* (yd ℤ.* (pd ℤ.* xd)) ℤ.+ yn ℤ.* (qd ℤ.* xd ℤ.* pd)
≡⟨ cong₂ (λ a b -> a ℤ.+ yn ℤ.* b)
  (P.sym (ℤ*-assoc qn yd (pd ℤ.* xd))) (ℤ*-assoc qd xd pd) ⟩
qn ℤ.* yd ℤ.* (pd ℤ.* xd) ℤ.+ yn ℤ.* (qd ℤ.* (xd ℤ.* pd))
≡⟨ cong (λ a -> qn ℤ.* yd ℤ.* (pd ℤ.* xd) ℤ.+ yn ℤ.* (qd ℤ.* a))
  (ℤ*-comm xd pd) ⟩
qn ℤ.* yd ℤ.* (pd ℤ.* xd) ℤ.+ yn ℤ.* (qd ℤ.* (pd ℤ.* xd))
≡⟨ cong (λ a -> qn ℤ.* yd ℤ.* (pd ℤ.* xd) ℤ.+ a)
  (P.sym (ℤ*-assoc yn qd (pd ℤ.* xd))) ⟩
```

```
  qn ℤ.* yd ℤ.* (pd ℤ.* xd) ℤ.+ yn ℤ.* qd ℤ.* (pd ℤ.* xd)
  ≡⟨ P.sym (proj₂ ℤdistrib (pd ℤ.* xd) (qn ℤ.* yd) (yn ℤ.* qd)) ⟩
  (qn ℤ.* yd ℤ.+ yn ℤ.* qd) ℤ.* (pd ℤ.* xd)
        ■
        where
          open P.≡-Reasoning
          pn = ℚ.numerator p
          pd = ℚ.denominator p
          qn = ℚ.numerator q
          qd = ℚ.denominator q
          xn = ℚ.numerator x
          xd = ℚ.denominator x
          yn = ℚ.numerator y
          yd = ℚ.denominator y

+-red₁ : (n : ℕ) ->
  ((+ 1) ÷suc (suc (n ℕ.+ n)) ℚ.+ (+ 1) ÷suc (suc (n ℕ.+ n)) ℚ.≃ (+ 1) ÷suc n)
+-red₁ n = begin
  ((+ 1) ℤ.* k ℤ.+ (+ 1) ℤ.* k) ℤ.* + suc n ≡⟨ cong (λ a -> ((a ℤ.+ a) ℤ.* + suc n))
    (proj₁ ℤ*-identity k) ⟩
  (k ℤ.+ k) ℤ.* + suc n ≡⟨ cong (λ a -> a ℤ.* + suc n) (P.sym (lem k)) ⟩
  (+ 2) ℤ.* k ℤ.* (+ suc n) ≡⟨ cong (λ a -> a ℤ.* + suc n) (ℤ*-comm (+ 2) k) ⟩
  k ℤ.* (+ 2) ℤ.* (+ suc n) ≡⟨ ℤ*-assoc k (+ 2) (+ suc n) ⟩
  k ℤ.* ((+ 2) ℤ.* (+ suc n)) ≡⟨ cong (λ a -> k ℤ.* a) (lem (+ suc n)) ⟩
  k ℤ.* (+ suc (n ℕ.+ suc n)) ≡⟨ cong (λ a -> k ℤ.* + suc a) (+-comm n (suc n)) ⟩
  k ℤ.* k ≡⟨ P.sym (proj₁ ℤ*-identity (k ℤ.* k)) ⟩
  (+ 1) ℤ.* (k ℤ.* k)
  ■
    where
      open P.≡-Reasoning
      k = + suc (suc (n ℕ.+ n))
      lem : (j : ℤ) -> ((+ 2) ℤ.* j ≡ j ℤ.+ j)
      lem j = trans (proj₂ ℤdistrib j (+ 1) (+ 1)) (cong₂ ℤ._+_ (proj₁ ℤ*-identity j)
        (proj₁ ℤ*-identity j))

+-red₂ : (n : ℕ) -> (((+ 1) ÷suc (suc ((suc (n ℕ.+ n)) ℕ.+ (suc (n ℕ.+ n))))
    + (+ 1) ÷suc (suc ((suc (n ℕ.+ n)) ℕ.+ (suc (n ℕ.+ n)))))
  + ((+ 1) ÷suc (suc ((suc (n ℕ.+ n)) ℕ.+ (suc (n ℕ.+ n))))
    + (+ 1) ÷suc (suc ((suc (n ℕ.+ n)) ℕ.+ (suc (n ℕ.+ n))))) ℚ.≃ ((+ 1) ÷suc n))
+-red₂ n = IsEquivalence.trans ℚ.isEquivalence {start} {middle} {end}
  (+-exist {1÷k + 1÷k}{1÷j}{1÷k + 1÷k}{1÷j} (+-red₁ j) (+-red₁ j)) (+-red₁ n)
  where
    j = suc (n ℕ.+ n)
```

```
    k = suc (j ℕ.+ j)
    1÷j = (+ 1) ÷suc j
    1÷k = (+ 1) ÷suc k
    start = (1÷k + 1÷k) + (1÷k + 1÷k)
    middle = 1÷j + 1÷j
    end = (+ 1) ÷suc n

ℚ≤lem : {m n : ℕ} -> ((+ 1) ÷suc (m ℕ.+ n) ≤ (+ 1) ÷suc m)
ℚ≤lem {m}{n} =  *≤* (ℤ.+≤+ (ℕ.s≤s ((m≤m+n m n) +-mono (z≤n))))

postulate ℚtriang : (x y z : ℚ) -> (| x - z | ≤ | x - y | + | y - z |)

postulate _ℚ+-mono_ :  _+_ Preserves₂ _≤_ ⟶ _≤_ ⟶ _≤_
```

4.3 Data/Real.agda

```
module Data.Real where

open import Data.Sum
open import Data.Rational as ℚ using (ℚ; -_ ; _*_; _÷suc_;
  _-_; _+_; |_|;  decTotalOrder; _≤_; *≤* ; _≤?_; _÷_)
open import Data.Rational.Properties using (ℚ-swap; ℚabs₂;
  +-red₂; ℚtriang; _ℚ+-mono_; ℚ≤lem; _⁻¹)
open import Data.Integer as ℤ using (ℤ; +_ ; -[1+_])
open import Data.Nat as ℕ using (ℕ; suc; zero)
open import Relation.Binary.Core using (Rel; IsEquivalence)
import Level
open import Relation.Binary using (module DecTotalOrder)
open import Relation.Binary.PropositionalEquality as P using
  (_≡_; refl; subst; cong; cong₂)
open import Data.Product
import Relation.Binary.PreorderReasoning as Pre


--Constructible Real numbers as described by Bishop

--A real number is defined to be a sequence along
--with a proof that the sequence is regular
record ℝ : Set where
  constructor Real
  field
    f : ℕ -> ℚ
    reg : {n m : ℕ} -> | f n ℚ.- f m | ℚ.≤ (suc n)⁻¹ ℚ.+ (suc m)⁻¹
```

```
------------------------------------------------------------------------
-- Equality of real numbers.
infix 4 _≃_

_≃_ : Rel ℝ Level.zero
x ≃ y =  {n : ℕ} -> | ℝ.f x n - ℝ.f y n | ≤ (suc n)⁻¹ ℚ.+ (suc n)⁻¹

-- Proof that this is an equivalence relation-------------------

--This lemma ((2.3) in Constructive Analysis) gives us a
--useful way to show equality
postulate Bishopslem : {x y : ℝ} ->
           ({j : ℕ} -> ∃ λ N -> ({m : ℕ} ->
           | ℝ.f x (N ℕ.+ m) - ℝ.f y (N ℕ.+ m) | ≤ (suc j)⁻¹))
           -> (x ≃ y)

isEquivalence : IsEquivalence _≃_
isEquivalence = record {
  refl = λ {x} -> refl≃ {x} ;
  sym = λ {x}{y} -> sym≃ {x}{y};
  trans = λ {a}{b}{c} -> trans≃ {a}{b}{c}
  }
  where

  --reflexitivity
  refl≃ : {x : ℝ} -> (x ≃ x)
  refl≃ {x} = ℝ.reg x

  --symmetry
  sym≃ : {x y : ℝ} -> (x ≃ y -> y ≃ x)
  sym≃ {x}{y} x≃y = λ {n} ->
    subst (λ a -> a ≤ (suc n)⁻¹ ℚ.+ (suc n)⁻¹)
    (ℚabs₂ (ℝ.f x n) (ℝ.f y n)) (x≃y {n})

  --transitivity
  trans≃ : {x y z : ℝ} -> (x ≃ y) -> (y ≃ z) -> (x ≃ z)
  trans≃ {x}{y}{z} x≃y y≃z = Bishopslem {x}{z} (λ {j} ->
    N {j} , λ {n} -> (begin
    | ℝ.f x (N {j} ℕ.+ n) - ℝ.f z (N {j} ℕ.+ n) |
      ∼⟨ ℚtriang (ℝ.f x (N {j} ℕ.+ n)) (ℝ.f y (N {j} ℕ.+ n)) (ℝ.f z (N {j} ℕ.+ n)) ⟩
    | ℝ.f x (N {j} ℕ.+ n) - ℝ.f y (N {j} ℕ.+ n) | +
    | ℝ.f y (N {j} ℕ.+ n) - ℝ.f z (N {j} ℕ.+ n) |
    ∼⟨ (x≃y {N {j} ℕ.+ n}) ℚ+-mono (y≃z {N {j} ℕ.+ n}) ⟩
```

```
((suc (N {j} ℕ.+ n))⁻¹ ℚ.+ (suc (N {j} ℕ.+ n))⁻¹) ℚ.+
((suc (N {j} ℕ.+ n))⁻¹ ℚ.+ (suc (N {j} ℕ.+ n))⁻¹)
∼⟨ ((ℚ≤lem {N {j}} {n}) ℚ+-mono (ℚ≤lem {N {j}} {n}))
       ℚ+-mono
     ((ℚ≤lem {N {j}} {n}) ℚ+-mono (ℚ≤lem {N {j}} {n})) ⟩
((suc (N {j}))⁻¹ ℚ.+ (suc (N {j}))⁻¹) ℚ.+
((suc (N {j}))⁻¹ ℚ.+ (suc (N {j}))⁻¹)
∼⟨ ≈->≤ (+-red₂ j) ⟩
((suc j)⁻¹ ■) ))
where
  open DecTotalOrder ℚ.decTotalOrder using ()
    renaming (reflexive to ≈->≤; trans to ≤trans; isPreorder to ℚisPreorder)
  open Pre record {isPreorder = ℚisPreorder}
  N = λ {j} -> suc ((suc (j ℕ.+ j) ℕ.+ (suc (j ℕ.+ j))))
```

References

Bishop, E. and Bridges, D.: 1985, Constructive analysis, volume 279 of grundlehren der mathematischen wissenschaften.

Curry, H. B., Hindley, J. R. and Seldin, J. P.: 1980, To hb curry: essays on combinatory logic, lambda calculus, and formalism.

sabry: 2014, operations on rationals, https://github.com/sabry/agda-stdlib/blob/master/src/Data/Rational.agda.