Université de Montréal

Quotients Types in Typer

par

James Tan Juan Whei

Département de mathématiques et de statistique
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Discipline

February 5, 2024

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Quotients Types in Typer

présenté par

James Tan Juan Whei

a été évalué par un jury composé des personnes suivantes :

Nom du président du jury

(président-rapporteur)

Nom du directeur de recherche

(directeur de recherche)

Nom du membre de jury

(membre du jury)

Résumé

...sommaire et mots clés en français...

Abstract

Contributions:

- Modified the Eq type by basing it on an Interval type

- Somewhat different version of transp that seems to still do the job

- Demonstrate that a 'cubical' equality makes it easier to manipulate equality proofs

- Some proofs are clearer and are more intuitive

- Add heterogenous equality, required to make depelim of quotients possible

- Notably this allows us to represent path overs

- Added quotient types to Typer

- Explored the usability and convenenience of Typer's quotients for programming tasks

- Demonstrate also that such quotients have little to no runtime cost

- Developed a library to facilitate the usage of quotients

- Convenience functions, macros etc

- Explore the writing of proofs in Typer, this is the first ever attempt to really write a substantial amount of proofs in Typer

- Added some constructs that facilitate the writing of proofs in general in Typer

- Macros such as equational reasoning

- Integer axioms

Contents

List of tables

Liste des sigles et des abréviations

ETT	Extensional Type Theory
HoTT	Homotopy Type Theory
UIP	Uniqueness of Identity Types

Remerciements

...remerciements...

Introduction

The 'propositions as types' paradigm states that proofs are analogous to programs. Writing programs in the form of code is equivalent to writing mathematical proofs. For instance, a function of the type `A → B` is analogous to an implication `A ⊃ B`. If we think of the types `A` and `B` as propositions, then such a function transforms all proofs of `A` to proofs of `B`[1]. Such a function corresponds directly to our intuition of 'A implies B', its existence is thus a proof of `A ⊃ B`. A conjunction `A ∧ B` is represented by a pair `A × B`. In order to prove that `A` and `B` are both true, we need a proof of `A` and `B` respectively, which is precisely what we need a construct the pair `A × B`. In the same vein, a disjunction `A ∨ B` is interpreted as a sum type `A + B`. A sum type has two constructors, one of them requires a proof of `A` while the other requires a proof of `B`, this corresponds to our interpretation of `A ∨ B` that either `A` or `B` is true.

However, in order to do anything meaningful, we would need a way to express the notion equality, i.e. a crucial ingredient in expressing mathematical truths. Since equalities are mathematical propositions, we require a type that plays the role of equality in mathematical proofs. For instance, we might want to state the theorem that the addition of natural numbers is commutative, i.e. $\forall xy.x+y \equiv y+x$. To this end, our system requires an equality type[2] that allows us to make statements such as the above. Such a type provides us with a proof object that can be manipulated like any other. This implies that we can manipulate it with other operations such as logical negation, conjunction, disjunction etc. This is in stark contrast to judgemental equality which only exists in the metatheory and cannot be manipulated directly in programs.

Not only can we write mathematical proofs, we may also prove properties about our programs. In general, proofs are not executed during runtime, they are merely verified at compile

[1] We only consider total functions.

[2] In this paper, we shall use the terms 'equality type', 'identity type' and sometimes 'path' interchangeably. (TODO: Could be a bad idea, maybe I should choose one and stick to it, in which case I'm leaning towards 'equality type'.)

time, or more precisely, during type-checking. This allows us to prove that our programs have the right behaviour, and this is achieved without incurring additional runtime cost. Consider a function that accesses the i-th element of an array with the below type:

```
arrayGet : (a : Array A) → (i : Nat) → i < length a → A
```

In order to invoke this function, the user would need to provide a proof that the index i is indeed within the bounds of the array. This gives us a guarantee that array accesses will never lead to a runtime error. Additionally, this could also allow us to skip runtime bounds-checking as the index is necessarily valid as long as the program is well-typed.

Equality proofs are also used extensionally in the internal language of the Glasgow Haskell Compiler (GHC) as they form the backbone of features such as generalised algebraic data types, functional dependencies and associated types [**?**]. Another notable example of the usage of proofs in real world programs is the CompCert project that worked on the formal verification of a C compiler [**?**].

0.1. Equality

For two terms x and y of the same type A, we would like to be able to state the proposition that the two terms are equal. For now, we merely consider equality between terms of the same type. This gives us a type former that takes a type as input, along with two terms of that type. In Typer, equality between terms is represented by the Eq type former[3] which has the below type signature.

```
Eq : (A : Type) ⇛ (x y : A) → Type
```

The Eq type former takes a type as input, along with two terms of that type that we wish to claim are equal[4]. We could then construct the type that stipulates that the addition of natural numbers is commutative by using the Eq type former.

```
Nat+_comm : (m : Nat) → (n : Nat) → m + n ≡ n + m;
```

$m + n \equiv n + m$ is merely a type and can be seen as a statement of the proposition that $m + n$ and $n + m$ are equal. In order to prove that the proposition is actually true, we would need to construct a term of this type. Such a term may be seen as a proof object that witnesses the validity of the proposition. Interested readers may find the complete proof of the commutativity of addition of natural numbers in **??**.

[3] A type former can be thought of as a constructor that constructs a Type.

[4] For aesthetic purposes, we shall use the operator $\equiv$ to represent the Eq type throughout the rest of the paper.

To see how such a proof may be utilised, consider the following definition of a length indexed vector:

```
data Vector (A : Type) : Nat → Type where
    Vnil : Vector A 0
    Vcons : {n : Nat} → A → Vector A n → Vector A (suc n)
```

As its name would suggest, the length of such a vector is found in its type itself. Since we are able to prove that the addition of natural numbers is commutative, we would like to be able to say something regarding the types *Vector* A $(m + n)$ and *Vector* A $(n + m)$. Since we expect that it should be possible to prove that the two types are equal, this then implies that we should also be able to convert a vector of type *Vector* A $(m + n)$ to one of type *Vector* A $(n + m)$.

```
Vlength-comm+ : (m n : Nat) ⇛ Vector ?A (m + n) → Vector ?A (n + m)
Vlength-comm+ = lambda A m n ⇛ Eq_cast (p := Nat+_comm m n) (f := Vector A)
```

By making use of a proof that $m + n \equiv n + m$, we are able to perform type casting on a vector of type *Vector* A $(m + n)$ to produce one of the desired type. This makes use of the fact that the equality type satisfies substitution, i.e. if two terms are equal, then if a predicate is true of the first term, then it is necessarily also true for the second term.

Now that we have seen how equality proofs may be used in programs, the next question that ought to be answered is how are such terms introduced. The one and only constructor of the Eq type is the following:

```
Eq_refl : (x : ?A) ⇛ x ≡ x;
```

Given a term x of an arbitrary type A, we can trivially construct the proof that x is equal to itself. This is known as the reflexive proof of equality.

Recent advances in type theory inspired from topology propose a different approach to represent equality proofs by modelling them as functions out of a unit interval to some type A. In this alternative formulation of equality proofs, an equality is viewed as a continuous path along the unit interval that connects two terms that are equal. The return value of such functions may not directly depend on the value of the interval type. For instance, the function may not do a case analysis on the interval and return a different term of type A depending on which endpoint of the interval is encountered. This essentially implies that such a function would indeed be constant on the unit interval and aptly respects the requirements of an equality proof. An equality type based on the interval also allows us to derive many proofs regarding the equality type in an alternative manner that is much cleaner and more elegant. It also allows to derive new theorems regarding the equality type that are desirable,

such as functional extensionality. This flavour of equality was first conceived in cubical type theories in order to provide a computational interpretation of the univalence axiom [**?**]. This allows the encoding of equivalences as equalities, in which case it is precisely because their equality proofs are non-trivial, i.e. proof relevant, that the univalence axiom may be proven as a theorem. However in the case of Typer, it is desirable for equality proofs to remain irrelevant so that they may be erased before runtime. This is essential so that operations involving the equality type may be treated as no-ops at runtime. Hence, we seek to reap the benefits of having a more powerful equality type, all while retaining the runtime irrelevance of equality proofs.

0.2. Quotients

In keeping with the theme of equality, we shall discuss the implementation of quotient types in Typer. Quotient types allow us to redefine the equality of a type, usually on the basis of an equivalence relaton on the type. This is inspired by quotients in set theory, where if we are given a set A and an equivalence relation $\sim$ on A, then we can construct the quotient set of A by $\sim$, which we denote as $A/\sim$. Equivalent elements in A are considered equal in $A/\sim$. The subsets of elements that are considered equal in $A/\sim$ are precisely the subsets whose elements are all equivalent under the equivalence relation. We can say that for some $a \in A$, the subset of A such that all elements are equivalent to a is $\{a' \in A \mid a' \sim a\}$. Indeed, this subset is what one would call the equivalence class of a, this is usually denoted as $[a]$. When we take the quotient of a set, we are essentially partitioning the set into equivalence classes, implying that we can regard a quotient set as simply the set of equivalence classes induced by the equivalence relation.

Quotient types often provide us with alternative ways of defining inductive types that are familiar to us. Some of these definitions could potentially be better than the original formulations depending on the use cases of the resulting data type. For example, as was noted in [**?**], by defining the List data type as a quotient of binary trees where the joining operation of two trees is associative, the resulting List datatype has a constant time append operation.

Quotient types are especially useful when defining datatypes that have redundancies. By taking the quotient of such types, we ensure that redundancies are identified. In a way, the type system does the book-keeping for us by ensuring that well-typed programs necessarily treat such redundancies in a consistent manner. Consider the set of rational numbers defined as such:

$$\mathbb{Q} = \{(x, y) : x \in \mathbb{Z}, y \in \mathbb{Z}, y \neq 0\} \tag{0.2.1}$$

We represent a fraction as a pair of integers such that the first integer is the numerator and the second non-zero integer is the denominator. We note that each rational number should then be considered equal to an infinite number of other rationals. For example, $\frac{0}{1}$ should be equal to $\frac{0}{-2}$, $\frac{0}{-1}$, $\frac{0}{2}$, $\frac{0}{3}$ etc. However, the naive encoding of the rationals as pairs of integers does nothing to ensure that equal rationals are indeed treated equally. This problem is remedied by quotienting pairs of integers by an appropriate equivalence relation.

We also show that there exists a class of quotients that can be defined without the addition of a new type former, i.e. quotients that are characterised by a normalisation function [**?**,**?**]. For these quotients, it is possible to elect a canonical representative for each equivalence class. In the rest of the paper, we first describe the typing and reduction rules of the introduced type, and then we discuss the library that was implemented to faciliate the usage of quotient types in Typer. Finally, we illustrate how the quotient type may be used by providing examples in the form of code along with commentary.

0.3. Contributions

Inspired by cubical type theory, we replace the existing implementation of the equality type in Typer by one that is based on the Interval type. We also demonstrate that proofs regarding the equality type are consequently generally easier to realise by providing a library of theorems regarding the equality type. We also introduce the notion of heterogenous equality, making it possible to identify terms of different types[5]. This is necessary to enable the dependent elimination of quotient types, notably because this enables the representation of pathovers [**?**] (TODO: The paper's definition of Heq has the proof alpha that both types are equal, this concerns me a little bit as our version doesn't have that, is this justifiable or should I adapt to their version instead?). We also implemented a proof of Hedberg's theorem [**?**] in Typer. The theorem states that if a type has decidable equalities, then its equalities are necessarily trivial. Next, we introduced quotient types to Typer. This work then explores the usability and convenience of quotient types in programming tasks. To facilitate the use of quotients, we developed a library of convenience functions, macros, and also theorems regarding quotient types. We also investigate the runtime cost of the utilisation of quotient types. This is the first work that has attempted to develop a substantial number of proofs in Typer. We introduce several constructs to facilitate the writing of proofs in general in Typer in the form of macros and axioms.

[5]To identify A and B is to claim that they are equal.

0.4. Notes and TODOs

- We have a proof of Hedberg's theorem to facilitate eliminations to 'Sets'. After explaining why this is necessary, we can motivate the addition of the set truncation constructor for Quotients. If we can prove that $\mathbb{Z}$ is a set when defined inductively (with the aforementioned theorem), then it would be odd if the same cannot be done if $\mathbb{Z}$ was defined as a quotient of $\mathbb{N} \times \mathbb{N}$. Indeed, this entails bigger consequences (as mentioned in https://staff.math.su.se/anders.mortberg/papers/cubicalagda2.pdf section 2.4.1). (TODO: refer to my notes for a proof that Unit $\neq$ Unit/R without the truncation).

Chapter 1

Cubical Equality

1.1. Interval type

First, we provide the typical definition of the *Interval* type, subsequently we present our alternative formulation of the *Interval* type and show that it is equivalent to the traditional definition.

$$\frac{}{\Gamma \vdash \mathrm{I} : \mathit{Type}}$$

$$\frac{}{\Gamma \vdash i_0 : \mathrm{I}}$$

$$\frac{}{\Gamma \vdash i_1 : \mathrm{I}}$$

$$\frac{}{\Gamma \vdash \mathit{seg} : i_0 \equiv i_1}$$

We postulate the existence of the unit interval, along with the ends of the interval i_0 and i_1. We also have *seg* which is a witness of a continuous path between i_0 and i_1.

Elimination of I is similar to that of *Booleans* the only difference is that a proof must be provided to prove that $i_0 \equiv i_1$ is respected. The recursion principle of I can thus be stated as follows[1]:

[1]When we say recursion principle, we refer to the definition as is employed in an introduction to the Coq proof assistant [**?**]. This terminology has also been employed in the HoTT book [**?**].

$$\frac{\Gamma \vdash A : Type \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A \quad \Gamma \vdash P : M \equiv N}{\Gamma, i : \mathrm{I} \vdash rec_{\mathrm{I}}(P, i) : A}$$

The recursion principle computes as follows:

$$rec_{\mathrm{I}}(\_.A, M, N, P, i_0) = M$$
$$rec_{\mathrm{I}}(\_.A, M, N, P, i_1) = N$$
$$\mathsf{cong}_{rec_{\mathrm{I}}(P)}(seg) = P$$

cong refers to the congruence property of equality and is explained in more detail in section 1.7. We can expect the first two equations to hold definitionally. If we imagine that the first two computation rules were defined via pattern-matching, it would make sense for these equations to hold definitionally. The third equation is a more curious case, it suggests that by congruence of equality, the application of the Interval recursor with the the equality proof P to *seg* should yield an equality proof between M and N. More specifically, such an equality proof should be equal to P itself. Equalities between equality proofs are only an important consideration in a proof relevant world, e.g. in the context of HoTT. In the world of Typer where we demand that equality proofs be erasable and play no role during computation, such considerations are not of huge importance. We mention nevertheless that in most cases, the third equation only holds propositionally, as is the case when it is defined as a user-defined datatype in other type systems such as Agda.

1.2. Motivation

We show that a function out of the interval into some type A is equivalent to a path between two terms of type A, in other words, we show the following

$$\mathrm{I} \to A \simeq \Sigma_{x:A}.\Sigma_{y:A}.x \equiv y$$

We proof this equivalence by proving that `g` is a quasi-inverse of `f` as described in chapter 2 of the HoTT book [**?**]. To this end, we define a function f and its inverse g as such:

$$f : (\mathrm{I} \to A) \to \Sigma_{x:A}.\Sigma_{y:A}.x \equiv y$$
$$f(h) := \langle h(i_0), \langle h(i_1), \mathsf{cong}_h(seg)\rangle\rangle$$

$$g : \Sigma_{x:A}.\Sigma_{y:A}.(x \equiv y) \to (\mathrm{I} \to A)$$
$$g\,\langle x, \langle y, p\rangle\rangle := \lambda i.rec_{\mathrm{I}}(\_.A, x, y, p, i)$$

Now, we need to prove that g is both the left and right inverse of f.

$$\alpha\text{-}helper : (h : \mathrm{I} \to A) \to (i : Interval) \to (g(f\ h))\ i \equiv h\ i$$
$$\alpha\text{-}helper(h, i) = rec_{\mathrm{I}}(i.\ (g(f\ h))\ i \equiv h\ i,\ \mathsf{refl}(h(i_0)),\ \mathsf{refl}(h(i_1)),\ \mathsf{cong}_{\lambda i.\mathsf{refl}(h(i))}(seg),\ i)$$
$$\alpha : (h : \mathrm{I} \to A) \to g(f(h)) \equiv h$$
$$\alpha(h) = \mathsf{funExt}(\alpha\text{-}helper(h))$$

$$\beta : (e : \Sigma_{x:A}.\Sigma_{y:A}.x \equiv y) \to f(g(e)) = e$$
$$\beta(e) = \mathsf{refl}_{\Sigma_{x:A}.\Sigma_{y:A}.x\equiv y}(e)$$

For the sake of simplifying the proof of β, we assume the η-rule for dependent pairs. And with that, the proof is complete.

1.3. Justification

We have established that the equality type (also known as the identity type) is equivalent to a function out of the interval. We have also illustrated that such a function is expected to be constant on the unit interval. This 'condition' is enforced by virtue of the fact that *Interval* is defined as a higher inductive type [**?**] with an identification of the two endpoints. The type system then enforces that all eliminations of the *Interval* type be necessarily constant on the two endpoints. An alternate way of enforcing this is by using functions with erasable arguments out of a 2-type, i.e. a type with two distinct constructors that take no arguments, such as the Boolean type. An argument that is marked as erasable is one that has no computational role during runtime. Within the function body, such an argument may only be used in an 'erasable' manner, for instance, it may be used in the computation of an expression that is passed as an erasable argument to yet another function. In other words, we can first define the *Interval* type as a normal algebraic data type. Next, we can model the

equality type as an erasable function out of the *Interval* type. (FIXME: Abuse of terminology here? When I say erasable function I mean that the argument is erasable, not the function itself.)

```
type I
  | i₀
  | i₁
```

And then define a function with the following type:

```
h : I ⇛ A
```

In Typer, we use ⇛ to denote erasable arguments in function types. It is vacuously true that $h(i_0) \equiv h(i_1)$ since what is returned by the function could not have been influenced by the argument that was passed to it by virtue of the fact that the argument is erasable. With that, we are able to postulate that

$$\mathrm{I} \Rrightarrow A \simeq \Sigma_{x:A}.\Sigma_{y:A}.x \equiv y$$

The above shall be the basis of our new definition of identity types.

1.4. Manipulation of the Interval type

We introduce a few operations on the interval type whose usefulness will become clear after we introduce our version of the equality type that is based on the interval type.

Negation

The negation operator does what its name suggests, i.e. it simply negates endpoints of the Interval type. Unsurprisingly, it has the following type:

```
I_not : I → I
```

The negation operator computes as follows:

$$\frac{}{\texttt{I\_not}\ i_0 \rightsquigarrow i_1}\ \text{I-NOT}_1 \qquad \frac{}{\texttt{I\_not}\ i_1 \rightsquigarrow i_0}\ \text{I-NOT}_2$$

$$\frac{}{\texttt{I\_not}\ (\texttt{I\_not}\ i) \rightsquigarrow i}\ \text{I-NOT-NOT}$$

I-NOT-NOT is a special rule that states that the composition of `I_not` with itself is equal to the identity function. This is something that we could have proven propositionally, however, for our purposes it is crucial that this equality holds definitionally. (TODO: Actually wait, I have to check this claim. It's also possible that this is not entirely necessary, though it makes proofs that do indeed require this property a lot simpler and more elegant)

Meet. The meet operator returns the minimum of two terms of the Interval type. The order of the Interval type is such that $i_0 \leq i \leq i_1, \forall i \in I$

```
I_meet : I → I → I
```

It has the following reduction rules:

$$\frac{}{\texttt{I\_meet}\ i_0\ i \rightsquigarrow i_0}\ \text{I-MEET}_1 \qquad \frac{}{\texttt{I\_meet}\ i_1\ i \rightsquigarrow i}\ \text{I-MEET}_2$$

$$\frac{}{\texttt{I\_meet}\ i\ i_0 \rightsquigarrow i_0}\ \text{I-MEET}_3 \qquad \frac{}{\texttt{I\_meet}\ i\ i_1 \rightsquigarrow i}\ \text{I-MEET}_4$$

$$\frac{}{\texttt{I\_meet}\ i\ i \rightsquigarrow i}\ \text{I-MEET}_5$$

I-MEET$_1$ is valid as i_0 is lesser or equal to anything else. In a similar fashion, I-MEET$_2$ is justified as everything is lesser or equal to i_1. Since the operator is commutative, I-MEET$_3$ and I-MEET$_4$ are complementary to I-MEET$_1$ and I-MEET$_2$. The minimum of i and i is i itself, justifying the I-MEET$_5$ rule. Note that since the Interval type does not behave like a Boolean algebra, it is not the case the `I_meet` i (`I_not` i) $\rightsquigarrow i_0$.

Join

TODO: So far I haven't found a need for this operator. Do some investigation to see if this is a worthy addition

1.5. Elimination of the Interval type

We present the one and only elimination rule of our Interval type. It is largely inspired by the (generalised) transport operation used in Cubical Agda [**?**]. It has the following typing rule:

$$\frac{\Gamma \vdash A : I \Rrightarrow \mathsf{Type} \quad \Gamma \vdash r : \mathsf{Interval} \quad \Gamma \vdash a : A\ (\_ := i_0)}{\Gamma \vdash I\_transp\ A\ r\ a : A\ (\_ := I\_not\ r)}$$

We claim that this is a generalised version of the transport operation as the argument r dictates when I_transp should behave like the identity function. When $r = i_1$, then the type of $I\_transp\ A\ r$ is $A\ (\_ := i_0) \to A\ (\_ := i_0)$, i.e. it behaves like the identity function. Since our version of I_transp is slightly different from that of Cubical Agda, we do not need the additional side condition that states that A needs to be a constant function when $r = i_1$ in order for the above expression to typecheck. This is primarily because the ultimate return type of the function is dependent on r itself. When $r = i_0$, then the type of $I\_transp\ A\ r$ is $A\ (\_ := i_0) \to A\ (\_ := i_1)$, this gives us a function that transports an expression of type $A\ (\_ := i_0)$ to type $A\ (\_ := i_1)$. We shall make use of this property subsequently to define the actual transport function. However, the other property of the I_transp function may appear bizarre at first sight, but in reality it ensures that $I\_transp\ A\ r\ a$ is equal to a[2].

The computational behaviour of the function is as follows:

$$\frac{}{I\_transp\ A\ r\ a \rightsquigarrow a}$$

The function is implemented in Typer as a primitive with the following type signature:

```
I_transp : (A : I ⇛ Type) ⇛ (r : I) ⇛ A (_ := i0) → A (_ := I_not r)
```

Its implementation is simple in that it simply returns the argument a untouched.

1.6. Identity type

1.6.1. Heterogenous equality

With the above definition of I, we are now ready to introduce the notion of heterogenous equality. Such an equality type allows us to identify two terms that are not necessarily of the same type. The heterogenous equality type has the following type signature[3] and is implemented axiomatically. Usually, when we define an axiom in type theory, we simply claim that it is an inhabitant of some type without saying anything about its behaviour. However, in our case we also supplement it with the appropriate reduction rules so it has the right computational behaviour. In the absence of this, e.g. if we were to define the

[2]This equality would be heteregenous as the two terms may not have the same type

[3]Although not mentioned explicitly, functions involving the identity type are universe polymorphic in their arguments and return type.

computation rules for axioms in the form of propositional equalities, then the usage of these axioms suffers greatly as programs and proofs tend to be clumsy and inelegant.

```
Heq : ?x → ?y → Type
```

The introduction rule of the above type leverages the fact that erasable functions out of the interval are equivalent to identity types.

```
Heq_eq : (t : I ⇛ Type)
         ⇛ (f : (i : I) ⇛ t (_ := i))
         ⇛ Heq (f (_ := i0)) (f (_ := i1))
```

The elimination of equality types is trivial, it merely returns a function out of the interval that represents the underlying equality.

```
Heq_uneq : (t : I ⇛ Type)
           ⇛ (x : t (_ := i0)) ⇒ (y : t (_ := i1))
           ⇛ (p : Heq x y) ⇛ (i : I) ⇛ t (_ := i)
```

The composition of *Heq_uneq* and *Heq_eq* (modulo the implicit arguments) is equal to the identity function, which gives us the η-equivalence for some `p:Heq ?x ?y`:

$$Heq\_eq\ (f := \lambda i \Rrightarrow Heq\_uneq\ (p := p)\ (i := i)) = p$$

Reduction rules.

$$\frac{}{Heq\_uneq\ x\ y\ p\ i_0 \rightsquigarrow x}\ \text{UNEQ-LEFT}$$

$$\frac{}{Heq\_uneq\ x\ y\ p\ i_1 \rightsquigarrow y}\ \text{UNEQ-RIGHT}$$

When `Heq_uneq` is called with canonical values of I, it reduces to the corresponding endpoint of the equality proof. This is a crucial property that allows derive theorems such as functional extensionality in section 1.7.

1.6.2. Homogenous equality

Homogenous equality is by far the more typical notion of equality. We are able to define it as a special case of heterogenous equality as a special case of of `Heq x y` where `x` and `y` are of the same type.

```
Eq : (t : Type) ⇛ t → t → Type
Eq x y = Heq x y
```

```
Eq_eq : (t : Type) ⇛ (f : I ⇛ t)
        ⇛ Eq (t := t) (f (_ := i0)) (f (_ := i1))
Eq_eq t f = Heq_eq (t := λ _ ⇛ t) (f := f)

Eq_uneq : (t : Type) ⇛ (x : t) ⇒ (y : t) ⇒ (p : Eq x y)
          ⇛ (i : I) ⇛ t
Eq_uneq p i = Heq_uneq (t := λ _ ⇛ t) (p := p) (i := i)
```

1.7. Examples

Common properties of the equality type remain true, some of them are easier to prove than with a traditional equality type.

Reflexivity

A reflexive equality is the most trivial equality that we are able to construct, and indeed, it is the one that we are able to construct with an erasable function out of I given that such a function is necessarily a constant function.

```
Eq_refl : (x : ?t) ⇛ Eq x x
Eq_refl x = Eq_eq (f := λ _ ⇛ x)
```

Commutativity

This is the first of many properties that can be proved in an elegant manner by virtue of defining the equality type based on functions out of the interval. This property has many names, such as the commutativity of equality, the symmetry of equality, or the inversion of a path. Consider an equality between x and y, we can think of the two terms as points, the equality itself can then be thought of as a continuous path between the two points. We think of the path as a directed path from x to y. The commutativity of equality basically means that we are able to construct a path that goes in the opposite direction, in other words, we are able to invert the path. This is the intuition behind the following proof.

```
Eq_comm : (x y : ?t) ⇛ Eq x y → Eq y x
Eq_comm x y = Eq_eq (f := λ i ⇛ Eq_uneq (p := p) (i := !i))
```

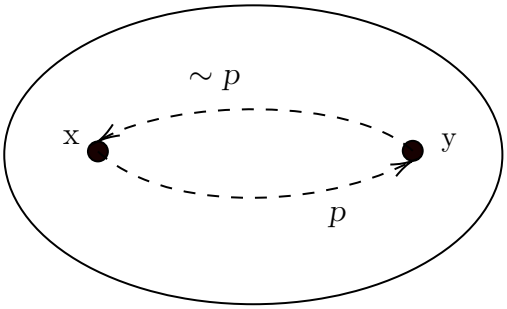


Fig. 1.1. A path from x to y along with its inversion

Functional extensionality

The notion of functional extensionality implies that pointwise-equal functions are indeed equal. By postulating the existence of the Interval type along with an equality type that's based on it, functional extensionality can be derived trivially as is illustrated by the below proof.

```
Eq_funExt : (f : (? → ?)) ⇒ (g : (? → ?))
          ⇒ (p : (x : ?) → Eq (f x) (g x)) → Eq f g
Eq_funExt p = Eq_eq (f := λ i ⇛ λ x →
                                  Eq_uneq (p := p x) (i := i))
```

We note that the above proof makes use of the η-equivalence of functions, i.e. $\lambda x \to f\, x \equiv f$. Another thing that is worth noting is that we interpret erasable functions out of I themselves as an equality type, i.e. if we do without the *Eq_eq* constructor, and if we think of *Eq_uneq* as the direct application of an expression of type I to such a function, we obtain the following interpretation of *Eq_funExt* as is the case in Cubical Agda.

```
funExt : ∀ {A B : Type ℓ} {f g: A → B}
       → (p : (x : A) → f x ≡ g x)
       → f ≡ g
funExt p i x = p x i
```

The proof p that the functions are extensionally equal may be seen to have the type $A \to \mathrm{I} \to B$, on other hand, a proof that the functions are equal would have the type $\mathrm{I} \to (A \to B)$. In this case, it becomes clear that a proof of functional extensionality merely needs to swap the order of its two arguments. This is observed in several recent proof assistants that implement cubical type theory, such as Cubical Agda as mentioned before, as well as coolTT and redTT.

Functional extensionality is an important property of the equality type, it will especially be required in order to prove the effectiveness of quotients in chapter 3. However, as we shall

see in ???, the existence of quotient types itself is actually sufficient to derive functional extensionality.

Congruence of equality

This is a another typical property of equality whose proof is significantly simplified in a system with a cubical equality.

```
Eq_cong : (x y : ?A) ⇒ (f : (?A → ?)) → (p : Eq x y)
          → Eq (f x) (f y)
Eq_cong f p = Eq_eq (f := λ i ⇛ f (Eq_uneq (p := p) (i := i)))
```

Like before, we examine the definition of the theorem in Cubical Agda for a change of perspective. We observe that the congruence of equality merely involves applying the function f to the terms at both endpoints of the equality.

```
congS : ∀ {A B : Type ℓ} → (f : A → B) (p : x ≡ y) → f x ≡ f y
congS f p i = f (p i)
```

By appealing to the η-equivalence of the equality type described in section 1.6, we also have that `Eq_cong id p` $\equiv$ `p` holds definitionally.

Transport

The transport operation (also known as *cast*) allows us to transport properties that are true for some type A to another type B if we have a proof that $A \equiv B$.

```
Eq_cast : (x y : ?A) ⇛ (p : Eq x y) ⇛ (f : (? → ?)) ⇛ f x
          → f y
Eq_cast A x y p f fx =
  I_transp (A := λ t ⇛ f (Eq_uneq (t := t) (p := p) (i := i)))
           (r := i₀) fx
```

Here, we make use of the built-in function *I_transp* by passing it the argument $r := i_0$, intending it to behave as a transport function. In other words, by passing it something of type $f\,x$, it returns something of type $f\,y$, which is precisely what the transport/cast function is meant to do.

The *I_transp* function that looks slightly strange at first sight allows us to prove a certain property of the transport operation. Transporting across a reflexive path should be equivalent to a no-op, as is illustrated in the following proof.

```
Eq_cast_refl : (A : Type) ⇛ (x : A)
               → Eq (Eq_cast (p := Eq_refl (x := x))
```

```
                                (f := λ _ → A) x)
                        x
Eq_cast_refl A x = Eq_eq (f := λ i ⇛
                            I_transp (A := λ _ ⇛ A) (r := i) x)
```

J-rule

The sole term former of the Eq only allows the construction of reflexive equalities, at first sight this leads us to believe that not much can actually be done with the Eq type. After all, if all we had were proofs of the form $x = x$, it would be difficult to imagine achieving anything with them. The true magic of the equality type lies in it's elimination principle. Traditionally, the equality type comes with an elimination principle known as the J-rule. This was first introduced by Per Martin-Löf in [**?**]. Suppose that we have a type family C that is parametrised by two terms x and y of some type A as well as a proof that x and y are equal, i.e. of type Eq x y. If we have have a proof term p that witnesses the equality of two arbitrary terms a and b of type A, and we wish to produce a term of type C a b p, then the J-rule says that it suffices for us to provide a means of producing for all $x : A$ a term of type C x x (refl x). We now state the type of the J-rule:

```
J : (C : (x : ?A) -> (y : ?A) -> Eq x y -> Type)
    -> ((x : A) -> C x x Eq_refl)
    -> (x : A) -> (y : A) -> (p : Eq x y)
    -> C x y p;
```

This rule is essentially saying that in order to know what to do with an equality between an arbitrary x and y, we merely need to know how to handle the case where we are dealing a reflexivity proof. Now, given the fact that the reflexivity constructor is the only constructor of the Eq type, one might think that the above elimination principle is well justified. Recall that elimination principle of strictly positive types [**?**] can usually be derived based on the constructors of the datatype itself. For instance, consider the below inductive definition of the Peano natural numbers.

```
type Nat
    | zero
    | succ Nat
```

The corresponding elimination principle then takes on the following logical form:

```
Nat_elim : (P : Nat → Type)
           ⇛ P zero → ((n : Nat) → P n → P (succ n))
           → (n : Nat) → P n;
```

First, we need to define what needs to be done if the natural number we are trying to eliminate is *zero*. Next, if the natural number we want to eliminate is something of the form $succ\ n$, then we can imagine making a recursive call to the eliminator to produce a result for $succ\ n$ for n, and now we need a means of transforming a proof of $P\ n$ to one of $P\ (succ\ n)$. Indeed, the elimination principle of natural numbers is just mathematical induction. Hence, it is reasonable to say that in order to eliminate an equality proof, it suffices for us define what needs to be done for reflexivity proofs.

In Typer, the elimination rule for the Eq type was not provided in the form of the J-rule prior to this work, instead a cast function was provided as mentioned in a previous section. It is possible to derive Eq_cast from the J-rule. The natural question that we should now ask ourselves is whether the J-rule can be derived from Eq_cast, in other words we would like to know if the two elimination principles are equivalent. It turns out that one way of doing it hinges upon whether we are able to inhabit the following type:

```
eqUnique : (A : Type) ⇛ (a b : A) (p : Eq a b)
         → Eq (t := Sigma A (lambda a' -> Eq a a'))
                            (sigma a Eq_refl) (sigma b p);
```

This can be seen as a justification of the J-rule, by fixing one endpoint of an equality proof to be `a`, it is provable that any equality proof between `a` and some other arbitrary `b` may be proved to be equal to the reflexivity proof at `a`. We provide more details of this in the appendix (TODO: actually do this).

- TODO: If want to be able to prove that J can be derived from cast, we need to leverage the fact that $(x, refl) = (y, p)$, check notes for more details.

In Typer and in intensional type theory in general, we need to explicitly invoke a cast function of some sort, this is in contrast with extensional type theory (ETT) [**?**] where equality proofs in the context are automatically used by the typechecker to carry out casts whenever necessary. This is shown by the following judgement in ETT that states that equality proofs can be converted to judgemental equalities.

$$\frac{\Gamma \vdash t : \mathsf{Eq}\ a\ b}{\Gamma \vdash a \equiv b}$$

This is also the reason why typechecking is not decidable in ETT as the necessity to invoke the above rule is not syntax directed.

Transitivity

Continuing our analogy of equality proofs as paths, the transitivity property allows us to concatenate two paths. We join the right endpoint of the first path with the left endpoint of the second path, producing a new path. The proof of this theorem is as follows:

```
Eq_trans : (x y z: ?t) ⇛ Eq x y → Eq y z → Eq x z
Eq_trans x y z x=y y=z = Eq_cast (p := y=z)
                                (f := λ x' → Eq x x')
                                x=y
```

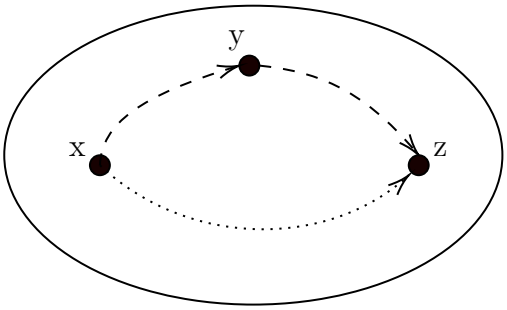


Fig. 1.2. The concatenation of a path between x and y and a path between y and z to produce a new path between x and z.

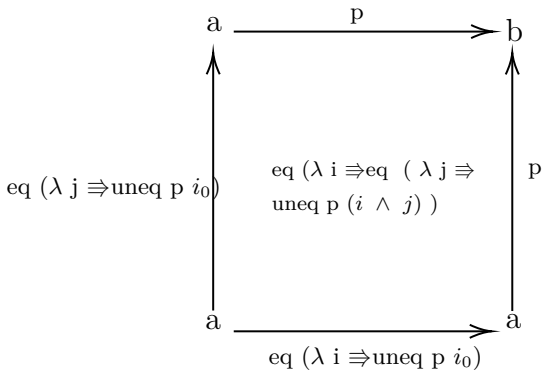
Connections

The operations that we defined on the Interval correspond to connections [**?**] that are typically drawn as squares. Such connections allow us construct squares from paths by adding degeneracies.

Meet. Suppose that we have some proof `p` between terms `a` and `b`, the interval meet operation allows us to construct a square that corresponds to the following expression:

```
sq = Eq_eq (f := λ i ⇛ Eq_eq (f := λ j ⇛ Eq_uneq (p := p) (i := I_meet i j)))
```

In this square, the `i` direction goes from left to right and the the `j` direction goes from bottom to up. Each side of the square corresponds to some equality proof. The top and right sides of the square are simply the original proof `p`. The left and bottom sides of the square are fixed to be constant paths to and from the left endpoint of the proof `p`. In other words, they can be seen to be reflexivity proofs of `a`. This is precisely what we meant earlier by degeneracies as such paths are trivial.



To illustrate why the given square corresponds to the expression given above, we shall examine each side of the square. Recalling the dimensions of the squares, on the left side of the square, the dimension `i` is i_0. So we if we fix `i` to be i_0 in `sq`, the resulting expression is the following, which we can reduce to the constant path on `a` by using the reduction rules of `I_meet` defined in section 1.4:

```
Eq_eq (f := λ j ⇛ Eq_uneq (p := p) (i := I_meet i0 j))
```

For the bottom side, we fix `j` to be i_0 to obtain the below expression, which is once again a constant path on `a`.

```
Eq_eq (f := λ i ⇛ Eq_uneq (p := p) (i := I_meet i i0))
```

The right side requires us to fix `i` to be i_1, giving us the below expression which simply reduces to `p`.

```
Eq_eq (f := λ j ⇛ Eq_uneq (p := p) (i := I_meet i1 j))
```

By fixing `j` to be i_1, we get the expression that corresponds to the top side, which also reduces to `p`.

```
Eq_eq (f := λ i ⇛ Eq_uneq (p := p) (i := I_meet i i1))
```

The construction of such a square is precisely what is required in order to prove the `eqUnique` property mentioned in section 1.7.

Join. TODO: TBD if I want to do this. Joins could be used to build the same square that we saw as above, the only difference is that the added degeneracy would be a refl path on b instead of a.

1.8. Equality as a mere proposition

Although we say that our equality type is a proposition, it is by no means a mere proposition (HoTT book [**?**] Chapter 3.3). This implies that it is not a given that two proofs of the same

equality are provably equal [?]. In other words, when we ask ourselves the question "Is it the case that a = b", we do not only care if the answer is yes or no, if they are indeed equal, we can interest ourselves in what ways exactly are they equal. This is in contrast to other systems where the Uniqueness of Identity Proofs principle (UIP) applies to all types. Notably, this is true of systems that admit axiom K [?] that states all identity proofs are equal to the reflexivity proof, and by extension all proofs of the same equality are equal. In the remainder of this section, we show that UIP is a property that is true for a certain class of types in Typer.

1.8.1. Hedberg's theorem

To set the stage, we provide a few definitions. A type is said to be **decidable** if we are able to decide whether the type is inhabited or not. In the case where it is inhabited, we must also have a witness of such an inhabitant. In Typer, decidability can be represented by the following inductive type.

```
type Dec (A : Type)
  | yes A
  | no (A -> False)
```

A type is said to be **collapsed** if any two inhabitants of the type may be proved to be equal, this corresponds directly the notion of a mere proposition. The two definitions that we just stated may also be used in the context of identity types. If we have say that a type has decidable equality, this implies that for any two terms of this type, we can decide whether or not they are equal. On top of that, if the type has collapsed identity types, then for any pair of terms of this type, any two proofs that they are equal are also equal. The notion of collapsible identity types corresponds to the notion of a Set or h-set in HoTT.

Theorem 1.8.1 (Hedberg's theorem [?])**.** *If a type* `A` *has decidable equality, then* `A` *has collapsed identity types.*

In this work, we provide a proof of this theorem in the form of Typer code. We present the type of the function without going into the details of its implementation[4].

```
dici : (di : (x : ?A) → (y : ?A) → Dec (x ≡ y))
       → (a : ?A) → (b : ?A)
       → (u : a ≡ b) → (v : a ≡ b)
       → u ≡ v
```

[4]The actual implementation may be found in the attached `samples/hedberg.typer` file

1.9. Limitations

We don't have primitives such as *hcomp*, what does this imply? Is there a certain class of proofs we are unable to construct? Or do we just suffer from some amount of inconvenience?

Chapter 2

Quotient Type

2.1. Typing rules

We present the *Quotient* primitives along with their typing rules.

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash R : A \to A \to \mathsf{Type}}{\Gamma \vdash \mathsf{Quotient}\ A\ R : \mathsf{Type}}\ \text{QUOT-FORMATION}$$

We introduce the Quotient type former, which takes a base type A along with a relation R as arguments. At this stage, we do not require the relation to be an equivalence relation, though in order to define a meaningful quotient type, the above requirement is often necessary. We also do not enforce that the relation be a mere proposition, however as we will see in chapter 3, this is required in order for the quotient to be effective.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash \mathit{Quotient}\ A\ R : \mathsf{Type}}{\Gamma \vdash \mathit{Quotient\_in}\ A\ R\ a : \mathit{Quotient}\ A\ R}\ \text{QUOT-INTRO-IN}$$

Quotient_in is none other than a term former that injects a term of type A into a quotient under some relation.

$$\frac{\Gamma \vdash a_1 : A \qquad \Gamma \vdash a_2 : A \qquad \Gamma \vdash r : R\ a_1\ a_2}{\Gamma \vdash \mathit{Quotient\_eq}\ A\ R\ a_1\ a_2\ r : \mathit{Quotient\_in}\ A\ R\ a_1 \equiv \mathit{Quotient\_in}\ A\ R\ a_2}\ \text{QUOT-INTRO-EQ}$$

By definition, given that two terms a_1 and a_2 of type A are related by some relation, then the injections of these two terms into a quotient under the aforementioned relation should be equal. *Quotient_eq* is a witness of the above property.

$$\frac{\Gamma \vdash P : \mathit{Quotient}\ A\ R \to \mathsf{Type} \qquad \Gamma \vdash q : \mathit{Quotient}\ A\ R \qquad \Gamma \vdash f : (a : A) \to P\ (\mathit{Quotient\_in}\ A\ R\ a) \qquad \Gamma \vdash h : (a_1\ a_2 : A) \to R\ a_1\ a_2 \to \mathit{Heq}\ (f\ a_1)\ (f\ a_2)}{\Gamma \vdash \mathit{Quotient\_elim}\ A\ R\ f\ h\ q : P\ q}\ \textsf{QUOT-ELIM}$$

We would like to enable the dependent elimination of quotients, in other words we would like to be able to eliminate a term of some quotient type *Quotient A R* to some type family P that is indexed by the quotient type itself. To that end, we require some function f that takes some a of the base type A to P (*Quotient_in A R a*). In a way, this implies that the function f has the chance of 'looking inside' the quotient, and thus we also require a proof that such a function f behaves consistently when applied to terms that are related by the underlying relation of the quotient. The usage of heterogenous equality here is necessary as $f\ a_1$ and $f\ a_2$ are not of the same type. We note as well that our dependent elimination of quotients is not limited to motives that are Props, as opposed to when quotient types are added to a theory interpreted in a setoid model as proposed by Li [**?**].

2.2. Reduction rules

We present the computation rule for the elimination of quotients.

$$\frac{q \rightsquigarrow \mathit{Quotient\_in}\ A\ R\ a}{\mathit{Quotient\_elim}\ A\ R\ f\ h\ q \rightsquigarrow f\ a}$$

We note that during *Quotient* elimination, the proof that the eliminator function respects the *Quotient* is ignored. Instead, the function is applied directly to the underlying element of the base type. The fact that the proof that the elimination respects the quotient is unused during computation and is merely utilised during type-checking justifies our subsequent choice of making the proof an erasable argument of *Quotient_elim*.

It is also interesting to note that we are not able to derive a meaningful η-rule. If we wish to apply *Quotient_in* after applying *Quotient_elim* such that the composition amounts to the identity function, this would imply that *Quotient_elim* would need to called with a function f that is itself the identity function. Unfortunately, in the general case this would not respect the underlying relation of the quotient. However, we could define an η-rule of sorts for quotients based on normalisation. (TODO: discuss and link)

2.3. Implementation

In this section, we describe how the rules described above were added to Typer. Subsequently, we describe the library that we provide to facilitate the usage of quotient types in the development of Typer programs and proofs.

2.3.1. Axioms

Quotient types are introduced in Typer in axiomatic manner, i.e. the existence of the type former, terms formers and eliminators are simply postulated. However, the appropriate computational behaviour of the above constructs are also added to the system whenever necessary such that we do not find ourselves with stuck terms that adversely affect the computational properties of our system.

Formation. The existence of the *Quotient* type is postulated as follows:

```
Quotient : (A : Type) → (R : A → A → Type) → Type
```

Introduction. Two axioms are added, corresponding to the two introduction rules of *Quotient*.

```
Quotient_in : (A : Type) ⇛ (R : A → A → Type) ⇛ A → Quotient A R
Quotient_eq : (A : Type) ⇛ (R A → A → Type)  ⇛ (a a' : A) ⇛ R a a'
              → Eq (Quotient_in (R := R) a) (Quotient_in (R := R) a')
```

TODO: In Quotient_eq, debate the choice of not making R a a' erasable? I'm unsure if there are any benefits of having it as non-erasable

As a constructor of equalities, *Quotient_eq* does not take part in computations. As such, no special reduction rules needed to be added for it.

Elimination.

```
Quotient_elim : (A : Type) ⇛ (R : A → A → Type) ⇛ (P : Quotient A R → Type)
                ⇛ ((a : A) → P (Quotient_in a))
                → ((a a' : A) → R a a' → Heq (f a) (f a')
                ⇛ (q : Quotient A R) → P q
```

As mentioned in the previous section, the appropriate β-reduction rule is added for the eliminator. When *Quotient_elim* is passed a term of the form *Quotient_in a*, the entire *Quotient_elim* term reduces to just *f a*, ignoring the coherence proof that is passed to the function as it acts simply as a form of static information.

2.3.2. Syntaxique sugar

TODO: Add more details after things get figured out

TODO: Maybe mention experience of trying to encode QITs using syntaxic sugar, though unfortunately it didn't work out very well

Chapter 3

Quotient Effectiveness

This only works for relations that are propositions and equivalence relations (i.e. reflexive, symmetric and transitive).

3.1. Propositional Extensionality

We provide a alternative version of propositional extensionality that is slightly weaker. Consequently, the version of quotient effectiveness that we prove is also weaker.

Representation of propositions

Intuitively speaking, propositions are a type that contain no intrinsic data, it is their mere existence that counts. This is precisely the notion of proof irrelevance. In Typer, this notion is represented via erasability. We introduce a built-in data type that captures this idea.

```
type Erased (T : Type)
  | erased (t ::: T)
```

By ensuring that the witness t of the type T is erasable, this provides us with a guarantee that during elimination, the witness t can only be used in an erasable, i.e. proof irrelevant manner.

To convince ourselves that this type does indeed capture the essence of a proposition, we provide a proof of *isProp* (*Erased* ?T). We first state the definition of *isProp*:

```
isProp : (T : Type) → Type
isProp T = (x y : T) → Eq x y
```

We now the present the proof in the form of Typer code.

```
erasedIsProp : (T : Type) ⇛ isProp (Erased T)
erasedIsProp t1 t2 =
  case t1
    | erased (_ := e1) ⇒
        let
          p1 = ##DeBruijn 0 : Eq (erased (t := e1)) t1
        in
          case t2
            | erased (_ := e2) ⇒
                let
                  p2 = ##DeBruijn 0 : Eq (erased (t := e2)) t2
                in
                  Eq_trans (Eq_comm p1) p2
```

Typer does not offer the direct specialisation of case targets in branches. In other words, in the first case branch, we do not have a definitional equality between *t1* and *erased* (_ := *e1*). Instead, an equality proof between them is injected into the context, which can be accessed as the variable at deBruijn index 0. Now that we have the two equality proofs, in order to concatenate them[1], we require *erased* (_ := *e1*) and *erased* (_ := *e2*) to be equal. A priori, the two terms are not equal as *e1* and *e2* are completely arbitrary. However, Typer ignores erasable terms when checking the convertibility of terms, this is a property that was proven to be sound in [?] as convertibility is tested on extracted terms. This allows the concatenation to succeed and thus concludes the proof.

Implementation details

Propositional extensionality essentially means that if two propositions that are logically equivalent, i.e. each proposition implies the other, then we can say that they are equal. And precisely because we are postulating a new equality constructor, we have to be very careful that the resulting equality proof is indeed proof irrelevant. In other words, we need this equality proof to not have any runtime relevance. This is principally because we insist that the transport operation to be a no-op during runtime. This is the motivation behind our choice to use the *Erased* type to represent propositions, as it is a guarantee that such a type carries no useful runtime information. This implies that a transport operation from one *Erased* type to another is necessarily simply a no-op.

[1]By considering equalities as path, the application of the transitivity property of equality is conceptually similar to concatenating two paths.

In a setting with univalence, propositional extensionality is simply a theorem that may be proven, more specifically, it may be seen as a degenerate case of univalence [**?**]. This is a result that is immediate as an equivalence between logically equivalent propositions is trivial. In the setting of Typer however, propositional extensionality is implemented as an axiom with the following type signature:

```
propExt : (A : Type) ⇛ (B : Type) ⇛ (Erased A → Erased B)
          → (Erased B → Erased A) → Erased A \equiv Erased B
```

The above is a weaker notion of propositional extensionality compared to what is proposed in Cubical Agda, which has the following type signature:

```
propExt : {A B : Type} → isProp A → isProp B → (A → B)
          → (B → A) → A ≡ B
```

However, this produces an equality proof that is not proof irrelevant, as A and B are not necessarily the 'same' type. We illustrate the above with a simple example. We define two instances of the unit type which are trivially 'logically equivalent'.

```
data Unit1 : Type where unit1 : Unit1

data Unit2 : Type where
  unit2 : Unit2
```

isProp *Unit1* and isProp *Unit2* are trivially true, as is the case for all $\mathbb{1}$-types. *Unit1* $\to$ *Unit2* and *Unit2* $\to$ *Unit1* are also inhabited. Propositional extensionality thus gives us a proof of *Unit1* $\equiv$ *Unit2*. This implies that we should be able to use said proof to transport *unit1* to *unit2*, but since the two terms do not have the same runtime representation, this simply cannot be a no-op. This is in stark contrast to the version of propositional extensional that we are proposing in Typer that only accepts types that are boxed in the *Erased* type. We would have no qualms about having a proof of equality between *Erased Unit*1 and *Erased Unit*2 as terms of these two types would essentially have the same runtime representation. It is also for this precise reason that our version of propositional extensionality is weaker, as we are constrained to construct an equality proof that is not computationally relevant.

Another worthy comparison to make is with Lean, which has a Prop sort. Terms of this sort are not autorised to be used in a proof relevant manner. Its version of propositional extensionality has the following form:

```
axiom propext {a b : Prop} : (a ↔ b) → a = b
```

This is similar to what we have in Typer, except that proof irrelevance is enforced in a different manner.

3.2. Equivalence relation

First, we state what we mean when we say 'relation' to set the stage for the subsequent discussion. For our purposes, we consider binary relations, which are type formers that take two arguments of some type A. Assuming that we call the relation R and its two arguments a and a' respectively, inhabitants of the type R a a' are witnesses of this binary relation. The type signature of a binary relation has the below form:

```
R : (A : Type) ⇛ (a : A) → (a' : A) → Type
```

For a quotient to be effective, its underlying relation has to be an equivalence relation, we shall use this property in the proof in the subsequent section. Recall that an equivalence relation implies that the relation is reflexive, symmetric and transitive. For it to be reflexive, it needs to be the case that given an arbitrary $a : \mathsf{A}$, R a a must be trivially inhabited. The symmetry of the relation implies that if we are given an arbitrary witness of R a a', we need to be able to transform that into an inhabitant of R a' a. The relation is transitive if we are able to transform witnesses of R a b and R b c into witnesses of R a c[2].

In the built-in library, we define the following type as a witness that a certain relation is indeed an equivalence relation.

```
type isEquivRel (A : Type) (R : A -> A -> Type) : Type
  | equivRel (isRefl : (a : A) -> R a a)
             (isSym : (a : A) -> (b : A) -> R a b -> R b a)
             (isTrans : (a : A) -> (b : A) -> (c : A)
                        -> R a b -> R b c -> R a c)
```

3.3. Proof

Given two arbitrary elements a and a' of a base type A, a quotient is said to be effective if given a proof that the injection of a and a' into the quotient are equal, then it must be the case that R a a' is true, i.e. we are able to find an inhabitant of it. A sufficient condition for a quotient to be effective is that the underlying relation must be both a proposition and an equivalence relation.

For an arbitrary binary relation R, we can define the following:

```
ErasedR : (A : Type) ⇛ (A → A → Type) → A → A → Type
ErasedR R a b = Erased (R a b)
```

Now, we state the proof of the effectiveness of quotients in the form of Typer code

[2]The equality type fulfills these criteria and is an equivalence relation.

```
effective : (A : Type) ⇛ (R : A → A → Type)
            → isEquivRel A (ErasedR R) → (a : A) → (b : A)
            → Eq (Quotient_in (R := ErasedR R) a)
                 (Quotient_in (R := ErasedR R) b)
            → ErasedR R a b
effective = lambda A ⇛ lambda R equiv a b eq →
  let
    helper : Quotient A (ErasedR R) -> Type
    helper x =
      Quotient_rec
        (R := ErasedR R)
        (lambda c → ErasedR R a c)
        (p := lambda c d cd →
          let
            ac->ad : Erased (R a c) → Erased (R a d)
            ac->ad ac = equiv.isTrans a c d ac cd
            ad->ac : Erased (R a d) → Erased (R a c)
            ad->ac ad = equiv.isTrans a d c ad
                                      (equiv.isSym c d cd)
          in
            propExt ac->ad ad->ac)
        x
    aa=ab : Eq (Erased (R a a)) (Erased (R a b))
    aa=ab = Eq_cong helper eq
  in
    Eq_cast (p := aa=ab) (f := id) (equiv.isRefl a)
```

Compared to the more general version of the theorem we state before, here we insist that the quotient we consider be based on the erased version of some binary relation R. And instead of proving that the base relation itself holds for some a and b, we can only show that the erased version of this relation holds.

In this proof, we can see that all aspects of the sufficient condition are at play. It is mainly used to prove that for arbitrary terms a, b and c of type A, if it is the case that if *Erased* (R b c) is inhabited, then *Erased* (R a b) $\iff$ *Erased* (R a c). Intuitively speaking this makes a lot of sense, *Erased* (R b c) implies that b and c are in the same equivalence class of the quotient. Hence, if we know that a is in the same equivalence class as b, we should naturally be able to infer that a and c are also in the same equivalence class, and vice versa.

Propositional extensionality takes us even further, by allowing us to prove that *Erased* (R a b) and *Erased* (R a c) are equal since they are both mere propositions.

Examples of usage of the above theorem are in Chapter XXX (TODO: Link to eventual proof of discreteness of Rationals, if I elect to do it that is.).

Chapter 4

Examples

The addition of quotient types to Typer allows us to define certain constructs that are typically defined in set theory as quotient sets. In this chapter, we discuss some types that are now natively definable by taking the quotient of existing types in the system.

4.1. Rational numbers

To define the rational numbers as a quotient type, we have two obvious options for the base type.

The first option is as follows:

$$\begin{aligned}\mathbb{Q} &:= \mathbb{Z} \times \mathbb{N} / \sim_{\mathbb{Q}} \\ (z_1, n_1) \sim_{\mathbb{Q}} (z_2, n_2) &:= Eq\ (z_1 * (n_2 + 1))\ (z_2 * (n_1 + 1))\end{aligned}$$

A pair (z, n) represents the rational number $z/(n+1)$, this trick allows us to ensure that the denominator is non-zero.

The second option is similar to the first:

$$\begin{aligned}\mathbb{Q} &:= \Sigma_{(x,y):\mathbb{Z}\times\mathbb{Z}}.y \neq 0 / \sim_{\mathbb{Q}} \\ (x_1, y_1, \_) \sim_{\mathbb{Q}} (x_2, y_2, \_) &:= Eq\ (x_1 * y_2)\ (y_1 * x_2)\end{aligned}$$

In this definition, the base type of a quotient is a pair of integers, along with a proof that the denominator is non-zero. We chose to use this as the basis of our definition of the rationals, as it is more convenient to work with a pair of elements of the same type.

Given that the equivalence of rational numbers is captured by a relation, we do not need to normalise our rational numbers before or after each computation, instead we prove that all of our operations respect the equivalence of rational numbers. We hoped that this would give us better runtime efficiency, as opposed to more naives implementations. To make this clearer, we illustrate one possible naive implementation. To ensure that operations on the base type of the quotient type that we intend to use to represent the rational numbers respect the equivalence relation, we may be tempted to simply normalise the operands before actually carrying out any computation. This implies that if the operands were indeed equivalent, then the normalisation of the operands would have yielded two terms that are provably equal. Hence by the congruence property of equality, the operation would necessarily respect the underlying relation. However, it is clear that such an approach might possibly be expensive. In order to normalise the base type, which we could imagine to be a pair of integers, we would have to divide both integers by their greatest commmon divisor (GCD). The calculation of their GCD could in itself be more costly that the actual operation itself. Thus, this unnecessary normalisation could potentially adversely affect the runtime efficiency of our operations on rational numbers. Hence, our definitions of operations on the rational numbers do not rely on normalisation. However, this is not without its downsides as it entails heavier proof obligations, this is a topic that we shall discuss subsequently.

Typer has a built-in Integer type. Behind the scenes, this is based on big integers in OCaml, leveraging the Zarith library. Operations on the typer Integer type such as addition, multiplication etc are implemented as axioms. At runtime, the responsibilities of these functions are simply delegated to their corresponding OCaml functions to carry out the necessary computations. So far, we are unable to reason about these operations in Typer as they are merely primitives that are implemented in the host language. In the following section, we propose the addition of several axioms that serve as witnesses of certain properties of the built-in Integer type, this will be crucial when we start reasoning about operations on rational numbers.

Integer axioms

We start off by axiomatising the associativity and commutativity of the addition and multiplication operations.

```
Integer_+-comm : (x : Integer) → (y : Integer)
                 → Eq (Integer_+ x y) (Integer_+ y x)

Integer_+-assoc : (x : Integer) → (y : Integer) → (z : Integer)
                  → Eq (Integer_+ x (Integer_+ y z))
```

```
                    (Integer_+ (Integer_+ x y) z)

Integer_*-comm : (x : Integer) → (y : Integer)
                → Eq (Integer_* x y) (Integer_* y x)

Integer_*-assoc : (x : Integer) → (y : Integer) → (z : Integer)
                 → Eq (Integer_* x (Integer_* y z))
                      (Integer_* (Integer_* x y) z)
```

We also want to be able to show that multipication is distributive with respect to addition. We only add left distributivity as an axiom, as right distributivity can be easily proven as a theorem, this is also the case for the other axioms that we will describe below.

```
Integer_*DistL+ : (x : Integer) → (y : Integer) → (z : Integer)
                 → Eq (Integer_* (Integer_+ x y) z)
                      (Integer_+ (Integer_* x z) (Integer_* y z))
```

The above properties can also be stated for the subtraction operation. We also want to be able to say that we have additive inverses for all integers and that 0 is the additive identity.

```
Integer_+Linv : (x : Integer) → Eq (Integer_+ (Integer_- 0 x) x) 0

Integer_+Lid : (x : Integer) → Eq (Integer_+ 0 x) x
```

Of course, we should also be able to say that 1 is the multiplicative identity.

```
Integer_*Lid : (x : Integer) → Eq (Integer_* 1 x) x
```

Now that we have all these axioms, we can say that Integer fulfills the properties of a commutative ring. Aside from these axioms, there are some other properties that we require in order to be able to construct some useful operations on the resulting Rational type. Notably, we need to be able to say that if the product of two integers is zero, then at least one of them is zero. We formulate this in the following manner:

```
Integer_isIntegral : (x : Integer) → (y : Integer)
                    → Eq (Integer_* x y) 0 → (Eq x 0 → Void)
                    → Eq y 0
```

Here, Void is the $\mathbb{0}$ type (also known as the empty type or bottom). This axiom essentially states that if a product $x \cdot y = 0$, then $x \neq 0 \to y = 0$.

Another useful axiom to have is one that asserts that 0 is an absorbing element with respect to multiplication. In other words, multiplying by 0 always yields 0 as a result.

```
Integer_*Lzero : (x : Integer) → Eq (Integer_* 0 x) 0
```

Integer theorems

As stated in the subsequent section, we only added the 'left' version of some axioms, as their 'right' counterparts may be trivially proven as theorems based on the commutativity property of addition and multiplication.

```
Integer_+Rid : (x : Integer) → Eq (Integer_+ x 0) x
Integer_+Rid x = Eq_trans (Integer_+-comm x 0) (Integer_+Lid x)

Integer_+Rinv : (x : Integer) → Eq (Integer_+ x (Integer_- 0 x)) 0
Integer_+Rinv x = Eq_trans (Integer_+-comm x (Integer_- 0 x))
                           (Integer_+Linv x)

Integer_*Rid : (x : Integer) → Eq (Integer_* x 1) x
Integer_*Rid x = Eq_trans (Integer_*-comm x 1) (Integer_*Lid x)

Integer_*Rzero : (x : Integer) → Eq (Integer_* x 0) 0
Integer_*Rzero x = Eq_trans (Integer_*-comm x 0) (Integer_*Lzero x)

Integer_*DistR+ : (x : Integer) → (y : Integer) → (z : Integer)
                  → Eq (Integer_* x (Integer_+ y z))
                       (Integer_+ (Integer_* x y) (Integer_* x z))
Integer_*DistR+ x y z =
  Eq_trans (Integer_*-comm x (Integer_+ y z))
           (Eq_trans (Integer_*DistL+ y z x)
                     (Eq_trans (Eq_cong (lambda e →
                                           Integer_+ e (Integer_* z x))
                                        (Integer_*-comm y x))
                               (Eq_cong (lambda e →
                                           Integer_+ (Integer_* x y) e)
                                        (Integer_*-comm z x))))
```

Another useful theorem that we can prove states that the product of two non-zero integers is necessarily non-zero.

```
Integer_0-product : (x : Integer) → (y : Integer)
                    → (Eq x 0 → Void) → (Eq y 0 → Void)
                    → (Eq (Integer_* x y) 0 → Void)
```

```
Integer_0-product x y x≠0 y≠0 xy≠0 =
    y≠0 (Integer_isIntegral x y xy≠0 x≠0)
```

We will eventually also make use of the below theorem that states that the negation of an integer may be distributed over multiplication. Its 'right' counterpart may also be proven in a similar manner.

```
Integer_NegateDistL* : (x : Integer) → (y : Integer)
                       → Eq (Integer_- 0 (Integer_* x y))
                            (Integer_* (Integer_- 0 x) y)
Integer_NegateDistL* x y =
  Integer_- 0 (Integer_* x y)
  ==< Eq_cong (lambda e → Integer_- e (Integer_* x y))
              (Eq_comm (Integer_*Lzero y)) >==
  Integer_- (Integer_* 0 y) (Integer_* x y)
  ==< Eq_comm (Integer_*DistL- 0 x y) >==
  Integer_* (Integer_- 0 x) y   □
```

Implementation of Rationals

As stated before, the base type of our quotient type is essentially a pair of integers along with a proof that the second integer is non-zero. We decided to implement this as an inductive type with a single constructor.

```
type ℤ×ℤ≠0
  | inR (z₁ : Integer) (z₂ : Integer) (p : Not (Eq z₂ 0))
```

Not has the typical definition as follows:

```
Not : (T : Type) → Type
Not T = T → Void
```

The underlying relation takes on the following form, this can be proven to be an equivalence relation.

```
equalℚ : ℤ×ℤ≠0 → ℤ×ℤ≠0 → Type
equalℚ p1 p2 = Eq (p1.z1 * p2.z2) (p1.z2 * p2.z1)
```

Finally, we can state the definition of the Rationals.

```
Rational : Type
Rational = Quotient ℤ×ℤ≠0 equalℚ
```

As a sanity check, we prove that the resulting Rational type is indeed a field as it should be. Negation is the simplest operation that we can define on the rationals, it is achieved by simply negating the integer numerator. This is done via the below function.

```
negate_fst : ℤ×ℤ≠0 → ℤ×ℤ≠0
negate_fst x = inR (-1 * x.z1) x.z2 x.p
```

We also need to prove that *negate_fst* respects the underlying relation $equal\mathbb{Q}$. In other words, we need to prove the following:

```
negate_compat : (a : ℤ×ℤ≠0) → (b : ℤ×ℤ≠0) →
                equalℚ a b → equalℚ (negate_fst a) (negate_fst b)
negate_compat a b r =
    (-1 * a.z1) * b.z2
    ==< Eq_comm (Integer_*-assoc -1 a.z1 b.z2) >==
    -1 * (a.z1 * b.z2)
    ==< Eq_cong (lambda e → -1 * e) r >==
    -1 * (a.z2 * b.z1)
    ==< Integer_*-assoc -1 a.z2 b.z1 >==
    (-1 * a.z2) * b.z1
    ==< Eq_cong (lambda e → e * b.z1)
                (Integer_*-comm -1 a.z2) >==
    (a.z2 * -1) * b.z1
    ==< Eq_comm (Integer_*-assoc a.z2 -1 b.z1) >==
    a.z2 * (-1 * b.z1)   □
```

We introduced equational reasoning to facilitate the development and reading of such proofs, more details are given in Section **??** of the appendix.

Finally, the negation operation of rational numbers can be constructed like so:

```
Rational_negate : Rational → Rational
Rational_negate x =
  qcase (x : ℤ×ℤ≠0 / equalℚ)
    | Quotient_in a ⇒ Quotient_in (R := equalℚ) (negate_fst a)
    | Quotient_eq a a' r ⇒ Quotient_eq (R := equalℚ)
                                      (a := negate_fst a)
                                      (a' := negate_fst a')
                                      (negate_compat a a' r)
```

We can also define the addition operation on the rationals. As before, we start off by defining a function that operates on expressions of the base type.

```
ℤ×ℤ+ : ℤ×ℤ≠0 → ℤ×ℤ≠0 → ℤ×ℤ≠0
ℤ×ℤ+ a b = inR ((a.z1 * b.z2) + (b.z1 * a.z2))
                (a.z2 * b.z2)
                (Integer_0-product a.z2 b.z2 a.p b.p)
```

This is an operation that can be shown to be commutative with respect to the equivalence relation.

```
ℤ×ℤ+-Comm : (a : ℤ×ℤ≠0) → (b : ℤ×ℤ≠0)
              → Eq (t := Quotient ℤ×ℤ≠0 equalℚ)
                   (Quotient_in (ℤ×ℤ+ a b)) (Quotient_in (ℤ×ℤ+ b a))
ℤ×ℤ+-Comm a b =
  let
    compat =
      (a.z1 * b.z2 + b.z1 * a.z2) * (b.z2 * a.z2)
        ==< Eq_cong (lambda e → (a.z1 * b.z2 + b.z1 * a.z2) * e)
                    (Integer_*-comm b.z2 a.z2) >==
      (a.z1 * b.z2 + b.z1 * a.z2) * (a.z2 * b.z2)
        ==< Integer_*-comm (a.z1 * b.z2 + b.z1 * a.z2) (a.z2 * b.z2) >==
      a.z2 * b.z2 * (a.z1 * b.z2 + b.z1 * a.z2)
        ==< Eq_cong (lambda e → a.z2 * b.z2 * e)
                    (Integer_+-comm (a.z1 * b.z2) (b.z1 * a.z2)) >==
      a.z2 * b.z2 * (b.z1 * a.z2 + a.z1 * b.z2)   □
  in
      Quotient_eq (R := equalℚ)
                  (a  := ℤ×ℤ+ a b)
                  (a' := ℤ×ℤ+ b a)
                  compat
```

We still need to show that the ℤ×ℤ+ function itself respects the underlying relation, and since this is a binary operation, we will have to prove it for 'both sides'. In other words, we have to produce the below proofs.

```
ℚ+_feql : (a : ℤ×ℤ≠0) → (a' : ℤ×ℤ≠0) → (b : ℤ×ℤ≠0) → equalℚ a a'
          → Eq (t := Quotient ℤ×ℤ≠0 equalℚ)
                (Quotient_in (ℤ×ℤ+ a b))
                (Quotient_in (ℤ×ℤ+ a' b))
ℚ+_feql = ?

ℚ+_feqr : (a : ℤ×ℤ≠0) → (b : ℤ×ℤ≠0) → (b' : ℤ×ℤ≠0) → equalℚ b b'
          → Eq (t := Quotient ℤ×ℤ≠0 equalℚ)
                (Quotient_in (ℤ×ℤ+ a b))
                (Quotient_in (ℤ×ℤ+ a b'))
ℚ+_feqr = ?
```

The proof of ℚ+_feql is long and uninteresting and thus will not described here[1]. However, once this has been proved, its counterpart is a trivial corollary by virtue of the commutativity of ℤ×ℤ+.

```
ℚ+_feqr a b b' p =
    Eq_trans (ℤ×ℤ+-Comm a b)
             (Eq_trans (ℚ+_feql b b' a p) (ℤ×ℤ+-Comm b' a))
```

We can now assemble the above to define both the addition and subtraction of rationals as follows:

```
Rational_+ : Rational → Rational → Rational
Rational_+ a b =
  Quotient_rec2 (R := equalℚ) (S := equalℚ)
                Rational_isSet (lambda x y → Quotient_in (ℤ×ℤ+ x y))
                ℚ+_feql ℚ+_feqr a b

Rational_- : Rational → Rational → Rational
Rational_- a b = Rational_+ a (Rational_negate b)
```

Quotient_rec2 is a library function that allows us to simultaneously eliminate two quotient expressions, more details are given in **??**. Finally, we want to prove that negation of a rational produces its additive inverse. We define Rational_0 to be the additive identity using a trivial representative of its equivalence class.

```
ℤ×ℤ+-Linv : (a : ℤ×ℤ≠0) → Eq (t := Rational)
                              (Quotient_in (ℤ×ℤ+ (negate_a.z1) a))
                              Rational_0
ℤ×ℤ+-Linv a =
```

[1] Interested readers may find the proof at TODO: link

```
  let
    h : Eq ((-1 * a.z1 * a.z2 + a.z1 * a.z2) * 1)
           (a.z2 * a.z2 * 0)
    h =
      (-1 * a.z1 * a.z2 + a.z1 * a.z2) * 1
      ==< Eq_cong (lambda e → (e * a.z2 + a.z1 * a.z2) * 1)
                  (Integer_Negate≡ a.z1) >==
      ((0 - a.z1) * a.z2 + a.z1 * a.z2) * 1
      ==< Integer_*Rid ((0 - a.z1) * a.z2 + a.z1 * a.z2) >==
      (0 - a.z1) * a.z2 + a.z1 * a.z2
      ==< Eq_cong (lambda e → e + a.z1 * a.z2)
                  (Eq_comm (Integer_NegateDistL* a.z1 a.z2)) >==
      (0 - (a.z1 * a.z2)) + a.z1 * a.z2
      ==< Integer_+-comm (0 - (a.z1 * a.z2)) (a.z1 * a.z2) >==
      a.z1 * a.z2 + (0 - (a.z1 * a.z2))
      ==< Integer_+Rinv (a.z1 * a.z2) >==
      0
      ==< Eq_comm (Integer_*Rzero (a.z2 * a.z2)) >==
      a.z2 * a.z2 * 0   □
  in
    Quotient_eq (R := equal ℚ)
                (a := ℤ×ℤ+ (negate_a.z1) a)
                (a' := inR 0 1 Integer_1≠0)
                h

Rational_+Linv : (q : Rational) → Eq (Rational_+ (Rational_negate q) q)
                                     Rational_0
Rational_+Linv q =
  Quotient_elimProp (R := equal ℚ)
                    (P := lambda q →
                            Eq (Rational_+ (Rational_negate q) q)
                               Rational_0)
                    (lambda q →
                        Rational_isSet (Rational_+ (Rational_negate q) q)
                                       Rational_0)
                    ℤ×ℤ+-Linv q

Rational_+Rinv : (q : Rational) → Eq (Rational_+ q (Rational_negate q))
                                     Rational_0
```

```
Rational_+Rinv q = Eq_trans (Rational_+-Comm q (Rational_negate q))
                            (Rational_+Linv q)
```

For brevity, we do not describe the proofs for multiplication as they can be carried out in a similar fashion as above.

4.1.1. Benchmarks

In this section, we use our implementation of rational numbers to explore the runtime cost of using quotient types in Typer. Quotient types are nothing more than a means of using the type system to ensure that programmers respect a certain relation on a base type by means of proof obligations. Such proofs can be expected to be erased before the actual execution of a program, our intuition would then lead us to expect that the usage of quotient types to have a negligible runtime overhead. Hence, the tests in our benchmark were designed with the simple objective of verifying the above claim.

Since the operations on rational numbers are essentially wrappers around operations on the base type $\mathbb{Z}\times\mathbb{Z}\neq 0$, our benchmark simply compares the execution time of operations on $\mathbb{Z}\times\mathbb{Z}\neq 0$ with their `Rational` counterparts[2]. We examine the runtime cost[3] of executing the functions `Rational_negate`, `Rational_+`, `Rational_-` and `Rational_*` on 500000 pairs of rational numbers.

Operation	**Execution Time (seconds)**		**Overhead**	
	$\mathbb{Z}\times\mathbb{Z}\neq 0$	**Rational**	**seconds**	**%**
Negation	1.82	2.53	0.71	39.0
Addition	3.18	4.82	1.64	51.6
Subtraction	4.10	6.26	2.16	52.7
Multiplication	2.97	4.59	1.62	54.5

Table 4.1. The runtime cost of operations on rational numbers with and without the use of quotient types.

We observe that there is indeed a non-negligible overhead caused by the usage of quotient types, which we will now try to justify. Without loss of generality, we discuss the addition operation. Once the $\mathbb{Z}\times\mathbb{Z}+$ function that handles the addition of $\mathbb{Z}\times\mathbb{Z}\neq 0$ and its `Rational` counterpart, `Rational_+` have undergone erasure, the resulting functions are extensionally

[2]The code for this may be found in the `rational_benchmark.typer` file.

[3]The tests were executed on the author's 2018 Macbook Pro with a quadcore 2.3GHz Intel Core i5 processor with 8GB of RAM.

equal. By definition of erasure [**?**], all erasable arguments are removed, hence all proof objects required by the quotient type will be absent in the erased versions of the above functions. The two erased functions are nevertheless not completely equal, as we shall see by examining the Elexp (erased lambda expression) [**?**] of `Rational_+`:

```
(lambda (a) → (lambda (b) →
  (Quotient_rec2 (lambda (x) →
                    (lambda (y) →
                        (Quotient_in (ℤ×ℤ+ x y)))) a b)))
```

Through a series of η-reductions and β-reductions, we can see that the above expression is equivalent to $\mathbb{Z}\times\mathbb{Z}+$. This implies that with an appropriate optimisation phase, Elexps containing quotient primitives could be simplified to completely neutralise the overhead of quotient types. We leave this optimisation for a future work. We also note that addition and multiplication unsurprisingly have a similar constant overhead due to how similarly they are defined. Since negation is a unary operation operation, it was defined directly using `Quotient_rec` and thus has a lesser overhead than the other binary operations. We also defined subtraction on the basis of addition combined with subtraction, so its greater overhead is well justified.

4.2. Multiset

A multiset is an abstract data type that may be seen as a generalisation of sets. In sets, multiple occurrences of the 'same' item are ignored, whereas multisets keep track of repeated occurrences of items. One way of defining such a data type (albeit not a very efficient one) is by taking a quotient of the list data type with an equivalence relation such that all permutations of a given list are equivalent.

To that end, we wish to define a relation `ListPerm` l_1 l_2 such that l_1 and l_2 are permutations of each other. The relation would thus need a constructor of the following form:

```
swapPerm : (x : A) → (y : A) → (l : List A)
           → ListPerm ?A (cons (y (cons x l))) (cons x (cons y l))
```

The relation is symmetric as it is, however, for it to be an equivalence relation, we also require the relation to be reflexive and transitive. We thus need to complement it with additional constructors, this is analogous to constructing the reflexive and transitive closure of a relation.

For the relation to be reflexive, we would require the following constructors such that `ListPerm` is closed under the base constructors of the List datatype.

```
nilPerm : ListPerm ?A nil nil

consPerm : (x : ?A) → (l1 : List ?A) → (l2 : List ?A)
           → (ListPerm ?A l1 l2) → (ListPerm ?A (cons x l1) (cons x l2))
```

Finally, the following addition makes the relation transitive:

```
transPerm : (l1 : List ?A) → (l2 : List ?A) → (l3 : List ?A)
            → (ListPerm ?A l1 l2) → (ListPerm ?A l2 l3)
            → (ListPerm ?A l1 l3)
```

The manner in which the `ListPerm` relation has been presented so far suggests that it would most appropriately be defined as a GADT. Typer does not have built-in support for GADTs, however, we draw inspiration from [**?**] to encode such a GADT as an inductive type with explicit equality proofs.

```
ListPerm : (A : Type_ ?) → List A → List A → Type_ ?
type ListPerm (l ::: TypeLevel) (A : Type_ l) (l1 : List A) (l2 : List A) : Type_ l
  | nilPerm (p1 :: Eq l1 nil) (p2 :: Eq l2 nil)
  | consPerm (x : A) (l1' : List A) (l2' : List A) (ListPerm A l1' l2')
             (p1 :: Eq l1 (cons x l1')) (p2 :: Eq l2 (cons x l2'))
  | swapPerm (x : A) (y : A) (l : List A)
             (p1 :: Eq l1 (cons x (cons y l)))
             (p2 :: Eq l2 (cons y (cons x l)))
  | transPerm (l1' : List A) (l2' : List A) (l3' : List A)
              (ListPerm A l1' l2') (ListPerm A l2' l3')
              (p1 :: Eq l1 l1') (p2 :: Eq l2 l3')
```

With that, we may now define the `Multiset` type former.

```
Multiset : Type_ ? → Type_ ?
Multiset A = Quotient (List A) (ListPerm A)
```

An empty multiset is simply the injection of the `nil` List into the Quotient.

```
Multiset_empty : Multiset ?A
Multiset_empty = Quotient_in nil
```

Insertion is merely the application of the `cons` constructor of List with the help of `consPerm` that we previously defined.

```
Multiset_cons : ?A → Multiset ?A → Multiset ?A
Multiset_cons = lambda _ A ⇛ lambda a s →
  qcase (s : List A / ListPerm A)
```

```
    | Quotient_in l ⇒ Quotient_in (R := ListPerm A) (cons a l)
    | Quotient_eq l l' r ⇒ Quotient_eq (R := ListPerm A)
                                       (a := cons a l)
                                       (a' := cons a l')
                                       (consPerm a l l' r (_ := Eq_refl)
                                                          (_ := Eq_refl))
```

Other useful operations on the Multiset datatype may be defined, such as `Multiset_append`, `Multiset_length`, `Multiset_mem` etc. The first two may be defined in a routine manner, hence we briefly describe the definition `Multiset_mem` as it is more interesting. This function determines if a certain term of type `A` is a member of a multiset. Its type is as follows:

```
Multiset_mem : (A : Type) ⇒ Discrete A → A → Multiset A → Bool
```

In order for this function to work, the underlying type `A` must be discrete, i.e. the equality between two terms of this type must be decidable. We define `Any` to be a witness that a certain predicate is valid for some element in a given list. We then define a predicate `isMember` such that `isMember a l` means that `a` is a member of the list `l`.

```
Any : (A : Type) -> (P : (a : A) -> Type) -> List A -> Type
type Any (A : Type) (P : ((a : A) -> Type)) (l : List A) : Type
  | here (x : A) (xs : List A) (p ::: P x) (eq ::: Eq l (cons x xs))
  | there (x : A) (xs : List A) (¬p ::: Not (P x)) (Any A P xs)
          (eq ::: Eq l (cons x xs))

isMember : (A : Type) => A -> List A -> Type
isMember = lambda A => lambda x l -> Any A (lambda x' -> Eq x x') l
```

We need to prove a lemma that says that if `isMember a` l_1 is valid, then for some other l_2 that is a permutation of l_1, `isMember a` l_2 is also valid, we omit this proof (TODO: Consider putting this in the appendix, it's long). Next, we just need to define `isMemberDec`, whose details we shall once again omit. Finally, we may proceed to define `Multiset_mem`.

```
isMemberDec : (A : Type) ⇒ Discrete A → (a : A) → (l : List A)
              → Decidable (isMember a l)
isMemberDec = ...

Multiset_mem : (A : Type) ⇒ Discrete A → A → Multiset A → Bool;
Multiset_mem disc a s =
  let
    Dec2Bool : Decidable ?A → Bool
    Dec2Bool d = case d
```

```
      | yes => true
      | no => false

    compat : (l1 : List A) → (l2 : List A) → ListPerm A l1 l2
             → Eq (Dec2Bool (isMemberDec disc a l1))
                  (Dec2Bool (isMemberDec disc a l2))
    compat l1 l2 r = ...
  in
    qcase (x : List A / ListPerm A)
        | Quotient_in l ⇒ Dec2Bool (isMemberDec disc a l)
        | Quotient_eq a a' r ⇒ compat a a' r
```

Another useful operation that one may find useful to have is one that counts how many elements of the multiset are equal to a certain term of type `A`.

4.3. Propositional Truncation

We can encode propositional truncation as a quotient type. The required properties are:

- A true $\to \|A\|$ true and $\|A\|$ is a subsingleton (exists unique)
- For some P that is a subsingleton and some $f : A \to \mathsf{P}$, there exists a function $g : \|A\| \to P$ s.t. $\forall a : A \,.\, g(a^*) = f(a)$, where $a^* \in \|A\|$.

The encoding is simple, we simply define the following type former along with its term former:

```
PropTrunc : (P : Type) → Type
PropTrunc P = Quotient P (λ p1 p2 → Unit)

inPropTrunc : (P : Type) ⇛ P → PropTrunc P
inPropTrunc p = Quotient_in (R := λ p1 p2 → Unit) p
```

The choice of R here essentially implies that every term of the base type P shall be equated in the resulting quotient type. This is witnessed by the following function:

```
squash : (P : Type) ⇛ (x : PropTrunc P) → (y : PropTrunc P) → Eq x y
squash = λ P ⇛ λ x y →
  let
    rel : P → P → Type
    rel _ _ = Unit
  in
```

```
    elimProp2 (R := rel) (S := rel)
              (P := λ x y → Eq x y)
              (λ x y → Quotient_trunc (R := rel) (x := x) (y := y))
              (λ a a' → Quotient_eq (R := rel)
                                    (a := a) (a' := a') unit)
              x y
```

inPropTrunc and *squash* combined imply that the first property is satisfied. Next, we state the elimination principle of the propositional truncation of some type P.

```
propTruncRec : (A P : Type) ⇛ isProp P → (A → P) → PropTrunc A → P
propTruncRec prop f a = elimProp (R := λ _ _ → Unit)
                                 (P := λ _ → P)
                                 (λ _ → prop)
                                 f a
```

By partially applying a proof that P is a proposition and a function of type $A \to P$ to *propTruncRec*, we obtain the second property.

4.4. Elimination (Special cases)

The elimination of quotients involves numerous proof obligations, and this quickly gets tedious especially if we are eliminating multiple quotient expressions at once. We provide a library that aims to reduce repetitive boilerplate in user code.

rec2

We start off by describing *Quotient_rec2*, which serves to eliminate two arbitrary quotient expressions.

```
rec2 : (A B C: Type_ ?) ⇛
       (R : A → A → Type_ ?) ⇛
       (S : B → B → Type_ ?) ⇛
       (C_isSet : isSet C) →
       (f : A → B → C) →
       ((a : A) → (b : A) → (c : B) → R a b → Eq (f a c) (f b c)) →
       ((a : A) → (b : B) → (c : B) → S b c → Eq (f a b) (f a c)) →
       Quotient A R → Quotient B S → C
rec2 =
  λ _ _ _ _ _ A B C R S ⇛
    λ C_isSet f feql feqr →
      Quotient_rec (R := R)
```

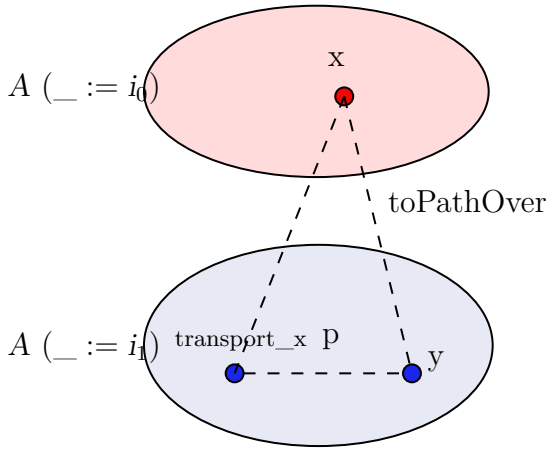
```
(λ a → λ b →
  Quotient_rec (R := S) (f a) (p := feqr a) b)
(p := λ a a' r →
  let
    eqf : (b : B) → Eq (f a b) (f a' b)
    eqf b = feql a a' b r
    p : (x : Quotient B S) →
        isProp (Eq (Quotient_rec (f a) (p := feqr a) x)
                   (Quotient_rec (f a') (p := feqr a') x))
    p x = C_isSet (Quotient_rec (f a) (p := feqr a) x)
                  (Quotient_rec (f a') (p := feqr a') x)
    compat : (x : Quotient B S) →
             (Eq (Quotient_rec (f a) (p := feqr a) x)
                 (Quotient_rec (f a') (p := feqr a') x))
    compat x = elimProp (R := S)
                        (P := λ x →
                          (Eq (Quotient_rec (f a) (p := feqr a) x)
                              (Quotient_rec (f a') (p := feqr a') x)))
                        p eqf x
  in
    Eq_funext (f := Quotient_rec (f a) (p := feqr a))
              (g := Quotient_rec (f a') (p := feqr a'))
              compat)
```

The function itself does the only logical thing it could do, it uses Quotient_rec to eliminate the first quotient, and then it makes another nested call to Quotient_rec to eliminate the second one. Essentially, the result of the elimination of the first quotient is a function that takes another quotient as an argument and then eliminates that. This is the intution behind the requirement that the output type C be a Set. In other words, this enables us to prove that for two quotients of type *Quotient A R*, we are indeed producting two output functions that are equal. We also made use of *elimProp*, a function that we shall describe next.

elimProp

elimProp is a function that simplifies the elimination of a quotient when the target type is a Prop as was introduced in section 3.1. We will see that the nature of the function f has no bearing on our proof obligations, as they can be discharged in a very systematic way by leveraging the fact that output type is a Prop. First, we need to proof a lemma, *toPathOver*.

Suppose that we have a type former $A : I \Rrightarrow \mathsf{Type}$ and two terms x and y of types $A\ (\_ := i_0)$ and $A\ (\_ := i_1)$ respectively, we get the following diagram.



The circles represent the two types and points in the circle are elements of the corresponding types. With our definition of equality, we can trivially construct an equality between $A\ (\_ := i_0)$ and $A\ (\_ := i_1)$. Now, given that the two circles are 'equal', intuitively speaking, every point in one circle should have a counterpart in the other circle, and this can be obtained transporting a point along the equality between the two types. The result of transporting x to the type $A\ (\_ := i_1)$ is a point that we shall call *transport_x*. Additionally, suppose also that we have an equality between *transport_x* and y, this is indicated by the path p between the two points in the diagram. Given that x is transported to *transport_x*, and that *transport_x* is equal to y, we also want to be able to say something about the relationship between x and y. Since they are not of the same type, all we can do is construct a heterogenous equality between them, and this is precisely what the function *toPathOver* seeks to do.

```
toPathOver : (A : I ⇛ Type) ⇛ (x : A (_ := i0)) ⇒ (y : A (_ := i1))
             ⇒ Eq (Eq_cast (p := Eq_eq (f := A)) (f := id) x) y
             → Heq x y;
toPathOver = λ _  A ⇛ λ x y ⇒ λ p →
  let
    l : Heq x (Eq_cast (p := Eq_eq (f := A)) (f := id) x);
    l = Heq_eq (t := A)
               (f := λ i ⇛
                     I_transp (A := (λ j ⇛ A (_ := I_meet i j)))
                              (r := I_not i) x);
  in
    Eq_cast
        (x := (Eq_cast (p := Eq_eq (f := A)) (f := id) x))
```

```
        (y := y)
        (p := p) (f := λ y' → Heq x y') l;
```

We construct a heterogenous equality between x and *transport_x* and we concatenate it with the equality proof between *transport_x* and y to complete the proof.

```
elimProp : (A : Type_ ?) ⇛ (R : A → A → Type)
           ⇛ (P : Quotient A R → Type)
           ⇛ (prop : (x : Quotient A R) → isProp (P x))
           → (f : (x : A) → P (Quotient_in x))
           → (x : Quotient A R) → P x
elimProp = λ _ _ _ A R P ⇛ λ prop f →
  Quotient_elim
    (R := R) (P := P) f
    (p := λ a a' r →
      let
        a=a' : Eq (t := Quotient A R) (Quotient_in a) (Quotient_in a')
        a=a' = Quotient_eq (R := R) (a := a) (a' := a') r
        fa=fa' : Heq (f a) (f a')
        fa=fa' = toPathOver
                   (A := λ i ⇛ P (Eq_uneq (p := a=a') (i := i)))
                   (prop (Quotient_in a')
                         (Eq_cast (p := a=a') (f := P) (f a))
                         (f a'))
      in
        fa=fa')
```

`toPathOver` does the heavy lifting in this proof by allowing us to leverage `isProp` $(P\ x)$ to easily produce a proof of $Heq\ (f\ a)\ (f\ a')$. We can also iterate this to define $\texttt{elimProp}_2$, $\texttt{elimProp}_3$ etc. In general, we can define $\texttt{elimProp}_n$ in terms of $\texttt{elimProp}_{n-1}$ and `elimProp`. To illustrate this, we define $\texttt{elimProp}_2$.

```
elimProp2 : (A B : Type)
            ⇛ (R : A → A → Type) ⇛ (S : B → B → Type)
            ⇛ (P : Quotient A R → Quotient B S → Type)
            ⇛ (prop : (x : Quotient A R) → (y : Quotient B S)
                      → isProp (P x y))
            → (f : (x : A) → (y : B) → P (Quotient_in x) (Quotient_in y))
            → (x : Quotient A R) → (y : Quotient B S) → P x y
elimProp2 = λ _ _ _ _ _ A B R S P ⇛ λ prop f →
  elimProp (P := λ x → (y : Quotient B S) → P x y)
```

```
(λ x → isPropΠ (B := P x) (prop x))
(λ a → elimProp (P := P (Quotient_in a))
                (prop (Quotient_in a)) (f a))
```

TODO Consider defining a macro that defines $\texttt{elimProp}_n$, i.e. the general case.

Chapter 5

Holy Trinity?

There is an intricate relationship between logic, programming (type theory) and category theory, as was proposed by Harper (TODO: link his article https://existentialtype.wordpress.com/2011/03/27/the-holy-trinity/). Everything that exists in each of these domains necessarily has an interesting analogue in the other two. In this chapter, we discuss quotients from the perspective of logic and category theory. In some cases, this will motivate the 'discovery' of some interesting properties of the Quotient type.

5.1. Relationship to Logic

TODO: Figure out what could be informative to say here. If this doesn't work out, then ditch this section and rename the chapter or something

5.2. Relationship to Category Theory

Numerous constructs in type theory have equivalents in category theory. To set the stage for this, we shall do a very short primer on category theory. For a more detailed introduction, interested readers are directed to other works such as Awodey's textbook [**?**].

5.2.1. Introduction

Category theory is built on merely a few foundational constructs, these alone give rise to a lot of interesting structures and properties. First, we postulate the existence of categories that contain objects. Next, we say that objects can be related to other objects via morphisms (also known as arrows, these two terms shall be used interchangeably in the following discussion). Assuming that readers are familiar with programming and/or type theory, one could think

of categories as types, objects as terms of a particular type, and morphisms as functions. Every object has an identity arrow that relates it with itself.

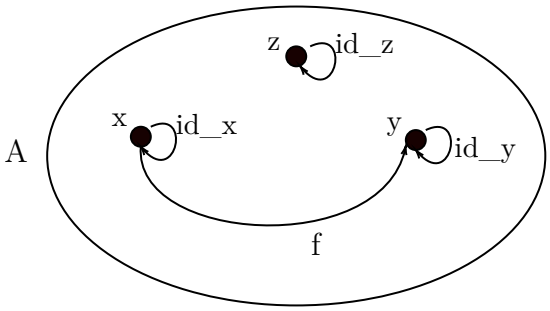


Fig. 5.1. Example of a category A with objects x, y and z with a morphism f that goes from x to y

At the heart of category theory, we have the notion of composition. Suppose that we have an arrow from A to B, and another arrow from B to C, then there is necessarily another arrow that goes directly from A to C. We call this the composition of the first two arrows. This is of course analogous to function composition in programming.

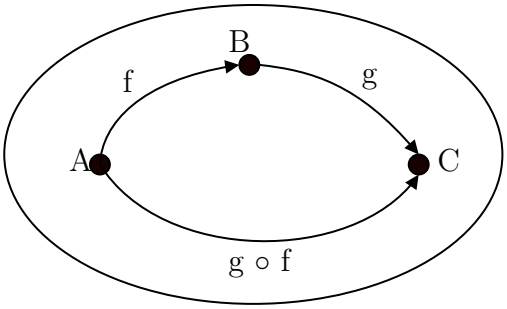


Fig. 5.2. The composition of two morphisms f and g. The identity arrows are not explicitly shown as we can take their existence for granted.

To conclude our short introduction to category theory, we would like to present the category theoretic equivalent of a ubiqituous construct in type theory, i.e. products (also known as tuples). One should be familiar with the notion of products being defined as pairs of elements, however in category theory, we have to define such objects in terms of its morphisms that relate it to other objects. If some object C were to be the product of objects A and B, then we require that it has two morphisms that behave as the first and second projections out of the product. In other words, we require the two morphisms $p : C \to A$ and $q : C \to B$.

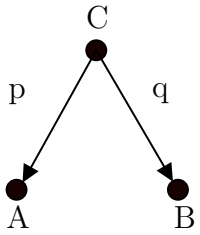


Fig. 5.3. A candidate product of A and B with its projection morphisms.

However, we notice that this is far too general of a definition since it is likely that there are many objects that fulfil these criteria. Here, we introduce the notion of a universal property which allows us to identify 'the best product'. The best product is the one that is not too general, and yet not too restrictive, it is 'just right'. More formally, the product of A and B is an object $A \times B$ with the morphisms $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$. Additionally, for any other object C with morphisms $p : C \to A$ and $q : C \to B$, there exists a unique morphism f such that $p = \pi_1 \circ f$ and $q = \pi_2 \circ f$. In other words, we require that the following diagram commute:

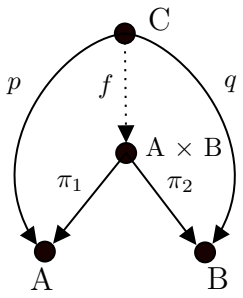


Fig. 5.4. Universal mapping property of the product

Another way of saying the above is that h factorises p and q. The universal product, if it exists, is unique up to isomorphism. Products are not guaranteed to exist in every category. In the category of sets, the universal product is none other than the Cartesian product.

5.2.2. Coequaliser

We now introduce the notion of a coequaliser. We shall see later that Quotient acts indeed as a coequaliser, or rather coequalisers are generalisation of quotients. For two arrows $f, g : A \to B$, a coequaliser is an arrow $q : B \to Q$ such that $qf = qg$ (we use juxtaposition to imply composition). The universal property of the coequaliser then dictates that for any other Z and $z : B \to Z$, if $zf = zg$, then there must exist a unique $u : Q \to Z$ such that $uq = g$. This allows us to call q **the** coequaliser of f and g.

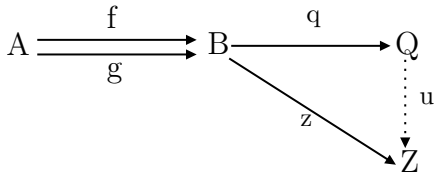


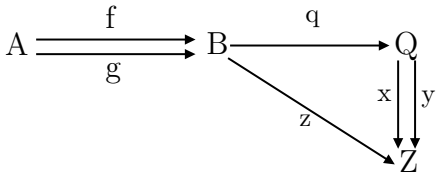
Fig. 5.5. Illustration of a coequaliser

Proposition In any category, if $q : B \to Q$ is a coequaliser of a pair of arrows $f,\ g : A \to B$, then q is an epimorphism.

Proof.

We first state the definition of an epimorphism. An epimorphism is a morphism $f : X \to Y$ such that for all objects C and morphisms $g_1, g_2 : Y \to Z$, if $g_1 f = g_2 f$ then $g_1 = g_2$.

Now, we consider the below diagram



in which q is the coequaliser of f and g. Supposing that $xq = yq$, we need to show that $x = y$. We have that $z = xq = yq$, implying that $zf = xqf = xqg = zg$. By the universal property of the coequaliser, there exists a unique u such that $z = uq$. Since $z = xq$ and $z = yq$, it must be the case that $u = x = y$. Hence, q is epic.

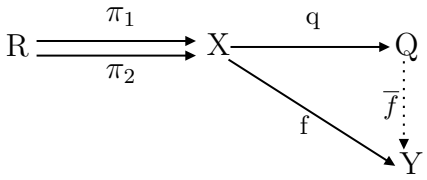
Suppose that R is a pair of some set X that is related by some equivalence relation, such that there are morphisms $\pi_1, \pi_2 : R \to X$ that act as projections out of the pair. Then, the injection into the quotient, $q : X \to X/R$ is an coequaliser of π_1 and π_2. Its counterpart in Typer is the function Quotient_in. Since the coequaliser is epic, this implies that Quotient_in is surjective, as can be proven within Typer. When we say that a function $f : X \to Y$ is surjective, we are essentially saying that $\forall y \in Y.\ \exists x \in X.\ f(x) = y$. Note that we should represent this by a weak sum, i.e. we do not require a concrete witness of x.

```
type SurjectiveQuotientProof (A : Type) (R : A -> A -> Type)
                             (x : Quotient A R) : Type
  | surjectiveQuotientProof (a : A) (Eq (Quotient_in (R := R) a) x);
```

We define the above type, intending to return the propositional truncation of it as a proof of the surjectivity of Quotient_in. The proof goes as follows:

```
Quotient_in_surjective : (A : Type) ⇛ (R : A → A → Type)
                       ⇛ (x : Quotient A R)
                       → PropTrunc (SurjectiveQuotientProof A R x)
Quotient_in_surjective = lambda A R ⇛
  elimProp (λ x → squash (P := SurjectiveQuotientProof A R x))
           (λ a → inPropTrunc (surjectiveQuotientProof a Eq_refl))
```

Additionally, if we have an $f : X \to Y$ such that it respects the underlying equivalence relation, i.e. $f \circ \pi_1 = f \circ \pi_2$, then there exists a unique function $\overline{f} : X/R \to Y$. For $x \in X$, we have that $f(x) = \overline{f}(q(x))$. The equivalent of $\overline{f}$ in Typer is the function Quotient_elim f $(p := p)$ where p is an appropriate proof that f respects the relation. This also justifies the reduction rule of Quotient_elim.



We can also phrase the universal property of Quotient in a type-theoretic way as follows:

$$(X/R \to Y) \simeq \Sigma\ (X \to Y)\ (\lambda\ f \to (x\ y : X) \to R\ x\ y \to f\ x \equiv f\ y)$$

This property may be proven within Typer itself. (TODO: Actually make this work and direct readers to it).

5.2.3. Initial object of quotient set algebras

Ok maybe let's scratch this, this isn't super interesting.

Chapter 6

Related works

6.1. HITs

Higher inductive types (HITs) are a generalisation of inductive types that has been popularized by homotopy type theory [**?**]. Aside from the usual definitions of constructors, HITs allow the definition of path constructors, i.e. equalities between terms of the inductive type. Additionally, we are allowed to define paths between paths. In other words, in a system that supports HITs, we could simply define quotient types as a HIT. For instance, here is how one might do it in Cubical Agda.

```
data _/_ {l l'} (A : Type l) (R : A → A → Type l') : Type (l-max l l')
   where
   [_] : (a : A) → A / R
   eq/ : (a b : A) → (r : R a b) → [ a ] ≡ [ b ]
   trunc/ : (a b : A / R) → (p q : a ≡ b) → p ≡ q
```

In such languages, the elimination of HITs is done in the same manner in which we typically eliminate inductive types, i.e. with pattern matching. Path constructors also need to be handled during pattern matching. Explicit side conditions are then added to ensure that the elimination of path constructors is coherent with the elimination of other constructors. To illustrate the above, we describe some HITs that are common in the context of synthetic homotopy theory.

```
data S¹ : Type where
  base : S¹
  loop : base ≡ base
```

The S^1 type is a synthetic way of representing a circle (also known as a 1-sphere). A reasonable way of describing a circle would be to say that it consists of a base point and

a continuous line that joins the base point to itself. In this HIT, we have the *base* as a normal constructor of the type. But more curiously, we also have a path constructor *loop* that identifies *base* with itself. By visualising the type, we see that it indeed ressembles a circle.

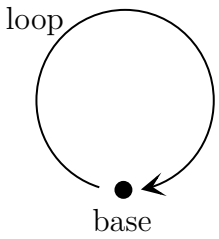


Fig. 6.1. Visualisation of S^1

To really show how HITs shine, we briefly describe the definition of *double* that is a function from S^1 to S^1 that sends the *loop* to the concatenation of two loops.

```
double : S¹ → S¹
double base = base
double (loop i) = (loop · loop) i
```

Elimination is done in cleaner manner when done using pattern matching. Aside from that, we get defitional equalities even for path constructors.

TODO: Wanted to show an example of HIT-style elimination of a quotient, but I didn't find anything worth showing for now.

6.2. QITs and Quotient Haskell

Points:

Quotient inductive types (QITs) are essentially HITs that are set-truncated. This essentially provides us with the full range of expressiveness of HITs without higher-dimensional paths. In other words, we are not allowed to define paths between paths. Indeed, such paths are not typically used in day-to-day programming tasks.

Compare to Quotient Haskell and cite examples of types that are convenient to define with QITs but are 'tedious' to express as our Quotient type.

Requires significantly more work to accomplish, but approximately the same expressive power as quotient types (how true is this!?)? Or do we draw the line at "in practice, it has comparable expressivity"? The implementation of QITs would negatively impact the modularity of inductive types.

6.3. Quotient by normalisation

When we take the quotient of a type/set based on an equivalence relation, we are essentially partitioning the elements of the type into their respective equivalence classes. Elements of the same equivalence class are then identified in the resulting quotient type/set.

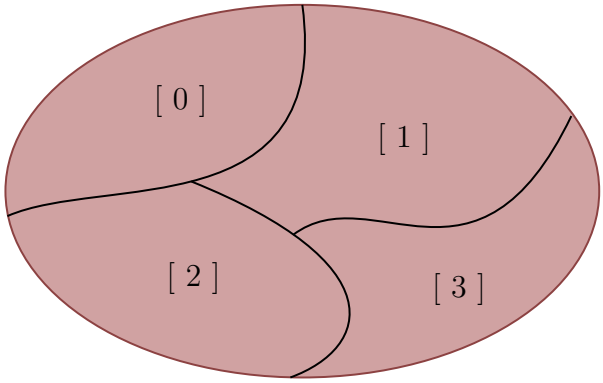


Fig. 6.2. Equivalence classes of $\mathbb{Z}$ under the relation $x \equiv y \pmod 4$

An alternative way of defining a quotient on type A under a relation R is to construct a function $nf : A \to A$ such that $nf\ a$ is the canonical representative of the equivalence class that a belongs to. This approach is comparable to [**?**]. In other words, for $x, y : A$, Eq (Quotient_in x) (Quotient_in y) $\iff$ Eq ($nf\ x$) ($nf\ y$). To represent this in our system, we can simply treat this as a quotient under the relation $R\ x\ y =$ Eq ($nf\ x$) ($nf\ y$).

To simplify the utilisation of this class of quotients in Typer programs, we define the following helper functions:

```
Qnorm : (A : Type) → (nf : A → A) → Type
Qnorm A nf = Quotient A (λ x y → Eq (nf x) (nf y))

Qnorm_in : ?A → Qnorm ?A ?nf;
Qnorm_in a = Quotient_in a;

Qnorm_elim : Qnorm ?A ?nf → (?A → ?B) → ?B
Qnorm_elim q f = Quotient_rec (f ∘ nf)
                              (p := λ x y r → Eq_cong f r)
                              q

Qnorm_eq : (a a' : ?A) → Eq (nf a) (nf a') → Eq (Qnorm_in a) (Qnorm_in a')
Qnorm_eq a a' eq = Quotient_eq (a := a') (a' = a') eq
```

We could have alternatively introduced quotients to Typer by implementing the above as a built-in construct instead of what we have implemented currently. However, as we have demonstrated above, this flavour of quotients is a strict subset of our implementation. Implying that this is less expressive and flexible compared to our implementation. Indeed, it is not always possible to define a normalisation function for a certain equivalence relation. Consider the Multiset example that was presented in section 4.2, if we would like to normalise a set of elements, a reasonable approach would be to sort them, but this is not ideal as this would imply that the base type of the set needs to have some sort of order. However, this approach also has its advantages, notably in Courtieu's system [**?**], Eq $(nf\ x)$ $(nf\ y) \Rightarrow$ Eq (Quotient_in x) (Quotient_in y) actually deals with definitional equivalences instead of propositional equivalences as is done with $Qnorm\_eq$.

Quotients based on normalisation may actually be represented in Typer without extending the language. Indeed, this can be done in any language that supports coproducts, as is suggested in [**?**]. Similarly, Courtieu's implementation may also be translated to the Calculus of Inductive Constructions [**?**]. We now describe the aforementioned construction in Typer, and then provide a proof that it fulfils the universal property of a quotient.

We define a dependent pair as follows (this is why we require coproducts):

```
type Sigma (A : Type) (B : A → Type)
  | sigma (fst : A) (snd : B fst);
```

Now, we can define the type Qnorm for some type A and normalisation function $r : A \to A$ as a dependent pair that contains some term of type A and a proof that it has been normalised. This of course necessitates the idempotency of the normalisation function.

```
Qnorm : (A : Type) → isSet A → (r : A → A)
        → (i : (x : A) → Eq (r (r x)) (r x)) → Type
Qnorm A p r i = Sigma A (λ (x : A) → Eq (r x) x)
```

The injection into Qnorm is simply the construction of a dependent pair containing a term that has been normalised and a proof of its normalisation. This is in contrast to the construction that was described previously with Quotient which applied the normalisation upon the elimination of a quotient, whereas here, normalisation immediately occurs upon the construction of a term of type Qnorm.

We can prove that the equality of Qnorm terms is characterised by the equality between the normalised forms of the underlying terms. As usual, we require isSet A to prove this equality. A simple invocation of the $\Sigma\equiv$_prop lemma completes the proof.

```
Qnorm_eq : (A : Type) ⇛ (x y : A) ⇒ (p : HoTT_isSet A)
           ⇒ (r : A → A) ⇒ (i : (x : A) → Eq (r (r x)) (r x))
           ⇒ Eq (r x) (r y) → Eq (Qnorm_in (p := p) (r := r) (i := i) x)
                                 (Qnorm_in (p := p) (r := r) (i := i) y)
Qnorm_eq = λ A ⇛ λ x y p r i ⇒ λ rx≡ry →
  Σ≡_prop (B := λ x → Eq (r x) x)
           (λ x → p (r x) x)
           (sigma (B := λ x → Eq (r x) x) (r x) (i x))
           (sigma (B := λ x → Eq (r x) x) (r y) (i y))
           rx≡ry
```

We can easily prove the reverse of this too, i.e. if we have that Eq (Qnorm_in x) (Qnorm_in y), then Eq $(r\ x)$ $(r\ y)$ is immediate by taking the first projection of the pairs.

Finally, we can also show that Qnorm fulfils the universal property of a quotient. In other words, we want show the following equivalence:

$$(\mathsf{Qnorm}\ A\ p\ r\ i \to B) \simeq \Sigma\ (A \to B)\ (\lambda g \to (x\ y : A) \to r\ x \equiv r\ y \to g\ x \equiv g\ y)$$

The proof itself is not very interesting, hence we do not describe it here. (TODO: Link to proof). We note however that the reverse direction of this equivalence gives us a means of eliminating Qnorm, as was the case for Quotient.

TODO: Mention how we could also have an NF of type $A \to B$, i.e. doesn't have to be the same type as the base type, but obviously this doesn't work with this precise construction.

References

[1] M. Sulzmann, M. M. Chakravarty, S. P. Jones, and K. Donnelly, "System f with type equality coercions," in *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pp. 53–66, 2007.

[2] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[3] M. Bezem, T. Coquand, and S. Huber, "A model of type theory in cubical sets," in *19th International conference on types for proofs and programs (TYPES 2013)*, vol. 26, pp. 107–128, 2014.

[4] B. Hewer and G. HUTTON, "Quotient haskell," 2023.

[5] N. Li, *Quotient types in type theory*. PhD thesis, University of Nottingham, 2015.

[6] P. Courtieu, "Normalized types," in *Computer Science Logic* (L. Fribourg, ed.), (Berlin, Heidelberg), pp. 554–569, Springer Berlin Heidelberg, 2001.

[7] D. R. Licata and G. Brunerie, "A cubical approach to synthetic homotopy theory," in *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, pp. 92–103, IEEE, 2015.

[8] M. Hedberg, "A coherence theorem for martin-löf's type theory," *Journal of Functional Programming*, vol. 8, no. 4, pp. 413–436, 1998.

[9] A. Chlipala, *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2022.

[10] T. U. F. Program, "Homotopy type theory: Univalent foundations of mathematics," tech. rep., Institute for Advanced Study, 2013.

[11] A. Vezzosi, A. Mörtberg, and A. Abel, "Cubical agda: A dependently typed programming language with univalence and higher inductive types," *Journal of Functional Programming*, vol. 31, p. e8, 2021.

[12] P. Martin-Löf, "An intuitionistic theory of types: Predicative part," in *Studies in Logic and the Foundations of Mathematics*, vol. 80, pp. 73–118, Elsevier, 1975.

[13] M. Abbott, T. Altenkirch, and N. Ghani, "Containers: Constructing strictly positive types," *Theoretical Computer Science*, vol. 342, no. 1, pp. 3–27, 2005.

[14] P. Martin-Löf, "Constructive mathematics and computer programming," in *Studies in Logic and the Foundations of Mathematics*, vol. 104, pp. 153–175, Elsevier, 1982.

[15] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, "Cubical type theory: a constructive interpretation of the univalence axiom," 2016.

[16] M. Hofmann and T. Streicher, "The groupoid interpretation of type theory," *Twenty-five years of constructive type theory (Venice, 1995)*, vol. 36, pp. 83–111, 1998.

[17] T. Streicher, *Investigations into intensional type theory*. PhD thesis, Habilitationsschrift, Ludwig-Maximilians-Universität München, 1993.

[18] B. Barras and B. Bernardo, "The implicit calculus of constructions as a programming language with dependent types," in *Foundations of Software Science and Computational Structures: 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 11*, pp. 365–379, Springer, 2008.

[19] M. Sozeau and N. Tabareau, "Univalence for free," 2013.

[20] S. Monnier, "Typer: Ml boosted with type theory and scheme," *Journées Francophones des Langages Applicatifs*, pp. 193–208, 2019.

[21] P. Delaunay, "Implémentation d'un langage fonctionnel orienté vers la méta programmation," 2017.

[22] S. Awodey, *Category Theory*. USA: Oxford University Press, Inc., 2nd ed., 2010.

[23] B. Werner, *Une théorie des constructions inductives*. PhD thesis, Université Paris-Diderot-Paris VII, 1994.

Appendix A

Equational Reasoning

Equational reasoning provides us with a neat way of expressing proofs in Typer. With the help of some syntactic sugar, this allows us to build chains of equality proofs by using the transitivity property of equality as shown in Section 1.7. This is something that is implemented in the libraries of most proof assistants, such as Agda, Lean, and Lean. This allows lengthy proofs to remain readable as intermediate steps are documented and are clearly seen.

First, we introduce a helper function that simply invokes the transitivity property of equality proofs. This function carries out a single step of equational reasoning.

```
step-≡ : (x : ?A) → (y : ?A) → (z : ?A) → Eq x y → Eq y z → Eq x z
step-≡ _ p q = Eq_trans p q
```

Next, we require a second function to conclude a chain of equational reasoning.

```
qed : (x : ?A) → Eq x x
qed x = Eq_refl
```

To make it all come together, we wrap the above functions with some syntaxic sugar.

```
_==<_>==_ = step-≡
_  □ = qed
```

`_==<_>==_` can be seen as a mixfix operator, whereas `_  □` is a postfix operator.

When we write the following

```
  x
  ==< p >==
  y
```

```
  ==<  ...  >==
  .
  .
  .
  ==<  ...  >==
  z   □
```

We require that p be a proof of equality between x and y, and this can be chained *ad infinitum*. The expression in its entirety is a proof of equality between x and z. The usage of this is best illustrated via a simple example:

```
example : (x : ?A) → (y : ?A) → (z : ?A) → (w : ?A)
          → (p : Eq x y) → (q : Eq y z) → (r : Eq z w) → Eq x w
example x y z w p q r =
  x ==< p >==
  y ==< q >==
  z ==< r >==
  w   □
```

For a more involved example, we rewrite the proof of `Integer_*DistR+` that was shown in Section 4.1 by using equational reasoning.

```
Integer_*DistR+ : (x : Integer) → (y : Integer) → (z : Integer)
                  → Eq (Integer_* x (Integer_+ y z))
                       (Integer_+ (Integer_* x y) (Integer_* x z))
Integer_*DistR+ x y z =
  Integer_* x (Integer_+ y z)
  ==< Integer_*-comm x (Integer_+ y z) >==
  Integer_* (Integer_+ y z) x
  ==< Integer_*DistL+ y z x >==
  Integer_+ (Integer_* y x) (Integer_* z x)
  ==< Eq_cong (flip Integer_+ (Integer_* z x)) (Integer_*-comm y x) >==
  Integer_+ (Integer_* x y) (Integer_* z x)
  ==< Eq_cong (Integer_+ (Integer_* x y)) (Integer_*-comm z x) >==
  Integer_+ (Integer_* x y) (Integer_* x z)   □
```

TODO: Consider adding a discussion on additional macros and tactics that could be added to ease the development of such proofs, e.g. it would be nice to have rewrite tactics that Lean has to avoid having to invoke Eq_cong that often

Appendix B

Other proofs

B.1. Commutativity of addition of natural numbers

We first define the type of natural numbers along with the addition operation the usual way.

```
Nat : Type

type Nat
  | zero
  | succ Nat

_+_ : Nat → Nat → Nat
_+_ x y = case x
  | zero ⇒ y
  | succ m ⇒ succ (m + y)
```

We then define two lemmas that we ultimately use to prove the commutativity of addition.

```
Nat+_zero : (m : Nat) → Eq (m + zero) m
Nat+_zero m =
  case m return Eq (m + zero) m
         | zero ⇒ Eq_refl
         | succ n ⇒ Eq_cong succ (Nat+_zero n)

Nat+_succ : (m : Nat) → (n : Nat) → Eq (m + succ n) (succ (m + n))
Nat+_succ m n =
  case m return Eq (m + succ n) (succ (m + n))
         | zero ⇒ Eq_refl
         | succ m' ⇒ Eq_cong succ (Nat+_succ m' n)
```

```
Nat+_comm : (m : Nat) → (n : Nat) → Eq (m + n) (n + m)
Nat+_comm m n =
  case n return Eq (m + n) (n + m)
         | zero ⇒ Nat+_zero m
         | succ n' ⇒ Eq_trans (Nat+_succ m n')
                              (Eq_cong succ (Nat+_comm m n'))
```

Appendix C

Les différentes parties et leur ordre d'apparition

J'ajoute ici les différentes parties d'un mémoire ou d'une thèse ainsi que leur ordre d'apparition tel que décrit dans le guide de présentation des mémoires et des thèses de la Faculté des études supérieures. Pour plus d'information, consultez le guide sur le site web de la facutlé (www.fes.umontreal.ca).

Ordre des éléments constitutifs du mémoire ou de la thèse		
1.	La page de titre	obligatoire
2.	La page d'identification des membres du jury	obligatoire
3.	Le résumé en français et les mots clés français	obligatoires
4.	Le résumé en anglais et les mots clés anglais	obligatoires
5.	Le résumé dans une autre langue que l'anglais ou le français (si le document est écrit dans une autre langue que l'anglais ou le français)	obligatoire
6.	Le résumé de vulgarisation	facultatif
7.	La table des matières, la liste des tableaux, la liste des figures ou autre	obligatoires
8.	La liste des sigles et des abréviations	obligatoire
9.	La dédicace	facultative
10.	Les remerciements	facultatifs
11.	L'avant-propos	facultatif
12.	Le corps de l'ouvrage	obligatoire
13.	Les index	facultatif
14.	Les références bibliographiques	obligatoires
15.	Les annexes	facultatifs
16.	Les documents spéciaux	facultatifs