Design of Quotient Inductive Types

Jason Hu
zhong.s.hu@mail.mcgill.ca

1 Introduction

In this project, our goal is to implement quotient inductive types (QITs) [Altenkirch and Kaposi, 2016], which are inductive types but with additional equational structures. Generally speaking, inductive types in Martin-Löf-style type theories (MLTT) are generated freely by constructors. Unfortunately, different constructors of the same type are disjoint. For example, in the definition of natural numbers,[1]

```
data Nat : Set where
  z : Nat
  1+ : Nat → Nat
```

we can show that `z` and `1+` are disjoint:

```
z≠1+ : ∀ n → z ≠ 1+ n
z≠1+ _ ()
```

Another way to consider inductive types is data generated by mere syntax.

In reality, however, we often want to have more structures in our data. For example, in simply typed lambda calculus, we want to equate $\pi_1(s,t)$ and s. In MLTT, we have no choice but to use a separate equivalence relation to encode this equality and show that many transformations we care about respect this relation. On the other hand, with QITs, this equation is baked in as part of the definition and thus all transformations are required to have accompanied proofs of preservation of equality. Thus, it is quite motivating to consider a prototypical implementation of QITs. In this brief report, we will dissect this problem into a theoretical side and a practical side, which we shall discuss next.

2 Quotient Inductive Types

The intended quotient inductive types to be implemented in this project is based on propositional extensional type theory, in which we admit the axiom K or, equivalently, Uniqueness of Identity Proofs (UIP). This is very important as we want to eliminate the computational content of equality, as opposed to homotopy type theory (HoTT) [Univalent Foundations Program, 2013]. This is because if we do not eliminate the computational content of equality proofs, then we fall back to the same situation as HoTT and the complexity grows immediately and exponentially. This implementation choice induces that the resulting type theory is equivalent to `hSet` in HoTT. This matches up our intuition as quotient is more or less a "set theoretic thing".

2.1 Equality

Considering what the current Typer has, we shall define the notion of propositional equality into our type theory, which is absent in Typer at the moment. Since the definition of a QIT requires to refer to the notion of equality, we must introduce this notion and the language must be aware of it. Following are the rules:

$$\frac{\Gamma \vdash s : T \qquad \Gamma \vdash u : T}{\Gamma \vdash s =_T u : \texttt{type}} \qquad\qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash refl_T(t) : t =_T t}$$

$$\frac{\Gamma \vdash M : \Pi(x\ y : T).\Pi(x =_T y).\texttt{type} \qquad \Gamma \vdash s : T \qquad \Gamma \vdash u : T \qquad \Gamma \vdash p : s =_T u \qquad \Gamma \vdash q : M(s,s,refl_T(s))}{\Gamma \vdash J_T(M,s,u,p;q) : M(s,u,p)}$$

[1] We use Agda syntax for demonstration, but flip the usages of `=` and $\equiv$ to comply with the standard notations.

We employ the notion of homogeneous equality here. $=_T$ means it is an equality between two terms of type T. We very often omit this subscript from now on. $refl(t)$ is its constructor, reflexivity; it proves $t = t$ (up to definitional equality) using reflexivity. J is the eliminator. M is the motive. J says that to prove $M(s, u, p)$ is true, it is sufficient to assume s in place of u everywhere and p is just $refl(s)$.

According to [Univalent Foundations Program, 2013], mere J allows an equality proof to contain computational content and that necessarily implies there can be two distinct proofs of $s = u$. Luckily, the computational content in equality can be eliminated by imposing another axiom, K:

$$\frac{\Gamma \vdash M : \Pi(x : T).\Pi(x =_T x).\texttt{type} \qquad \Gamma \vdash s : T \qquad \Gamma \vdash p : s = s \qquad \Gamma \vdash q : M(s, refl_T(s))}{\Gamma \vdash K_T(M, s, p; q) : M(s, p)}$$

In its appearance, K seems to be a special case of J, but it is not. Effectively, it asserts that the reflexivity proof is the only proof of $s = s$. That is

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash p : t = t \qquad \Gamma \vdash q : t = t}{\Gamma \vdash K(\lambda x\ p'.p' = q, t, p; K(\lambda y\ q'.refl(t) = q', t, q; refl(refl(t)))) : p = q}$$

The above derivation defines UIP. By applying K twice, we can show that any two equality proofs of the same term can be identified. Since $refl$ is one of that, all proofs are identified with $refl$.

β equivalences are the following:

$$J(M, s, s, refl(s); q) \equiv q \qquad\qquad K(M, s, refl(s); q) \equiv q$$

2.2 Inductive Types

With propositional equality, we can then consider how QITs looks like. Let us begin with a simpler example and generalize that to other cases.

2.2.1 An Example: Integers

Consider the theory of groups. We know that the free group generated by a singleton set is (isomorphic to) the integers. Syntactically, we have the following definitions:

```
data Z : Set where
  z : Z          -- unit
  1+ : Z → Z     -- successor
  _-1 : Z → Z    -- predecessor
```

Next, we want to take quotient of this definition by neutralizing the effect of the successor and the predecessor. In other words, `1+` and `-1` should form isomorphism:

```
  -- ...
  1+-1 : ∀ x → 1+ (x -1) = x
  -11+ : ∀ x → (1+ x) -1 = x
```

The elimination principle of Z is more curious: other than the cases for the term constructors, the elimination principle should ensure the definition respect the custom equality. Since we claim that Z is a group, next we define `_+_` on the left hand side:

```
_+_ : Z → Z → Z
z + n       ≡ n
(1+ m) + n ≡ 1+ (m + n)
(m -1) + n ≡ (m + n) -1
-- case for 1+-1
-- ∀ m → 1+ (m -1) ≡ m → (1+ (m -1)) + n = m + n
_+_ (1+-1 m) n ≡ 1+-1 (m + n)
-- case for -11+
-- ∀ m → (1+ m) -1 ≡ m → ((1+ m) -1) + n = m + n
_+_ (-11+ m) n ≡ -11+ (m + n)
```

```
-_ : Z → Z
- z          ≡ z
- (1+ m)     ≡ (- m) -1
- (m -1)     ≡ 1+ (- m)
-- equality proofs
-_ (1+-1 m) ≡ -11+ (- m)
-_ (-11+ m) ≡ 1+-1 (- m)
```

Luckily, we do not have to resort to more complex proof for equality preservation. Let us reason about it more carefully. Since the definition is based on elimination of the first argument, the elimination principle thus requires `_+_` to respect the additional equality of `Z`. As shown in the comment, the first equality requires to show `(1+ (m -1)) + n = m + n`. We reason about the left hand side of the equation definitionally:

```
(1+ (m -1)) + n ≡ 1+ ((m -1) + n) ≡ 1+ ((m + n) -1)
```

We can see that `1+-1` discharges this obligation. The next equality is very similar.

The negation function `-_` on the right hand side is defined very similarly. The proofs of the cases are flipped precisely because negation flips the direction.

2.2.2 General Formulation

For simplicity, let us only consider algebraic data types. We will see that even with algebraic data types, the elimination principle is already very complex. If we have more time, we might attempt to tackle general inductive data types[2]. In the former case, we have the following formulation:

```
data T : Set where
  c_i : ∀ (x_0 : S_0) (x_1 : S_1(x_0)) ... (x_n : S_n(x_0, ... , x_{n-1})) → T
  -- ...
  eq_j : t_1 = t_2
  -- ...
```

$\mathtt{c}_i$ is some constructor, which has n arguments. eq_j is the equality part (with the parameter omitted), in which two terms of type `T` are equalized.

Let us consider the recursor next. Similar to standard inductive types, the recursor maps constructions of type `T` to another type `U`. If the constructor is

```
c_i : ∀ (x_0 : S_0) (x_1 : S_1(x_0)) ... (x_n : S_n(x_0, ... , x_{n-1})) (x_{n+1} : T) ... (x_{n+m} : T) → T
```

then the case for eliminator should have the following type

$$\forall(x_0 : S_0)\cdots(x_n : S_n(x_0, \cdots, x_{n-1}))(r_1 : U)\cdots(r_m : U) \to U$$

Here we replace x_{n+1} to x_{n+m} which are subterms of type `T` to r_1 t or$_m$ of type `U` as the result of recursive calls. We collect all these cases in the recursor for term constructors, rec_T. The computational behavior of rec_T is precisely the same as usual inductive types.

QITs requires that equalities are coherent w.r.t. the quotient equality. That means we must supply the proof of equality preservation. Thus for each quotient constructor $\mathtt{eq}_j$, we must have a proof of the following type:

$$\forall(t_1\ t_2 : T), t_1 =_T t_2 \to rec_T(t_1) =_U rec_T(t_2)$$

We collect these cases in $rec_T^{=}$. Computationally, given $\mathtt{eq}_j$, $rec_T^{=}$ dispatches to this case and return its implementation given by the programmer. Nonetheless, we shall not be concerned about the computational behavior of $rec_T^{=}$ due to our assumption of UIP. We can unequivocally consider reflexivity is the proof.

Next we consider the induction principle. The induction principle requires a motive $M : T \to \mathtt{type}$. We must construct from the cases so that the eliminator gives a term of type $M(t)$ where $t : T$. We follow the usual formulation of induction principles, for the constructor $\mathtt{c}_i$ above, we require the following type for the corresponding case:

$$\forall(x_0 : S_0)\cdots(x_n : S_n(x_0, \cdots, x_{n-1}))(x_{n+1} : T)\cdots(x_{n+m} : T)(r_1 : M(x_{n+1}))\cdots(r_{n+m} : M(x_{n+m})) \to M(c_i(x_0, \cdots x_{n+m}))$$

The inductive principles are r_1 to r_m for each of r_{n+1} to r_{n+m} respectively. We collect these cases in the eliminator $elim_T(M)$.

The preservation of equality is more complex. Notice that we cannot equalize terms of type $M(t_1)$ and $M(t_2)$ as they are generally different, while propositional equality requires their equivalence up to definitional equality. Thus we must use apply J to convert $M(t_1)$ to a $M(t_2)$ along $\mathtt{eq}_j$. In other words, we apply $\mathtt{eq}_j$ as a substitution of obtain a function $\mathtt{eq}_j{}^*$ of type $M(t_1) \to M(t_2)$. If $t_1' : M(t_1)$ and $t_2' : M(t_2)$, notationally we write the desired equality as $t_1' =^{eq_j} t_2'$, which is precisely $\mathtt{eq}_j{}^*(t_1') =_{M(t_2)} t_2'$. We do not omit this superscript as it is essential in the meaning of this equation. Thus the preservation requires the following equation to hold:

$$elim_T(M, t_1) =^{eq_j} elim_T(M, t_2)$$

These cases are collected in $elim_T^{=}(M)$.

[2]We probably should start inductive types with one index in that case.

2.3 Heterogeneous Equality

In the previous section, we can see that the preservation of equality in eliminator is rather complex because we must express equality between two terms with different indices. It looks quite unnatural when the eliminator of a subsequently defined type to rely on a particular application of J. One route to get around that is to consider using heterogeneous equality [McBride, 2000]. Heterogeneous equality generalizes propositional equality and its elimination implies the axiom K. It would be quite interesting to understand whether heterogeneous equality actually reacts better than propositional equality.

3 Targeted Modifications to Typer

In the previous section, we list the related theoretical setup for this project. In this section, let us look into how we can adapt Typer to QITs. As laid out in the previous section, we primarily have to deal with two concepts, equality and inductive types.

3.1 Equality

Supporting QITs requires the language itself to have a fixed notion of equality. For this reason, we must implement equality. There is also another benefit: in the current Typer, we do not have universe polymorphism, so if we define equality as an inductive type, then we are bound to define one for each universe level.

The current plan to implement equality is as follows:

1. We add an extra case to `lexp`, `Eq`, to represent the equality. For convenience, it is in its fully applied form; that is, it takes three arguments, a type and two terms. It's partially applied form can still be obtained by lambda abstracting. This representation also has the benefit in weak head normal form reduction.

   This strategy handles homogeneous equality. If we go by heterogeneous equality, we shall have `Eq` to contain two types.

2. We add an infix operator `===` to the grammar with proper fixity.

3. $refl$ can be included as another case in `lexp` or as a builtin. The former probably has the benefit of making the logic clear.

4. The elimination is done in case expression. We intend to follow [Cockx et al., 2014] without removing K. That is

   ```
   case t      %% : x === y
   | refl => ...
   ```

5. It might be convenient to have a builtin notion of substitution of the following type

   ```
   subst : (T : Type) (M : T -> Type) (x y : T) -> x === y -> M x -> M y
   subst T M x y p Mx = case p
   | refl => Mx
   ```

   Having the language be aware of this term will come in handy because as shown in Section 2.2.2 the elimination principle would need to apply it.

3.2 Inductive Types

We have more freedom to handle inductive types. We essentially need to make two big changes to the language:

1. How do we add quotient equalities to the type constructor?

2. How do we change pattern matching so that it also expands on the quotient equalities?

Currently, types in Typer are constructed by `typecons`. This constructs a type with some given number of constructors and arguments. We could add another builtin construct, `typeeqcons`, which is responsible for receiving the quotient equalities. For example, the definition of integers in Section 2.2.1 can be defined as follows:

```
Z : Type;
Z = typeeqcons (typecons z (1+ Z) (-1 Z))
               1+-1 ((x : Z) -> 1+ (-1 x) === x)
               -11+ ((x : Z) -> -1 (1+ x) === x);
z = datacons Z z;
1+ = datacons Z 1+;
-1 = datacons Z -1;
```

`typeeqcons` requires at least one arguments, which must be returned by `typecons`. When there is no subsequent arguments, `typeeqcons` behaves like an identity function. Subsequent arguments must come in pairs: the first is the name of the quotient equality and the second is the type of that equality.

We also require another construct `eqcons` to map quotient equalities to definitions:

```
1+-1 = eqcons Z 1+-1;
-11+ = eqcons Z -11+;
```

We need to extend `Inductive` in `lexp` with one additional argument of type `ltype SMap.t` to remember these equations. One invariant of each `ltype` is that it needs to have the form of `(x : T) -> ... -> t1 === t2` where `t1` and `t2` must have type of the currently defined type.

We might also improve the macro for `type_`, so that it also takes equations. The intended syntax is as follows:

```
type Z
| z | 1+ Z | -1 Z
quotient
| 1+-1 : (x : Z) -> 1+ (-1 x) === x
| -11+ : (x : Z) -> -1 (1+ x) === x;
```

The `quotient` part is optional if no quotient is intended. This syntax also has the benefit of forcing a separation between term constructors and quotient equalities.

Changes to pattern matching is more complex as it must handle the preservation proofs, and the preservation proofs need to refer to the cases for constructors. To accommodate this requirement, we change the `Case` constructor in `lexp` to take another `SMap` which maps the quotient equalities to the preservation proofs.

In the same spirit of separating cases for constructors and quotient equalities, we also need to change the syntax of pattern matching to the following (fitting the use case to the integers example):

```
case t
| z    => ...
| 1+ n => ...
| -1 n => ...
preserves
| 1+-1 n => ...    %% The left hand side sends (1+ (-1 n)) to the pattern matching.
| -11+ n => ...    %% This case goes similarly.
```

Cases after `preserves` are optional if the definition of the type does not have quotient equalities. We need to implement a covering check so that when these equalities exist they must be handled. A potentially tricky part is that the equality preservation part might need to apply `subst` depending on whether the result type depends on `t` or not. In the current stage, it is hard to predict how complex this might become. If it is overwhelming, then we would just consider recursors for simplicity.

At this point, we are hoping that we do not have to change the reduction of the language (modulo necessary adaptations) because equalities do not have interesting computational content after all. Unlikely in cubical type theory [Cohen et al., 2015], we do not even have a way to trigger the computational behavior of the preservation proofs.

4 Summary

To summarize, in order to implement QITs, we need to, first, support equality and, second, extend inductive types. Currently the intention is to only support quotient for algebraic data types. When one attempts to define quotient for inductive types with indices, an error should be given to signify the limitation. Nonetheless, we should be able to use quotiented algebraic data types as indices for inductive types out of the box. Pattern matching is extended to require proofs of equality preservation.

References

[Altenkirch and Kaposi, 2016] Altenkirch, T. and Kaposi, A. (2016). Type theory in type theory using quotient inductive types. In Bodík, R. and Majumdar, R., editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM.

[Cockx et al., 2014] Cockx, J., Devriese, D., and Piessens, F. (2014). Pattern matching without K. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 257–268.

[Cohen et al., 2015] Cohen, C., Coquand, T., Huber, S., and Mörtberg, A. (2015). Cubical type theory: A constructive interpretation of the univalence axiom. In Uustalu, T., editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 5:1–5:34. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[McBride, 2000] McBride, C. (2000). *Dependently typed functional programs and their proofs*. PhD thesis.

[Univalent Foundations Program, 2013] Univalent Foundations Program, T. (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study.

Implementing Quotient Inductive Types

Jason Z.S. Hu
McGill University
zhong.s.hu@mail.mcgill.ca

Abstract

Taking quotient is one of the fundamental and ubiquitous phenomena in set-theoretic mathematics: a quotient group of a kernel group remains a group; a quotient space in linear algebra is a projection to some hyperplane, etc. However, in type theories, taking quotient has not been easy, because type theories primarily focus on manipulation of syntax, which is very difficult to reflect arbitrary quotient equalities. In this work, we explore the notion of quotient inductive types, which add to usual inductive types quotient equalities. We discuss the implementation in the Typer language and the corresponding theory under the hood.

1 Introduction

Many fields in computer science and mathematics have been benefited from the development of type theories, which allows automatic verification of logical proofs and thus amplifies our confidence on mechanized proofs. However, during the translation from informal proofs to syntactic ones recognized by the type theories, we must pick a syntactic representation for every involved concept, which places an unwanted importance on the actual syntactic representation. Consider the definition of a list:

```
type List (A : Type)
| nil
| cons A (List A);
```

This inductive definition has two cases: `nil` represents the empty case while `cons` grows a list by one. This effectively defines a *free* monoid generated by `A`. What about a free group generated by a set? One might be tempted to add one case to represent the inverse operator in groups, e.g.:

```
type Group (A : Type)
| unit
| add (Group A) (Group A)
| neg (Group A);
```

Unfortunately, however the concept is defined, an additional constructor in a usual inductive type is disjoint from the other ones, because type theories distinguish them by their *distinct* syntax. As a consequence, we are not able to gracefully define many common set-theoretic concepts which require additional equational structures in many common type theories.

In this work, we explore one possible solution to this problem, quotient inductive types (QITs) [Altenkirch and Kaposi, 2016]. With QITs, we can introduce additional equalities (called quotient equalities) between constructors to an inductive definition. The elimination principle of this type must then be accompanied with the preservation proofs of those quotient equalities. QITs reconcile the differences between different syntactic representations of the same concept to some extent and thus allow us to focus on the actual problems of interest.

This work has been implemented as an extension to Typer, a dependently typed programming language. We add to this language a built-in notion of equality, dependent pattern matching, and support for QITs. We will discuss the choices made during the implementation as well as the theory underneath.

2 Disjointness in Inductive Types

In a typical inductive definition, constructors are disjoint. Consider the following definition of natural numbers:

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

Intuitively, we expect that `zero` and `succ` are completely *distinguished*, in the sense that the following lemma is provable:

```
succ-zero-disjoint : (m : Nat) -> succ m === zero -> Void;
```

Here `succ m === zero` means `succ m` and `zero` are propositionally equal and `Void` is the falsehood. That is, for any `m : Nat`, it is impossible to equate `succ m` and `zero`. In a type theory with strong pattern matching, this lemma can be readily proved by pattern matching against the equality, but this proof does not inform us how the Intuitively, `J` says that given the motive `M`, in order to prove `M a b p`, it suffices to supply a proof for `M a a (refl a)`, where `refl a : a === a` is the reflexivity proof. The insight here is that we can control the motive `M`, so that `M a a (refl a)` is easily inhabited, but `M a b p` reduces to `Void`:

```
succ-zero-disjoint m eq =
  J Nat (lambda x y _ -> case y
         | succ _ => Unit
         | zero => case x
                   | zero => Unit
                   | succ x => Void)
    (succ m) zero eq
    (lambda a -> case a
     | zero => unit
     | succ _ => unit);
```

In this proof, the motive is a double case split on both `x` and `y`, so that it reduces to `Void` when `x` is in the `succ` case and `y` is `zero`. The proof holds because we only need to handle the reflexivity case, in which we just need to supply a `unit` proof after a case split. If we look closer at the proof, we will see that for this proof to work, it is important to supply the motive with two unrelated types, `Unit` and `Void`, behind a case split. The disjointness of constructors is then a consequence: all branches of a case split are completely unrelated. This proof technique uses nothing about `Nat` in particular; it can be used to prove disjointness of all constructors in general. Observing this, we then understand how to overcome this restriction of inductive types: the secrete lies in introducing equational obligations between branches of a case split, just as implemented in the system.

In order to implement QITs, we must augment the system with two concepts:

1. propositional (homogeneous) equality: the version of Typer we base on does not have any notion of equality, while in order for the system to recognize quotients, we would need a built-in notion of equality.

2. QITs: with a built-in notion of equality, we can move on to support QITs. Our implementation splits in three parts: formations, constructions and eliminations. Formations and constructions are relatively easier. Eliminations are a bit trickier. Since we use pattern matching for eliminations, we need to ensure the quotient equalities are preserved in a case split.

3 A Taste of Quotient Inductive Types

Let us consider a definition of integers with which the same numbers are propositionally equal:

```
type ZInt
| zero | succ ZInt | pred ZInt
quotient
| succ-pred : (x : ZInt) -> succ (pred x) === x
| pred-succ : (x : ZInt) -> pred (succ x) === x;
```

`ZInt` models integers via three constructions: an integer can either be `zero`, a `succ`essor of an integer, or a `pred`ecessor of an integer. However, this definition as pure inductive one does not have good propositional equality. For example, `succ (pred zero)` should just be the same as `zero`, but they cannot be related by propositional equality. With QITs, we have an additional keyword, `quotient`, after which we list a number of equations. The overall type is then the inductive definition taking the equations as quotients. As a result, `succ (pred zero) === zero` becomes a fact and is witnessed by `succ-pred zero`.

In order to preserve consistency of the language, every elimination of `ZInt` must show that the quotients are preserved. Consider the following definition of addition:

```
plus : ZInt -> ZInt -> ZInt;
plus x y = case x
  | zero => y
  | succ x => succ (plus x y)
  | pred x => pred (plus x y)
  preserves
  | succ-pred x => succ-pred (plus x y)
  | pred-succ x => pred-succ (plus x y);
```

The case split should be straightforward: we just move the constructors of `x` over to `y` until it hits `zero`. What adds to the definition is the keyword `preserves` and the equational proofs after it. Intuitively, since `plus` is defined via eliminating `x`, we must show that `plus` respects quotients of `x`. For the `succ-pred` case, we must show `plus (succ (pred x)) y === plus x y`. By expanding the left hand side definitionally, we see that this obligation can be discharged by `succ-pred (plus x y)`. The case for `pred-succ` is very similar.

If we compare this pattern matching with the previous disjointness proof, we will see that the constructors of `ZInt` are no longer disjoint. Indeed, if we apply the same technique, we would be asked to provide a proof of `Unit === Void`, which is infeasible.

4 Implementing Equality

One and the first of the major problems in support QITs is (propositional) equality. Indeed, quotients are expressed as equations, so the type theory must recognize a unique notion of equality. Besides being able to specify equalities, we also want to reason about them; that implies we want to enable pattern matching for equality.

4.1 Representing Equality

Before worrying about the implementation, we shall begin by considering how one typically defines propositional equality:

$$\frac{\Gamma \vdash s : T \qquad \Gamma \vdash u : T}{\Gamma \vdash s =_T u : \texttt{type}} \qquad\qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash refl_T(t) : t =_T t}$$

$$\frac{\Gamma \vdash M : \Pi(x\ y : T).\Pi(x =_T y).\texttt{type} \qquad \Gamma \vdash s : T \qquad \Gamma \vdash u : T \qquad \Gamma \vdash p : s =_T u \qquad \Gamma \vdash C : \Pi(x : T).M(x, x, refl_T(x))}{\Gamma \vdash J_T(M, s, u, p; C) : M(s, u, p)}$$

The rules indicate three elements to represent: the formation, the construction and the elimination of equality. Since we use pattern matching as a uniform elimination form, we will just need to add two cases in the internal syntax to represent the equality and the reflexivity proofs, respectively:

```
type lexp =
  (* ... *)
  | Eq of U.location * ltype * lexp * lexp
  | Refl of U.location * lexp
```

The representation encodes all necessary information and is not very surprising. `Eq` represents equality and `Refl` represents the reflexivity proof, written as `refl` in Typer. It is worth noting that `_===_` does not have a type in Typer. Our version of Typer does not implement universe polymorphism, while the universe in which an instance of `Eq` resides is determined by the universe in which the `ltype` resides. In other words, equality in our augmentation is universe-polymorphic, so we cannot find a type for `Eq` within the type theory.

4.2 Dependent Pattern Matching

After resolving the formation and the construction of equality, we shall look into the elimination of equality. Besides the elimination principle J above, we also want the elimination to admit the following slightly different principle:

$$\frac{\Gamma \vdash M : \Pi(x : T).\Pi(x =_T x).\texttt{type} \qquad \Gamma \vdash s : T \qquad \Gamma \vdash p : s =_T s \qquad \Gamma \vdash C : \Pi(x : T).M(x, refl_T(x))}{\Gamma \vdash K_T(M, s, p; C) : M(s, p)}$$

In its appearance, K seems to be a special case of J, but it is not. In fact, this elimination principle has the consequence of *uniqueness of identity proofs*, which means that any proof of $s = s$ for any s is equal to the reflexivity proof:

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash p : t =_T t}{\Gamma \vdash K_T(\lambda x\ p'.p' =_{x =_T x} refl_T(x), t, p; \lambda x.refl_{x =_T x}(refl_T(x))) : p =_{t =_T t} refl_T(t)}$$

Though we often have no direct use of K, we need this elimination principle to reduce proof obligations in pattern matching on QITs and lay the theoretical foundation down to a simpler and more set-theoretic one. We will expand on this when discussing QITs.

To support pattern matching on equality, we need to decide how complex this pattern matcher should be: on one hand, the one similar to Agda's is too strong, because its pattern matching even has injectivity and disjointness of constructors built in; one the other hand, the one close to Coq's is probably too weak, as writing out convoy pattern [Chlipala, 2013] is too tedious. We want to be able to prove J and K but still control the power of pattern matching. We eventually come to a very simple criterion: pattern matching on `e1 === e2` is admitted only when `e1` and `e2` are convertible to variables. This strategy hits a sweet spot: this criterion ensures that the unification must be successful (because unification of two variables must succeed) and is general enough to prove J and K. Given a context `x : T, y : T` and the following code with `e1` has type `x === y`:

```
case e1 | refl z => e2
```

we follow the following steps in the type checking phase:

1. We extend the current typing context with `z : T`.
2. If `x` and `y` are the same variable, then we unify that variable with `z`. This effectively implements K.

3. If `x` and `y` are not the same variable, then we unify both variables with `z`. This effectively implements J.

4. If `e1` is convertible to a variable, then we unify that variable with `refl z`.

5. Then we proceed to type check `e2`.

This algorithm is much easier to understand and reason about than stronger pattern matcher presented in e.g. [Cockx et al., 2014] while we can still use it to prove J and K, which indicates that equality has all logical strength that we want it to have. Moreover, with quotients, injectivity and disjointness of constructors are no longer natural, so they should be removed from pattern matching after all:

```
J : (A : Type) -> (M : (a : A) -> (b : A) -> (p : a === b) -> Type) ->
    (a : A) -> (b : A) -> (p : a === b) -> (C : (a : A) -> M a a (refl a)) -> M a b p;
J A M a b p C = case p | refl x => C x;

K : (A : Type) -> (M : (a : A) -> (p : a === a) -> Type) ->
    (a : A) -> (p : a === a) -> (C : (a : A) -> M a (refl a)) -> M a p;
K A M a p C = case p | refl x => C x;
```

One disadvantage of this strategy is that if we have a proof of `x === f y`, then we cannot use pattern matching alone to substitute `x` with `f y`. Instead, we must rely on helper functions to achieve this:

```
subst : (A : Type) -> (M : A -> Type) -> (a : A) -> (b : A) -> a === b -> M a -> M b;
subst A M a b p Ma = case p | refl _ => Ma;
```

4.3 A Note on Unification

In the previous section, we left the details of unification vague. Indeed, there are many ways to perform unification, but we employed a method which we believe is not quite typical. Specifically, we use negative de Bruijn indices. In the previous section, when unifying the variable `x` with `z`, since `x` appears before `z`, we use a negative de Bruijn index to create a let binding from `x` to `z`. For example, if the top of the context is `x : T, z : T`, then we update the context to `x = z[-2] : T, z : T`, where `[-2]` indicates that from `x`'s perspective, `z` has de Bruijn index `-2`. Nonetheless, when we actually normalize `x`, we will performance enough shifting so that all de Bruijn indices are nonnegative.

This treatment is quite strange because negative de Bruijn indices are not common, but if we regard the typing context as a constraint set, then this idea becomes more acceptable. Negative de Bruijn indices are just forward referencing in a constraint set. As long as the variable actually exists (which must be the case because unification only occurs after an extension of the context), an enough shifting guarantees that the eventual reference will have a nonnegative index. Moreover, this treatment implicitly makes the effect of unification *transitive*. Consider the following proof of transitivity:

```
eq-trans : (A : Type) -> (a : A) -> (b : A) -> (c : A) -> a === b -> b === c -> a === c;
eq-trans A a b c p q = case p | refl x => q;
```

After pattern matching `p`, we simply use `q` as the proof. It is quite intuitive at the first glance, but a closer look will show that this proof type checks because of our treatment. Since both `a` and `b` forward reference `x`, the convertibility checker will see that they are both convertible to `x`, and thus accepts `q` as a proof.

4.4 Computational Behavior

Since our system admits K, the computational behavior of propositional equality is not important, as K "kills it off". Nevertheless, we have implemented it for completeness. The following program

```
case refl t | refl x => e
```

reduces to `e[t/x]`, which is the capture-avoiding substitution of `x` for `t` in `e`.

5 Quotient Inductive Types

Having implemented equality, we can now consider how to implement QITs. In this project, this goal is a little bit simplified: Typer only has algebraic data types, so we do not have to worry about how quotient equations should react to J applied to an index of a type. To add quotient equations to algebraic data types, we need to represent these equations in all of the formation, the construction and the elimination, which is naturally reflected in the internal syntax:

```
type lexp =
(* ... *)
| Inductive of U.location * ((vname * ltype) list) SMap.t * quotient SMap.t
| EqCons of ltype * symbol
| Case of U.location * lexp
          * ltype                                        (* The type of the return value *)
          * (U.location * vname list * lexp) SMap.t
          * (vname * lexp) option                        (* default branch  *)
          * (U.location * vname list * lexp) SMap.t (* proofs of preservation *)
and quotient = {
    premises: (vname * ltype) list;  qloc: U.location;
    lhs: lexp;  rhs: lexp }
```

Let us explain the representation in more details.

5.1 Formation and Construction of QITs

In Typer, a QIT has the following general form[1]:

```
type Name (S1 : Type) ... (Sn : Type)
| c1 (x1 : T1) ... Tm
...
quotient
| eq1 : (x1 : T1) -> ... (xk : Tk) -> e1 === e2
...
```

That is, we augment usually algebraic data types with a number of quotient equations. Moreover, each equation (say `eq1`) must satisfy the following criteria:

1. It has to be well-typed.
2. `e1` and `e2` must have type `Name S1 ... Sn`.

Each quotient equation is represented as a `quotient` in OCaml and all of them are collected in a string map (`SMap`). A `quotient` remembers the premises (the arguments of the quotient), the location and both sides of the equation.

Each QIT has two kinds of constructors: term constructors and quotient constructors. Term constructors are regular constructors of the type. Quotient constructors represent the quotient equations, represented by `EqCons` in the internal syntax. We can easily infer the type of a quotient constructor given the inductive type it belongs to and its name.

5.2 Quotient Pattern Matching

Finally let us consider how pattern matching should handle QITs. Intuitively, when pattern matching against a QIT, we would expect that the quotient equations are preserved by the branches. In Typer, we have the following general front-end syntax:

[1] This form is the ideal frontend sytnax, which has not been implemented, but the primitives for supporting QITs are ready.

```
case e
| c1 x1 ... xm => branch1
...
preserves
| eq1 x1 ... xk => proof1
...
```

Everything before the `preserves` keyword is just typical in any pattern matching. It splits `e` by cases. Each case handles a constructor of the inductive type which `e` has. With quotients, we in addition require preservation proofs, with an extra field of `proofs of preservation` in `Case`. These proofs are responsible for relating the branches with equational obligations.

Before supporting quotient pattern matching, we also need to extend pattern matching on inductive types with the unification handling shown in Section 4.2 as well. That is, when type checking a branch, say `branch1`, if `e1` is convertible to a variable, then we unify it with `c1 x1 ... xm`. This allows us not only to prove general fact about an inductively defined type, but also to support the necessary equational reasoning after the `preserves` keyword.

The preservation proofs are type checked according to the following criteria:

1. The scrutinee of pattern matching must be convertible to a variable. This is essential for performing unification, so that the obligations of preservation proofs can actually be meaningfully generated.

2. If the result type of the pattern matching is an equality type, then there must be no preservation proofs. That is, if the overall type of a pattern matching has type `x === y`, then we do no need preservation proofs, because if the branches are type checked, then all obligations of the preservation proofs must hold due to the consequence of uniqueness of identity proofs (UIP) of K. This is why we must support K even though we do not often have explicit use of it. K is very important for the underlying mathematics to work out for this type theory. We even took a further step to forbid any proofs to reduce unnecessary type checking effort.

3. If the result type is not an equality, then we demand preservation proofs of all quotients.

4. Moreover, we require the result type must not depend on the scrutinee of the pattern matching. That is, if we have

   ```
   case x | ... preserves | ...
   ```

   Then this expression must not in any form has a type containing `x` as a free variable (except for equality type, but this criterion only applies for non-equality result types). This criterion is not theoretically necessary, but is added to simplify the proof obligations. Imagine for a moment the result type of the overall expression is `M x` for some irreducible motive `M`. Furthermore let us assume a quotient between two distinct constructors `c1` and `c2`: `c1 === c2`. Then this quotient will generate an equation with both sides having type `M c1` and `M c2`, respectively. However, these two types are not convertible. Since we implement homogeneous equality, terms of the types cannot be equated directly. We could use the `subst` function in Section 4.2, but this adds some complexity to the type system. Nevertheless, this limitation is not substantial, so we leave it as a future work.

5. Finally we require the preservation proofs to be well-typed.

5.3 Quotient Checking

When we look at a piece of complete code, we often have an intuitive understanding of what preservation of quotients means. In the implementation, we need to make this intuition concrete. Consider again the general syntax of quotient pattern matching:

```
case e
| c1 x1 ... xm => branch1
...
preserves
| eq1 x1 ... xk => proof1
...
```

we formulate the preservation checking steps as follows:

1. Let the pattern matching before the `preserves` keyword be `E`. That is, let `E` be

   ```
   case e
   | c1 x1 ... xm => branch1
   ...
   ```

2. We also know `e` must be convertible to a variable. Let that variable be `x`.

3. Now we move on to type checking the preservation proofs. Let us consider the `eq1` case without loss of generality. We first extend the context with `x1` to `xk`.

4. If `eq1 x1 ... xk` has type `e1 === e2`, then the proof obligation is `E[e1/x] === E[e2/x]`. If `e` is not convertible to a variable, we would not be able to formulate the goal using substitution.

5. We then check whether `branch1` has the specified obligation.

One advantage of this schema is again simple to understand and implement. Moreover, this schema naturally extends to nested pattern matching. The nested layers will require correct equational obligations at the end.

5.4 Computational Behavior

The computational behavior of quotient pattern matching is identical to the usual pattern matching. In particular, the preservation proofs do not participate in any computation. This is intended and is theoretically justified by the admissibility of K.

6 Notes on Limitations

Due to the general design of the system, our implementation has certain limitations which do not have theoretical implications. In this section, we discuss these limitations, potential solutions, and their causes.

6.1 Recursion in Quotient Pattern Matching

In quotient pattern matching, we often need to refer to the function being currently recursively defined in the equational obligations. In our current implementation, we have problems handling recursive calls behind a quotient pattern matching. Consider the following program:

```
type Foo
| c1 Nat
| c2 Nat
preserves
eq : (x : Nat) -> c1 x === c2 x;
```

```
bar x = case x
| c1 y => bar (c2 y)
| c2 y => y
preserves
| eq y => refl y;
```

On the left hand side, we define a (trivial) type `Foo`, and on the right hand side, we have a fancy way to extract the `Nat` in it. Consider the equational obligation in the `eq` case, we should prove, according to the precise schema, the following equational obligation:

```
case c1 y                                          case c2 y
| c1 y => bar (c2 y)              ===             | c1 y => bar (c2 y)
| c2 y => y                                       | c2 y => y
```

It reduces to `bar (c2 y) === y`. We know it is true and should be provable using `refl y`, except that Typer will not accept it. This behavior is a natural limitation of how Typer is structured. In Typer, there is an elaboration phase, in which an elaborator might perform some type directed operation to turn a parse tree into an internal syntax tree. The dilemma here is that in order to know that `bar (c2 y)` is convertible to `y`, we need an already parsed internal syntax tree; but since we cannot learn this convertibility relation without the internal syntax tree, the elaborator refuses to parse, and thus this program is rejected.

In general, the system has problems accepting certain programs, the proofs of equational obligations of which require expansion of the recursive function that is being defined. Depending on the problem, we might be able to work around this limitation by reorganizing code or employ an alternative definition.

6.2 Overcoming Quotient Obligations

Mandatory obligations in quotient pattern matching are very nice and helpful as it ensures that pattern matching respects the quotients everywhere. However, it might soon become a limitation for some problems or for some intended solutions.

Consider the `ZInt` type defined in Section 3. We might be interested in its "syntactic normal form", namely, those in the form of either `zero`, a number of `succ` of `zero`, or a number of `pred` of `zero`. We can definitely define a `normalize` function, but this function has much less use than intended. Imagine how we obtain the sign of a `ZInt`. Given a normal form, the `sign` function simply inspects its first constructor: if it is a `zero`, then we return `zero`[2], if it is a `succ`, we return positive, and if it is a `pred`, we return negative:

```
type Sign | Zero | Pos | Neg;

sign : ZInt -> Sign;
sign x = case normalize x
| zero => Zero
| succ _ => Pos
| pred _ => Neg
preserves
???    %% we are not able to generate proper obligations here
```

Clearly the pattern matching does not generally respect quotients at all! Its correctness is based on the special syntactic property of a normalized `ZInt`. Nonetheless, the result of composing `normalize` and the pattern matching does respect quotients. That is, this solution is valid, but its intermediate step violates the "preservation everywhere" requirement from the type theory.

The question to ask is whether we can relax the type theory somehow, such that this form of "internal violations" becomes acceptable. This would be particularly useful in the study of programming languages, where we most of time are concerned about structure preserving transformation, except that some properties are about the mere syntax.

7 Conclusion

In this project, we considered the theoretical aspects of quotient inductive types and provided an experimental implementation in the Typer programming language. We discuss some critical design choices in the implementation enabling the feature. Though this feature has demonstrated benefits, the current implementation and the theoretical foundation have various degrees of limitation and room for future improvements. We outlined two possible future directions for further investigation.

[2]Let us say `zero` has its own sign.

References

[Altenkirch and Kaposi, 2016] Altenkirch, T. and Kaposi, A. (2016). Type theory in type theory using quotient inductive types. In Bodík, R. and Majumdar, R., editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM.

[Chlipala, 2013] Chlipala, A. (2013). *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press.

[Cockx et al., 2014] Cockx, J., Devriese, D., and Piessens, F. (2014). Pattern matching without K. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 257–268.