Quotients by Idempotent Functions in Cedille

Andrew Marmaduke, Christopher Jenkins, and Aaron Stump

The University of Iowa
{andrew-marmaduke,christopher-jenkins,aaron-stump}@uiowa.edu

Abstract. We present a simple characterization of *definable quotient types* as being induced by idempotent functions, and an encoding of this in Cedille (a dependently typed programming language) in which both equational constraints *and the packaging* that associates these with elements of the carrier type are irrelevant, facilitating equational reasoning in proofs. We provide several concrete examples of definable quotients using this encoding and give combinators for function lifting (with one variant having zero run-time cost).

Keywords: Quotients · Quotient Types · Type Theory · Cedille

1 Introduction

Every dependently typed programming language has some built-in notion of *definitional equality* of expressions which is induced by its operational semantics. This notion can then be internalized as an equality type within the language, called *propositional equality.* Propositional equality often enjoys a privileged status, with language and library authors providing support for reasoning with it in the form of, e.g., special rewriting syntax or tactics specifically for it. However, it sometimes occurs that the programmer wishes to consider two expressions of some type A equal up to some arbitrary equivalence relation $\sim$, which will not have the same support as propositional equality. Quotient types provide a solution to this problem by allowing the formation of a new type $A/\sim$ for which the equivalence $a \sim b$ corresponds precisely to propositional equality of the *equivalence classes* $[a]$ and $[b]$ of type $A/\sim$.

As an example, the rational numbers constitute an archetypal application of quotients. First, fractions are defined as a pair of two natural numbers. Next, an equivalence relation is defined between fractions such that a/b is equivalent to c/d if and only if $ad = cb$. The quotient type with respect to this equivalence relation constructs the rational numbers. Alternatively, we can decide whether a/b and c/d are equivalent by comparing *canonical representatives* of their equivalence classes, computed by dividing both numerator and denominator by their greatest common divisor. Observe that this canonical choice operation for rationals is necessarily idempotent. Generalizing, it turns out that rational numbers and all other *definable quotient types* (in the sense of Li [15]) can be characterized by the set of fixpoints of some idempotent function on the carrier type. We call this *quotients by idempotent functions*, and to the best of our knowledge are the first to work with this characterization of definable quotients explicitly.

Of course, definable quotients are, for any suitable characterization of them, definable already in existing proof assistants like Agda or Coq [15,4]. In this work, we argue that certain features of Cedille's type theory makes the encoding of our formulation especially simple, as in particular not only are all required proofs erased (as one expects already in theories with proof irrelevance) but indeed the very *packaging* used to associate terms with their equational constraints is *also* erased during equational reasoning. In summary, our contributions are:

1. a novel and simple characterization of definable quotients by *idempotent functions*;
2. an encoding of this characterization that takes advantage of Cedille's extrinsic typing and notion of erasure to allow every definable quotient to be *definitionally equal to* an element of the carrier type;
3. examples of definable quotients formalized in Cedille (with a code repository available at `github.com/cedille/cedille-developments/tree/master/idem-quotients`) including:
   - a quotiented identity type (the carrier of which lacks decidable equality);
   - naturals modulo some k;
   - the even and odd subset types of naturals considered as quotient types;
   - finite sets as lists whose elements have decidable equality, whose combination with other definable quotient types highlights the advantages of our encoding;
   - integers as a definable quotient *inductive* type, with constructors compatible with the intended equivalence relation and an induction principle in terms of these;
4. combinators for *lifting* of functions, which for compatible functions can be done such that the lifted function is definitionally equal to the original.

This version of the paper improves upon an earlier draft by more clearly identifying the class of quotient types to which quotients by idempotent functions belong, emphasizing that the advantage of our encoding in Cedille is the disappearance in equations of explicit type coercions between quotient and carrier, focusing on examples of quotients by idempotent functions that contribute to the central argument of the paper, and better contextualizing our contributions in the existing literature on quotient types.

We begin the paper with a brief overview of Cedille's type theory and language features (Section 2). Next, we give a general definition of quotients by idempotent functions in Cedille and consider several examples (Section 3). Then, we present the satisfied properties of and combinators for our quotients by idempotent functions (Section 4). After, we consider the benefits and limitations of our work with respect to the existing literature on quotient types (Section 5). Finally, we conclude the paper and reflect on our contributions (Section 6).

(a) Equality

$$\frac{FV(t\ t') \subseteq dom(\Gamma)}{\Gamma \vdash \beta\{t'\} : \{t \simeq t\}} \qquad \frac{\Gamma \vdash t : \{t_1 \simeq t_2\} \quad \Gamma \vdash t' : [t_2/x]T}{\Gamma \vdash \rho\ t\ @\ x.T - t' : [t_1/x]T} \qquad \frac{\Gamma \vdash t : \{t_1 \simeq t_2\} \quad \Gamma \vdash t_1 : T}{\Gamma \vdash \varphi\ t - t_1\ \{t_2\} : T}$$

$$|\beta\{t'\}| \ = \ |t'|, \qquad |\rho\ t\ @\ x.T - t'| \ = \ |t'|, \qquad |\varphi\ t - t_1\ \{t_2\}| \ = \ |t_2|,$$

(b) Dependent Intersection

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : [t_1/x]T_2 \quad |t_1| = |t_2|}{\Gamma \vdash [t_1, t_2] : \iota\, x{:}T_1.\, T_2} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.1 : T_1} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.2 : [t.1/x]T_2}$$

$$|[t_1, t_2]| \ = \ |t_1|, \qquad |t.1| = |t|, \qquad |t.2| = |t|,$$

(c) Implicit Products

$$\frac{\Gamma, x : T \vdash t' : T' \quad x \notin FV(|t'|)}{\Gamma \vdash \Lambda\, x{:}T.\, t' : \forall\, x{:}T.\, T'} \qquad \frac{\Gamma \vdash t : \forall\, x{:}T'.\, T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t\ \text{-}t' : [t'/x]T}$$

$$|\Lambda\, x{:}T.\, t| \ = \ |t|, \qquad |t\ \text{-}t'| \ = \ |t|$$

Fig. 1: Typing and erasure for a fragment of Cedille

2 Background

2.1 CDLE

Cedille's core theory is the *Calculus of Dependent Lambda Eliminations* (CDLE) [21,22]. CDLE is an extension of the impredicative extrinsically-typed Calculus of Constructions [5] with three additional type formers: the *dependent intersections* $\iota\, x{:}T_1.\, T_2$ of Kopylov [14]; the *implicit products* $\forall\, x{:}T_1.\, T_2$ of Miquel [18] (which we may write $T_1 \Rightarrow T_2$ if $x \notin FV(T_2)$); and an equality type $\{t_1 \simeq t_2\}$ of untyped terms. The term language of CDLE is just the untyped λ-calculus, so to make type checking algorithmic Cedille requires users provide some type annotations, and definitional equality of terms is modulo *erasure* of these annotations. Figure 1 gives the term annotations in Cedille associated with these additional type constructs and their erasures. In particular, the erasure of the β axiom and dependent intersections is essential to our encoding of quotients by idempotent functions.

Equality $\{t_1 \simeq t_2\}$ is the type of proofs that t_1 and t_2 are equal, where these two terms are only required to be well-scoped. It is introduced with $\beta\{t'\}$ (for an unrelated t', discussed below) if $|t_1|$ and $|t_2|$ (the *erasures* of t_1 and t_2) are $\beta\eta$-convertible. If t has type $\{t_1 \simeq t_2\}$ it can be eliminated using ρ or φ where $\rho\ t\ @\ x.T - t'$ (which erases to $|t'|$) *rewrites* all occurrences of t_2 in the type $[t_2/x]T$ with t_1 using the guide $@\ x.T$, and $\varphi\ t - t_1\ \{t_2\}$ (which erases to $|t_2|$) *casts* t_2 to the type T assuming that t_1 has type T. For convenience in equational reasoning, Cedille allows the guide to be omitted when the type $[t_1/x]T$ of the

ρ-expression is known contextually, and provides the alternative form $\rho+\ t - t'$ which normalizes the expected type until it finds occurrences of t_1 to rewrite.

The fact that β may erase to an arbitrary (well-scoped) given term is called the *Kleene trick* [22] as it goes back to Kleene's numeric realizability: any evidence at all may stand as proof for a trivially true equation (in Cedille, if β is written without some desired erasure t' then by default it erases to $\lambda\, x.\, x$). In practice, when combined with dependent intersections the Kleene trick enables the formation of a kind of *equational subset type* where elements of a carrier type may also act as proof that some equation concerning them holds, provided the equation is indeed true.

Dependent Intersection $\iota\, x{:}T_1.\, T_2$ is the type of terms t which can be seen to have *both* the types T_1 and $[t/x]T_2$. The introduction form $[t_1, t_2]$ is conceptually similar to that of a dependent pair, except that $|t_1|$ must be $\beta\eta$-equivalent to $|t_2|$, thus allowing the erasure of this introduced intersection to simply be $|t_1|$. If t has type $\iota\, x{:}T_1.\, T_2$, then the projections $t.1$ and $t.2$ resp. have types T_1 and $[t.1/x]T_2$. Both projections erase to $|t|$.

We provide a simple example of the technique employed in our construction of definable quotients in Cedille: assume we have the type `Nat` of natural numbers with constructor `zero : Nat` and function `pred : Nat` $\rightarrow$ `Nat` defined in the usual way. Then, the expression $[\mathtt{zero}, \beta\{\mathtt{zero}\}]$ has type $\iota\, x{:}\mathtt{Nat}.\, \{x \simeq \mathtt{pred}\ x\}$ and erases to `zero`.

Implicit Product $\forall\, x : T_1.\, T_2$ is the type of functions whose argument x of type T_1 is *erased* and thus not used to compute the result value of type T_2. It is introduced by $\Lambda\, x.\, t_2$ (which erases to $|t_2|$), provided that t_2 has type T_2 and further that x *does not occur in the erasure of* t_2. If t has type $\forall\, x : T_1.\, T_2$ and t_1 has type T_1, then we may form an *erased application* t -t_1 (which erases to $|t|$) of type $[t_1/x]T_2$. Our use of implicit products in this paper is primarily for clarity of presentation, as in for example Section 4.1 where it allows a function that is *compatible* with an equivalence relation of some carrier type to be lifted to a function over the (definable) quotient in such a way that the lifted function is *definitionally equal* to the original.

Additional term and type constructs not given in Figure 1 are summarized here. All types are quantified over with $\forall$ (such as $\forall\, X : \star.\, X \rightarrow X$) and within terms abstracted over with Λ (such as $\Lambda\, X.\, \lambda\, x.\, x$). Term-to-type and type-to-type applications are written with a center-dot (such as $t \cdot T$). Local definitions are written $[x\ =\ t] - t'$, analogous to `let` x `=` t `in` t' in other languages. Cedille also provides a built-in operator ς for symmetry of equality – this could be replaced by a definition using ρ but is provided for convenience.

2.2 Datatypes in Cedille

CDLE lacks a primitive notion of inductive datatype. Firsov et al. [7] show how these may be derived *generically* (in the sense of for any covariant signature functor). As of version 1.1.0, the Cedille tool incorporates this result by allowing users to declare inductive datatypes with usual notation. For example, natural numbers and pairs can be declared with:

```
data Nat : ⋆ =
  | zero : Nat
  | succ : Nat → Nat.

data Pair (A: ⋆) (B: ⋆) = pair : A → B → Pair.
```

Cedille also has facilities for simple pattern-matching using operator μ', and for combined pattern-matching and recursion with operator μ:

```
fst : ∀ A: ⋆. ∀ B: ⋆. Pair·A·B → A
= Λ A. Λ B. λ p. μ' p { pair a b → a }.

add : Nat → Nat → Nat
= λ m. λ n. μ addN. m { zero → n | succ m' → succ (addN m') }.
```

The operational semantics of μ' is case-branch selection, so for example `fst (pair zero (succ zero))` reduces to `zero`. The operational semantics of μ is combined case-branch selection and fixpoint unrolling. For example, for any m and n of type `Nat`, `add` (`succ` m) n reduces to `succ` (μ' *addN*. m { `zero` $\rightarrow n$ | `succ` m' $\rightarrow$ `succ` (*addN* m')}).

Declared datatypes automatically come with an induction principle invoked by pattern-matching and recursion with μ (and similarly a non-recursive "proof-by-cases" principle invoked by μ'). An example of this is given below in the proof `addZeroRight` showing `zero` is a right-identity of addition.

```
addZeroRight : Π n: Nat. {add n zero ≃ n}
= λ n. μ ih. n @(λ x: Nat. {add x zero ≃ x}) {
  | zero → β
  | succ n' → ρ (ih n') @ y. {succ y ≃ succ n'} - β
  }.
```

Here, a guiding type annotation is given explicitly with `@` to help type check each case branch, with the bound variable `x` replaced with the corresponding constructor pattern. In the `zero` case the expected type is {`add zero zero` $\simeq$ `zero`}, which holds by β (the Cedille tool also considers the operational semantics of μ and μ' when checking convertibility of terms in an equation). In the `succ` case the expected type is {`add` (`succ` n') `zero` $\simeq$ `succ` n'}. A guide for rewriting is given with `@` where the expected type is first checked to be convertible with {`succ` (`add` n' `zero`) $\simeq$ `succ` n'}, then the inductive hypothesis `ih` n' is used to perform a rewrite, and finally β is checked against type {`succ` n' $\simeq$ `succ` n'}

Cedille uses a *type-based* approach to termination checking of recursive functions defined with μ, described in [13]. However, this method sometimes requires type coercions be used explicitly on the recursive subdata revealed in case patterns. Most of the recursive functions and proofs in this paper do not require the full power of Cedille's termination checker, so we remove these type coercions in order to de-clutter our presentations and indicate explicitly those functions for which a syntactic guard [10] would be insufficient to ensure termination.

3 Quotient types by idempotent functions

3.1 General construction

We now present in Cedille quotient types by *idempotent functions*, which we prove in Section 5 precisely characterizes *definable* quotients.

```
IdemFn : ⋆ → ⋆ = λ A: ⋆. ι f: A → A. Π a: A. {f (f a) ≃ f a}.

Quotient : Π A: ⋆. IdemFn·A → ⋆
= λ A: ⋆. λ f: IdemFn·A. ι a: A. {f a ≃ a}.

qcanon : ∀ A: ⋆. Π f: IdemFn·A. A → Quotient·A f
= Λ A. λ f. λ a. [f.1 a, ρ (f.2 a) - β{f.1 a}].
```

For any carrier A, `IdemFn`$\cdot A$ is the type of functions f over A which also prove themselves idempotent. Thanks to the Kleene trick, this obligation amounts to requiring only *that* they are idempotent. Similarly, `Quotient`$\cdot A\ f$ (for any type A and f : `IdemFn`$\cdot A$) is the type of elements of A which are the fixpoints of f, and for any element a of the carrier `qcanon` f maps a to a representative of type `Quotient`$\cdot A\ f$ by simply applying (the first projection of) f to a and discharging the proof obligation that $\{f\ (f\ a) \simeq f\ a\}$ by idempotency of f. The intended equivalence relation $a \sim b$ on A then implicitly arises from the propositional equality $\{$`qcanon` $a \simeq$ `qcanon` $b\}$, and need not be given explicitly.

Our motivation for this characterization of definable quotients is its two-fold simplicity, which we reinforce with examples given in the remainder of this section: first, the number of required components is small, being only a carrier type, unary operation, and proof this operation is idempotent; second, the additional term-level structure packing components with these properties is all *erasable*, convenient for equational reasoning within proofs and especially so when these proofs concern multiple quotient types. From this second feature arises in particular the pleasing fact that every element of the quotient type is *definitionally equal* to some element of the carrier type. This can be demonstrated within Cedille by a kind of internalized subtyping relation: the existence of a coercion `qcoerce` from `Quotient`$\cdot A\ f$ (for every A and f) to A which is definitionally equal to $\lambda\, x.\, x$.

```
qcoerce : ∀ A: ⋆. ∀ f: IdemFn·A. Quotient·A f → A
= Λ A. Λ f. λ q. q.1 .

qcoerceId : {qcoerce ≃ λ q. q} = β.
```

3.2 Typed equality with UIP

An important property of equality within type theory is whether it validates the principle of *uniqueness of identity proofs* (UIP), which is the statement that any two proofs p_1 and p_2 of $\{t_1 \simeq t_2\}$ (for any t_1 and t_2) are themselves equal. The Kleene trick causes Cedille's built-in equality to be anti-UIP because e.g. both $\beta\{\lambda\, x.\, \lambda\, y.\, x\}$ and $\beta\{\lambda\, x.\, \lambda\, y.\, y\}$ prove $\{\lambda\, x.\, x \simeq \lambda\, x.\, x\}$. However, if UIP is desired then it is possible to construct as a definable quotient an equality type `Id` that validates UIP. This construction is simple but rather interesting as, unlike other examples we consider, the carrier of this quotient type has *undecidable* equality (it contains divergent λ-expressions). Yet, this does not impede our choosing a canonical representative – for any proof *eq* we return an equivalent proof which erases to $\lambda\, x.\, x$.

```
eqRep : ∀ A: ⋆. Π a: A. Π b: A. {a ≃ b} → {a ≃ b}
= Λ A. λ a. λ b. λ eq. ρ eq - β.

eqRepIdemFn : ∀ A: ⋆. Π a: A. Π b: A. IdemFn·{a ≃ b}
= Λ A. λ a. λ b. [eqRep a b, λ eq. β].
```

We retain typing information by using indices for the quotient type `Id`.

```
Id : Π A: ⋆. A → A → ⋆
= λ A: ⋆. λ a: A. λ b: A. Quotient·{a ≃ b} (eqRepIdemFn a b).
```

Note that we choose a homogeneous identity type for ease of demonstration, but a heterogeneous or untyped version with UIP is also possible. Finally, we prove that `Id` validates UIP.

```
UIP : ∀ A: ⋆. Π a: A. Π b: A. Π p: Id·A a b. Π q: Id·A a b.
  Id·(Id·A a b) p q
= Λ A. λ a. λ b. λ p. λ q. [ρ ς p.2 - ρ ς q.2 - β, β].
```

3.3 Natural numbers modulo *k*

The natural numbers modulo k is a family of quotient types where two numbers are equivalent modulo k if their remainders with respect to k are equal. Below, we define the remainder function `rem` [1] and show with `remIdem` that it is idempotent. Note that some definitions are omitted (indicated by `<..>`) in the paper but available in the supplementary code repository [2].

[1] The listing of function `rem` omits necessary type coercions for Cedille to ensure that the recursive call on `minus n' k'` is well-founded.

[2] `github.com/cedille/cedille-developments/tree/master/idem-quotients`

```
rem : Nat → Nat → Nat
= λ n. λ k. μ rec. n {
    | zero → zero
    | succ n' →
      [k' = pred k]
      - if (lt n' k') (succ n') (rec (minus n' k'))
  }.
remIdem : Π n: Nat. Π k: Nat. {rem (rem n k) k ≃ rem n k} = <..>

remIdemFn : Π k: Nat. IdemFn·Nat
= λ k. [λ n. rem n k, λ n. ρ+ (remIdem n k) - β{rem n k}].

Mod : Nat → ⋆ = λ k: Nat. Quotient·Nat (remIdemFn k).
```

In the case of `Mod k` the idempotent function `λ n. rem n k` canonicalizes the input natural number to a value in the range $[0, k-1]$.

Functions on the natural numbers can be lifted to `Mod k` either by canonicalizing the output or proving that all outputs of the function are fixpoints of `rem`. For instance, we can lift natural number addition to `Mod k` by coercion from `Mod k` to `Nat`, followed by addition and canonicalization of the output.

```
addMod : Π k: Nat. Mod k → Mod k → Mod k
= λ k. λ n. λ m. qcanon (remIdemFn k) (add n.1 m.1).
```

In [15] Li argues that definable quotients aid in reasoning about the quotient type because both setoid and set views of the data are available. We show that facts about the carrier type that are preserved in the quotient type can be easily demonstrated. Here, it is easy to show that `addMod` is commutative and has an identity element by appealing to the fact that `add` has these properties.

```
addModComm : Π k: Nat. Π a: Mod k. Π b: Mod k.
  {addMod k a b ≃ addMod k b a}
= λ k. λ a. λ b.
  ρ (addComm a.1 b.1) @ x. {rem x k ≃ addMod k b a} - β.

addModIdLeft : Π k: Nat. Π a: Mod k. {addMod k a zero ≃ a}
= λ k. λ a. ρ (addZeroRight a.1) @ x. {rem x k ≃ a}
  - ρ (a.2) - β.

addModIdRight : Π k: Nat. Π a: Mod k. {addMod k zero a ≃ a}
= λ k. λ a. ρ (a.2) - β.
```

Notice in the proofs `addModIdLeft` and `addModIdRight` that we may use `zero` directly when reasoning about it as an identity element for `addMod`. This is possible thanks to Cedille's equality being for untyped terms, and sensible because we know that (for all k) `qcannon` (`remIdemFn` k) `zero` is `zero` *by definition.* Thus, explicit canonicalization of zero is neither required nor desired.

3.4 Even and odd natural numbers

As we have seen, the elements of quotients by idempotent functions in Cedille have (definitionally) equal elements in the carrier type. With this property in mind, we construct the type of even natural numbers and odd natural numbers.

```
toEven : Nat → Nat
= λ n. μ rec. n {
  | zero → zero
  | succ n' → μ' n' {
    | zero → zero
    | succ n'' → succ (succ (rec n''))
    }
  }.
toOdd : Nat → Nat = λ n. succ (toEven n).

toEvenIdem : Π n: Nat. {toEven (toEven n) ≃ toEven n} = <..>
toEvenIdemFn : IdemFn·Nat = <..>

toOddIdem : Π n: Nat. {toOdd (toOdd n) ≃ toOdd n} = <..>
toOddIdemFn : IdemFn·Nat = <..>.

Even : ⋆ = Quotient·Nat toEvenIdemFn.
Odd : ⋆ = Quotient·Nat toOddIdemFn.
```

The idempotent function `toEven` relates every two consecutive natural numbers, picking the smaller number as the canonical representative. The idempotent function `toOdd` is similar but chooses the larger number instead. The pair of idempotent functions `toEven` and `toOdd` define the same equivalence relation with the only difference being which of the two related numbers are picked. This is in contrast to `Mod k` where the equivalence relation alone gives a desired computational behavior or algebraic structure. Indeed, the algebraic structure of `Mod k` is present regardless of the selection of the canonical elements. But for `Even` and `Odd` as quotients we are interested in the particular fixpoints of the functions `toEven` and `toOdd`.

A fundamental property about even numbers is that addition by two produces an even number. By defining addition by two first on the natural numbers we can lift the function to `Even`. However, unlike the lifting we did previously we can avoid having to apply `toEven` on the result and instead prove it is *already* a canonical representative.

```
succSucc : Nat → Nat = λ n. succ (succ n).

evenSSCompat : Π e: Even. {toEven (succSucc e) ≃ succSucc e}
= <..>

evenSuccSucc : Even → Even
= λ e. [succSucc e.1, ρ (evenSSCompat e) - β{succSucc e}].
```

Of course, a similar development can be carried out for odd natural numbers. We call this version of function lifting *compatible*, following Cohen [4]. With compatible lifting the resulting function is definitionally equal to the original function.

A core benefit to our approach is that we can mention elements of the carrier type and the quotient type in equational contexts without any additional type coercions such as projections for dependent records. In a property that decomposes a natural number into an even or odd number we can directly say that the natural number is equal to the corresponding even or odd number (in the return type of `evenOrOdd` below, `Or` is the disjoint union type, and the dependent intersections should be read as a kind of existential quantification):

```
evenOrOdd : Π n: Nat. Or·(ι x: Even. {n ≃ x})·(ι x: Odd. {n ≃ x})
= <..>
```

3.5 List as finite set

As Cohen argued, quotients are a useful feature in formalizing mathematics [4]. However, they can also be a useful abstraction for computer science. As an example, finite sets are usually defined in terms of trees where an order on the elements is needed. With quotients, we can instead form finite sets as an abstraction over those lists whose elements have decidable equality.

```
EqFn : ⋆ → ⋆ = λ A: ⋆. ι f: A → A → Bool.
  (Π a: A. Π b: A. {f a b ≃ true} ⇒ {a ≃ b}).

distinctCons : ∀ A: ⋆. EqFn·A → A → List·A → List·A
= Λ A. λ eq. λ a. λ l. μ' (find eq a l) {
  | tt → l
  | ff → cons a l
  }.

distinct : ∀ A: ⋆. EqFn·A → List·A → List·A
= Λ A. λ eq. λ l. μ rec. l {
  | nil → nil·A
  | cons a l → distinctCons eq a (rec l)
  }.

distinctIdem : ∀ A: ⋆. Π eq: EqFn·A. Π l: List·A.
  {distinct eq (distinct eq l) ≃ distinct eq l} = <..>
distinctIdemFn : ∀ A: ⋆. Π eq: EqFn·A. IdemFn·(List·A) = <..>

ListSet : Π A: ⋆. EqFn·A → ⋆
= λ A: ⋆. λ eq: EqFn·A. Quotient·(List·A) distinctIdemFn.
```

We need an equality function that decides the equality of terms of the parameter type in order to prove that `distinct` is idempotent. A quotient of `List` provides a guarantee about the list that does not alter the underlying structure and does not need to be proven because the list can always be canonicalized. If we allow ourselves an ordering on the elements of A in addition to decidable equality, then we could quotient by a `sort` function to construct a `SortedList` type. Alternatively, we could quotient a tree instead of a list to form a `TreeSet` type.

Throughout the paper, we have highlighted that elements of the quotient type are definitionally equal to certain elements of the carrier type. The advantage of our encoding, and its interaction with Cedille's equality type, is most apparent in the combination of `ListSet` with other definable quotient types. If for example we wished to define a specialized notion of equality between `List·Nat` and `ListSet·Even`, it would be as simple as asking that two terms are equal.

```
EqEvenSet1 : List·Nat → ListSet·Even eqEven → ⋆
= λ l1: List·Nat. λ l2: ListSet·Even eqEven. {l1 ≃ l2}.
```

However, if we were to use an encoding of definable quotients based on a dependent record or pair type, we would be required to use a homogeneous equality type like `Id` (Section 3.2) and explicit coercions between the two sets of quotient and carrier types.

```
EqEvenSet2 : List·Nat → ListSet·Even eqEven → ⋆
= λ l1: List·Nat. λ l2: ListSet·Even eqEven.
  Id·(List·Nat) l1
    (map (qcoerce -evenIdemFn)
       (qcoerce -(distinctIdemFn eqEven) l2)).
```

With `EqEvenSet1` *l1 l2* we know that *l1* and *l2* are (propositionally) equal. An intrinsically typed theory (like Coq or Agda) however must use `EqEvenSet2`, where `qcoerce` would be implemented by record accessors or product projections rather than an identity function. As such, they would remain in proof obligations unless explicitly discharged, meaning for this example we would have to more carefully track coercions for the `ListSet` *and* the `Even` elements when manipulating terms in proofs.

3.6 Quotient inductive integers

In prior examples where the carrier type had an induction principle, we would expect that the quotient type constructed from it is also inductive with respect to some canonicity-preserving constructors. Take for example a non-canonical encoding of integers (which we call the pre-integers).

```
data PreInt : ⋆ =
  | pzero : PreInt
  | psucc : PreInt → PreInt
  | ppred : PreInt → PreInt.
```

When phrased as a quotient inductive type, the definition includes the following axioms.

```
spCancel : Π p: PreInt. {psucc (ppred p) ≃ p}
psCancel : Π p: PreInt. {ppred (psucc p) ≃ p}
```

However, these axioms are false because of how type theories like Cedille, Agda, and Coq encode the constructors of `PreInt`. To fix this problem the axioms must be considered as part of the definition of the type. In systems like Cubical Agda the solution is to extend the notion of inductive types to higher inductive types which are allowed to specify path constructors that may depend on previously defined constructors [24]. The underlying semantics of the system then encodes the type appropriately so that all path constructors are satisfied.

With quotient types by idempotent functions we can take a different approach to the problem by definining canonicity-preserving constructors on the type `PreInt`:

```
psucc' : PreInt → PreInt
= λ p. μ' p {
  | pzero → psucc pzero
  | psucc x → psucc (psucc x)
  | ppred x → x
  }.
ppred' : PreInt → PreInt
= λ p. μ' p {
  | pzero → ppred pzero
  | psucc x → x
  | ppred x → ppred (ppred x)
  }.
```

We do not need to define a `pzero'` constructor because it would be definitionally equal to `pzero`. Next, we define the idempotent function which induces the intended equivalence relation by noticing that it should replace every `PreInt` constructor with the corresponding canonicity-preserving version.

```
integer : PreInt → PreInt
= λ p. μ rec. p {
  | pzero → pzero
  | psucc x → psucc' (rec x)
  | ppred x → ppred' (rec x)
  }.
```

When a quotient type is designed with the constructors (`psucc'` and `ppred'`) first and the canonizer (`integer`) second, then the proof that the canonizer is idempotent is equivalent to knowing that the constructors commute with it. To show that `psucc'` and `ppred'` commute with `integer` we need to know that the equational axioms hold. That is, we need to show that the canonical choice for any pre-integer satisfies the cancellation axioms of the canonicity-preserving constructors.

```
EqSP : PreInt → ⋆ = λ p: PreInt. {psucc' (ppred' p) ≃ p}.
eqSP : Π p: PreInt. EqSP (integer p) = <..>

EqPS : PreInt → ⋆ = λ p: PreInt. {ppred' (psucc' p) ≃ p}.
eqPS : Π p: PreInt. EqPS (integer p) = <..>

integerIdem : Π p: PreInt. {integer (integer p) ≃ integer p}
= <..>
integerIdemFn : IdemFn·PreInt = <..>
```

With these lemmas, `eqSP` and `eqPS`, the function `integer` can be shown idempotent without difficulty.

Next, we define the quotient type `Int`, its corresponding constructors, and prove the cancellation properties.

```
Int : ⋆ = Quotient·PreInt integerIdemFn.

izero : Int = [pzero, β{pzero}].
isucc : Int → Int = <..>
ipred : Int → Int = <..>
sp : Π i: Int. {isucc (ipred i) ≃ i} = <..>
ps : Π i: Int. {ipred (isucc i) ≃ i} = <..>
```

Finally, we can prove an induction principle on `Int` that references the quotient constructors by induction on the underlying `PreInt`.

```
induct : ∀ P: Int → ⋆. P izero →
  (Π x: Int. P x → P (isucc x)) →
  (Π y: Int. P y → P (ipred y)) →
  Π i: Int. P i = <..>
```

Furthermore, we can use the induction principle to define addition on the quotient inductive integers as expected.

```
iadd : Int → Int → Int
= λ x. λ y. induct·(λ x: Int. Int) y
  (λ a. λ b. isucc b)
  (λ a. λ b. ipred b)
  x.
```

In contrast to a Cubical Agda definition of quotient inductive integer our construction does not mention a coherence condition about the equational constraints `ps` and `sp`. Both the coherence condition and the set truncation condition in [20] are not true for the equality type of Cedille. However, the type `PreInt` has a decidable equality. This implies that `Int` also has a decidable equality because every element can be coerced to an element in `PreInt`. Thus, construction of quotient inductive types using idempotent functions will always have decidable equalities if the underlying carrier type does. This means that these types, in a

Homotopy Type Theory setting, are already sets which is why a coherence condition is not needed. Also, it is important to note that quotient inductive types, as described in Cubical Agda, are more expressive than quotients by idempotent functions. Indeed, we have only the *definable* quotient inductive types.

4 Properties of quotient types by idempotent functions

In the literature several desired properties of quotient types are listed. Li lists soundness and completeness of the canonicalization function relative to the equivalence relation as requirements [15]. Cohen lists, additionally, a surjection property and lifting properties [4]. First, we briefly demonstrate that some of these properties trivially hold for quotients by idempotent functions. Second, we demonstrate function and property lifting in Section 4.1.

In this section we will use abbreviations for the idempotent function, carrier type, and quotient type.

```
import quotient-defs.
module quotient (A: ⋆) (f: IdemFn·A).
Q : ⋆ = Quotient·A f.
canon : A → Q = λ a. qcanon f a.
```

Here, `quotient-defs` contains the definitions found in the beginning of Section 3. Now, we define the equivalence relation `Equiv` on `A` that arises from f and show that `canon` is sound and complete (as defined by Li for definable quotients) with respect to it.

```
Equiv : A → A → ⋆ = λ a: A. λ b: A. {f a ≃ f b}.

sound : Π a: A. Π b: A. Equiv a b → {canon a ≃ canon b}
= λ a. λ b. λ eq. eq.

complete : Π a: A. Equiv (f.1 a) a
= λ a. ρ (f.2 a) @ x. {x ≃ f a} - β.
```

In Section 5 we expand on the equivalence between Li's definable quotients and quotients by idempotent functions. It is also straightforward to show the surjection property of Cohen.

```
surjection : Π q: Q. ι a: A. {q ≃ canon a}
= λ q. [q.1, ρ ς q.2 @ a. {a ≃ canon a} - β{q.1}].
```

Function and property lifting are more interesting in Cedille because `Q` terms have corresponding `A` terms that are definitionally equal.

4.1 Function and property lifting

When working with the quotient type `Q`, there may be functions on the carrier type `A` that would be useful to use on `Q`. We have seen this briefly already for both

`Mod k` (where addition on `Nat` was lifted) and `Even` (where applying the successor twice on `Nat` was lifted). These two applications are different. Addition on `Nat` was lifted by restricting the input arguments and canonicalizing the output. Successor twice on `Nat` was lifted by proving that the output returns a canonical element for any canonical input.

We abstract lifting by *canonicalization* to automatically lift any simply typed function on `A` to a simply typed function (with the same shape) on `Q`. This requires that the output of any higher-order inputs are also canonicalized. To accomplish this in Cedille we use an inductive relation `IsSimple`.

```
data IsSimple : (⋆ → ⋆) → ⋆ =
  | base : IsSimple·(λ x: ⋆. x)
  | any : ∀ T: ⋆. IsSimple·(λ x: ⋆. T)
  | arrow : ∀ A: ⋆ → ⋆. ∀ B: ⋆ → ⋆.
    IsSimple·A → IsSimple·B → IsSimple·(λ x: ⋆. A·x → B·x).

liftByCanon : ∀ F: ⋆ → ⋆.
  IsSimple·F → Pair·(F·A → F·Q)·(F·Q → F·A) = <..>
```

This construction allows for both instances of `A` where it is replaced by `Q` (using the `base` constructor) and also instances of `A` that are not replaced (using the `any` constructor). In the unary case applying `liftByCanon` is definitionally equal to applying `canon` on the output of the operation.

```
liftByCanon1 : (A → A) → Q → Q
= λ op. (fst (liftByCanon (arrow base base))) op.

liftByCanon1' : (A → A) → Q → Q
= λ op. λ q. canon (op q.1).

liftByCanon1Eq : Π op: A → A.
  {liftByCanon1 op ≃ liftByCanon1' op}
= λ op. β.
```

Although lifting by canonicalization is very flexible there may be some idempotent functions that are either expensive to compute or would otherwise be unnecessary to re-apply. For example, applying a `filter` function over a `ListSet` would not invalidate the fact that it is a fixpoint of `distinct` but reapplying `distinct` to the output of `filter` will change the complexity from linear to quadratic. This is because `distinct` replaces every `cons` with the `distinct_cons` operation, every application of which destructs and rebuilds (in linear time) the entire list set. To avoid this, an additional compatibility property about the operation to be lifted needs to be proven.

```
Compatible : Π T: ⋆. (T → A) → ⋆
= λ T: ⋆. λ op: T → A. Π t: T. {f (op t) ≃ op t}.
```

```
liftArg : ∀ R: ⋆. Π op: A → R. Q → R
= Λ R. λ op. λ q. op q.1.

lift : ∀ T: ⋆. Π op: T → A. Compatible·T op ⇒ T → Q
= Λ T. λ op. Λ c. λ t. [op t, ρ (c t) - β{op t}].
```

Knowing that the operation is compatible with the idempotent function is only necessary for lifting the return type of the operation `op`. Lifting arguments of the function, as long as they are not higher order arguments, is always possible. With `liftArg` and `lift` binary functions can be lifted by applying inputs to the operation in the compatibility evidence.

```
lift2 : Π op: (A → A → A). (Π a: A. Compatible·A (op a))
  ⇒ Q → Q → Q
= λ op. Λ c. λ x. liftArg (lift (op x.1) -(c x.1)).
```

A similar approach is possible for any n-ary function. As expected, compatible lifting will return a definitionally equal operation.

```
liftArgId : ∀ R: ⋆. Π op: A → R. {liftArg op ≃ op}
= Λ T. λ op. β.

liftId : ∀ T: ⋆. Π op: T → A. {lift op ≃ op}
= Λ T. λ op. β.
```

Aside from lifting functions we also wish to lift properties. Given a property on `A` we lift it to a property on `Q` by forgetting the fixpoint evidence.

```
Lift : (A → ⋆) → Q → ⋆ = λ P: A → ⋆. λ q: Q. P q.1.

dlift : ∀ P: A → ⋆. (Π a: A. P a) → Π q: Q. Lift·P q
= Λ P. λ p. λ q. p q.1.
```

Alternatively, as stated by Hofmann, we have quotient induction where we start with a property on `Q` and show that it holds for all elements of `Q` if it holds for the canonical representatives of elements of `A`.

```
qind : ∀ B: Q → ⋆. (Π a: A. B (canon a)) → Π q: Q. B q
= Λ B. λ c. λ q. ρ ς q.2 - c q.1.
```

Quotient induction lets us prove, by induction on `A`, a fact about the quotient type. However, this is not the same as being able to perform induction directly on `Q` using canonicity-preserving constructors as we showed for quotient inductive integers in Section 3.6.

5 Related work

Quotients have been explored in several existing systems including: Agda, Coq, HOL Light, NuPRL, and others. We survey the existing literature and comment on what is relevant to results presented in this work.

Definable quotients as given by Li [15] are closely related to quotients by idempotent functions. In Li's thesis he formalizes, in Agda, examples of definable quotients and additionally proves that not all quotients of interest are definable. One such example is unordered pairs that lack a total ordering of its components. The type of unordered pairs is also undefinable with an idempotent function. Indeed, the idempotent function must either keep the order of elements or swap the elements and neither choice is fixed without an imposed order.

Li also provides and proves equivalent to definability a notion of *quotients by normalization*. We now show that our formulation of *quotients by idempotent functions* is an equivalent condition to this. The quotient $A/\!\sim$ of a setoid $(A, \sim)$ is definable by normalization if:

1. there is a function $f : A \to A$;
2. which is *sound*, $\forall a, b : A,\ a \sim b \Rightarrow f(a) = g(b)$;
3. and *complete*, $\forall a : A.\ f(a) \sim a$

and definable by an idempotent function if:

1. there is a function $g : A \to A$;
2. which is *idempotent*, $\forall a : A.\ g(g(a)) = g(a)$;
3. and is *image equivalent*, $\forall a, b : A.\ a \sim b \Leftrightarrow g(a) = g(b)$

Theorem 1. *The two conditions above on setoid* $(A, \sim)$ *are equivalent.*

Proof. ($\Rightarrow$) Assume a function f which is sound and complete. We wish to provide some function which is idempotent and image equivalent. We pick f: for all $a : A$, we have by completeness that $f(a) \sim a$, and applying soundness to this yields $f(f(a)) = f(a)$, so f is idempotent. For all $a, b : A$, we have already by soundness that $a \sim b \Rightarrow f(a) = f(a)$, and assuming that $f(a) = f(b)$ we have by completeness that $a \sim f(a) = f(b) \sim b$, so f is image equivalent.

($\Leftarrow$) Assume a function g which is idempotent and image equivalent. We wish to provide some function which is sound and complete. Pick g: we have soundness as a direct consequence of image equivalence. For all $a : A$, we have by idempotence that $g(g(a)) = g(a)$, so by image equivalence we have $g(a) \sim a$ showing g is complete. □

In Coq, Cohen [4] developed two notions of quotient types. The first consists of two functions $pi : Q \to T$ and $repr : T \to Q$ where $\forall x : T.\ pi(repr(x)) = x$; much like as in our presentation, the equivalence relation $x \sim y$ then arises from the equality $repr\ x = repr\ y$. The second notion arises from carrier types T with a *choice structure* [9] which guarantees that, for every equivalence relation $\sim\, : T \to T \to Prop$ there exists a canonical choice operation $canon : T \to T$. In translating from the second notion of quotient to the first, Cohen shows that choice structure guarantees that *canon* is idempotent and defines the quotient type Q as a dependent record containing an element of T and a proof that it is a fixpoint of *canon*. Though this is similar to quotients by idempotent functions, we start with the requirement that the canonical choice operation is idempotent rather than deriving it as a consequence of the seemingly stronger requirement that T has a choice structure.

Moreover, because Coq is an intensional type theory the packaging of the dependent record will not be erased when reasoning about terms of the quotient type Q. Also, the lack of a truly heterogeneous equality type (as opposed to *John Major* equality [17]) in Coq will prevent the direct equational reasoning between carrier and quotient type that is possible in Cedille. This situation is also the same for constructing quotient by idempotent functions in Agda: even using *Prop* or irrelevant record fields so that Q is in a sense a subtype of T, it is not the case that every q of type Q is definitionally equal to some element of type T, and so coercions between these the two must be managed explicitly when performing equational reasoning.

Quotient types in type theory have been studied as early as the 1990s with Hofmann's work on interpreting quotient types in both predicative and impredicative variants of the Calculus of Constructions [11]. Hofmann's work is expanded upon by Veltri who works with impredicative encodings and some additional primitive types to show versions of dependent lifting for quotients [23]. The approach of utilizing normalization is explored in Courtieu's work where he expands the Calculus of Inductive Constructions with type constructors for "normalized types" [6].

Outside of intensional proof assistants like Coq and Agda, Nogin has worked on modular definitions of quotients in the NuPRL system to ease the development burden when using quotients [19]. Prior to Nogin's work NuPRL included quotients as a primitive construct. In modern NuPRL types are identified as partial equivalence relations and quotient types are constructed from this interpretation directly [2]. Quotients are also defined and used in HOL Light and similar systems [12].

In this work we have focused on definable quotient types, but there are several interesting quotients that do not fit into this category. For instance, higher inductive types (of which quotient inductive types are a special case) have been used to model type theory in type theory [1] and finite sets [8]. With the existence of a small core of higher inductive types (one of which is the higher inductive quotient), all set-truncated higher inductive types have been shown to be derivable [25]. Although there are some quotient inductive types that can be modeled as quotients by idempotent functions (such as the quotient inductive integers) it is clear that quotient inductive types are a more expressive formalism.

The presence of non-definable quotients in type theories can have significant consequences. Indeed, Maietti demonstrates that when effective quotients are added to constructive set theory and two universes are postulated that the law of the excluded middle holds for small sets [16]. Likewise, Chicli et al. show in Coq that if quotients of functions spaces are available, where all such quotients have a section mapping, and there is an impredicative universe then the theory is inconsistent [3]. With a theory like Cedille that does not have a universe hierarchy and has impredicative quantification, caution must be used in extending the theory with undefinable quotients (or more generally higher inductive types) as it could make the theory inconsistent.

6 Conclusions

In this work we have described a novel and relatively simple characterization of definable quotient types by idempotent functions, and described an encoding of it within Cedille. We have presented concrete examples of quotient types: an equality type with UIP, naturals modulo k, the even and odd subset of naturals, finite sets (and their combination with even numbers), and a quotiented integer type with an induction principle. We have also developed function lifting operations, showing that in particular compatible functions can be lifted to a definitionally equal function over the quotient type. Moreover, dependent intersection and the Kleene trick in Cedille allow full erasure of the *packaging* of elements of a carrier type with proofs they are fixpoints of some idempotent function, meaning no explicit coercions between the quotient and carrier type are needed for equational reasoning, as would be the case for a similar encoding in other dependently typed languages like Coq and Agda.

We are interested in expanding on this work by investigating what equational constraints for higher inductive types would always guarantee that it is a definable quotient inductive type, and thus be derivable within CDLE. Also, in Cedille a notion of (Mendler-style) histomorphism is derivable that allows for more flexibility in recursive definitions [7]. We have shown that induction is possible in terms of the quotient constructors, but we also want to extend this result to histomorphisms on the quotient type.

References

1. Altenkirch, T., Kaposi, A.: Type theory in type theory using quotient inductive types. In: POPL'16, Proceedings of the 43rd Annual ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages. vol. 51, pp. 18–29. ACM (2016). https://doi.org/10.1145/2837614.2837638
2. Anand, A., Bickford, M., Constable, R.L., Rahli, V.: A type theory with partial equivalence relations as types. In: 20th International Conference on Types for Proofs and Programs (2014)
3. Chicli, L., Pottier, L., Simpson, C.: Mathematical quotients and quotient types in Coq. In: International Workshop on Types for Proofs and Programs. pp. 95–107. Springer (2002). https://doi.org/10.1007/3-540-39185-1_6
4. Cohen, C.: Pragmatic quotient types in Coq. In: International Conference on Interactive Theorem Proving. pp. 213–228. Springer (2013). https://doi.org/10.1007/978-3-642-39634-2_17
5. Coquand, T., Huet, G.: The calculus of constructions. Ph.D. thesis, INRIA (1986)
6. Courtieu, P.: Normalized types. In: International Workshop on Computer Science Logic. pp. 554–569. Springer (2001). https://doi.org/10.1007/3-540-44802-0_39
7. Firsov, D., Blair, R., Stump, A.: Efficient Mendler-style lambda-encodings in Cedille. In: International Conference on Interactive Theorem Proving. pp. 235–252. Springer (2018)

8. Frumin, D., Geuvers, H., Gondelman, L., Weide, N.v.d.: Finite sets in homotopy type theory. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 201–214. ACM (2018)
9. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: International Conference on Theorem Proving in Higher Order Logics. pp. 327–342. Springer (2009)
10. Giménez, E.: Codifying guarded definitions with recursive schemes. In: Dybjer, P., Nordström, B., Smith, J. (eds.) Types for Proofs and Programs. pp. 39–59. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
11. Hofmann, M.: A simple model for quotient types. In: International Conference on Typed Lambda Calculi and Applications. pp. 216–234. Springer (1995). https://doi.org/10.1007/BFb0014055
12. Homeier, P.V.: Quotient types. pp. 191–206 (2001)
13. Jenkins, C., McDonald, C., Stump, A.: Elaborating inductive datatypes and course-of-values pattern matching to Cedille. CoRR (2019), `http://arxiv.org/abs/1903.08233`
14. Kopylov, A.: Dependent intersection: A new way of defining records in type theory. In: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science. pp. 86–. LICS '03, IEEE Computer Society, Washington, DC, USA (2003), `http://dl.acm.org/citation.cfm?id=788023.789073`
15. Li, N.: Quotient types in type theory. Ph.D. thesis, University of Nottingham (2015)
16. Maietti, M.E.: About effective quotients in constructive type theory. In: International Workshop on Types for Proofs and Programs. pp. 166–178. Springer (1998). https://doi.org/10.1007/3-540-48167-2_12
17. McBride, C.: Elimination with a motive. In: International Workshop on Types for Proofs and Programs. pp. 197–216. Springer (2000)
18. Miquel, A.: The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In: Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications. pp. 344–359. TLCA'01, Springer-Verlag, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-45413-6_27
19. Nogin, A.: Quotient types: A modular approach. In: International Conference on Theorem Proving in Higher Order Logics. pp. 263–280. Springer (2002). https://doi.org/10.1007/3-540-45685-6_18
20. Pinyo, G., Altenkirch, T.: Integers as a higher inductive type. In: 24th International Conference on Types for Proofs and Programs (2018)
21. Stump, A.: The calculus of dependent lambda eliminations. Journal of Functional Programming **27**, e14 (2017). https://doi.org/10.1017/S0956796817000053
22. Stump, A.: Syntax and semantics of Cedille (2018), `https://arxiv.org/abs/1806.04709`
23. Veltri, N.: Two set-based implementations of quotients in type theory. In: Proceedings of the 14th Symposium on Programming Languages and Software Tools. pp. 194–205 (2015)
24. Vezzosi, A., Mörtberg, A., Abel, A.: Cubical agda: a dependently typed programming language with univalence and higher inductive types. vol. 3, p. 87. ACM (2019)
25. van der Weide, N., Geuvers, H.: The construction of set-truncated higher inductive types. In: Thirty-fifth Conference on the Mathematical Foundations of Programming Semantics