Two Set-based Implementations of Quotients in Type Theory

Niccolò Veltri

Institute of Cybernetics, Tallinn University of Technology
Akadeemia tee 21, 12618 Tallinn, Estonia,
niccolo@cs.ioc.ee

Abstract. We present and compare two different implementations of quotient types in Intensional Type Theory. We first introduce quotients as particular inductive-like types following Martin Hofmann's extension of Calculus of Constructions with quotient types [6]. Then we give an impredicative encoding of quotients. This implementation is reminiscent of Church numerals and more generally of encodings of inductive types in Calculus of Constructions.

1 Introduction

In mathematics, given a set X and an equivalence relation R on X, the quotient set X/R is the set of equivalence classes of X with respect to R, i.e. $X/R = \{[x] \,|\, x \in X\}$, where $[x] = \{y \in X \,|\, x \, R \, y\}$. An important example is the set of integer numbers, constructed as the quotient set $(\mathbb{N} \times \mathbb{N})/\mathsf{SameDiff}$, where $\mathsf{SameDiff}\,(n_1, m_1)\,(n_2, m_2)$ if and only if $n_1 + m_2 = n_2 + m_1$. Another example is the set of real numbers, constructed as the quotient set $\mathsf{Cauchy}_{\mathbb{Q}}/\mathsf{Diff}_{\to 0}$, where $\mathsf{Cauchy}_{\mathbb{Q}}$ is the set of Cauchy sequences of rational numbers and $\mathsf{Diff}_{\to 0}\,\{x_n\}\,\{y_n\}$ if and only if the sequence $\{x_n - y_n\}$ converges to 0. A fundamental usage of quotients in programming is the construction of finite multisubsets of a given type X as "lists modulo permutations", and of finite subsets of X as "lists modulo permutations and multiplicity".

In Martin-Löf type theory (MLTT) [8] and in Calculus of Inductive Constructions (CIC) [1] quotients are typically represented by setoids. A setoid is a pair (A, R) where A is a set and R is an equivalence relation on A. A map between setoids (A, R) and (B, S) is a map $f : A \to B$ compatible with the relations, i.e. if $a \, R \, b$ then $(f a) \, S \, (f b)$. Every set A can be represented as the setoid $(A, \equiv)$, where $\equiv$ is propositional equality on A. Given an equivalence relation R on A the quotient A/R is represented as the setoid (A, R). There is a canonical setoid map $\mathsf{abs} : (A, \equiv) \to (A, R)$, $\mathsf{abs} = \mathsf{id}$, that is clearly compatible, and every setoid map $f : (A, \equiv) \to (B, S)$ such that $(f a) \, S \, (f b)$ whenever $a \, R \, b$ extends to a setoid map $\mathsf{lift}\, f : (A, R) \to (B, S)$.

The implementation of quotients as setoids forces us to lift every type former to setoids. For example the type formers of products, function spaces, lists and trees must become setoid transformers. Moreover in several applications it is preferable to work with sets instead of setoids.

In this paper we present two different frameworks for reasoning about set-based quotients, i.e. quotients as types. We first introduce in Section 2 quotients as particular inductive-like types. The presentation is inspired by quotient types in Martin Hofmann's PhD thesis [6], and works fine both in MLTT and in CIC. Our presentation is settled in MLTT. In Section 3 we show an alternative encoding of quotients in a small extension of Calculus of Constructions (CC). The two implementations are pretty different in flavor. We highlight their main features and show some examples. In Section 4 we present integer numbers as the quotient of $\mathbb{N} \times \mathbb{N}$ mentioned in the introduction, and in Section 5 we present finite multisubsets of a given type X as the quotient of $\mathsf{List}\, X$ also mentioned above. The presentations work fine both in MLTT and in our extension of CC.

Note that integer numbers are already definable in type theory without the need of quotient types. In MLTT, for example, integers are implemented as two distinct copies of natural numbers $\mathbb{N} + \mathbb{N}$, interpreted as the negative and non-negative numbers. Note that in order to avoid the presence of two zeros, the elements of the first copy of $\mathbb{N}$ have to be considered as "shifted by one", i.e. $\mathsf{inl}\, n$ has to be read as $-(n + 1)$. Another possibility is to introduce integers as the type $\top + \mathbb{N} + \mathbb{N}$, specifying zero explicitly and "shifting by one" both copies. Using such implementations, defining operations on integers and proving that such operations satisfy the laws of arithmetic (e.g. $\mathbb{Z}$ is a integral domain) become tedious due to the number of cases involved in the definitions. In Section 4 we want to show that our implementation is more elegant and less tedious to work with than the other two presented above.

We have fully formalized the results of this paper in the dependently typed programming language Agda [9]. The formalization is available at `http://cs.ioc.ee/~niccolo/quotients/`. In order to be consistent with the formalization, in this paper we use the notation of MLTT.

2 Inductive-Like Quotients

In this section, we introduce quotient types as particular inductive-like types introduced by M. Hofmann [6]. First we briefly describe the type theory under consideration.

2.1 The Type Theory under Consideration

We consider Martin-Löf type theory (MLTT) with inductive types and a cumulative hierarchy of universes $\mathcal{U}_k$. We allow dependent functions to have implicit arguments and indicated implicit argument positions with curly brackets (as in Agda). We write $\equiv$ for propositional equality (identity types) and $=$ for judgmental (definitional) equality. Reflexivity, symmetry, transitivity and substitutivity of $\equiv$ are named refl, sym, trans and subst, respectively.

We assume *uniqueness of identity proofs* for all types, i.e., an inhabitant for

$$\mathsf{UIP} = \prod_{\{X:\mathcal{U}\}} \prod_{\{x_1,x_2:X\}} \prod_{p_1,p_2:x_1 \equiv x_2} p_1 \equiv p_2.$$

A type X is said to be a *proposition*, if it has at most one inhabitant, i.e., if the type

$$\mathsf{isProp}\,X = \prod_{x_1,x_2:X} x_1 \equiv x_2$$

is inhabited.

Uniqueness of identity proofs is needed only to prove that the propositional truncation of a type is a proposition (Subsection 2.4), which in turn is needed in the proof of Proposition 1.

2.2 The Implementation

We now describe quotient types à la Hofmann. We call them "inductive-like quotients" because they are given a dependent elimination principle (sometimes also called induction principle). Let X be a type and R an equivalence relation on X. For any type Y and function $f : X \to Y$, we say that f is *R-compatible* (or simply *compatible*, when the intended equivalence relation is clear from the context), if the type

$$\mathsf{compat}\,f = \prod_{\{x_1,x_2:X\}} x_1 R x_2 \to f\,x_1 \equiv f\,x_2$$

is inhabited. The quotient of X by the relation R is described by the following data:

(i) a carrier type X/R;
(ii) a constructor $\mathsf{abs} : X \to X/R$ together with a proof $\mathsf{sound} : \mathsf{compat}\,\mathsf{abs}$;
(iii) a dependent eliminator: for every family of types $Y : X/R \to \mathcal{U}_k$ and function $f : \prod_{x:X} Y\,(\mathsf{abs}\,x)$ with $p : \mathsf{dcompat}\,f$, there exists a function $\mathsf{lift}\,f\,p : \prod_{q:X/R} Y\,q$;
(iv) a computation rule: for every family of types $Y : X/R \to \mathcal{U}_k$, function $f : \prod_{x:X} Y\,(\mathsf{abs}\,x)$ with $p : \mathsf{dcompat}\,f$ and $x : X$ we have

$$\mathsf{lift}_\beta\,f\,p\,x : \mathsf{lift}\,f\,p\,(\mathsf{abs}\,x) \equiv f\,x$$

The predicate $\mathsf{dcompat}$ represents compatibility for dependent functions $f : \prod_{x:X} Y\,(\mathsf{abs}\,x)$:

$$\mathsf{dcompat}\,f = \prod_{\{x_1,x_2:X\}} \prod_{r:x_1 R x_2} \mathsf{subst}\,Y\,(\mathsf{sound}\,r)\,(f\,x_1) \equiv f\,x_2.$$

We postulate the existence of data (i)–(iv) for all types X and equivalence relations R on X. Notice that the predicate $\mathsf{dcompat}$ depends on the availability of sound. Also notice that, in (iii), we allow elimination on every universe $\mathcal{U}_k$. In our development, we actually eliminate only on $\mathcal{U}$ and once on $\mathcal{U}_1$ (Proposition 1).

We now take a look at some derived results and examples.

2.3 Classical Quotients

Classically every equivalence class in a quotient X/R has a representative element in the original set, i.e. a map $\mathsf{rep} : X/R \to X$ that satisfies the following conditions:

$$\begin{aligned}\mathsf{complete} : \prod_{x:X} (\mathsf{rep}\,(\mathsf{abs}\,x))\,R\,x \\ \mathsf{stable} : \prod_{q:X/R} \mathsf{abs}\,(\mathsf{rep}\,q) \equiv q\end{aligned}$$

If we postulate the existence of such quotients for all sets and equivalence relations it is possible to derive the law of excluded middle [2].

In general in constructive mathematics, for a given equivalence class there is no canonical choice of a representative. This idea is reflected in the implementation of quotients we presented in the previous section. Every map of type $X/R \to X$ is of the form $\mathsf{lift}\,f\,p$ for a certain R-compatible map $f : X \to X$. But for a general type X and equivalence relation R strictly weaker then equality, there is no such canonical f.

2.4 Propositional Truncation

The *propositional truncation* (or *squash*) $\|X\|$ of a type X is the quotient of X by the total relation $\lambda\,x_1\,x_2.\,\top$. Intuitively $\|X\|$ is the unit type $\top$ if X is inhabited and it is empty otherwise. In other words, $\|X\|$ is the proposition associated with the type X. Indeed:

$$\begin{aligned}&\mathsf{isProp}_{\|\|} : \mathsf{isProp}\,\|X\| \\ &\mathsf{isProp}_{\|\|}\,x_1\,x_2 = \mathsf{lift}\,(\lambda\,y_1.\,\mathsf{lift}\,(\lambda\,y_2.\,\mathsf{sound}\,\star)\,p_1\,x_2)\,p_2\,x_1\end{aligned}$$

where $\star : \top$ is the constructor of the unit type, while p_1 and p_2 are simple compatibility proofs. Note that in these compatibility proofs we need to show that two equality proofs are equal, and we do it by using the uniqueness of identity proofs.

Note that the propositional truncation operation defines a monad: the unit is $|\_|$ and multiplication $\mu_{\|\|} : \|\|X\|\| \to \|X\|$ is defined as $\mu_{\|\|} = \mathsf{lift}\,\mathsf{id}\,p$, where p is the easy proof of compatibility that follows from the fact that $\|X\|$ is a proposition. In general, for a given family of equivalence relations $R_X : X \to X \to \mathcal{U}$, indexed by $X : \mathcal{U}$, the functor $F\,X = X/R_X$ is not a monad, since there is no way of constructing a multiplication $\mu : (X/R_X)/R_{X/R_X} \to X/R_X$.

2.5 Function Extensionality

Let X and Y be types. Extensional equality of functions is an equivalence relation on $X \to Y$:

$$\begin{aligned}&\mathsf{FunExt}_{\equiv} : (X \to Y) \to (X \to Y) \to \mathcal{U} \\ &\mathsf{FunExt}_{\equiv}\,f\,g = \prod_{x:X} f x \equiv g x\end{aligned}$$

For the quotient $(X \to Y)/\mathsf{FunExt}_{\equiv}$ there exists a map that associates a representative function to each equivalence class.

$$\begin{aligned}&\mathsf{rep}\ : (X \to Y)/\mathsf{FunExt}_{\equiv} \to (X \to Y)\\ &\mathsf{rep}\, q\, x = \mathsf{lift}\,(\lambda f.\, f\, x)\,(\lambda p.\, p\, x)\, q\end{aligned}$$

Using the computation rule lift_{β} of quotients we obtain $\mathsf{rep}\,(\mathsf{abs}\, f)\, x \equiv f\, x$, for all $f : X \to Y$ and $x : X$. The computation rule holds only up to propositional equality. If equality in lift_{β} were definitional, one could prove, using rep, the principle of function extensionality. Indeed, consider $f, g : X \to Y$ with $\mathsf{FunExt}_{\equiv}\, f\, g$. Then the following sequence of equations holds:

$$\begin{aligned}f &= \lambda x.\, f\, x = \lambda x.\, \mathsf{rep}\,(\mathsf{abs}\, f)\, x = \mathsf{rep}\,(\mathsf{abs}\, f)\\ &\equiv \mathsf{rep}\,(\mathsf{abs}\, g) = \lambda x.\, \mathsf{rep}\,(\mathsf{abs}\, g)\, x = \lambda x.\, g\, x = g\end{aligned}$$

2.6 Effectiveness

A quotient X/R is said to be *effective*, if the type $\prod_{x_1,x_2:X} \mathsf{abs}\, x_1 \equiv \mathsf{abs}\, x_2 \to x_1\ R\ x_2$ is inhabited. In general, effectiveness does not hold for all quotients. Moreover, postulating effectiveness for all quotients implies the law of excluded middle [7]. Clearly classical quotients, discussed in Subsection 2.3, are effective. Indeed, if for $x_1, x_2 : X$ we have $\mathsf{abs}\, x_1 \equiv \mathsf{abs}\, x_2$ then, using $\mathsf{complete}$ we are done, since $\mathsf{rep}\,(\mathsf{abs}\, x_1)\, R\, x_1$, $\mathsf{rep}\,(\mathsf{abs}\, x_2)\, R\, x_2$ and $\mathsf{rep}\,(\mathsf{abs}\, x_1) \equiv \mathsf{rep}\,(\mathsf{abs}\, x_2)$.

For a general type X and a general equivalence relation R on X, we can only prove that, under the assumption of proposition extensionality, the quotient X/R satisfies a weaker property. The principle of *proposition extensionality* states that logically equivalent propositions are equal:[1]

$$\mathsf{PropExt} = \prod_{\{X,Y:\mathcal{U}\}} \mathsf{isProp}\, X \to \mathsf{isProp}\, Y \to X \leftrightarrow Y \to X \equiv Y$$

where $X \leftrightarrow Y = (X \to Y) \times (Y \to X)$. We say that a quotient X/R is *weakly effective*, if the type $\prod_{x_1,x_2:X} \mathsf{abs}\, x_1 \equiv \mathsf{abs}\, x_2 \to \|x_1\ R\ x_2\|$ is inhabited.

If we extend our type theory with $\mathsf{PropExt}$, we can prove that all quotients are weakly effective.

Proposition 1. *Under the hypothesis of proposition extensionality, all quotients are weakly effective.*

Proof. In fact, let X be a type, R an equivalence relation on X and $x : X$. Consider the function $\|x\ R\ \_\| : X \to \mathcal{U}$, $\|x\ R\ \_\| = \lambda\, x'.\, \|x\ R\ x'\|$. We show that $\|x R \_\|$ is R-compatible. Let $x_1, x_2 : X$ with $x_1 R x_2$. We have $x R x_1 \leftrightarrow x R x_2$ and therefore $\|x\, R\, x_1\| \leftrightarrow \|x\, R\, x_2\|$. Since propositional truncations are propositions

[1] Note that proposition extensionality is accepted in homotopy type theory [12]. Propositions are (-1)-types and proposition extensionality is univalence for (-1)-types.

(proof $\mathsf{isProp}_{\|}$ in Subsection 2.4), using proposition extensionality, we conclude $\|x\,R\,x_1\| \equiv \|x\,R\,x_2\|$. We have constructed a term $p_x : \mathsf{compat}\,\|x\;R\;\_\|$, and therefore a function $\mathsf{lift}\,\|x\;R\;\_\|\,p_x : X/R \to \mathcal{U}$ (large elimination is fundamental in order to apply lift, since $\|x\;R\;\_\| : X \to \mathcal{U}$ and $X \to \mathcal{U} : \mathcal{U}$). Moreover, $\mathsf{lift}\,\|x\;R\;\_\|\,p_x\,(\mathsf{abs}\,y) \equiv \|x\;R\;y\|$ by its computation rule.

Let $\mathsf{abs}\,x_1 \equiv \mathsf{abs}\,x_2$ for some $x_1, x_2 : X$. We have:

$$\|x_1\;R\,x_2\| \equiv \mathsf{lift}\,\|x_1 R\,\_\,\|\,p_{x_1}\,(\mathsf{abs}\,x_2) \equiv \mathsf{lift}\,\|x_1\;R\;\_\,\|\,p_{x_1}\,(\mathsf{abs}\,x_1) \equiv \|x_1\;R\,x_1\|$$

and $x_1\;R\,x_1$ holds, since R is reflexive. □

3 Impredicative Encoding of Quotients

In this section, we present an implementation of quotients in Calculus of Constructions (CC). The implementation is different in flavor from the one discussed in Section 2.

3.1 The Type Theory under Consideration

Remember that our presentation is done using the language of MLTT. Our Agda formalization makes use of type-in-type instead of Agda's current implementation of universe polymorphism. This means that we are working in a type theory with only one universe $\mathcal{U}$ and $\mathcal{U} : \mathcal{U}$. Type-in-type is known to be inconsistent [5, 3], but we are using it only to simulate in Agda the impredicativity of CC, which is consistent.

In Subsection 3.3 we need the existence of dependent sums and identity types. Both are definable in CC. Consider $X : \mathcal{U}$ and $P : X \to \mathcal{U}$. The dependent sum $\sum_{x:X} P\,x$ can be defined as follows:

$$\sum_{x:X} P\,x = \prod_{Y:\mathcal{U}} \left(\prod_{x:X} P\,x \to Y\right) \to Y$$

Consider $X : \mathcal{U}$ and $x_1, x_2 : X$. We can define (*Leibniz*) equality $x_1 \equiv x_2$ as follows:

$$x_1 \equiv x_2 = \prod_{P:X\to\mathcal{U}} P\,x_1 \to P\,x_2$$

One can easily define the constructor and the first projection map of dependent sums.

$$\begin{aligned}&\mathsf{pair} : \prod_{x:X} \left(P\,x \to \sum_{x:X} P\,x\right)\\ &\mathsf{pair}\,x\,p = \lambda Y\;f.\,f\,x\,p\end{aligned}$$

$$\begin{aligned}&\mathsf{fst} : \sum_{x:X} P\,x \to X\\ &\mathsf{fst}\,c = c\,X\,(\lambda x\,p.\,x)\end{aligned}$$

It is also possible to prove that Leibniz equality is a substitutive equivalence relation. But is not possible to construct the second projection map $\mathsf{snd} : \prod_{c:\sum_{x:X} P\,x} P\,(\mathsf{fst}\,c)$, showing that the type $\sum_{x:X} P\,x$ defined above is a *weak* dependent sum. Leibniz equality is also weak, since it is not possible to prove "dependent substitutivity", i.e. given a type X, a family of types $Y : X \to \mathcal{U}$ and a predicate $P : \prod_{x:X} \to Y\,x \to \mathcal{U}$, we cannot construct a term subst_2 of type

$$\prod_{p:x_1 \equiv x_2} \mathsf{subst}\,Y\,p\,y_1 \equiv y_2 \to P\,x_1\,y_1 \to P\,x_2\,y_2$$

for all $x_1, x_2 : X$, $y_1 : Y\,x_1$ and $y_2 : Y\,x_2$.

The results of Subsection 3.3 rely on the existence of terms snd and subst_2. Therefore we extend CC with identity types and dependent sums as primitives. As a consequence we obtain that the terms snd and subst_2 are easily definable. An instance of subst_2 gives us sufficient conditions for proving equality of pairs. Let X be a type and $P : X \to \mathcal{U}$ a family of types. Then for all $x_1, x_2 : X$, $p_1 : P\,x_1$ and $p_2 : P\,x_2$:

$$\begin{aligned}
\mathsf{pair}_{\equiv} &: \prod_{r:x_1 \equiv x_2} \mathsf{subst}\,P\,r\,p_1 \equiv p_2 \to \mathsf{pair}\,x_1\,p_1 \equiv \mathsf{pair}\,x_2\,p_2 \\
\mathsf{pair}_{\equiv}\,r\,s &= \mathsf{subst}_2\,(\lambda x\,p.\,\mathsf{pair}\,x_1\,p_1 \equiv \mathsf{pair}\,x\,p)\,r\,s\,\mathsf{refl}
\end{aligned}$$

We also assume the dependent version of the principle of function extensionality, i.e. there is a term $\mathsf{dfunext}$ that inhabits the type

$$\mathsf{DFunExt} = \prod_{\{X:\mathcal{U}\}} \prod_{\{Y:X\to\mathcal{U}\}} \prod_{\{f_1\,f_2:\prod_{x:X}\to Y\,x\}} \left(\prod_{x:X} f_1\,x \equiv f_2\,x \right) \to f_1 \equiv f_2$$

3.2 The Implementation

We now describe our impredicative implementation of quotients. Let X be a type and R an equivalence relation on X. We define the quotient of X over R as the following type:

$$X/R = \prod_{Y:\mathcal{U}} \prod_{f:X\to Y} \mathsf{compat}\,f \to Y$$

In other words, X/R is a polymorphic function which assigns, to every type Y equipped with a compatible function $f : X \to Y$, an element of Y. One can then define the constructor abs:

$$\begin{aligned}
&\mathsf{abs} : X \to X/R \\
&\mathsf{abs}\,x = \lambda Y\,f\,r.\,f\,x
\end{aligned}$$

Using the principle of function extensionality one proves that abs is an R-compatible map. Notice that the dependent version of the principle of function extensionality is needed here, since elements of type X/R are dependent maps.

$$\begin{aligned}&\mathsf{sound} : \mathsf{compat}\,\mathsf{abs}\\&\mathsf{sound}\, r = \mathsf{dfunext}\,(\lambda Y.\, \mathsf{dfunext}\,(\lambda f.\, \mathsf{dfunext}\,(\lambda p.\, p\, r)))\end{aligned}$$

One can then define the non-dependent elimination principle, which turns out to be just function application. Crucially the computation rule holds definitionally, as witnessed below in the observation that refl proves the corresponding propositional equality.

$$\begin{aligned}&\mathsf{lift} : \prod_{\{Y : \mathcal{U}\}} \prod_{f : X \to Y} \mathsf{compat}\, f \to X/R \to Y\\&\mathsf{lift}\,\{Y\}\, f\, r\, q = q\, Y\, f\, p\end{aligned}$$

$$\begin{aligned}&\mathsf{lift}_\beta : \prod_{\{Y : \mathcal{U}\}} \prod_{f : X \to Y} \prod_{r : \mathsf{compat}\, f} \prod_{x : X} \mathsf{lift}\, f\, p\, (\mathsf{abs}\, x) \equiv f\, x\\&\mathsf{lift}_\beta\, f\, r\, x = \mathsf{refl}\end{aligned}$$

Note the similarity with Church numerals and the implementation of dependent sums given above, and more generally the similarity with the impredicative encoding of inductive types in CC [10]. Moreover this representation is inspired by the impredicative encoding of higher inductive types [12, Ch. 6] in CIC [11].

3.3 Dependent Elimination

While in practice having a definitional computation rule is convenient, it is impossible to derive a dependent elimination principle. Implementations of inductive types in CC in general suffer from this problem [4].

In this subsection we assume the *uniqueness property* of lift i.e. the fact that, for every type Y and R-compatible function $f : X \to Y$, $\mathsf{lift}\, f\, r$ is the only map that makes the following diagram commute:

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "\mathsf{abs}"'] & Y \\
X/R \arrow[ur, dashed, "\mathsf{lift}\, f\, r"'] &
\end{tikzcd}$$

From the uniqueness property we derive the dependent elimination principle. Let $Y : X/R \to \mathcal{U}$ be a type family and $f : \prod_{x:X} Y(\mathsf{abs}\, x)$ a map with compatibility proof $r : \mathsf{dcompat}\, f$. Using the non-dependent eliminator we define a map of type $X/R \to \sum_{q:X/R} Y\, q$.

$$\begin{aligned}&\mathsf{dlift}' : \prod_{\{Y : X \to \mathcal{U}\}} \prod_{f : \prod_{x:X} Y\, (\mathsf{abs}\, x)} \mathsf{dcompat}\, f \to X/R \to \sum_{q : X/R} Y\, q\\&\mathsf{dlift}'\, f\, r = \mathsf{lift}\,(\lambda x.\, \mathsf{pair}\,(\mathsf{abs}\, x)\,(f\, x))\,(\lambda p.\, \mathsf{pair}_{\equiv}\,(\mathsf{sound}\, p)\,(r\, p))\end{aligned}$$

Notice that, for all $x : X$, $\mathsf{fst}\,(\mathsf{dlift}'\,f\,r\,(\mathsf{abs}\,x)) = \mathsf{abs}\,x = \mathsf{id}\,(\mathsf{abs}\,x)$. Therefore, by the uniqueness property, we obtain a term $s_q : \mathsf{fst}\,(\mathsf{dlift}'\,f\,r\,q) \equiv q$ for all $q : X/R$. This allows us to derive the dependent elimination principle:

$$
\begin{aligned}
&\mathsf{dlift} : \prod_{\{Y : X \to \mathcal{U}\}} \prod_{f : \prod_{x:X} Y\,(\mathsf{abs}\,x)} \mathsf{dcompat}\,f \to \prod_{q : X/R} Y\,q \\
&\mathsf{dlift}\,\{Y\}\,f\,r\,q = \mathsf{subst}\,Y\,s_q\,(\mathsf{snd}\,(\mathsf{dlift}'\,f\,r\,q))
\end{aligned}
$$

4 Integer Numbers

As an example we present integer numbers. In order to do that we need to have natural numbers in our system (defined as Church numerals in CC or defined inductively in MLTT, it does not matter). We introduce a synonym for pairs of natural numbers, $\mathsf{Diff} = \mathbb{N} \times \mathbb{N}$, and we use the notation $\_-\_$ for the constructor of Diff. Elements of Diff represent differences of natural numbers. We define an equivalence relation $\mathsf{SameDiff}$ on Diff relating pairs with the same difference:

$$
\begin{aligned}
&\mathsf{SameDiff} : \mathsf{Diff} \to \mathsf{Diff} \to \mathcal{U} \\
&\mathsf{SameDiff}\,(n_1 - m_1)\,(n_2 - m_2) = \mathsf{plus}\,n_1\,m_2 \equiv \mathsf{plus}\,n_2\,m_1
\end{aligned}
$$

where plus is addition on $\mathbb{N}$. We define $\mathbb{Z} = \mathsf{Diff}/\mathsf{SameDiff}$. We show formally that $\mathbb{Z}$ is a commutative monoid. The unit $\mathsf{zero}_{\mathbb{Z}}$ is the equivalence class of $\mathsf{zero}_{\mathsf{Diff}} = \mathsf{zero} - \mathsf{zero}$, where zero is the unit of $\mathbb{N}$. Addition is defined in two steps. First we introduce an addition operation on Diff.

$$
\begin{aligned}
&\mathsf{plus}_{\mathsf{Diff}} : \mathsf{Diff} \to \mathsf{Diff} \to \mathsf{Diff} \\
&\mathsf{plus}_{\mathsf{Diff}}\,(n_1 - m_1)\,(n_2 - m_2) = \mathsf{plus}\,n_1\,n_2 - \mathsf{plus}\,m_1\,m_2
\end{aligned}
$$

Before lifting addition to $\mathbb{Z}$, we introduce a useful variant of compat_2, the compatibility predicate for two-argument functions. Let X, Y and Z be types and R, S and T equivalence relations on X, Y and Z respectively. The predicate compat_2' on $X \to Y \to Z$ is defined as follows:

$$
\mathsf{compat}_2'\,f = \prod_{\{x_1, x_2 : X\}} \prod_{\{y_1, y_2 : Y\}} x_1\,R\,x_2 \to y_1\,S\,y_2 \to (f\,x_1\,y_1)\,T\,(f\,x_2\,y_2)
$$

A function f satisfies compat_2' if it sends R-related and S-related inputs to T-related outputs. It is easy to construct a proof $p : \mathsf{compat}_2'\,\mathsf{plus}_{\mathsf{Diff}}$. We are ready to lift the addition $\mathsf{plus}_{\mathsf{Diff}}$ to $\mathbb{Z}$:

$$
\begin{aligned}
&\mathsf{plus}_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \\
&\mathsf{plus}_{\mathbb{Z}} = \mathsf{lift}_2\,(\lambda\,d\,e.\,\mathsf{abs}\,(\mathsf{plus}_{\mathsf{Diff}}\,d\,e))\,(\lambda\,r\,s.\,\mathsf{sound}\,(p\,r\,s))
\end{aligned}
$$

where lift_2 is the two-argument version of lift. We prove the right unit law. First notice that the law holds in Diff up to $\mathsf{SameDiff}$, i.e. for all $d : \mathsf{Diff}$, we have a proof s_d : $\mathsf{SameDiff}\,(\mathsf{plus}_{\mathsf{Diff}}\,d\,(\mathsf{zero}-\mathsf{zero}))\,d$. We lift this proof to $\mathbb{Z}$:

$$\mathsf{rightUnit}_{\mathbb{Z}} : \prod_{z:\mathbb{Z}} \mathsf{plus}_{\mathbb{Z}}\,z\,\mathsf{zero}_{\mathbb{Z}} \equiv z$$
$$\mathsf{rightUnit}_{\mathbb{Z}} = \mathsf{absEpi}\,(\lambda\,d.\,\mathsf{sound}\,s_d)$$

where absEpi is a proof that the map $\mathsf{abs} : X \to X/R$ is an epimorphism, for all types X and equivalence relations R on X, i.e. for all types Y and maps $f_1, f_2 : X/R \to Y$, if $f_1\,(\mathsf{abs}\,x) \equiv f_2\,(\mathsf{abs}\,x)$ for all $x : X$, then for all $q : X/R$ we have $f_1\,q \equiv f_2\,q$. This is an easy consequence of the uniqueness property.

We observe that working with impredicative quotients facilitates proofs, since the computation rule holds definitionally.

5 Finite Multisubsets

Another example we present is finite multisubsets of a given type X. In this section we work in MLTT. Let X be a type with decidable equality, i.e. there exists a function $\mathsf{dec}_{\equiv} : X \to X \to \mathsf{Bool}$ such that $\mathsf{dec}_{\equiv}\,x_1\,x_2 = \mathsf{true}$ if and only if $x_1 \equiv x_2$. We introduce the binary relation Perm on $\mathsf{List}\,X$, inductively defined by the rules:

$$\frac{}{\mathsf{Perm}\,[]\,[]} \qquad \frac{\mathsf{Perm}\,xs\,ys}{\mathsf{Perm}\,(x :: xs)\,(x :: ys)}$$

$$\frac{\mathsf{Perm}\,xs\,ys}{\mathsf{Perm}\,(x :: y :: xs)\,(y :: x :: ys)} \qquad \frac{\mathsf{Perm}\,xs\,ys \quad \mathsf{Perm}\,ys\,zs}{\mathsf{Perm}\,xs\,zs}$$

Two lists xs and ys are in the relation Perm if xs is a permutation of ys. The relation is transitive by construction, and it is easily provable reflexive and symmetric. Therefore we form the quotient $\mathsf{Multisubset}\,X = \mathsf{List}\,X/\mathsf{Perm}$, i.e. a finite multisubset of X is a list modulo permutations.

We introduce a function counting the multiplicity of an element x in a list xs. If the element does not belong to the list, then its multiplicity is zero. Note that decidable equality on X is fundamental in order to count the number of occurrences of x in xs.

$$\mathsf{multiplicity} : X \to \mathsf{List}\,X \to \mathbb{N}$$
$$\mathsf{multiplicity}\,x\,[] = \mathsf{zero}$$
$$\mathsf{multiplicity}\,x\,(y :: xs)\,\mathsf{with}\,\mathsf{dec}_{\equiv}\,x\,y$$
$$\mathsf{multiplicity}\,x\,(y :: xs)\,|\,\mathsf{true} = \mathsf{suc}\,(\mathsf{multiplicity}\,x\,xs)$$
$$\mathsf{multiplicity}\,x\,(y :: xs)\,|\,\mathsf{false} = \mathsf{multiplicity}\,x\,xs$$

The function $\mathsf{multiplicity}$ can be proved compatible with the relation Perm. This is true since permuting a list does not alter the number of occurrences of an element in it. The proof is easily done by induction on the structure of Perm. Therefore the function $\mathsf{multiplicity}$ lifts to $\mathsf{Multisubset}\,X$.

We conclude this section by noting that there are other possible definitions of "equality" on finite multisubsets of X. For example one could define a relation Perm' on $\mathsf{List}\,X$ as $\mathsf{Perm}'\,xs\,ys = \prod_{x:X}\,(x \in xs) \cong (x \in ys)$, where $\cong$ is type isomorphism and $\in$ is list membership. The definition of Perm' is more concise that the definition of Perm. The two relations are logically equivalent, but proving $\mathsf{multiplicity}$ compatible with Perm' is much more complicated than proving $\mathsf{multiplicity}$ compatible with Perm.

6 Conclusions

In this paper we showed two different implementation of quotient types. Both are set-based and therefore different from the setoid-based approach.

In Section 2 we presented inductive-like quotients in Martin-Löf type theory. They do not need impredicativity in order to be introduced, but their existence has to be postulated. Moreover the computation rule only holds up to propositional equality. Hofmann's extension of Calculus of Constructions [6] is consistent, therefore the same holds for our implementation in MLTT.

In Section 3 we presented an impredicative encoding of quotients in Calculus of Constructions. In order to derive the dependent elimination principle from the uniqueness property we need to extend CC with dependent sums and identity types. Our implementation shows that, at the cost of impredicativity, quotient types are definable. However they are "weak", similarly to Leibniz equality or the impredicative encoding of dependent sums given in Subsection 3.2. To get "strong" quotients, one needs to introduce postulates, such as the uniqueness property. Geuvers [4] showed that postulating dependent elimination for impredicative encodings of inductive types is safe. Similarly this can be extended to our quotient types. The uniqueness property of quotients is logically equivalent to the dependent elimination principle, therefore assuming the uniqueness property is also safe. There are other ways of deriving the dependent elimination principle for inductive types in impredicative systems such as CC, most notably parametricity [13].

Acknowledgement This research was supported by the ERDF funded ICT national programme project "Coinduction", the Estonian Science Foundation grant no. 9219 and the Estonian Ministry of Education and Research institutional research grant no. PUT33-13.

References

1. Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer, 2004.
2. L. Chicli, L. Pottier, and C. Simpson. Mathematical quotients and quotient types in Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2003.
3. T. Coquand. An analysis of Girard's paradox. In *Symposium on Logic in Computer Science*, pages 227–236. IEEE Computer Society, 1986.

4. H. Geuvers. Induction is not derivable in second order dependent type theory. In S. Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 166–181. Springer, 2001.
5. J.-Y. Girard. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur, Ph.D. thesis, Université Paris VII, 1972.
6. M. Hofmann. Extensional concepts in intensional type theory, Ph.D. thesis, University of Edinburgh, 1995.
7. M. Maietti. About effective quotients in constructive type theory. In T. Altenkirch, B. Reus, and W. Naraschewski, editors, *Types for Proofs and Programs*, volume 1657 of *Lecture Notes in Computer Science*, pages 166–178. Springer, 1999.
8. B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's type theory.* Oxford University Press Oxford, 1990.
9. U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, and S. D. Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2009.
10. F. Pfenning and C. Paulin-Mohring. Inductively defined types in the calculus of constructions. In M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 1989.
11. M. Shulman. Higher inductive types via impredicative polymorphism. Blog post, 2011. `http://homotopytypetheory.org/2011/04/25/higher-inductive-types-via-impredicative-polymorphism`.
12. The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study, 2013. `http://homotopytypetheory.org/book`.
13. P. Wadler. The Girard–Reynolds isomorphism. *Theoretical Computer Science*, 375(1):201–226, 2007.