Quotient Types — a Modular Approach ⋆,⋆⋆

Aleksey Nogin

Department of Computer Science
Cornell University
Ithaca, NY 14853
E-mail: nogin@cs.cornell.edu
Web: http://nogin.org/

Abstract. In this paper we introduce a new approach to defining quotient types in type theory. We suggest replacing the existing monolithic rule set by a modular set of rules for a specially chosen set of primitive operations. This modular formalization of quotient types turns out to be very powerful and free of many limitations of the traditional monolithic formalization. To illustrate the advantages of the new formalization, we show how the type of collections (that is known to be very hard to formalize using traditional quotient types) can be naturally formalized using the new primitives. We also show how modularity allows us to reuse one of the new primitives to simplify and enhance the rules for the set types.

1 Introduction

NuPRL type theory differs from most other type theories used in theorem provers in its treatment of equality. In Coq's Calculus of Constructions, for example, there is a single global equality relation which is not the desired one for many types (e.g. function types). The desired equalities have to be handled explicitly, which is quite burdensome. As in Martin-Löf type theory [17] (of which NuPRL type theory is an extension), in NuPRL each type comes with its own equality relation (the extensional one in the case of functions), and the typing rules guarantee that well–typed terms respect these equalities. Semantically, a quotient in NuPRL is trivial to define: it is simply a type with a new equality.

Such quotient types proved to be a powerful and useful mechanism for natural formalization of various notions in type theory. For example, rational numbers can be naturally formalized as a quotient type of the type of pairs of integer numbers (which would represent a numerator and a denominator of a fraction) with the appropriate equality predicate.

Somewhat surprisingly, it turns out that formulating rules for these quotient types is far from being trivial and numerous applications of NuPRL [6,16] have run into difficulties. Often a definition involving a quotient will look plausible,

⋆ This work was partially supported by AFRL grant F49620-00-1-0209 and ONR grant N00014-01-1-0765
⋆⋆ Cornell University Technical Report TR2002-1869, April 2002

but after some (sometimes substantial) work it turns out that a key property is unprovable, or false.

A common source of problems is that in NuPRL type theory the equality predicate enjoys only a trivial proof witness. This means that even when we know that two elements are equal in a quotient type, we can not in general recover the witness of the equality predicate. In other words, $a = b \in (A /\!/ E)$ (where "$A /\!/ E$" is a quotient of type A with equivalence relation E) does not always imply $E[a;b]$ (however it does imply $\neg\neg E[a;b]$).

Another common class of problems occurs when we consider some predicate P on type A. Because we are dealing with a constructive theory, even if we can show that $P[a] \Leftrightarrow P[b]$ for any $a, b \in A$ such that $E[a;b]$, $P[a]$ can still have proof witnesses different from those that $P[b]$ has, so P would not be, strictly speaking, a predicate on the quotient type $A /\!/ E$ [1].

These problems suggest that there is more to concept of quotient types, than just the idea of changing the equality relation of a type. In this paper we show how we can decompose the concept of quotient type into several simpler concepts and to formalize quotient types based on formalization of those simpler concepts.

We claim that such a "decomposed" theory makes operating with quotient types significantly easier. In particular we show how the new type constructors can be used to formalize the notion of indexed collections of objects. We also claim that the "decomposing" process makes the theory more modular. In particular, we show how to reuse one of the new type constructors to improve and simplify the rules for the set type.

For each of the new (or modified) type constructors, we present a set of derivation rules for this type — both the axioms to be added to the type theory and the rules that can be derived from these axioms. As we will explain in Section 3, the particular axioms we use were carefully chosen to make the theory as modular as possible and to make them as usable as possible in a tactic–based interactive prover. All the new rules were checked and found valid in S. Allen's semantics of type theory [1,2]. Proofs of all the derived rules were developed and checked in the MetaPRL system [12,13].

Although this paper focuses on NuPRL type theory, the author believes that many ideas presented here are relevant to managing witnessing and functionality information in a constructive setting in general.

The paper is organized as follows. First, in Sections 1.1 through 1.3 we will describe some features of NuPRL type theory that are necessary for understanding this work. Sections 2, 4, 5, 6 and 7 present a few new primitive type constructors and show how they help to formulate the rules for quotient and set types; Section 3 explains our approach to choosing particular axioms; Section 8 shows how to use the new type constructors to formalize the notion of collections; Appendix A lists the NuPRL-4 type theory rules relevant to quotient types.

---

[1] It will be a function from $A /\!/ E$ to $\mathbf{Prop} /\!/ \Leftrightarrow$, rather than a predicate (a function from $A /\!/ E$ to $\mathbf{Prop}$).

1.1 Propositions–as–Types

NuPRL type theory adheres to the *propositions–as–types principle*. This principle means that a proposition is identified with the type of all its witnesses. A proposition is considered true if the corresponding type is inhabited and is considered false otherwise. In this paper we will use words "*type*" and "*proposition*" interchangeably; same with "*witness*" and "*member*".

1.2 Partial Equivalence Relations Semantics

The key to understanding the idea of quotient types is understanding the most commonly used semantics of the NuPRL type theory (and some other type theories as well) — the PER (partial equivalence relations) semantics [24,1,2]. In PER semantics each type is identified with a set of objects and an equivalence relation on that set that serves as an *equality relation* for objects of that type. This causes the equality predicate to be three–place: "$a = b \in C$" stands for "a and b are equal elements of type C", or, semantically, "a and b are related by the equality relation of type C".

Remark 1.1. Note that in this approach an object is an element of a type *iff* it is equal to itself in that type. This allows us to identify $a \in A$ with $a = a \in A$.

According to PER approach, whenever something ranges over a certain type, it not only has to span the whole type, it also has to respect the equality of that type.

Example 1.2. In order for a function f to be considered a function from type A to type B, not only for every $a \in A$, $f(a)$ has to be B, but also whenever a and a' are equal in the type A, $f(a)$ should be equal to $f(a')$ in the type B. Note that in this example the second condition is sufficient since it actually implies the first one. However it is often useful to consider the first condition separately.

Example 1.3. Now consider a set type $T := \{x : A \mid B(x)\}$. Similarly to Example 1.2 above, in order for T to be a well–formed type, not only $B(a)$ has to be a well–formed type for any $a \in A$, but also for any $a = a' \in A$ it should be the case that $B(a)$ and $B(a')$ are equal types.

1.3 Extensional and Intensional Approaches

In this paper we devote significant amount of attention to discussion of choices between what we call *intensional* and *extensional* approaches to certain type operators. The difference between these approaches is in deciding when two objects should be considered equal. In general, in the intensional approach two objects would be considered equal if their *internal* structure is the same, while in the extensional approach two objects would be considered equal if they exhibit the same external behavior.

Example 1.4. In NuPRL type theory the function equality is extensional. Namely, we say that $f = f' \in (A \to B)$ *iff* they both are in $A \to B$ and for all $a \in A$ $f(a) = f'(a) \in B$.[2]

Example 1.5. It is easy to define an extensional equality on types — $A =_e B$ *iff* A and B have the same membership and equality relations. However, in NuPRL type theory the main equality relation on types is intensional. For example, if A and B are two non-empty types, then $(A \to \bot) = (B \to \bot)$ only when $A = B$, even though we have $(A \to \bot) =_e (B \to \bot)$ since they are both empty types.[3]

Example 1.6. When introducing certain type constructors, such as a set (cf. Example 1.3 above) or a quotient (cf. Section 7) one, into NuPRL type theory, there are often two choices for an equality definition:

Completely intensional. The predicates have to be equal, for example $\{x : A \mid B(x)\} = \{x : A' \mid B'(x)\}$ iff $A = A'$ and for all $a = a' \in A$, $B(a){=}B'(a')$.

Somewhat extensional. The predicates have to be equivalent, for example $\{x : A \mid B(x)\} = \{x : A' \mid B'(x)\}$ iff $A = A'$ and for all $a = a' \in A$, $B(a){\Leftrightarrow}B'(a')$.

Essentially, in the intensional case the map $x \leadsto B(x)$ has to respect A's equality relation in order for $\{x : A \mid B(x)\}$ to be well–formed and in the extensional case $B(x)$ only needs to respect it up to $\Leftrightarrow$.

We will continue the discussion of the differences between these two choices in Sections 2.3 and 7.1.

2 `Squash` Operator

2.1 `Squash` Operator: Introduction

For each type A we define a type $[A]$ ("squashed A") which is empty *if and only if* A is empty and contains a single element $\bullet$ [4] when A is inhabited. Informally one can think of $[A]$ as a proposition that says that A is a *non-empty type*, but "squashes down to a point" all the information on *why* A is non-empty. The `squash` operator is *intensional*, e.g. $[A] = [B]$ iff $A = B$ (see also Remark 2.2).

Remark 2.1. We could define `squash` operator as $[A] := \{Unit \mid A\}$. Note that it does not make sense for us to actually add such a definition to the system since we want to formalize the set type using the `squash` operator and not the other way around.

[2] It is interesting to note that this causes the type $\bot \to \bot$ (where $\bot$ is an empty type) to be a type that all functions belong to and are all equal in.

[3] Strictly speaking, NuPRL type theory does not contain Martin-Löf's "$A = B$" judgment form. Instead, NuPRL uses proposition of the form $A = B \in \mathbb{U}_i$ where $\mathbb{U}_i$ is the i-th universe of types. However in this paper we will often omit "$\in \mathbb{U}_i$" for simplicity.

[4] MetaPRL system uses the unit element () or "it" as a $\bullet$, NuPRL uses `Ax` and [23] uses *Triv*.

The `squash` operator (sometimes also called `hide`) was introduced in [9]. It is also used in MetaPRL [11,12,13] [5].

In the next section we will present the axiomatization we chose for the `squash` operator and we will explain our choices in Section 3.

2.2 Squash Operator: Axioms

First, whenever A is non-empty, $[A]$ must be non-empty as well:

$$\frac{\Gamma \vdash A}{\Gamma \vdash [A]} \qquad (\mathit{SquashIntro})$$

Second, if we know $[A]$ and we are trying to prove an equality (or a membership) statement, we can allow "unhiding" contents of A and continue with the proof:

$$\frac{\Gamma;\, x : A;\, \Delta \vdash t_1 = t_2 \in C}{\Gamma;\, x : [A];\, \Delta \vdash t_1 = t_2 \in C} \qquad (\mathit{SquashElim})$$

(assuming x does not occur freely in Δ, t_i and C [6]). This rule is valid because in Martin-Löf type theory equality has no *computational context* and is always witnessed by $\bullet$, so knowing the witness of A does not add any "additional power".

Finally, the only possible element of a `squash` type is $\bullet$:

$$\frac{\Gamma;\, x : [A];\, \Delta[\bullet] \vdash C[\bullet]}{\Gamma;\, x : [A];\, \Delta[x] \vdash C[x]} \qquad (\mathit{SquashMemElim})$$

As mentioned in the introduction, all these new axioms can be proved sound in Allen's semantics of type theory [1,2]. All soundness proofs are very straightforward, and we omit them in this paper. We omit some purely technical axioms (such as well–formedness ones) that are unnecessary for understanding this work.

2.3 Squash Operator: Derived Rules

Here are the rules that can be derived from the axioms we have introduced above. First, whenever $[A]$ is non-empty, $\bullet$ must be in it:

$$\frac{\Gamma \vdash [A]}{\Gamma \vdash \bullet \in [A]} \qquad (\mathit{SquashMemIntro})$$

[5] In MetaPRL `squash` was first introduced by J.Hickey as a replacement for NuPRL's hidden hypotheses mechanism, but eventually it became clear that it gives a mechanism substantially widely useful than NuPRL's hidden hypotheses.

[6] We use the sequent schema syntax of [19] for specifying rules. Essentially, variables that are explicitly mentioned may occur freely only where they are explicitly mentioned.

Second, using (*SquashMemElim*) we can prove a stronger version of (*SquashElim*):

$$\frac{\Gamma;\ x : A;\ \Delta[\bullet] \ \vdash\ t_1[\bullet] = t_2[\bullet] \in B[\bullet]}{\Gamma;\ x : \left[A\right];\ \Delta[x] \ \vdash\ t_1[x] = t_2[x] \in B[x]} \qquad (\mathit{SquashElim2})$$

Third, we can prove that squashed equality implies equality:

$$\frac{\Gamma \ \vdash\ \left[t_1 = t_2 \in A\right]}{\Gamma \ \vdash\ t_1 = t_2 \in A} \qquad (\mathit{SquashEqual})$$

Remark 2.2. Note that if we would have tried to make the `squash` operator extensional, we would have needed an extra well–typeness assumption in the (*SquashElim*) rule (as we had to do in (*EsquashElim*) rule in Section 7.2) which would have made it useless for proving well–typeness and membership statements. In particular, the (*SquashEqual*) rule (as well as any reasonable modification of it) would not have been valid.

Next, we can prove that if we can deduce a witness of a type A just by knowing that some unknown x is in A (we call such A a *squash–stable* type), then $\left[A\right]$ implies A:

$$\frac{\Gamma \ \vdash\ \left[A\right] \qquad \Gamma;\ x : A \ \vdash\ t \in A}{\Gamma \ \vdash\ A} \qquad (\mathit{SquashStable})$$

Remark 2.3. The notion of *squash stability* is also discussed in [15] and is very similar to the notion of *computational redundancy* [5, Section 3.4].

Finally, we can prove that we can always eliminate the squashes in hypotheses not only when the conclusion is an equality (as in (*SquashElim*) and (*SquashElim2*)), but also when it is a squash [7]:

$$\frac{\Gamma;\ x : A;\ \Delta[\bullet] \ \vdash\ \left[C[\bullet]\right]}{\Gamma;\ x : \left[A\right];\ \Delta[x] \ \vdash\ \left[C[x]\right]} \qquad (\mathit{Unsquash})$$

3 Choosing the Rules

When choosing a particular set of axioms we were using several general guidelines. First, in a context of an interactive tactic-based theorem prover it is very important to ensure that each rule is formulated in a *reversible* way whenever possible. By reversible rule we mean a rule where conclusion is valid *if and only if* the premises are valid. This means that it is always "safe" to apply such a rule when (backward) searching for a proof of some sequent — there is no "danger" that back–chaining through the rule would turn a provable statement into a statement that is no longer true. This property allows us to add such

[7] In general, it is true whenever the conclusion is squash–stable.

rules to proof tactics more freely without having to worry about a possibility that applying such tactic can bring the proof into a "dead end" [8]. For example, among the `squash` axioms of Section 2.2 only (*SquashIntro*) is irreversible and the other axioms are reversible.

Second, we wanted to make sure that each rule makes the smallest "step" possible. For example, the (*SquashElim*) rule only eliminates the `squash` operator, but does not attempt to eliminate the witness of the `squash` type while the (*SquashMemElim*) only eliminates the witness of the `squash` type and does not attempt to eliminates the `squash` operator. This gives users a flexibility to "steer" proofs exactly where they want them to go. Of course, we often do want to make several connected steps at once, but that can be accomplished by providing derived rules [9] while still retaining the flexibility of the basic axioms. For example, the (*SquashElim2*) allows one to both eliminate the `squash` operator and its witness in a single step, while still using (*SquashElim*) or (*SquashMemElim*) when only one and not the other is needed. As we will see in Section 4 this "small step" requirement is especially important for the irreversible rules.

Finally, it is important for elimination rules to match corresponding introduction rules in their "power"[10]. Such balance helps insure that most rules are reversible not only with respect to validity, but also with respect to provability (which is obviously needed to make applying such rules truly "safe" in a theorem prover).

4 Intensional Set Type

4.1 Set Type: Introduction

Almost everything in this section equally applies to dependent ($\{x : A \mid P[x]\}$) and independent ($\{A \mid P\}$) set types, but for simplicity we only talk about independent set types.

Set types were first introduced in [7] and were also formalized in [3,9,20]. Informally, $\{A \mid B\}$ is the same type as A when B is a true proposition and it is empty when B is a false proposition. The key property of set type is that when we have a witness $w \in \{A \mid B\}$, we know that $w \in A$ and we know that B is non-empty; but in general we have no way of reconstructing a witness for B.

[8] Of course, some tactics are designed to fall back if they fail to find a complete derivation for the statement being proved and such tactics would not become dangerous if we allow them to use an irreversible rule (although they still might become more likely to fail). But if a tactic is only meant to propel the proof further without necessarily completing it (such as for example NuPRL's `Auto` and MetaPRL's `autoT`), then allowing such tactic to use irreversible rules can make things substantially less pleasant to the user.

[9] See [19] for a description of MetaPRL's derived rules mechanism.

[10] More specifically, elimination rules should be *locally complete* and *locally sound* with respect to the introduction rules, as described in [22]. But since we believe that this third guideline is not as crucial as the first two, we chose not provide a detailed discussion of it.

In the traditional implementation of type theory the rules for set types are somewhat asymmetric. To prove something like

$$\frac{\Gamma;\, x : A \vdash A' \qquad \Gamma;\, y : B \vdash B'}{\Gamma;\, x : \{A \mid B\} \vdash \{A' \mid B'\}}$$

we had to apply the set elimination rule before the set introduction rule. As we will see in a moment, the problem was that the traditional set introduction rule is irreversible and would go "too far" if we apply it right away. It would yield a subgoal $\Gamma;\, x : \{A \mid B\} \vdash B'$ that would only be valid if we could reconstruct the proof witness of B' without having access to proof witness of B.

4.2 Set Type: New Axioms

Using the `squash` operator we only need[11] the following two simple axioms to formalize the set type:

$$\frac{\Gamma;\, x : A;\, y : [B];\, \Delta[x] \vdash C[x]}{\Gamma;\, x : \{A \mid B\};\, \Delta[x] \vdash C[x]} \qquad (\mathit{SetElim})$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash [B]}{\Gamma \vdash \{A \mid B\}} \qquad (\mathit{SetIntro})$$

Now we can explain the problem with the traditional approach [8,9,20,23] — there the set introduction rule is somewhat analogous to applying (*SetIntro*) and then as much (*Unsquash*) as possible and then (*SquashIntro*). Such rule does too many things at once and one of those things (*SquashIntro*) is irreversible. Because of that with such rule we can deconstruct the set operator in the conclusion only when the irreversible part of this rule would not render the resulting subgoals unprovable.

The reason this traditional formalization required a rule that does so much at once was the lack of a way to express the intermediate results. In a sense, in that implementation, set (and quotient) types had at least two separate "jobs" — one was to change the type membership (equality) and another — to hide the proof of the membership (equality) predicate. And there was only a single collection of rules for both of the "jobs", which made the rules hard to use.

The `squash` operator now takes over the second "job" which allows us to express the properties of each of the two jobs in a separate set of rules. Our rules (*SetElim*) and (*SetIntro*) are now both reversible, both perform only a singly small step of the set type and they exactly match each other. The set introduction rule now does only that — introduces the set type into the conclusion of the sequent and leaves it to the `squash` rules (such as (*Unsquash*) and (*SquashIntro*)) to manage the "hiding/unhiding the proof predicate" aspect. We believe this makes the theory more modular and easier to use.

[11] As in Section 2.2 we omit some unessential axioms.

5 Extensional `Squash` Operator (`Esquash`)

5.1 `Esquash` Operator: Introduction

As we mentioned in Remark 2.2, the `squash` operator needs to be intensional. However, we also need to be able to express the notion of extensionally — as we will see in Section 7, even the intensional quotient type has some extensionality in it. In order to make the theory modular, we want to express the concept of the extensionality directly, not through some complex operator for which the extensionality is just a "side-effect". In order to achieve this, we will define a new operation, called `esquash`, that acts very similar to `squash` except that two "esquashed" types are equal whenever they are simultaneously non-empty or simultaneously empty. In oder words, `esquash` completely "hides" not only the witnesses of a type, but also the intensional structure of a type, leaving only the information on whether a type is non-empty or not.

5.2 `Esquash` Operator: Axioms

First, equality — two `esquash` types are equal *iff* they are simultaneously true or simultaneously false:

$$\frac{\Gamma \vdash [\![A]\!] \Leftrightarrow [\![B]\!]}{\Gamma \vdash [\![A]\!] = [\![B]\!]} \qquad (\mathit{EsquashEquality})$$

Second, `esquash` of an intensional type is equivalent to `squash` [12]:

$$\frac{\Gamma \vdash [\![A]\!] \qquad \Gamma \vdash A\,\mathrm{Type}}{\Gamma \vdash \left[A\right]} \qquad (\mathit{EsquashElim})$$

$$\frac{\Gamma \vdash \left[A\right]}{\Gamma \vdash [\![A]\!]} \qquad (\mathit{EsquashIntro})$$

Finally, the only member of a non-empty `esquash` type is $\bullet$:

$$\frac{\Gamma;\, x : [\![A]\!];\, \Delta[\bullet] \vdash C[\bullet]}{\Gamma;\, x : [\![A]\!];\, \Delta[x] \vdash C[x]} \qquad (\mathit{EsquashMemElim})$$

Remark 5.1. We could define the `esquash` operator as

$$[\![A]\!]_i \;:=\; A = \mathrm{True} \in \left(x, y : \mathbb{U}_i /\!/ (x \Leftrightarrow y)\right) \; .$$

Unfortunately, this definition increases the universe level. With this definition if $A \in \mathbb{U}_i$, then $[\![A]\!]_i$ is in $\mathbb{U}_{i+1}$. This can create many difficulties, especially when we want to be able to iterate the `esquash` operator. And in any case we want to formalize quotient types using the `esquash` operator, not the other way around.

[12] $x : T \vdash [\![A[x]]\!]$ only requires $A[x]$ to be non-empty when $x \in T$. However since `squash` is intensional, $x : T \vdash \left[A[x]\right]$ also requires $A[x] = A[x']$ when $x = x' \in T$. Because of this we need the well–typeness condition in (*EsquashElim*).

Remark 5.2. In MetaPRL J. Hickey had initially defined `esquash` operator using the extensional quotient type [13]: $[\![A]\!] := \mathtt{tt} = \mathtt{ff} \in (x, y : \mathbb{B}/\!\!/_{e}(x = y \in \mathbb{B} \vee A))$ [14]. This definition does not increase the universe level like the previous one, but on the other hand it requires an extensional quotient type while the previous one works with both intensional and extensional quotients. Another problem with this definition is that almost all NuPRL-4 rules on quotient types require one to prove that the equality predicate is actually intensional, so it would be impossible to prove the properties of `esquash` from this definition using NuPRL-4 rules.

5.3 `Esquash` Operator: Derived Rules

First, using (*EsquashMemElim*) we can prove that any non-empty `esquash` type has an $\bullet$ in it:

$$\frac{\Gamma \vdash [\![A]\!]}{\Gamma \vdash \bullet \in [\![A]\!]} \qquad (\textit{EsquashMemIntro})$$

Second, we can derive a more general and complex version of (*EsquashElim*):

$$\frac{\Gamma;\, x : [A];\, \Delta[x] \vdash B[x] \qquad \Gamma;\, x : [\![A]\!];\, \Delta[x] \vdash A\,\mathrm{Type}}{\Gamma;\, x : [\![A]\!];\, \Delta[x] \vdash B[x]} \qquad (\textit{EsquashElim2})$$

6 Explicit Nondeterminicity

6.1 Explicit Nondeterminicity: Introduction

The idea of introducing explicit nondeterminicity first came up as a way to be able to express the elimination rules for quotient types in a more natural way, but it seems that it is also a useful tool to have on its own.

At first, we considered adding the $\mathsf{nd}\{\}$ operation similar to *amb* in [18] and to approach used in [14]. The idea was to have $\mathsf{nd}\{t_1; t_2\}$ which can be either t_1 or t_2 nondeterministically. Then we were going to say that the expression that contains $\mathsf{nd}\{\}$ operators is well–formed iff its meaning does not depend on choosing which of $\mathsf{nd}\{\}$'s arguments to use. Unfortunately, the problem with such an approach is that we need some way of specifying that several occurrences of the same $\mathsf{nd}\{t_1; t_2\}$ have to be considered together — either all of them would go to t_1 or all of them would go to t_2. For example, we can say that $\mathsf{nd}\{1; -1\}^2 = 1 \in \mathbb{Z}$ (which is true), but if we expand the ^{2} operator, we will get $\mathsf{nd}\{1; -1\} * \mathsf{nd}\{1; -1\} = 1 \in \mathbb{Z}$ which is only true if we require both $\mathsf{nd}\{\}$'s in it to expand to the same thing.

The example above suggests using some index on $\mathsf{nd}\{\}$ operator, which would keep track of what occurrences of $\mathsf{nd}\{\}$ should go together. In such a case it is

[13] See discussion of the relation between the extensional and intensional quotient types in Section 7.1.

[14] Where $\mathbb{B}$ is the type of booleans and `tt` ("true") and `ff` ("false") are its two members.

natural for that index to be of the type ($\mathbb{B}/\!/\mathrm{True}$) and as it turns out, this type represents a key idea that is worth formalizing on its own. As "usual", since we want to express the properties of the quotient types using the $\mathsf{ND} == (\mathbb{B}/\!/\mathrm{True})$ type, so it can not be defined using the quotient operator and needs to be introduced as a primitive.

The basic idea behind this ND type is that it contains two elements, say $\mathtt{tt}$ and $\mathtt{ff}$ and $\mathtt{tt} = \mathtt{ff} \in \mathsf{ND}$. In addition to these two constants we also need the $\mathtt{if}\ \ldots\ \mathtt{then}\ \ldots\ \mathtt{else}\ \ldots\ \mathtt{fi}$ operator such that $\mathtt{if\ tt\ then}\ t_1\ \mathtt{else}\ t_2\ \mathtt{fi}$ is computationally equivalent to t_1 and $\mathtt{if\ ff\ then}\ t_1\ \mathtt{else}\ t_2\ \mathtt{fi}$ is computationally equivalent to t_2. A natural approach would be to "borrow" these constants and this operator from $\mathbb{B}$ (the type of booleans) [15], but we can create new ones, it does not matter. We will write "$\mathtt{nd}_x\{t_1; t_2\}$" as an abbreviation for "$\mathtt{if}\ x\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2\ \mathtt{fi}$".

6.2 Explicit Nondeterminicity: Axioms

$$\frac{\Gamma;\, u : A[\mathtt{tt}] = A[\mathtt{ff}];\, y : A[\mathtt{tt}];\, x : \mathsf{ND};\, \Delta[x;y] \vdash C[x;y]}{\Gamma;\, x : \mathsf{ND};\, y : A[x];\, \Delta[x;y] \vdash C[x;y]} \qquad (\textit{ND-elim})$$

$$\frac{\Gamma \vdash C[\mathtt{tt}] = C[\mathtt{ff}] \qquad \Gamma \vdash C[\mathtt{tt}]}{\Gamma;\, x : \mathsf{ND} \vdash C[x]} \qquad (\textit{ND-elim2})$$

Notice that (*ND-elim*) does not completely eliminate the ND hypothesis, but only "moves" it one hypothesis to the right, so to completely eliminate the ND hypothesis, we will need to apply (*ND-elim*) repeatedly and then apply (*ND-elim2*) in the end.

For the purpose of formalizing the quotient operators we only need the two rules above. A complete formalization of ND would also include the axiom

$$\frac{}{\vdash \mathtt{tt} = \mathtt{ff} \in \mathsf{ND}} \qquad (\textit{ND-intro})$$

6.3 Explicit Nondeterminicity: Derived Rule

$$\frac{\Gamma \vdash t[\mathtt{tt}] = t[\mathtt{ff}] \in A}{\Gamma;\, x : \mathsf{ND} \vdash t[x] \in A} \qquad (\textit{ND-memb})$$

7 Intensional Quotient Type

7.1 Quotient Type: Introduction

The quotient types were originally introduced in [9]. They are also presented in [23].

While extensional quotient type can be useful sometimes, usually the intensional quotient type is sufficient and the extensionality just unnecessary complicates proofs by requiring us to prove extra well–typeness statements. In addition

[15] This would mean that ND is just $\mathbb{B}/\!/\mathrm{True}$.

to that NuPRL formalization of quotient types (see Appendix A) does not allow one to take full advantage of extensionality since most of the rules for the quotient type have an assumption that the equality predicate is in fact intensional. While early versions of NuPRL type theory considered extensional set and quotient types, these problems forced the change of set constructor (which is used substantially more often than the quotient) into an intensional one.

In order to avoid the problems outlined above, in this paper we introduce the intensional quotient type as primitive, and we concentrate our discussion of quotient types on intensional quotient types. But since we have the `esquash` operator in our theory, an extensional quotient type can be naturally defined if needed, using $A \underset{e}{/\!/} E := A \underset{i}{/\!/} [\![E]\!]$ and an extensional set type can be defined the same way: $\{x : A \mid_e P[x]\} := \{x : A \mid_i [\![P[x]]\!]\}$.

7.2 Intensional Quotient Type: Axioms

Two intensional quotient types are equal when both the quotiented types are equal, and the equality relations are equal:

$$\frac{\Gamma \vdash A = A' \qquad \Gamma; x : A; y : A \vdash E[x;y] = E'[x;y] \qquad \text{“}E \text{ is an ER over } A\text{”}}{\Gamma \vdash (A/\!/E) = (A'/\!/E')} \quad (\mathit{IquotEqualIntro})$$

where “E is an ER over A” is just an abbreviation for conditions that force E to be an equivalence relation over A.

Next, when two elements are equal in a quotient type, the equality predicate must be true on those elements. However, we know neither the witnesses of this predicate nor its intensional structure, therefore the equality in a quotient type only implies the `esquash` of the equality predicate:

$$\frac{\Gamma; u : x = y \in (A/\!/E); v : [\![E[x;y]]\!]; \Delta[u] \vdash C[u]}{\Gamma; u : x = y \in (A/\!/E); \Delta[u] \vdash C[u]} \quad (\mathit{IquotEqualElim})$$

The opposite is also true — we only need to prove the `esquash` of the equality predicate to be able to conclude that corresponding elements are equal in the quotient type:

$$\frac{\Gamma \vdash [\![E[x;y]]\!] \qquad \Gamma \vdash x \in (A/\!/E) \qquad \Gamma \vdash y \in (A/\!/E)}{\Gamma \vdash x = y \in (A/\!/E)} \quad (\mathit{IquotMemEqual})$$

Note that this rule has equality[16] in the quotient type in both the conclusion and the assumptions, so we still need a “base case” — an element of a type base type will also be an element of any well–typed quotient of that type:

$$\frac{\Gamma \vdash x \in A \qquad \Gamma \vdash (A/\!/E)\,\mathrm{Type}}{\Gamma \vdash x \in (A/\!/E)} \quad (\mathit{IquotMemIntro})$$

[16] As we explained in Remark 1.1, in Martin-Löf type theory membership is just a particular case of the equality.

Finally, we need to provide an elimination rule for quotient types. It turns out that being functional over some equivalence class of a quotient type is the same as being functional over an ND of any two elements of such class, so we can formulate the elimination rule as follows:

$$\frac{\Gamma;\, u_1 : A;\, u_2 : A;\, v : E[u_1;u_2];\, x : \mathsf{ND};\, \Delta[\mathsf{nd}_x\{u_1;u_2\}] \vdash \big[C[\mathsf{nd}_x\{u_1;u_2\}]\big]}{\Gamma;\, u : A/\!/E;\, \Delta[u] \vdash \big[C[u]\big]} \quad (\mathit{IquotElim})$$

7.3 Intensional Quotient Type: Derived Rules

From (*IquotElim*) and (*SquashEqual*) we can derive

$$\frac{\Gamma;\, u_1 : A;\, u_2 : A;\, v : E[u_1;u_2];\, x : \mathsf{ND};\, \Delta[\mathsf{nd}_x\{u_1;u_2\}] \vdash t[u_1] = t[u_2] \in C}{\Gamma;\, u : A/\!/E;\, \Delta[u] \vdash t[u] \in C} \quad (\mathit{IquotElim2})$$

From (*IquotEqualElim*) and (*EsquashElim2*), we can derive

$$\frac{\begin{array}{l}\Gamma;\, u : x = y \in (A/\!/E);\, v : \big[E[x;y]\big];\, \Delta[u] \vdash C[u] \\ \Gamma;\, u : x = y \in (A/\!/E);\, \Delta[u] \vdash E[x;y]\,\mathrm{Type}\end{array}}{\Gamma;\, u : x = y \in (A/\!/E);\, \Delta[u] \vdash C[u]} \quad (\mathit{IquotEqualElim2})$$

which is equivalent to NuPRL's (*quotient_equalityElimination*). However, (*IquotEqualElim*) is more general than (*quotient_equalityElimination*).

Example 7.1. We now can prove things like $x : \mathsf{ND};\, y : \mathsf{ND} \vdash \mathsf{nd}_x\{2;4\} = \mathsf{nd}_y\{4;6\} \in \mathbb{Z}_2$ where $\mathbb{Z}_2$ is $\mathbb{Z}$ quotiented over a "`mod` 2" equivalence relation.

7.4 Intensional Quotient Type: Comparison with NuPRL-4 Rules

The (*IquotMemEqual*) rule is similar to NuPRL-4's (*quotient_memberEquality*) rule (see Appendix A). However, in the (*IquotMemEqual*) rule, the first assumption has the equality predicate `esquash`ed. Also, the next two assumptions only require x and y to be in $A/\!/E$, not necessarily in A, as NuPRL-4 does (which also allowed to get rid of well–typeness assumption and only leave it in (*IquotMemIntro*)). These two changes make the rule much stronger than it was in NuPRL-4.

The (*IquotElim*) rule may look unusual, but it is just a generalization (and simplification) of NuPRL-4's (*quotientElimination*) and (*quotientElimination_2*) rules. It does not need the well–typeness assumption for E since we are using the intensional quotient type. It is important to mention that while NuPRL-4 quotient elimination rules are not irreversible, the (*IquotElim*) rule is reversible. In fact, all the rules of Section 7.2 are reversible, except for the (*IquotMemIntro*).

The (*IquotEqualElim*) rule is similar to NuPRL-4's (*quotient_equalityElimination*) rule, but it does not have an extra well–typeness assumption for E that is only necessary for extensional quotient type. The (*IquotEqualIntro*) and (*IquotMemIntro*) rules are the same as NuPRL-4's rules (*quotientWeakEquality*) and (*quotient_memberFormation*) and are only presented here for completeness.

8 Indexed Collections

8.1 Indexed and Predicated Collections

Consider an arbitrary type T in universe $\mathbb{U}$. We want to define the type of collections of elements of T. Such a type turned out to be very useful for various verification tasks. In particular, T may be a type of states (or type of transitions) of some complex automaton (or an I/O automaton) with potentially infinite number of states and we may want to be able to reason, for example, about collections of states (of transitions) reachable from certain configurations. We also want to formalize collections in the most general way possible, without assuming anything about T. In particular, we do not want to assume that T is enumerable or that T has decidable equality. And in fact, the constant problems we were facing when trying to formalize collections properly were the main reason for the research that lead to this paper.

There are at least two different approaches we can take to start formalizing such collections.

1. We can start formalizing collections collection as pairs consisting of an index set $I : \mathbb{U}$ and an index function $f : (I \to T)$. In other words, we can start with the type $I : \mathbb{U} \times (I \to T)$.
2. We can start formalizing collections by concentrating on membership predicates of collections. In other words, we can start with the type $T \to \mathbf{Prop}$.

It is easy to see that these two approaches are equivalent. Indeed, if we have a pair $< I, f >$, we can get a predicate $\lambda t.\, \exists i \in I.\, f(i) = t \in T$ and if we have a predicate P, we can take a pair $< \{t : T \mid P(t)\}, \lambda t.\, t >$. Because of this isomorphism, everywhere below we will allow ourselves to use $T \to \mathbf{Prop}$ as a base for the collection type even though $I : \mathbb{U} \times (I \to T)$ is a little closer to our intuition about collections.

Clearly, the type $T \to \mathbf{Prop}$ is not quite what we want yet since two different predicates from that type can represent the same collection. An obvious way of addressing this problem is to use a quotient type. In other words, we want to define the type of collections as

$$\mathsf{Col}(T) \;:=\; c_1, c_2 : (T \to \mathbb{U})//(\forall t \in T.\, c_1(t) \Leftrightarrow c_2(t)). \tag{1}$$

8.2 Collections: the Problem

Once we have defined the type of collections, the next natural step is to start defining various basic operations on that type. In particular, we want to have the following operations:

- A predicate telling us whether some element is a member of some collection: $\forall c \in \mathsf{Col}(T).\, \forall t \in T.\, \mathtt{mem}(c; t) \in \mathbf{Prop}$
- An operator that would produce a union of a family of collections: $\forall I \in \mathbb{U}.\, \forall C \in (I \to \mathsf{Col}(T)).\, \bigcup\limits_{i \in I} C(i) \in \mathsf{Col}(T)$

And we want our operators to have the following natural properties:

- $\forall c \in T \to \mathbb{U}.\, \forall t \in T.\, c(t) \Rightarrow \mathtt{mem}(c;t)$
- $\forall c \in T \to \mathbb{U}.\, \forall t \in T.\, \neg c(t) \Rightarrow \neg\mathtt{mem}(c;t)$
- $\forall c_1, c_2 \in \mathsf{Col}(T).\, \Big(\Big(\forall t : T.\, \big(\mathtt{mem}(c_1;t) \Leftrightarrow \mathtt{mem}(c_2;t)\big)\Big) \Leftrightarrow c_1 = c_2 \in \mathsf{Col}(T)\Big)$
  (note that the $\Leftarrow$ direction follows from the typing requirement for `mem`).
- $\forall I \in \mathbb{U}.\, \forall C : I \to \mathsf{Col}(T).\, \forall t \in T.\, \big(\exists i : I.\, \mathtt{mem}(C(i);t) \Rightarrow \mathtt{mem}(\bigcup\limits_{i \in I} C(i);t)\big)$
  Note that we do not require an implication in the opposite direction since that would mean that we will have to be able to reconstruct i constructively just from some indirect knowledge that it exists. Instead we only require
- $\forall I \in \mathbb{U}, C \in I \to \mathsf{Col}(T), t \in T.\, \Big(\neg\Big(\exists i : I.\, \mathtt{mem}\big(C(i);t\big)\Big) \Rightarrow \neg\Big(\mathtt{mem}\big(\bigcup\limits_{i \in I} C(i);t\big)\Big)\Big)$

It turned out that formulating these operations with these properties is nearly impossible [17] in NuPRL-4 type theory with its monolithic approach to quotient types. The problems we were constantly experiencing when trying to come up with a solution included `mem` erroneously returning an element of $\mathbf{Prop}/\!/\Leftrightarrow$ instead of $\mathbf{Prop}$, `union` being able to accept only arguments of type

$$C_1, C_2 : \big(I \to (T \to \mathbf{Prop})\big) /\!/ \big(\forall i \in I.\, \forall t \in T\, C_1(i;t) \Leftrightarrow C_2(i;t)\big)$$

(which is a subtype of the $I \to \mathsf{Col}(T)$ type that we are interested in), *etc.*

8.3 Collections: a Possible Solution

Now that we have $[\ ]$ and $[\![\ ]\!]$ operators, it is relatively easy to give the proper definitions. If we take $\mathtt{mem}(c;t) := [\![c(t)]\!]$ and $\bigcup\limits_{i \in I} C(i) := \lambda t. \exists i \in I. \mathtt{mem}(C(i);t)$, we can prove all the properties listed in Section 8.2.

9 Related Work

In [10] Pierre Courtieu attempts to add to Coq's Calculus of Constructions a notion very similar to quotient type. Instead of aiming at "general" quotient type, [10] considers types that have a "normalization" function that, essentially, maps all the members of each equivalence class to a canonical member of the class. Courtieu shows how by equipping a quotient type with such normalization function, one can substantially simplify handling of such a quotient type. In a sense, `esquash` works the same way — it acts as a normalization function for the $\mathbf{Prop}/\!/\Leftrightarrow$. The main difference here is that instead of considering a normalization function that returns *an existing* element of each equivalence class, with `esquash` we utilize the open–ended nature of the type theory to equip each equivalence class with *a new* normal element.

[17] Several members of NuPRL community made many attempts to come up with a satisfactory formalization. The formalization presented in this paper was the only one that worked.

10 Acknowledgments

This work would not be possible without Mark Bickford, who came up with the problem of formalizing collections. This work was also a result of very productive discussions with Robert Constable, Alexei Kopylov, Mark Bickford and Jason Hickey. The author is also very grateful to Vladimir Krupski who was the first to notice the modularity consequences of this work. The text of this paper would not be nearly as good as it currently is without the very useful critique of Sergei Artemov, Robert Constable, Stuart Allen, Alexei Kopylov and others.

References

1. Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.
2. Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In *Proceedings of the Second Symposium on Logic in Computer Science*, pages 215–224. IEEE, June 1987.
3. Roland Backhouse. A note of subtypes in Martin-Löf's theory of types. Technical Report CSM-90, University of Essex, November 1984.
4. Roland Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. In A. Avron, editor, *Workshop on General Logic, Edinburgh, February 1987*, number ECS-LFCS-88-52. Department of Computer Science, University of Edinburgh, May 1988.
5. Roland C. Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.
6. Ken Birman, Robert Constable, Mark Hayden, Jason J. Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–161. IEEE, 2000.
7. Robert L. Constable. Mathematics as programming. In *Proceedings of the Workshop on Programming and Logics, Lectures Notes in Computer Science 164*, pages 116–128. Springer-Verlag, 1983.
8. Robert L. Constable. Types in logic, mathematics and programming. In S. R. Buss, editor, *Handbook of Proof Theory*, chapter X, pages 683–786. Elsevier Science B.V., 1998.
9. Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Development System*. Prentice-Hall, NJ, 1986.
10. Pierre Courtieu. Normalized types. In L. Fribourg, editor, *Computer Science Logic, Proceedings of the 10th Annual Conference of the EACSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 554–569. Springer-Verlag, 2001. `http://link.springer-ny.com/link/service/series/0558/tocs/t2142.htm`.
11. Jason J. Hickey. NuPRL-Light: An implementation framework for higer-order logics. In William McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of Lecture Notes on Artificial Intelligence, pages 395–399, Berlin, July 13–17 1997. Springer. CADE '97. An extended version of the paper can be found at `http://www.cs.caltech.edu/~jyh/papers/cade14_nl/default.html`.

12. Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.
13. Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.
14. Douglas J. Howe. Semantic foundations for embedding HOL in NuPRL. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of Lecture Notes in Computer Science, pages 85–101. Springer-Verlag, Berlin, 1996.
15. Alexei Kopylov and Aleksey Nogin. Markov's principle for propositional type theory. In L. Fribourg, editor, *Computer Science Logic, Proceedings of the 10th Annual Conference of the EACSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 570–584. Springer-Verlag, 2001. `http://link.springer-ny.com/link/service/series/0558/tocs/t2142.htm`.
16. Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason J. Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *17^{th} ACM Symposium on Operating Systems Principles*, December 1999.
17. Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.
18. J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. Amsterdam:North-Holland, 1963.
19. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. Submitted to TPHOLs 2002, 2002.
20. Bengt Nordström and Kent Petersson. Types and specifications. In *IFIP'93*. Elsvier, 1983.
21. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Sciences Publication, Oxford, 1990.
22. Frank Pfenning and Rowan Davies. Judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4), August 2001.
23. Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.
24. Anne Sjerp Troelstra. *Metamathematical Investigation of Intuitionistic Mathematics*, volume 344 of *Lecture Notes in Mathematics*. Springer-Verlag, 1973.

A Appendix. NuPRL-4 Quotient Rules

$$\frac{\begin{array}{l} H \vdash A_1 = A_2 \in \mathbb{U} \\ H;\, x : A_1;\, y : A_1 \vdash E_1[u_1; v_1] = E_2[u_2; v_2] \in \mathbb{U} \\ H \vdash \text{“}E \text{ is an ER on } A\text{”} \end{array}}{H \vdash (u_1, v_1 : A_1//E_1) = (u_2, v_2 : A_2//E_2) \in \mathbb{U}} \quad (\mathit{quotientWeakEquality})$$

$$\frac{\begin{array}{l} H \vdash (x, y : A//E) = (x, y : A//E) \in \mathbb{U} \\ H \vdash a \in A \end{array}}{H \vdash a \in x, y : A//E} \quad (\mathit{quotient\_memberFormation})$$

$$\frac{\begin{array}{l} H \vdash (x, y : A//E) = (x, y : A//E) \in \mathbb{U} \\ H \vdash a_1 = a_2 \in A \end{array}}{H \vdash a_1 = a_2 \in (x, y : A//E)} \quad (\mathit{quotient\_memberWeakEquality})$$

$$\frac{\begin{array}{l} H;\, u:(x,y:A//E);\, J;\, v:A;\, w:A \vdash E[x;y]=E[x;y]\in\mathbb{U} \\ H;\, u:(x,y:A//E);\, J \vdash T=T\in\mathbb{U} \\ H;\, u:(x,y:A//E);\, J;\, v:A;\, w:A;\, z:E[x;y] \vdash s[u]=t[u]\in T[u] \end{array}}{H;\, u:(x,y:A//E);\, J \vdash s=t\in T} \quad (quotientElimination)$$

$$\frac{\begin{array}{l} H;\, u:(x,y:A//E);\, J;\, v:A;\, w:A \vdash E[x;y]=E[x;y]\in\mathbb{U} \\ H;\, u:(x,y:A//E);\, J \vdash T=T\in\mathbb{U} \\ H;\, u:(x,y:A//E);\, v:A;\, w:A;\, z:E[x;y];\, J[u] \vdash s[u]=t[u]\in T[u] \end{array}}{H;\, u:(x,y:A//E);\, J \vdash s=t\in T} \quad (quotientElimination\_2)$$

$$\frac{\begin{array}{l} H;\, u:(a=b\in(x,y:S//E));\, [v:E[x;y]];\, J \vdash t\in T \\ H;\, u:(a=b\in(x,y:S//E));\, J \vdash E[x;y]=E[x;y]\in\mathbb{U}\{j\} \end{array}}{H;\, u:(a=b\in(x,y:S//E));\, J \vdash t\in T} \quad (quotient\_equalityElimination)$$

$$\frac{\begin{array}{ll} H \vdash x,y:A//E\in\mathbb{U} & H \vdash u,v:B//F\in\mathbb{U} \\ H \vdash A=B\in\mathbb{U} & H;\, w:(A=B\in\mathbb{U});\, r:A;\, s:A \vdash E[x;y] \Leftrightarrow F[u;v] \end{array}}{H \vdash (x,y:A//E)=(u,v:B//F)\in\mathbb{U}} \quad (quotientEquality)$$

$$\frac{\begin{array}{l} H \vdash (x,y:A//E)=(x,y:A//E)\in\mathbb{U} \\ H \vdash s=s\in A \quad H \vdash t=t\in A \quad H \vdash E[x;y] \end{array}}{H \vdash s=t\in(x,y:A//E)} \quad (quotient\_memberEquality)$$