Quotients Revisited for Isabelle/HOL

Cezary Kaliszyk
University of Tsukuba, Japan
kaliszyk@cs.tsukuba.ac.jp

Christian Urban
Technical University of Munich, Germany
urbanc@in.tum.de

ABSTRACT

Higher-Order Logic (HOL) is based on a small logic kernel, whose only mechanism for extension is the introduction of safe definitions and of non-empty types. Both extensions are often performed in quotient constructions. To ease the work involved with such quotient constructions, we re-implemented in the Isabelle/HOL theorem prover the quotient package by Homeier. In doing so we extended his work in order to deal with compositions of quotients and also specified completely the procedure of lifting theorems from the raw level to the quotient level. The importance for theorem proving is that many formal verifications, in order to be feasible, require a convenient reasoning infrastructure for quotient constructions.

Keywords

Quotients, Isabelle theorem prover, Higher-Order Logic

1. INTRODUCTION

One might think quotients have been studied to death, but in the context of theorem provers many questions concerning them are far from settled. In this paper we address the question of how to establish a convenient reasoning infrastructure for quotient constructions in the Isabelle/HOL, theorem prover. Higher-Order Logic (HOL) consists of a small number of axioms and inference rules over a simply-typed term-language. Safe reasoning in HOL is ensured by two very restricted mechanisms for extending the logic: one is the definition of new constants in terms of existing ones; the other is the introduction of new types by identifying non-empty subsets in existing types. Previous work has shown how to use both mechanisms for dealing with quotient constructions in HOL (see [6, 9]). For example the integers in Isabelle/HOL are constructed by a quotient construction over the type *nat* $\times$ *nat* and the equivalence relation

$$(n_1, n_2) \approx (m_1, m_2) \triangleq n_1 + m_2 = m_1 + n_2 \tag{1}$$

This constructions yields the new type *int*, and definitions for *0* and *1* of type *int* can be given in terms of pairs of natural numbers (namely *(0, 0)* and *(1, 0)*). Operations such as *add* with type *int* $\Rightarrow$ *int* $\Rightarrow$ *int* can be defined in terms of operations on pairs of natural numbers (namely *add_pair* (n_1, m_1) $(n_2, m_2) \triangleq (n_1 + n_2, m_1 + m_2)$). Similarly one can construct finite sets, written α *fset*, by quotienting the type α *list* according to the equivalence relation

$$xs \approx ys \triangleq (\forall x.\ memb\ x\ xs \longleftrightarrow memb\ x\ ys) \tag{2}$$

which states that two lists are equivalent if every element in one list is also member in the other. The empty finite set, written $\varnothing$, can then be defined as the empty list and the union of two finite sets, written $\cup$, as list append.

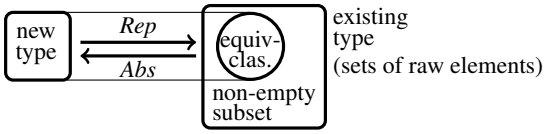
Quotients are important in a variety of areas, but they are really ubiquitous in the area of reasoning about programming language calculi. A simple example is the lambda-calculus, whose raw terms are defined as

$$t ::= x \mid t\ t \mid \lambda x.t$$

The problem with this definition arises, for instance, when one attempts to prove formally the substitution lemma [1] by induction over the structure of terms. This can be fiendishly complicated (see [4, Pages 94–104] for some "rough" sketches of a proof about raw lambda-terms). In contrast, if we reason about α-equated lambda-terms, that means terms quotient according to α-equivalence, then the reasoning infrastructure provided, for example, by Nominal Isabelle makes the formal proof of the substitution lemma almost trivial.

The difficulty is that in order to be able to reason about integers, finite sets or α-equated lambda-terms one needs to establish a reasoning infrastructure by transferring, or *lifting*, definitions and theorems from the raw type *nat* $\times$ *nat* to the quotient type *int* (similarly for finite sets and α-equated lambda-terms). This lifting usually requires a *lot* of tedious reasoning effort [9]. In principle it is feasible to do this work manually, if one has only a few quotient constructions at hand. But if they have to be done over and over again, as in Nominal Isabelle, then manual reasoning is not an option.

The purpose of a *quotient package* is to ease the lifting of theorems and automate the reasoning as much as possible. In the context of HOL, there have been a few quotient packages already [5, 10]. The most notable one is by Homeier [6] implemented in HOL4. The fundamental construction these quotient packages perform can be illustrated by the following picture:



The starting point is an existing type, to which we refer as the *raw type* and over which an equivalence relation is given by the user.

SAC'11 March 21-25, 2011, TaiChung, Taiwan.

With this input the package introduces a new type, to which we refer as the *quotient type*. This type comes with an *abstraction* and a *representation* function, written *Abs* and *Rep*.[1] They relate elements in the existing type to elements in the new type, and can be uniquely identified by their quotient type. For example for the integer quotient construction the types of *Abs* and *Rep* are

$$Abs :: nat \times nat \Rightarrow int \qquad Rep :: int \Rightarrow nat \times nat$$

We therefore often write *Abs_int* and *Rep_int* if the typing information is important.

Every abstraction and representation function stands for an isomorphism between the non-empty subset and elements in the new type. They are necessary for making definitions involving the new type. For example *0* and *1* of type *int* can be defined as

$$0 \triangleq Abs\_int\ (0, 0) \qquad 1 \triangleq Abs\_int\ (1, 0)$$

Slightly more complicated is the definition of *add* having type *int* $\Rightarrow$ *int* $\Rightarrow$ *int*. Its definition is as follows

$$add\ n\ m \triangleq Abs\_int\ (add\_pair\ (Rep\_int\ n)\ (Rep\_int\ m)) \tag{3}$$

where we take the representation of the arguments n and m, add them according to the function *add_pair* and then take the abstraction of the result. This is all straightforward and the existing quotient packages can deal with such definitions. But what is surprising is that none of them can deal with slightly more complicated definitions involving *compositions* of quotients. Such compositions are needed for example in case of quotienting lists to yield finite sets and the operator that flattens lists of lists, defined as follows

$$flat\ [] \triangleq [] \qquad flat\ x::xs \triangleq x\ @\ flat\ xs$$

where @ is the usual list append. We expect that the corresponding operator on finite sets, written *fconcat*, builds finite unions of finite sets:

$$\bigcup \varnothing \triangleq \varnothing \qquad \bigcup \{x\} \cup S \triangleq x \cup \bigcup S$$

The quotient package should automatically provide us with a definition for $\bigcup$ in terms of *flat*, *Rep_fset* and *Abs_fset*. The problem is that the method used in the existing quotient packages of just taking the representation of the arguments and then taking the abstraction of the result is *not* enough. The reason is that in case of $\bigcup$ we obtain the incorrect definition

$$\bigcup S \triangleq Abs\_fset\ (flat\ (Rep\_fset\ S))$$

where the right-hand side is not even typable! This problem can be remedied in the existing quotient packages by introducing an intermediate step and reasoning about flattening of lists of finite sets. However, this remedy is rather cumbersome and inelegant in light of our work, which can deal with such definitions directly. The solution is that we need to build aggregate representation and abstraction functions, which in case of $\bigcup$ generate the definition

$$\bigcup S \triangleq Abs\_fset\ (flat\ ((map\_list\ Rep\_fset \circ Rep\_fset)\ S))$$

where *map_list* is the usual mapping function for lists. In this paper we will present a formal definition of our aggregate abstraction and representation functions (this definition was omitted in [6]). They generate definitions, like the one above for $\bigcup$, according to the type of the raw constant and the type of the quotient constant. This means we also have to extend the notions of *aggregate equivalence relation*, *respectfulness* and *preservation* from Homeier [6].

In addition we are able to clearly specify what is involved in the lifting process (this was only hinted at in [6] and implemented as a "rough recipe" in ML-code). A pleasing side-result is that our procedure for lifting theorems is completely deterministic following the structure of the theorem being lifted and the theorem on the quotient level. Space constraints, unfortunately, allow us to only sketch this part of our work in Section 5 and we defer the reader to a longer version for the details. However, we will give in Section 3 and 4 all definitions that specify the input and output data of our three-step lifting procedure. Appendix A gives an example how our quotient package works in practise.

2. PRELIMINARIES AND GENERAL QUOTIENTS

We will give in this section a crude overview of HOL and describe the main definitions given by Homeier for quotients [6].

At its core, HOL is based on a simply-typed term language, where types are recorded in Church-style fashion (that means, we can always infer the type of a term and its subterms without any additional information). The grammars for types and terms are

$$\sigma, \tau ::= \alpha \mid (\sigma, \ldots, \sigma)\ \kappa \qquad t, s ::= x^{\sigma} \mid c^{\sigma} \mid t\ t \mid \lambda x^{\sigma}.\ t$$

with types being either type variables or type constructors and terms being variables, constants, applications or abstractions. We often write just κ for $()\ \kappa$, and use αs and σs to stand for collections of type variables and types, respectively. The type of a term is often made explicit by writing $t :: \sigma$. HOL includes a type *bool* for booleans and the function type, written $\sigma \Rightarrow \tau$. HOL also contains many primitive and defined constants; for example, a primitive constant is equality, with type $= :: \sigma \Rightarrow \sigma \Rightarrow bool$, and the identity function with type $id :: \sigma \Rightarrow \sigma$ is defined as $\lambda x^{\sigma}.\ x^{\sigma}$.

An important point to note is that theorems in HOL can be seen as a subset of terms that are constructed specially (namely through axioms and proof rules). As a result we are able to define automatic proof procedures showing that one theorem implies another by decomposing the term underlying the first theorem.

Like Homeier's, our work relies on map-functions defined for every type constructor taking some arguments, for example *map_list* for lists. Homeier describes in [6] map-functions for products, sums, options and also the following map for function types

$$(f \mapsto g)\ h \triangleq g \circ h \circ f$$

Using this map-function, we can give the following, equivalent, but more uniform definition for *add* shown in (3):

$$add \triangleq (Rep\_int \mapsto Rep\_int \mapsto Abs\_int)\ add\_pair$$

Using extensionality and unfolding the definition of $\mapsto$, we can get back to (3). In what follows we shall use the convention to write map_κ for a map-function of the type-constructor κ. In our implementation we maintain a database of these map-functions that can be dynamically extended.

It will also be necessary to have operators, referred to as rel_κ, which define equivalence relations in terms of constituent equivalence relations. For example given two equivalence relations R_1 and R_2, we can define an equivalence relations over products as

$$(R_1\ \#\#\#\ R_2)\ (x_1, x_2)\ (y_1, y_2) \triangleq R_1\ x_1\ y_1 \wedge R_2\ x_2\ y_2$$

[1] Actually slightly more basic functions are given; the functions *Abs* and *Rep* need to be derived from them. We will show the details later.

Homeier gives also the following operator for defining equivalence relations over function types

$$R_1 \Mapsto R_2 \triangleq \lambda f\ g.\ \forall x\ y.\ R_1\ x\ y \longrightarrow R_2\ (f\ x)\ (g\ y) \tag{4}$$

In the context of quotients, the following two notions from [6] are needed later on.

DEFINITION 1 (RESPECTS). *An element* x *respects a relation* R *provided* R x x.

DEFINITION 2 (BOUNDED $\forall$ AND λ). $\forall$ x $\in$ S. P x *holds if for all* x, x $\in$ S *implies* P x*; and* (λx $\in$ S. f x) = f x *provided* x $\in$ S.

The central definition in Homeier's work [6] relates equivalence relations, abstraction and representation functions:

DEFINITION 3 (QUOTIENT TYPES). *Given a relation R, an abstraction function Abs and a representation function Rep, the predicate* Quot R Abs Rep *holds if and only if*

(i) $\forall a.\ Abs\ (Rep\ a) = a$
(ii) $\forall a.\ R\ (Rep\ a)\ (Rep\ a)$
(iii) $\forall r\ s.\ R\ r\ r = (R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s)$

The value of this definition lies in the fact that validity of *Quot R Abs Rep* can often be proved in terms of the validity of *Quot* over the constituent types of *R*, *Abs* and *Rep*. For example Homeier proves the following property for higher-order quotient types:

PROPOSITION 1. *If Quot* R_1 Abs_1 Rep_1 *and Quot* R_2 Abs_2 Rep_2 *then Quot* $R_1 \Mapsto R_2$ $Rep_1 \mapsto Abs_2$ $Abs_1 \mapsto Rep_2$.

As a result, Homeier is able to build an automatic prover that can nearly always discharge a proof obligation involving *Quot*. Our quotient package makes heavy use of this part of Homeier's work including an extension for dealing with *conjugations* of equivalence relations[2] defined as follows:

DEFINITION 4. $R_1 \circ\circ\circ R_2 \triangleq R_1 \circ\circ R_2 \circ\circ R_1$ *where* $\circ\circ$ *is the predicate composition defined by* ($R_1 \circ\circ R_2$) x z *holds if and only if there exists a* y *such that* R_1 x y *and* R_2 y z.

Unfortunately a general quotient theorem for $\circ\circ\circ$, analogous to the one for $\mapsto$ given in Proposition 1, would not be true in general. It cannot even be stated inside HOL, because of restrictions on types. However, we can prove specific instances of a quotient theorem for composing particular quotient relations. For example, to lift theorems involving *flat* the quotient theorem for composing $\approx_{list}$ will be necessary: given *Quot R Abs Rep* with *R* being an equivalence relation, then

$$\begin{aligned} &Quot\ (rel\_list\ R \circ\circ\circ \approx_{list}) \\ &\quad (Abs\_fset \circ map\_list\ Abs)\ (map\_list\ Rep \circ Rep\_fset) \end{aligned}$$

3. QUOTIENT TYPES AND QUOTIENT DEFINITIONS

The first step in a quotient construction is to take a name for the new type, say κ_q, and an equivalence relation, say *R*, defined over a raw type, say σ. The type of the equivalence relation must be $\sigma \Rightarrow \sigma \Rightarrow bool$. The user-visible part of the quotient type declaration is therefore

[2]That are symmetric by definition.

quotient_type $\alpha s\ \kappa_q = \sigma\ /\ R$ (5)

and a proof that *R* is indeed an equivalence relation. The αs indicate the arity of the new type and the type-variables of σ can only be contained in αs. Two concrete examples are

quotient_type $int = nat \times nat\ /\ \approx_{nat \times nat}$
quotient_type $\alpha\ fset = \alpha\ list\ /\ \approx_{list}$

which introduce the type of integers and of finite sets using the equivalence relations $\approx_{nat \times nat}$ and $\approx_{list}$ defined in (1) and (2), respectively (the proofs about being equivalence relations is omitted). Given this data, we define for declarations shown in (5) the quotient types internally as

typedef $\alpha s\ \kappa_q = \{c.\ \exists x.\ c = R\ x\}$

where the right-hand side is the (non-empty) set of equivalence classes of *R*. The constraint in this declaration is that the type variables in the raw type σ must be included in the type variables αs declared for κ_q. HOL will then provide us with the following abstraction and representation functions

$$abs\_\kappa_q :: \sigma\ set \Rightarrow \alpha s\ \kappa_q \qquad rep\_\kappa_q :: \alpha s\ \kappa_q \Rightarrow \sigma\ set$$

As can be seen from the type, they relate the new quotient type and equivalence classes of the raw type. However, as Homeier [6] noted, it is much more convenient to work with the following derived abstraction and representation functions

$$Abs\_\kappa_q\ x \triangleq abs\_\kappa_q\ (R\ x) \qquad Rep\_\kappa_q\ x \triangleq \varepsilon\ (rep\_\kappa_q\ x)$$

on the expense of having to use Hilbert's choice operator ε in the definition of Rep_κ_q. These derived notions relate the quotient type and the raw type directly, as can be seen from their type, namely $\sigma \Rightarrow \alpha s\ \kappa_q$ and $\alpha s\ \kappa_q \Rightarrow \sigma$, respectively. Given that *R* is an equivalence relation, the following property holds for every quotient type (for the proof see [6]).

PROPOSITION 2. *Quot R* Abs_κ_q Rep_κ_q.

The next step in a quotient construction is to introduce definitions of new constants involving the quotient type. These definitions need to be given in terms of concepts of the raw type (remember this is the only way how to extend HOL with new definitions). For the user the visible part of such definitions is the declaration

quotient_definition $c :: \tau$ **is** $t :: \sigma$

where *t* is the definiens (its type σ can always be inferred) and *c* is the name of definiendum, whose type τ needs to be given explicitly (the point is that τ and σ can only differ in places where a quotient and raw type is involved). Two concrete examples are

quotient_definition $0 :: int$ **is** $(0::nat, 0::nat)$
quotient_definition $\bigcup :: (\alpha\ fset)\ fset \Rightarrow \alpha\ fset$ **is** *flat*

The first one declares zero for integers and the second the operator for building unions of finite sets (*flat* having the type $(\alpha\ list)\ list \Rightarrow \alpha\ list$).

From such declarations given by the user, the quotient package needs to derive proper definitions using *Abs* and *Rep*. The data we rely on is the given quotient type τ and the raw type σ. They allow us to define *aggregate abstraction* and *representation functions* using the functions *ABS* (σ, τ) and *REP* (σ, τ) whose clauses we

shall give below. The idea behind these two functions is to simultaneously descend into the raw types σ and quotient types τ, and generate the appropriate *Abs* and *Rep* in places where the types differ. Therefore we generate just the identity whenever the types are equal. On the "way" down, however we might have to use map-functions to let *Abs* and *Rep* act over the appropriate types. In what follows we use the short-hand notation *ABS* $(\sigma s, \tau s)$ to mean *ABS* $(\sigma_1, \tau_1) \ldots$ *ABS* (σ_n, τ_n); similarly for *REP*.

$$
\begin{array}{l}
\text{equal types:} \\
\quad \mathit{ABS}\ (\sigma, \sigma) \triangleq \mathit{id} :: \sigma \Rightarrow \sigma \qquad \mathit{REP}\ (\sigma, \sigma) \triangleq \mathit{id} :: \sigma \Rightarrow \sigma \\
\text{function types:} \\
\quad \mathit{ABS}\ (\sigma_1 \Rightarrow \sigma_2, \tau_1 \Rightarrow \tau_2) \triangleq \mathit{REP}\ (\sigma_1, \tau_1) \mapsto \mathit{ABS}\ (\sigma_2, \tau_2) \\
\quad \mathit{REP}\ (\sigma_1 \Rightarrow \sigma_2, \tau_1 \Rightarrow \tau_2) \triangleq \mathit{ABS}\ (\sigma_1, \tau_1) \mapsto \mathit{REP}\ (\sigma_2, \tau_2) \\
\text{equal type constructors:} \\
\quad \mathit{ABS}\ (\sigma s\ \kappa, \tau s\ \kappa) \triangleq \mathit{map\_}\kappa\ (\mathit{ABS}\ (\sigma s, \tau s)) \\
\quad \mathit{REP}\ (\sigma s\ \kappa, \tau s\ \kappa) \triangleq \mathit{map\_}\kappa\ (\mathit{REP}\ (\sigma s, \tau s)) \\
\text{unequal type constructors:} \\
\quad \mathit{ABS}\ (\sigma s\ \kappa, \tau s\ \kappa_q) \triangleq \mathit{Abs\_}\kappa_q \circ (\mathit{MAP}(\varrho s\ \kappa)\ (\mathit{ABS}\ (\sigma s', \tau s))) \\
\quad \mathit{REP}\ (\sigma s\ \kappa, \tau s\ \kappa_q) \triangleq (\mathit{MAP}(\varrho s\ \kappa)\ (\mathit{REP}\ (\sigma s', \tau s))) \circ \mathit{Rep\_}\kappa_q
\end{array} \tag{6}
$$

In the last two clauses are subtle. We rely in them on the fact that the type $\alpha s\ \kappa_q$ is the quotient of the raw type $\varrho s\ \kappa$ (for example *int* and *nat* $\times$ *nat*, or α *fset* and α *list*). This data is given by declarations shown in (5). The quotient construction ensures that the type variables in $\varrho s\ \kappa$ must be among the αs. The $\sigma s'$ are given by the substitutions for the αs when matching $\sigma s\ \kappa$ against $\varrho s\ \kappa$. This calculation determines what are the types in place of the type variables αs in the instance of quotient type $\alpha s\ \kappa_q$—namely τs, and the corresponding types in place of the αs in the raw type $\varrho s\ \kappa$—namely $\sigma s'$. The function *MAP* calculates an *aggregate map-function* for a raw type as follows:

$$
\begin{array}{rcl}
\mathit{MAP'}\ (\alpha) & \triangleq & a^{\alpha} \\
\mathit{MAP'}\ (\kappa) & \triangleq & \mathit{id} :: \kappa \Rightarrow \kappa \\
\mathit{MAP'}\ (\sigma s\ \kappa) & \triangleq & \mathit{map\_}\kappa\ (\mathit{MAP'}(\sigma s)) \\
\mathit{MAP}\ (\sigma) & \triangleq & \lambda as.\ \mathit{MAP'}(\sigma)
\end{array}
$$

In this definition we rely on the fact that in the first clause we can interpret type-variables α as term variables a. In the last clause we build an abstraction over all term-variables of the map-function generated by the auxiliary function *MAP'*. The need for aggregate map-functions can be seen in cases where we build quotients, say $(\alpha, \beta)\ \kappa_q$, out of compound raw types, say $(\alpha\ \mathit{list}) \times \beta$. In this case *MAP* generates the aggregate map-function:

$$\lambda a\ b.\ \mathit{map\_prod}\ (\mathit{map\_list}\ a)\ b$$

which is essential in order to define the corresponding aggregate abstraction and representation functions.

To see how these definitions pan out in practise, let us return to our example about *flat* and *fconcat*, where we have the raw type $(\alpha\ \mathit{list})\ \mathit{list} \Rightarrow \alpha\ \mathit{list}$ and the quotient type $(\alpha\ \mathit{fset})\ \mathit{fset} \Rightarrow \alpha\ \mathit{fset}$. Feeding these types into *ABS* gives us (after some β-simplifications) the abstraction function

$$(\mathit{map\_list}\ (\mathit{map\_list}\ \mathit{id} \circ \mathit{Rep\_fset}) \circ \mathit{Rep\_fset}) \mapsto \mathit{Abs\_fset} \circ \mathit{map\_list}\ \mathit{id}$$

In our implementation we further simplify this function by rewriting with the usual laws about *map*s and *id*, for example $\mathit{map\_list}\ \mathit{id} = \mathit{id}$ and $f \circ \mathit{id} = \mathit{id} \circ f = f$. This gives us the simpler abstraction function

$$(\mathit{map\_list}\ \mathit{Rep\_fset} \circ \mathit{Rep\_fset}) \mapsto \mathit{Abs\_fset}$$

which we can use for defining *fconcat* as follows

$$\bigcup \triangleq ((\mathit{map\_list}\ \mathit{Rep\_fset} \circ \mathit{Rep\_fset}) \mapsto \mathit{Abs\_fset})\ \mathit{flat}$$

Note that by using the operator $\mapsto$ and special clauses for function types in (6), we do not have to distinguish between arguments and results, but can deal with them uniformly. Consequently, all definitions in the quotient package are of the general form

$$c \triangleq \mathit{ABS}\ (\sigma, \tau)\ t$$

where σ is the type of the definiens t and τ the type of the defined quotient constant c. This data can be easily generated from the declaration given by the user. To increase the confidence in this way of making definitions, we can prove that the terms involved are all typable.

LEMMA 1. *If* ABS (σ, τ) *returns some abstraction function* Abs *and* REP (σ, τ) *some representation function* Rep*, then* Abs *is of type* $\sigma \Rightarrow \tau$ *and* Rep *of type* $\tau \Rightarrow \sigma$.

PROOF. By mutual induction and analysing the definitions of *ABS* and *REP*. The cases of equal types and function types are straightforward (the latter follows from $\mapsto$ having the type $(\alpha \Rightarrow \beta) \Rightarrow (\gamma \Rightarrow \delta) \Rightarrow (\beta \Rightarrow \gamma) \Rightarrow (\alpha \Rightarrow \delta)$). In case of equal type constructors we can observe that a map-function after applying the functions *ABS* $(\sigma s, \tau s)$ produces a term of type $\sigma s\ \kappa \Rightarrow \tau s\ \kappa$. The interesting case is the one with unequal type constructors. Since we know the quotient is between $\alpha s\ \kappa_q$ and $\varrho s\ \kappa$, we have that $\mathit{Abs\_}\kappa_q$ is of type $\varrho s\ \kappa \Rightarrow \alpha s\ \kappa_q$. This type can be more specialised to $\varrho s[\tau s]\ \kappa \Rightarrow \tau s\ \kappa_q$ where the type variables αs are instantiated with the τs. The complete type can be calculated by observing that *MAP* $(\varrho s\ \kappa)$, after applying the functions *ABS* $(\sigma s', \tau s)$ to it, returns a term of type $\varrho s[\sigma s']\ \kappa \Rightarrow \varrho s[\tau s]\ \kappa$. This type is equivalent to $\sigma s\ \kappa \Rightarrow \varrho s[\tau s]\ \kappa$, which we just have to compose with $\varrho s[\tau s]\ \kappa \Rightarrow \tau s\ \kappa_q$ according to the type of $\circ$. □

4. RESPECTFULNESS AND PRESERVATION

The main point of the quotient package is to automatically "lift" theorems involving constants over the raw type to theorems involving constants over the quotient type. Before we can describe this lifting process, we need to impose two restrictions in form of proof obligations that arise during the lifting. The reason is that even if definitions for all raw constants can be given, *not* all theorems can be lifted to the quotient type. Most notable is the bound variable function, that is the constant *bn*, defined for raw lambda-terms as follows

$$
\begin{array}{c}
\mathit{bn}\ (x) \triangleq \varnothing \qquad \mathit{bn}\ (t_1\ t_2) \triangleq \mathit{bn}\ (t_1) \cup \mathit{bn}\ (t_2) \\
\mathit{bn}\ (\lambda x.\ t) \triangleq \{x\} \cup \mathit{bn}\ (t)
\end{array}
$$

We can generate a definition for this constant using *ABS* and *REP*. But this constant does *not* respect α-equivalence and consequently no theorem involving this constant can be lifted to α-equated lambda terms. Homeier formulates the restrictions in terms of the properties of *respectfulness* and *preservation*. We have to slightly extend Homeier's definitions in order to deal with quotient compositions.

To formally define what respectfulness is, we have to first define the notion of *aggregate equivalence relations* using the function *REL*(σ, τ) The idea behind this function is to simultaneously descend into the raw types σ and quotient types τ, and generate the appropriate quotient equivalence relations in places where the types differ and equalities elsewhere.

equal types: $REL\ (\sigma,\ \sigma) \triangleq = :: \sigma \Rightarrow \sigma \Rightarrow bool$

equal type constructors:

$$REL\ (\sigma s\ \kappa,\ \tau s\ \kappa) \triangleq rel\_\kappa\ (REL\ (\sigma s,\ \tau s)) \quad (7)$$

unequal type constructors:

$REL\ (\sigma s\ \kappa,\ \tau s\ \kappa_q) \triangleq rel\_\kappa_q\ (REL\ (\sigma s',\ \tau s))$

The $\sigma s'$ in the last clause are calculated as in (6): again we know that type $\alpha s\ \kappa_q$ is the quotient of the raw type $\varrho s\ \kappa$. The $\sigma s'$ are the substitutions for αs obtained by matching $\varrho s\ \kappa$ and $\sigma s\ \kappa$.

Let us return to the lifting procedure of theorems. Assume we have a theorem that contains the raw constant $c_r :: \sigma$ and which we want to lift to a theorem where c_r is replaced by the corresponding constant $c_q :: \tau$ defined over a quotient type. In this situation we generate the following proof obligation

$$REL\ (\sigma,\ \tau)\ c_r\ c_r.$$

Homeier calls these proof obligations *respectfulness theorems*. However, unlike his quotient package, we might have several respectfulness theorems for one constant—he has at most one. The reason is that because of our quotient compositions, the types σ and τ are not completely determined by c_r. And for every instantiation of the types, a corresponding respectfulness theorem is necessary.

Before lifting a theorem, we require the user to discharge respectfulness proof obligations. In case of *bn* this obligation is

$$(\approx_\alpha \Rightarrow =)\ bn\ bn$$

and the point is that the user cannot discharge it: because it is not true. To see this, we can just unfold the definition of $\Rightarrow$ (4) using extensionality to obtain the false statement

$$\forall\, t_1\ t_2.\ if\ t_1 \approx_\alpha t_2\ then\ bn(t_1) = bn(t_2)$$

In contrast, lifting a theorem about *append* to a theorem describing the union of finite sets will mean to discharge the proof obligation

$$(\approx_{list} \Rightarrow \approx_{list} \Rightarrow \approx_{list})\ append\ append$$

To do so, we have to establish

$$\text{if } xs \approx_{list} ys \text{ and } us \approx_{list} vs \text{ then } xs\ @\ us \approx_{list} ys\ @\ vs$$

which is straightforward given the definition shown in (2).

The second restriction we have to impose arises from non-lifted polymorphic constants, which are instantiated to a type being quotient. For example, take the *cons*-constructor to add a pair of natural numbers to a list, whereby we assume the pair of natural numbers turns into an integer in the quotient construction. The point is that we still want to use *cons* for adding integers to lists—just with a different type. To be able to lift such theorems, we need a *preservation property* for *cons*. Assuming we have a polymorphic raw constant $c_r :: \sigma$ and a corresponding quotient constant $c_q :: \tau$, then a preservation property is as follows

$$Quot\ R_{\alpha s}\ Abs_{\alpha s}\ Rep_{\alpha s}\ implies\ ABS\ (\sigma,\ \tau)\ c_r = c_r$$

where the αs stand for the type variables in the type of c_r. In case of *cons* (which has type $\alpha \Rightarrow \alpha\ list \Rightarrow \alpha\ list$) we have

$$(Rep \mapsto map\_list\ Rep \mapsto map\_list\ Abs)\ cons = cons$$

under the assumption *Quot R Abs Rep*. The point is that if we have an instance of *cons* where the type variable α is instantiated with $nat \times nat$ and we also quotient this type to yield integers, then we need to show this preservation property.

5. LIFTING OF THEOREMS

The main benefit of a quotient package is to lift automatically theorems over raw types to theorems over quotient types. We will perform this lifting in three phases, called *regularization*, *injection* and *cleaning* according to procedures in Homeier's ML-code. Space restrictions, unfortunately, prevent us from giving anything but a sketch of these three phases. However, we will precisely define the input and output data of these phases (this was omitted in [6]).

The purpose of regularization is to change the quantifiers and abstractions in a "raw" theorem to quantifiers over variables that respect their respective relations (Definition 1 states what respects means). The purpose of injection is to add *Rep* and *Abs* of appropriate types in front of constants and variables of the raw type so that they can be replaced by the corresponding constants from the quotient type. The purpose of cleaning is to bring the theorem derived in the first two phases into the form the user has specified. Abstractly, our package establishes the following three proof steps:

1.) Regularization	$raw\_thm \longrightarrow reg\_thm$
2.) Injection	$reg\_thm \longleftrightarrow inj\_thm$
3.) Cleaning	$inj\_thm \longleftrightarrow quot\_thm$

which means, stringed together, the raw theorem implies the quotient theorem. In contrast to other quotient packages, our package requires that the user specifies both, the *raw_thm* (as theorem) and the *term* of the *quot_thm*.[3] As a result, the user has fine control over which parts of a raw theorem should be lifted.

The second and third proof step performed in package will always succeed if the appropriate respectfulness and preservation theorems are given. In contrast, the first proof step can fail: a theorem given by the user does not always imply a regularized version and a stronger one needs to be proved. An example for this kind of failure is the simple statement for integers $0 \neq 1$. One might hope that it can be proved by lifting $(0, 0) \neq (1, 0)$, but this raw theorem only shows that two particular elements in the equivalence classes are not equal. In order to obtain $0 \neq 1$, a more general statement stipulating that the equivalence classes are not equal is necessary. This kind of failure is beyond the scope where the quotient package can help: the user has to provide a raw theorem that can be regularized automatically, or has to provide an explicit proof for the first proof step. Homeier gives more details about this issue in the long version of [6].

In the following we will first define the statement of the regularized theorem based on *raw_thm* and *quot_thm*. Then we define the statement of the injected theorem, based on *reg_thm* and *quot_thm*. We then show the three proof steps, which can all be performed independently from each other.

We first define the function *REG*, which takes the terms of the *raw_thm* and *quot_thm* as input and returns *reg_thm*. The idea behind this function is that it replaces quantifiers and abstractions involving raw types by bounded ones, and equalities involving raw types by appropriate aggregate equivalence relations. It is defined by simultaneous recursion on the structure of the terms of *raw_thm* and *quot_thm* as follows:

[3]Though we also provide a fully automated mode, where the *quot_thm* is guessed from the form of *raw_thm*.

abstractions:

$$REG\ (\lambda x^{\sigma}.\ t,\ \lambda x^{\tau}.\ s) \triangleq \begin{cases} \lambda x^{\sigma}.\ REG\ (t,\ s) & \text{provided } \sigma = \tau \\ \lambda x^{\sigma} \in Resp\ (REL\ (\sigma,\ \tau)).\ REG\ (t,\ s) \end{cases}$$

universal quantifiers:

$$REG\ (\forall x^{\sigma}.\ t,\ \forall x^{\tau}.\ s) \triangleq \begin{cases} \forall x^{\sigma}.\ REG\ (t,\ s) & \text{provided } \sigma = \tau \\ \forall x^{\sigma} \in Resp\ (REL\ (\sigma,\ \tau)).\ REG\ (t,\ s) \end{cases}$$

equality: $REG\ (=^{\sigma \Rightarrow \sigma \Rightarrow bool},\ =^{\tau \Rightarrow \tau \Rightarrow bool}) \triangleq REL\ (\sigma,\ \tau)$

applications, variables and constants:

$REG\ (t_1\ t_2,\ s_1\ s_2) \triangleq REG\ (t_1,\ s_1)\ REG\ (t_2,\ s_2)$
$REG\ (x_1,\ x_2) \triangleq x_1$
$REG\ (c_1,\ c_2) \triangleq c_1$

In the above definition we omitted the cases for existential quantifiers and unique existential quantifiers, as they are very similar to the cases for the universal quantifier.

Next we define the function *INJ* which takes as argument *reg_thm* and *quot_thm* (both as terms) and returns *inj_thm*:

abstractions:

$$INJ\ (\lambda x.\ t :: \sigma,\ \lambda x.\ s :: \tau) \triangleq \begin{cases} \lambda x.\ INJ\ (t,\ s) & \text{provided } \sigma = \tau \\ REP\ (\sigma,\ \tau)\ (ABS\ (\sigma,\ \tau)\ (\lambda x.\ INJ\ (t,\ s))) \end{cases}$$

$$INJ\ (\lambda x \in R.\ t :: \sigma,\ \lambda x.\ s :: \tau) \triangleq REP\ (\sigma,\ \tau)\ (ABS\ (\sigma,\ \tau)\ (\lambda x \in R.\ INJ\ (t,\ s)))$$

universal quantifiers:

$INJ\ (\forall\ t,\ \forall\ s) \triangleq \forall\ INJ\ (t,\ s)$
$INJ\ (\forall\ t \in R,\ \forall\ s) \triangleq \forall\ INJ\ (t,\ s) \in R$

applications, variables and constants:

$INJ\ (t_1\ t_2,\ s_1\ s_2) \triangleq INJ\ (t_1,\ s_1)\ INJ\ (t_2,\ s_2)$

$$INJ\ ({x_1}^{\sigma},\ {x_2}^{\tau}) \triangleq \begin{cases} x_1 & \text{provided } \sigma = \tau \\ REP\ (\sigma,\ \tau)\ (ABS\ (\sigma,\ \tau)\ x_1) \end{cases}$$

$$INJ\ ({c_1}^{\sigma},\ {c_2}^{\tau}) \triangleq \begin{cases} c_1 & \text{provided } \sigma = \tau \\ REP\ (\sigma,\ \tau)\ (ABS\ (\sigma,\ \tau)\ c_1) \end{cases}$$

In this definition we again omitted the cases for existential and unique existential quantifiers.

In the first phase, establishing *raw_thm* $\longrightarrow$ *reg_thm*, we always start with an implication. Isabelle provides *mono* rules that can split up the implications into simpler implicational subgoals. This succeeds for every monotone connective, except in places where the function *REG* replaced, for instance, a quantifier by a bounded quantifier. To decompose them, we have to prove that the relations involved are aggregate equivalence relations.

The second phase, establishing *reg_thm* $\longleftrightarrow$ *inj_thm*, starts with an equality between the terms of the regularized theorem and the injected theorem. The proof again follows the structure of the two underlying terms taking respectfulness theorems into account.

We defined the theorem *inj_thm* in such a way that establishing in the third phase the equivalence *inj_thm* $\longleftrightarrow$ *quot_thm* can be achieved by rewriting *inj_thm* with the preservation theorems and quotient definitions. This step also requires that the definitions of all lifted constants are used to fold the *Rep* with the raw constants. We will give more details about our lifting procedure in a longer version of this paper.

6. CONCLUSION AND RELATED WORK

The code of the quotient package and the examples described here are already included in the standard distribution of Isabelle. [4] The package is heavily used in the new version of Nominal Isabelle, which provides a convenient reasoning infrastructure for programming language calculi involving general binders. To achieve this, it builds types representing α-equivalent terms. Earlier versions of Nominal Isabelle have been used successfully in formalisations of an equivalence checking algorithm for LF [12], Typed Scheme [11], several calculi for concurrency [2] and a strong normalisation result for cut-elimination in classical logic [13].

There is a wide range of existing literature for dealing with quotients in theorem provers. Slotosch [10] implemented a mechanism that automatically defines quotient types for Isabelle/HOL. But he did not include theorem lifting. Harrison's quotient package [5] is the first one that is able to automatically lift theorems, however only first-order theorems (that is theorems where abstractions, quantifiers and variables do not involve functions that include the quotient type). There is also some work on quotient types in non-HOL based systems and logical frameworks, including theory interpretations in PVS [8], new types in MetaPRL [7], and setoids in Coq [3]. Paulson showed a construction of quotients that does not require the Hilbert Choice operator, but also only first-order theorems can be lifted [9]. The most related work to our package is the package for HOL4 by Homeier [6]. He introduced most of the abstract notions about quotients and also deals with lifting of higher-order theorems. However, he cannot deal with quotient compositions (needed for lifting theorems about *flat*). Also, a number of his definitions, like *ABS*, *REP* and *INJ* etc only exist in [6] as ML-code, not included in the paper. Like Homeier's, our quotient package can deal with partial equivalence relations, but for lack of space we do not describe the mechanisms needed for this kind of quotient constructions.

One feature of our quotient package is that when lifting theorems, the user can precisely specify what the lifted theorem should look like. This feature is necessary, for example, when lifting an induction principle for two lists. Assuming this principle has as the conclusion a predicate of the form *P xs ys*, then we can precisely specify whether we want to quotient *xs* or *ys*, or both. We found this feature very useful in the new version of Nominal Isabelle, where such a choice is required to generate a reasoning infrastructure for alpha-equated terms.

Acknowledgements: We would like to thank Peter Homeier for the many discussions about his HOL4 quotient package and explaining to us some of its finer points in the implementation. Without his patient help, this work would have been impossible.

7. REFERENCES

[1] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1981.

[2] J. Bengtson and J. Parrow. Psi-Calculi in Isabelle. In *Proc of the 22nd TPHOLs Conference*, volume 5674 of *LNCS*, pages 99–114, 2009.

[3] L. Chicli, L. Pottier, and C. Simpson. Mathematical Quotients and Quotient Types in Coq. In *Proc of the TYPES workshop*, volume 2646 of *LNCS*, pages 95–107, 2002.

[4] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1958.

[5] J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[6] P. V. Homeier. A design structure for higher order quotients. In *Proc of the 18th TPHOLs conference*, volume 3603 of *LNCS*, pages 130–146, 2005.

[4] Available from http://isabelle.in.tum.de/.

[7] A. Nogin. Quotient types: A modular approach. In *Proc. of the 15th TPHOLs conference*, volume 2646 of *LNCS*, pages 263–280, 2002.
[8] S. Owre and N. Shankar. Theory Interpretations in PVS. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001.
[9] L. C. Paulson. Defining functions on equivalence classes. *ACM Trans. Comput. Log.*, 7(4):658–675, 2006.
[10] O. Slotosch. Higher order quotients and their implementation in isabelle hol. In *Proc. of the 10th TPHOLs conference*, volume 1275 of *LNCS*, pages 291–306, 1997.
[11] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *Proc. of the 35rd POPL Symposium*, pages 395–406. ACM, 2008.
[12] C. Urban, J. Cheney, and S. Berghofer. Mechanizing the Metatheory of LF. In *Proc. of the 23rd LICS Symposium*, pages 45–56, 2008.
[13] C. Urban and B. Zhu. Revisiting Cut-Elimination: One Difficult Proof is Really a Proof. In *Proc. of the 9th RTA Conference*, volume 5117 of *LNCS*, pages 409–424, 2008.

APPENDIX

A. EXAMPLES

In this appendix we will show a sequence of declarations for defining the type of integers by quotienting pairs of natural numbers, and lifting one theorem.

A user of our quotient package first needs to define a relation on the raw type with which the quotienting will be performed. We give the same integer relation as the one presented in (1):

fun *int_rel :: (nat × nat) ⇒ (nat × nat) ⇒ (nat × nat)*
where *int_rel (m, n) (p, q) = (m + q = n + p)*

Next the quotient type must be defined. This generates a proof obligation that the relation is an equivalence relation, which is solved automatically using the definition of equivalence and extensionality:

quotient_type *int* **=** *(nat × nat)* **/** *int_rel*
by (auto simp add: equivp_def expand_fun_eq)

The user can then specify the constants on the quotient type:

quotient_definition *0 :: int* **is** *(0 :: nat, 0 :: nat)*

fun *add_pair*
where *add_pair (m, n) (p, q)* ≜ *(m + p :: nat, n + q :: nat)*
quotient_definition *+ :: int ⇒ int ⇒ int* **is** *add_pair*

The following theorem about addition on the raw level can be proved.

lemma *add_pair_zero: int_rel (add_pair (0, 0) x) x*

If the user lifts this theorem, the quotient package performs all the lifting automatically leaving the respectfulness proof for the constant *add_pair* as the only remaining proof obligation. This property needs to be proved by the user:

lemma *[quot_respect]:*
(int_rel ⇾ int_rel ⇾ int_rel) add_pair add_pair

It can be discharged automatically by Isabelle when hinting to unfold the definition of ⇾. After this, the user can prove the lifted lemma as follows:

lemma *0 + (x :: int) = x* **by** *lifting add_pair_zero*

or by using the completely automated mode stating just:

thm *add_pair_zero[quot_lifted]*

Both methods give the same result, namely $0 + x = x$ where x is of type integer. Although seemingly simple, arriving at this result without the help of a quotient package requires a substantial reasoning effort (see [9]).