Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL

Brian Huffman and Ondřej Kunčar

Technische Universität München
{huffman,kuncar}@in.tum.de

Abstract. Quotients, subtypes, and other forms of type abstraction are ubiquitous in formal reasoning with higher-order logic. Typically, users want to build a library of operations and theorems about an abstract type, but they want to write definitions and proofs in terms of a more concrete representation type, or “raw” type. Earlier work on the Isabelle Quotient package [3, 4] has yielded great progress in automation, but it still has many technical limitations.
We present an improved, modular design centered around two new packages: the *Transfer* package for proving theorems, and the *Lifting* package for defining constants. Our new design is simpler, applicable in more situations, and has more user-friendly automation. An initial implementation is available in Isabelle 2012.

1 Introduction

Quotients and subtypes are everywhere in Isabelle/HOL. For example, basic numeric types like integers, rationals, reals, and finite words are all quotients. Many other types in Isabelle are implemented as subtypes, including multisets, finite maps, polynomials, fixed-length vectors, matrices, and formal power series, to name a few.

Quotients and subtypes are useful as type abstractions: Instead of explicitly asserting that a function respects an equivalence relation or preserves an invariant, this information can be encoded in the function’s type. Quotients are also particularly useful in Isabelle, because reasoning about equality on an abstract type is supported much better than reasoning modulo an equivalence relation.

Building a theory library that implements a new abstract type can take a lot of work. The challenges are similar for both quotients and subtypes: Isabelle requires explicit coercion functions (often “*Rep*” and “*Abs*”) to convert between old “raw” types and new abstract types. Definitions of functions on abstract types require complex combinations of these coercions. Users must prove numerous lemmas about how the coercions interact with the abstract functions. Finally, it takes much effort to transfer all the interesting properties of raw functions up to the abstract level. Clearly, there is a need for good proof automation for this process.

1.1 Related Work

Much previous work has been done on formalizing quotients in theorem provers. Slotosch [8] and Paulson [6] each developed techniques for defining quotient types and defining first-order functions on them. They provided limited automation for transferring properties from raw to abstract types in the form of lemmas that facilitate manual proofs. Harrison [2] implemented tools in the HOL theorem prover for lifting constants and transferring theorems automatically, although this work was still limited to first-order constants and theorems. In 2005, Homeier [3] published a design for a new HOL package, which was the first system capable of lifting higher-order functions and transferring higher-order theorems.

Isabelle's Quotient package was implemented by Kaliszyk and Urban [4], based upon Homeier's design. It was first released with Isabelle 2009-2, in June 2010. The Quotient package implements a collection of commands, proof methods, and theorem attributes. The primary ones are as follows:

– **quotient_type** command: Defines a new quotient type, based on a user-specified raw type and a (total or partial) equivalence relation.
– **quotient_definition** command: Defines a function on a quotient type, based on a user-provided function on a raw type.
– descending method: Replaces a proof goal containing operations on a quotient type with a goal about the corresponding raw operations.
– lifting method: Solves a proof goal about a quotient type, using the given theorem about raw types.

The Quotient package is designed around the notion of a *quotient*, which involves two types and three constants: First we have a raw type 'a with a partial equivalence relation R :: 'a ⇒ 'a ⇒ bool. Next we have the abstract type 'b, whose elements are in one-to-one correspondence with the equivalence classes of R: The *abstraction* function *Abs* :: 'a ⇒ 'b maps each equivalence class of R onto a single abstract value, and the *representation* function *Rep* :: 'b ⇒ 'a takes each abstract value to an arbitrary element of its corresponding equivalence class.

Given a raw type and a relation R, the **quotient_type** command introduces a new type with *Abs* and *Rep* that form a quotient. Given a function g on the raw type, the **quotient_definition** command then defines a new abstract function g' in terms of g, *Abs*, and *Rep*. The user must also provide a *respectfulness theorem* showing that g respects R. Finally the descending and lifting methods can transfer propositions between g and g'. Internally, this involves respectfulness theorems, the definition of g', and the quotient properties of R, *Abs* and *Rep*.

1.2 Limitations of the Quotient package

We decided to redesign the existing Quotient package after identifying several limitations of its implementation. While some are relatively superficial and could be remedied with minor changes, others have deeper roots. A few such technical limitations were described by Alex Krauss [5]:

1. The quotient relation R must be a dedicated constant. This is inconvenient when we want to use equality, since it requires an extra constant that must be unfolded frequently.
2. Using quotient definition, one can only lift constants, not arbitrary terms. This prevents the use of the tool on things like locale parameters and some definitions in a local theory.
3. One cannot turn a type defined by other means into a quotient afterwards.
4. One cannot declare a user-defined constant on the quotient type as the lifted version of another constant.

To solve problems 1 and 2 does not require major organizational changes, just local internal ones (see Section 3.2 for details). However, problems 3 and 4 suggested to us that splitting the Quotient package into various layers might make sense: By having separate components with well-defined interfaces, we could make it easier for users to connect with the package in non-standard ways.

Besides the problems noted by Krauss, we have identified some additional problems with the descending/lifting methods ourselves. Consider 'a fset, a type of finite sets which is a quotient of 'a list. The Quotient package can generate fset versions of the list functions map :: ('a ⇒ 'b) ⇒ 'a list ⇒ 'b list and concat :: 'a list list ⇒ 'a list, but it has difficulty transferring the following collection of theorems to fset:

concat (map ($\lambda x.$ [x]) xs) = xs
map f (concat xss) = concat (map (map f) xss)
concat (map concat $xsss$) = concat (concat $xsss$)

The problem is with the user-supplied respectfulness theorems. Note that map occurs at several different type instances here: It is used with functions of types 'a ⇒ 'b, 'a ⇒ 'a list, and 'a list ⇒ 'b list. Unfortunately a single respectfulness theorem for map will not work in all these cases—each type instance requires a different respectfulness theorem. On top of that, the user must also prove additional *preservation lemmas*, essentially alternative definitions of map_fset at different types. These rules can be tricky to state correctly and tedious to prove.

The Quotient package's complex, three-phase transfer procedure was another motivation to look for a new design. We wanted to have a simpler implementation, involving fewer separate phases. We also wanted to ease the burden of user-supplied rules, by requiring only one rule per constant. Finally, we wanted a more general, more widely applicable transfer procedure that did not have so many hard-wired assumptions about quotients.

1.3 Overview

Our new system uses a layered design, with multiple components and interfaces that are related as shown in Figure 1. Each component depends only on the components underneath it. At the bottom is the Transfer package, which is responsible for transferring propositions between raw and abstract types (Section 2). Note that the Transfer package has no dependencies; it does not know anything about *Rep* and *Abs* functions or quotient predicates.

Above Transfer is the Lifting package, which lifts constant definitions from raw to abstract types (Section 3). It configures each new constant to work with Transfer. At the top are commands that configure new types to work with Lifting, such as **setup_lifting** and **quotient_type** (Section 3). We expect that additional type definition commands might be implemented later; we discuss this and other ideas for future work in the conclusion (Section 4).

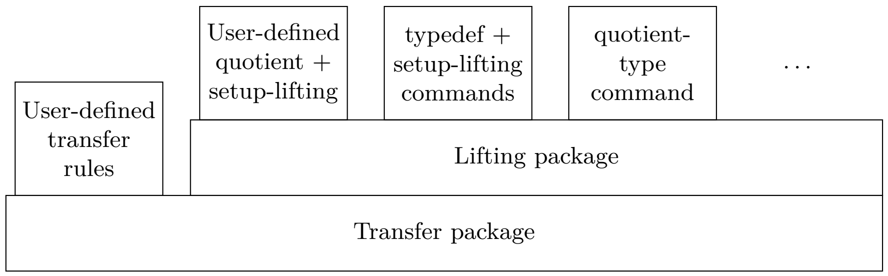

User-defined transfer rules
User-defined quotient + setup-lifting
typedef + setup-lifting commands
quotient-type command
. . .
Lifting package
Transfer package

Fig. 1. Modular design of packages for formalizing quotients

2 Transfer package

The Transfer package allows users to establish connections between pairs of related types or constants, by registering *transfer rules*. The core functionality of the Transfer package is then to use these rules to prove equivalences between pairs of related propositions.

The Transfer package provides the transfer proof method, which replaces the current subgoal with a logically equivalent one that uses different types and constants. In the context of quotient types, transfer reduces a subgoal about a quotient type to a subgoal about the underlying raw type. But it is not restricted to quotients; it works more generally to transfer between any combination of types, if it is configured with an appropriate set of transfer rules.

Types as relations. The design of the Transfer package is based on the idea of types as binary relations. This concept comes from work on relational parametricity by Reynolds [7] and the "free theorems" of Wadler [9].

Relational parametricity tells us that different type instances of a parametrically polymorphic function must behave uniformly—that is, they must be related by a binary relation derived from the function's type. For example, the standard filter function on lists satisfies the parametricity property shown below in Eq. (2). The relation is derived from filter's type by replacing each type constructor with an appropriate relator; Figure 2 gives the definitions of a few standard relators.

```
definition prod_rel ::
    "('a₁ ⇒ 'a₂ ⇒ bool) ⇒ ('b₁ ⇒ 'b₂ ⇒ bool) ⇒ 'a₁ × 'b₁ ⇒ 'a₂ × 'b₂ ⇒ bool"
  where "(prod_rel A B) x y ≡ A (fst x) (fst y) ∧ B (snd x) (snd y)

definition fun_rel ::
    "('a₁ ⇒ 'a₂ ⇒ bool) ⇒ ('b₁ ⇒ 'b₂ ⇒ bool) ⇒ ('a₁ ⇒ 'b₁) ⇒ ('a₂ ⇒ 'b₂) ⇒ bool"
    (infixr "⤇" 55)
  where "(A ⤇ B) f g ≡ (∀x y. A x y ⟶ B (f x) (g y))"

definition list_all2 :: "('a₁ ⇒ 'a₂ ⇒ bool) ⇒ 'a₁ list ⇒ 'a₂ list ⇒ bool"
  where "(list_all2 A) xs ys ≡ length xs = length ys ∧ (∀(x, y) ∈ set (zip xs ys). A x y)"

definition set_rel :: "('a₁ ⇒ 'a₂ ⇒ bool) ⇒ 'a₁ set ⇒ 'a₂ set ⇒ bool"
  where "(set_rel A) X Y ≡ (∀x∈X. ∃y∈Y. A x y) ∧ (∀y∈Y. ∃x∈X. A x y)"
```

Fig. 2. Definitions of relators for various type constructors

For base types like bool or int we use identity relations ($\longleftrightarrow$ or $=$).

$$\mathsf{filter} :: (\text{'a} \Rightarrow \mathsf{bool}) \Rightarrow \text{'a}\ \mathsf{list} \Rightarrow \text{'a}\ \mathsf{list} \tag{1}$$

$$\forall A.\ ((A \Mapsto op \longleftrightarrow) \Mapsto \mathsf{list\_all2}\ A \Mapsto \mathsf{list\_all2}\ A)\ \mathsf{filter}\ \mathsf{filter} \tag{2}$$

This parametricity property means that if predicates p_1 and p_2 agree on related inputs (i.e., $A\ x_1\ x_2$ implies $p_1\ x_1 \longleftrightarrow p_2\ x_2$) then filter p_1 and filter p_2 applied to related lists will yield related results. (Wadler-style free theorems are derived by instantiating A with the graph of a function f; in this manner, we can obtain a rule stating essentially that filter commutes with map.)

The Transfer package uses parametricity rules in the style of Eq. (2) as transfer rules. However, we also generalize the scheme a bit—not every transfer rule is simply a parametricity rule derived from a single polymorphic constant. In general, a transfer rule may relate two different constants, and in addition to identity relations and the standard relators, it may also use *transfer relations* between distinct types or type constructors.

Example: Int/nat transfer. We consider a simple use case, transferring propositions between the integers and natural numbers. To specify the connection between the two types, we define a transfer relation ZN :: int $\Rightarrow$ nat $\Rightarrow$ bool.

$$\mathsf{ZN}\ x\ n \equiv (x = \mathsf{int}\ n) \tag{3}$$

We can then use ZN to express relationships between constants in the form of transfer rules. Obviously, the integer 1 corresponds to the natural number 1. Integer addition corresponds to addition on naturals, in that related arguments are mapped to related results. Similarly, less-than on integers corresponds to less-than on naturals. Finally, bounded quantification over the non-negative integers corresponds to universal quantification over type nat.

$$(\mathsf{ZN})\ (1{::}\mathsf{int})\ (1{::}\mathsf{nat}) \tag{4}$$

$$(\mathsf{ZN} \Mapsto \mathsf{ZN} \Mapsto \mathsf{ZN})\ (op\ +)\ (op\ +) \tag{5}$$

$$(\mathsf{ZN} \Mapsto \mathsf{ZN} \Mapsto op\ =)\ (op\ <)\ (op\ <) \tag{6}$$

$$((\mathsf{ZN} \Mapsto op\ =) \Mapsto op\ =)\ (\mathsf{Ball}\ \{0..\})\ \mathsf{All} \tag{7}$$

The Transfer package can use the transfer rules above, along with introduction and elimination rules for $\Mapsto$, to derive equivalences like the following. The derivation follows the syntactic structure of the terms.

$$(\forall x{::}\mathsf{int} \in \{0..\}.\ x < x + 1) \longleftrightarrow (\forall n{::}\mathsf{nat}.\ n < n + 1) \tag{8}$$

If we apply the transfer method to a subgoal of the form $\forall n{::}\mathsf{nat}.\ n < n + 1$, the Transfer package will prove the equivalence above, and then use it to replace the subgoal with $\forall x{::}\mathsf{int} \in \{0..\}.\ x < x + 1$. In general, transfer can handle any lambda term constructed from constants for which it has transfer rules.

2.1 Parameterized transfer relations

The design of the new Transfer package generalizes easily to transfer relations with parameters. As an example, we define a relation between lists and a finite set type; it is parameterized by a relation on the element types. We assume a function $\mathsf{Fset} :: \mathsf{'a\ list} \Rightarrow \mathsf{'a\ fset}$ that converts the given list to a finite set.

$$\mathsf{LF} :: (\mathsf{'a}_1 \Rightarrow \mathsf{'a}_2 \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{'a}_1\ \mathsf{list} \Rightarrow \mathsf{'a}_2\ \mathsf{fset} \Rightarrow \mathsf{bool} \tag{9}$$

$$(\mathsf{LF}\ A)\ xs\ Y \equiv \exists ys.\ \mathsf{list\_all2}\ A\ xs\ ys \land \mathsf{Fset}\ ys = Y \tag{10}$$

If we define versions of the functions map and concat that work on finite sets, we can relate them to the list versions with the transfer rules shown here.

$$((A \Mapsto B) \Mapsto \mathsf{LF}\ A \Mapsto \mathsf{LF}\ B)\ \mathsf{map}\ \mathsf{map\_fset} \tag{11}$$

$$(\mathsf{LF}\ (\mathsf{LF}\ A) \Mapsto \mathsf{LF}\ A)\ \mathsf{concat}\ \mathsf{concat\_fset} \tag{12}$$

These rules allow the transfer method to work on formerly problematic goals such as $\mathsf{map\_fset}\ f\,(\mathsf{concat\_fset}\ xss) = \mathsf{concat\_fset}\ (\mathsf{map\_fset}\ (\mathsf{map\_fset}\ f)\ xss)$ and $\mathsf{concat\_fset}\ (\mathsf{map\_fset}\ \mathsf{concat\_fset}\ xsss) = \mathsf{concat\_fset}\ (\mathsf{concat\_fset}\ xsss)$, as long as appropriate transfer rules for equality are also present. Unlike the Quotient package, the same transfer rules work for all type instances of these constants.

2.2 Bi-total and bi-unique relations

Some polymorphic functions in Isabelle require side conditions on their parametricity theorems. For example, consider the equality relation =, which has the polymorphic type $\mathsf{'a} \Rightarrow \mathsf{'a} \Rightarrow \mathsf{bool}$. Its type would suggest the parametricity property $(A \Mapsto A \Mapsto op \longleftrightarrow)\ (op =)\ (op =)$, but this does not hold for all relations A—it only holds if A is *bi-unique*, i.e., single-valued and injective.

$$\mathsf{bi\_unique}\ A \Longrightarrow (A \Mapsto A \Mapsto op \longleftrightarrow)\ (op =)\ (op =) \tag{13}$$

As pointed out by Wadler [9], this restriction on relations is akin to an *eqtype* annotation in ML, or an *Eq* class constraint in Haskell. While Haskell allows users to provide *Eq* instance declarations, the Transfer package allows us to provide additional rules about bi-uniqueness that serve the same purpose.

$$\mathsf{bi\_unique}\ A \Longrightarrow \mathsf{bi\_unique}\ (\mathsf{set\_rel}\ A) \tag{14}$$

$$\mathsf{bi\_unique}\ A \Longrightarrow \mathsf{bi\_unique}\ (\mathsf{list\_all2}\ A) \tag{15}$$

$$\mathsf{bi\_unique}\ \mathsf{ZN} \tag{16}$$

Using the above rules, the Transfer package is able to relate equality on lists of integers with equality on lists of naturals, using the relation $\mathsf{list\_all2}\ \mathsf{ZN}$. It can similarly relate equality on sets, lists of sets, sets of lists, and so on.

The universal quantifier requires a different side condition on its parametricity rule. While equality requires bi-uniqueness, the universal quantifier requires the relation A to be *bi-total*—i.e., A must be both total and surjective.

$$\mathsf{bi\_total}\ A \Longrightarrow ((A \Mapsto op \longleftrightarrow) \Mapsto op \longleftrightarrow)\ \mathsf{All}\ \mathsf{All} \tag{17}$$

Like bi-uniqueness, bi-totality is preserved by many relators, including those for lists and sets. The relation ZN is not bi-total, but transfer relations induced by total quotient types are.

2.3 Proving implications instead of equivalences

If supplied with appropriate transfer rules, the Transfer package can be made to prove implications instead of logical equivalences. We provide a variant proof method $\mathsf{transfer}'$ for this purpose. While transfer always replaces a proof goal with an equivalent one, $\mathsf{transfer}'$ is free to replace a goal with a stronger one.

Using implication, we can state transfer rules for equality and universal quantification with weaker side conditions, namely right-uniqueness and right-totality. A relation is right-unique if it is single-valued, and right-total if it is surjective. Both properties are preserved by relators for many type constructors.

$$\mathsf{right\_unique}\ A \Longrightarrow (A \Mapsto A \Mapsto op \longrightarrow)\ (op =)\ (op =) \tag{18}$$

$$\mathsf{right\_total}\ A \Longrightarrow ((A \Mapsto op \longrightarrow) \Mapsto op \longrightarrow)\ \mathsf{All}\ \mathsf{All} \tag{19}$$

Our example relation ZN is not bi-total, but it is right-total. This means that the Transfer package can prove implications like the one here, stating that a universally quantified proposition on type int implies a related one on nat.

$$(\forall x{::}\mathsf{int}.\ x + 0 = x) \longrightarrow (\forall n{::}\mathsf{nat}.\ n + 0 = n) \tag{20}$$

Similarly, our example relator LF from Section 2.1 does not preserve bi-uniqueness, but it does preserve right-uniqueness. In particular, the relation $\mathsf{LF}\ (\mathsf{LF}\ (op\ =))$ is right unique, which means that an equality on $\mathsf{'a\ list\ list}$ implies a corresponding equality on $\mathsf{'a\ fset\ fset}$. If the Transfer package were restricted to $\longleftrightarrow$, then we would have to find a relation on $\mathsf{'a\ list\ list}$ that is logically equivalent to equality on $\mathsf{'a\ fset\ fset}$. Such a relation is awkward to state and to reason about, so implication-based rules are very convenient in this case.

3 Lifting package

The Lifting package allows users to lift functions operating on the raw level to the abstract level. Doing this lifting manually usually implies tedious, uninteresting work—the main point of the Lifting package is to automate this work as much as possible. Besides defining the constant, another non-trivial feature is to generate a code equation for the lifted constant.

The Lifting package provides two main commands: **setup_lifting** for initializing the package to work with a new type (Section 3.1), and **lift_definition** for actually lifting constants (Section 3.2).

3.1 Setting up the Lifting package

There is a small, well-defined interface for setting up the Lifting package via ML. The setup requires only a quotient theorem of the form $\mathsf{Quotient}\ R\ Abs\ Rep\ T$, which captures the fact that the partial equivalence relation R and the abstraction and representation functions Abs and Rep form a quotient. The predicate $\mathsf{Quotient}$ is similar to the predicate used in the original Quotient package [4] (see also Section 1.1) but an additional fourth parameter T has been added. T is a *transfer relation* relating the raw type and the quotient (abstract) type. (For details see Section 2.) The transfer relation appears in the generated transfer rule for each lifted constant, which is used by the Transfer package. The full, updated definition of the $\mathsf{Quotient}$ predicate in Isabelle 2012 is as follows:

definition
"$\mathsf{Quotient}\ R\ Abs\ Rep\ T \longleftrightarrow$
$(\forall a.\ Abs\ (Rep\ a) = a)\ \wedge$
$(\forall a.\ R\ (Rep\ a)\ (Rep\ a))\ \wedge$
$(\forall r\ s.\ R\ r\ s \longleftrightarrow R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s)\ \wedge$
$T = (\lambda x\ y.\ R\ x\ x \wedge Abs\ x = y)$"

The setup interface also accepts an additional optional argument: a theorem $\mathsf{reflp}\ R$ that witnesses that R is reflexive, and thus that it is a total equivalence relation. The Lifting package generates a different set of transfer rules and different kinds of code equations depending on whether R is total, and also on whether R is a subset of the identity relation; see Table 1 for a comparison of the four recognized categories of quotients.

[1] $\mathsf{ratrel}\ x\ y \equiv \mathsf{snd}\ x \neq 0 \wedge \mathsf{snd}\ y \neq 0 \wedge \mathsf{fst}\ x * \mathsf{snd}\ y = \mathsf{fst}\ y * \mathsf{snd}\ x$

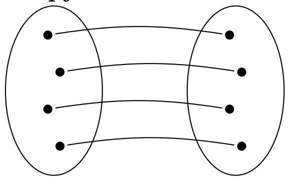
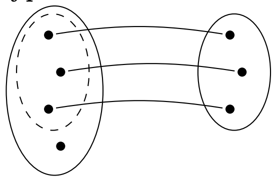
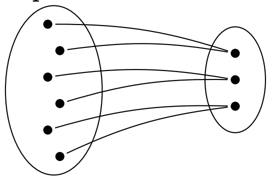
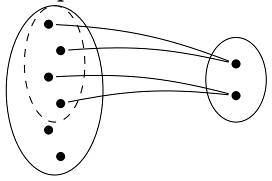
	total equivalence relation	partial equivalence relation
trivial relation (subset of =)	**type copy**  rep_eq: yes abs_eq: yes code: abs_eq relation: bi-unique, bi-total example: Executable_Relation 'a rel = ('a × 'a) set	**subtype**  rep_eq: yes abs_eq: only with assumptions code: rep_eq relation: bi-unique, right-total example: Lift_Dlist 'a dlist = $\{x ::$ 'a list. distinct $x\}$
non-trivial relation	**total quotient**  rep_eq: no abs_eq: yes code: abs_eq relation: right-unique, bi-total example: Lift_FSet 'a fset = 'a list / ($\lambda x\ y.$ set x = set y)	**partial quotient**  rep_eq: no abs_eq: only with assumptions code: none relation: right-unique, right-total example: Rat 'a rat = int × int / ratrel[1]

Table 1. Categorization of quotients and respective equations

Setting up the code generator Our ML interface also sets up the code generator [1] for the new type. For a total quotient or type copy we use the standard code generator setup, i.e. the abstraction function *Abs* is registered as a code datatype contructor. For these types, we use an *abstraction function equation* (expressed in terms of *Abs*) as a code equation for each lifted constant. For a subtype, an advanced mode of the code generator is used where the quotient type is registered as an abstract type. In this setting, a *representation function equation*[2] (expressed in terms of *Rep*) is used as a code equation. See Table 1 for an overview of code setup; we discuss generation of the abstraction and representation function equations in Section 3.2.

Transfer rules Various transfer rules are generated during the setup process. The set of generated rules depends on which category of quotient we have.

[2] In the context of the code generator, representation function equation is also called a *code certificate*.

Given a quotient theorem Quotient R Abs Rep T, we can always prove that T is right-unique and right-total. If we have a subtype or a type copy, we generate bi_unique T as a transfer rule. In case of a type copy or a total quotient then bi_total T is generated. Special transfer rules for equality and quantifiers are generated if appropriate. For example, partial and total quotients get a transfer rule relating R to equality. For subtypes and partial quotients, we generate a transfer rule for the universal quantifier, where Respects R is defined as $\{x.\ R\ x\ x\}$.

$$(T \Mapsto T \Mapsto op \longleftrightarrow)\ R\ (op =)$$
$$((T \Mapsto op \longleftrightarrow) \Mapsto op \longleftrightarrow)\ (\mathsf{Ball}\ (\mathsf{Respects}\ R))\ \mathsf{All}$$

Theory file interface We provide multiple ways to access the described setup interface from a theory file; these will be described in the following paragraphs.

The command **setup_lifting** *with a quotient theorem.* One can use the command **setup_lifting**, which is an Isar counterpart of the ML interface described above. It takes a quotient theorem Quotient R Abs Rep T as a first argument and reflp R as an optional second argument.

A good example of this usage is the formalization of the Isabelle word type, which is a type of finite bit strings. The idea is that we may not want to define this type using the standard quotient construction, in terms of equivalence classes of integers modulo 2^{32}. Instead, note that we have a *normalizing function* mod that maps each equivalence class onto a single well-determined representative between 0 and $2^{32} - 1$. This means that we can define type word in a simpler way, using **typedef**.

```
typedef (open) word = "{(0::int) ..< 2^32}"
  morphisms uint Abs_word

definition word_of_int :: "int ⇒ word"
  where "word_of_int k ≡ Abs_word (k mod 2^32)"

definition cr_word :: "int ⇒ word ⇒ bool"
  where "cr_word ≡ (λx y. word_of_int x = y)"

lemma Quotient_word:
  "Quotient (λx y. x mod 2^32 = y mod 2^32) word_of_int uint cr_word"

lemma reflp_word: "reflp (λx y. x mod 2^32 = y mod 2^32)"

setup_lifting (no_code) Quotient_word reflp_word
```

Fig. 3. Definition of word type and manual configuration of Lifting package

As shown in Figure 3, we can state the quotient theorem using the normalizing function mod, a new abstraction function word_of_int, and a transfer relation cr_word.[3]

With the **setup_lifting** command, we can use the simplest construction of type word, and we still get to take advantage of the automation of the Lifting package for defining word operations. One can imagine that the whole process can be automatized by a new command that would allow a user to specify a normalizing function.

The command **setup_lifting** *with a typedef theorem.* The command **setup_lifting** is overloaded: It can be also used with a typedef theorem instead of a quotient theorem. Internally, the typedef theorem is used to derive a quotient theorem, which is then used to set up the Lifting package as usual. An example of the described usage is a formalization of distinct lists[4]:

```
typedef (open) 'a dlist = "{xs::'a list. distinct xs}"
  morphisms list_of_dlist Abs_dlist
```

The theorem type_definition_dlist is generated by the **typedef** command and this theorem can be used for setting up the Lifting package:

```
setup_lifting type_definition_dlist
```

In order to state a quotient theorem using the Quotient predicate, we have to encode the invariant distinct as a partial equivalence relation. The corresponding relation is invariant distinct, where invariant is defined as follows:[5]

```
definition invariant :: "('a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"
  where "invariant P = (λx y. P x ∧ x = y)"
```

Besides the partial equivalence relation, a transfer relation T between 'a list and 'a dlist has to be defined. A canonical name cr_dlist is chosen for this relation, and it is defined in terms of the representation function list_of_dlist:

$$\text{cr\_dlist} \equiv \lambda x\ y.\ x = \text{list\_of\_dlist}\ y$$

The definition of the Quotient predicate specifies the transfer relation in terms of the abstraction function, $T = (\lambda x\ y.\ R\ x\ x \wedge \mathit{Abs}\ x = y)$. For a partial quotient the definition would take that form, but for a subtype the above definition using the representation function is equivalent. (Generally it is nicer to work with *Rep* than *Abs* from a typedef, because *Rep* is completely specified, while *Abs* is an underspecified function.)

[3] This is a simplification of the type defined in Isabelle's Word library, which has a variable size determined by a type parameter.

[4] A *distinct* list is defined as a list with no repeated elements.

[5] The short name invariant is hidden outside the Lifting package; users must refer to the fully qualified name Lifting.invariant.

The command **quotient_type**. The last possibility is to use the good old command **quotient_type** from the Quotient package. This command defines a new type using the standard quotient construction and proves a quotient theorem. For Isabelle 2012 we have patched the original **quotient_type** command so that it registers the quotient theorem with the Lifting package, saving the user from having to do this manually.

3.2 Lifting functions

The command **lift_definition** $f :: \tau$ **is** t defines a new function f with the abstract type τ in terms of a corresponding operation t on a representation type, using an appropriate combination of abstraction and representation functions. The term t does not have to be necessarily a constant but it can be any term. The following theorems are usually generated when using **lift_definition**, although f.rep_eq and f.abs_eq are generated conditionally on the category of quotient (see Table 1).

– f_def – definition of f
– f.rsp – respectfulness theorem in its internal form
– f.transfer – transfer rule for the Transfer package
– f.rep_eq – representation function equation, relates the lifted function and the raw function using only representation functions
– f.abs_eq – abstraction function equation, relates the lifted function and the raw function using only abstration functions[6]

Definition and respectfulness theorem We will present how the lifting process works on a running example. Let us use the type 'a dlist of distinct lists defined in the previous section, and define an abstract insert operation. We do it by lifting an insert operation on normal lists List.insert:

lift_definition insert :: "'a ⇒ 'a dlist ⇒ 'a dlist" **is** List.insert

The command **lift_definition** opens a proof environment where a user has to prove the *respectfulness theorem*. The respectfulness theorem is a correctness condition certifying that it makes sense to lift this particular function to the abstract level. In its internal form, the respectfulness theorem consists of a combination of equivalence relations. For insert it is this theorem:

$(op = \Mapsto \text{invariant distinct} \Mapsto \text{invariant distinct})$ List.insert List.insert

However, the user does not see this—the Lifting package does some preprocessing to present a proof goal in a user-friendly, readable form. The level of preprocessing depends on the style of quotient involved: For non-trivial quotients, the goal is processed so that users do not see $\Mapsto$, and for type copies we have implemented a procedure that discharges the entire proof automatically. In our insert example, we must prove the following respectfulness obligation:

[6] Available in post–Isabelle 2012 development versions

$\bigwedge a\ list.$ distinct $list \Longrightarrow$ distinct (List.insert $a\ list$)

Because the type 'a dlist is defined as a subtype, the respectfulness obligation is simpler than it would be for a non-trivial quotient type. The processing ensures that the relation constructor invariant is not visible to users.[7] The presented goal says merely that List.insert preserves the subtype invariant.

After the respectfulness obligation is proved, a new constant insert is defined and its type is 'a ⇒ 'a dlist ⇒ 'a dlist. The definition is the following:

insert ≡ (id ↦ list_of_dlist ↦ Abs_dlist) List.insert

This slightly cryptic definition uses a map operator ↦ for the function type[8]. (We will explain the reason for this odd style of definition shortly.) We can get a more familiar-looking definition by unfolding the definitions of ↦, ∘, and id:

insert $x\ y$ ≡ Abs_dlist (List.insert x (list_of_dlist y))

One can see that the definition of insert is a straightforward composition of the abstraction and representation functions Abs_dlist and list_of_dlist.

```
lemma indentity_quotient: "Quotient (op =) id id (op =)"

lemma fun_quotient:
  assumes "Quotient R₁ abs₁ rep₁ T₁"
  assumes "Quotient R₂ abs₂ rep₂ T₂"
  shows "Quotient (R₁ ⤇ R₂) (rep₁ ↦ abs₂) (abs₁ ↦ rep₂) (T₁ ⤇ T₂)"

lemma Quotient_list [quot_map]:
  assumes "Quotient R Abs Rep T"
  shows "Quotient (list_all2 R) (map Abs) (map Rep) (list_all2 T)"

lemma Quotient_dlist:
  Quotient (invariant distinct) Abs_dlist list_of_dlist cr_dlist
```

Fig. 4. Rules for building compound Quotient theorems

How lifting works: Compound quotient theorems We now make a small detour and sketch how lifting is implemented. The process of lifting is based on generating a *compound quotient theorem* that relates the type of the raw term with the type of the new abstract function. With our insert example, we need a quotient between types 'a ⇒ 'a list ⇒ 'a list and 'a ⇒ 'a dlist ⇒ 'a dlist.

[7] The only case when invariant is visible is if a composition of two quotients is used; see Section 3.3.

[8] $(f \mapsto g)\ h \equiv g \circ h \circ f$

We prove compound quotient theorems using a syntax-driven procedure that recursively traverses the structures of the raw type and the abstract type. Quotient theorems are built using the rules in Figure 4. With a bottom-up traversal, we start with theorems Quotient_dlist (obtained from the **setup_lifting** command) and indentity_quotient, which relate the the basic types. For each function type, we combine quotient theorems using rule fun_quotient. (Support for the function type is pre-configured, but a user may provide additional quotient rules for other type constructors and register them with the quot_map attribute, like Quotient_list in Figure 4.) The end result is the following theorem:

$$\begin{array}{l} \mathsf{Quotient} \\ \quad (op = \Mapsto \mathsf{invariant\ distinct} \Mapsto \mathsf{invariant\ distinct}) \\ \quad (\mathsf{id} \mapsto \mathsf{list\_of\_dlist} \mapsto \mathsf{Abs\_dlist}) \\ \quad (\mathsf{id} \mapsto \mathsf{Abs\_dlist} \mapsto \mathsf{list\_of\_dlist}) \\ \quad (op = \Mapsto \mathsf{cr\_dlist} \Mapsto \mathsf{cr\_dlist}) \end{array} \tag{21}$$

The Quotient theorem in Eq. (21) is central to the lifting of insert. The first argument of the Quotient predicate is the relation used in the respectfulness theorem for insert, and the second argument is precisely the abstraction function used in the definition theorem. The last argument will be used to derive the transfer rule, as we will see shortly.

Transfer rule The **lift_definition** command generates a transfer rule for every newly defined constant. This rule is declared to the Transfer package to enable transferring propositions involving the new constant. The transfer rule for insert is as follows:

$$(op = \Mapsto \mathsf{cr\_dlist} \Mapsto \mathsf{cr\_dlist})\ \mathsf{List.insert\ insert}$$

The transfer rule is derived using the following theorem:

lemma Quotient_to_transfer:
  assumes "Quotient R Abs Rep T" **and** "$R\ c\ c$" **and** "$c' \equiv Abs\ c$"
  shows "$T\ c\ c'$"

The first assumption is instantiated by the compound quotient theorem for insert (see Eq. (21)), the second one by the respectfulness theorem, and the last one by the definition theorem. The derivation of the transfer rule is one of the main reasons why we added the fourth parameter T to the Quotient predicate: With the four-parameter predicate, the transfer relation is built automatically (using $\Mapsto$) as a side effect of the lifting procedure.

Abstraction and representation function equations Besides the definition theorem, two other important user-relevant equations are also generated. The former is called the *abstraction function equation* (abs_eq for short) and relates the lifted function and the raw function using only abstraction functions. For our insert it is the following equation:

$\mathsf{distinct}\ y \Longrightarrow$
$\quad \mathsf{insert}\ x\ (\mathsf{Abs\_dlist}\ y) = \mathsf{Abs\_dlist}\ (\mathsf{List.insert}\ x\ y)$

The latter is called the *representation function equation* (rep_eq for short) and relates the lifted function and the raw function using only representation functions. For insert:

$\mathsf{list\_of\_dlist}\ (\mathsf{insert}\ x\ y) = \mathsf{List.insert}\ x\ (\mathsf{list\_of\_dlist}\ y)$

Observe that abs_eq has an additional assumption distinct y. It is because 'a dlist is a proper subtype, and thus the function Abs_dlist is not a totally-specified function. Therefore we need that extra assumption to make abs_eq meaningful. In general, abs_eq will always have extra assumptions in the case of a partial quotient or subtype. Conditional equations cannot be used as code equations in the code generator, but fortunately, we can still use rep_eq as a code equation.

Either equation could be used as a code equation in the case of a type copy. But we prefer abs_eq because usage of rep_eq depends on an extension of the code generator, which steps outside of the Isabelle logic in the generated code (and requires a subtle meta argument for correctness). The whole situtation with abs_eq, rep_eq, and code generation is summarized in Table 1.

3.3 Composition of quotients

Things get more complicated when we start lifting constants with more demanding types. The most interesting case is a composition of quotients when a type variable of a quotient type is instantiated again to a quotient type. A good example for 'a dlist is a concat function:

lift_definition concat :: "'a dlist dlist ⇒ 'a dlist" **is** "remdups ∘ List.concat"

Note that to construct a quotient between 'a list list and 'a dlist dlist, we must use a composition of quotients: first from 'a list list to 'a dlist list, and then to 'a dlist dlist. This explains the appearance of ∘ in the constant definition:

$\mathsf{concat} \equiv (\mathsf{map\ list\_of\_dlist} \circ \mathsf{list\_of\_dlist} \mapsto \mathsf{Abs\_dlist})\ (\mathsf{remdups} \circ \mathsf{List.concat})$

Relation composition ($\circ\circ$) also appears in the respectfulness theorem:

$\bigwedge list_1\ list_2.$
$(\mathsf{list\_all2\ cr\_dlist} \circ\circ \mathsf{invariant\ distinct} \circ\circ (\mathsf{list\_all2\ cr\_dlist})^{-1})\ list_1\ list_2$
$\Longrightarrow \mathsf{invariant\ distinct}\ ((\mathsf{remdups} \circ \mathsf{concat})\ list_1)\ ((\mathsf{remdups} \circ \mathsf{concat})\ list_2)$

We used a different relator for a quotient composition than it was used in [4]. Kaliszyk and Urban would use this respectfulness theorem:

$\bigwedge list_1\ list_2.\ (\mathsf{list\_all2}\ R \circ\circ R \circ\circ \mathsf{list\_all2}\ R)\ list_1\ list_2$
$\Longrightarrow \mathsf{invariant\ distinct}\ ((\mathsf{remdups} \circ \mathsf{concat})\ list_1)\ ((\mathsf{remdups} \circ \mathsf{concat})\ list_2)$

where $R = \mathsf{invariant\ distinct}$. Unfortunately that proof obligation is not provable; as they noticed, “unfortunately a general quotient theorem for $R_1 \circ\circ R_2 \circ\circ R_1$

would not be true in general". But if we choose a different relator $T_1 \circ\circ R_2 \circ\circ T_1{}^{-1}$, we do have a general quotient theorem for a quotient composition. We proved the following theorem in Isabelle:

lemma Quotient_compose:
  assumes "Quotient R_1 Abs_1 Rep_1 T_1"
  assumes "Quotient R_2 Abs_2 Rep_2 T_2"
  shows
  "Quotient $(T_1 \circ\circ R_2 \circ\circ T_1{}^{-1})$ $(Abs_2 \circ Abs_1)$ $(Rep_1 \circ Rep_2)$ $(T_1 \circ\circ T_2)$"

Because $R_1 \circ\circ R_2 \circ\circ R_1$ and $T_1 \circ\circ R_2 \circ\circ T_1{}^{-1}$ could be equivalent under certain conditions for total quotients, Kaliszyk and Urban were able to work with quotient compositions in the case of total quotients. Further discussion of this issue would be beyond the scope of this paper.

Other equations and theorems for concat are also accordingly complicated. For example, consider the transfer rule:

(list_all2 cr_dlist $\circ\circ$ cr_dlist $\Rrightarrow$ cr_dlist) (remdups $\circ$ List.concat) concat

With the presence of the composition operator ($\circ\circ$) this rule will not work very well with the Transfer package. However, it is easy to define a parameterized transfer relation as presented in Section 2.1.

definition "cr_dlist′ $A \equiv$ list_all2 A $\circ\circ$ cr_dlist"

We can then rephrase the transfer rule in terms of the new relation:

(cr_dlist′ (cr_dlist′ $(op =)$) $\Rrightarrow$ cr_dlist′ $(op =)$) (remdups $\circ$ List.concat) concat

In this form, the transfer rule will already work in many situations. The next step would be to combine this with a parametricity property of List.concat to get a more general rule (where $=$ is replaced by a variable A). We have not automated this process yet, but it is a possible direction for future work.

4 Conclusions and Future Work

We have presented a new design for automation of quotients in Isabelle/HOL, consisting of cleanly separated components that interact through well-defined interfaces: The Transfer package is configured using transfer rules, and the Lifting package is configured using quotient theorems.

The modular design yields flexibility—having well-defined interfaces makes it possible to connect the packages together in non-standard ways. In particular, the Lifting package is not limited to types defined by **quotient_type**; types defined by other means can be configured using a quotient rule. It would also be feasible to implement new type definition commands on top of the Lifting package, e.g. a command for defining a type from a normalization function.

Similarly, Transfer is not limited to constants defined by **lift_definition**; using easy-to-state transfer rules, one may configure constants defined by other means. Neither is the Transfer package limited to transfer relations produced by

quotient_type or **setup_lifting**. Transfer rules do not refer to the Quotient predicate, and the Transfer package imposes no restrictions on transfer relations; users may therefore define and use transfer relations between any two types.

The Lifting and Transfer packages are still in development; some features (such as abs_eq) were completed after the Isabelle 2012 release, and other planned features have yet to be implemented. In particular, the Quotient package [4] supports some features that we have not yet added to the Transfer package:

- *regularization* phase for descending method: Transfer currently has limited support for proving implications (Section 2.3), handling only equality and quantifiers. The Quotient package's *regularization* phase handles more complex propositions, including case analysis rules.
- lifting method: The Quotient package can solve goals about quotient types using given theorems about raw types.
- *lifted* theorem attribute: The Quotient package can automatically transform the given theorem from raw to abstract types.

Paulson's quotient library [6] provides a desirable feature that the Quotient package does not have: When defining a partial quotient, the domain of the partial equivalence relation is explicit. For example, since 2010 type real is defined as $\{x.\ \mathsf{cauchy}\ x\}$ // realrel, which specifies a partial quotient of nat $\Rightarrow$ rat. On the other hand, the domain is implicit with the **quotient_type** command:

quotient_type real = "nat $\Rightarrow$ rat" / (*partial*) realrel

All abs_eq rules generated by the Lifting package for type real have assumptions like realrel $x\ x$, but it would be more useful to have cauchy x instead. We envision a new partial quotient definition command that lets users specify a domain predicate like cauchy and a proof that realrel $x\ x \longleftrightarrow$ cauchy x. The Lifting package could then generate conditional rules with the preferred predicate.

In the post–Isabelle 2012 development version, we have converted the numeric types int, rat, and real to use Lifting and Transfer. (Previously they were constructed as quotients with typedef, in the style of Paulson [6].) The conversion yielded significant code savings: Int.thy went from 1770 lines to 1665 (-105), Rat.thy from 1262 to 1210 (-52), and RealDef.thy from 1808 to 1703 (-105). Automatically-generated abs_eq lemmas accounted for a significant portion of the savings. We can conclude that the packages are effective at reducing the amount of effort and boilerplate needed to define quotient types.

References

1. Florian Haftmann and Tobias Nipkow. Code Generation via Higher-Order Rewrite Systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming: 10th International Symposium: FLOPS 2010*, volume 6009 of *Lecture Notes in Computer Science*. Springer, 2010.
2. John Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

3. Peter V. Homeier. A Design Structure for Higher Order Quotients. In *Proc. of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *LNCS*, pages 130–146, 2005.
4. Cezary Kaliszyk and Christian Urban. Quotients revisited for Isabelle/HOL. In William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proc. of the 26th ACM Symposium on Applied Computing (SAC'11)*, pages 1639–1644. ACM, 2011.
5. Alexander Krauss. Simplifying Automated Data Refinement via Quotients. Technical report, Technische Universität München, July 2011. `http://www21.in.tum.de/~krauss/papers/refinement.pdf`.
6. Lawrence C. Paulson. Defining functions on equivalence classes. *ACM Trans. Comput. Logic*, 7(4):658–675, October 2006.
7. John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, pages 513–523, 1983.
8. Oscar Slotosch. Higher Order Quotients and their Implementation in Isabelle/HOL. In Elsa L. Gunter and Amy P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference (TPHOLs'97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 291–306, Murray Hill, NJ, USA, August 1997. Springer.
9. Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.