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

Brian Huffman[1] and Ondřej Kunčar[2]

[1] Galois, Inc.
[2] Technische Universität München

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 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.

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 the properties of raw functions to the abstract level. Clearly, this process needs good proof automation.

1.1 Related Work

Much previous work has been done on formalizing quotients in theorem provers. Slotosch [12] and Paulson [10] 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 [3] implemented tools for lifting constants and transferring theorems automatically, although this work was still limited to first-order constants and theorems. In 2005, Homeier [4] 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 [5], based upon Homeier's design. It was first released with Isabelle 2009-2. The Quotient package is designed around the notion of a *quotient*, which involves two types and three constants: a raw type 'a with a partial equivalence relation R :: 'a ⇒ 'a ⇒ bool and 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.

The Quotient package implements a collection of commands, proof methods, and theorem attributes. Given a raw type and a (total or partial) equivalence relation R, the **quotient_type** command defines a new type with *Abs* and *Rep* that form a quotient. Given a function g on the raw type and an abstract type, the **quotient_definition** command defines a new abstract function g' in terms of g, *Abs*, and *Rep*. The user must provide a *respectfulness theorem* showing that g respects R. Finally the descending and lifting methods can transfer propositions between g and g'. Internally, this uses respectfulness theorems, the definition of g', and the quotient properties of R, *Abs* and *Rep*.

Lammich's automatic procedure for data refinement [7] was directly inspired by our packages, especially by the idea to represent types as relations.

In Coq, implementations of generalized rewriting by Coen [1] and Sozeau [13] are similar to our Transfer method—in particular, Sozeau's "signatures" for higher-order functions are like our transfer rules. Sozeau's work has better support for subrelations, but our Transfer package is more general in allowing relations over two different types.

Magaud [8] transfers Coq theorems between different types, but unlike our work, his approach is based on transforming proof terms.

1.2 Limitations of the Quotient Package

We decided to redesign the Quotient package after identifying several limitations of its implementation. A few such limitations were described by Krauss [6]: 1.) The quotient relation R and raw function f must be dedicated constants, not arbitrary terms. Thus the tool cannot be used on locale parameters and some definitions in a local theory. 2.) One cannot turn a pre-existing type into a quotient afterwards; nor can one declare a user-defined constant on the quotient type as the lifted version of another constant.

To solve problem 1 does not require major organizational changes. However, problem 2 has deeper roots and suggested splitting the Quotient package into various layers: 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. 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 theorems to fset:

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

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 without 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 Fig. 1. Each component depends only on the components underneath it. At the bottom is the Transfer package, which transfers propositions between raw and abstract types (§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 (§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**. We expect that additional type definition commands might be implemented later. We conclude with the contribution and results of our packages (§4).

Our work was released in Isabelle 2013-1.

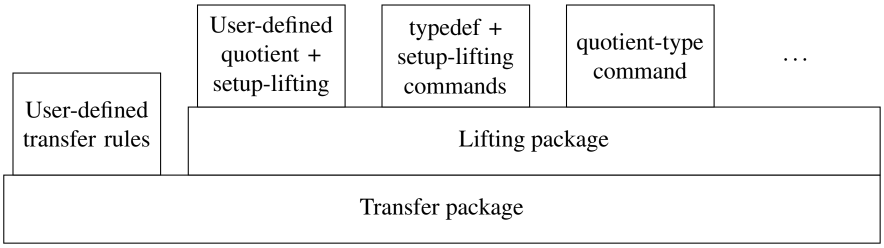

User-defined quotient + setup-lifting
typedef + setup-lifting commands
quotient-type command
...
User-defined transfer rules
Lifting package
Transfer package

Fig. 1. Modular design of packages for formalizing quotients

2 Transfer Package

The primary function of the Transfer package is to transfer theorems from one type to another, by proving equivalences between pairs of related propositions. This process is guided by an extensible collection of *transfer rules*, which establish connections between pairs of related types or constants.

The Transfer package provides multiple user interfaces: The transfer proof method replaces the current subgoal by a logically equivalent subgoal—typically, it replaces a goal about an abstract type by a goal about the raw type. The package also provides the transferred theorem attribute, which yields a theorem about an abstract type when given a theorem involving a raw type.

2.1 Types as Relations

The design of the Transfer package is based on the idea of types as binary relations. The notions of *relational parametricity* by Reynolds [11], *free theorems* by Wadler [14], and *representation independence* by Mitchell [9] were primary sources of inspiration.

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. Relators lift relations over type constructors: Related data structures have the same shape, with pointwise-related elements, and related functions map related input to related output (see Fig. 2). For base types like bool or int we use identity relations ($\longleftrightarrow$ or $=$).

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

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

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.) Parametricity rules in the style of Eq. (2) can serve as transfer rules, relating two different type instances of the same polymorphic function.

Representation independence is one useful application of relational parametricity. Mitchell [9] used it to reason about data abstraction in functional programming. Imagine we have an interface to an abstract datatype (e.g. queues) with two different implementations. We would hope for any queue-using program to behave identically no

$$(\mathsf{prod\_rel}\ A\ B)\ x\ y \equiv A\ (\mathsf{fst}\ x)\ (\mathsf{fst}\ y) \wedge B\ (\mathsf{snd}\ x)\ (\mathsf{snd}\ y)$$
$$(A \mapsto B)\ f\ g \equiv (\forall x\ y.\ A\ x\ y \longrightarrow B\ (f\ x)\ (g\ y))$$
$$(\mathsf{set\_rel}\ A)\ X\ Y \equiv (\forall x \in X.\ \exists y \in Y.\ A\ x\ y) \wedge (\forall y \in Y.\ \exists x \in X.\ A\ x\ y)$$
$$(\mathsf{list\_all2}\ A)\ xs\ ys \equiv \mathsf{length}\ xs = \mathsf{length}\ ys \wedge (\forall (x, y) \in \mathsf{set}\ (\mathsf{zip}\ xs\ ys).\ A\ x\ y)$$

Fig. 2. Relators for various type constructors

matter which queue implementation is used—i.e., that the two queue implementations are *contextually equivalent*. Representation independence implies that this is so, as long as we can find a relation between the two implementation types that is preserved by all the corresponding operations. In our work, we refer to such a relation as a *transfer relation*.

The Transfer package is essentially a working implementation of the idea of representation independence, but in a slightly different setting: Instead of a typical functional programming language, we use higher-order logic; and instead of showing contextual equivalence of programs, we show logical equivalence of propositions.

Example: Int/nat transfer. We consider a simple use case, transferring propositions between the integers and natural numbers. We can think of type int as a concrete representation of the more abstract type nat; each type has its own implementation of numerals, arithmetic operations, comparisons, and so on. To specify the connection between the two types, we define a transfer relation $\mathsf{ZN} :: \mathsf{int} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{bool}$.

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

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. The respective addition operators map related arguments 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.

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

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

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

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

The Transfer package can use the rules above to derive equivalences like the following.

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

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 $\forall x{::}\mathsf{int} \in \{0..\}.\ x < x + 1$. The transferred attribute works in the opposite direction: Given the theorem $\forall x{::}\mathsf{int} \in \{0..\}.\ x < x + 1$, it would prove the same equivalence, and return the theorem $\forall n{::}\mathsf{nat}.\ n < n + 1$. In general, the Transfer package can handle any lambda term constructed from constants for which it has transfer rules.

2.2 Transfer Algorithm

The core functionality of the Transfer package is to prove equivalence theorems in the style of Eq. (8). To derive an equivalence theorem, the Transfer package uses transfer rules for constants, along with elimination and introduction rules for $\Mapsto$.

$$\frac{(A \Mapsto B)\ f\ g \qquad A\ x\ y}{B\ (f\ x)\ (g\ y)}\ (\Mapsto\text{-}\textsc{Elim}) \qquad\qquad \frac{\forall x\ y.\ A\ x\ y \longrightarrow B\ (f\ x)\ (g\ y)}{(A \Mapsto B)\ f\ g}\ (\Mapsto\text{-}\textsc{Intro})$$

Alternatively, these rules can be restated in the form of structural typing rules, similar to those for the simply typed lambda calculus. A typing judgment here involves two terms instead of one, and a binary relation takes the place of a type. The environment Γ collects the local assumptions for bound variables.

$$\textsc{App}\ \frac{\Gamma \vdash (A \Mapsto B)\ f\ g \qquad \Gamma \vdash A\ x\ y}{\Gamma \vdash B\ (f\ x)\ (g\ y)} \qquad \textsc{Abs}\ \frac{\Gamma, A\ x\ y \vdash B\ (f\ x)\ (g\ y)}{\Gamma \vdash (A \Mapsto B)\ (\lambda x.\ f\ x)\ (\lambda y.\ g\ y)} \qquad \textsc{Var}\ \frac{A\ x\ y \in \Gamma}{\Gamma \vdash A\ x\ y}$$

To transfer a theorem requires us to build a derivation tree using these rules, with transfer rules for constants at the leaves of the tree. For the transfer method, we are given only the abstract right-hand side; for the transferred attribute, only the left-hand side. The job of the Transfer package is to fill in the remainder of the tree—essentially a type inference problem.

Our implementation splits the process into two steps. Step one is to determine the overall shape of the derivation tree: the arrangement of App, Abs, and Var nodes, and the pattern of unknown term and relation variables. Step two is then to fill in the leaves of the tree using the collection of transfer rules, at the same time instantiating the unknown variables.

Step one starts by building a "skeleton" s of the known term t—a lambda term with the same structure, but with constants replaced by fresh variables. Using Isabelle's standard type inference algorithm, we annotate s with types; the inferred types determine the pattern of relation variables in the derivation tree. For step two, we set up a schematic proof state with one goal for each leaf of the tree, and then match transfer rules with subgoals. We use backtracking search in case multiple transfer rules match a given left- or right-hand side.

As an example, we will transfer the proposition $\forall n{::}\mathsf{nat}.\ n \le n$. This is actually syntax for $\mathsf{All}\ (\lambda n{::}\mathsf{nat}.\ \mathsf{le}\ n\ n)$, so its skeleton has the form $t\ (\lambda x.\ u\ x\ x)$. Type inference yields a most general typing with $t :: (\text{'a} \Rightarrow \text{'b}) \Rightarrow \text{'c}$ and $u :: \text{'a} \Rightarrow \text{'a} \Rightarrow \text{'b}$, where 'a, 'b, and 'c are fresh type variables. We generate fresh relation variables $?a$, $?b$, and $?c$ corresponding to these, and use them to build an initial derivation tree following the skeleton's structure and inferred types:

$$\frac{\vdash ((?a \Mapsto ?b) \Mapsto ?c)\ ?t\ \mathsf{All} \qquad \dfrac{\dfrac{\dfrac{?a\ x\ n \vdash (?a \Mapsto ?a \Mapsto ?b)\ ?u\ \mathsf{le} \qquad \dfrac{}{?a\ x\ n \vdash ?a\ x\ n}}{?a\ x\ n \vdash (?a \Mapsto ?b)\ (?u\ x)\ (\mathsf{le}\ n)} \qquad \dfrac{}{?a\ x\ n \vdash ?a\ x\ n}}{?a\ x\ n \vdash ?b\ (?u\ x\ x)\ (\mathsf{le}\ n\ n)}}{\vdash (?a \Mapsto ?b)\ (\lambda x.\ ?u\ x\ x)\ (\lambda n.\ \mathsf{le}\ n\ n)}}{\vdash ?c\ (?t\ (\lambda x.\ ?u\ x\ x))\ (\mathsf{All}\ (\lambda n.\ \mathsf{le}\ n\ n))}$$

Note that the leaves with $?a\ x\ n$ are solved with rule Var, but the leaves with constants All and le are as yet unsolved. Therefore this derivation tree yields a theorem with two hypotheses, $((?a \Mapsto ?b) \Mapsto ?c)\ ?t\ \mathsf{All}$ and $(?a \Mapsto ?a \Mapsto ?b)\ ?u\ \mathsf{le}$, and a conclusion $?c\ (?t\ (\lambda x.\ ?u\ x\ x))\ (\mathsf{All}\ (\lambda n.\ \mathsf{le}\ n\ n))$. In step two, we set up a proof state with the hypotheses as subgoals. The first goal is matched by Eq. (7), and the second goal by $(\mathsf{ZN} \Mapsto \mathsf{ZN} \Mapsto op \longleftrightarrow)\ (op \le)\ (op \le)$. Similarly instantiating the schematic variables in the conclusion yields the final equivalence theorem:

$$(\forall x{::}\mathsf{int} \in \{0..\}.\ x \le x) \longleftrightarrow (\forall n{::}\mathsf{nat}.\ n \le n) \tag{9}$$

2.3 Parameterized Transfer Relations

The design of the Transfer package generalizes easily to transfer relations with parameters. As an example, we define a transfer 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.

$$\mathsf{LF} :: (\mathsf{'a_1} \Rightarrow \mathsf{'a_2} \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{'a_1\ list} \Rightarrow \mathsf{'a_2\ fset} \Rightarrow \mathsf{bool} \tag{10}$$

$$(\mathsf{LF}\ A)\ xs\ Y \equiv \exists ys.\ \mathsf{list\_all2}\ A\ xs\ ys \wedge \mathsf{Fset}\ ys = Y \tag{11}$$

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.

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

$$(\mathsf{LF}\ (\mathsf{LF}\ A) \Mapsto \mathsf{LF}\ A)\ \mathsf{concat}\ \mathsf{concat\_fset} \tag{13}$$

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

2.4 Transfer Rules with Side Conditions

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 $(A \Mapsto A \Mapsto op \longleftrightarrow)\ (op =)\ (op =)$, but this does not hold for all relations A—it only holds if A is *bi-unique*, i.e., single-valued and injective.

$$\mathsf{bi\_unique}\ A \Longrightarrow (A \Mapsto A \Mapsto op \longleftrightarrow)\ (op =)\ (op =) \tag{14}$$

As pointed out by Wadler [14], 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, for example: $\mathsf{bi\_unique\ ZN}$, $\mathsf{bi\_unique}\ A \Longrightarrow \mathsf{bi\_unique}\ (\mathsf{set\_rel}\ A)$ and $\mathsf{bi\_unique}\ A \Longrightarrow \mathsf{bi\_unique}\ (\mathsf{list\_all2}\ A)$.

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 $\mathsf{list\_all2\ ZN}$. 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.

$$\mathsf{bi\_total}\ A \Longrightarrow ((A \Mapsto op \longleftrightarrow) \Mapsto op \longleftrightarrow)\ \mathsf{All}\ \mathsf{All} \tag{15}$$

Universal quantifiers appear in most propositions used with transfer; however, many transfer relations (including ZN) are not bi-total, but only *right-total*, i.e., surjective. The following transfer rule can then be used if no other specialized rule is provided:

$$\mathsf{right\_total}\ A \Longrightarrow ((A \Mapsto op \longleftrightarrow) \Mapsto op \longleftrightarrow)\ (\mathsf{Ball}\ \{x.\ \mathsf{Domainp}\ A\ x\})\ \mathsf{All} \tag{16}$$

The predicate $\mathsf{Domainp}$ is defined as $\mathsf{Domainp}\ T\ x \equiv \exists y.\ T\ x\ y$. Because it is awkward to work with expressions like $\mathsf{Domainp}\ T$ in the transferred goal, we implemented a post-processing step that can replace $\mathsf{Domainp}$ expressions with equivalent but more convenient predicates. This is configured by registering a *transfer domain rule*: $\mathsf{Domainp}\ \mathsf{ZN} = (\lambda x.\ x \geq 0)$. We provide transfer domain rules for lists and other types; thus we can replace, for example, $\mathsf{Domainp}\ (\mathsf{list\_all2}\ \mathsf{ZN})$ by $\mathsf{list\_all}\ (\lambda x.\ x \geq 0)$. The use of $\mathsf{Domainp}$ is not limited to quantifiers—the usual parametricity rules for constants like UNIV, $\mathsf{Collect}$, and set intersection $\bigcap$ require bi-totality, but we also provide more widely applicable transfer rules using $\mathsf{Domainp}$.

The last condition we can use to restrict relations is being *right-unique*, i.e., single-valued. Bi-totality, right-totality and right-uniqueness are like bi-uniqueness preserved by many relators, including those for lists and sets. We mentioned that ZN is not bi-total but, e.g, total quotients yield bi-total transfer relations; see the overview in Tab. 1.

Handling equality relations. Many propositions contain non-polymorphic constants that remain unchanged by the transfer procedure, e.g., boolean operations. We would like to avoid the necessity for lots of trivial transfer rules like the rule for the boolean conjunction: $(op \longleftrightarrow \Mapsto op \longleftrightarrow \Mapsto op \longleftrightarrow)\ (op \wedge)\ (op \wedge)$. Instead we define a predicate $\mathsf{is\_equality}\ A$, which holds if and only if A is the equality relation on its type, and register a single reflexivity transfer rule $\mathsf{is\_equality}\ A \Longrightarrow A\ x\ x$. The $\mathsf{is\_equality}$ predicate is preserved by all of the standard relators, including lists, sets, pairs, and function space.

2.5 Proving Implications Instead of Equivalences

The $\mathsf{transfer}$ proof method can replace a universal with an equivalent bounded quantifier: e.g., $(\forall n{::}\mathsf{nat}.\ n < n + 1)$ is transferred to $(\forall x{::}\mathsf{int} \in \{0..\}.\ x < x + 1)$. This yields a useful extra assumption in the new subgoal. With the $\mathsf{transferred}$ attribute, however, it may be preferable to start with a stronger theorem $(\forall x{::}\mathsf{int}.\ x < x + 1)$, without the bounded quantifier. In this case, the Transfer package can prove an implication:

$$(\forall x{::}\mathsf{int}.\ x < x + 1) \longrightarrow (\forall n{::}\mathsf{nat}.\ n < n + 1) \tag{17}$$

The Transfer algorithm works exactly the same; we just need some new transfer rules that encode monotonicity. We provide rules for quantifiers and implication, using various combinations of $\longrightarrow$, $\longleftarrow$, and $\longleftrightarrow$; a few are shown here.

$$\mathsf{right\_total}\ A \Longrightarrow ((A \Mapsto op \longrightarrow) \Mapsto op \longrightarrow)\ \mathsf{All}\ \mathsf{All} \tag{18}$$

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

$$(op \longleftarrow \Mapsto op \longrightarrow \Mapsto op \longrightarrow)\ (op \longrightarrow)\ (op \longrightarrow) \tag{20}$$

The derivation of Eq. (17) uses transfer rule (19); rule (18) comes into play when quantifiers are nested. These rules are applicable to relation ZN because it is right-total. Further variants of these rules (involving reverse implication) are used to transfer induction and case analysis rules, which have many nested implications and quantifiers.

Having many different transfer rules for the same constants would tend to introduce a large amount of backtracking search in step two of the transfer algorithm. To counter this, we pre-instantiate some of the relation variables to $\longrightarrow$, $\longleftarrow$, or $\longleftrightarrow$, guided by a simple monotonicity analysis.

3 Lifting Package

The Lifting package allows users to lift terms of the raw type to the abstract type, which is a necessary step in building a library for an abstract type. Lifting defines a new constant by combining coercion functions (Abs and Rep) with the raw term. It also proves an appropriate transfer rule for the Transfer package and, if possible, an equation for the code generator. Doing this lifting manually is mostly tedious and uninteresting; our goal is to automate as much as possible, so users can focus on the interesting bits.

The Lifting package provides two commands: **setup_lifting** for initializing the package to work with a new type, and **lift_definition** for lifting constants. The Lifting package works with four kinds of type abstraction: type copies, subtypes, total quotients and partial quotients. See Tab. 1 for an overview of these.

Example: finite sets. Let us define a type of finite sets as a quotient of lists, where two lists are in the same equivalence class if they represent the same set:

$$\textbf{quotient\_type}\ \text{'a fset} = \text{'a list} \ / \ (\lambda xs\ ys.\ \text{set}\ xs = \text{set}\ ys)$$

Now we can define the union of two finite sets as a lifted function of concatenation of two lists append :: 'a list ⇒ 'a list ⇒ 'a list, which has infix syntax _ @ _:

$$\textbf{lift\_definition}\ \text{funion} :: \text{'a fset} \Rightarrow \text{'a fset} \Rightarrow \text{'a fset"}\ \textbf{is}\ \text{append}$$

The command opens a proof environment with the following obligation:

$$\bigwedge l_1\ l_2\ l_3\ l_4.\ \text{set}\ l_1 = \text{set}\ l_2 \Longrightarrow \text{set}\ l_3 = \text{set}\ l_4 \Longrightarrow \text{set}\ (l_1\ @\ l_3) = \text{set}\ (l_2\ @\ l_4)$$

The obligation is called a *respectfulness theorem* and says that append respects the equivalence relation that defines 'a fset. When the user proves the obligation, the new function funion is defined as follows:

$$\text{funion}\ A\ B \equiv \text{abs\_fset}\ ((\text{rep\_fset}\ A)\ @\ (\text{rep\_fset}\ B))$$

The package also generates a code equation for the code generator:

$$\text{funion}\ (\text{abs\_fset}\ A)\ (\text{abs\_fset}\ B) = \text{abs\_fset}\ (A\ @\ B)$$

And finally, because the package proved internally a corresponding transfer rule, we can prove, e.g., that funion commutes: **lemma** funion A B = funion B A. If we apply the method transfer, we get $\bigwedge A\ B.\ \text{set}\ (A\ @\ B) = \text{set}\ (B\ @\ A)$, which is easily provable.

If we defined 'a fset by **typedef** 'a fset = $\{A :: \text{'a set. finite}\ A\}$, i.e., as a subtype of sets, and funion as a lifted function of the set union ∪, we would get the proof obligation $\bigwedge s_1\ s_2.\ \text{finite}\ s_1 \Longrightarrow \text{finite}\ s_2 \Longrightarrow \text{finite}\ (s_1 \cup s_2)$ and the code equation $\text{rep\_fset}\ (\text{funion}\ A\ B) = \text{rep\_fset}\ A \cup \text{rep\_fset}\ B$.[1]

3.1 General Case

We abstract from the presented example now and give a description that covers the general case of what the Lifting package does. The input of the lifting is a term $t :: \tau_1$ on

[1] See §3.3 for more about what the code equations are and how they are derived.

the concrete level, an abstract type τ_2 and a name f of the new constant. In our example, $t =$ append, $\tau_1 =$ 'a list $\Rightarrow$ 'a list $\Rightarrow$ 'a list, $\tau_2 =$ 'a fset $\Rightarrow$ 'a fset $\Rightarrow$ 'a fset and $f =$ funion. We work generally with types τ which are composed from type constructors κ and other types $\overline{\vartheta}$. Then we write $\tau = \overline{\vartheta}\ \kappa$. Each type parameter of κ can be either co-variant (we write $+$) or contra-variant ($-$). E.g., in the function type $\alpha \rightarrow \beta$, α is contra-variant whereas β is co-variant.

In this section, we define three functions Morph^p, Relat and Trans. Morph^p is a combination of abstraction and representation functions and gives us the definition of f. The polarity superscript p ($+$ or $-$) encodes if an abstraction or a representation function should be generated. Relat is a combination of equivalence relations and allows us to describe that t behaves correctly—respects the equivalence classes. Finally, Trans is a composed transfer relation and describes how t and f are related. More formally, if the user proves the respectfulness theorem $\mathrm{Relat}(\tau_1,\tau_2)\ t\ t$, the Lifting package will define the new constant f as $f = \mathrm{Morph}^+(\tau_1,\tau_2)\ t$ and proves the transfer rule $\mathrm{Trans}(\tau_1,\tau_2)\ t\ f$.

For now we will not distinguish between quotients, subtypes, etc. Instead we unify all four kinds of type abstraction with a general notion of an abstract type.

Definition 1. *We say that κ_2 is an* abstract type *of κ_1 if there is a* transfer relation $T_{\kappa_1,\kappa_2} :: (\overline{\vartheta})\ \kappa_1 \rightarrow (\overline{\alpha})\ \kappa_2 \rightarrow \mathrm{bool}$ *associated with κ_1 and κ_2 (see also Fig. 3a) such that*

1. *T_{κ_1,κ_2} is right-total and right-unique,*
2. *all type variables in $\overline{\vartheta}$ are in $\overline{\alpha}$, which contains only distinct type variables.*

We say that $\tau_2 = (\overline{\rho})\ \kappa_2$ is an instance of an abstract type *of $\tau_1 = (\overline{\sigma})\ \kappa_1$ if*

1. *κ_2 is an abstract type of κ_1 certified by $T_{\kappa_1,\kappa_2} :: (\overline{\vartheta})\ \kappa_1 \rightarrow (\overline{\alpha})\ \kappa_2 \rightarrow \mathrm{bool}$,*
2. *$\overline{\sigma} = \theta\ \overline{\vartheta}$, where $\theta = \mathrm{match}(\overline{\rho},\overline{\alpha})$* [2].

In our finite sets example, the **quotient_type** command internally generates a transfer relation between the concrete type 'a list and the abstract type 'a fset. In principle, such a transfer relation alone is sufficient to characterize all four kinds of type abstraction: type copies, subtypes, total and partial quotients. We could build compound transfer relations for compound types and get other components (e.g., the morphisms Morph^p for the definition) from this relation using the choice operator. But it turns out that it is useful to have these other components explicitly, e.g. for generating code equations. The other components that we can derive from each transfer relation T_{κ_1,κ_2} and associate with each abstract type are ($\circ\circ$ is the relation composition):

– Partial equivalence relation R_{κ_1,κ_2} (Fig. 3b), specified by $R_{\kappa_1,\kappa_2} = T_{\kappa_1,\kappa_2} \circ\circ T^{-1}_{\kappa_1,\kappa_2}$.
– Abstraction function Abs_{κ_1,κ_2} (Fig. 3c), specified by $T_{\kappa_1,\kappa_2}\ a\ b \longrightarrow Abs_{\kappa_1,\kappa_2}\ a = b$.
– Representation function Rep_{κ_1,κ_2} (Fig. 3d), specified by $T_{\kappa_1,\kappa_2}\ (Rep_{\kappa_1,\kappa_2}\ a)\ a$.

Since T_{κ_1,κ_2} is right-total and right-unique, there always exist some Abs and Rep functions that meet the above given specification. On the other hand, given R, Abs and Rep, the specification allows only right-total and right-unique T.

The reflexive part of the partial equivalence relation R_{κ_1,κ_2} implicitly specifies which values of the concrete type are used for the construction of the abstract type.[3] The

[2] Match is a usual matching algorithm, i.e., $\mathrm{match}(\overline{\beta},\overline{\alpha})$ yields a substitution θ such that $\beta = \theta\ \alpha$.
[3] We omitted reflexive edges of R_{κ_1,κ_2} in Fig. 3b.

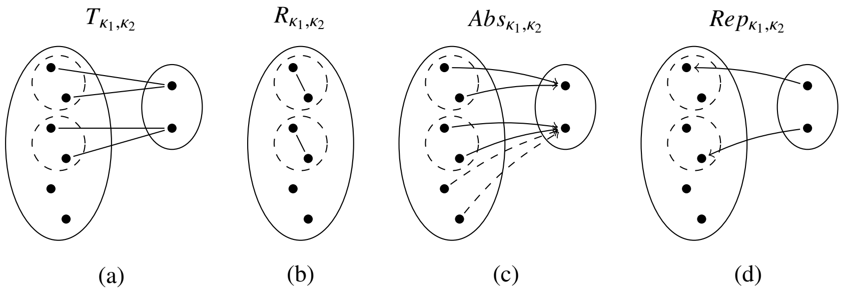


Fig. 3. Components of an abstract type

representation and abstraction functions map abstract values to concrete values and vice versa. Abs_{κ_1,κ_2} is underspecified outside of a range of T_{κ_1,κ_2} (dashed lines in Fig. 3c) and Rep_{κ_1,κ_2} can select only one of the values in the corresponding class.

Now we come to the key definition of this section. We derived *R*, *Abs* and *Rep* only for transfer relations that are associated with a type constructor. But later on, we build compound transfer relations for general types. What are *R*, *Abs* and *Rep* in this case? Again any functions meeting the above given specification. The following quotient predicate captures this idea and bundles all the components together.

Definition 2. *We define a* quotient predicate *with the syntax* $\langle .,.,.,. \rangle$ *and we say that* $\langle R, Abs, Rep, T \rangle$ *if 1.* $R = T \circ\circ T^{-1}$, *2.* $T\ a\ b \longrightarrow Abs\ a = b$ *and 3.* $T\ (Rep\ a)\ a$.

The following definition requires that $\langle .,.,.,. \rangle$ is preserved by going through the type universe using map functions and relators.

Definition 3. *We say that* map_κ *is a map function for* κ *and* rel_κ *is a relator for* κ*, where* κ *has arity n, if the assumptions* $\langle R_1, m_1^+, m_1^-, T_1\rangle, \ldots, \langle R_n, m_n^+, m_n^-, T_n\rangle$ *implies* $\langle \text{rel}_\kappa R_1\ \ldots\ R_n, \text{map}_\kappa\ m^{p_\kappa^1}\ \ldots\ m^{p_\kappa^{2n}}, \text{map}_\kappa\ m^{-p_\kappa^1}\ \ldots\ m^{-p_\kappa^{2n}}, \text{rel}_\kappa T_1\ \ldots\ T_n\rangle$.

Indexes p_κ^i encode which arguments of the map function are co-variant (+) or contra-variant (−). The map function can in general take $2n$ arguments because each type parameter of κ can be co-variant and contra-variant at the same time, e.g., $\alpha\ \kappa \equiv \alpha \rightarrow \alpha$.

Now we finally define Morph^p, Relat and Trans, as we promised to be the main goal of this section. First, let us define auxiliary functions morph^p, relat and trans, which are going to be used as the single step in the main definition of Morph^p, Relat and Trans. Functions morph^p, relat and trans are defined for all types $\tau_1 = (\overline{\sigma})\ \kappa_1$ and $\tau_2 = (\overline{\rho})\ \kappa_2$, where τ_2 is an instance of an abstract type of τ_1, as follows:

– $\text{morph}^+(\tau_1,\tau_2) = Abs_{\kappa_1,\kappa_2} :: \tau_1 \rightarrow \tau_2$
– $\text{morph}^-(\tau_1,\tau_2) = Rep_{\kappa_1,\kappa_2} :: \tau_2 \rightarrow \tau_1$
– $\text{relat}(\tau_1,\tau_2) = R_{\kappa_1,\kappa_2} :: \tau_1 \rightarrow \tau_1 \rightarrow \text{bool}$
– $\text{trans}(\tau_1,\tau_2) = T_{\kappa_1,\kappa_2} :: \tau_1 \rightarrow \tau_2 \rightarrow \text{bool}$

Now we extend the simple step functions morph^p, relat and trans defined only for abstract types to functions Morph^p, Relat and Trans, which take general types τ_1 and τ_2, by doing induction and case split on the type structure:

- **Base case.** If $\tau_1 = \tau_2$, then

$$\begin{aligned}\mathrm{Morph}^p(\tau_1,\tau_2) &= \mathrm{id} :: \tau_1 \to \tau_1,\\ \mathrm{Relat}(\tau_1,\tau_2) &= op = :: \tau_1 \to \tau_1 \to \mathrm{bool},\\ \mathrm{Trans}(\tau_1,\tau_2) &= op = :: \tau_1 \to \tau_1 \to \mathrm{bool}.\end{aligned}$$

- **Non-abstract type case.** If $\tau_1 = (\overline{\sigma})\ \kappa$ and $\tau_2 = (\overline{\rho})\ \kappa$, then

$$\begin{aligned}\mathrm{Morph}^p(\tau_1,\tau_2) &= \mathrm{map}_\kappa \mathrm{Morph}^{p^1_\kappa p}(\sigma_1,\rho_1)\ \dots\ \mathrm{Morph}^{p^{2n}_\kappa p}(\sigma_n,\rho_n),\\ \mathrm{Relat}(\tau_1,\tau_2) &= \mathrm{rel}_\kappa \mathrm{Relat}(\sigma_1,\rho_1)\ \dots\ \mathrm{Relat}(\sigma_n,\rho_n),\\ \mathrm{Trans}(\tau_1,\tau_2) &= \mathrm{rel}_\kappa \mathrm{Trans}(\sigma_1,\rho_1)\ \dots\ \mathrm{Trans}(\sigma_n,\rho_n),\end{aligned}$$

  where map_κ is a map function for κ and $p^i_\kappa p$ is a usual multiplication of polarities: $+\cdot- = -\cdot+ = -$ and $+\cdot+ = -\cdot- = +$. The function rel_κ is a relator for type κ.
- **Abstract type case.** If $\tau_1 = (\overline{\sigma})\ \kappa_1$, $\tau_2 = (\overline{\rho})\ \kappa_2$, $\kappa_1 \neq \kappa_2$, and κ_2 is an abstract type of κ_1 certified by $T_{\kappa_1,\kappa_2} :: (\overline{\vartheta})\ \kappa_1 \to (\overline{\alpha})\ \kappa_2 \to \mathrm{bool}$, let us define $\overline{\sigma'} = \theta\,\overline{\vartheta}$, where $\theta = \mathrm{match}(\overline{\rho},\overline{\alpha})$. [4] Then we define these equations

$$\begin{aligned}\mathrm{Morph}^+(\tau_1,\tau_2) &= \mathrm{morph}^+((\overline{\sigma'})\ \kappa_1,\tau_2) \circ \mathrm{Morph}^+(\tau_1,(\overline{\sigma'})\ \kappa_1),\\ \mathrm{Morph}^-(\tau_1,\tau_2) &= \mathrm{Morph}^-(\tau_1,(\overline{\sigma'})\ \kappa_1) \circ \mathrm{morph}^-((\overline{\sigma'})\ \kappa_1,\tau_2),\\ \mathrm{Relat}(\tau_1,\tau_2) &= \mathrm{Trans}(\tau_1,(\overline{\sigma'})\ \kappa_1) \circ\circ \mathrm{relat}((\overline{\sigma'})\ \kappa_1,\tau_2) \circ\circ \mathrm{Trans}(\tau_1,(\overline{\sigma'})\ \kappa_1)^{-1},\\ \mathrm{Trans}(\tau_1,\tau_2) &= \mathrm{Trans}(\tau_1,(\overline{\sigma'})\ \kappa_1) \circ\circ \mathrm{trans}((\overline{\sigma'})\ \kappa_1,\tau_2).\end{aligned}$$

The functions Morph^p, Relat and Trans are undefined if κ_2 is not an abstract type for κ_1 in the abstract type case. In such a case the Lifting package reports an error. Let us assume for the rest that we work only with such τ_1 and τ_2 that this does not happen.

Theorem 1. Morph^p*,* Relat *and* Trans *have the following types:* $\mathrm{Morph}^+(\tau_1,\tau_2) :: \tau_1 \to \tau_2$*,* $\mathrm{Morph}^-(\tau_1,\tau_2) :: \tau_2 \to \tau_1$*,* $\mathrm{Relat}(\tau_1,\tau_2) :: \tau_1 \to \tau_1 \to \mathrm{bool}$ *and* $\mathrm{Trans}(\tau_1,\tau_2) :: \tau_1 \to \tau_2 \to \mathrm{bool}$.

Proof. By induction on defining equations of Morph^p, Relat and Trans. □

Thus in our context, where $t :: \tau_1$, the terms $\mathrm{Relat}(\tau_1,\tau_2)\ t\ t$, $f = \mathrm{Morph}^+(\tau_1,\tau_2)\ t$ and $\mathrm{Trans}(\tau_1,\tau_2)\ t\ f$ are well-typed terms and f has indeed type τ_2. The respectfulness theorem $\mathrm{Relat}(\tau_1,\tau_2)\ t\ t$ has to be proven by the user. The definitional theorem $f = \mathrm{Morph}^+(\tau_1,\tau_2)\ t$ is proven by Isabelle. The remaining question is how we get the transfer rule $\mathrm{Trans}(\tau_1,\tau_2)\ t\ f$. Two following theorems give us the desired transfer rule.

Theorem 2. *If* $\langle R, Abs, Rep, T\rangle$*,* $f = Abs\ t$ *and* $R\ t\ t$*, then* $T\ t\ f$*.*

Proof. Because $R = T \circ\circ T^{-1}$, and $R\ t\ t$, we have $\exists x.\ T\ t\ x$. Let us denote this x as g. Thus $Abs\ t = g$ follows from $T\ t\ g$. But from $f = Abs\ t$ we have $f = g$ and thus $T\ t\ f$. □

The following theorem is the key theorem of this section: it proves that our definitions of Morph^p, Relat and Trans are legal, i.e., they have the desired property that they still form a (compound) abstract type, i.e., they meet the quotient predicate.

[4] Definition 1 guarantees that all type variables in $\overline{\vartheta}$ are in $\overline{\alpha}$ and thus $\overline{\rho}$ uniquely determines $\overline{\sigma'}$.

Theorem 3. $\langle \mathrm{Relat}(\tau_1,\tau_2), \mathrm{Morph}^+(\tau_1,\tau_2), \mathrm{Morph}^-(\tau_1,\tau_2), \mathrm{Trans}(\tau_1,\tau_2)\rangle$

Proof. By induction on defining equations of Morph^p, Relat and Trans: Base case: $\langle op =, \mathrm{id}, \mathrm{id}, op =\rangle$ holds. Non-abstract type case: $\langle .,.,.,.\rangle$ is preserved as it is required in Definition 3. Abstract type case: for all τ_2, an instance of the abstract type of τ_1, $\langle \mathrm{relat}(\tau_1,\tau_2), \mathrm{morph}^+(\tau_1,\tau_2), \mathrm{morph}^-(\tau_1,\tau_2), \mathrm{trans}(\tau_1,\tau_2)\rangle$ holds by construction. And finally, the key fact that we need: $\langle R_1, Abs_1, Rep_1, T_1\rangle$ and $\langle R_2, Abs_2, Rep_2, T_2\rangle$ implies $\langle T_1 \circ\circ R_2 \circ\circ T_1^{-1}, Abs_2 \circ Abs_1, Rep_1 \circ Rep_2, T_1 \circ\circ T_2\rangle$, i.e., $\langle .,.,.,.\rangle$ is preserved through the composition of abstract types. We proved this key fact in Isabelle/HOL. □

If we compose Theorem 3 with the Theorem 2 and use $f = \mathrm{Morph}^+(\tau_1,\tau_2)\ t$ and $\mathrm{Relat}(\tau_1,\tau_2)\ t\ t$, we get the desired transfer rule $\mathrm{Trans}(\tau_1,\tau_2)\ t\ f$.

In the original Quotient package, $\mathrm{Relat}(\tau_1,\tau_2)$ in the abstract type case was defined as $\mathrm{Relat}(\tau_1, (\overline{\sigma'})\ \kappa_1) \circ\circ \mathrm{relat}((\overline{\sigma'})\ \kappa_1, \tau_2) \circ\circ \mathrm{Relat}(\tau_1, (\overline{\sigma'})\ \kappa_1)$. Although Theorem 1 still holds, Theorem 3 cannot be proven and thus the Quotient package does not cover the whole possible type universe. As a consequence, transferring of theorems did not work for general case of composed abstract types, but the package was still a great progress.

3.2 Implementation

In Isabelle/HOL, $\langle .,.,.,.\rangle$ is defined as the Quotient predicate, whose definition is equivalent to Definition 2. A new abstract type can be registered by a command **setup_lifting** by providing such a Quotient *R Abs Rep T* theorem, which certifies that the given components *R*, *Abs*, *Rep* and *T* constitute an abstract type. Quotient theorems for relators and map functions are registered by the attribute quot_map. Such a theorem for the function type is the most prominent one; another example is a theorem for the list type:

lemma fun_quotient: Quotient R_1 abs_1 rep_1 $T_1 \Longrightarrow$ Quotient R_2 abs_2 rep_2 $T_2 \Longrightarrow$
  Quotient $(R_1 \Mapsto R_2)$ $(rep_1 \mapsto abs_2)$ $(abs_1 \mapsto rep_2)$ $(T_1 \Mapsto T_2)$
lemma Quotient_list: Quotient *R Abs Rep T* $\Longrightarrow$
  Quotient (list_all2 *R*) (map *Abs*) (map *Rep*) (list_all2 *T*)

We implemented a syntax-driven procedure that proves a Quotient theorem for a given pair of types τ_1 and τ_2. This procedure recursively descends τ_1 and τ_2 to prove Theorem 3 for τ_1 and τ_2 and the implementation basically follows our induction definition of Morph^p, Relat and Trans in the previous section. The advantage is that this procedure not only proves the compound Quotient theorem in order to derive the transfer theorem, but it also synthesizes the terms $\mathrm{Morph}^p(\tau_1,\tau_2)$, $\mathrm{Relat}(\tau_1,\tau_2)$ and $\mathrm{Trans}(\tau_1,\tau_2)$ as a side effect. This approach was not used in the Quotient package; thanks to it, we got a simpler implementation and managed to remove many technical limitations of the original Quotient package with surprising ease.

Users generally will not prove the Quotient theorem manually for new types, as special commands exist to automate the process. The command **quotient_type** defines a new quotient type, internally proves the corresponding Quotient theorem and registers it with **setup_lifting**. We also support types defined by the command **typedef**. The theorem type_definition *Rep Abs* $\{x.\ P\ x\}$, which axiomatizes the newly defined subtype, can be supplied to **setup_lifting**. It then internally proves the quotient theorem Quotient (invariant *P*) *Abs Rep T*, where the transfer relation *T* is defined as $T\ x\ y \equiv Rep\ y = x$ and the equivalence relation invariant $P \equiv (\lambda x\ y.\ x = y \wedge P\ x)$.

Since the respectfulness theorem is the only proof obligation presented to the user, we also implemented a procedure that does some preprocessing to present this obligation in a user-friendly, readable form in **lift_definition**. The procedure also simplifies the goal if the involved relations come from a subtype. Then the user gets predicates and predicators (e.g., list_all for 'a list) instead of relations and relators. If the relation comes only from type copies, the respectfulness theorem is fully proven by our procedure.

We also implemented a procedure that automatically proves a parameterized transfer rule, which is a stronger transfer rule (see §2.3), in **lift_definition** if the user provides a theorem certifying that the concrete term used in the definition is parametric.

3.3 Code Equations

The code generator is a central component in Isabelle/HOL and is used in a lot of projects for algorithm verification. That is why when we define a new constant by **lift_definition**, we are concerned with how to execute the new constant provided the concrete term is also executable. This can be done by providing *code equations*. See [2] for more about the code equations and how Lifting and Transfer provide efficient code for operations on abstract types. The code generator accepts two types of equations:

– *Representation function equation* has form $Rep\ f = t$, where Rep is not in t and $Abs\ (Rep\ x) = x$ holds, which is provable for any abstract type in our context.
– *Abstract function equation* has form $f\ (Abs_1\ x_1) \ldots (Abs_n\ x_n) = t$.

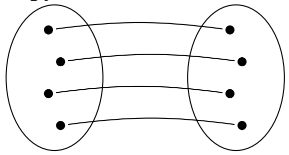
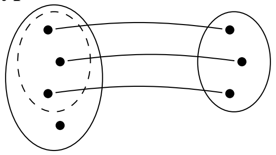
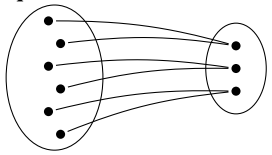
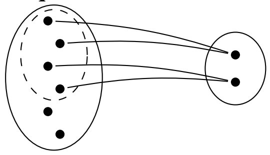
Here we give only a glimpse how the equations are proven. We assume the usual map function $\mapsto$ $((f \mapsto g)\ h = g \circ h \circ f)$ and relator $\Mapsto$ (as in Fig. 2) for function type. Let us have the definition $f = Abs\ t$, $f :: \sigma_1 \to \cdots \to \sigma_n \to \sigma_{n+1}$ and the proven respectfulness theorem $R\ t\ t$. Then from the Quotient theorem for the function type and from the construction described in §3.1 it easily follows that $R = R_1 \Mapsto \ldots \Mapsto R_n \Mapsto R_{n+1}$, $Abs = Rep_1 \mapsto \ldots \mapsto Rep_n \mapsto Abs_{n+1}$ and $Rep = Abs_1 \mapsto \ldots \mapsto Abs_n \mapsto Rep_{n+1}$.

Representation function equation. By unfolding the map function $\mapsto$ in the definition of f and using simple facts we get $Rep_{n+1}\ (f\ x_1 \ldots x_n) = Rep_{n+1}\ (Abs_{n+1}\ T)$, where $T = t\ (Rep_1\ x_1) \ldots (Rep_n\ x_n)$. If R_{n+1}, Rep_{n+1} and Abs_{n+1} represent a subtype or a type copy, the relation R_{n+1} is a subset of the equality and thus $Rep_{n+1}\ (Abs_{n+1}\ T) = T$ and finally $Rep_{n+1}\ (f\ x_1 \ldots x_n) = t\ (Rep_1\ x_1) \ldots (Rep_n\ x_n)$.

Abstraction function equation. By unfolding $\Mapsto$ in R and $\mapsto$ in Rep and using simple facts we get this equation $R_1\ x_1\ x_1 \longrightarrow \ldots \longrightarrow R_n\ x_n\ x_n \longrightarrow f\ (Abs_1\ x_1) \ldots (Abs_n\ x_n) = Abs_{n+1}\ (t\ x_1 \ldots x_n)$. If R_1 to R_n are relations that are composed from relators that preserve reflexivity (e.g., holds for any datatype relator) and the abstract types that are involved are total (i.e., a type copy or a total quotient), the procedure that we implemented discharges automatically each of these assumptions and gives us a plain equation.[5]

[5] The function relator $\Mapsto$ does not preserve reflexivity in the negative position. But this is not a limitation in practice, because there is hardly a function with a functional parameter that would have an abstract type in the negative position. Because $\Mapsto$ preserves equality, we can still discharge functional parameters using type copies or non-abstract functional parameters.

Table 1. Categorization of abstract types and respective equations

	total equivalence relation	partial equivalence relation
trivial relation (subset of =)	**type copy**  rep_eq: $+$ code: abs_eq abs_eq: $+$ relation: bi-unique, bi-total example: Mappings ('a, 'b) mapping = 'a $\Rightarrow$ 'b option	**subtype**  rep_eq: $+$ code: rep_eq abs_eq: $\sim$ relation: bi-unique, right-total example: Lift_Dlist 'a dlist = $\{x$ :: 'a list. distinct $x\}$
non-trivial relation	**total quotient**  rep_eq: $-$ code: abs_eq 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: $-$ code: none abs_eq: $\sim$ relation: right-unique, right-total example: Rat 'a rat = int $\times$ int / ratrel[6]

$+$ . . . yes $-$ . . . no $\sim$ . . . only with assumptions

Thus we can generate representation function equations for type copies and subtypes and abstraction function equations for any abstract type, but we can discharge the extra assumptions only for total types. See Tab. 1 for an overview.

4 Conclusion

We have presented a new design for automation of abstract types in Isabelle/HOL. The distinctive features and the main contributions are:

- The modular design of cleanly separated components with well-defined interfaces yields flexibility, i.e., Lifting is not limited to types defined by **quotient_type** and similarly Transfer to constants defined by **lift_definition**.
- Only one transfer rule is needed for different instances of polymorphic constants like fmap :: ('a $\Rightarrow$ 'b) $\Rightarrow$ 'a fset $\Rightarrow$ 'b fset.
- The Lifting package supports arbitrary type constructors, rather than only co-variant ones plus hard-coded function type.
- The Lifting package can handle a composition of abstract types in all cases.
- The Lifting package generates code equations for the code generator.
- The package generates the statement of the respectfulness theorem, discharges it automatically for type copies, and simplifies it to user-friendly form in other cases.

[6] ratrel $x\ y \equiv$ snd $x \neq 0 \wedge$ snd $y \neq 0 \wedge$ fst $x *$ snd $y =$ fst $y *$ snd x.

In Isabelle 2013, 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 [10].). This reduced the amount of boilerplate. But what has greater merit is that we can conclude that the packages singificantly improve the level of abstraction when building abstract types and moving terms and lemmas between different types.

Our packages are used in many theories by the Isabelle community: there are almost 400 calls of **lift_definition** and over 900 calls of transfer.[7] Besides other things, they are used to define various executable data structures: e.g., numerals, red-black trees, distinct lists, mappings, finite sets, associative lists, intervals, floats, multisets, finite bit strings, co-inductive streams, almost constant functions and others. We can conclude that our packages have found their users and become a standard part of Isabelle/HOL.

References

1. Coen, C.S.: A Semi-reflexive Tactic for (Sub-)Equational Reasoning. In: Filliâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) TYPES 2004. LNCS, vol. 3839, pp. 98–114. Springer, Heidelberg (2006)
2. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data Refinement in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 100–115. Springer, Heidelberg (2013)
3. Harrison, J.: Theorem Proving with the Real Numbers. Springer (1998)
4. Homeier, P.V.: A Design Structure for Higher Order Quotients. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 130–146. Springer, Heidelberg (2005)
5. Kaliszyk, C., Urban, C.: Quotients revisited for Isabelle/HOL. In: Proc. of the 26th ACM Symposium on Applied Computing (SAC 2011), pp. 1639–1644. ACM (2011)
6. Krauss, A.: Simplifying Automated Data Refinement via Quotients. Tech. rep., TU München (2011), `http://www21.in.tum.de/~krauss/papers/refinement.pdf`
7. Lammich, P.: Automatic data refinement. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 84–99. Springer, Heidelberg (2013)
8. Magaud, N.: Changing data representation within the Coq system. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 87–102. Springer, Heidelberg (2003)
9. Mitchell, J.C.: Representation Independence and Data Abstraction. In: POPL, pp. 263–276. ACM Press (January 1986)
10. Paulson, L.C.: Defining functions on equivalence classes. ACM Trans. Comput. Logic 7(4), 658–675 (2006)
11. Reynolds, J.C.: Types, Abstraction and Parametric Polymorphism. In: IFIP Congress, pp. 513–523 (1983)
12. Slotosch, O.: Higher Order Quotients and their Implementation in Isabelle HOL. In: Gunter, E.L., Felty, A.P. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 291–306. Springer, Heidelberg (1997)
13. Sozeau, M.: A New Look at Generalized Rewriting in Type Theory. In: 1st Coq Workshop Proceedings (2009)
14. Wadler, P.: Theorems for free! In: Functional Programming Languages and Computer Architecture, pp. 347–359. ACM Press (1989)

[7] Isabelle distribution and *Archive of Formal Proofs* (`http://afp.sf.net`), release 2013-1.