

# Higher Order Quotients in Higher Order Logic

Peter V. Homeier

U. S. Department of Defense, homeier@saul.cis.upenn.edu  
<http://www.cis.upenn.edu/~homeier>

**Abstract.** The quotient operation is a standard feature of set theory, where a set is partitioned into subsets by an equivalence relation. We reinterpret this idea for Higher Order Logic (HOL), where types are divided by an equivalence relation to create new types, called quotient types. We present a tool for the Higher Order Logic theorem prover to mechanically construct quotient types as new types in the HOL logic, and to automatically lift constants and theorems about the original types to corresponding constants and theorems about the quotient types. This package exceeds the functionality of Harrison's package, creating quotients of multiple mutually recursive types simultaneously, and supporting the equivalence of aggregate types, such as lists and pairs. Most importantly, this package successfully creates higher-order quotients, automatically lifting theorems with quantification over functions of any higher order. This is accomplished through the use of *partial equivalence relations*, a possibly nonreflexive version of equivalence relations. We demonstrate this tool by lifting Abadi and Cardelli's sigma calculus.

## 1 Introduction

The quotient operation is a standard feature of mathematics, including set theory and abstract algebra. It provides a way to cleanly identify elements that previously were distinct. This simplifies the system by removing unneeded structure.

Quotients have found many applications. Classic examples are the construction of the integers from pairs of non-negative natural numbers, or the construction of the rationals from pairs of integers. In the lambda calculus and similar calculi, it is common to identify terms which are alpha-equivalent, that differ only by the choice of local names used by binding operators. Other examples include the construction of bags from lists by ignoring order, and the construction of sets from bags by ignoring duplicates.

The ubiquity of quotients has recommended their investigation within the field of mechanical theorem proving. Isabelle/HOL has mechanical support for the creation of higher order quotients by Oscar Slotosch [18]. Independently, Larry Paulson has shown a construction of first-order quotients in Isabelle without any use of the Hilbert choice operator [16]. PVS uses quotients to support theory interpretations [14]. MetaPRL has quotients in its foundations, as a type with a new equality [15]. Coq, based on the Calculus of Constructions [10], supports quotients [6] but has some difficulties with higher order [4].

John Harrison has developed a package for the HOL theorem prover which supports first order quotients, including automation to define new quotient types and to lift to the quotient level both constants and theorems previously established [9]. This automatic lifting is key to practical support for quotients. A quotient of a group would be incomplete without also mapping the original group operation to a corresponding one for the quotient group. Similarly, theorems about the group which are independent of the equivalence relation should also be true of the quotient group. Mechanizing this lifting is vital for avoiding the repetition of proofs at the higher level which were already proved at the lower level. Such automation is not only practical, but intellectually cogent.

Despite the quality of Harrison’s excellent package, it does have limitations. It can only lift one type at a time, and does not deal with aggregate types, such as lists or pairs involving types being lifted, which makes it difficult to lift a family of mutually nested recursive types. Most importantly, it is limited to lifting only first order theorems, where quantification is permitted over the type being lifted, but not over functions or predicates involving the type being lifted.

In this paper we describe a new package for quotients for the Higher Order Logic theorem prover that meets these three concerns. It provides a tool for lifting multiple types across multiple equivalence relations simultaneously. Aggregate equivalence relations are produced and used automatically. But most significantly, this package supports the automatic lifting of theorems that involve higher order functions, including quantification, of any finite order. This is possible through the use of *partial equivalence relations* [17], as a possibly non-reflexive variant of equivalence relations, enabling the creation of quotients of function types. The relationship between these partial equivalence relations and their associated abstraction and representation functions (mapping between the lower and higher types) is expressed in *quotient theorems* which play a central and vital role in this package’s work.

The precise definition of the quotient relationship between the original and lifted types, and the proof of that relationship’s preservation for a function type, given existing quotients for the function’s domain and range, are the heart of this paper, and are presented in full detail. These form the core theory that justifies the treatment of all higher order functions, including higher order universal, existential, and unique existential quantification.

In addition, many existing polymorphic operators from the theories of lists, pairs, sums, and options have theorems pre-proven showing the operators’ respectfulness of equivalence relations and their preservation across quotients, yielding results at the high level which correspond properly with their results at the lower level. These respectfulness and preservation theorems enable the automatic lifting of theorems that mention these operators. Other operators can also be lifted by proving and including the corresponding respectfulness and preservation theorems which justify the operator’s use across quotient operations.

The system is thus extensible, both in terms of new operators, and even in terms of new polymorphic type operators, by proving and including theorems providing a quotient of the operator, given quotients of the argument types.

Higher order quotients are also implemented in the Isabelle/HOL system [13], using partial equivalence relations represented as a type class, with equivalence relations as a subclass [18]. That system provides a definitional framework for establishing quotient types, including higher order, but provides little automatic support. In particular, there is no automatic lifting of constants or theorems.

The structure of this paper is as follows. In section 2 we briefly review quotient sets. In section 3 we re-interpret this idea for type theory. Section 4 discusses equivalence relations and their aggregates. Section 5 describes partial equivalence relations, their extension for aggregate and function types, the properties of a quotient relationship, and conditional quotient theorems for lifting aggregate and function types. Section 6 describes an alternative design that avoids use of the Axiom of Choice. Section 7 presents the main tool of the quotient package, which lifts types, constants, and theorems across the quotients. The next several sections discuss its work in more detail. Section 8 describes the definition of new quotient types, and section 9 describes the definition of lifted versions of constants. Section 10 describes the various input theorems needed by the tool to automatically lift theorems. Section 11 describes the restrictions on theorems in order for them to lift properly, and section 12 loosens these restrictions, describing how the package will try to generate the proper theorem to lift even given an improper one. In sections 13 through 15, we present Abadi and Cardelli’s sigma calculus [1], its formulation in HOL and its lifting by quotients over alpha-equivalence. Finally, our conclusions are presented in section 16.

We are grateful for the helpful comments and suggestions made by Randolph Johnson, Sylvan Pinsky, Yvonne V. Shashoua, and Konrad Slind, and especially Michael Mislove for identifying partial equivalence relations and William Schneeberger for the key idea in the proof of theorem 27.

## 2 Quotient Sets

Quotient sets are a standard construction of set theory. They have found wide application in many areas of mathematics, including algebra and logic. The following description is abstracted from [5].

A binary relation  $\sim$  on  $S$  is an *equivalence relation* if it is reflexive, symmetric, and transitive.

$$\begin{array}{ll} \text{reflexive:} & \forall x \in S. \quad x \sim x \\ \text{symmetric:} & \forall x, y \in S. \quad x \sim y \Rightarrow y \sim x \\ \text{transitive:} & \forall x, y, z \in S. \quad x \sim y \wedge y \sim z \Rightarrow x \sim z \end{array}$$

Let  $\sim$  be an equivalence relation. Then the *equivalence class* of  $x$  (*modulo*  $\sim$ ) is defined as  $[x]_{\sim} \stackrel{\text{def}}{=} \{y \mid x \sim y\}$ . It follows [5] that

$$[x]_{\sim} = [y]_{\sim} \Leftrightarrow x \sim y.$$

The *quotient set*  $S/\sim$  is defined as

$$S/\sim \stackrel{\text{def}}{=} \{[x]_{\sim} \mid x \in S\}.$$

This is the set of all equivalence classes modulo  $\sim$  of elements in  $S$ .

### 3 Quotient Types

Now let us reinterpret this for the Higher Order Logic theorem prover, whose logic is a type theory, rather than a set theory. The following definitions and proofs are accomplished within HOL [7]. Let  $\tau$  be any type; then a binary relation on  $\tau$  can be represented in HOL as a curried function of type  $\tau \rightarrow (\tau \rightarrow \mathbf{bool})$ . Let  $E$  be an equivalence relation, which is a binary relation satisfying

$$\begin{aligned} \text{reflexivity:} & \quad \forall x : \tau. \quad E \ x \ x \\ \text{symmetry:} & \quad \forall x \ y : \tau. \quad E \ x \ y \Rightarrow E \ y \ x \\ \text{transitivity:} & \quad \forall x \ y \ z : \tau. \quad E \ x \ y \wedge E \ y \ z \Rightarrow E \ x \ z \end{aligned}$$

We will take advantage of the curried nature of  $E$ , where  $E \ x \ y = (E \ x) \ y$ .

**Definition 1.** *The equivalence class of  $x : \tau$  (modulo  $E$ ) is  $[x]_E \stackrel{\text{def}}{=} E \ x$ .*

This is the characteristic function of the  $[x]_E$  from set theory. As before,

**Lemma 2.**  $[x]_E = [y]_E \Leftrightarrow E \ x \ y$

Proof: We prove this as a biconditional.

$\Rightarrow$ : Assume  $[x]_E = [y]_E$ , which is  $E \ x = E \ y$  by definition 1. Then  $E \ x \ y = E \ y \ y$ , which is true by reflexivity.

$\Leftarrow$ : Conversely, assume  $E \ x \ y$ . By symmetry,  $E \ y \ x$ . We must show  $[x]_E = [y]_E$ , which by definition 1 is  $E \ x = E \ y$ . This is equality between functions, which by extension follows from  $\forall z. E \ x \ z \Leftrightarrow E \ y \ z$ . We split as a biconditional.

$\Rightarrow$ : Assume  $E \ x \ z$ . Since  $E \ y \ x$ ,  $E \ y \ z$  follows by transitivity.

$\Leftarrow$ : Assume  $E \ y \ z$ . Since  $E \ x \ y$ ,  $E \ x \ z$  follows by transitivity.  $\square$

Also, any relation  $E$  satisfying Lemma 2 is reflexive, symmetric, and transitive.

New types may be defined in HOL using the function `new_type_definition` [7, sections 18.2.2.3-5]. This function requires us to choose a representing type, and a predicate on that type denoting a subset that is nonempty.

We define as a new type the *quotient type*  $\tau/E$  by the representing type  $\tau \rightarrow \mathbf{bool}$ , and the predicate  $P : (\tau \rightarrow \mathbf{bool}) \rightarrow \mathbf{bool}$  where

**Definition 3.**  $P \ f \stackrel{\text{def}}{=} \exists x. f = [x]_E$ .

$P$  is nonempty because  $P \ [x]_E$  for all  $x : \tau$ . Let  $\tau' = \tau/E$ . We then use the HOL tool `define_new_type_bijections` [7], which automatically defines a function  $[-]_c : (\tau \rightarrow \mathbf{bool}) \rightarrow \tau'$  and its inverse  $[_]_c : \tau' \rightarrow (\tau \rightarrow \mathbf{bool})$ , such that

**Theorem 4.**  $(\forall a : \tau'. \ [ [a]_c ]_c = a) \wedge (\forall f : \tau \rightarrow \mathbf{bool}. P \ f \Leftrightarrow ([ [f]_c ]_c = f))$

and returns theorem 4 as its result. For this  $P$ , we can prove:

**Theorem 5.**  $(\forall a : \tau'. \ [ [a]_c ]_c = a) \wedge (\forall r : \tau. \ [ [r]_E ]_c = [r]_E)$ .

Proof: The left conjunct comes directly from theorem 4. Also by that theorem, the right conjunct is equivalent to  $\forall r. P \ [r]_E$ . Then  $P \ [r]_E = (\exists x. [r]_E = [x]_E)$  which is proven true by choosing the witness  $x = r$ .  $\square$

**Lemma 6** ( $\lfloor \_ \rfloor_c$  maps to equivalence classes).  $\forall a. \exists r. \lfloor a \rfloor_c = [r]_E$

Proof: By the left clause of theorem 4,  $\lceil \lfloor a \rfloor_c \rceil_c = a$ , so taking the  $\lfloor \_ \rfloor_c$  of both sides,  $\lfloor \lceil \lfloor a \rfloor_c \rceil_c \rfloor_c = \lfloor a \rfloor_c$ . By the right clause of theorem 4, this implies  $P \lfloor a \rfloor_c$ . By the definition of  $P$ , this is the goal.  $\square$

**Lemma 7** ( $\lfloor \_ \rfloor_c$  is one-to-one).  $\forall x y. \lceil [x]_E \rceil_c = \lceil [y]_E \rceil_c \Leftrightarrow [x]_E = [y]_E$

Proof: We will prove this as a biconditional.

$\Rightarrow$ : Assume  $\lceil [x]_E \rceil_c = \lceil [y]_E \rceil_c$ . Applying the function  $\lfloor \_ \rfloor_c$  to both sides gives us  $\lfloor \lceil [x]_E \rceil_c \rfloor_c = \lfloor \lceil [y]_E \rceil_c \rfloor_c$ . By the right clause of theorem 5,  $[x]_E = [y]_E$ .

$\Leftarrow$ : Assume  $[x]_E = [y]_E$ . Then applying  $\lceil \_ \rceil_c$  to both sides yields the goal,  $\lceil [x]_E \rceil_c = \lceil [y]_E \rceil_c$ .  $\square$

The functions  $\lceil \_ \rceil_c$  and  $\lfloor \_ \rfloor_c$  map between equivalence classes of type  $\tau \rightarrow \mathbf{bool}$  and the quotient type  $\tau'$ . Using these functions, we can define new functions  $\lceil \_ \rceil$  and  $\lfloor \_ \rfloor$  between the original type  $\tau$  and the quotient type  $\tau'$  as follows.

**Definition 8** (Quotient maps).

$$\begin{aligned} \lceil \_ \rceil : \tau \rightarrow \tau' & \quad [x] \stackrel{\text{def}}{=} \lceil [x]_E \rceil_c \quad (= \ [E \ x]_c) \\ \lfloor \_ \rfloor : \tau' \rightarrow \tau & \quad [a] \stackrel{\text{def}}{=} \$\mathbb{Q} \lfloor a \rfloor_c \quad (= \ \mathbb{Q}x. \lfloor a \rfloor_c \ x = \ \mathbb{Q}x. ([x]_E = \lfloor a \rfloor_c)) \end{aligned}$$

The  $\mathbb{Q}$  operator, which is here used as a prefix unary operator  $\$\mathbb{Q}$ , is a higher order version of Hilbert's choice operator  $\varepsilon$  [7]. It has type  $(\alpha \rightarrow \mathbf{bool}) \rightarrow \alpha$ , and is usually used as a binder, where  $\$\mathbb{Q} \ t = \mathbb{Q}x. \ t \ x$ . Given a predicate  $Q$  on a type, if any element of the type satisfies the predicate, then  $\$\mathbb{Q} \ Q$  returns an arbitrary element of that type which satisfies  $Q$ . If no element of the type satisfies  $Q$ , then  $\$\mathbb{Q} \ Q$  will return simply some unknown, arbitrary element of the type. The axiom in HOL about the behavior of  $\mathbb{Q}$  is  $\forall t \ x. \ t \ x \Rightarrow t \ (\mathbb{Q} \ t)$ . In the definition above, it selects some arbitrary representative  $x$  such that the equivalence class of  $x$  is the class representing  $a$ .

**Lemma 9.**  $\forall r. \lfloor \$\mathbb{Q} \ [r]_E \rfloor_E = [r]_E$

Proof: The axiom for the  $\mathbb{Q}$  operator is  $\forall t \ x. \ t \ x \Rightarrow t \ (\mathbb{Q} \ t)$ . Taking  $t = E \ r$  and  $x = r$ , we have  $E \ r \ r \Rightarrow E \ r \ (\mathbb{Q} \ E \ r)$ . But  $E \ r \ r$  by reflexivity, so by modus ponens,  $E \ r \ (\mathbb{Q} \ (E \ r))$ . Then by lemma 2,  $[r]_E = \lfloor \$\mathbb{Q} \ (E \ r) \rfloor_E$ , from which the goal follows by definition 1.  $\square$

**Theorem 10.**  $\forall a. \lceil \lfloor a \rfloor \rceil = a$

Proof: By lemma 6, there exists an  $r$  such that  $\lfloor a \rfloor_c = [r]_E$ . Then

$$\begin{aligned} \lceil \lfloor a \rfloor \rceil &= \lceil \lfloor \$\mathbb{Q} \ [a]_c \rfloor_E \rceil_c && \text{definition 8} \\ &= \lceil \lfloor \$\mathbb{Q} \ [r]_E \rfloor_E \rceil_c && \text{selection of } r \\ &= \lceil [r]_E \rceil_c && \text{lemma 9} \\ &= \lceil [a]_c \rceil_c && \text{selection of } r \\ &= a && \text{theorem 4} \end{aligned}$$

$\square$

**Theorem 11.**  $\forall r r'. E r r' \Leftrightarrow ([r] = [r'])$

Proof:  $E r r' \Leftrightarrow [r]_E = [r']_E$  lemma 2  
 $\Leftrightarrow [[r]_E]_c = [[r']_E]_c$  lemma 7  
 $\Leftrightarrow [r] = [r']$  definition 8

□

It is significant to note that the combination of theorems 10 and 11 succinctly express all that is necessary to use the quotient type. These theorems completely characterize the mapping functions between the original and quotient types.

## 4 Equivalence Relations and Equivalence Theorems

The statement that a given binary relation  $E$  is an equivalence relation means that the relation is reflexive, symmetric, and transitive:

*reflexivity*  $\quad |- \forall a. E a a$   
*symmetry*  $\quad |- \forall a b. E a b \Rightarrow E b a$   
*transitivity*  $\quad |- \forall a b c. E a b \wedge E b c \Rightarrow E a c$

These three properties can be encompassed in the *equivalence property*:

*equivalence*  $\quad |- \forall x y. E x y \Leftrightarrow (E x = E y)$

$E$  is a curried function of type  $\text{ty} \rightarrow \text{ty} \rightarrow \text{bool}$ .  $E$  may be either a constant or an expression. Such a theorem is called an *equivalence theorem* on type  $\text{ty}$ .

Given an equivalence theorem, we can obtain the reflexive, symmetric, and transitive properties, or given those three, construct the corresponding equivalence theorem, using the following Standard ML functions of our package.

```

equiv_refl : thm -> thm
equiv_sym  : thm -> thm
equiv_trans : thm -> thm
refl_sym_trans_equiv : thm -> thm -> thm -> thm
```

Given an equivalence relation  $E$  on values of type  $\text{ty}$ , there is a natural extension of  $E$  to lists of values of  $\text{ty}$ . This is expressed as `LIST_REL E`, which forms an equivalence relation of type  $\text{ty list} \rightarrow \text{ty list} \rightarrow \text{bool}$ . Similarly, equivalence relations on pairs, sums, and options may be formed from their constituent equivalence relations by the following operators.

Type	Operator	Type of operator
list	<code>LIST_REL</code>	$( 'a \rightarrow 'a \rightarrow \text{bool} ) \rightarrow 'a \text{ list} \rightarrow 'a \text{ list} \rightarrow \text{bool}$
pair	<code>###</code>	$( 'a \rightarrow 'a \rightarrow \text{bool} ) \rightarrow ( 'b \rightarrow 'b \rightarrow \text{bool} ) \rightarrow 'a \# 'b \rightarrow 'a \# 'b \rightarrow \text{bool}$
sum	<code>+++</code>	$( 'a \rightarrow 'a \rightarrow \text{bool} ) \rightarrow ( 'b \rightarrow 'b \rightarrow \text{bool} ) \rightarrow 'a + 'b \rightarrow 'a + 'b \rightarrow \text{bool}$
option	<code>OPTION_REL</code>	$( 'a \rightarrow 'a \rightarrow \text{bool} ) \rightarrow 'a \text{ option} \rightarrow 'a \text{ option} \rightarrow \text{bool}$

These operators are defined as indicated below in `quotientTheory`, a theory in this package.

**Definition 12.** `LIST_REL R [] [] = true`  
`LIST_REL R (a::as) [] = false`  
`LIST_REL R [] (b::bs) = false`  
`LIST_REL R (a::as) (b::bs) = R a b ∧ LIST_REL R as bs`

**Definition 13.** `(R1 ### R2) (a1, a2) (b1, b2) = R1 a1 b1 ∧ R2 a2 b2`

**Definition 14.** `(R1 +++ R2) (INL a1) (INL b1) = R1 a1 b1`  
`(R1 +++ R2) (INL a1) (INR b2) = false`  
`(R1 +++ R2) (INR a2) (INL b1) = false`  
`(R1 +++ R2) (INR a2) (INR b2) = R2 a2 b2`

**Definition 15.** `OPTION_REL R NONE NONE = true`  
`OPTION_REL R (SOME a) NONE = false`  
`OPTION_REL R NONE (SOME b) = false`  
`OPTION_REL R (SOME a) (SOME b) = R a b`

They take arguments which are the equivalence relations for component subtypes, and return an equivalence relation for the aggregate type.

Since the pair and sum relation operators have two arguments, they are infix, whereas the list and option relation operators are unary, prefix operators. The operator definitions may be needed to prove respectfulness (see sections 10.1, 10.3).

Given equivalence theorems for the constituent subtypes, the equivalence theorems for the natural extensions to aggregate types (e.g., lists, pairs, sums, and options) may be created by the following SML functions of our package.

```
list_equiv : thm -> thm
pair_equiv : thm -> thm -> thm
sum_equiv : thm -> thm -> thm
option_equiv : thm -> thm

identity_equiv : hol_type -> thm
make_equiv : thm list -> hol_type -> thm
```

`identity_equiv ty` creates the trivial equivalence theorem for the given type `ty`, using equality (=) as the equivalence relation.

`make_equiv equivs ty` creates an equivalence theorem for the given type `ty`, which may be a complex type expression with lists, pairs, etc. Here `equivs` is a list of both equivalence theorems for the base types and conditional equivalence theorems for type operators (see section 4.1).

#### 4.1 Conditional equivalence theorems

Conditional equivalence theorems for type operators have the form:

$$\begin{aligned}
&|- \forall E1 \dots En. \\
&\quad (\forall (x:\alpha_1) (y:\alpha_1). E1 \ x \ y \Leftrightarrow (E1 \ x = E1 \ y)) \Rightarrow \\
&\quad \dots \\
&\quad (\forall (x:\alpha_n) (y:\alpha_n). En \ x \ y \Leftrightarrow (En \ x = En \ y)) \Rightarrow \\
&\quad (\forall (x:(\alpha_1, \dots, \alpha_n)op) (y:(\alpha_1, \dots, \alpha_n)op). \\
&\quad \quad OP\_REL \ E1 \ \dots \ En \ x \ y \Leftrightarrow \\
&\quad \quad (OP\_REL \ E1 \ \dots \ En \ x = OP\_REL \ E1 \ \dots \ En \ y))
\end{aligned}$$

Given the type operator  $(\alpha_1, \dots, \alpha_n)op$ ,  $OP\_REL$  is an operator which takes  $n$  arguments, which are the equivalence relations  $E1$  through  $En$  on the types  $\alpha_1$  through  $\alpha_n$ , and yields an equivalence relation for the type  $(\alpha_1, \dots, \alpha_n)op$ . The type operators `list`, `prod`, `sum`, and `option` have the following conditional equivalence theorems in `quotientTheory`.

**LIST\_EQUIV:**

$$\begin{aligned}
&|- \forall R. (\forall x \ y. R \ x \ y \Leftrightarrow (R \ x = R \ y)) \Rightarrow \\
&\quad (\forall x \ y. LIST\_REL \ R \ x \ y \Leftrightarrow (LIST\_REL \ R \ x = LIST\_REL \ R \ y))
\end{aligned}$$

**PAIR\_EQUIV:**

$$\begin{aligned}
&|- \forall R1 \ R2. \\
&\quad (\forall x \ y. R1 \ x \ y \Leftrightarrow (R1 \ x = R1 \ y)) \Rightarrow \\
&\quad (\forall x \ y. R2 \ x \ y \Leftrightarrow (R2 \ x = R2 \ y)) \Rightarrow \\
&\quad (\forall x \ y. (R1 \ ### \ R2) \ x \ y \Leftrightarrow ((R1 \ ### \ R2) \ x = (R1 \ ### \ R2) \ y))
\end{aligned}$$

**SUM\_EQUIV:**

$$\begin{aligned}
&|- \forall R1 \ R2. \\
&\quad (\forall x \ y. R1 \ x \ y \Leftrightarrow (R1 \ x = R1 \ y)) \Rightarrow \\
&\quad (\forall x \ y. R2 \ x \ y \Leftrightarrow (R2 \ x = R2 \ y)) \Rightarrow \\
&\quad (\forall x \ y. (R1 \ +++ \ R2) \ x \ y \Leftrightarrow ((R1 \ +++ \ R2) \ x = (R1 \ +++ \ R2) \ y))
\end{aligned}$$

**OPTION\_EQUIV:**

$$\begin{aligned}
&|- \forall R. (\forall x \ y. R \ x \ y \Leftrightarrow (R \ x = R \ y)) \Rightarrow \\
&\quad (\forall x \ y. OPTION\_REL \ R \ x \ y \Leftrightarrow (OPTION\_REL \ R \ x = OPTION\_REL \ R \ y))
\end{aligned}$$

## 5 Partial Equivalence Relations and Quotient Theorems

In this section we introduce a new definition of the quotient relationship, based on *partial equivalence relations*, a related but different concept than equivalence relations. Every equivalence relation is a partial equivalence relation, but not every partial equivalence relation is an equivalence relation. An equivalence relation is reflexive, symmetric and transitive, while a partial equivalence relation is symmetric and transitive, but not necessarily reflexive on all of its domain.

Why use partial equivalence relations with a weaker reflexivity condition? The reason involves forming quotients of higher order types, that is, functions whose domains or ranges involve types being lifted. Unlike lists and pairs, the equivalence relations for the domain and range do not naturally extend to a useful equivalence relation for functions from the domain to the range.

The reason is that not all functions which are elements of the function type are *respectful* of the associated equivalence relations, as described in section 10.1. For example, given an equivalence relation  $E : \mathbf{ty} \rightarrow \mathbf{ty} \rightarrow \mathbf{bool}$ , the set of functions from  $\mathbf{ty}$  to  $\mathbf{ty}$  may contain a function  $f^?$  where for some  $x$  and  $y$  which are equivalent ( $E x y$ ), the results of  $f^?$  are not equivalent ( $\neg E(f^? x)(f^? y)$ ). Such disrespectful functions cannot be worked with; they do not correspond to any function at the abstract quotient level. Any equivalence relation  $E^2$  for the function type  $\mathbf{ty} \rightarrow \mathbf{ty}$  would have to be reflexive, so we would have  $E^2 f^? f^?$ . But this would invalidate  $\forall f_1, f_2, x, y. E^2 f_1 f_2 \wedge E x y \Rightarrow E(f_1 x)(f_2 y)$ , the higher version of which is  $\forall f_1, f_2, x, y. f_1 = f_2 \wedge x = y \Rightarrow f_1 x = f_2 y$ , which is obviously true. Every theorem at the higher level should correspond to a theorem at the lower level. Therefore we exclude such disrespectful functions from partial equivalence relations. The resulting relation cannot be a true equivalence, but it could be described as an equivalence relation on the field of the relation, where the field is the set of elements which are related to any element.

First, we say an element  $r$  *respects*  $R$  if and only if  $R r r$ .

**Definition 16 (Quotient).** *A relation  $R$  with abstraction function  $abs$  and representation function  $rep$  (between the representation type  $\mathbf{ty}$  and the abstraction type  $\mathbf{qty}$ ) is a quotient (notated as  $\langle R, abs, rep \rangle$ ) if and only if*

- (1)  $\forall a : \mathbf{qty}. abs (rep a) = a$
- (2)  $\forall a : \mathbf{qty}. R (rep a) (rep a)$
- (3)  $\forall r, s : \mathbf{ty}. R r s \Leftrightarrow R r r \wedge R s s \wedge (abs r = abs s)$

Property (1) states that  $abs$  is the left inverse of  $rep$ .

Property (2) states that  $R$  is reflexive on the range of  $rep$ .

Property (3) states that two elements of  $\mathbf{ty}$  are related by  $R$  if and only if each element respects  $R$  and their abstractions are equal.

These three properties (1-3) describe the way the partial equivalence relation  $R$  works together with  $abs$  and  $rep$  to establish the correct quotient relationship between the lower type  $\mathbf{ty}$  and the quotient type  $\mathbf{qty}$ . The precise definition of this quotient relationship is a central contribution of this work. This relationship is defined in the HOL logic as a new predicate:

```

QUOTIENT (R: 'a -> 'a -> bool) (abs: 'a -> 'b) (rep: 'b -> 'a) =
  (∀a. abs (rep a) = a) ∧
  (∀a. R (rep a) (rep a)) ∧
  (∀r s. R r s ⇔ R r r ∧ R s s ∧ (abs r = abs s))

```

The relationship that  $R$  with  $abs$  and  $rep$  is a quotient is expressed in HOL as

|- QUOTIENT  $R$   $abs$   $rep$

A theorem of this form is called a *quotient theorem*.

As will be shown in section 5.2, these three properties support the inference of a quotient theorem for a function type, given the quotient theorems for the domain and the range. This inference is necessary to enable higher order quotients.

### 5.1 Aggregate Quotient Theorems

Quotient theorems are used to control the process of defining lifted constants and lifting theorems. A quotient theorem is needed for each type involved in a constant being lifted, where each type involved is either the type of a parameter or the type of the result. For aggregate types, the tool automatically proves any needed quotient theorems for the aggregate types from the available quotient theorems for the constituent subtypes of the aggregate types. To accomplish this, the tool uses conditional quotient theorems (section 5.2). These are provided preproven for some standard aggregate type operators. For new aggregate types that the user may declare, new conditional quotient theorems can be added to extend the tool to handle the new aggregate types.

This section discusses the creation of these quotient theorems for aggregate types. Some aggregate equivalence relation operators have been already described in section 4, and these can equally be used to build aggregate partial equivalence relations. In addition, for function types, the following is added:

Type Operator	Type of operator
fun	$\begin{aligned} & \text{===> : } ('a \rightarrow 'a \rightarrow \text{bool}) \rightarrow ('b \rightarrow 'b \rightarrow \text{bool}) \rightarrow \\ & \quad ('a \rightarrow 'b) \rightarrow ('a \rightarrow 'b) \rightarrow \text{bool} \end{aligned}$

The definition of the function relation operator is:

**Definition 17.**  $(R_1 \text{===>} R_2) f g \Leftrightarrow \forall x y. R_1 x y \Rightarrow R_2 (f x) (g y)$ .

Note  $R_1 \text{===>} R_2$  is *not* in general an equivalence relation (it is not reflexive). It is reflexive at a function  $f$ ,  $(R_1 \text{===>} R_2) f f$ , if and only if  $f$  is respectful.

To ease the creation of quotient theorems, we provide several Standard ML functions that automatically prove the necessary quotient theorems for lists, pairs, sums, options, and function types, given the quotient theorems for the subtypes which are the arguments to these type operators.

```

list_quotient : thm -> thm
pair_quotient : thm -> thm -> thm
sum_quotient : thm -> thm -> thm
option_quotient : thm -> thm
fun_quotient : thm -> thm -> thm

identity_quotient : hol_type -> thm
make_quotient : thm list -> hol_type -> thm

```

These functions prove and return quotient theorems, of the form

|- QUOTIENT *R abs rep*

The first five functions all take as arguments quotient theorems for the constituent subtypes that are arguments to the aggregate type operator. The last two take an `hol_type` as an argument, which is the type of elements compared by the partial equivalence relation of the desired quotient theorem. None of these create new types, they simply apply existing type operators.

Sometimes one desires to perform the quotient operation on some arguments of the type operator but not on others. In these cases, to indicate an argument which is not to be changed, supply in that place a quotient theorem created by the `identity_quotient` function, which takes any HOL type and returns the identity quotient theorem for that type, using equality and the identity function,

|- QUOTIENT ( $\$=$  : *ty* -> *ty* -> bool) (I: *ty* -> *ty*) (I: *ty* -> *ty*)

In case one would need to create a quotient theorem for a complex type, the `make_quotient` function takes a list of quotient theorems and an HOL type, and returns a quotient theorem for that type, automatically constructing it recursively according to the structure of the type and the supplied quotient theorems.

The quotient theorems returned by the aggregate quotient functions involve not only the aggregate partial equivalence relation, but also aggregate abstraction and representation functions. These are constructed from the component abstraction and representation functions using the following “map” operators.

Type	Operator	Type of operator, examples of <i>abs</i> and <i>rep</i> fns
list	MAP :	$(\text{'a} \rightarrow \text{'b}) \rightarrow \text{'a list} \rightarrow \text{'b list}$ <i>examples: (MAP abs) , (MAP rep)</i>
pair	## :	$(\text{'a} \rightarrow \text{'b}) \rightarrow (\text{'c} \rightarrow \text{'d}) \rightarrow$ $\text{'a} \# \text{'c} \rightarrow \text{'b} \# \text{'d}$ <i>examples: (abs<sub>1</sub> ## abs<sub>2</sub>) , (rep<sub>1</sub> ## rep<sub>2</sub>)</i>
sum	++ :	$(\text{'a} \rightarrow \text{'b}) \rightarrow (\text{'c} \rightarrow \text{'d}) \rightarrow$ $\text{'a} + \text{'c} \rightarrow \text{'b} + \text{'d}$ <i>examples: (abs<sub>1</sub> ++ abs<sub>2</sub>) , (rep<sub>1</sub> ++ rep<sub>2</sub>)</i>
option	OPTION_MAP :	$(\text{'a} \rightarrow \text{'b}) \rightarrow \text{'a option} \rightarrow \text{'b option}$ <i>examples: (OPTION_MAP abs) , (OPTION_MAP rep)</i>
fun	--> :	$(\text{'a} \rightarrow \text{'b}) \rightarrow (\text{'c} \rightarrow \text{'d}) \rightarrow$ $(\text{'b} \rightarrow \text{'c}) \rightarrow \text{'a} \rightarrow \text{'d}$ <i>examples: (rep<sub>1</sub> --&gt; abs<sub>2</sub>) , (abs<sub>1</sub> --&gt; rep<sub>2</sub>)</i>

The definitions of the above operators are indicated below. They are created in theory `quotientTheory` or in standard HOL.

**Definition 18.**  $\text{MAP } f \ \square = \square$   
 $\text{MAP } f (a :: as) = (f a) :: (\text{MAP } f as)$

**Definition 19.**  $(f_1 \ \#\# \ f_2) (a_1, a_2) = (f_1 a_1, f_2 a_2)$

**Definition 20.**  $(f_1 \ ++ \ f_2) (\text{INL } a_1) = \text{INL } (f_1 a_1)$   
 $(f_1 \ ++ \ f_2) (\text{INR } a_2) = \text{INR } (f_2 a_2)$

**Definition 21.**  $\text{OPTION\_MAP } f \ \text{NONE} = \text{NONE}$   
 $\text{OPTION\_MAP } f (\text{SOME } a) = \text{SOME } (f a)$

The function map operator definition is of special interest:

**Definition 22.**  $(f \ --> \ g) h \ x = g (h (f \ x))$ .

MAP and OPTION\_MAP are prefix operators, and the others are infix.

The identity quotient map operator is the identity operator  $I: 'a \rightarrow 'a$ . These map operators are inserted automatically in the quotient theorems created by the aggregate quotient functions. Each resulting quotient theorem establishes that the aggregate type’s partial equivalence relation, abstraction function, and representation function properly relate together to form a quotient of the appropriate aggregate type. The map operator definitions may be needed to prove the preservation of new polymorphic operators (see section 10.2).

The key feature of these map operators is that they map a value of the given polymorphic type to a value of the same polymorphic type operator, but whose component types are of the other level (quotient vs. original). Thus MAP  $f$  takes a list and returns a list,  $f_1 \ \#\# \ f_2$  takes a pair and returns a pair, etc.

In other designs for quotients, the values at the quotient level were considered to be exactly the sets of equivalent values from the original level. So a quotient value corresponding to a list, say, would not be a list but a set of lists. That approach meant that in order to manipulate the quotient versions of lists one would have to define new constants to mimic each of the normal list operations like HD, TL, NIL, CONS, MAP, FOLDR, and APPEND, but operating on the quotient values. We would also need to define functions to translate between these “sets that are behaving like lists” and the true list type. All of this would be a tremendous burden and only obfuscate the essential structure of the system.

Rather than this, we take the approach that polymorphic type operators are used to form both the original and the quotient types from the underlying original and quotient types, and that the map functions map between these two applications of the same polymorphic type operator, applied to different type arguments from the two different levels.

## 5.2 Conditional quotient theorems

The `make_quotient` function supports the extension of quotients over any type operator for which a conditional quotient theorem is included in the first argument of `make_quotient`. A conditional quotient theorem has antecedents which

are statements that the constituent subtypes of the type operator are quotients, and a consequent that states that the extension over the type operator is a quotient. Each antecedent has the form of a quotient theorem, but the types involved are simple type variables in the HOL logic.

```
|- ∀R1 (abs1:α1->β1) rep1. QUOTIENT R1 abs1 rep1 ⇒ ...
  ∀Rn (absn:αn->βn) repn. QUOTIENT Rn absn repn ⇒
  QUOTIENT (OP_REL R1 ... Rn) (OP_MAP abs1 ... absn rep1 ... repn)
  (OP_MAP rep1 ... repn abs1 ... absn)
```

OP\_REL creates a partial equivalence relation for the type operator  $(\alpha_1, \dots, \alpha_n)op$ . OP\_MAP is a map operator which takes up to  $2n$  arguments, taken from the abstraction or representation functions between the types  $\alpha_1$  through  $\alpha_n$  and  $\beta_1$  through  $\beta_n$ . Depending on the order of the arguments, it yields either an abstraction or representation function between  $(\alpha_1, \dots, \alpha_n)op$  and  $(\beta_1, \dots, \beta_n)op$ .

Here are the conditional quotient theorems for the `list`, `prod`, `sum`, `option`, and, most significantly, `fun` type operators, as found in `quotientTheory`:

LIST\_QUOTIENT:

```
|- ∀R abs rep. QUOTIENT R abs rep ⇒
  QUOTIENT (LIST_REL R) (MAP abs) (MAP rep)
```

PAIR\_QUOTIENT:

```
|- ∀R1 abs1 rep1. QUOTIENT R1 abs1 rep1 ⇒
  ∀R2 abs2 rep2. QUOTIENT R2 abs2 rep2 ⇒
  QUOTIENT (R1 ### R2) (abs1 ## abs2) (rep1 ## rep2)
```

SUM\_QUOTIENT:

```
|- ∀R1 abs1 rep1. QUOTIENT R1 abs1 rep1 ⇒
  ∀R2 abs2 rep2. QUOTIENT R2 abs2 rep2 ⇒
  QUOTIENT (R1 +++ R2) (abs1 ++ abs2) (rep1 ++ rep2)
```

OPTION\_QUOTIENT:

```
|- ∀R abs rep. QUOTIENT R abs rep ⇒
  QUOTIENT (OPTION_REL R) (OPTION_MAP abs) (OPTION_MAP rep)
```

FUN\_QUOTIENT:

```
|- ∀R1 abs1 rep1. QUOTIENT R1 abs1 rep1 ⇒
  ∀R2 abs2 rep2. QUOTIENT R2 abs2 rep2 ⇒
  QUOTIENT (R1 ===> R2) (rep1 --> abs2) (abs1 --> rep2)
```

This last theorem is of critical importance to forming higher-order quotients.

**Theorem 23 (Function quotients).** *If relations  $R_1$  and  $R_2$  with abstraction functions  $abs_1$  and  $abs_2$  and representation functions  $rep_1$  and  $rep_2$ , respectively, are quotients, then  $R_1 ===> R_2$  with abstraction function  $rep_1 --> abs_2$  and representation function  $abs_1 --> rep_2$  is a quotient.*

**Proof:** We need to prove the three properties of definition 16:

*Property 1.* Prove for all  $a$ ,  $(rep_1 \dashrightarrow abs_2) ((abs_1 \dashrightarrow rep_2) a) = a$ .

Proof: The equality here is between functions, and by extension, true if for all values  $x$  in  $a$ 's domain,  $(rep_1 \dashrightarrow abs_2) ((abs_1 \dashrightarrow rep_2) a) x = a x$ .

By the definition of  $\dashrightarrow$ , this is  $abs_2 ((abs_1 \dashrightarrow rep_2) a (rep_1 x)) = a x$ , and then  $abs_2 (rep_2 (a (abs_1 (rep_1 x)))) = a x$ . By Property 1 of  $\langle R_1, abs_1, rep_1 \rangle$ ,  $abs_1 (rep_1 x) = x$ , and by Property 1 of  $\langle R_2, abs_2, rep_2 \rangle$ ,  $\forall b. abs_2 (rep_2 b) = b$ , so this reduces to  $a x = a x$ , true.

*Property 2.* Prove  $(R_1 \implies R_2) ((abs_1 \dashrightarrow rep_2) a) ((abs_1 \dashrightarrow rep_2) a)$ .

Proof: By the definition of  $\implies$ , this is

$\forall x, y. R_1 x y \implies R_2 ((abs_1 \dashrightarrow rep_2) a x) ((abs_1 \dashrightarrow rep_2) a y)$ . Assume  $R_1 x y$ , and show  $R_2 ((abs_1 \dashrightarrow rep_2) a x) ((abs_1 \dashrightarrow rep_2) a y)$ . By the definition of  $\dashrightarrow$ , this is  $R_2 (rep_2 (a (abs_1 x))) (rep_2 (a (abs_1 y)))$ . Now since  $R_1 x y$ , by property 3 of  $\langle R_1, abs_1, rep_1 \rangle$ ,  $abs_1 x = abs_1 y$ . Substituting this into our goal, we must prove  $R_2 (rep_2 (a (abs_1 y))) (rep_2 (a (abs_1 y)))$ . But this is an instance of property 2 of  $\langle R_2, abs_2, rep_2 \rangle$ , and so the goal is proven.

*Property 3.* Prove  $(R_1 \implies R_2) r s \Leftrightarrow$

$(R_1 \implies R_2) r r \wedge (R_1 \implies R_2) s s \wedge ((rep_1 \dashrightarrow abs_2) r = (rep_1 \dashrightarrow abs_2) s)$ .

The last conjunct on the right side is equality between functions, so by extension this is  $(R_1 \implies R_2) r s \Leftrightarrow (R_1 \implies R_2) r r \wedge (R_1 \implies R_2) s s \wedge$

$$(\forall x. (rep_1 \dashrightarrow abs_2) r x = (rep_1 \dashrightarrow abs_2) s x).$$

By the definitions of  $\implies$  and  $\dashrightarrow$ , this is  $(1) \Leftrightarrow (2) \wedge (3) \wedge (4)$ , where

- (1)  $(\forall x, y. R_1 x y \implies R_2 (r x) (s y))$
- (2)  $(\forall x, y. R_1 x y \implies R_2 (r x) (r y))$
- (3)  $(\forall x, y. R_1 x y \implies R_2 (s x) (s y))$
- (4)  $(\forall x. (abs_2 (r (rep_1 x)) = abs_2 (s (rep_1 x))))$ .

We prove  $(1) \Leftrightarrow (2) \wedge (3) \wedge (4)$  as a biconditional with two goals.

*Goal 1.*  $(\implies)$  Assume (1). Then we must prove (2), (3), and (4).

*Subgoal 1.1.* (Proof of (2)) Assume  $R_1 x y$ . We must prove  $R_2 (r x) (r y)$ .

From  $R_1 x y$  and property 3 of  $\langle R_1, abs_1, rep_1 \rangle$ , we also have  $R_1 x x$  and  $R_1 y y$ . From (1) and  $R_1 x y$ , we have  $R_2 (r x) (s y)$ . From (1) and  $R_1 y y$ , we have  $R_2 (r y) (s y)$ . Then by symmetry and transitivity of  $R_2$ , the goal is proven.

*Subgoal 1.2.* (Proof of (3)) Similar to the previous subgoal.

*Subgoal 1.3.* (Proof of (4))  $R_1 (rep_1 x) (rep_1 x)$  follows from property 2 of  $\langle R_1, abs_1, rep_1 \rangle$ . From (1), we have  $R_2 (r (rep_1 x)) (s (rep_1 x))$ . Then the goal follows from this and the third conjunct of property 3 of  $\langle R_2, abs_2, rep_2 \rangle$ .

*Goal 2.*  $(\Leftarrow)$  Assume (2), (3), and (4). We must prove (1). Assume  $R_1 x y$ . Then we must prove  $R_2 (r x) (s y)$ . From  $R_1 x y$  and property 3 of  $\langle R_1, abs_1, rep_1 \rangle$ , we also have  $R_1 x x$ ,  $R_1 y y$ , and  $abs_1 x = abs_1 y$ . By property 3 of  $\langle R_2, abs_2, rep_2 \rangle$ , the goal is  $R_2 (r x) (r x) \wedge R_2 (s y) (s y) \wedge abs_2 (r x) = abs_2 (s y)$ . This breaks into three subgoals.

*Subgoal 2.1.* Prove  $R_2 (r x) (r x)$ . This follows from  $R_1 x x$  and (2).

*Subgoal 2.2.* Prove  $R_2 (s y) (s y)$ . This follows from  $R_1 y y$  and (3).

*Subgoal 2.3.* Prove  $abs_2 (r x) = abs_2 (s y)$ .

**Lemma.** *If  $\langle R, abs, rep \rangle$  and  $R z z$ , then  $R (rep (abs z)) z$ .  
 $R (rep (abs z)) (rep (abs z))$ , by property 2 of  $\langle R, abs, rep \rangle$ .  
 From the hypothesis,  $R z z$ . From property 1 of  $\langle R, abs, rep \rangle$ ,  
 $abs (rep (abs z)) = abs z$ . From these three statements and  
 property 3 of  $\langle R, abs, rep \rangle$ , we have  $R (rep (abs z)) z$ .  $\square$*

By the lemma and  $R_1 x x$ , we have  $R_1 (rep_1 (abs_1 x)) x$ . Similarly, by the lemma and  $R_1 y y$ , we have  $R_1 (rep_1 (abs_1 y)) y$ . Then by (2), we have  $R_2 (r (rep_1 (abs_1 x))) (r x)$ , and by (3),  $R_2 (s (rep_1 (abs_1 y))) (s y)$ . From these and property 3 of  $\langle R_2, abs_2, rep_2 \rangle$ ,

$$\begin{aligned} abs_2 (r (rep_1 (abs_1 x))) &= abs_2 (r x) \text{ and} \\ abs_2 (s (rep_1 (abs_1 y))) &= abs_2 (s y). \end{aligned}$$

But by  $abs_1 x = abs_1 y$  and (4), the left hand sides of these two equations are equal, so their right hand sides must be also,  $abs_2 (r x) = abs_2 (s y)$ , which proves the goal.  $\square$

## 6 The Axiom of Choice

Gregory Moore wrote that “Rarely have the practitioners of mathematics, a discipline known for the certainty of its conclusions, differed so vehemently over one of its central premises as they have done over the Axiom of Choice. Yet without the Axiom, mathematics today would be quite different.” [12] Today, this discussion continues. Some theorem provers are based on classical logic, and others on a constructivist logic. In higher order logic, the Axiom of Choice is represented by Hilbert’s  $\varepsilon$ -operator [11], also called the indefinite description operator. Paulson’s lucid recent work [16] exhibits an approach to quotients which avoids the use of Hilbert’s  $\varepsilon$ -operator, by instead using the definite description operator  $\iota$  ([13], §5.10). These two operators may be axiomatized as follows:

$$\begin{aligned} \forall P x. P x \Rightarrow P(\varepsilon P) & \quad \text{or} \quad \forall P. (\exists x. P x) \Rightarrow P(\varepsilon P) \\ \forall P x. P x \Rightarrow (\forall y. P y \Rightarrow x = y) \Rightarrow P(\iota P) & \quad \text{or} \quad \forall P. (\exists! x. P x) \Rightarrow P(\iota P) \end{aligned}$$

The  $\iota$ -operator yields the single element of a singleton set,  $\iota\{z\} = z$ , but its result on non-singleton sets is indeterminate. By contrast, the  $\varepsilon$ -operator chooses some indeterminate element of any non-empty set, even nondenumerable. The  $\iota$ -operator is weaker than the  $\varepsilon$ -operator, and less objectionable to constructivists.

Thus it is of interest to determine if a design for higher order quotients may be formulated using only  $\iota$ , not  $\varepsilon$ . Inspired by Paulson, we investigate this.

**Definition 24 (Quotient).** *A relation  $R$  with abstraction function  $abs$  (between the representation type  $\mathbf{ty}$  and the abstraction type  $\mathbf{qty}$ ) is a quotient (notated as  $\langle R, abs \rangle$ ) if and only if*

- (1)  $\forall a : \mathbf{qty}. \exists r : \mathbf{ty}. R r r \wedge (abs r = a)$
- (2)  $\forall r, s : \mathbf{ty}. R r s \Leftrightarrow R r r \wedge R s s \wedge (abs r = abs s)$

Property (1) states that for every abstract element  $a$  of  $\mathbf{qty}$  there is a representative in  $\mathbf{ty}$  which respects  $R$  and whose abstraction is  $a$ .

Property (2) states that two elements of  $\mathbf{ty}$  are related by  $R$  if and only if each element respects  $R$  and their abstractions are equal.

The quotients for new quotient types based on equivalence relations may now be proven without any use of the Hilbert  $\varepsilon$ -operator, in contrast to definition 8 in §3. So may the identity quotient for type  $\alpha$ ,  $\langle \mathbb{S} = : \alpha \rightarrow \alpha \rightarrow \mathbf{bool}, \mathbf{I} : \alpha \rightarrow \alpha \rangle$ . From definition 24 also follow analogs of the conditional quotient theorems, e.g.,

$$\langle R, \mathit{abs} \rangle \Rightarrow \langle \mathbf{LIST\_REL} \ R, \mathbf{MAP} \ \mathit{abs} \rangle$$

for lists and similarly for pairs, sums and option types. Functions are lifted by the abstraction operation for functions, which requires two new definitions:

$$\begin{aligned} (\mathit{abs} \Downarrow R) \ a \ r &\stackrel{\text{def}}{=} R \ r \ r \ \wedge \ \mathit{abs} \ r = a \\ (\mathit{rep} \dashrightarrow \mathit{abs}) \ f \ x &\stackrel{\text{def}}{=} \iota \ (\mathbf{IMAGE} \ \mathit{abs} \ (\mathbf{IMAGE} \ f \ (\mathit{rep} \ x))) \end{aligned}$$

Note that for the identity quotient,  $\langle \mathbf{I} \Downarrow \mathbb{S} = \rangle = \mathbb{S} =$ .

The critical conditional quotient theorem for functions has also been proven:

**Theorem 25 (Function Conditional Quotient).**

$$\langle R_1, \mathit{abs}_1 \rangle \Rightarrow \langle R_2, \mathit{abs}_2 \rangle \Rightarrow \langle R_1 \implies R_2, (\mathit{abs}_1 \Downarrow R_1) \dashrightarrow \mathit{abs}_2 \rangle$$

Unfortunately, the proof requires using the Axiom of Choice. In fact, this theorem implies the Axiom of Choice, in that it implies the existence of an operator which obeys the axiom of the Hilbert  $\varepsilon$ -operator, as seen by the following development.

**Theorem 26 (Partial abstraction quotients).** *If  $f$  is any function from type  $\alpha$  to  $\beta$ , and  $Q$  is any predicate on  $\alpha$ , such that  $\forall y:\beta. \exists x:\alpha. Q \ x \wedge (f \ x = y)$ , then the partial equivalence relation  $R = \lambda r \ s. Q \ r \wedge Q \ s \wedge (f \ r = f \ s)$  with abstraction function  $f$  is a quotient.*

**Proof:** Follows easily from substituting  $R$  in definition 24 and simplifying.  $\square$

**Theorem 27 (Partial inverses exist).** *If  $f$  is any function from type  $\alpha$  to  $\beta$ , and  $Q$  is any predicate on  $\alpha$ , such that  $\forall y:\beta. \exists x:\alpha. Q \ x \wedge (f \ x = y)$ , then there exists a function  $g$  such that  $f \circ g = \mathbf{I}$  and  $\forall y. Q \ (g \ y)$ . [William Schneeberger]*

**Proof:** Assuming the function conditional quotient theorem 25, we apply it to two quotient theorems; first, the identity quotient for type  $\beta$ , and second, the partial abstraction quotient  $\langle R, f \rangle$  from theorem 26. This yields the quotient  $\langle \mathbb{S} = \implies R, \mathbb{S} = \dashrightarrow f \rangle$ , since  $\langle \mathbf{I} \Downarrow \mathbb{S} = \rangle = \mathbb{S} =$ . By property 1 of definition 24,  $\forall a. \exists r. (\mathbb{S} = \implies R) \ r \ r \wedge ((\mathbb{S} = \dashrightarrow f) \ r = a)$ . Specializing  $a = \mathbf{I} : \beta \rightarrow \beta$ , and renaming  $r$  as  $g$ , we obtain  $\exists g. (\mathbb{S} = \implies R) \ g \ g \wedge (\mathbb{S} = \dashrightarrow f) \ g = \mathbf{I}$ . By the definitions of  $\implies$ ,  $(\mathbb{S} = \implies R) \ g \ g$  is  $\forall x \ y. x = y \Rightarrow R \ (g \ x) \ (g \ y)$ , which simplifies by the definition of  $R$  to  $\forall y. Q \ (g \ y)$ . The right conjunct is  $(\mathbb{S} = \dashrightarrow f) \ g = \mathbf{I}$ , which by the definition of  $\dashrightarrow$  is  $(\lambda x. \iota \ (\mathbf{IMAGE} \ f \ (\mathbf{IMAGE} \ g \ (\mathbb{S} = \ x)))) = \mathbf{I}$ . But  $\mathbb{S} = \ x$  is the singleton  $\{x\}$ , so since  $\mathbf{IMAGE} \ h \ \{z\} = \{h \ z\}$ ,  $\iota \{z\} = z$ , and  $(\lambda x. f \ (g \ x)) = f \circ g$ , this simplifies to  $f \circ g = \mathbf{I}$ , and the conclusion follows.  $\square$

**Theorem 28 (Existence of Hilbert choice).** *There exists an operator  $c : (\alpha \rightarrow \text{bool}) \rightarrow \alpha$  which obeys the Hilbert choice axiom,  $\forall P x. P x \Rightarrow P (c P)$ .*

**Proof:** In theorem 27, take  $Q = (\lambda(P:\alpha \rightarrow \text{bool}, a:\alpha). (\exists x. P x) \Rightarrow P a)$  and  $f = \text{FST}$ . Then its antecedent is  $\forall P'. \exists(P, a). ((\exists x. P x) \Rightarrow P a) \wedge (\text{FST}(P, a) = P')$ . For any  $P'$ , take  $P = P'$ , and if  $\exists x. P x$ , then take  $a$  to be such an  $x$ . Otherwise take  $a$  to be any value of  $\alpha$ . In either case the antecedent is true. Therefore there exists a function  $g$  such that  $\text{FST} \circ g = \text{I}$  and  $\forall P. Q (g P)$ , which is  $\forall P. (\exists x. (\text{FST} (g P)) x) \Rightarrow (\text{FST} (g P)) (\text{SND} (g P))$ . The operator  $c$  is then taken as  $\text{SND} \circ g$ , and since  $\text{FST} (g P) = P$ , the Hilbert choice axiom follows.  $\square$

The significance of theorem 28 is that even if we are able to avoid all use of the Axiom of Choice up to this point, it is not possible to prove the function conditional quotient theorem 25 without it. This section's design may be used to build a theory of quotients which is constructive and which extends naturally to quotients of lists, pairs, sums, and options. However, it is not possible to extend it to higher order quotients while remaining constructive. Therefore the designs presented in this paper cannot be used to create higher order quotients in strictly constructive theorem provers. Alternatively, in theorem provers like HOL which admit the Hilbert choice operator, if higher order quotients are desired, there is no advantage in avoiding using the Axiom of Choice through using the design of this section. The main design presented earlier is much simpler to automate.

## 7 Lifting Types, Constants, and Theorems

The definition of new types corresponding to the quotients of existing types by equivalence relations is called “lifting” the types from a lower, more representational level to a higher, more abstract level. Both levels describe similar objects, but some details which are apparent at the lower level are no longer visible at the higher level. The logic is simplified.

However, simply forming a new type does not complete the quotient operation. Rather, one wishes to recreate the pre-existing logical environment at the new, higher, and more abstract level. This includes not only the new types, but also new versions of the constants that form and manipulate values of those types, and also new versions of the theorems that describe properties of those constants. All of these form a logical layer, above which all the lower representational details may be safely and forever forgotten.

This can be done in a single call of the main tool of this package.

```
define_quotient_types :
  {types: {name: string,
           equiv: thm} list,
   defs: {def_name: string,
          fname: string,
          func: Term.term,
          fixity: Parse.fixity} list,
   tyop_equivs : thm list,
```

```

    tyop_quotients : thm list,
    tyop_simps : thm list,
    respects : thm list,
    poly_preserves : thm list,
    poly_respects : thm list,
    old_thms : thm list} ->
thm list

```

`define_quotient_types` takes a single argument which is a record with the following fields.

*types* is a list of records, each of which contains two fields: *name*, which is the name of a new quotient type to be created, and *equiv*, which is either 1) a theorem that a binary relation  $R$  is an equivalence relation (see section 4), or 2) a theorem that  $R$  is a nonempty partial equivalence relation, of the form

$$\vdash (\exists x. R x x) \wedge (\forall x y. R x y \Leftrightarrow R x x \wedge R y y \wedge (R x = R y)).$$

*defs* is a list of records specifying the constants to be lifted. Each record contains the following four fields: *func* is an HOL term, which must be a single constant, which is the constant to be lifted. *fname* is the name of the new constant being defined as the lifted version of *func*. *fixity* is the HOL fixity of the new constant being created, as specified in the HOL structure `Parse.def_name` is the name under which the new constant definition is to be stored in the current theory. The process of defining lifted constants is described in §9.

*tyop\_equivs* is a list of conditional equivalence theorems for type operators (see §4.1). These are used for bringing into regular form theorems on new type operators, so that they can be lifted (see sections 11 and 12).

*tyop\_quotients* is a list of conditional quotient theorems for type operators (see §5.2). These are used for lifting both constants and theorems.

*tyop\_simps* is a list of theorems used to simplify type operator relations and map functions, e.g., for pairs,  $\vdash (\$ = \### \$) = \$ =$  and  $\vdash (I \## I) = I$ .

The rest of the arguments refer to the general process of lifting theorems over the quotients being defined, as described in section 10.

*respects* is a list of theorems about the respectfulness of the constants being lifted. These theorems are described in section 10.1.

*poly\_preserves* is a list of theorems about the preservation of polymorphic constants in the HOL logic across a quotient operation. In other words, they state that any quotient operation preserves these constants as a homomorphism. These theorems are described in section 10.2.

*poly\_respects* is a list of theorems showing the respectfulness of the polymorphic constants mentioned in *poly\_preserves*. These are described in §10.3.

*old\_thms* is a list of theorems concerning the lower, representative types and constants, which are to be automatically lifted and proved at the higher, more abstract quotient level. These theorems are described in section 10.4.

`define_quotient_types` returns a list of theorems, which are the lifted versions of the *old\_thms*.

A similar function, `define_quotient_types_rule`, takes a single argument which is a record with the same fields as above except for `old_thms`, and returns an SML function of type `thm -> thm`. This result, typically called `LIFT_RULE`, is then used to lift the old theorems individually, one at a time.

For backwards compatibility with John Harrison's excellent quotients package [9] (which provided much inspiration), the following function is also provided:

```
define_equivalence_type :
  {name: string,
   equiv: thm,
   defs: {def_name: string,
          fname: string,
          func: Term.term,
          fixity: Parse.fixity} list,
   welldefs : thm list,
   old_thms : thm list} ->
  thm list
```

This function is defined in terms of `define_quotient_types` as

```
fun define_equivalence_type {name,equiv,defs,welldefs,old_thms} =
  define_quotient_types
    {types=[{name=name, equiv=equiv}], defs=defs, tyop_equivs=[],
     tyop_quotients=[FUN_QUOTIENT],
     tyop_simps=[FUN_REL_EQ,FUN_MAP_I], respects=welldefs,
     poly_preserves=[FORALL_PRS,EXISTS_PRS],
     poly_respects=[RES_FORALL_RSP,RES_EXISTS_RSP],
     old_thms=old_thms};
```

## 8 New Quotient Type Definitions

In this section we describe how the function `define_quotient_types` creates new quotient types. It automates the reasoning described in section 3, creating the quotient type as a new type in the HOL logic. It also defines the mapping functions between the types, relating them as described in theorems 10 and 11.

The new type is created by the quotient type package by internally making use of the HOL primitive, `new_type_definition`. All definitions are accomplished as definitional extensions of HOL, and thus preserve HOL's consistency.

Before invoking `define_quotient_types`, the user should define a relation on the original type `ty`, and prove that it is either 1) an equivalence relation on the original type `ty`, as described in section 4, as a theorem of the form:

$$\text{R\_EQUIV} \quad |- \quad (\forall x y. R x y \Leftrightarrow (R x = R y))$$

or 2) that the relation is a nonempty partial equivalence relation, of the form

$$\text{R\_EQUIV} \quad |- \quad (\exists x. R x x) \wedge (\forall x y. R x y \Leftrightarrow R x x \wedge R y y \wedge (R x = R y)).$$

Evaluating `define_quotient_types` with the argument field `types` containing the record `{name="tyname", equiv=R_EQUIV}` automatically declares a new type `tyname` in the HOL logic as the quotient type `ty/R`, which we will refer to from here on as `qty`, and declares two new constants `abs : ty->qty` and `rep : qty->ty`, such that the relation `R` with `abs` and `rep` is a quotient as described in section 5. The `define_quotient_types` tool proves the quotient theorem

$$\vdash \text{QUOTIENT } R \text{ abs rep}$$

and stores it in the current theory under the automatically-generated name `tyname_QUOTIENT`.

## 9 Lifting Definitions of Constants

The previous section (§8) dealt with lifting types across a quotient operation. This section deals with lifting constants, including functions, whose types involve the lifted types.

Evaluating `define_quotient_types` with the argument field `defs` containing the record

```
{ def_name = "defname",
  fname    = "fname",
  func     = function,
  fixity   = fixity }
```

automatically declares a new constant `fname` as a lifted version of the existing constant `function`. Here `function` is an HOL term which is a single existing constant of the HOL logic. The new constant `fname` is given the fixity specified as `fixity`; this is typically `Prefix`, but might be, e.g., `Infix(NONASSOC,150)`, or some other fixity as supported by the structure `Parse`. The definition of the new constant is stored in the current theory under the name `defname`.

The theorem which is produced as the definition of the new constant has the same form as the preservation theorems of section 10.2, but without antecedents. After the quotient operation, the definition theorem will not be of further value, as the lifted theorems will be the basis for all further proof efforts.

Here are the values related to fixity from the structure `Parse`:

```
LEFT      : associativity
RIGHT     : associativity
NONASSOC  : associativity

Infixl    : int -> fixity
Infixr    : int -> fixity
Infix     : associativity * int -> fixity
TruePrefix : int -> fixity
Closefix  : fixity
Suffix    : int -> fixity
fixity    : string -> fixity
```

## 10 Lifting Theorems of Properties

Previously we have seen how to lift types, and how to lift constants on those types. This section describes how to lift general theorems on those constants, to restate them in terms of the lifted versions of the constants and prove them correct, given the existing theorems relating the lower versions of the constants.

This turns out to be a substantially more complex process than those previously described for lifting types and functions. Because of its difficulty, the `define_quotient_types` tool automates the process completely, and greatly eases the entire task of forming quotients. In order for the tool to work effectively and exert its full power, several ingredients need to be provided by the user in the form of lists of theorems that describe key properties of the functions used in the theorem to be lifted. These kinds of theorems will be described in detail in the subsections that follow. First we review the arguments used by `define_quotient_types` for lifting theorems.

```
define_quotient_types :
  {types: {name: string, equiv: thm} list,
   defs: {def_name: string, fname: string,
          func: Term.term, fixity: Parse.fixity} list,
   tyop_equivs : thm list,
   tyop_quotients : thm list,
   tyop_simps : thm list,
   respects : thm list,
   poly_preserves : thm list,
   poly_respects : thm list,
   old_thms : thm list} ->
  thm list
```

The last four fields of the argument to `define_quotient_types` are lists of theorems. The last field (`old_thms`) is the list of theorems to be lifted, and the result produced by the tool is the list of the lifted versions of those theorems. The meanings of the other three fields (`respects`, `poly_preserves`, `poly_respects`) are described in the following subsections.

### 10.1 Respectfulness theorems: `respects`

`respects` is a list of theorems demonstrating the respectfulness of all constants used in `old_thms` (except polymorphic operators). These state that for each such constant, considered as a function, the equivalence class of the result yielded depends only on the equivalence classes of the inputs, not on any input's particular value within the class.

Not all functions defined at the lower level respect the equivalence relations involved in the lifting process. Theorems that mention such disrespectful functions cannot in general be lifted. The respectfulness of each function involved must be demonstrated through a theorem of the general form

$$\begin{aligned} &|- \forall(x_1:\gamma_1) (y_1:\gamma_1) \dots (x_n:\gamma_n) (y_n:\gamma_n). \\ &R_1 x_1 y_1 \wedge \dots \wedge R_n x_n y_n \Rightarrow \\ &R_c (C x_1 \dots x_n) (C y_1 \dots y_n) \end{aligned}$$

where the constant  $C$  has type  $\gamma_1 \rightarrow \dots \rightarrow \gamma_n \rightarrow \gamma_c$ , and each relation  $R_i$  has type  $\gamma_i \rightarrow \gamma_i \rightarrow \text{bool}$  for all  $i$ . Depending on the types involved, the partial equivalence relations  $R_1, \dots, R_n, R_c$  may be simple equality, equivalence relations, aggregate partial equivalence relations, or higher-order partial equivalence relations (see section 5.1), as illustrated in examples below.

In fact, in the special case where one of the relations  $R_i$  is simple equality (which happens when the type  $\gamma_i$  is not a type being lifted), the above general form may be simplified, in the following ways. The two variables  $x_i$  and  $y_i$  may be combined into one, say  $(z_i:\gamma_i)$ , e.g. in the list of universally quantified variables. The antecedent conjunct  $R_i x_i y_i$ , which here is  $x_i = y_i$ , may be omitted. In the conclusion  $R_c (C x_1 \dots x_n) (C y_1 \dots y_n)$ , the same variable  $z_i$  may be used in place of both  $x_i$  in the first operand of  $R_c$  and  $y_i$  in the second operand. This simpler version is not completely standard, but may be more convenient for the user to provide. The package will compensate by automatically creating the standard version from this simplified one.

To illustrate this, the following functions are taken from an example of syntactic terms of the untyped lambda calculus, where the term type `term1` is being lifted to a new type `term` by identifying terms that are equivalent according to the equivalence relation `ALPHA: term1 -> term1 -> bool`.

The function `Var1 : var -> term1` is used to construct lambda calculus terms which are single variables. The respectfulness theorem for `Var1` is

$$|- \forall x_1 x_2. (x_1 = x_2) \Rightarrow \text{ALPHA} (\text{Var1 } x_1) (\text{Var1 } x_2)$$

or possibly the simplified (but nonstandard) form

$$|- \forall x. \text{ALPHA} (\text{Var1 } x) (\text{Var1 } x)$$

`Var1` has one argument of type `var`, which type is not lifted, so the partial equivalence relation of this argument is simple equality. The result type is `term1`, so the partial equivalence relation for the result is `ALPHA`.

The function `App1 : term1 -> term1 -> term1` is used to construct terms which are applications of a function to an argument. The respectfulness theorem for `App1` is

$$\begin{aligned} &|- \forall t_1 t_2 u_1 u_2. \\ &\text{ALPHA } t_1 t_2 \wedge \text{ALPHA } u_1 u_2 \Rightarrow \\ &\text{ALPHA} (\text{App1 } t_1 u_1) (\text{App1 } t_2 u_2) \end{aligned}$$

`App1` has two arguments. The first argument has type `term1`, so the partial equivalence relation of this argument is `ALPHA`. The second argument also has type `term1`, so the partial equivalence relation of this argument is also `ALPHA`. The result type is `term1`, so the partial equivalence relation for the result is `ALPHA`.

The function `HEIGHT1 : term1 -> num` is used to calculate the height of a term as a tree. The respectfulness theorem for `HEIGHT1` is

|-  $\forall t1\ t2. \text{ALPHA } t1\ t2 \Rightarrow (\text{HEIGHT1 } t1 = \text{HEIGHT1 } t2)$

`HEIGHT1` has one argument, which has type `term1`, so the partial equivalence relation of the argument is `ALPHA`. The result type is `num`, which is not being lifted, so the partial equivalence relation for the result is simple equality.

The function `FV1 : term1 -> (var -> bool)` is used to calculate the set of free variables of a term. The respectfulness theorem for `FV1` is

|-  $\forall t1\ t2. \text{ALPHA } t1\ t2 \Rightarrow (\text{FV1 } t1 = \text{FV1 } t2)$

`FV1` has one argument, which has type `term1`, so the partial equivalence relation of the argument is `ALPHA`. The result type is `var -> bool`. Even though this is a functional type, it is not affected by lifting, so the partial equivalence relation for the result is simple equality.

The function `<[ : term1 -> (var # term1) list -> term1` is used to simultaneously substitute a list of terms for a corresponding list of variables where they occur free across a target term. The respectfulness theorem for `<[` is

|-  $\forall t1\ t2\ s1\ s2.$   
 $\text{ALPHA } t1\ t2 \wedge \text{LIST\_REL } (\$= \text{### ALPHA})\ s1\ s2 \Rightarrow$   
 $\text{ALPHA } (t1\ <[ s1)\ (t2\ <[ s2)$

`<[` has two arguments. The first argument has type `term1`, so the partial equivalence relation of this argument is `ALPHA`. The second argument is a substitution, which has type `(var # term1) list`, so the partial equivalence relation of this argument is `LIST_REL ($= ### ALPHA)`. The result type is `term1`, so the partial equivalence relation for the result is `ALPHA`.

Please note how the antecedents of each theorem relate to the arguments of the function. The arguments which have types not being lifted are compared by equality, while the arguments which have types being lifted are compared by the corresponding partial equivalence relation. Similarly, the consequent of each of the theorems above is either a partial equivalence or an equality, depending on whether the type of the value returned by the function is of a type being lifted or not. There is a direct one-to-one relationship between the type of the argument/result and the partial equivalence relation used to compare it.

Therefore the antecedent of the theorem on the respectfulness of `Var1` is an equality, since the type of the argument is `var` which is not being lifted, while the antecedents of the theorem on the respectfulness of `App1` are both partial equivalences, since the type of those arguments is `term1`, which is being lifted according to the equivalence relation `ALPHA`. Also, the consequent of the theorem on the respectfulness of `HEIGHT1` is an equality, since the type of the value returned by `HEIGHT1` is `num`, which is not being lifted. If the arguments and the value returned by a function are all of types not being lifted, then there is no need for a respectfulness theorem for that function.

Also, where a function has an argument of an aggregate type, the corresponding partial equivalence relation is created by an aggregate operator. For example, in the theorem on the respectfulness of `<[`, the type of the second argument is `(var # term1) list`. The partial equivalence relation that corresponds to this

type is `LIST_REL ($= ### ALPHA)`, constructed by first using the `###` operator to create the relation for `var # term1`, and then using the `LIST_REL` operator to create the relation for `(var # term1) list`. These partial equivalence relation operators are described in section 5.1.

Whenever there are arguments to the constant, there are multiple equivalent ways to state the respectfulness theorem. For example, the respectfulness theorem for `FV1:term1 -> var -> bool` may be given equally well as any of the following completely equivalent versions:

```
|- ∀t1 t2 x1 x2.
    ALPHA t1 t2 ∧ (x1 = x2) ⇒ (FV1 t1 x1 = FV1 t2 x2)
|- ∀t1 t2. ALPHA t1 t2 ⇒ (FV1 t1 = FV1 t2)
|- (ALPHA ===> $=) FV1 FV1
```

The last version has higher order and lesser arity than the earlier versions. In fact, the three theorems above have arities 2, 1, and 0, respectively, while the last theorem is the only one with a higher order partial equivalence relation. The earlier versions may be easier to understand and prove than the last version. For the quotient package's internal use, all respectfulness theorems are automatically converted to the highest order and lowest arity possible, usually with arity zero.

## 10.2 Preservation of polymorphic functions: `poly_preserves`

`poly_preserves` is a list of theorems expressing the preservation of generic, polymorphic functions across quotient operations. This is expressed as an equality between the value of the function applied to arguments of lifted types, and the lifted version of the value of the same function applied to arguments of the lower types. The equalities are conditioned on the component types being quotients.

These theorems have the following general form.

```
|- ∀R1 (abs1:α1->β1) rep1. QUOTIENT R1 abs1 rep1 ⇒
    ...
    ∀Rn (absn:αn->βn) repn. QUOTIENT Rn absn repn ⇒
    ∀(x1:δ1) ... (xk:δk).
        C x1 ... xk = absc (C' (rep1 x1) ... (repk xk))
```

where

1. the constant `C` has a polymorphic type with  $n$  type variables,  $\alpha_1, \dots, \alpha_n$ ,
2. `C'` is usually `C`, but may be a different constant in case `C` is not preserved,
3. the type of `C` is of the form  $\gamma_1 \rightarrow \dots \rightarrow \gamma_k \rightarrow \gamma_c$   
(`C` is a curried function of  $k$  arguments, with a return value of type  $\gamma_c$ ),  
with all type variables in the  $\gamma_1, \dots, \gamma_k, \gamma_c$  types contained within  $\alpha_1, \dots, \alpha_n$ ,
4.  $\delta_i$  is the lifted version of  $\gamma_i$  for all  $i = 1, \dots, k, c$   
( $\delta_i = \gamma_i[\alpha_j \mapsto \beta_j$  for all  $j = 1, \dots, n]$ ),  
with all type variables in the  $\delta_1, \dots, \delta_k, \delta_c$  types contained within  $\beta_1, \dots, \beta_n$ ,  
and

5.  $abs_i:\gamma_i \rightarrow \delta_i$  and  $rep_i:\delta_i \rightarrow \gamma_i$  are the (possibly identity, aggregate, or higher-order) abstraction and representation functions for the type  $\gamma_i$ , for all  $i = 1, \dots, k, c$ . These are expressions, not simply an `absi` or `repi` variable.

Depending on the types involved, some of the abstraction or representation functions may be the identity function `I`, in which case they disappear, as illustrated in some of the examples below.

For clarity, we will discuss examples for particular polymorphic operators, beginning with simpler ones. In each case, the main driver of the form of the resulting theorem is the the operator's type, in particular the number of arguments, the type of each argument, and the type returned by the operator.

The preservation theorem for `NIL` is

$$\begin{aligned} &|- \forall R \text{ (abs:'a} \rightarrow \text{'b) rep. QUOTIENT R abs rep} \Rightarrow \\ &\quad (\text{[]} = \text{MAP abs []}) \end{aligned}$$

The operator `NIL` has polymorphic type `('a)list`. It has one type variable, `'a`, so  $n = 1$ . It has no arguments, so  $k = 0$ . The result type is `('a)list`, for which the abstraction function is `MAP abs`.

The preservation theorem for `LENGTH` is

$$\begin{aligned} &|- \forall R \text{ (abs:'a} \rightarrow \text{'b) rep. QUOTIENT R abs rep} \Rightarrow \\ &\quad \forall l. \text{LENGTH l} = \text{LENGTH (MAP rep l)} \end{aligned}$$

The operator `LENGTH` has polymorphic type `('a)list -> num`. It has one type variable, `'a`, so  $n = 1$ . It has one argument, so  $k = 1$ . The argument type is `('a)list`, for which the representation function is `MAP rep`. The result type is `num`, for which the abstraction function is `I`, which disappears.

The preservation theorem for `CONS` is

$$\begin{aligned} &|- \forall R \text{ (abs:'a} \rightarrow \text{'b) rep. QUOTIENT R abs rep} \Rightarrow \\ &\quad \forall t \text{ h. h::t} = \text{MAP abs (rep h::MAP rep t)} \end{aligned}$$

The operator `CONS` has polymorphic type `'a -> ('a)list -> ('a)list`. It has one type variable, `'a`, so  $n = 1$ . It has two arguments, so  $k = 2$ . The first argument type is `'a`, for which the representation function is `rep`. The second argument type is `('a)list`, for which the representation function is `MAP rep`. The result type is `('a)list`, for which the abstraction function is `MAP abs`.

The preservation theorem for `FST` is

$$\begin{aligned} &|- \forall R1 \text{ (abs1:'a} \rightarrow \text{'c) rep1. QUOTIENT R1 abs1 rep1} \Rightarrow \\ &\quad \forall R2 \text{ (abs2:'b} \rightarrow \text{'d) rep2. QUOTIENT R2 abs2 rep2} \Rightarrow \\ &\quad \forall p:\text{'c} \# \text{'d. FST p} = \text{abs1 (FST ((rep1 ## rep2) p))} \end{aligned}$$

The operator `FST` has polymorphic type `('a # 'b) -> 'a`. It has two type variables, `'a` and `'b`, so  $n = 2$ . It has one argument, so  $k = 1$ . The argument type is `('a # 'b)`, for which the representation function is `rep1 ## rep2`. The result type is `'a`, for which the abstraction function is `abs1`.

The preservation theorem for the operator `o` is

$$\begin{aligned} &|- \forall R1 \text{ (abs1:'a} \rightarrow \text{'d) rep1. QUOTIENT R1 abs1 rep1} \Rightarrow \\ &\quad \forall R2 \text{ (abs2:'b} \rightarrow \text{'e) rep2. QUOTIENT R2 abs2 rep2} \Rightarrow \\ &\quad \forall R3 \text{ (abs3:'c} \rightarrow \text{'f) rep3. QUOTIENT R3 abs3 rep3} \Rightarrow \\ &\quad \forall f \ g. f \circ g = \\ &\quad \text{(rep1} \rightarrow \text{abs3) ((abs2} \rightarrow \text{rep3) f o (abs1} \rightarrow \text{rep2) g) \end{aligned}$$

The operator `o` has type  $(\text{'b} \rightarrow \text{'c}) \rightarrow (\text{'a} \rightarrow \text{'b}) \rightarrow (\text{'a} \rightarrow \text{'c})$ , which is polymorphic and also higher order. It has three type variables, `'a`, `'b`, and `'c`, so  $n = 3$ . It has two arguments, so  $k = 2$ . The first argument type is `'b`  $\rightarrow$  `'c`, for which the representation function is `abs2`  $\rightarrow$  `rep3`. The second argument type is `'a`  $\rightarrow$  `'b`, for which the representation function is `abs1`  $\rightarrow$  `rep2`. The result type is `'a`  $\rightarrow$  `'c`, for which the abstraction function is `rep1`  $\rightarrow$  `abs3`.

Since the types of these functions are polymorphic, instances of the functions may be applied on arguments of either the types being lifted or the higher, lifted types. In a style very similar to the definition theorems constructed by `define_quotient_types` for the new versions of the constants being lifted, each theorem expresses that the value of the operator applied to arguments of the lifted types is equal to the lifted version of the value of the same operator applied to arguments of the lower types, computed by lowering the original arguments.

So for example, in the `CONS` example above, the `CONS` operator is equal to taking its arguments `h` and `t`, lowering each argument as `rep h` and `MAP rep t`, then applying `CONS` to these two lowered arguments to compute the result at the lower level, and then raising the result to the lifted level by `MAP abs`.

These express the use of each polymorphic function at the lifted level in terms of its use at the lower level.

These theorems differ from the definition theorems for new constants is that each theorem conditions the preservation statement upon the polymorphic types being proper quotients. The conditions follow the form of a quotient theorem as given in section 8, but each describes a quotient for a polymorphic type variable instead of for a specific type. If the function is polymorphic in more than one type, then the theorem will be conditioned on all of the type variables being quotient types.

Thus in the `o` example above, the preservation of the composition operator `o` is conditioned on three types being lifted, where `'a` is being lifted to `'d`, `'b` is being lifted to `'e`, and `'c` is being lifted to `'f`. This calls for three antecedents which are quotient theorems of `'a`, `'b`, and `'c`. In a theorem to be lifted, when the composition operator is actually applied to arguments which are functions from specific domains to specific ranges, the `o` preservation theorem will be instantiated with those types, to be resolved against actual quotient theorems for those types.

A substantial collection of these preservation theorems for various standard polymorphic functions of the HOL logic have been proven already and are available in `quotientTheory` (see Table 1). If there is a need to use a polymorphic function not covered by these, the corresponding preservation theorem can be

proven by the user, using the same approach as for the example theorems above, as shown in `quotientScript.sml`.

Whenever there are arguments to the constant, there are multiple equivalent ways to state the preservation theorem. For example, the consequent of the preservation theorem for `o` may be given equally well as any of the following completely equivalent versions:

```

∀f g x. (f o g) x =
          abs3 (((abs2 --> rep3) f o (abs1 --> rep2) g) (rep1 x))
∀f g. f o g =
          (rep1 --> abs3) ((abs2 --> rep3) f o (abs1 --> rep2) g)
∀f. $o f =
          ((abs1 --> rep2) --> (rep1 --> abs3)) ($o ((abs2 --> rep3) f))
$o = ((abs2 --> rep3) --> (abs1 --> rep2) --> (rep1 --> abs3)) $o

```

In each of the versions, the type of the `$o` on the left hand side of the equality is  $(\text{'e} \rightarrow \text{'f}) \rightarrow (\text{'d} \rightarrow \text{'e}) \rightarrow (\text{'d} \rightarrow \text{'f})$ , while the type of the `$o` on the right hand side is  $(\text{'b} \rightarrow \text{'c}) \rightarrow (\text{'a} \rightarrow \text{'b}) \rightarrow (\text{'a} \rightarrow \text{'c})$ . The last version has higher order and lesser arity than the earlier versions. In fact, the four versions above have arities 3, 2, 1, and 0, respectively. The earlier versions may be easier to understand and prove than the last version. For the quotient package's internal use, all preservation theorems are automatically converted to the lowest order and highest arity possible, and then all higher order versions are produced and used for lifting theorems which involve the operator.

### 10.3 Respectfulness of polymorphic functions: `poly_respects`

`poly_respects` is a list of theorems expressing the respectfulness of generic, polymorphic functions.

Since the types of these functions are polymorphic, instances of the functions may be applied to arguments of the types being lifted. Each theorem expresses that an operator respects the partial equivalence relations involved, just as for the `respects` field (§10.1), except that the theorem conditions the respectfulness of the operator upon the polymorphic types being quotients. The conditioning is the same as that described in the previous section for `poly_preserves` (§10.2).

These theorems have the following general form.

$$\begin{aligned}
&|- \forall R1 \text{ (abs1:}\alpha_1\text{-}\rightarrow\beta_1\text{) rep1. QUOTIENT R1 abs1 rep1} \Rightarrow \\
&\quad \dots \\
&\quad \forall Rn \text{ (absn:}\alpha_n\text{-}\rightarrow\beta_n\text{) repn. QUOTIENT Rn absn repn} \Rightarrow \\
&\quad \forall (\mathbf{x1}:\gamma_1) (\mathbf{y1}:\gamma_1) \dots (\mathbf{xk}:\gamma_k) (\mathbf{yk}:\gamma_k). \\
&\quad R_1 \mathbf{x1} \mathbf{y1} \wedge \dots \wedge R_k \mathbf{xk} \mathbf{yk} \Rightarrow \\
&\quad R_c (\mathbf{C} \mathbf{x1} \dots \mathbf{xk}) (\mathbf{C} \mathbf{y1} \dots \mathbf{yk})
\end{aligned}$$

where

1. the constant `C` has a polymorphic type with  $n$  type variables,  $\alpha_1, \dots, \alpha_n$ ,
2.  $\beta_1, \dots, \beta_n$  are  $n$  other type variables,

**TABLE 1.**  
**Preservation and Respectfulness Theorems for Polymorphic Operators**

Lifted Operators	Preservation Theorems	Respectfulness Theorems
$\forall_{::} \dots \rightarrow \forall_{\dots}$	FORALL_PRS	RES_FORALL_RSP
$\exists_{::} \dots \rightarrow \exists_{\dots}$	EXISTS_PRS	RES_EXISTS_RSP
$\exists!!_{::} \dots \rightarrow \exists!_{\dots}$	EXISTS_UNIQUE_PRS	RES_EXISTS_EQUIV_RSP
$\lambda_{::} \dots \rightarrow \lambda_{\dots}$	ABSTRACT_PRS	RES_ABSTRACT_RSP
COND	COND_PRS	COND_RSP
LET	LET_PRS	LET_RSP
FST	FST_PRS	FST_RSP
SND	SND_PRS	SND_RSP
,	COMMA_PRS	COMMA_RSP
CURRY	CURRY_PRS	CURRY_RSP
UNCURRY	UNCURRY_PRS	UNCURRY_RSP
##	PAIR_MAP_PRS	PAIR_MAP_RSP
INL	INL_PRS	INL_RSP
INR	INR_PRS	INR_RSP
ISL	ISL_PRS	ISL_RSP
ISR	ISR_PRS	ISR_RSP
++	SUM_MAP_PRS	SUM_MAP_RSP
CONS	CONS_PRS	CONS_RSP
NIL	NIL_PRS	NIL_RSP
MAP	MAP_PRS	MAP_RSP
LENGTH	LENGTH_PRS	LENGTH_RSP
APPEND	APPEND_PRS	APPEND_RSP
FLAT	FLAT_PRS	FLAT_RSP
REVERSE	REVERSE_PRS	REVERSE_RSP
FILTER	FILTER_PRS	FILTER_RSP
NULL	NULL_PRS	NULL_RSP
SOME_EL	SOME_EL_PRS	SOME_EL_RSP
ALL_EL	ALL_EL_PRS	ALL_EL_RSP
FOLDL	FOLDL_PRS	FOLDL_RSP
FOLDR	FOLDR_PRS	FOLDR_RSP
NONE	NONE_PRS	NONE_RSP
SOME	SOME_PRS	SOME_RSP
IS_SOME	IS_SOME_PRS	IS_SOME_RSP
IS_NONE	IS_NONE_PRS	IS_NONE_RSP
OPTION_MAP	OPTION_MAP_PRS	OPTION_MAP_RSP
I, K, o, C, W	I_PRS, K_PRS, o_PRS, etc.	I_RSP, K_RSP, o_RSP, etc.
IN	IN_PRS	IN_RSP
EMPTY	EMPTY_PRS	EMPTY_RSP
UNIV	UNIV_PRS	UNIV_RSP
INTER, UNION	INTER_PRS, UNION_PRS	INTER_RSP, UNION_RSP
SUBSETR $\rightarrow$ SUBSET	SUBSET_PRS	SUBSETR_RSP
PSUBSETR $\rightarrow$ PSUBSET	PSUBSET_PRS	PSUBSETR_RSP
IMAGER $\rightarrow$ IMAGE	IMAGE_PRS	IMAGER_RSP

An arrow indicates that in the preservation theorem, the lower operator is different from the higher (*lower*  $\rightarrow$  *higher*), otherwise the operator is unchanged. The respectfulness theorems concern the lower operator. These theorems are found in `quotientTheory`.

3. the type of  $C$  is of the form  $\gamma_1 \rightarrow \dots \rightarrow \gamma_k \rightarrow \gamma_c$   
 $(C$  is a curried function of  $k$  arguments, with a return value of type  $\gamma_c$ ),  
with all type variables in the  $\gamma_1, \dots, \gamma_k, \gamma_c$  types contained within  $\alpha_1, \dots, \alpha_n$ ,  
and
4.  $R_i: \gamma_i \rightarrow \gamma_i \rightarrow \text{bool}$  is the (possibly equality, aggregate, or higher-order)  
partial equivalence relation for the type  $\gamma_i$ , for all  $i = 1, \dots, k, c$ . This is an  
expression, not simply an  $R_i$  variable.

For clarity, we will discuss examples for particular polymorphic operators, beginning with simpler ones. In each case, the main driver of the form of the resulting theorem is the operator's type, in particular the number of arguments, the type of each, and the type returned by the operator.

The respectfulness theorem for `NIL` is

$$\begin{aligned} &|- \forall R \text{ (abs: 'a} \rightarrow \text{'b) rep. QUOTIENT R abs rep} \Rightarrow \\ &\quad \text{LIST\_REL R [] []} \end{aligned}$$

The operator `NIL` has polymorphic type  $(\text{'a})\text{list}$ . It has one type variable,  $\text{'a}$ , so  $n = 1$ . It has no arguments, so  $k = 0$ . The result type is  $(\text{'a})\text{list}$ , for which the partial equivalence relation is `LIST_REL R`.

The respectfulness theorem for `LENGTH` is

$$\begin{aligned} &|- \forall R \text{ (abs: 'a} \rightarrow \text{'b) rep. QUOTIENT R abs rep} \Rightarrow \\ &\quad \forall l1 \ l2. \text{LIST\_REL R l1 l2} \Rightarrow \\ &\quad \quad (\text{LENGTH l1} = \text{LENGTH l2}) \end{aligned}$$

The operator `LENGTH` has polymorphic type  $(\text{'a})\text{list} \rightarrow \text{num}$ . It has one type variable,  $\text{'a}$ , so  $n = 1$ . It has one argument, so  $k = 1$ . The argument type is  $(\text{'a})\text{list}$ , for which the partial equivalence relation is `LIST_REL R`. The result type is `num`, for which the partial equivalence relation is `=`.

The respectfulness theorem for `CONS` is

$$\begin{aligned} &|- \forall R \text{ (abs: 'a} \rightarrow \text{'b) rep. QUOTIENT R abs rep} \Rightarrow \\ &\quad \forall t1 \ t2 \ h1 \ h2. \\ &\quad \quad R \ h1 \ h2 \wedge \text{LIST\_REL R t1 t2} \Rightarrow \\ &\quad \quad \text{LIST\_REL R (h1::t1) (h2::t2)} \end{aligned}$$

The operator `CONS` has polymorphic type  $\text{'a} \rightarrow (\text{'a})\text{list} \rightarrow (\text{'a})\text{list}$ . It has one type variable,  $\text{'a}$ , so  $n = 1$ . It has two arguments, so  $k = 2$ . The first argument type is  $\text{'a}$ , for which the partial equivalence relation is `R`. The second argument type is  $(\text{'a})\text{list}$ , for which the partial equivalence relation is `LIST_REL R`. The result type is  $(\text{'a})\text{list}$ , for which the partial equivalence relation is `LIST_REL R`.

The respectfulness theorem for `FST` is

$$\begin{aligned} &|- \forall R1 \text{ (abs1: 'a} \rightarrow \text{'c) rep1. QUOTIENT R1 abs1 rep1} \Rightarrow \\ &\quad \forall R2 \text{ (abs2: 'b} \rightarrow \text{'d) rep2. QUOTIENT R2 abs2 rep2} \Rightarrow \\ &\quad \forall p1 \ p2. (R1 \ ### \ R2) \ p1 \ p2 \Rightarrow \\ &\quad \quad R1 \ (\text{FST p1}) \ (\text{FST p2}) \end{aligned}$$

The operator `FST` has polymorphic type  $(\text{'a} \# \text{'b}) \rightarrow \text{'a}$ . It has two type variables, `'a` and `'b`, so  $n = 2$ . It has one argument, so  $k = 1$ . The argument type is  $(\text{'a} \# \text{'b})$ , for which the partial equivalence relation is `R1 ### R2`. The result type is `'a`, for which the partial equivalence relation is `R1`.

The respectfulness theorem for the operator `o` is

```
|- ∀R1 (abs1:'a -> 'd) rep1. QUOTIENT R1 abs1 rep1 ⇒
   ∀R2 (abs2:'b -> 'e) rep2. QUOTIENT R2 abs2 rep2 ⇒
   ∀R3 (abs3:'c -> 'f) rep3. QUOTIENT R3 abs3 rep3 ⇒
   ∀f1 f2 g1 g2.
     (R2 ==> R3) f1 f2 ∧ (R1 ==> R2) g1 g2 ⇒
     (R1 ==> R3) (f1 o g1) (f2 o g2)
```

The operator `o` has type  $(\text{'b} \rightarrow \text{'c}) \rightarrow (\text{'a} \rightarrow \text{'b}) \rightarrow (\text{'a} \rightarrow \text{'c})$ , which is polymorphic and also higher order. It has three type variables, `'a`, `'b`, and `'c`, so  $n = 3$ . It has two arguments, so  $k = 2$ . The first argument type is  $\text{'b} \rightarrow \text{'c}$ , for which the partial equivalence relation is `R2 ==> R3`. The second argument type is  $\text{'a} \rightarrow \text{'b}$ , for which the partial equivalence relation is `R1 ==> R2`. The result type is  $\text{'a} \rightarrow \text{'c}$ , for which the partial equivalence relation is `R1 ==> R3`.

Whenever there are arguments to the constant, there are multiple equivalent ways to state the respectfulness theorem. For example, the respectfulness theorem for `o` may be given equally well with any of the following as the main part, after the `QUOTIENT` conditions:

```
... ∀f1 f2 g1 g2 x1 x2.
   (R2 ==> R3) f1 f2 ∧ (R1 ==> R2) g1 g2 ∧ R1 x1 x2 ⇒
   R3 ((f1 o g1) x1) ((f2 o g2) x2)
... ∀f1 f2 g1 g2. (R2 ==> R3) f1 f2 ∧ (R1 ==> R2) g1 g2 ⇒
   (R1 ==> R3) (f1 o g1) (f2 o g2)
... ∀f1 f2. (R2 ==> R3) f1 f2 ⇒
   ((R1 ==> R2) ==> (R1 ==> R3)) ($o f1) ($o f2)
... ((R2 ==> R3) ==> (R1 ==> R2) ==> (R1 ==> R3)) $o $o
```

The last version has higher order and lesser arity than the earlier versions. In fact, the four versions above have arities 3, 2, 1, and 0, respectively. Interestingly, the last version contains no variables (outside of the relation variables). The earlier versions may be easier to understand and prove than the last version. For the quotient package's internal use, all respectfulness theorems are automatically converted to the highest order and lowest arity possible, usually with arity zero.

A substantial collection of these respectfulness theorems for various standard polymorphic functions of the HOL logic have been proven already and are available in `quotientTheory` (see Table 1). If there is a need to use a polymorphic function not covered by these, the corresponding respectfulness theorem can be proven by the user, using the same approach as for the example theorems above, as shown in `quotientScript.sml`.

#### 10.4 Theorems to be lifted: `old_thms`

`old_thms` are the theorems to be lifted. They should involve only constants (including functions) of the lower, original level, with no mention of any lifted constants or functions. The constants involved must respect the equivalence relations involved, as justified by theorems included in `respects` (see section 10.1) and `poly_respects` (see section 10.3). The constants must also be either constants being lifted, or else constants preserved by the quotient, as justified by theorems included in `poly_preserves` (see section 10.2). Each theorem must not have any free variables, and it also must not have any hypotheses, only a conclusion.

### 11 Restrictions on lifting of theorems

This section describes the restrictions needed for theorems to lift; the next section relaxes these somewhat, but properly, theorems should obey these restrictions.

It is important to remember that even if all the necessary information is provided properly, not all theorems of the lower level can be successfully lifted.

To successfully lift, all of the functions a theorem mentions must respect the equivalence relations involved in creating the lifted types. While for the most part properties that are intended to be true at both levels will be expressed in theorems that will lift, there are significant issues that can arise.

The first issue is that the normal equality relation (`=`) between elements of a lower type is a function that does not respect the equivalence relation for that type. This means that theorems that mention the equality of elements of the lower type will not in general lift. Usually the statement of the theorem should be revised, with the equivalence relation substituted for the equality relation; this is a different theorem which will in general require its own proof. Then if the lifting is successful, it will lift to a higher version where the equivalence between lower values has been replaced by equality between higher values.

The second issue is that the universal and existential quantification operators are not in general respectful. In particular, quantification over function types may consider functions that are not respectful. For example, in the lambda calculus example, one theorem to be lifted is the induction theorem:

$$\begin{aligned} &|- \forall P. \\ &\quad (\forall v. P (\text{Var1 } v)) \wedge \\ &\quad (\forall t \text{ t0}. P t \wedge P \text{ t0} \Rightarrow P (\text{App1 } t \text{ t0})) \wedge \\ &\quad (\forall t. P t \Rightarrow \forall v. P (\text{Lam1 } v t)) \Rightarrow \\ &\quad (\forall t. P t) \end{aligned}$$

In this theorem the variable `P` has type `term1 -> bool`. This variable is universally quantified, so all possible functions of this type are considered as possible values of `P`. Unfortunately, some functions of this type do not respect the alpha-equivalence of terms. This respectfulness would be expressed as

$$\forall t1 \text{ t2}. \text{ALPHA } t1 \text{ t2} \Rightarrow (P t1 = P t2)$$

But this is not true of all functions of the type `term1 -> bool`. For example, consider the particular function `P1` defined by structural recursion as

$$\begin{aligned} (\forall x. \quad P1 \text{ (Var1 } x) &= F) \wedge \\ (\forall t \ u. P1 \text{ (App1 } t \ u) &= P1 \ t \vee P1 \ u) \wedge \\ (\forall x \ u. P1 \text{ (Lam1 } x \ u) &= P1 \ u \vee (x = \text{"a"})) \end{aligned}$$

Then `P1 t` is true if and only if the term `t` contains a subexpression of the form `Lam1 "a" u`, where the bound variable is `"a"`. This `P1` does not satisfy the respectfulness principle since, for example, `ALPHA (Lam1 "a" (Var1 "a")) (Lam1 "b" (Var1 "b"))` but then `P1 (Lam1 "a" (Var1 "a"))` is true while `P1 (Lam1 "b" (Var1 "b"))` is false.

The point is that if such non-respectful functions are considered as possible values of `P`, then it becomes impossible to lift the theorem, and in fact in general it would be unsound to do so. The higher version of the theorem will describe quantification over functions of the higher types, and these functions correspond only to respectful functions of the lower types. Non-respectful functions at the lower level have **no** corresponding function at the higher level. So the statement of the theorem needs to be revised to quantify only over functions which are respectful. For the example above, the theorem needs to be rephrased as

$$\begin{aligned} |- \forall P :: \text{respects (ALPHA ==> \$=)} . \\ (\forall v. P \text{ (Var1 } v)) \wedge \\ (\forall t \ t0 :: \text{respects ALPHA. } P \ t \wedge P \ t0 \Rightarrow P \text{ (App1 } t \ t0)) \wedge \\ (\forall t :: \text{respects ALPHA. } P \ t \Rightarrow \forall v. P \text{ (Lam1 } v \ t)) \Rightarrow \\ (\forall t :: \text{respects ALPHA. } P \ t) \end{aligned}$$

This notation uses the restricted quantifier notation  $\forall x :: \text{restriction. body}$ , the same as `RES_FORALL restriction ( $\lambda x. \text{body}$ )`, introduced in the `res_quan` library. It also uses the `respects` operator, defined in `quotientTheory` as

$$\text{respects } R \ (x : 'a) = R \ x \ x$$

The rephrased induction theorem states that the variable `P` is universally quantified over all functions of type `term1 -> bool` such that those functions respect alpha-equivalence on their domain `term1` (and equality on their range, `bool`), i.e., `respects (ALPHA ==> $=) P`, which is equal to

$$\forall t, t_0 : \text{term1. ALPHA } t \ t_0 \Rightarrow (P \ t = P \ t_0).$$

Then the revised version of the induction theorem can be successfully lifted.

In general, the  $\forall$  and  $\exists$  operators should be replaced by the `RES_FORALL` and `RES_EXISTS` operators, with `respects R` as the restriction, where `R` is the partial equivalence relation for the type being quantified over. This replacement applies even if `R` is simply an equivalence relation, where of course `R x x` always holds by reflexivity, and so no elements of the quantification are excluded. However, when `R` is simple equality, as for types not being lifted, no replacement of  $\forall$  or  $\exists$  is necessary. `RES_FORALL` and `RES_EXISTS` can be expressed more conveniently using the “ $\forall\_::\_::\_$ ” and “ $\exists\_::\_::\_$ ” restriction notation, as shown above.

Finally, on occasion we wish to lift a theorem to create a higher version that contains a unique existence quantifier ( $\exists!$ ). Such a theorem states that for one and only one instance of the quantified variable, a property holds. But that single element at the higher level corresponds to an entire equivalence class of elements at the lower level. To lift some lower theorem to such a statement, the lower theorem must state that the property holds just and only for elements of that equivalence class, rather than for some single element. Therefore one cannot simply lift from a lower version of  $\exists!$  to a higher version of  $\exists!$ . Instead, we must introduce a new operator, `RES_EXISTS_EQUIV` of type `('a -> 'a -> bool) -> ('a -> bool) -> bool`.

```
RES_EXISTS_EQUIV R P =
  ( $\exists x :: \text{respects } R. P x$ )  $\wedge$ 
  ( $\forall x y :: \text{respects } R. P x \wedge P y \Rightarrow R x y$ )
```

The first argument `R` is the partial equivalence relation for the type being quantified over, and the second argument `P` is the predicate which is being stated as true for some element of that type which respects `R`. `RES_EXISTS_EQUIV R P` means that there exist some elements which are respectful of `R` and which satisfy `P`, and all such elements are equivalent according to `R`.

For convenience, `RES_EXISTS_EQUIV` can also be represented using a new restricted quantification binder,  $\exists!!$ . The parser will translate  $\exists!!x :: R. P x$  into `RES_EXISTS_EQUIV R ( $\lambda x. P x$ )`, and the prettyprinter will reverse this. In order to use this notation, in each HOL session one must first execute

```
val _ = associate_restriction ("?!!", "RES_EXISTS_EQUIV");
```

When attempting to lift a theorem, all instances of quantifying by  $\exists!$  over types being lifted should be replaced by instances of  $\exists!!$  (`RES_EXISTS_EQUIV`) along with the appropriate partial equivalence relation. Instances of quantifying by  $\exists!$  over types not being lifted need not be modified at all. Note that where the restricted quantifier versions of  $\forall$  and  $\exists$  use restrictions of the form `respects R`, the restricted quantifier version of  $\exists!!$  uses just `R` as the restriction.

For example, this arises naturally in the function existence theorem proposed by Gordon and Melham for the lambda calculus [8]:

```
|-  $\forall \text{var app abs.}$ 
   $\exists! \text{hom.}$ 
  ( $\forall x. \text{hom (Var } x) = \text{var } x$ )  $\wedge$ 
  ( $\forall t u. \text{hom (App } t u) = \text{app (hom } t) (\text{hom } u)$ )  $\wedge$ 
  ( $\forall x u. \text{hom (Lam } x u) = \text{abs } (\lambda y. \text{hom } (u <[ [(x, \text{Var } y)]])$ )
```

To create the above theorem, we need to prove the proper lower version

```
|-  $\forall \text{var app abs.}$ 
   $\exists!! \text{hom} :: (\text{ALPHA} ==> \$=)$ .
  ( $\forall x. \text{hom (Var1 } x) = \text{var } x$ )  $\wedge$ 
  ( $\forall t u :: \text{respects ALPHA.}$ 
```

$$\begin{aligned} \text{hom (App1 t u)} &= \text{app (hom t) (hom u)} \wedge \\ (\forall x (u :: \text{respects ALPHA}). \\ \text{hom (Lam1 x u)} &= \text{abs } (\lambda y. \text{hom (u <[ [(x, \text{Var1 y})])})) \end{aligned}$$

which will then lift.

## 12 Support for non-standard lifting

The `define_quotient_types` tool for lifting theorems is actually less demanding and more forgiving than has been described up to this point. Automation has been added to recognize several common situations of theorems which may not be in the proper form, but are strong enough to imply the proper form. These are quietly converted and then lifted.

Despite the objections to the use of equality in theorems to be lifted mentioned above, in practice equality is commonly used. Many theorems to be lifted have the form  $\vdash a = b$ , for some expressions  $a$  and  $b$  whose type is being lifted. In fact, if  $\sim$  is an equivalence relation, this equality implies  $\vdash a \sim b$ . The tool recognizes this common case and automatically proves the needed revised theorem that only mentions equivalence instead of equality. This can then be lifted.

Similarly, theorems of the form  $\vdash P \Rightarrow (a = b)$ , or even more general examples, can also be automatically revised and then lifted.

Universal or existential restricted quantification, where the restriction is of the form `respects R` for  $R$  an equivalence relation, do not actually restrict any elements from the quantification. Thus these are really equal to the original ordinary quantification, and the tool is able to create and prove the version with such restricted quantifications from a user-supplied theorem with normal quantification in those places.

In addition, theorems which are universal quantifications at the outer level of the theorem, may imply the restricted universal quantifications (their proper form) over respectful values. The proper form is automatically proven and then lifted. For example, the tool can lift the lambda calculus induction theorem given earlier without the user first converting it to proper form.

Finally, theorems which involve unique existence quantification  $\exists!$  restricted over functions which are respectful, may imply the corresponding theorem using the restricted  $\exists!!$  operator. For example, in the lambda calculus example, we have proven at the lower level

$$\begin{aligned} \vdash \forall \text{var app abs}. \\ \exists! \text{hom} :: \text{respects (ALPHA ==> \$=)}. \\ (\forall x. \text{hom (Var1 x)} = \text{var x}) \wedge \\ (\forall t u. \text{hom (App1 t u)} = \text{app (hom t) (hom u)}) \wedge \\ (\forall x u. \text{hom (Lam1 x u)} = \text{abs } (\lambda y. \text{hom (u <[ [(x, \text{Var1 y})])})) \end{aligned}$$

The tool will first automatically prove the proper version:

$$\begin{aligned} \vdash \forall \text{var app abs}. \\ \exists!! \text{hom} :: \text{ALPHA ==> \$=}. \end{aligned}$$

```

(∀x. hom (Var1 x) = var x) ∧
(∀t u :: respects ALPHA.
 hom (App1 t u) = app (hom t) (hom u)) ∧
(∀x (u :: respects ALPHA).
 hom (Lam1 x u) = abs (λy. hom (u <[ [(x,Var1 y)]]))

```

and then lift the proper version to the desired higher version of this theorem:

```

|- ∀var app abs.
  ∃!hom.
    (∀x. hom (Var x) = var x) ∧
    (∀t u. hom (App t u) = app (hom t) (hom u)) ∧
    (∀x u. hom (Lam x u) = abs (λy. hom (u <[ [(x,Var y)]]))

```

In general, of course, this revising may not work. If the tool cannot automatically prove the proper version from the theorem it is given, it will print out an error message showing the needed revised proper form. The user should prove the proper version manually and then submit it to the tool for lifting.

Finally, we would like to describe some possible reasons to consider when theorems do not lift, apart from improper form as above. First, every constant mentioned in the theorem that takes or returns values of types being lifted must be described in a respectfulness theorem, whether in the `respects` or `poly_respects` arguments. Also, every such constant in the theorem must be either being lifted to a new constant in the `defs` argument, or described in a preservation theorem in the `poly_preserves` argument. Every partial equivalence relation mentioned in any of these theorems should be either one of those mentioned in the `types` argument, or an extension of these, using the relation operators mentioned in sections 4 and 5. If the type of any constant in any theorem involves type operators, like lists or functions, then the associated conditional quotient theorems must be provided in the `tyop_quotients` argument. Likewise, the conditional equivalence theorems (if available), and the associated relation and map function simplification theorems should be provided in the `tyop_equivs` and `tyop_simps` arguments. The error message associated with the exception thrown may help localize the error. The process of the quotient operation may be observed in more detail by setting the variable `quotient.chatting := true`. Lastly, during development it may be more convenient to use the `define_quotient_types_rule` tool, so that the `LIFT_RULE` function it returns may be applied to candidate lower theorems individually and repeatedly.

### 13 The Sigma Calculus

The untyped sigma calculus was introduced by Abadi and Cardelli in *A Theory of Objects* [1]. It highlights the concept of objects, rather than functions.

We will use the sigma calculus as an example to demonstrate the quotient package tools. We will first define an initial or “pre-” version of the language syntax, and then create the refined or “pure” version by performing a quotient operation on the initial version.

The pre-sigma calculus contains terms denoting objects and methods. We define the sets of object terms  $O_1$  and method terms  $M_1$  inductively as

- (1)  $x \in O_1$  for all variables  $x$ ;
- (2)  $m_1, \dots, m_n \in M_1 \Rightarrow [l_1 = m_1, \dots, l_n = m_n] \in O_1$  for all labels  $l_1, \dots, l_n$ ;
- (3)  $a \in O_1 \Rightarrow a.l \in O_1$  for all labels  $l$ ;
- (4)  $a \in O_1 \wedge m \in M_1 \Rightarrow a.l \Leftarrow m \in O_1$  for all labels  $l$ ;
- (5)  $a \in O_1 \Rightarrow \zeta(x)a \in M_1$  for all variables  $x$ .

$[l_1 = m_1, \dots, l_n = m_n]$  denotes a *method dictionary*, as a finite list of *entries*, each  $l_i = m_i$  consisting of a label and a method. There should be no duplicates among the labels, but if there are, the first one takes precedence.

The form  $a.l$  denotes the invocation of the method labelled  $l$  in the object  $a$ . The form  $a.l \Leftarrow m$  denotes the update of the object  $a$ , where the method labelled  $l$  (if any) is replaced by the new method  $m$ . The form  $\zeta(x)a$  denotes a method with one formal parameter,  $x$ , and a body  $a$ .  $\zeta$  is a binder, like  $\lambda$  in the lambda calculus.  $x$  is a bound variable, and the scope of  $x$  is the body  $a$ . In this scope,  $x$  represents the “self” parameter, the object itself which contains this method.

Given the pre-sigma calculus, we define the pure sigma calculus by identifying object and method terms which are alpha-equivalent [2]. Thus in the pure sigma calculus,  $\zeta(x)x.l_1 = \zeta(y)y.l_1$ ,  $[l_1 = \zeta(x)x] = [l_1 = \zeta(y)y]$ , et cetera. This is accomplished by forming the quotients of the types of pre-sigma calculus object and method terms by their alpha-equivalence relations. Thus  $O = O_1 / \equiv_\alpha^o$  and  $M = M_1 / \equiv_\alpha^m$ , where  $\equiv_\alpha^o$  and  $\equiv_\alpha^m$  are the respective alpha-equivalence relations.

## 14 The Pre-Sigma Calculus in HOL

Hol98 supports the definition of new nested mutually recursive types by the `Hol_datatype` function in the `bossLib` library.

The syntax of the pre-sigma calculus is defined as follows.

```
val _ = Hol_datatype

(* obj1 ::= x | [li=mi] i in 1..n | a.l | a.l:=m *)
  ' obj1 = OVAR1 of var
    | OBJ1 of (string # method1) list
    | INVOKE1 of obj1 => string
    | UPDATE1 of obj1 => string => method1 ;

(* method1 ::= sigma(x)a *)
  method1 = SIGMA1 of var => obj1 ' ;
```

This creates the new mutually recursive types `obj1` and `method1`, and also the constructor functions

```

OVAR1      : var -> obj1
OBJ1       : (string # method1) list -> obj1
INVOKE1    : obj1 -> string -> obj1
UPDATE1    : obj1 -> string -> method1 -> obj1
SIGMA1     : var -> obj1 -> method1

```

It also creates associated theorems for induction, function existence, and one-to-one and distinctiveness properties of the constructors.

The definition above goes beyond simple mutual recursion of types, to involve what is called “nested recursion,” where a type being defined may appear deeply nested under type operators such as `list`, `prod`, or `sum`. In the above definition, in the line defining the `OBJ1` constructor function, the type `method1` is nested, first as the right part of a pair type, and then as the element type of a list type.

The `Hol_definition` tool automatically compensates for this complexity, creating in effect *four* new types, not simply two. It is as if the tool created the intermediate types

```

entry1     = string # method1
dict1      = (entry1)list

```

except that these types are actually formed by the `prod` and `list` type operators, not by creating new types. It turns out that when defining mutually recursive functions on these types, there must be *four* related functions defined simultaneously, one for each of the types `obj1`, `dict1`, `entry1`, and `method1`. Similarly, when proving theorems about these functions, one must use mutually recursive structural induction, where the goal has four parts, one for each of the types.

Now we will construct the pure sigma calculus from the pre-sigma calculus.

## 15 The Pure Sigma Calculus in HOL

We here define the pure sigma calculus in HOL.

Let us assume that we have defined alpha-equivalence relations for each of the two types `obj1` and `method1`, called `ALPHA_obj` and `ALPHA_method`, and that we have proven the equivalence theorems for these,

```

ALPHA_obj_EQUIV:
  |- ∀x y. ALPHA_obj x y = (ALPHA_obj x = ALPHA_obj y)
ALPHA_method_EQUIV:
  |- ∀x y. ALPHA_method x y = (ALPHA_method x = ALPHA_method y)

```

We specify the constants that are to be lifted:

```

- val defs = [{def_name="OVAR_def", fname="OVAR",
               func= (--'OVAR1'--), fixity=Prefix},
              {def_name="OBJ_def", fname="OBJ",
               func= (--'OBJ1'--), fixity=Prefix},
              {def_name="INVOKE_def", fname="INVOKE",
               func= (--'INVOKE1'--), fixity=Prefix},

```

```

    func= (--'INVOKE1'--), fixity=Prefix},
  {def_name="UPDATE_def", fname="UPDATE",
    func= (--'UPDATE1'--), fixity=Prefix},
  {def_name="SIGMA_def", fname="SIGMA",
    func= (--'SIGMA1'--), fixity=Prefix},
  {def_name="HEIGHT_def", fname="HEIGHT",
    func= (--'HEIGHT1'--), fixity=Prefix},
  {def_name="FV_def", fname="FV",
    func= (--'FV1'--), fixity=Prefix},
  {def_name="SUB_def", fname="SUB",
    func= (--'SUB1'--), fixity=Prefix},
  {def_name="FV_subst_def", fname="FV_subst",
    func= (--'FV_subst1'--), fixity=Prefix},
  {def_name="SUBo_def", fname="SUBo",
    func= (--'SUB1o :^obj -> ^subs -> ^obj'--),
    fixity=Infix(NONASSOC,150)},
  {def_name="SUBd_def", fname="SUBd",
    func= (--'SUB1d :^dict -> ^subs -> ^dict'--),
    fixity=Infix(NONASSOC,150)},
  {def_name="SUBe_def", fname="SUBe",
    func= (--'SUB1e :^entry -> ^subs -> ^entry'--),
    fixity=Infix(NONASSOC,150)},
  {def_name="SUBm_def", fname="SUBm",
    func= (--'SUB1m :^method -> ^subs -> ^method'--),
    fixity=Infix(NONASSOC,150)},
  {def_name="vsubst_def", fname="/",
    func= (--'$//'--), fixity=Infix(NONASSOC,150)},
  ...
];

```

We specify the respectfulness theorems to assist the lifting:

```

val respects =
  [OVAR1_RSP, OBJ1_RSP, INVOKE1_RSP, UPDATE1_RSP, SIGMA1_RSP,
    HEIGHT_obj1_RSP, HEIGHT_dict1_RSP, HEIGHT_entry1_RSP,
    HEIGHT_method1_RSP, FV_obj1_RSP, FV_dict1_RSP, FV_entry1_RSP,
    FV_method1_RSP, SUB1_RSP, FV_subst_RSP, vsubst1_RSP,
    SUBo_RSP, SUBd_RSP, SUBe_RSP, SUBm_RSP,
    ... ]

```

We specify the polymorphic preservation theorems to assist the lifting:

```

val polyprs = [BV_subst_PRS, COND_PRS, CONS_PRS, NIL_PRS,
  COMMA_PRS, FST_PRS, SND_PRS,
  LET_PRS, o_PRS, UNCURRY_PRS,
  FORALL_PRS, EXISTS_PRS,
  EXISTS_UNIQUE_PRS, ABSTRACT_PRS];

```

We specify the polymorphic conditional respectfulness theorems to assist the lifting:

```
val polyrsp = [BV_subst_RSP, COND_RSP, CONS_RSP, NIL_RSP,
              COMMA_RSP, FST_RSP, SND_RSP,
              LET_RSP, o_RSP, UNCURRY_RSP,
              RES_FORALL_RSP, RES_EXISTS_RSP,
              RES_EXISTS_EQUIV_RSP, RES_ABSTRACT_RSP];
```

The old theorems to be lifted are too many to list, but some will be shown later.

```
- val old_thms = [...];
```

We now define the pure sigma calculus types `obj` and `method`, and lift all constants and theorems:

```
- val new_thms =
  define_quotient_types
    {types = [{name = "obj",    equiv = ALPHA_obj_EQUIV},
              {name = "method", equiv = ALPHA_method_EQUIV}],
    defs = defs,
    tyop_equivs = [LIST_EQUIV, PAIR_EQUIV],
    tyop_quotients = [LIST_QUOTIENT, PAIR_QUOTIENT,
                      FUN_QUOTIENT],
    tyop_simps = [LIST_REL_EQ, LIST_MAP_I,
                  PAIR_REL_EQ, PAIR_MAP_I,
                  FUN_REL_EQ,  FUN_MAP_I],
    respects = respects,
    poly_preserves = polyprs,
    poly_respects = polyrsp,
    old_thms = old_thms};
```

The tool is able to lift many theorems to the abstract, quotient level. Here is the original definition of substitution on a variable:

```
|- (∀y. SUB1 [] y = OVAR1 y) ∧
   (∀y x c s. SUB1 ((x,c)::s) y = (if y = x then c else SUB1 s y))
```

This theorem lifts to its abstract version:

```
|- (∀y. SUB [] y = OVAR y) ∧
   (∀y x c s. SUB ((x,c)::s) y = (if y = x then c else SUB s y))
```

Note how the different constants lift, e.g., `SUB1` to `SUB`, `OVAR1` to `OVAR`. Also note that the quantification of `c:obj1` has now become of `obj`. What may not be so obvious is how the polymorphic operators lift to the abstract versions of themselves: `[]`, `,`, `::`, `if...then...else`. In fact, though the operators look the same, all the types have changed from the lower to the higher versions. Proving

this lifted theorem took considerable automation, hidden behind the simplicity of the result. In addition, the original theorem was not regular; before lifting, it actually was first quietly converted to:

```
|- (∀y. ALPHA_obj (SUB1 [] y) (OVAR1 y)) ∧
    (∀x (c :: respects ALPHA_obj)
      (s :: respects (LIST_REL ($= ### ALPHA_obj))).
      ALPHA_obj (SUB1 ((x,c)::s) y) (if y = x then c else SUB1 s y))
```

Similarly, here is a version of this definition which uses the `let...in` form:

```
|- (∀p s y.
    SUB1 (p::s) y =
      (let (x,c) = p in (if y = x then c else SUB1 s y))) ∧
    (∀y. SUB1 [] y = OVAR1 y)
```

This lifts to the following abstract version:

```
|- (∀p s y.
    SUB (p::s) y =
      (let (x,c) = p in (if y = x then c else SUB s y))) ∧
    (∀y. SUB [] y = OVAR y)
```

In addition to the previous issues, this involves the `LET` operator, which is higher-order, taking a function as an argument. Furthermore, because the `let` involves a pair, the pair is implemented by the `UNCURRY` operator, which is also higher-order. The following regular version was quietly proven and then lifted:

```
|- (∀(p :: respects ($= ### ALPHA_obj))
    (s :: respects (LIST_REL ($= ### ALPHA_obj))) y.
    ALPHA_obj (SUB1 (p::s) y)
      (LET
        (UNCURRY
          (λx (c::respects ALPHA_obj).
            (if y = x then c else SUB1 s y))) p)) ∧
    (∀y. ALPHA_obj (SUB1 [] y) (OVAR1 y))
```

One of the most difficult theorems to lift is the induction theorem, because it is second-order, as it involves quantification over predicates.

```
|- ∀P0 P1 P2 P3.
    (∀v. P0 (OVAR1 v)) ∧ (∀l. P2 l ⇒ P0 (OBJ1 l)) ∧
    (∀o'. P0 o' ⇒ ∀s. P0 (INVOKE1 o' s)) ∧
    (∀o' m. P0 o' ∧ P1 m ⇒ ∀s. P0 (UPDATE1 o' s m)) ∧
    (∀o'. P0 o' ⇒ ∀v. P1 (SIGMA1 v o')) ∧ P2 [] ∧
    (∀p l. P3 p ∧ P2 l ⇒ P2 (p::l)) ∧
    (∀m. P1 m ⇒ ∀s. P3 (s,m)) ⇒
    (∀o'. P0 o') ∧ (∀m. P1 m) ∧ (∀l. P2 l) ∧ (∀p. P3 p)
```

This lifts to the abstract version:

```

|-  $\forall P0\ P1\ P2\ P3.$ 
  ( $\forall v. P0\ (OVAR\ v)$ )  $\wedge$  ( $\forall l. P2\ l \Rightarrow P0\ (OBJ\ l)$ )  $\wedge$ 
  ( $\forall o'. P0\ o' \Rightarrow \forall s. P0\ (INVOKE\ o'\ s)$ )  $\wedge$ 
  ( $\forall o'\ m. P0\ o' \wedge P1\ m \Rightarrow \forall s. P0\ (UPDATE\ o'\ s\ m)$ )  $\wedge$ 
  ( $\forall o'. P0\ o' \Rightarrow \forall v. P1\ (SIGMA\ v\ o')$ )  $\wedge$   $P2\ []$   $\wedge$ 
  ( $\forall p\ l. P3\ p \wedge P2\ l \Rightarrow P2\ (p::l)$ )  $\wedge$ 
  ( $\forall m. P1\ m \Rightarrow \forall s. P3\ (s,m)$ )  $\Rightarrow$ 
  ( $\forall o'. P0\ o'$ )  $\wedge$  ( $\forall m. P1\ m$ )  $\wedge$  ( $\forall l. P2\ l$ )  $\wedge$  ( $\forall p. P3\ p$ )

```

Note how the quantifications over  $P0:obj1 \rightarrow bool$ , etc. lift to quantification over  $P0:obj \rightarrow bool$ , etc. The following regular version was quietly proven and then lifted:

```

|-  $\forall (P0::respects\ (ALPHA\_obj\ ==>\ \$=))$ 
  ( $P1::respects\ (ALPHA\_method\ ==>\ \$=)$ )
  ( $P2::respects\ (LIST\_REL\ (\$= \###\ ALPHA\_method)\ ==>\ \$=)$ )
  ( $P3::respects\ ((\$= \###\ ALPHA\_method)\ ==>\ \$=)$ ).
  ( $\forall v. P0\ (OVAR1\ v)$ )  $\wedge$ 
  ( $\forall l::respects\ (LIST\_REL\ (\$= \###\ ALPHA\_method))$ ).
  ( $P2\ l \Rightarrow P0\ (OBJ1\ l)$ )  $\wedge$ 
  ( $\forall o'::respects\ ALPHA\_obj. P0\ o' \Rightarrow \forall s. P0\ (INVOKE1\ o'\ s)$ )  $\wedge$ 
  ( $\forall (o'::respects\ ALPHA\_obj)\ (m::respects\ ALPHA\_method)$ ).
  ( $P0\ o' \wedge P1\ m \Rightarrow \forall s. P0\ (UPDATE1\ o'\ s\ m)$ )  $\wedge$ 
  ( $\forall o'::respects\ ALPHA\_obj. P0\ o' \Rightarrow \forall v. P1\ (SIGMA1\ v\ o')$ )  $\wedge$ 
   $P2\ []$   $\wedge$ 
  ( $\forall (p::respects\ (\$= \###\ ALPHA\_method))$ 
    ( $l::respects\ (LIST\_REL\ (\$= \###\ ALPHA\_method))$ )).
  ( $P3\ p \wedge P2\ l \Rightarrow P2\ (p::l)$ )  $\wedge$ 
  ( $\forall m::respects\ ALPHA\_method. P1\ m \Rightarrow \forall s. P3\ (s,m)$ )  $\Rightarrow$ 
  ( $\forall o'::respects\ ALPHA\_obj. P0\ o'$ )  $\wedge$ 
  ( $\forall m::respects\ ALPHA\_method. P1\ m$ )  $\wedge$ 
  ( $\forall l::respects\ (LIST\_REL\ (\$= \###\ ALPHA\_method))$ ).  $P2\ l$ )  $\wedge$ 
  ( $\forall p::respects\ (\$= \###\ ALPHA\_method)$ ).  $P3\ p$ )

```

Finally, the most difficult theorem to lift is the function existence theorem, after the style proposed by Gordon and Melham [8]. This uses higher-order unique existence quantification, where the unique existence is not of a simple function, but a tuple of four functions, involving a higher-order partial equivalence relation of tuples of functions. This relation is *not* an equivalence relation.

```

|-  $\forall var$ 
  ( $obj::respects\ (\$= ==>\ LIST\_REL\ (\$= \###\ ALPHA\_method)\ ==>\ \$=)$ )
  ( $inv::respects\ (\$= ==>\ ALPHA\_obj\ ==>\ \$= ==>\ \$=)$ )
  ( $upd::respects$ 
    ( $\$= ==>\ \$= ==>\ ALPHA\_obj\ ==>\ \$= ==>\ ALPHA\_method\ ==>\ \$=)$ )
  ( $cns::respects\ (\$= ==>\ \$= ==>\ (\$= \###\ ALPHA\_method)\ ==>$ 
     $LIST\_REL\ (\$= \###\ ALPHA\_method)\ ==>\ \$=)$ )

```

```

nil (par::respects($= ==> $= ==> ALPHA_method ==> $=))
(sgm::respects($= ==> ($= ==> ALPHA_obj) ==> $=)).
  ∃!(hom_o,hom_d,hom_e,hom_m)::respects
    ((ALPHA_obj ==> $=) ###
     (LIST_REL ($= ### ALPHA_method) ==> $=) ###
     (($= ### ALPHA_method) ==> $=) ###
     (ALPHA_method ==> $=)).
  (∀x. hom_o (OVAR1 x) = var x) ∧
  (∀d. hom_o (OBJ1 d) = obj (hom_d d) d) ∧
  (∀a l. hom_o (INVOKE1 a l) = inv (hom_o a) a l) ∧
  (∀a l m. hom_o (UPDATE1 a l m) =
    upd (hom_o a) (hom_m m) a l m) ∧
  (∀e d. hom_d (e::d) = cns (hom_e e) (hom_d d) e d) ∧
  (hom_d [] = nil) ∧
  (∀l m. hom_e (l,m) = par (hom_m m) l m) ∧
  (∀x a. hom_m (SIGMA1 x a) =
    sgm (λy. hom_o (a <[ [(x,OVAR1 y)]]))
      (λy. a <[ [(x,OVAR1 y)]]))

```

This lifts to the abstract version:

```

|- ∀var obj inv upd cns nil par sgm.
  ∃!(hom_o,hom_d,hom_e,hom_m).
    (∀x. hom_o (OVAR x) = var x) ∧
    (∀d. hom_o (OBJ d) = obj (hom_d d) d) ∧
    (∀a l. hom_o (INVOKE a l) = inv (hom_o a) a l) ∧
    (∀a l m. hom_o (UPDATE a l m) =
      upd (hom_o a) (hom_m m) a l m) ∧
    (∀e d. hom_d (e::d) = cns (hom_e e) (hom_d d) e d) ∧
    (hom_d [] = nil) ∧
    (∀l m. hom_e (l,m) = par (hom_m m) l m) ∧
    (∀x a. hom_m (SIGMA x a) =
      sgm (λy. hom_o (a SUBo [(x,OVAR y)]))
        (λy. a SUBo [(x,OVAR y)]))

```

Note how the restricted unique existence quantification over  $(\text{hom\_o} : \text{obj1} \rightarrow 'a, \dots)$  lifts to unique existence quantification over  $(\text{hom\_o} : \text{obj} \rightarrow 'a, \dots)$ . To accomplish this, the following regular version was quietly proven and then lifted:

```

|- ∀var
  (obj::respects($= ==> LIST_REL ($= ### ALPHA_method) ==> $=))
  (inv::respects($= ==> ALPHA_obj ==> $=))
  (upd::respects
    ($= ==> $= ==> ALPHA_obj ==> $= ==> ALPHA_method ==> $=))
  (cns::respects($= ==> $= ==> ($= ### ALPHA_method) ==>
    LIST_REL ($= ### ALPHA_method) ==> $=))
  nil

```

```

(par::respects ($= ==> $= ==> ALPHA_method ==> $=))
(sgm::respects ($= ==> ($= ==> ALPHA_obj) ==> $=)).
  ∃!!(hom_o,hom_d,hom_e,hom_m)::
    (ALPHA_obj ==> $=) ###
    (LIST_REL ($= ### ALPHA_method) ==> $=) ###
    (($= ### ALPHA_method) ==> $=) ###
    (ALPHA_method ==> $=).
(∀x. hom_o (OVAR1 x) = var x) ∧
(∀d::respects (LIST_REL ($= ### ALPHA_method))).
  hom_o (OBJ1 d) = obj (hom_d d) d) ∧
(∀(a::respects ALPHA_obj) l.
  hom_o (INVOKE1 a l) = inv (hom_o a) a l) ∧
(∀(a::respects ALPHA_obj) l (m::respects ALPHA_method).
  hom_o (UPDATE1 a l m) =
    upd (hom_o a) (hom_m m) a l m) ∧
(∀(e::respects ($= ### ALPHA_method))
  (d::respects (LIST_REL ($= ### ALPHA_method))).
  hom_d (e::d) = cns (hom_e e) (hom_d d) e d) ∧
(hom_d [] = nil) ∧
(∀l (m::respects ALPHA_method).
  hom_e (l,m) = par (hom_m m) l m) ∧
(∀x (a::respects ALPHA_obj).
  hom_m (SIGMA1 x a) =
    sgm (λy. hom_o (a <[ [(x,OVAR1 y)]])
      (λy. a <[ [(x,OVAR1 y)]])

```

This automatic higher order lifting was not available before this package.

Now we have  $SIGMA\ x\ (OVAR\ x) = SIGMA\ y\ (OVAR\ y)$ , etc., as true equality within the HOL logic, as intended. This accomplishes the creation of the pure sigma calculus by identifying alpha-equivalent terms.

## 16 Conclusions

We have implemented a package for mechanically defining higher-order quotient types which is a conservative, definitional extension of the HOL logic. The package automatically lifts not only types, but also constants and theorems from the original level to the quotient level.

This involves the use of partial equivalence relations, as symmetric and transitive but not necessarily reflexive on all elements of their domains. This is necessary to treat higher-order quotients, where elements should not be compared by reflexive relations.

The relationship between the lower type and the quotient type is characterized by the partial equivalence relation, the abstraction function, and the representation function. Three necessary properties have been identified for these to properly describe a quotient, which are preserved in the creation of aggregate

and higher order quotients. Most normal polymorphic operators both respect and are preserved across such quotients, including higher order quotients.

Quotients are useful in a variety of contexts, as shown in the literature. For example, the syntax of programming languages may be modelled as recursive types. Terms which are alpha-equivalent may be identified by taking quotients. This eases the problem of capture of bound variables.

Some systems are convenient to model initially at one level of granularity, but have properties which are only true at a level with less detail. For example, the pure sigma calculus is Church-Rosser, while the pre-sigma calculus is not.

Such quotients may now be more easily modeled within a theorem prover, using the package described here.

*Soli Deo Gloria.*

## References

1. Abadi, M., Cardelli, L.: *A Theory of Objects*. Springer-Verlag 1996.
2. Barendregt, H.P.: *The Lambda Calculus, Syntax and Semantics*. North-Holland, 1981.
3. Bruce, K., Mitchell, J. C.: ‘PER models of subtyping, recursive types and higher-order polymorphism’, in *Principles of Programming Languages 19*, Albuquerque, New Mexico, 1992, pp. 316-327.
4. Chichi, L., Pottier, L., Simpson C.: ‘Mathematical Quotients and Quotient Types in Coq’, Proceedings of *TYPES 2002*, Lecture Notes in Computer Science, vol. 2646 (Springer-Verlag, 2002).
5. Enderton, H. B.: *Elements of Set Theory*. Academic Press, 1977.
6. Geuvers, H., Pollack, R., Wiekijk, F., Zwanenburg, J.: ‘A constructive algebraic hierarchy in Coq’, in *Journal of Symbolic Computation*, 34(4), 2002, pp. 271-286.
7. Gordon, M. J. C., Melham, T. F.: *Introduction to HOL*. Cambridge University Press, Cambridge, 1993.
8. Gordon, A. D., Melham, T. F.: ‘Five Axioms of Alpha Conversion’, in *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs’96*, edited by J. von Wright, J. Grundy and J. Harrison, Lecture Notes in Computer Science, vol. 1125 (Springer-Verlag, 1996), pp. 173-190.
9. Harrison, J.: *Theorem Proving with the Real Numbers*, §2.11, pp. 33-37. Springer-Verlag 1998.
10. Hofmann, M.: ‘A simple model for quotient types,’ in *Typed Lambda Calculus and Applications*, Lecture Notes in Computer Science, vol. 902 (Springer-Verlag, 1995), pp. 216-234.
11. Leisenring, A. C.: *Mathematical Logic and Hilbert’s  $\varepsilon$ -Symbol*. Gordon and Breach, 1969.
12. Moore, G. H.: *Zermelo’s Axiom of Choice: It’s Origins, Development, and Influence*. Springer-Verlag 1982.
13. Nipkow, T., Paulson, L.C., Wenzel, M.: *Isabelle/HOL*. Springer-Verlag 2002.
14. Owre, S., Shankar, N.: *Theory Interpretations in PVS*, Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001.
15. Nogin, A.: ‘Quotient Types: A Modular Approach,’ in *Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002*, edited by V. A. Carreño, C. Muñoz, and S. Tahar, Lecture Notes in Computer Science, vol. 2410 (Springer-Verlag, 2002), pp. 263-280.

16. Paulson, L.: 'Defining Functions on Equivalence Classes,' *ACM Transactions on Computational Logic*, in press. Previously issued as Report, Computer Lab, University of Cambridge, April 20, 2004.
17. Robinson, E.: 'How Complete is PER?', in *Fourth Annual Symposium on Logic in Computer Science (LICS)*, 1989, pp. 106-111.
18. Slotosch, O.: 'Higher Order Quotients and their Implementation in Isabelle HOL', in *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLS'97*, edited by Elsa L. Gunter and Amy Felty, Lecture Notes in Computer Science, vol. 1275 (Springer-Verlag, 1997), pp. 291-306.