Higher Order Quotients in Higher Order Logic

Peter V. Homeier

U. S. Department of Defense
palantir@trustworthytools.com
http://www.trustworthytools.com

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.

Traditionally, quotients have found many applications. Classic examples are the construction of the integers from pairs of non-negative natural numbers, or the rationals from pairs of integers. In the lambda calculus [2] 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. The first to appear was Ton Kalker's 1989 package for HOL88 [11]. Isabelle/HOL [14] has mechanical support for the creation of higher order quotients by Oscar Slotosch [19], using partial equivalence relations represented as a type class, with equivalence relations as a subclass. That system provides a definitional framework for establishing quotient types, including higher order. Independently, Larry Paulson has shown a construction of first-order quotients in Isabelle without any use of the Hilbert choice operator [17]. PVS uses quotients to support theory interpretations [15]. MetaPRL has quotients in its foundations, as a type with a new equality [16]. Coq, based on the Calculus of Constructions [10], supports quotients [6] but has some difficulties with higher order [4]. These systems provide little automatic support. In particular, there is no automatic lifting of constants or theorems.

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, a theorem stating that the original group was abelian 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 mathematically incisive.

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. 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* [18], as a possibly non-reflexive variant of equivalence relations, enabling 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.

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 quotient 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. Additional operators can also be lifted by proving and including the corresponding respectfulness and preservation theorems, which justify the operator's use across quotients.

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 entailing a quotient of the type operator, given quotients of the argument types.

The structure of this paper is as follows. In section 2 we briefly review quotient sets. In section 5 we re-interpret this idea for type theory. Section 3 discusses equivalence relations and their aggregates. Section 4 describes partial equivalence relations, their extension for aggregate and function types, the properties of a quotient relationship, and quotient extension theorems for lifting aggregate and function types. Section 7 describes an alternative design that avoids use of the Axiom of Choice. Section 8 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 9 describes the definition of new quotient types, and section 10 describes the definition of lifted versions of constants. Section 11 describes the various input theorems needed by the tool to automatically lift theorems. Section 12 describes the restrictions on theorems in order for them to lift properly, and section 13 loosens these restrictions, helpfully attempting to lift even some improper theorems. Section 14 discusses lifting theorems about sets. In sections 15 through 17, 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 18.

We are grateful for the helpful comments and suggestions made by Rob Arthan, 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.

reflexive:	$\forall x \in S.$	$x \sim x$
symmetric:	$\forall x, y \in S.$	$x \sim y \;\Rightarrow\; y \sim x$
transitive:	$\forall x, y, z \in S.$	$x \sim y \;\wedge\; y \sim z \;\Rightarrow\; x \sim z$

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

$$[x]_\sim = [y]_\sim \quad \Leftrightarrow \quad x \sim y.$$

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 Equivalence Relations and Equivalence Theorems

Before considering quotients, we examine equivalence relations, on which such traditional quotients as those mentioned in the introduction have been based.

Let τ be any type. A binary relation R on τ can be represented in HOL as a curried function of type $\tau \to (\tau \to \texttt{bool})$. We will take advantage of the curried nature of R, where $R\ x\ y = (R\ x)\ y$.

An equivalence relation is a binary relation E satisfying

reflexivity:	$\forall x : \tau.$	$E\ x\ x$
symmetry:	$\forall x\ y : \tau.$	$E\ x\ y \ \Rightarrow\ E\ y\ x$
transitivity:	$\forall x\ y\ z : \tau.$	$E\ x\ y\ \wedge\ E\ y\ z\ \Rightarrow\ E\ x\ z$

These three properties are encompassed in the *equivalence property:*

$$\textit{equivalence:} \qquad \texttt{EQUIV}\ E \stackrel{\text{def}}{=} \forall x\ y : \tau.\ E\ x\ y\ \Leftrightarrow\ (E\ x = E\ y)$$

A theorem of the form $\vdash \texttt{EQUIV}\ E$ is called an *equivalence theorem* on type τ.

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
```

3.1 Equivalence Extension Theorems

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

Type	Operator	Type of operator
list	`LIST_REL :`	`('a -> 'a -> bool) -> 'a list -> 'a list -> bool`
pair	`### :`	`('a -> 'a -> bool) -> ('b -> 'b -> bool) -> 'a # 'b -> 'a # 'b -> bool`
sum	`+++ :`	`('a -> 'a -> bool) -> ('b -> 'b -> bool) -> 'a + 'b -> 'a + 'b -> bool`
option	`OPTION_REL :`	`('a -> 'a -> bool) -> 'a option -> 'a option -> bool`

These operators are defined as indicated below.

Definition 1.
$$
\begin{array}{llll}
\texttt{LIST\_REL}\ R & [] & [] & = \textbf{true} \\
\texttt{LIST\_REL}\ R\ (a\texttt{::}as) & [] & & = \textbf{false} \\
\texttt{LIST\_REL}\ R & [] & (b\texttt{::}bs) & = \textbf{false} \\
\texttt{LIST\_REL}\ R\ (a\texttt{::}as) & (b\texttt{::}bs) & & = R\ a\ b\ \wedge \texttt{LIST\_REL}\ R\ as\ bs
\end{array}
$$

Definition 2. $(R_1\ \texttt{\#\#\#}\ R_2)\ (a_1, a_2)\ (b_1, b_2)\ =\ R_1\ a_1\ b_1\ \wedge\ R_2\ a_2\ b_2$

Definition 3.
$$
\begin{array}{l}
(R_1\ \texttt{+++}\ R_2)\ (\texttt{INL}\ a_1)\ (\texttt{INL}\ b_1)\ =\ R_1\ a_1\ b_1 \\
(R_1\ \texttt{+++}\ R_2)\ (\texttt{INL}\ a_1)\ (\texttt{INR}\ b_2)\ =\ \textbf{false} \\
(R_1\ \texttt{+++}\ R_2)\ (\texttt{INR}\ a_2)\ (\texttt{INL}\ b_1)\ =\ \textbf{false} \\
(R_1\ \texttt{+++}\ R_2)\ (\texttt{INR}\ a_2)\ (\texttt{INR}\ b_2)\ =\ R_2\ a_2\ b_2
\end{array}
$$

Definition 4.
$$
\begin{array}{llll}
\texttt{OPTION\_REL}\ R & \texttt{NONE} & \texttt{NONE} & = \textbf{true} \\
\texttt{OPTION\_REL}\ R & (\texttt{SOME}\ a) & \texttt{NONE} & = \textbf{false} \\
\texttt{OPTION\_REL}\ R & \texttt{NONE} & (\texttt{SOME}\ b) & = \textbf{false} \\
\texttt{OPTION\_REL}\ R & (\texttt{SOME}\ a) & (\texttt{SOME}\ b) & = R\ a\ b
\end{array}
$$

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 §11.1, 11.3).

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 equivalence extension theorems for type operators (see section 3.1).

Equivalence extension theorems for type operators have the form:

$$
\begin{array}{l}
\vdash \forall \texttt{E1 ... En.} \\
\quad (\forall (\texttt{x}:\alpha_1)\ (\texttt{y}:\alpha_1).\ \texttt{E1 x y} \Leftrightarrow (\texttt{E1 x = E1 y})) \Rightarrow \ldots \\
\quad (\forall (\texttt{x}:\alpha_n)\ (\texttt{y}:\alpha_n).\ \texttt{En x y} \Leftrightarrow (\texttt{En x = En y})) \Rightarrow \\
\quad (\forall (\texttt{x}:(\alpha_1, ..., \alpha_n)op)\ (\texttt{y}:(\alpha_1, ..., \alpha_n)op). \\
\quad\quad \texttt{OP\_REL E1 ... En x y} \Leftrightarrow \\
\quad\quad\ (\texttt{OP\_REL E1 ... En x = OP\_REL E1 ... En y}))
\end{array}
$$

Given the type operator $(\alpha_1, ..., \alpha_n)op$, `OP_REL` should be an operator which takes n arguments, which are the equivalence relations `E1` through `En` on the types α_1 through α_n, yielding an equivalence relation for the type $(\alpha_1, ..., \alpha_n)op$.

Using the above relation extension operators, the aggregate type operators `list`, `prod`, `sum`, and `option` have the following equivalence extension theorems:

LIST_EQUIV: $\vdash \forall E.$ EQUIV $E \Rightarrow$ EQUIV (LIST_REL E)
PAIR_EQUIV: $\vdash \forall E_1\ E_2.$ EQUIV $E_1 \Rightarrow$ EQUIV $E_2 \Rightarrow$ EQUIV (E_1 ### E_2)
SUM_EQUIV: $\vdash \forall E_1\ E_2.$ EQUIV $E_1 \Rightarrow$ EQUIV $E_2 \Rightarrow$ EQUIV (E_1 +++ E_2)
OPTION_EQUIV: $\vdash \forall E.$ EQUIV $E \Rightarrow$ EQUIV (OPTION_REL E)

4 Partial Equivalence Relations and Quotient Theorems

In this section we introduce a new definition of the quotient relationship, based on *partial equivalence relations* (PERs), related to but different from 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 11.1. For example, given an equivalence relation $E : \tau \to \tau \to$ `bool`, the set of functions from τ to τ 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. Suppose instead that $f^?$ did lift. Let $\lceil\phi\rceil$ be the lifted version of ϕ. As $\lceil f^?\rceil$ is the lifted version of $f^?$, it should act just like $f^?$ on its argument, except that it should not consider the lower level details that E disregards. Thus $\forall u.\ \lceil f^?\rceil\lceil u\rceil = \lceil f^?\ u\rceil$. Then certainly $\forall u\, v.\ E\ u\ v \Leftrightarrow (\lceil u\rceil = \lceil v\rceil)$, and because $E\ x\ y$, we must have $\lceil x\rceil = \lceil y\rceil$. Applying $\lceil f^?\rceil$ to both sides, $\lceil f^?\rceil\lceil x\rceil = \lceil f^?\rceil\lceil y\rceil$. But this implies $\lceil f^?\ x\rceil = \lceil f^?\ y\rceil$, which means that $E\ (f^?\ x)\ (f^?\ y)$, which we have said is false, a contradiction. Therefore such disrespectful functions cannot be lifted, and we must exclude them. Using partial equivalence relations accomplishes this exclusion.

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

Definition 5 (Quotient). *A relation R with abstraction function abs and representation function rep (between the representation, lower type τ and the abstract, quotient type ξ) is a quotient (notated as $\langle R, abs, rep\rangle$) if and only if*

(1) $\forall a : \xi.\ abs\ (rep\ a) = a$
(2) $\forall a : \xi.\ R\ (rep\ a)\ (rep\ a)$
(3) $\forall r, s : \tau.\ R\ r\ s \Leftrightarrow R\ r\ r \wedge R\ s\ s \wedge (abs\ r = abs\ s)$

Property 1 states that *rep* is a right inverse of *abs*.

Property 2 states that the range of *rep* respects R.

Property 3 states that two elements of τ 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 τ and the quotient type ξ. 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

$$\vdash \texttt{QUOTIENT}\ R\ \ abs\ \ rep\ \ .$$

A theorem of this form is called a *quotient theorem*. The identity is $\vdash \langle \texttt{\$=}, \texttt{I}, \texttt{I} \rangle$.

These three properties support the inference of a quotient theorem for a function type, given quotient theorems for the domain and the range. This key inference is central and necessary to enable higher order quotients.

5 Quotient Types

The user may specify a quotient of a type τ by a relation R (written τ/R) by giving either a theorem that the relation is an equivalence relation, of the form

$$\vdash \forall x\ y.\ R\ x\ y\ \ \Leftrightarrow\ \ (R\ x = R\ y)\ , \tag{1}$$

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

$$\vdash (\exists x.\ R\ x\ x)\ \wedge\ (\forall x\ y.\ R\ x\ y\ \ \Leftrightarrow\ \ R\ x\ x \wedge R\ y\ y \wedge (R\ x = R\ y))\ . \tag{2}$$

Alternatively, these theorems may be equivalently expressed as $\vdash \texttt{EQUIV}\ R$ or $\vdash \texttt{PARTIAL\_EQUIV}\ R$, respectively, defined in `EQUIV_def` and `PARTIAL_EQUIV_def`. In this section we will develop the second, more difficult case (2). The first follows immediately. In the following, x, y, r, $s : \tau$, $c : \tau \rightarrow \texttt{bool}$, and $a : \tau/R$.

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.

Definition 6. *We define the new quotient type τ/R as isomorphic to the subset of the representing type $\tau \to \texttt{bool}$ by the predicate $P : (\tau \to \texttt{bool}) \to \texttt{bool}$, where $P\ c \stackrel{\text{def}}{=} \exists x.\ R\ x\ x \wedge (c = R\ x)$.*

P is nonempty because $P\ (R\ x)$ for the $x : \tau$ such that $R\ x\ x$ by (2). Let $\xi = \tau/R$. The HOL tool `define_new_type_bijections` [7] automatically defines a function $abs_c : (\tau \to \texttt{bool}) \to \xi$ and its right inverse $rep_c : \xi \to (\tau \to \texttt{bool})$ satisfying

Definition 7. (a) $\forall a : \xi.\ abs_c\ (rep_c\ a) = a$
(b) $\forall c : \tau \to \texttt{bool}.\ P\ c \iff rep_c\ (abs_c\ c) = c$

PER classes are subsets of τ (of type $\tau \to \texttt{bool}$) which satisfy P. Then abs_c and rep_c map between the quotient type ξ and PER classes (hence the "c").

Lemma 8 (rep_c maps to PER classes). $\forall a.\ P\ (rep_c\ a)$.

Proof: By Definition 7(a), $abs_c\ (rep_c\ a) = a$, so taking the rep_c of both sides, $rep_c\ (abs_c\ (rep_c\ a)) = rep_c\ a$. By Definition 7(b), $P\ (rep_c\ a)$. □

Lemma 9. $\forall r.\ R\ r\ r \Rightarrow (rep_c\ (abs_c\ (R\ r)) = R\ r)$.

Proof: Assume $R\ r\ r$; then $P\ (R\ r)$. By Definition 7(b), the goal follows.

Lemma 10 (abs_c is one-to-one on PER classes).
$\forall r\ s.\ R\ r\ r \Rightarrow R\ s\ s \Rightarrow (abs_c\ (R\ r) = abs_c\ (R\ s) \iff R\ r = R\ s)$.

Proof: Assume $R\ r\ r$ and $R\ s\ s$. The right-to-left implication of the biconditional is trivial. Assume $abs_c\ (R\ x) = abs_c\ (R\ y)$. Applying rep_c to both sides gives us $rep_c\ (abs_c\ (R\ x)) = rep_c\ (abs_c\ (R\ y))$. Then by Lemma 9 twice, $R\ x = R\ y$. □

The functions abs_c and rep_c map between PER classes of type $\tau \to \texttt{bool}$ and the quotient type ξ. Using these functions, we can define new functions abs and rep between the original type τ and the quotient type ξ as follows.

Definition 11 (Quotient abstraction and representation functions).

$$\begin{array}{ll} abs : \tau \to \xi & abs\ r \stackrel{\text{def}}{=} abs_c\ (R\ r) \\ rep : \xi \to \tau & rep\ a \stackrel{\text{def}}{=} \texttt{\$@}\ (rep_c\ a) \quad (\ = \ \texttt{@}r.\ rep_c\ a\ r\ ) \end{array}$$

The `@` operator is a higher order version of Hilbert's choice operator ε [7,12]. It has type $(\alpha \to \texttt{bool}) \to \alpha$, and is usually used as a binder, where $\texttt{\$@}\ P = \texttt{@}x.\ P\ x$. (The `$` converts an operator to prefix syntax.) `@` satisfies the HOL axiom $\forall P\ x.\ P\ x \Rightarrow P\ (\texttt{\$@}\ P)$. Given any predicate P on a type, if any element of the type satisfies the predicate, then $\texttt{\$@}\ P$ returns an arbitrary element of that type which satisfies P. If no element of the type satisfies P, then $\texttt{\$@}\ P$ will return simply some arbitrary, unknown element of the type. Such definitions have been questioned by constructivist critics of the Axiom of Choice. An alternative design for quotients avoiding the Axiom of Choice is described in section 7.

Lemma 12. $\forall r.\ R\ r\ r \Rightarrow (R\ (\$\texttt{@}\ (R\ r)) = R\ r)$.

Proof: The axiom for the `@` operator is $\forall P\ x.\ P\ x \Rightarrow P\ (\$\texttt{@}\ P)$. Taking $P = R\ r$ and $x = r$, we have $R\ r\ r \Rightarrow R\ r\ (\$\texttt{@}\ R\ r)$. Assuming $R\ r\ r$, $R\ r\ (\$\texttt{@}\ (R\ r))$ follows. Then by (2), $R\ r\ (\$\texttt{@}\ (R\ r))$ implies the equality $R\ r = R\ (\$\texttt{@}\ (R\ r))$. □

Theorem 13. $\forall a.\ abs\ (rep\ a) = a$

Proof: By Lemma 8 and the definition of P, for each a there exists an r such that $R\ r\ r$ and $rep_c\ a = R\ r$. Then by Lemma 12, $R\ (\$\texttt{@}\ (R\ r)) = R\ r$. Now by Definition 11, $abs\ (rep\ a) = abs_c\ (R\ (\$\texttt{@}\ (rep_c\ a)))$, which simplifies by the above and Definition 7(a) to a. □

Theorem 14. $\forall a.\ R\ (rep\ a)\ (rep\ a)$.

Proof: As before, for each a there exists an r such that $R\ r\ r$ and $rep_c\ a = R\ r$.

$$\begin{array}{rcll}
R\ (rep\ a)\ (rep\ a) & \Leftrightarrow & R\ (\$\texttt{@}\ (rep_c\ a))\ (\$\texttt{@}\ (rep_c\ a)) & \text{Definition 11} \\
& \Leftrightarrow & R\ (\$\texttt{@}\ (R\ r))\ (\$\texttt{@}\ (R\ r)) & \text{selection of } r \\
& \Leftrightarrow & R\ r\ (\$\texttt{@}\ (R\ r)) & \text{Lemma 12} \\
& \Leftrightarrow & R\ (\$\texttt{@}\ (R\ r))\ r & \text{symmetry of } R \\
& \Leftrightarrow & R\ r\ r\ \Leftrightarrow\ T & \text{Lemma 12, selection of } r
\end{array}$$

□

Theorem 15. $\forall r\ s.\ \ R\ r\ s\ \Leftrightarrow\ R\ r\ r\ \wedge\ R\ s\ s\ \wedge\ (abs\ r = abs\ s)$

Proof:
$$\begin{array}{rcll}
R\ r\ s & \Leftrightarrow & R\ r\ r\ \wedge\ R\ s\ s\ \wedge\ (R\ r = R\ s) & (2) \\
& \Leftrightarrow & R\ r\ r\ \wedge\ R\ s\ s\ \wedge\ (abs_c\ (R\ r) = abs_c\ (R\ s)) & \text{Lemma 10} \\
& \Leftrightarrow & R\ r\ r\ \wedge\ R\ s\ s\ \wedge\ (abs\ r = abs\ s) & \text{Definition 11}
\end{array}$$

□

Theorem 16. $\langle R,\ abs,\ rep\rangle$.

Proof: By Theorems 13, 14, and 15, with Definition 5. □

6 Aggregate and Higher Order Quotient Theorems

Traditional quotients that lift τ to a set of τ also lift lists of τ to sets of lists of τ. These sets are isomorphic to lists, but *they are not lists.* In this design, when τ is lifted to ξ, then we lift lists of τ to lists of ξ. We preserve the type operator structure built on top of the types being lifted. Similarly, we want to preserve polymorphic constants. Thus we need not create a new type for each lifted version of lists, but simply reuse the existing list type operator, now applied to ξ. This preserves the type structure and enables direct use of the polymorphic constants of that type operator, such as `HD` for lists. In a theorem being lifted, we want an occurrence of $\texttt{HD} : \tau\ \texttt{list} \rightarrow \tau$ to lift to an occurrence of $\texttt{HD} : \xi\ \texttt{list} \rightarrow \xi$. If such a constant is not lifted to itself, the lifted version of the theorem will not look like the original. Hence Definition 5 was intentionally designed to preserve the vital type operator structure.

At times one wishes to not only lift a number of types across a quotient operation, but also lift by extension a number of other types which are dependent on the first set. For example, if we lift the type of terms of the lambda calculus across alpha-equivalence, then we would also expect that the types of lists or pairs involving the lifted terms would follow naturally.

In fact these do follow; one merely has to apply the type operator, say `list`, to the lifted type `term` to produce the type of lists of lifted terms `(term)list`. All of the theorems about lists in general now apply to lists of lifted terms, and all of the theorems about lifted terms apply to the elements of these lifted lists.

In the process of lifting constants and theorems, quotient theorems are needed for each argument and result type of each constant being lifted. For aggregate and higher order types, the tool automatically proves any needed quotient theorems from the available quotient theorems for the constituent subtypes. To accomplish this, the tool uses *quotient extension theorems* (section 6.2). These are provided preproven for some standard type operators. For others, new quotient extension theorems may be manually proven and then included to extend the tool's power.

6.1 Aggregate and Higher Order PERs and Map Operators

Some aggregate equivalence relation operators have been already described in section 3, 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	`===>` :	`('a -> 'a -> bool) -> ('b -> 'b -> bool) -> ('a -> 'b) -> ('a -> 'b) -> bool`

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

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

$\vdash$ `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 any 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,

$\vdash$ `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 created for aggregate types involve not only aggregate partial equivalence relations, 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 *abs* and *rep* fns
list	`MAP` :	`('a -> 'b) -> 'a list -> 'b list` *examples:* (`MAP` *abs*) , (`MAP` *rep*)

pair	`##` :	`('a -> 'b) -> ('c -> 'd) ->`
		`'a # 'c -> 'b # 'd`
		examples: (abs_1 `##` abs_2) , (rep_1 `##` rep_2)

sum	`++` :	`('a -> 'b) -> ('c -> 'd) ->`
		`'a + 'c -> 'b + 'd`
		examples: (abs_1 `++` abs_2) , (rep_1 `++` rep_2)

option	`OPTION_MAP` :	`('a -> 'b) -> 'a option -> 'b option` *examples:* (`OPTION_MAP` *abs*) , (`OPTION_MAP` *rep*)

fun	`-->` :	`('a -> 'b) -> ('c -> 'd) ->`
		`('b -> 'c) -> 'a -> 'd`
		examples: (rep_1 `-->` abs_2) , (abs_1 `-->` rep_2)

The definitions of the above operators are indicated below. They are created either in the quotient package or in standard HOL.

Definition 18. $\texttt{MAP}\ f\ \ \texttt{[]}\ = \texttt{[]}$
$\texttt{MAP}\ f\ (a\texttt{::}as)\ =\ (f\ a)\ \texttt{::}\ (\texttt{MAP}\ f\ as)$

Definition 19. $(f_1\ \texttt{\#\#}\ f_2)\ (a_1,\ a_2)\ =\ (f_1\ a_1,\ f_2\ a_2)$

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

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

The function map operator definition is of special interest:

Definition 22. $(f\ \texttt{-->}\ g)\ h\ x\ \stackrel{\text{def}}{=}\ g\ (h\ (f\ x))$.

MAP and OPTION_MAP are prefix operators, and the others are infix. The identity quotient map operator is the identity operator $\texttt{I} : \alpha \to \alpha$.

These map operators are inserted automatically in the quotient theorems created for extended types. Each resulting quotient theorem establishes that the extended type's partial equivalence relation, abstraction function, and representation function properly relate together to form a quotient of the extended type.

6.2 Quotient Extension Theorems

Here are the quotient extension theorems for the list, prod, sum, option, and, most significantly, fun type operators:

LIST_QUOTIENT:
$\vdash\ \forall R\ abs\ rep.\ \langle R, abs, rep\rangle \Rightarrow \langle \texttt{LIST\_REL}\ R,\ \ \texttt{MAP}\ abs,\ \ \texttt{MAP}\ rep\rangle$

PAIR_QUOTIENT:
$\vdash\ \forall R_1\ abs_1\ rep_1.\ \langle R_1, abs_1, rep_1\rangle \Rightarrow\ \ \forall R_2\ abs_2\ rep_2.\ \langle R_2, abs_2, rep_2\rangle \Rightarrow$
$\langle R_1\ \texttt{\#\#\#}\ R_2,\ \ abs_1\ \texttt{\#\#}\ abs_2,\ \ rep_1\ \texttt{\#\#}\ rep_2\rangle$

SUM_QUOTIENT:
$\vdash\ \forall R_1\ abs_1\ rep_1.\ \langle R_1, abs_1, rep_1\rangle \Rightarrow\ \ \forall R_2\ abs_2\ rep_2.\ \langle R_2, abs_2, rep_2\rangle \Rightarrow$
$\langle R_1\ \texttt{+++}\ R_2,\ \ abs_1\ \texttt{++}\ abs_2,\ \ rep_1\ \texttt{++}\ rep_2\rangle$

OPTION_QUOTIENT:
$\vdash\ \forall R\ abs\ rep.\ \langle R, abs, rep\rangle \Rightarrow$
$\langle \texttt{OPTION\_REL}\ R,\ \ \texttt{OPTION\_MAP}\ abs,\ \ \texttt{OPTION\_MAP}\ rep\rangle$

FUN_QUOTIENT:
$\vdash\ \forall R_1\ abs_1\ rep_1.\ \langle R_1, abs_1, rep_1\rangle \Rightarrow\ \ \forall R_2\ abs_2\ rep_2.\ \langle R_2, abs_2, rep_2\rangle \Rightarrow$
$\langle R_1\ \texttt{===>}\ R_2,\ \ rep_1\ \texttt{-->}\ abs_2,\ \ abs_1\ \texttt{-->}\ rep_2\rangle$

This last theorem is of central and critical importance to forming higher order quotients. We present here its proof in detail.

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 5:

Property 1. Prove for all a, $(rep_1 \texttt{-->} abs_2)\ ((abs_1 \texttt{-->} rep_2)\ a) = a$.
Proof: The equality here is between functions, and by extension, true if for all values x in a's domain, $(rep_1 \texttt{-->} abs_2)\ ((abs_1 \texttt{-->} rep_2)\ a)\ x = a\ x$.
By the definition of `-->`, this is $abs_2\ ((abs_1 \texttt{-->} rep_2)\ a\ (rep_1\ x)) = a\ x$, 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 $(R_1 \texttt{===>} R_2)\ ((abs_1 \texttt{-->} rep_2)\ a)\ ((abs_1 \texttt{-->} rep_2)\ a)$.
Proof: By the definition of `===>`, this is
$\forall x, y.\ R_1\ x\ y\ \Rightarrow R_2\ ((abs_1 \texttt{-->} rep_2)\ a\ x)\ ((abs_1 \texttt{-->} rep_2)\ a\ y)$. Assume $R_1\ x\ y$, and show $R_2\ ((abs_1 \texttt{-->} rep_2)\ a\ x)\ ((abs_1 \texttt{-->} rep_2)\ a\ y)$. By the definition of `-->`, 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 $(R_1 \texttt{===>} R_2)\ r\ s \Leftrightarrow$
$(R_1 \texttt{===>} R_2)\ r\ r \wedge (R_1 \texttt{===>} R_2)\ s\ s \wedge ((rep_1 \texttt{-->} abs_2)\ r = (rep_1 \texttt{-->} abs_2)\ s)$.
The last conjunct on the right side is equality between functions, so by extension this is $(R_1 \texttt{===>} R_2)\ r\ s \Leftrightarrow (R_1 \texttt{===>} R_2)\ r\ r \wedge (R_1 \texttt{===>} R_2)\ s\ s\ \wedge$
$(\forall x.\ (rep_1 \texttt{-->} abs_2)\ r\ x = (rep_1 \texttt{-->} abs_2)\ s\ x)$.
By the definitions of `===>` and `-->`, this is (1) $\Leftrightarrow$ (2) $\wedge$ (3) $\wedge$ (4), where

$$
\begin{array}{ll}
(1) & (\forall x\ y.\ R_1\ x\ y\ \Rightarrow R_2\ (r\ x)\ (s\ y)) \\
(2) & (\forall x\ y.\ R_1\ x\ y\ \Rightarrow R_2\ (r\ x)\ (r\ y)) \\
(3) & (\forall x\ y.\ R_1\ x\ y\ \Rightarrow R_2\ (s\ x)\ (s\ y)) \\
(4) & (\forall x.\ (abs_2\ (r\ (rep_1\ x)) = abs_2\ (s\ (rep_1\ x))).
\end{array}
$$

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

Goal 1. ($\Rightarrow$) 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 $R\ (rep\ (abs\ z))\ z$. □

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 $\langle R_2, abs_2, rep_2\rangle$,

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

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. □

7 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" [13]. 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 ε-operator [12, §4.4], also called the indefinite description operator. Paulson's lucid recent work [17] exhibits an approach to quotients which avoids the use of Hilbert's ε-operator, by instead using the definite description operator ι [14, §5.10]. These two operators may be axiomatized as follows:

$$\begin{array}{rcl} \forall P\ x.\ P\ x \Rightarrow P(\varepsilon\ P) & \textit{or} & \forall P.\ (\exists x.\ P\ x) \Rightarrow P(\varepsilon\ P) \\ \forall P\ x.\ P\ x \Rightarrow (\forall y.\ P\ y \Rightarrow x = y) \Rightarrow P(\iota\ P) & \textit{or} & \forall P.\ (\exists! x.\ P\ x) \Rightarrow P(\iota\ P) \end{array}$$

The ι-operator yields the single element of a singleton set, $\iota\{z\} = z$, but its result on non-singleton sets is indeterminate. By contrast, the ε-operator chooses some indeterminate element of any non-empty set, even if nondenumerable. The ι-operator is weaker than the ε-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 ι, not ε. Inspired by Paulson, we investigate this by forming an alternative design, eliminating the representation functions.

Definition 24 (Constructive quotient, replacing Definition 5). *A relation R with abstraction function abs (between the representation type τ and the abstraction type ξ) is a quotient (notated as $\langle R, abs\rangle$) if and only if*

$$\begin{array}{ll} (1) & \forall a : \xi.\ \exists r : \tau.\ R\ r\ r \wedge (abs\ r = a) \\ (2) & \forall r\ s : \tau.\ R\ r\ s \Leftrightarrow R\ r\ r \wedge R\ s\ s \wedge (abs\ r = abs\ s) \end{array}$$

Property 1 states that for every abstract element a of ξ there exists a representative in τ which respects R and whose abstraction is a.

Property 2 states that two elements of τ 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 (partial) equivalence relations may now be constructed by a modified version of §5, where the representation function *rep* is omitted from Definition 11, so there is no use of the Hilbert ε-operator. Property 1 follows from Lemma 8. The identity quotient is $\langle \texttt{\$=}, \texttt{I}\rangle$. From Definition 24 also follow analogs of the quotient extension theorems, e.g.,

$$\forall R\ abs.\ \langle R,\ abs\rangle \Rightarrow \langle \texttt{LIST\_REL}\ R,\ \texttt{MAP}\ abs\rangle$$

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

$$\begin{aligned} (abs \Downarrow R)\ a\ r &\stackrel{\text{def}}{=}\ R\ r\ r\ \wedge\ abs\ r = a \\ (reps\ \texttt{+->}\ abs)\ f\ x &\stackrel{\text{def}}{=}\ \iota\ (\texttt{IMAGE}\ abs\ (\texttt{IMAGE}\ f\ (reps\ x))) \end{aligned}$$

Note that for the identity quotient, $(\texttt{I} \Downarrow \texttt{\$=}) = \texttt{\$=}$.

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

Theorem 25 (Function quotient extension).

$$\langle R_1,\ abs_1\rangle \Rightarrow \langle R_2,\ abs_2\rangle \Rightarrow \langle R_1\ \texttt{===>}\ R_2,\ (abs_1 \Downarrow R_1)\ \texttt{+->}\ abs_2\rangle$$

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 ε-operator, as seen by the following development.

Theorem 26 (Partial abstraction quotients). *If f is any function from type α to β, and Q is any predicate on α, 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 ($\langle R, f\rangle$).*

Proof: Follows easily from substituting R in Definition 24 and simplifying. □

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

Proof: Assuming the function quotient extension theorem 25, we apply it to two quotient theorems; first, the identity quotient $\langle \texttt{\$=}, \texttt{I} \rangle$ for type β, and second, the partial abstraction quotient $\langle R, f \rangle$ from Theorem 26. This yields the quotient $\langle \texttt{\$= ===>}\ R,\ \texttt{\$= +->}\ f \rangle$, since $(\texttt{I} \Downarrow \texttt{\$=}) = \texttt{\$=}$. By Property 1 of Definition 24, $\forall a.\ \exists r.\ (\texttt{\$= ===>}\ R)\ r\ r \wedge ((\texttt{\$= +->}\ f) r = a)$. Specializing $a = \texttt{I} : \beta \to \beta$, and renaming r as g, we obtain $\exists g.\ (\texttt{\$= ===>}\ R)\ g\ g \wedge (\texttt{\$= +->}\ f) g = \texttt{I})$. By the definition of $\texttt{===>}$, $(\texttt{\$= ===>}\ R) g\ g$ 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 $(\texttt{\$= +->}\ f) g = \texttt{I}$, which by the definition of $\texttt{+->}$ is $(\lambda x.\ \iota\ (\texttt{IMAGE}\ f\ (\texttt{IMAGE}\ g\ (\texttt{\$=}\ x)))) = \texttt{I}$. But $\texttt{\$=}\ x$ is the singleton $\{x\}$, so since $\texttt{IMAGE}\ h\ \{z\} = \{h\ z\}$, $\iota\{z\} = z$, and $(\lambda x.\ f\ (g\ x)) = f \circ g$, this simplifies to $f \circ g = \texttt{I}$, and the conclusion follows. □

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

Proof: In Theorem 27, let $Q = (\lambda (P{:}\alpha \to \texttt{bool},\ a{:}\alpha).\ (\exists x.\ P\ x) \Rightarrow P\ a)$ and $f = \texttt{FST}$. Then its antecedent is $\forall P'.\ \exists (P, a).\ ((\exists x. P\ x) \Rightarrow P\ a) \wedge (\texttt{FST}(P, a) = P')$. 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 α. In either case the antecedent is true. Therefore by Theorem 27 there exists a function g such that $\texttt{FST} \circ g = \texttt{I}$ and $\forall P.\ Q\ (g\ P)$, which is $\forall P.\ (\exists x.\ (\texttt{FST}\ (g\ P))\ x) \Rightarrow (\texttt{FST}\ (g\ P))\ (\texttt{SND}\ (g\ P))$. The operator c is taken as $\texttt{SND} \circ g$, and since $\texttt{FST}\ (g\ P) = P$, the Hilbert choice axiom follows. □

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 quotient extension 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.

8 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 3), or 2) a theorem that R is a nonempty partial equivalence relation, of the form

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

or using the abbreviated forms $\vdash$ `EQUIV` R or $\vdash$ `PARTIAL_EQUIV` R, respectively.

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 §10.

tyop_equivs is a list of equivalence extension theorems for type operators (see §3.1). These are used for bringing into regular form theorems on new type operators, so that they can be lifted (see sections 12 and 13).

tyop_quotients is a list of quotient extension theorems for type operators (see §6.2). 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 for identity quotients, e.g., for pairs, ⊢ (`$= ### $=`) = `$=` and ⊢ (`I ## I`) = `I`, or for lists, ⊢ `LIST_REL $=` = `$=` and ⊢ `MAP I` = `I`.

The rest of the arguments refer to the general process of lifting theorems over the quotients being defined, as described in section 11.

respects is a list of theorems about the respectfulness of the constants being lifted. These theorems are described in section 11.1.

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 11.2.

poly_respects is a list of theorems showing the respectfulness of the polymorphic constants mentioned in *poly_preserves*. These are described in §11.3.

old_thms is a list of theorems concerning the lower, representative types and contants, which are to be automatically lifted and proved at the higher, more abstract quotient level. These theorems are described in section 11.4.

`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.

In addition to these, two related functions, `define_quotient_full_types` and `define_quotient_full_types_rule`, are provided that automatically include the standard pre-proven quotient, equivalence, and simplification theorems relating to the list, pair, sum, option, and function type operators, along with all the pre-proven polymorphic respectfulness and preservation theorems for many standard polymorphic operators of those theories in HOL. Other type operators and/or polymorphic operators may be supported by including their theorems in the appropriate input fields, which are named the same as before.

The function `define_quotient_types_full_rule` can be defined in terms of `define_quotient_types_rule` as

```
fun define_quotient_types_full_rule
          {types, defs, tyop_equivs, tyop_quotients, tyop_simps,
           respects, poly_preserves, poly_respects} =
  let
      val tyop_equivs = tyop_equivs @
              [LIST_EQUIV, PAIR_EQUIV, SUM_EQUIV, OPTION_EQUIV]
```

```
    val tyop_quotients = tyop_quotients @
            [LIST_QUOTIENT, PAIR_QUOTIENT,
             SUM_QUOTIENT, OPTION_QUOTIENT, FUN_QUOTIENT]
    val tyop_simps = tyop_simps @
            [LIST_MAP_I, LIST_REL_EQ, PAIR_MAP_I, PAIR_REL_EQ,
             SUM_MAP_I, SUM_REL_EQ, OPTION_MAP_I, OPTION_REL_EQ,
             FUN_MAP_I, FUN_REL_EQ]
    val poly_preserves = poly_preserves @
            [CONS_PRS, NIL_PRS, MAP_PRS, LENGTH_PRS, APPEND_PRS,
             FLAT_PRS, REVERSE_PRS, FILTER_PRS, NULL_PRS,
             SOME_EL_PRS, ALL_EL_PRS, FOLDL_PRS, FOLDR_PRS,
             FST_PRS, SND_PRS, COMMA_PRS, CURRY_PRS,
             UNCURRY_PRS, PAIR_MAP_PRS,
             INL_PRS, INR_PRS, ISL_PRS, ISR_PRS, SUM_MAP_PRS,
             NONE_PRS, SOME_PRS, IS_SOME_PRS, IS_NONE_PRS,
             OPTION_MAP_PRS,
             FORALL_PRS, EXISTS_PRS,
             EXISTS_UNIQUE_PRS, ABSTRACT_PRS,
             COND_PRS, LET_PRS,
             I_PRS, K_PRS, o_PRS, C_PRS, W_PRS]
    val poly_respects  = poly_respects @
            [CONS_RSP, NIL_RSP, MAP_RSP, LENGTH_RSP, APPEND_RSP,
             FLAT_RSP, REVERSE_RSP, FILTER_RSP, NULL_RSP,
             SOME_EL_RSP, ALL_EL_RSP, FOLDL_RSP, FOLDR_RSP,
             FST_RSP, SND_RSP, COMMA_RSP, CURRY_RSP,
             UNCURRY_RSP, PAIR_MAP_RSP,
             INL_RSP, INR_RSP, ISL_RSP, ISR_RSP, SUM_MAP_RSP,
             NONE_RSP, SOME_RSP, IS_SOME_RSP, IS_NONE_RSP,
             OPTION_MAP_RSP,
             RES_FORALL_RSP, RES_EXISTS_RSP,
             RES_EXISTS_EQUIV_RSP, RES_ABSTRACT_RSP,
             COND_RSP, LET_RSP,
             I_RSP, K_RSP, o_RSP, C_RSP, W_RSP]
in
  define_quotient_types_rule
        {types=types,
         defs=defs,
         tyop_equivs=tyop_equivs,
         tyop_quotients=tyop_quotients,
         tyop_simps=tyop_simps,
         respects=respects,
         poly_preserves=poly_preserves,
         poly_respects=poly_respects}
end;
```

Furthermore, two more related functions, `define_quotient_types_std` and `define_quotient_types_std_rule`, are provided. These are the same as the "`full`" functions above, but without the input record fields for `tyop_equivs`, `tyop_quotients`, `tyop_simps`, `poly_preserves`, or `poly_respects`. The "`std`" functions may be the easiest to use, providing much of the power of higher order quotients without the need for the user to worry about choosing theorems to include. For many applications the "`std`" functions will be all that is needed.

Similar to the above, the `define_quotient_types_std` function is defined in terms of `define_quotient_types_full` as

```
fun define_quotient_types_std {types, defs, respects, old_thms} =
    define_quotient_types_full
          {types=types, defs=defs,
           tyop_equivs=[], tyop_quotients=[],
           tyop_simps=[],
           respects=respects,
           poly_preserves=[], poly_respects=[],
           old_thms=old_thms};
```

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};
```

9 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 5, creating the quotient type as a new type in the HOL logic. It also defines the mapping functions between the types, forming a quotient as described in theorem 16.

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 τ, and prove that it is either 1) an equivalence relation on the original type τ, as described in section 3, as a theorem of the form:

$$\texttt{R\_EQUIV} \ \ \vdash (\forall x\ y.\ R\ x\ y \ \Leftrightarrow\ (R\ x = R\ y))$$

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

$$\texttt{R\_EQUIV} \ \ \vdash (\exists x.\ R\ x\ x)\ \wedge\ (\forall x\ y.\ R\ x\ y \ \Leftrightarrow\ R\ x\ x \wedge R\ y\ y \wedge (R\ x = R\ y)).$$

Equivalently, `R_EQUIV` may be of the form $\vdash$ `EQUIV` R or $\vdash$ `PARTIAL_EQUIV` R, respectively. These abbreviations are defined in the theorems `EQUIV_def` and `PARTIAL_EQUIV_def`. Evaluating `define_quotient_types` with the argument `types` containing the record {`name="`*tyname*`", equiv=R_EQUIV`} automatically declares a new type *tyname* in the HOL logic as the quotient type τ/R, which we will refer to from here on as ξ, and declares two new constants $abs : \tau \to \xi$ with name "*tyname*`_ABS`" and $rep : \xi \to \tau$ with name "*tyname*`_REP`", such that the relation R with *abs* and *rep* is a quotient, as described in Definition 5 of section 4. The `define_quotient_types` tool proves the quotient theorem

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

according to the development of Section 5, and stores it in the current theory under the automatically-generated name *tyname*`_QUOTIENT`.

10 Lifting Definitions of Constants

The previous section (§9) 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 11.2, but without antecedents. After the quotient operation, the definition theorem will not usually 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
```

11 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 type operators and constants 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 of `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.

11.1 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

$$\begin{array}{l} \vdash \forall (\mathtt{x1}{:}\gamma_1)\ (\mathtt{y1}{:}\gamma_1)\ \ldots\ (\mathtt{xn}{:}\gamma_n)\ (\mathtt{yn}{:}\gamma_n). \\ \qquad R_1\ \mathtt{x1}\ \mathtt{y1}\ \wedge\ \ldots\ \wedge\ R_n\ \mathtt{xn}\ \mathtt{yn}\ \Rightarrow \\ \qquad R_c\ (\mathtt{C}\ \mathtt{x1}\ \ldots\ \mathtt{xn})\ (\mathtt{C}\ \mathtt{y1}\ \ldots\ \mathtt{yn}) \end{array}$$

where the constant `C` has type γ_1 `->` ... `->` γ_n `->` γ_c ($n \geq 0$), and each relation R_i has type γ_i `->` γ_i `-> bool` for all i. Depending on the types involved, the partial equivalence relations R_1, ..., R_n, R_c may be simple equality, equivalence relations, aggregate partial equivalence relations, or higher-order partial equivalence relations (see section 6.1), 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 γ_i is not a type being lifted), the above general form may be simplified, in the following ways. The two variables `xi` and `yi` may be combined into one, say (`zi`:γ_i), e.g. in the list of universally quantified variables. The antecedent conjunct R_i `xi yi`, which here is `xi = yi`, may be omitted. In the conclusion R_c `(C x1 ... xn) (C y1 ... yn)`, the same variable `zi` may be used in place of both `xi` in the first operand of R_c and `yi` 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. Also, if all the antecedents thus simplify, or if $n = 0$ with no antecedents, then the implication simplifies to just the consequent, R_c `(C z1 ... zn) (C z1 ... zn)`.

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

$$\vdash \forall \mathtt{x1}\ \mathtt{x2}.\ (\mathtt{x1} = \mathtt{x2}) \Rightarrow \mathtt{ALPHA}\ (\mathtt{Var1}\ \mathtt{x1})\ (\mathtt{Var1}\ \mathtt{x2})$$

or possibly the simplified (but nonstandard) form

```
⊢ ∀x. ALPHA (Var1 x) (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

```
⊢ ∀t1 t2 u1 u2.
    ALPHA t1 t2 ∧ ALPHA u1 u2 ⇒
    ALPHA (App1 t1 u1) (App1 t2 u2)
```

`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

```
⊢ ∀t1 t2. ALPHA t1 t2 ⇒ (HEIGHT1 t1 = 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

```
⊢ ∀t1 t2. ALPHA t1 t2 ⇒ (FV1 t1 = 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

```
⊢ ∀t1 t2 s1 s2.
    ALPHA t1 t2 ∧ LIST_REL ($= ### ALPHA) s1 s2 ⇒
    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 its values.

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 6.1.

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.

11.2 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 preservation theorems have the following general form.

$\vdash$ ∀R1 (abs1:α_1->β_1) rep1. QUOTIENT R1 abs1 rep1 $\Rightarrow$
...
∀Rn (absn:α_n->β_n) repn. QUOTIENT Rn absn repn $\Rightarrow$
∀(x1:δ_1) ... (xk:δ_k).
C' x1 ... xk = abs_c (C (rep_1 x1) ... (rep_k xk))

where

1. the constant `C` has a polymorphic type with $n > 0$ type variables, $\alpha_1, ..., \alpha_n$,
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 γ_1 `-> ... ->` γ_k `->` γ_c $(k \geq 0)$
   (`C` is a curried function of k arguments, with a return value of type γ_c),
   with all type variables in the γ_1, ..., γ_k, γ_c types contained within α_1, ..., α_n,
4. the type of `C'` is of the form δ_1 `-> ... ->` δ_k `->` δ_c $(k \geq 0)$
   (`C'` is a curried function of k arguments, with a return value of type δ_c),
5. δ_i is the lifted version of γ_i for all $i = 1, ..., k, c$
   ($\delta_i = \gamma_i[\alpha_j \mapsto \beta_j$ for all $j = 1, ..., n]$),
   with all type variables in the δ_1, ..., δ_k, δ_c types contained within β_1, ..., β_n, and
6. abs_i:γ_i `->` δ_i and rep_i:δ_i `->` γ_i are the (possibly identity, aggregate, or higher-order) abstraction and representation functions for the type γ_i, for all $i = 1, ..., k, c$. 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

```
⊢ ∀R (abs:'a -> 'b) rep. QUOTIENT R abs rep ⇒
        ([] = MAP abs [])
```

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

```
⊢ ∀R (abs:'a -> 'b) rep. QUOTIENT R abs rep ⇒
     ∀l. LENGTH l = LENGTH (MAP rep l)
```

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

```
⊢ ∀R (abs:'a -> 'b) rep. QUOTIENT R abs rep ⇒
    ∀t h. h::t = MAP abs (rep h::MAP rep t)
```

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

```
⊢ ∀R1 (abs1:'a -> 'c) rep1. QUOTIENT R1 abs1 rep1 ⇒
   ∀R2 (abs2:'b -> 'd) rep2. QUOTIENT R2 abs2 rep2 ⇒
     ∀p:'c # 'd. FST p = abs1 (FST ((rep1 ## rep2) p))
```

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 composition 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 ⇒
     ∀f g. f o g =
       (rep1 --> abs3) ((abs2 --> rep3) f o (abs1 --> rep2) g)
```

The operator `o` has type `('b -> 'c) -> ('a -> 'b) -> ('a -> '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 -> 'c`, for which the representation function is `abs2 --> rep3`. The second argument type is `'a -> 'b`, for which the representation function is `abs1 --> rep2`. The result type is `'a -> 'c`, for which the abstraction function is `rep1 --> 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 9, 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 is available in the theories of the quotient library (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 the theory scripts of the quotient library.

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 `$o` on the left hand side of the equality is `('e -> 'f) -> ('d -> 'e) -> ('d -> 'f)`, while the type of the `$o` on the right hand side is `('b -> 'c) -> ('a -> 'b) -> ('a -> '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.

TABLE 1.
Preservation and Respectfulness Theorems for Polymorphic Operators

Lifted Operators	Preservation Theorems	Respectfulness Theorems
∀_::_._ → ∀_._	FORALL_PRS	RES_FORALL_RSP
∃_::_._ → ∃_._	EXISTS_PRS	RES_EXISTS_RSP
∃!!_::_._ → ∃!_._	EXISTS_UNIQUE_PRS	RES_EXISTS_EQUIV_RSP
λ_::_._ → λ_._	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	I_PRS	I_RSP
K	K_PRS	K_RSP
o	o_PRS	o_RSP
C	C_PRS	C_RSP
W	W_PRS	W_RSP

An arrow (*lower* → *higher*) indicates that in preservation theorems, the lower operator is different from the higher, else it is the same. Respectfulness theorems concern *lower*.

11.3 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 (§11.1), 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 (§11.2).

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:γ1) (y1:γ1) ... (xk:γk) (yk:γk).
         R1 x1 y1  ∧  ...  ∧  Rk xk yk ⇒
         Rc (C x1 ... xk) (C y1 ... yk)
```

where

1. the constant C has a polymorphic type with n type variables, $\alpha_1, ..., \alpha_n$,
2. $\beta_1, ..., \beta_n$ are n other type variables,
3. the type of C is of the form γ_1 -> ... -> γ_k -> γ_c (C is a curried function of k arguments, with a return value of type γ_c), with all type variables in the γ_1, ..., γ_k, γ_c types contained within α_1, ..., α_n, and
4. R_i:γ_i -> γ_i -> bool is the (possibly equality, aggregate, or higher-order) partial equivalence relation for the type γ_i, for all $i = 1, ..., k, c$. This is an expression, not simply an Ri 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

```
⊢ ∀R (abs:'a -> 'b) rep. QUOTIENT R abs rep ⇒
      LIST_REL R [] []
```

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 partial equivalence relation is LIST_REL R.

The respectfulness theorem for LENGTH is

```
⊢ ∀R (abs:'a -> 'b) rep. QUOTIENT R abs rep ⇒
      ∀l1 l2. LIST_REL R l1 l2 ⇒
              (LENGTH l1 = LENGTH l2)
```

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 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

```
⊢ ∀R (abs:'a -> 'b) rep. QUOTIENT R abs rep ⇒
    ∀t1 t2 h1 h2.
      R h1 h2 ∧ LIST_REL R t1 t2 ⇒
      LIST_REL R (h1::t1) (h2::t2)
```

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 partial equivalence relation is `R`. The second argument type is `('a)list`, for which the partial equivalence relation is `LIST_REL R`. The result type is `('a)list`, for which the partial equivalence relation is `LIST_REL R`.

The respectfulness theorem for `FST` is

```
⊢ ∀R1 (abs1:'a -> 'c) rep1. QUOTIENT R1 abs1 rep1 ⇒
  ∀R2 (abs2:'b -> 'd) rep2. QUOTIENT R2 abs2 rep2 ⇒
    ∀p1 p2. (R1 ### R2) p1 p2 ⇒
            R1 (FST p1) (FST p2)
```

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 partial equivalence relation is `R1 ### R2`. The result type is `'a`, for which the partial equivalence relation is `R1`.

The respectfulness theorem for the composition 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 `('b -> 'c) -> ('a -> 'b) -> ('a -> '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 -> 'c`, for which the partial equivalence relation is `R2 ===> R3`. The second argument type is `'a -> 'b`, for which the partial equivalence relation is `R1 ===> R2`. The result type is `'a -> '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 is available in the quotient library theories (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 the theory scripts of the quotient library.

11.4 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 11.1) and `poly_respects` (see section 11.3). 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 11.2). Each theorem must not have any free variables, and it also must not have any hypotheses, only a conclusion.

12 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:

```
⊢ ∀P.
     (∀v. P (Var1 v)) ∧
     (∀t t0. P t ∧ P t0 ⇒ P (App1 t t0)) ∧
     (∀t. P t ⇒ ∀v. P (Lam1 v t)) ⇒
     (∀t. P t)
```

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

```
∀t1 t2. ALPHA t1 t2 ⇒ (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

```
(∀x.   P1 (Var1 x)   = F) ∧
(∀t u. P1 (App1 t u) = P1 t ∨ P1 u) ∧
(∀x u. P1 (Lam1 x u) = P1 u ∨ (x = "a"))
```

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

```
⊢ ∀P :: respects (ALPHA ===> $=).
    (∀v. P (Var1 v)) ∧
    (∀t t0 :: respects ALPHA. P t ∧ P t0 ⇒ P (App1 t t0)) ∧
    (∀t :: respects ALPHA. P t ⇒ ∀v. P (Lam1 v t)) ⇒
    (∀t :: respects ALPHA. P t)
```

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

```
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

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

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 existance 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 =
   (∃x::respects R. P x) ∧
   (∀x y::respects R. P x ∧ P y ⇒ 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 (`λ`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 ∃! over types being lifted should be replaced by instances of ∃!! (`RES_EXISTS_EQUIV`) along with the appropriate partial equivalence relation. Instances of quantifying by ∃! over types not being lifted need not be modified at all. Note that where the restricted quantifier versions of ∀ and ∃ use restrictions of the form `respects R`, the restricted quantifier version of ∃!! 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]:

```
⊢ ∀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)])))
```

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

```
⊢ ∀var app abs.
    ∃!!hom :: (ALPHA ===> $=).
       (∀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)])))
```

which will then lift.

13 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 ⊢ `a = b`, for some expressions `a` and `b` whose type is being lifted. In fact, if $\sim$ is an equivalence relation, this equality implies ⊢ `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 ⊢ `P` ⇒ `(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 existance quantification ∃! restricted over functions which are respectful, may imply the corresponding theorem using the restricted ∃!! operator. For example, in the lambda calculus example, we have proven at the lower level

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

The tool will first automatically prove the proper version:

```
⊢ ∀var app abs.
    ∃!!hom :: ALPHA ===> $=.
      (∀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 one 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 3 and 4. If the type of any constant in any theorem involves type operators, like lists or functions, then the associated quotient extension theorems must be provided in the `tyop_quotients` argument. Likewise, the equivalence extension 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. This helps to retry lifting a troublesome theorem in isolation until it successfully lifts.

14 Lifting Sets

The facilities provided so far for higher order quotients are flexible and extensible for the addition of support for new type operators, with their own associated quotient extension theorems, simplification theorems, perhaps equivalence extension theorems, and respectfulness and preservation theorems for the constants associated with the type operator.

One commonly used type is sets, which are implemented in the Higher Order Logic theorem prover through the library `pred_set`. No actual new type is used here, but sets are represented by the function type `'a -> bool`, where `'a` is the underlying type of the elements of the set. In fact, there is a close identity between a set and its characteristic function. Nevertheless, it is helpful and intuitive at times to think of sets as sets, not functions, and many normal set operators, such as `INSERT`, `SUBSET`, and `UNION`, are provided.

For this type of sets, if the partial equivalence relation on the underlying type is R, then the extension of R to the set type is R `===>` (`$=:bool->bool->bool`). Essentially, sets behave as a specialization of the function quotient extention theorem, where the domain is the base type of R and the range is `bool`, as in

`SET_QUOTIENT:`
$\vdash\ \forall R\ abs\ rep.\ \langle R, abs, rep\rangle \Rightarrow \langle R\ \texttt{===>}\ \texttt{\$=},\quad rep\ \texttt{-->}\ \texttt{I},\quad abs\ \texttt{-->}\ \texttt{I}\rangle$

The associated abstraction function is *rep* `-->` `I`, and the associated representation function is *abs* `-->` `I`. Note that the use of *abs* vs. *rep* is counterintuitive.

When lifting theorems that contain instances of polymorphic set operators applied to values of types being lifted, some of the normal set operators are preserved across the quotient operation, but several are not. For these operators, we have provided new, similar operators which take into account the partial equivalence relation(s) involved, and thus do successfully lift to their normal set operator counterpart.

For example, for the set operator `DIFF`, no problems arise, and we can prove its respectfulness and preservation theorems of the normal form.

```
DIFF_PRS:
    ⊢ ∀R (abs:'a -> 'b) rep. QUOTIENT R abs rep ⇒
         ∀s t. s DIFF t =
                (rep --> I) (((abs --> I) s) DIFF ((abs --> I) t))

DIFF_RSP:
    ⊢ ∀R (abs:'a -> 'b) rep. QUOTIENT R abs rep ⇒
         ∀s1 s2 t1 t2.
                (R ===> $=) s1 s2 ∧ (R ===> $=) t1 t2 ⇒
                (R ===> $=) (s1 DIFF t1) (s2 DIFF t2)
```

But the following operators are *not* preserved, and have the indicated associated "regular" versions:

Normal	Regular
INSERT	INSERTR
DELETE	DELETER
SUBSET	SUBSETR
PSUBSET	PSUBSETR
DISJOINT	DISJOINTR
FINITE	FINITER
GSPEC	GSPECR
IMAGE	IMAGER

Even if the original operator is infix, all of the new operators are prefix operators, to ease the addition of the new arguments.

Most of these regular operators take one new argument, which is the partial equivalence relation of the underlying type of the set. For GSPECR and IMAGER, there are two new arguments, which are the partial equivalence relations of the types used for the polymorphic type variables of the original GSPEC or IMAGE. To make this clearer, here are the preservation theorems for these two operators.

```
GSPEC_PRS:
  ⊢ ∀R1 (abs1:'a -> 'c) rep1. QUOTIENT R1 abs1 rep1 ⇒
    ∀R2 (abs2:'b -> 'd) rep2. QUOTIENT R2 abs2 rep2 ⇒
      ∀f. GSPEC f =
          (rep2 --> I) (GSPECR R1 R2 ((abs1 --> (rep2 ## I)) f))

IMAGE_PRS:
  ⊢ ∀R1 (abs1:'a -> 'c) rep1. QUOTIENT R1 abs1 rep1 ⇒
    ∀R2 (abs2:'b -> 'd) rep2. QUOTIENT R2 abs2 rep2 ⇒
      ∀f s. IMAGE f s =
          (rep2 --> I) (IMAGER R1 R2 ((abs1 --> rep2) f)
                                     ((abs1 --> I) s))
```

The polymorphic preservation and respectfulness theorems for all these operators are found in quotient_pred_setTheory, where e.g. the names of these theorems for the INSERT operator are INSERT_PRS and INSERTR_RSP, respectively.

To ease the process of formulating the appropriate regular version of a theorem that a user wishes to lift, the tools will examine a given theorem to see if it is regular, and if not, will construct a regular version which if proved by the user will (hopefully) lift. Here are some examples.

```
val LIFT_RULE =
    define_quotient_types_full_rule
    {types = [{name = "term", equiv = ALPHA_EQUIV}],
     defs = fnlist,
     tyop_equivs = [],
     tyop_quotients = [],
     tyop_simps = [],
     respects = respects,
     poly_preserves = [IN_PRS,EMPTY_PRS,UNIV_PRS,UNION_PRS,
                       INTER_PRS,SUBSET_PRS,PSUBSET_PRS,
                       DELETE_PRS,INSERT_PRS,DIFF_PRS,GSPEC_PRS,
                       DISJOINT_PRS,FINITE_PRS,IMAGE_PRS],
     poly_respects  = [IN_RSP,EMPTY_RSP,UNIV_RSP,UNION_RSP,
                       INTER_RSP,SUBSETR_RSP,PSUBSETR_RSP,
                       DELETER_RSP,INSERTR_RSP,DIFF_RSP,GSPECR_RSP,
                       DISJOINTR_RSP,FINITER_RSP,IMAGER_RSP],

- LIFT_RULE (INST_TYPE [alpha |-> ``:'a term1``] EXTENSION)
  handle e => Raise e;

Exception raised at quotient.REGULARIZE:
Could not lift the irregular theorem
⊢ ∀s t. (s = t) ⇔ ∀x. x IN s ⇔ x IN t
May try proving and then lifting
∀s t::respects (ALPHA ===> $=).
  (ALPHA ===> $=) s t ⇔ ∀x::respects ALPHA. x IN s ⇔ x IN t
! Uncaught exception:
! HOL_ERR

- LIFT_RULE (INST_TYPE [alpha |-> ``:'a term1``] SUBSET_TRANS)
  handle e => Raise e;

Exception raised at quotient.REGULARIZE:
Could not lift the irregular theorem
⊢ ∀s t u. s SUBSET t ∧ t SUBSET u ⇒ s SUBSET u
May try proving and then lifting
∀s t u::respects (ALPHA ===> $=).
  SUBSETR ALPHA s t ∧ SUBSETR ALPHA t u ⇒ SUBSETR ALPHA s u
! Uncaught exception:
! HOL_ERR
```

```
- LIFT_RULE
    (INST_TYPE [alpha |-> ``:'a term1``, beta |-> ``:'b term1``]
      INJECTIVE_IMAGE_FINITE);

Exception raised at quotient.REGULARIZE:
Could not lift the irregular theorem
⊢ ∀f. (∀x y. (f x = f y) ⇔ (x = y)) ⇒
      ∀s. FINITE (IMAGE f s) ⇔ FINITE s
May try proving and then lifting
∀f::respects (ALPHA ===> ALPHA).
  (∀x y::respects ALPHA. ALPHA (f x) (f y) ⇔ ALPHA x y) ⇒
   ∀s::respects (ALPHA ===> $=).
     FINITER ALPHA (IMAGER ALPHA ALPHA f s) ⇔ FINITER ALPHA s
! Uncaught exception:
! HOL_ERR
```

In each case the suggested regular version of the theorem can be copied from the error message, proven by hand, and then submitted to the tool for lifting.

Here are the names of the preservation and respectfulness theorems for polymorphic set operators provided in the quotient library.

TABLE 2.
Preservation and Respectfulness Theorems for Polymorphic Set Operators

Lifted Operators	Preservation Theorems	Respectfulness Theorems
IN	IN_PRS	IN_RSP
EMPTY	EMPTY_PRS	EMPTY_RSP
UNIV	UNIV_PRS	UNIV_RSP
INTER	INTER_PRS	INTER_RSP
UNION	UNION_PRS	UNION_RSP
DIFF	DIFF_PRS	DIFF_RSP
INSERTR → INSERT	INSERT_PRS	INSERTR_RSP
DELETER → DELETE	DELETE_PRS	DELETER_RSP
DISJOINTR → DISJOINT	DISJOINT_PRS	DISJOINTR_RSP
GSPECR → GSPEC	GSPEC_PRS	GSPECR_RSP
SUBSETR → SUBSET	SUBSET_PRS	SUBSETR_RSP
PSUBSETR → PSUBSET	PSUBSET_PRS	PSUBSETR_RSP
FINITER → FINITE	FINITE_PRS	FINITER_RSP
IMAGER → IMAGE	IMAGE_PRS	IMAGER_RSP

An arrow (*lower* → *higher*) indicates that in preservation theorems, the lower operator is different from the higher, else it is the same. Respectfulness theorems concern *lower*.

In addition, several related theorems are provided, including the definitions of the new operators and some of their interactions with other set operators.

```
IN_INSERTR:
    ⊢ ∀Rx s y. y IN INSERTR R x s ⇔ R y x ∨ y IN s
IN_DELETER:
    ⊢ ∀R s x y. y IN DELETER R s x ⇔ y IN s ∧ ~R x y
IN_GSPECR:
    ⊢ ∀R1 R2 f v.
        v IN GSPECR R1 R2 f ⇔
        ∃x::respects R1. (R2 ### $=) (v,T) (f x)
IN_IMAGER:
    ⊢ ∀R1 R2 y f s.
        y IN IMAGER R1 R2 f s ⇔
        ∃x::respects R1. R2 y (f x) ∧ x IN s
```

These facilities for set operators are presented to be helpful, but they are in development, and should be considered experimental and subject to change.

15 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) $m_1, \ldots, m_n \in M_1 \;\Rightarrow\; [l_1 = m_1,\, \ldots,\, l_n = m_n] \in O_1$ for all labels $l_1, \ldots, l_n$;
(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) $a \in O_1 \;\Rightarrow\; \varsigma(x)a \in M_1$ for all varibles x.

$[l_1 = m_1,\, \ldots,\, l_n = m_n]$ 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 $\varsigma(x)a$ denotes a method with one formal parameter, x, and a body a. ς is a binder, like λ 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, $\varsigma(x)x.l_1 = \varsigma(y)y.l_1$, $[l_1 = \varsigma(x)x] = [l_1 = \varsigma(y)y]$, 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 $O = O_1/\!\equiv^o_\alpha$ and $M = M_1/\!\equiv^m_\alpha$, where $\equiv^o_\alpha$ and $\equiv^m_\alpha$ are the respective alpha-equivalence relations.

16 The Pre-Sigma Calculus in HOL

HOL 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 existance, 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.

17 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},
              {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 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)) ∧
  (∀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 higher-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:

```
⊢ ∀P0 P1 P2 P3.
    (∀v. P0 (OVAR v)) ∧ (∀l. P2 l ⇒ P0 (OBJ l)) ∧
    (∀o'. P0 o' ⇒ ∀s. P0 (INVOKE o' s)) ∧
    (∀o' m. P0 o' ∧ P1 m ⇒ ∀s. P0 (UPDATE o' s m)) ∧
    (∀o'. P0 o' ⇒ ∀v. P1 (SIGMA 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)
```

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

```
⊢ ∀(P0::respects (ALPHA_obj ===> $=))
      (P1::respects (ALPHA_method ===> $=))
      (P2::respects (LIST_REL ($= ### ALPHA_method) ===> $=))
      (P3::respects (($= ### ALPHA_method) ===> $=)).
    (∀v. P0 (OVAR1 v)) ∧
    (∀l::respects (LIST_REL ($= ### ALPHA_method)).
       P2 l ⇒ P0 (OBJ1 l)) ∧
    (∀o'::respects ALPHA_obj. P0 o' ⇒ ∀s. P0 (INVOKE1 o' s)) ∧
    (∀(o'::respects ALPHA_obj) (m::respects ALPHA_method).
       P0 o' ∧ P1 m ⇒ ∀s. P0 (UPDATE1 o' s m)) ∧
    (∀o'::respects ALPHA_obj. P0 o' ⇒ ∀v. P1 (SIGMA1 v o')) ∧
    P2 [] ∧
    (∀(p::respects ($= ### ALPHA_method))
        (l::respects (LIST_REL ($= ### ALPHA_method))).
       P3 p ∧ P2 l ⇒ P2 (p::l)) ∧
    (∀m::respects ALPHA_method. P1 m ⇒ ∀s. P3 (s,m)) ⇒
    (∀o'::respects ALPHA_obj. P0 o') ∧
    (∀m::respects ALPHA_method. P1 m) ∧
    (∀l::respects (LIST_REL ($= ### ALPHA_method)). P2 l) ∧
    (∀p::respects ($= ### ALPHA_method). P3 p)
```

Finally, the most difficult theorem to lift is the function existance theorem, after the style proposed by Gordon and Melham [8]. This uses higher-order unique existance quantification, where the unique existance 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.

```
⊢ ∀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 existance quantification over (hom_o :obj1->'a, ...) lifts to unique existance quantification over (hom_o :obj->'a, ...). 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.

18 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.

Higher order quotients require the use of partial equivalence relations, as symmetric and transitive but not necessarily reflexive on all their domains.

The relationship between the lower type and the quotient type is characterized by the partial equivalence relation, the abstraction function, and the representation function. As a key contribution, three necessary properties have been identified for these to properly describe a quotient, which are preserved in the creation of both aggregate and higher order quotients. Most normal polymorphic operators both respect and are preserved across such quotients, including higher order quotients.

The Axiom of Choice was used in this design. We showed that an alternative design may be constructed without dependence on the Axiom of Choice, but that it may not be extended to higher order quotients while remaining constructive.

Prior to this work, only Harrison [9] went beyond support for modeling the quotient types to provide automation for the lifting of constant definitions and theorems from their original statements concerning the original types to the corresponding analogous statements concerning the new quotient types. This is important for the practical application of quotients to sizable problems like quotients on the syntax of complex, realistic programming or specification languages. These may be modelled as recursive types, where terms which are partially equivalent by being well-typed and alpha-equivalent are identified by taking quotients. This eases the traditional problem of the capture of bound variables [8].

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*, Albequerque, New Mexico, 1992, pp. 316-327.
4. Chicli, 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. Kalker, T.: at `www.ftp.cl.cam.ac.uk/ftp/hvg/info-hol-archive/00xx/0082`.
12. Leisenring, A. C.: *Mathematical Logic and Hilbert's ε-Symbol.* Gordon and Breach, 1969.
13. Moore, G. H.: *Zermelo's Axiom of Choice: It's Origins, Development, and Influence.* Springer-Verlag 1982.
14. Nipkow, T., Paulson, L.C., Wenzel, M.: *Isabelle/HOL*. Springer-Verlag 2002.
15. Owre, S., Shankar, N.: *Theory Interpretations in PVS*, Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001.
16. 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.
17. 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.
18. Robinson, E.: 'How Complete is PER?', in *Fourth Annual Symposium on Logic in Computer Science* (LICS), 1989, pp. 106-111.
19. 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.